Fundamentals of

Digital and Computer Design with VHDL

Richard S. Sandige
Michael L. Sandige

This page intentionally left blank

Fundamentals of Digital and Computer Design with VHDL

Richard S. Sandige
Professor Emeritus, California Polytechnic State University

Michael L. Sandige
Principal Engineer, WildTangent

The McGraw-Hill Companies

FUNDAMENTALS OF DIGITAL AND COMPUTER DESIGN WITH VHDL

This book is printed on recycled, acid-free paper containing 10% postconsumer waste.

1 2 3 4 5 6 7 8 9 0 QDB/QDB 1 0 9 8 7 6 5 4 3 2 1

ISBN 978-0-07-338069-8
MHID 0-07-338069-5

Vice President & Editor-in-Chief: *Marty Lange*
Vice President & Director of Specialized Publishing: *Janice M. Roerig-Blong*
Global Publisher: *Raghothaman Srinivasan*
Senior Sponsoring Editor: *Peter E. Massar*
Senior Marketing Manager: *Curtis Reynolds*
Developmental Editor: *Lorraine K. Buczek*
Lead Project Manager: *Jane Mohr*
Design Coordinator: *Brenda A. Rowles*
Cover Designer: *Studio Montage, St. Louis, Missouri*
Cover Image: © *R.A. - Fotolia.com*
Buyer: *Nicole Baumgartner*
Media Project Manager: *Balaji Sundararaman*
Compositor: *Lachina*
Typeface: *10/12 Times LT Std*
Printer: *Quad Graphics*

All credits appearing on pages are considered to be an extension of the copyright page.

Library of Congress Cataloging-in-Publication Data

Sandige, Richard S.
Fundamentals of digital and computer design with VHDL / Richard S. Sandige, Michael L. Sandige.
p. cm.
Includes index.
ISBN-13: 978-0-07-338069-8 (alk. paper)
ISBN-10: 0-07-338069-5 (alk. paper)
1. Digital electronics. 2. Electronic digital computers—Design and construction—Data processing. 3. VHDL (Computer hardware description language) I. Sandige, Michael L. II. Title.
TK7868.D5S253 2012
621.39'2—dc23

2011033683

www.mhhe.com

To
my loving wife
Edie

Brief Contents

Contents

This page intentionally left blank

Preface

This book is intended for an introductory digital design course for students at the freshman level; it is also intended for an introductory computer design course with assembly language programming for students at the sophomore level or above. The material in the book is suitable for students who study computer engineering, computer science, and electrical engineering. This book uses a spiral teaching approach by introducing a design problem and then, in the same or a later chapter, either (1) reemphasizing the same concepts when a different design problem is presented, or (2) working the same problem using a different technique. This is done to increase the likelihood of retention.

There is no prerequisite for the book; however, computer familiarity and/or a first programming course usually helps students to learn VHDL and assembly language programming.

KEY FEATURES OF THIS TEXT

- Generic VHDL code is taught and used throughout the book so that different companies' VHDL tools can be used if desired.
- Classical and modern VHDL designs are taught to provide a balance throughout the course.
- A Karnaugh Map Explorer program is provided as an interactive tool to teach students to use Karnaugh maps with 2, 3, and 4 variables.
- Students are taught how to design VBC1 (Very Basic Computer 1) and VBC1-E (an enhanced version of VBC1), which are 4-bit educational computers. Both computers can be downloaded into a field programmable gate array (FPGA) on a development board and programmed in assembly language.
- An editor/assembler/simulator program called EASY1 is provided to teach students how to program VBC1 and VBC1-E.
- A memory loader software program is provided to teach students how to design a loader for instruction memory.
- One or more experiments are provided for every chapter and keyed by number; many have a recommended pre-lab assignment, related to either writing assembly language programs or obtaining simulations for VHDL designs.
- Homework problems are keyed to every section of the book.

CHAPTER AND TOPICAL OVERVIEW

In the digital design section of this book, the following topics are covered in Chapters 1 through 9:

- VHDL (Very High Speed Integrated Circuit Hardware Description Language) is introduced in Chapter 1 for combinational logic circuits.
- Thirty-four experiments are provided to allow students to "learn by doing." Experiments 1A through 9B provide practice for students to learn how to design VHDL circuits for digital

designs in the laboratory. The experiment numbers are keyed to the chapter numbers. The experiments may be used for homework assignments or special projects.

- Students are introduced in Chapter 2 to a 7-segment display that uses a letter display system that they can design and build with an FPGA board via VHDL to display the high (H) or low (L) level of a slide switch. The Karnaugh Map Explorer program is also provided as an interactive tool to teach students to use Karnaugh maps with 2, 3, and 4 variables.
- A graphical design method is introduced for manually designing logic circuits in NAND/NAND and NOR/NOR form in Chapter 3 and then followed up with equivalent designs using VHDL. Decoders and multiplexers (MUXs) are introduced and manual methods are used to design circuits with both decoders and MUXs. These methods are followed with equivalent designs using VHDL. Function and logic hazards are presented and students are shown how to eliminate logic hazards with logic hazard cover terms.
- Combinational logic circuit design with VHDL is presented in Chapter 4 with complete coverage of dataflow, behavioral, and structural design styles. VHDL examples that include both scalar and vector inputs and outputs are provided.
- Set-reset (S-R) latches, D latchs, and D flip-flops are designed both classically and with VHDL in Chapter 5. In this chapter students learn how to design D edge-triggered flip-flops. Experiment 5A provides hands-on learning experience in the gate-level design of a D latch and a positive edge-triggered D flip-flop with a CLR (clear) input. Experiment 5B provides a similar hands-on experience of an 8-bit register and a positive edge-triggered D flip-flop with a PRE (preset) input.
- Finite state machine design is divided into simple and complex state machines. Simple state machines are basic counters without an external input to change the counting sequence. Complex state machines have an external input(s) to change the counting sequence. Simple state machines are presented in Chapter 6. An algorithmic equation method for simple state machines is presented to show students how to design simple state machines manually. This method works for practically any size state machine but it is tedious because students must learn how to write *D* excitation equations. Students learn how to use the arithmetic method, which is especially useful when generating a slower clock frequency (or frequency divider). In addition, a present-state/next-state (PS/NS) tabular method is introduced that allows students to write the VHDL equations for basic counters using a process that eliminates the hassle of obtaining *D* excitation equations. A new counter design description called a *counting* or *state sequence diagram* is introduced alongside the traditional state diagram.
- In Chapter 7 various computer circuits are presented including three-state outputs, data bus sharing, adder and subtractor design, ripple carry adders, and carry look-ahead adders. Experiments 7A and 7B provide hands-on design experience of special combinational logic systems related to Chapter 7, that is, a simple (single-bit) error detection system and a 4-bit simple adder-subtractor system.
- Circuit implementation techniques presented in Chapter 8 show the implementation of programmable logic devices—that is, PROMs, PLAs, PALs, GALs and LUTs. This chapter also contains a brief introduction to the positive logic convention and direct polarity indication and how to convert between the two representations. A modular design technique is presented to show how to design MUX and DMUX trees. Designing and testing of a LUT design system is provided in Experiment 8.
- Complex state machine design with VHDL is presented in Chapter 9 using the two-process PS/ NS method. The first process, called the *synchronous process*, generates the flip-flops, and the second process, called the *combinational process*, decodes the next-state functions and any Moore and Mealy outputs that may be present in the design. State machine encoding styles that are used for complex programmable logic devices (CPLDs) and FPGAs are also presented in this chapter. Experiment 9A provides hands-on design experience in designing a one-hot up/down counter system using a flat design approach. Experiment 9B

provides hands-on design experience in designing a 10-state counter system using a hierarchal design approach. A synchronizer circuit is introduced that improves the reliability of complex state machine designs by reducing the possibility of metastability. To complete the discussion, two additional state machine design methods are presented—the two-assignment PS/NS method and the hybrid PS/NS method.

In the computer design section of this book, the following topics are covered in Chapters 10 through 17:

- After a brief introduction to Harvard and Princeton (von Neumann)-type computer architectures, students are introduced to a very basic Harvard-type computer called VBC1 (Very Basic Computer 1) in Chapter 10. Students learn the programmer's register model, the instruction set architecture, and the format for writing assembly language for VBC1. Students get their first introduction to writing assembly language programs for VBC1 in Experiment 10. Appendix D provides a tutorial for the editor/assembler/simulator 1 that we call EASY1.
- In Chapter 11, the assembly language form, transfer function form, and machine code form are presented for all the instructions for VBC1. Programming examples and techniques are presented for VBC1.
- The experiments accompanying Chapter 12 through Chapter 17 allow students to construct, design, and implement VBC1 via an FPGA over a period of six weeks if one experiment is performed per week. The number of experiments per week can be decided on by the instructor teaching the laboratory class. Recommended pre-lab assignments are provided.
- Chapter 12 presents the design of input/output (I/O) circuits for VBC1 and deals mainly with bus steering circuits, loadable D registers, driving light-emitting diodes (LEDs) and driving 7-segment displays.
- The design of instruction memory (for storing programs), a loading program counter (for loading instruction memory), and a debounced circuit (for single stepping through instruction memory) is presented in Chapter 13. An introduction to gated clock circuits and how to remove them is also emphasized in this chapter.
- Chapter 14 presents the design of a multiplexed display system for VBC1. To provide additional experience, students learn how to design a word display system in this chapter.
- The design of the instruction decoder for VBC1 is presented in Chapter 15 because each instruction must be decoded to automatically provide instruction execution.
- The design of the arithmetic logic unit (ALU) for VBC1 is covered in Chapter 16. In addition to an expanded ALU design, students also learn how to design the following circuits: shifter circuits, barrel shifter circuits, and shift register circuits via VHDL.
- Chapter 17 covers the final design for VBC1, which includes the design of the running program counter that allows VBC1 to run at a specified clock frequency. Experiment 17L is designed to provide the capability to automatically load programs into the instruction memory of VBC1 via a memory loader program. Up to this point the instruction memory was either loaded manually via slide switches or preloaded via initialization of the instruction memory in the VHDL code for VBC1.

In the computer design section of this book, the following topics are covered in Chapters 18 through 25:

- In Chapter 18, the assembly language form, transfer function form, and machine code form are presented for all the instructions for VBC1-E. This includes modified IN and OUT instructions (four ports each), additional instructions for a data memory (STORE and FETCH), additional arithmetic and logic instructions (SUB, NOT, AND, OR, and XNOR), additional shift and rotate instructions (SR1, SL0, SL1, RR, and RL), additional control instructions (JMP, JMPR, and HALT), additional software interrupt instructions (INT and

IRET), and a hardware interrupt capability. Two assembler directives (BIPROC and EQU) are also included in the EASY1-E assembler for VBC1-E. Experiment 18 provides students with hands-on experience with writing and simulating assembly language programs for both VBC1 and VBC1-E.

- The experiments accompanying Chapters 19 through Chapter 25 allow students to construct, design, and implement VBC1-E via an FPGA over a period of seven weeks (if one experiment is performed per week). The number of experiments per week can be decided on by the instructor teaching the laboratory class.
- In Chapter 19 students learn how to expand the I/O design of VBC1 to make VBC1-E. This includes redesigning the instruction decoder to handle the expanded I/O design for the IN and OUT instructions.
- Data memory did not exist in VBC1 so Chapter 20 covers the design of a simple data memory with four storage locations for VBC1-E.
- In Chapter 21, students learn how to enhance the design of the ALU for VBC1 to include new arithmetic and logic instructions, and new shift and rotate instructions. This includes redesigning the instruction decoder to handle the additional instructions. The design for the new control instructions JMP, JMPR, and HALT are presented, and the instruction decoder is redesigned to handle these new instructions.
- Chapter 22 is a very short chapter showing how to design a circuit to prevent program execution during manual loading. Students often find manual loading of VBC1-E to be distracting, which led to this chapter in the book.
- VBC1-E has additional storage locations and Chapter 23 covers the modification of the instruction memory to include the additional storage locations.
- VBC1-E has new software interrupt instructions and Chapter 24 covers the design of the necessary circuits to handle the instructions INT and IRET. The instruction decoder is also redesigned to handle these new instructions.
- Chapter 25 covers the final design for VBC1-E, which includes the design of the hardware interrupt capability. Experiment 25L is designed to provide the capability to automatically load programs into the instruction memory of VBC1-E via a memory loader program.

Information on CAD Tools and FPGA Boards

For the digital design portion of this book, circuits and systems are presented using both a classical methodology—that is, manual calculations—and VHDL designs. For the computer design portion of this book, VHDL is used to design VBC1 and VBC1-E. This approach encourages students to design their own digital systems and/or games once they learn how easy it is to design with VHDL. Students are not restricted to generating circuits on a small circuit board, where they must place the IC (integrated circuit) packages and wire them together. The FPGA (field programmable gate array) chips that are used on modern digital boards remove this arduous task. Xilinx® ISE® WebPACK is the primary CAD (computer-aided design) tool used in this book. The ISE WebPACK is available to students and instructors via the Xilinx web site at http://www.xilinx.com/support/download/index.htm.

A programmable logic FPGA chip can be reprogrammed over and over, which is an ideal way for students to learn by trial and error. Designs with VHDL code are simulated to verify that the VHDL code works. If there is an error in the simulation of their VHDL code, students can simply find and fix the error or errors and rerun the simulation. When the simulation is correct, the correct bit pattern for the VHDL code can be downloaded into the FPGA chip. Students can then observe their design working in hardware.

The Digilent® Company manufactures and distributes two very popular FPGA boards. Their web site is http://www.digilentinc.com. The boards are (1) the BASYS 2™ board, which contains a Spartan 3E FPGA manufactured by Xilinx, and (2) the NEXYS 2™ board, which also contains a Spartan 3E FPGA. The cost of the BASYS 2 board is about $49; the cost of the

NEXYS 2 board is about $99. If the book is used for a digital design course, either the BASYS 2 or NEXYS 2 board can be used. If the book is used for a computer design course, either the BASYS 2 or NEXYS 2 board can be used to design a stripped-down version of VBC1-E, for an added cost of about $55 for a few additional peripheral modules. The NEXYS 2 board has additional flexibility that allows an add-on board called the FX2 MIB (module interface board) to be used. This add-on board allows a full-blown version of VBC1-E to be designed with additional peripheral modules. The cost of the FX2 MIB is about $20 and the additional peripheral modules are about $40.

Information for Laboratory Experiments

Appendix A contains 34 laboratory experiments. To perform Experiments 1B through 25L requires the use of a BASYS 2 or NEXYS 2 board. These boards are quite popular and will be upgraded as newer FPGAs become available. The experiments may be performed on a different FPGA board from another vender, provided that the FPGA board has the required input/output capability, or the VHDL code is modified to match the input/output capability of the different FPGA board. The Altera® DE1 and DE2 boards can also be used to perform the experiments using Quartus-II software, because generic VHDL code is used throughout the book. The I/O for the DE1 and DE2 boards are slightly different than that of the BASYS 2 and NEXYS 2 boards, thus requiring changes to be made in the generic VHDL code to accommodate the differences in I/O capability.

Recommended pre-lab assignments are included so students can learn how to modify test bench code or write test bench code to simulate their VHDL designs. Appendix B provides students with help in how to modify test bench code.

ACKNOWLEDGMENTS

We would like to thank the many reviewers who made valuable comments during the development and writing phases of this book. We hope the final product meets with their approval.

Bharat Bhuva	Vanderbuilt University
Suresh Borkar	Illinois Institute of Technology
C. Hwa Chang	Tufts University
Katherine Compton	University of Wisconsin–Madison
Steve Crist	Western New England College
Nila Y. Desai	Sardar Vallabhbhai National Institution of Technology, India
Rahul Dubey	DA-IICT (Dhirubhai Ambani Institute of Information and Communication Technology), Gujarat India
Mark Faust	Portland State University
Maria Garazaran	University of Illinois–Urbana
Roger Haggard	Tennessee Technological University
Ronald Hayne	The Citadel
M. Nazrul Islam	Old Dominion University
Anura Jayasumana	Colorado State University
Brock LaMeres	Montana State University
Yong Li	University of Wisconsin–Platteville
Yufeng Lu	Bradley University
Aleksander Malinowski	Bradley University
Chad Mano	Utah State University
James C. Maxted	University of Iowa

David G. Meyer	Purdue University
Venkatesan Muthukumar	University of Nevada–Las Vegas
Shahin Nazarian	University of Southern California
Patricia Nava	University of Texas–El Paso
Haluk Ozemek	San Jose State University
James K. Peckol	University of Washington
Reginald Perry	Florida State University
Arvind Rajawat	Maulana Azad National Institute of Technology, Bhopal India
Jackie Rice	University of Lethbridge
Salam Salloum	California State Polytechnic University–Pomona
Martha Sloan	Michigan Technological University
James Stine	Oklahoma State University
Somanath Tripathy	Indian Institute of Technology, Patna India
Kenneth S. Vastola	Rensselaer Polytechnic University
Earl Wells	University of Alabama–Huntsville
Phillip Wilsey	University of Cincinnati

We would also like to provide a special acknowledgment to Scott Marshall, one of Richard's students, who developed and wrote Experiments 17L and 25L. Scott also wrote the memory loader program for VBC1-L and VBC1-EL. Scott did this for Digilent's BASYS 2 and NEXYS 2 boards and also for Altera's DE1 and DE2 boards. The memory loader program provides the means to automatically download the instruction bit patterns into the instruction memory of VBC1 and VBC1-E without using switches or initializing instruction memory at startup.

We would also like to thank the many dedicated and studious students at California State Polytechnic University in San Luis Obispo, in both electrical and computer engineering. Many students took an interest in our work, making suggestions and corrections as they completed the digital and computer design courses taught from our notes while the book was being developed. Without these caring students—who are too many to mention by name—it would not have been possible to write this book. Thank you.

Without the wonderful caring help of the following people at McGraw-Hill, this book would not have made it through the publication process: Raghu Srinivasan, global publisher; Peter Massar, senior sponsoring editor; Lorraine Buczek, developmental editor; and Jane Mohr, project manager. We really appreciate all your help in making this book become a reality.

We would like to thank the Digilent Company for supplying the BASYS 2 and NEXYS 2 boards, providing the technical assistance in using them, and providing the photos in Experiments 1A and 19. The people who helped us are Clint Cole, Gene Apperson, Jim O'Dell, Joe Harris, Joshua Pederson, Fiona Cole, Stephanie Roberts, Norman MacDonald, and Roy Bean.

We would like to thank the Altera® Corporation for supplying the DE1 and DE2 boards and for providing the technical assistance in using them. The people who helped us are Blair Fort and Ralene Marcoccia.

Richard would like to personally thank the Xilinx Company for allowing him to attend in-house courses on the ISE software and to work at Xilinx over the summer with their software and hardware engineers. The people who helped him are Rina Raman, Peter Alfke, and Paul Hartke. Richard would also like to thank Ken Chapman who reviewed the manuscript very early in the development of the book and made important comments. Ken is the designer of the Xilinx PicoBlaze 8-bit microcontroller. There were many more engineers who provided excellent advice and help while Richard worked at Xilinx, and sincere thanks goes out to all of them.

Last, but not least, we would like to thank the faculty in the electrical engineering department and CPE program at Cal Poly who had a part in making this book better. The people who helped us are Jim Harris, Michael Cirovic, Art MacCarley, Albert Liddicoat, Xiao-Hua (Helen) Yu, Tina Smilkstein, David Braun, Dennis Derickson, Fred DePiero, John Oliver, Wayne Pilk-

ington, Lynne Silvovsky, Bryan Mealy, John Saghri, and Hugh Smith. Some of these people taught the courses using our notes and others encouraged or discouraged us from doing certain things, which we believe made the book better.

Online Resources

A number of valuable resources are available to students and instructors at the following website: www.mhhe.com/sandige. Students may download the programs used in the text, such as the Karnaugh map explorer program, the EASY1 program, and the Memory Loader program. PowerPoint slides, solutions to the end-of-chapter problems, stimulus processes for pre-labs, and solutions to the laboratory experiments are available to instructors only.

No book is totally error free. If you find an undiscovered error, please e-mail your comments to richard@sandige.com. An up-to-date list of discovered errors will be available to all readers on the book's website.

Richard Sandige
Michael Sandige

About the Authors

Richard Sandige is a Professor Emeritus at California Polytechnic State University (Cal Poly) at San Luis Obispo, California. He taught from 1998 to 2010 at Cal Poly in the CPE (computer engineering) program. He received a PhD in electrical engineering from Texas A&M University in 1978, an MS in electrical engineering from West Virginia University in 1969, and a BS in electrical engineering from West Virginia University in 1963. He taught from 1989 to 1998 at the University of Wyoming in the Electrical Engineering Department. He worked at Hewlett-Packard (HP) Company from 1979 to 1989, where he was on the team that designed the first HP desktop computer. He taught at Texas A&M University from 1973 to 1979 in the Engineering Technology Department while working on his PhD. He taught at Southwest Virginia Community College from 1970 to 1973 in the Electronics Department. He worked at Conductron Missouri (Electronic Branch of McDonnell Douglas) from 1969 to 1970 on flight simulators. He taught at West Virginia Institute of Technology from 1967 to 1969 while working on his MS. He served in the United States Air Force from 1963 to 1966, working on the research and development of intercontinental ballistic missiles, after receiving his commission as Second Lieutenant in the Air Force Reserve Officer Training Corps at West Virginia University. While working at the University of Wyoming, he served as an assistant editor for the *IEEE Transactions on Education*. Dr. Sandige has more than 20 publications in professional journals and has published four textbooks and one lab manual.

Mike Sandige is a software architect and developer with a history of and interest in game and game technology development. He received a BS in electrical engineering and computer science in 1990 from the University of Colorado, Boulder. He started in the game industry as an independent contractor, working as the sole or lead programmer on several titles from 1987 through 1994. In 1994, he co-founded Cinematronics LLC, with the charter to make games for Microsoft Windows, an emerging platform for games. As vice president of product development, he worked on games, and as the company grew, he managed and directed the engineering staff. One of the more popular games he developed during this time was the "Space Cadet" pinball game that shipped with Windows 95. Cinematronics became a successful game development studio, leading to acquisition by Maxis (the makers of "The Sims") in 1996. In 1997, Mike Sandige joined Eclipse Entertainment, a small startup, as vice president of research and development. He worked on game engine architecture, rendering technology and helping make Eclipse Entertainment's Genesis3D game engine a leading product. He joined WildTangent when it acquired Eclipse Entertainment in 1999 to leverage the 3D engine team and technology into a new web browser–based technology, the "WildTangent Web Driver." He architected the technology and managed the technology team, as well as assisting with the development of a few of the many games that used the engine. Mike has remained with WildTangent for more than 10 years, helping the company with different technical challenges as it transitioned from a game and technology developer to a game distributer; he currently serves as a principal engineer. Mike Sandige has credits on more than 20 games and other published software titles.

Chapter 1

Boolean Algebra, Boolean Functions, VHDL, and Gates

Chapter Outline

1.1 INTRODUCTION

In this chapter, you will learn three different ways to express Boolean functions. **Boolean algebra** is a mathematical form used to represent a Boolean function. Gates are a graphical form to represent a Boolean function. VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a textual form to represent a Boolean function. Boolean functions are used in industry to specify logic or digital circuits. Logic gates or digital circuits represent a class of circuits that produce operations of the yes–no (true–false or 1–0) variety. These types of circuits are the ones used in the computer industry and many other areas that use digital circuits. It is hard to get through each day without using a digital circuit. For example, digital circuits are used in cell phones, calculators, computers, washing machines, microwave ovens, and kid's and adult's toys (from simple music producing toys to automobiles of all types)—the list is practically endless.

1.2 BASICS OF BOOLEAN ALGEBRA

Boolean algebra consists of a set of independent statements called **postulates** that are assumed to be true without proof. The term *Boolean* is derived from the writings of George Boole and a book he published in 1854 called *An Investigation of the Laws of Thought.* Today, there are several ways to form Boolean algebra. H. E. Huntington published several sets of independent statements or postulates for Boolean algebra in a paper published in 1904 called "Sets of Independent Postulates for the Algebra of Logic." Both Boole's book and Huntington's paper are quite mathematical. Books and papers are archived and are available in the library for those interested in reading about their work. Boole and Huntington were early pioneers in the area of Boolean algebra.

Perhaps the simplest way to describe Boolean algebra is to illustrate the major binary operators ("$\overline{\ }$", "·", and "+") with variables (or signals) in a list like the one shown in Table 1.1.

TABLE 1.1 Major binary operators with variables (or signals) in a Boolean expression

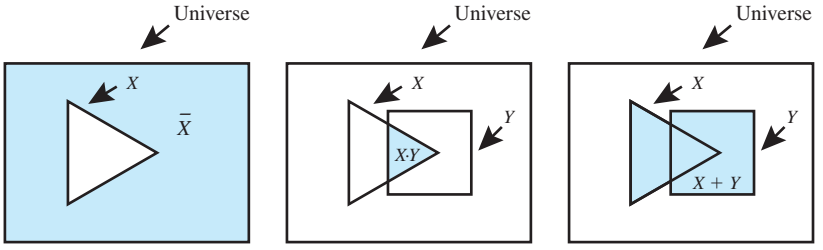
Boolean expression	Type of operation	VHDL equivalent
$\overline{X}$	**Complement** of X	**NOT** X
$X \cdot Y$	**Intersection** of X, Y	X **AND** Y
$X + Y$	**Union** of X, Y	X **OR** Y

Things you should notice about the Boolean expressions in Table 1.1:

- The signal X with a bar over it, or $\overline{X}$, is called the complement of X. This is equivalent to **NOT** X in VHDL. VHDL is a language for designing digital circuits. You will learn VHDL in a natural manner by simply studying the design examples that we provide.
- The signal X with a raised dot followed by the signal Y, or $X \cdot Y$, is called the intersection of X, Y and is equivalent to X **AND** Y in VHDL.
- The signal X with a plus sign followed by the signal Y, or $X + Y$ is called the union of X, Y and is equivalent to X **OR** Y in VHDL.

1.2.1 Venn Diagrams

Figure 1.1 shows Venn diagrams that illustrate the complement, the intersection, and the union operations in a graphical form in a universe (the universe is the area of the large rectangle).



Complement of X (shaded area, $\overline{X}$)

Intersection of X, Y (shaded area, $X \cdot Y$)

Union of X, Y (shaded area, $X + Y$)

FIGURE 1.1 Venn diagrams for the complement, the intersection, and the union operations

A Venn diagram is a graphical interpretation of an algebraic operation. In Figure 1.1, the area for X is represented as a triangle, while the area for Y is shown as a square. It is not important what objects are used for the areas of X and Y.

Things you should notice about the Venn diagrams in Figure 1.1:

- Each Venn diagram represents an interpretation of an algebraic operation or Boolean expression inside a universe.
- The Boolean expression for the complement of X is $\overline{X}$. The Boolean expression for the intersection of X, Y is $X \cdot Y$. The Boolean expression for the union of X, Y is $X + Y$.

- The complement of X in the Venn diagram for the Boolean expression $\overline{X}$ is the area outside of object X.
- The intersection of X, Y in the Venn diagram for the Boolean expression $X{\cdot}Y$ is the area where object X and object Y overlap.
- The union of X, Y in the Venn diagram for the Boolean expression $X + Y$ is the area for both object X and object Y.

1.2.2 Black Boxes for Boolean Functions

The term **black box** is used in circuit theory to specify the inputs and outputs for a digital circuit. A digital circuit may be described by a Boolean function (or Boolean equation). The definition of a Boolean function (or Boolean equation) is a dependent variable such as F set equal to a Boolean expression such as $X + Y$, where X and Y are independent variables. The functional notation such as $F(X,Y)$ is also used in Boolean algebra in the same way it is used in ordinary algebra. A function may be assigned a value of 0 or 1 for each of the possible 2^n combinations of the binary values for n independent variables. Consider the Boolean function $F(X) = \overline{X}$ or simply $F = \overline{X}$. In the Boolean function $F(X) = \overline{X}$, X is an independent variable or signal and $\overline{X}$ is the complement of X. A dependent variable is normally an output signal in a digital circuit while the independent variable is the input signal in the digital circuit as shown by the black box in Figure 1.2.

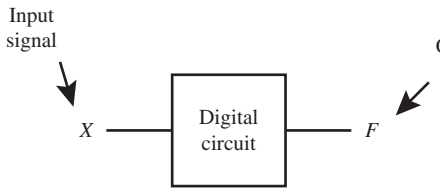


FIGURE 1.2 Black box for a digital circuit with a single input and output

What's inside the black box or the contents of the black box are not visible. The black box shows only the input signal and the output signal for a digital circuit, but it does not show the actual logic circuitry for the Boolean function. In the black box in Figure 1.2, the output signal F is a function of or is dependent on the input signal X. The relationship of the output signal and the input signal is provided by the Boolean function for F, that is, $F = \overline{X}$. The function $F = X$ also satisfies the criteria for the black box in Figure 1.2. This function will be covered later.

A black box for a function with two input signals and one output signal is shown in Figure 1.3. This black box can represent any Boolean function that has the input signals X and Y and also has an output signal F.

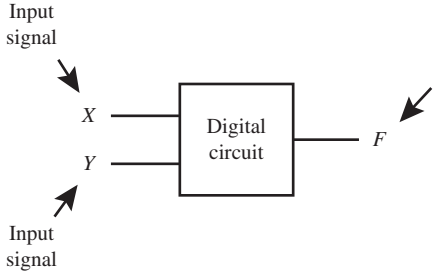


FIGURE 1.3 Black box for a digital circuit with two inputs and one output

Two Boolean functions that meet the criteria for the black box in Figure 1.3 are written as $F(X,Y) = X{\cdot}Y$ or simply $F = X{\cdot}Y$, and $F(X,Y) = X + Y$ or simply $F = X + Y$. There are other Boolean functions that meet the criteria for the black box in Figure 1.3. Some of these functions will be covered later.

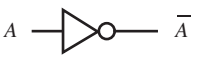
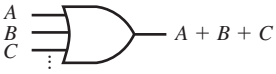
Things you should notice about the black boxes in Figures 1.2 and 1.3:

- Black boxes do not show the actual logic circuits inside the black boxes.
- Black boxes show only the inputs and output for a Boolean function.
- Input signals are normally placed on the left side of a black box.
- Output signals are normally placed on the right side of a black box.
- Each line going into or coming out of a black box represents a single wire or a net. The electrical signals X, Y, and F travel along their respective wire or net.

1.2.3 Basic Logic Symbols

To draw a logic circuit diagram, a digital circuit, or a schematic for each of the Boolean functions $F1 = \overline{X}$, $F2 = X{\cdot}Y$, and $F3 = X + Y$, you need to know the gate symbols. Figure 1.4 shows a logic symbol summary for the **NOT** operation, **AND** operation, and **OR** operation.

FIGURE 1.4 Logic symbol summary for the **NOT** operation, **AND** operation, and **OR** operation

Boolean algebra expression	Name of expression	Distinctive-shape logic symbol	Logic name
$\overline{A}$	NOT or Complement operation		Inverter or NOT Gate
$A{\cdot}B{\cdot}C{\cdot}\ldots$	AND operation		AND Gate
$A + B + C + \ldots$	OR operation		OR Gate

Things you should notice about the logic symbol summary in Figure 1.4:

- The Inverter or NOT gate has only one input and one output.
- The AND gate has two or more inputs as indicated by the dotted lines below the C input and one output. For three signals, $A{\cdot}B{\cdot}C$ is equivalent to A **AND** B **AND** C in VHDL.
- The OR gate has two or more inputs as indicated by the dotted lines below the C input and one output. For three signals, $A + B + C$ is equivalent to A **OR** B **OR** C in VHDL.
- Logic symbols may be drawn using distinctive shapes or using rectangular shapes. Rectangular-shape logic symbols will be covered later in this chapter.
- The shape of the symbols in Figure 1.4 indicates the gate type. The NOT gate has a bubble on the output. In Figure 1.4, the bubble is the circle that is placed on the apex or tip of the triangle on the output. The AND gate has a straight line on its input side and a semicircle on the output. The OR gate has a curved line on its input side and two curved lines forming an apex on the output. The distinctive-shape AND gate and the OR gate do not have a bubble on their outputs.

You need to learn the distinctive-shape gate symbols because that is what we will be using throughout this book.

The logic circuit diagrams or schematic diagrams for the Boolean functions $F1 = \overline{X}$, $F2 = X{\cdot}Y$, and $F3 = X + Y$ are shown in Figure 1.5 using distinctive-shape gate symbols.

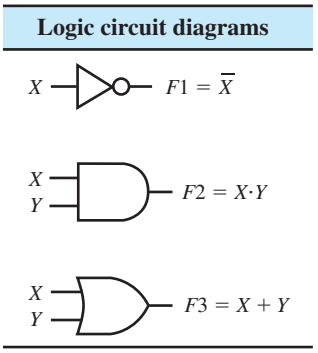


FIGURE 1.5 Logic circuit diagrams for Boolean functions *F*1, *F*2, *and F*3

Things you should notice about the logic circuit diagrams in Figure 1.5:

- The logic circuit diagram for the Boolean function $F1 = \overline{X}$ is represented by the distinctive-shape symbol for the NOT gate. The signal X is shown on the input side (left side) of the NOT gate symbol and the signal $F1$ is shown on the output side (right side) of the symbol.
- The logic circuit diagram for the Boolean function $F2 = X{\cdot}Y$ is represented by the distinctive-shape symbol for the AND gate. The signals X and Y are shown on the input side (left side) of the AND gate symbol and the signal $F2$ is shown on the output side (right side) of the symbol.
- The logic circuit diagram for the Boolean function $F3 = X + Y$ is represented by the distinctive-shape symbol for the OR gate. The signals X and Y are shown on the input side (left side) of the OR gate symbol and the signal $F3$ is shown on the output side (right side) of the symbol.

A truth table is a tabular form for presenting the yes–no, true–false, or 1–0, values of its variables or signals. The binary operators ("$\overline{\ }$", "·", and "+") or **NOT**, **AND**, and **OR** operators, respectively, are defined by the truth tables shown in Table 1.2.

TABLE 1.2 Truth tables for the NOT, AND, and OR operator definitions

Definitions					
NOT operation		**AND operation**		**OR operation**	
X	$\overline{X}$	$X\ Y$	$X{\cdot}Y$	$X\ Y$	$X + Y$
0	1	0 0	0	0 0	0
1	0	0 1	0	0 1	1
		1 0	0	1 0	1
		1 1	1	1 1	1

Things you should notice about the truth tables in Table 1.2:

- A truth table is a tabular form for presenting the yes–no, true–false, or 1–0 values of its variables or signals. Note: Number of rows in a truth table $= 2^{\text{Number of input signals or variables}}$. For one variable, $2^1 = 2$, so the truth table for the NOT operation has two rows. For two variables, $2^2 = 4$, so the truth table for the AND operation and also the OR operation has four rows.

- The values of the signals X and Y only have the values of 0 or 1. The values 0 and 1 are called **binary digits** or **bits** for short. In the truth tables for the AND and the OR operations, row 00 represents decimal 0, row 01 represents decimal 1, row 10 represents decimal 2, and row 11 represents decimal 3. This is assuming that signal X represents the **MSB** (most significant bit) and signal Y represents the **LSB** (least significant bit) in each of the binary numbers 00, 01, 10, and 11 as read from left to right.
- The value of 0 represents false, while the value of 1 represents true. The value 0 and the value 1 are also called *identity elements* in Boolean algebra.
- The **NOT** operator represents the complement of a variable—that is, if the variable is 0 its complement is 1, or if the variable is 1 its complement is 0.
- The **AND** operator represents the intersection of two variables such that their intersection is 1 (or true) only if both variables are 1 (or true).
- The **OR** operator represents the union of two variables such that their union is 1 (or true) when one or both variables are true.
- The complement only applies to one variable at a time such that the complement of X is $\overline{X}$, or the complement of Y is $\overline{Y}$.
- The intersection applies to two or more variables, such that $X{\cdot}Y{\cdot}Z$ has the value of 1 only when all values of the variables X, Y, and Z are 1.
- The union applies to two or more variables, such that that $X + Y + Z$ has the value of 1 when one or more of the values of the variables X, Y, and Z is 1.

The binary operators can also be defined by writing all possible combinations of the identity elements (0 and 1) with each respective operator as illustrated in Table 1.3.

TABLE 1.3 Definition of binary operators NOT, AND, and OR

Definition of NOT operator (for 1 identity element)	Definition of AND operator (for 2 identity elements)	Definition of AND operator (for 3 identity elements)	Definition of AND operator (for 4 identity elements)	Definition of OR operator (for 2 identity elements)	Definition of OR operator (for 3 identity elements)	Definition of OR operator (for 4 identity elements)
$\overline{1} = 0$	$0{\cdot}0 = 0$	$0{\cdot}0{\cdot}0 = 0$	$0{\cdot}0{\cdot}0{\cdot}0 = 0$	$0 + 0 = 0$	$0 + 0 + 0 = 0$	$0 + 0 + 0 + 0 = 0$
$\overline{0} = 1$	$0{\cdot}1 = 0$	$0{\cdot}0{\cdot}1 = 0$	$0{\cdot}0{\cdot}0{\cdot}1 = 0$	$0 + 1 = 1$	$0 + 0 + 1 = 1$	$0 + 0 + 1 + 0 = 1$
	$1{\cdot}0 = 0$	$0{\cdot}1{\cdot}0 = 0$	$0{\cdot}0{\cdot}1{\cdot}0 = 0$	$1 + 0 = 1$	$0 + 1 + 0 = 1$	$0 + 0 + 1 + 0 = 1$
	$1{\cdot}1 = 1$	$0{\cdot}1{\cdot}1 = 0$	·	$1 + 1 = 1$	$0 + 1 + 1 = 1$	·
		$1{\cdot}0{\cdot}0 = 0$	·		$1 + 0 + 0 = 1$	·
		$1{\cdot}0{\cdot}1 = 0$	·		$1 + 0 + 1 = 1$	·
		$1{\cdot}1{\cdot}0 = 0$	$1{\cdot}1{\cdot}1{\cdot}0 = 0$		$1 + 1 + 0 = 1$	$1 + 1 + 1 + 0 = 1$
		$1{\cdot}1{\cdot}1 = 1$	$1{\cdot}1{\cdot}1{\cdot}1 = 1$		$1 + 1 + 1 = 1$	$1 + 1 + 1 + 1 = 1$

Things you should notice about the definitions of the binary operators in Table 1.3:

- The NOT operator definition only applies to a single identity element.
- The AND operator definition applies to two or more identity elements.
- The OR operator definition applies to two or more identity elements.
- The NOT operator definition provides the complement of the identity element (either 0 or 1).

- The AND operator definition provides a 1 only when all the identity elements that are ANDed are a 1, else it provides a 0.
- The OR operator definition provides a 1 when any one of the identity elements that are ORed are a 1, else it provides a 0.

1.2.4 Boolean Algebra Postulates

As we mentioned earlier, Boolean algebra can be represented by a list of postulates. Table 1.4 shows a list of postulates from Huntington's first set. We accept Huntington's postulates and use them for defining two-valued Boolean algebra.

TABLE 1.4 List of Huntington's first set of postulates

Postulates for Boolean algebra with two distinct values (1 or 0) for each variable		
Variable dominant rule	P1a:	$X \cdot 1 = X$
	P1b:	$X + 0 = X$
Commutative rule	P2a:	$X \cdot Y = Y \cdot X$
	P2b:	$X + Y = Y + X$
Distributive rule	P3a:	$X \cdot (Y + Z) = X \cdot Y + X \cdot Z$
	P3b:	$X + Y \cdot Z = (X + Y) \cdot (X + Z)$
Complement rule	P4a:	$X \cdot \overline{X} = 0$
	P4b:	$X + \overline{X} = 1$

In Boolean algebra, there is a hierarchy or order of precedence for the binary operators ("¯", "·", and "+"). When writing Boolean algebra expressions, the **NOT** operator has the highest precedence followed by the **AND** operator followed by the **OR** operator. In postulate P3a, the expression on the right side is evaluated in the order $X \cdot Y$ and $X \cdot Z$ then the two expressions are **OR**ed. In postulate P4b, the expression on the left side is evaluated in the order X and $\overline{X}$ then the two expressions are **OR**ed. In the postulate P3a, the expression on the left side is evaluated in the order X and $Y + Z$ then the two expressions are **AND**ed. In this case, the parentheses surrounding the expression $Y + Z$ forces the **OR** operator to have a higher precedence than the **AND** operator. In Boolean algebra, as in normal algebra, parentheses may be used to establish the desired order of precedence of the binary operators.

Each postulate or rule has a dual. In Table 1.4, P1a and P1b, P2a and P2b, etc., are dual relationships. The dual of a Boolean relationship or a Boolean expression is obtained by interchanging the identity elements (1 and 0) and the binary operators (· and +) while maintaining the order of precedence of the operators using parentheses if required. Notice that parentheses are required to obtain the dual of the right side of P3b. The dual $X + Y \cdot Z$ can be written as $(X + Y \cdot Z)^{\mathrm{D}}$. The superscript D is used to indicate that the dual of the expression in parentheses needs to be taken. First add parentheses around $Y \cdot Z$ and then interchange the binary operators (· and +), so the dual of $X + Y \cdot Z$ or $(X + Y \cdot Z)^{\mathrm{D}} = (X + (Y \cdot Z))^{\mathrm{D}}$ is $X \cdot (Y + Z)$. The result of the interchange is a new Boolean expression. The dual of $X \cdot \overline{X} = 0$ or $(X \cdot \overline{X} = 0)^{\mathrm{D}}$ is $X + \overline{X} = 1$. Caution: When obtaining the dual of a relationship that has complemented variables, only interchange the identity elements (1 and 0) and the binary operator (· and +) and not the complemented variables. Interchanging the complemented variables appears to work for obtaining the dual of P4a or P4b, but this is not true in general.

1.2.5 Boolean Algebra Theorems

Postulates can be used to prove a number of useful Boolean algebra theorems. Theorems are useful equalities that are helpful in designing logic circuits. Table 1.5 shows a list of theorems for Boolean algebra and their associated names. Notice that each of the theorems have a dual except Theorem T7, because theorem T7 does not contain any identity elements or binary operators to interchange.

TABLE 1.5 List of useful Boolean algebra theorems

Theorems for Boolean algebra	
Absorption Theorem	T1a: $X \cdot (X + Y) = X$
	T1b: $X + X \cdot Y = X$
Adjacency Theorem	T2a: $X \cdot Y + X \cdot \overline{Y} = X$
	T2b: $(X + Y) \cdot (X + \overline{Y}) = X$
Associative Theorem	T3a: $X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$
	T3b: $X + (Y + Z) = (X + Y) + Z$
Consensus Theorem	T4a: $X \cdot Y + \overline{X} \cdot Z + Y \cdot Z = X \cdot Y + \overline{X} \cdot Z$
	T4b: $(X + Y) \cdot (\overline{X} + Z) \cdot (Y + Z) = (X + Y) \cdot (\overline{X} + Z)$
DeMorgan's Theorem (with two variables)	T5a: $\overline{X \cdot Y} = \overline{X} + \overline{Y}$
	T5b: $\overline{X + Y} = \overline{X} \cdot \overline{Y}$
DeMorgan's Theorem (with multiple variables)	T6a: $\overline{X \cdot Y \cdot Z \ldots} = \overline{X} + \overline{Y} + \overline{Z} + \ldots$
	T6b: $\overline{X + Y + Z + \ldots} = \overline{X} \cdot \overline{Y} \cdot \overline{Z} \cdot \ldots$
Double Complementation or Double Negation Theorem	T7: $\overline{\overline{X}} = X$
Idempotency Theorem	T8a: $X \cdot X = X$
	T8b: $X + X = X$
Identity Element Theorem	T9a: $X \cdot 0 = 0$
	T9b: $X + 1 = 1$
Simplification Theorem	T10a: $X \cdot (\overline{X} + Y) = X \cdot Y$
	T10b: $X + \overline{X} \cdot Y = X + Y$

Boolean algebra postulates and theorems are used to minimize Boolean expressions so that logic circuits can be constructed using the fewest number of gates. For example, the expression on the left side of Adjacency Theory T2a requires a circuit with three Boolean operators that results in multiple gates, while the expression on the right side is equivalent and only requires a single wire and thus results in a minimum logic circuit. Figure 1.6 shows the comparison of the two different, but equivalent, circuits provided by the Boolean function $F1 = X \cdot Y + X \cdot \overline{Y}$, and the equivalent Boolean function $F2 = X$, for Adjacency Theorem T2a.

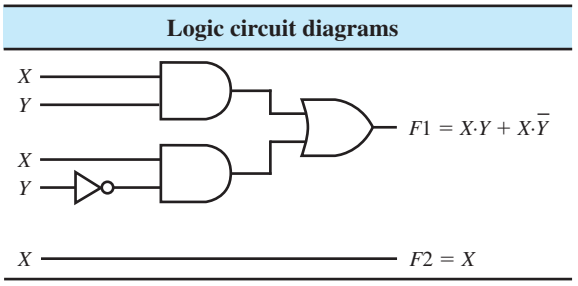


FIGURE 1.6 Comparing circuit complexity for the Boolean functions *F*1 and *F*2

In general, it is better to use fewer logic gates to minimize complexity, cost, and the power requirement when designing logic circuits. Theorems are mainly used to minimize simple Boolean functions, or functions with only a few operators, so that circuits can be implemented with a minimum gate count, that is, a minimum number of gates.

1.2.6 Proving Boolean Algebra Theorems

We are not interested in deriving theorems because we leave that task up to logicians or mathematicians. A method called **perfect induction** is perhaps the easiest way to prove that the expression on the left (EOL) side of a theorem is equivalent to the expression on the right (EOR) side. Selectively using the postulates and/or other theorems via a method called **mathematical manipulation** is another way to prove a theorem is correct. The latter method is considered somewhat tricky compared to the former method, because it involves trial and error. When using the mathematical manipulation method, you may choose to prove EOL = EOR or that EOR = EOL. Figure 1.7 illustrates the perfect induction method and also the mathematical manipulation method for proving the validity of Simplification Theorem T10b.

Simplification Theorem T10b: $X + \overline{X}\cdot Y = X + Y$

Proof by perfect induction method

Step 1: Make the truth table

X	Y	$\overline{X}$	$\overline{X}\cdot Y$	$X + \overline{X}\cdot Y$	$X + Y$
0	0				
0	1				
1	0				
1	1				

Step 2: Fill in each column in the table using the operator definitions

X	Y	$\overline{X}$	$\overline{X}\cdot Y$	$X + \overline{X}\cdot Y$	$X + Y$
0	0	1	0	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	1	0	0	1	1

EOL = EOR

Step 3: Verify that EOL = EOR

Proof by mathematical manipulation method (proving that EOR = EOL)

Substitute postulates and/or theorems to show that EOR = EOL, which can be tricky because it involves trial and error

$X + Y$	Using the following postulates and/or theorems:	
$X + Y\cdot 1$	Variable dominate rule	P1a: $X\cdot 1 = X$
$X + Y\cdot(X + \overline{X})$	Complement rule	P4b: $X + \overline{X} = 1$
$X + Y\cdot X + Y\cdot\overline{X}$	Distributive rule	P3a: $X\cdot(Y + Z) = X\cdot Y + X\cdot Z$
$X\cdot 1 + Y\cdot X + Y\cdot\overline{X}$	Variable dominate rule	P1a: $X\cdot 1 = X$
$X\cdot 1 + X\cdot Y + \overline{X}\cdot Y$	Commutative rule	P2a: $X\cdot Y = Y\cdot X$
$X\cdot(1 + Y) + \overline{X}\cdot Y$	Distributive rule	P3a: $X\cdot(Y + Z) = X\cdot Y + X\cdot Z$
$X\cdot 1 + \overline{X}\cdot Y$	Identity element theorem	T9b: $X + 1 = 1$
$X + \overline{X}\cdot Y$	Variable dominate rule	P1a: $X\cdot 1 = X$

Note that EOR = EOL

FIGURE 1.7 Proving Simplification Theorem T10b by the perfect induction method and the mathematical manipulation method

One of the most important theorems in Boolean algebra is Adjacency Theorem T2a: $X \cdot Y + X \cdot \overline{Y} = X$. Figure 1.8 shows a proof of the Adjacency Theorem using the perfect induction method and also using the mathematical manipulation method.

Adjacency Theorem T2a: $X \cdot Y + X \cdot \overline{Y} = X$

Proof by perfect induction method

X Y	$\overline{Y}$	$X \cdot Y$	$X \cdot \overline{Y}$	$X \cdot Y + X \cdot \overline{Y}$
0 0	1	0	0	0
0 1	0	0	0	0
1 0	1	0	1	1
1 1	0	1	0	1
EOR		=		EOL

Proof by mathematical manipulation method (proving that EOL = EOR)

$X \cdot Y + X \cdot \overline{Y}$	Using the following postulates and/or theorems:
$X \cdot (Y + \overline{Y})$	Distributive rule — P3a: $X \cdot (Y + Z) = X \cdot Y + X \cdot Z$
$X \cdot 1$	Complement rule — P4b: $X + \overline{X} = 1$
X	Variable dominate rule — P1a: $X \cdot 1 = X$
Note that EOL = EOR	

FIGURE 1.8 Proof of the Adjacency Theorem using the perfect induction method and the mathematical manipulation method

1.3 DERIVING BOOLEAN FUNCTIONS FROM TRUTH TABLES

A standard way of expressing a logic function or Boolean function is by a truth table. After a truth table is written and filled in, the Boolean function can be derived in terms of the input signals for the 1s of the function written as a Boolean expression, or a Boolean function can be derived in terms of the input signals for the 0s of the function written as a Boolean expression. So,

Boolean function:

F = Boolean expression (in terms of the 1s of the function), or

F = Boolean expression (in terms of the 0s of the function)

In the following discussions we show how to derive Boolean functions both ways.

1.3.1 Deriving Boolean Functions Using the 1s of the Functions

In this section we show how to derive Boolean functions from truth tables. To do this, we will use the Boolean function called an **XOR (Exclusive OR)** function. The truth table for an XOR function with two inputs is shown in Table 1.6. When the two input signal values are different, the function F_{XOR} is 1, otherwise the function is 0.

TABLE 1.6 Truth table for an XOR function

XOR function		
Decimal	Binary	
X Y	X Y	F_{XOR}
0	0 0	0
1	0 1	1
2	1 0	1
3	1 1	0

Notice that the output value of the function F_{XOR} is 1 when signal X and signal Y are different values. This occurs in two places in the truth table, when $X\ Y = 01$ and when $X\ Y = 10$. A Boolean expression for the case when $X\ Y = 01$ or $X = 0$ occurs *at the same time* that $Y = 1$ occurs is simply $\overline{X}\cdot Y$.

Note: $X = 0$ so we write $\overline{X}$
$Y = 1$ so we write Y

An **AND** operator must be used between the expression $\overline{X}$ and the expression Y to form the expression $\overline{X}\cdot Y$, because the signals $X\ Y = 01$ occur at the same time, which means that they have an **intersection**.

Observe: $X\ Y = 01$ substituted into the expression $\overline{X}\cdot Y$
results in $\overline{X}\cdot Y = \overline{0}\cdot 1 = 1$, which is required for F_{XOR} to be equal to 1

A Boolean expression for the case when $X\ Y = 10$ or $X = 1$ occurs at the same time that $Y = 0$ occurs is simply $X\cdot\overline{Y}$.

Note: $X = 1$ so we write X
$Y = 0$ so we write $\overline{Y}$

An **AND** operator must be used between the expression X and the expression $\overline{Y}$ to form the expression $X\cdot\overline{Y}$, because the signals $X\ Y = 10$ occur at the same time or have an intersection.

Observe: $X\ Y = 10$ substituted into the expression $X\cdot\overline{Y}$
results in $X\cdot\overline{Y} = 1\cdot\overline{0} = 1$, which is required for F_{XOR} to be equal to 1

The Boolean function for the union of $\overline{X}\cdot Y$ and $X\cdot\overline{Y}$ is written as $F_{XOR} = \overline{X}\cdot Y + X\cdot\overline{Y}$. An **OR** operator must be used between the two expressions $\overline{X}\cdot Y$ and $X\cdot\overline{Y}$, because the two expressions occur either at the same time or at a different time, which means that they have a **union**.

The Boolean expression $\overline{X}\cdot Y + X\cdot\overline{Y}$ is used so often in digital design that the special **XOR** operator $(\oplus)$ is used to simplify writing the expression, so $X\oplus Y = \overline{X}\cdot Y + X\cdot\overline{Y}$. The simplified expression $X\oplus Y$ is equivalent to X ***XOR*** Y in VHDL. The Boolean function for F_{XOR} can be written as $F_{XOR} = X\oplus Y$.

1.3.2 Deriving Boolean Functions Using the 0s of the Functions

A Boolean expression can be written for the complement of the **XOR** function in Table 1.6 as $\overline{F_{XOR}}$. To derive the Boolean function for $\overline{F_{XOR}}$ requires using the values of F_{XOR} that are 0s. When a function has fewer 0s, it may be advantageous to use the 0s to obtain the function rather than using the 1s. Because the number of 1s and 0s is the same for the **XOR** function, the complexity of the Boolean expression for each Boolean function will be the same. By *complexity*, we are referring to the number of variables that are contained in the Boolean expression for the function. A smaller number of variables results in less complexity. Notice that the output value of the function F_{XOR} is 0 when signals X and Y are the same. This occurs in two places in the truth table: when $X\ Y = 00$ and when $X\ Y = 11$. A Boolean expression for the case when $X\ Y = 00$ or $X = 0$ occurs at the same time that $Y = 0$ occurs is simply $\overline{X}\cdot\overline{Y}$.

Note: $X = 0$ so we write $\overline{X}$
$Y = 0$ so we write $\overline{Y}$

An **AND** operation is used in the Boolean expression $\overline{X}\cdot\overline{Y}$ because the signals $X\ Y = 00$ occur at the same time or have an intersection.

A Boolean expression for the case when $X\ Y = 11$ or $X = 1$ occurs at the same time that $Y = 1$ occurs is simply $X{\cdot}Y$.

Note: $X = 1$ so we write X
$Y = 1$ so we write Y

An **AND** operation is used in the Boolean expression $X{\cdot}Y$ because the signals $X\ Y = 11$ occur at the same time or have an intersection.

Because the two expressions $\overline{X}{\cdot}\overline{Y}$ and $X{\cdot}Y$ each provide a 1 at the same time or at a different time, they have a union. Because $F_{XOR} = 0$ for these expressions, the complement F_{XOR} must be used when combining the Boolean expressions to form the Boolean function, or $\overline{F_{XOR}} = \overline{X}{\cdot}\overline{Y} + X{\cdot}Y$. Complementing both sides of the Boolean function $\overline{F_{XOR}} = \overline{X}{\cdot}\overline{Y} + X{\cdot}Y$ results in the Boolean function $F_{XOR} = \overline{\overline{X}{\cdot}\overline{Y} + X{\cdot}Y}$. Apply DeMorgan's theorem to the Boolean expression, and we obtain $F_{XOR} = (X + Y){\cdot}(\overline{X} + \overline{Y})$. Either the first Boolean function we derived for F_{XOR} using the 1s of function or the second Boolean function we derived for F_{XOR} using the 0s of function can be used to **design** or **implement** a circuit for F_{XOR}.

1.3.3 Deriving Boolean Functions Using Minterms and Maxterms

There is an elegant way to derive a Boolean function from a truth table by obtaining intermediate **product terms** called **minterms**. The minterms are then **OR**ed together to form the Boolean function. Minterms are used to write Boolean functions in a simple and concise form. A minterm designator m_i is assigned to every row of a truth table where $i = 0, 1, 2, \ldots$, which represents the decimal values of the inputs. In the **XOR** Boolean function truth table in Table 1.6, the minterms are uniquely defined as follows: $m_0 = \overline{X}{\cdot}\overline{Y}$, $m_1 = \overline{X}{\cdot}Y$, $m_2 = X{\cdot}\overline{Y}$, and $m_3 = X{\cdot}Y$. Observe that each minterm expression is defined such that its value evaluates to 1 when the minterm number i in binary is applied to the expression, because 0s are replaced by a complemented variable and 1s are replaced by an uncomplemented variable. Note: $m_1 = \overline{X}{\cdot}Y$, where $X\ Y = 01$. **Memory Jogger**: For minterms, place an overbar just over the 0s. For minterm 1 (m_1), $X\ Y = 01$, so the minterm expression is $\overline{X}{\cdot}Y$ and the expression evaluates to 1 when 01 is substituted into the expression. The same process is used to obtain any minterm for any number of variables. Example: For the three variables X, Y, and Z, $m_6 = X{\cdot}Y{\cdot}\overline{Z}$, and for the four variables A, B, C, and D, $m_{10} = A{\cdot}\overline{B}{\cdot}C{\cdot}\overline{D}$.

The **XOR** Boolean function can be written using the following **compact minterm form**: $F_{XOR}(X,Y) = \Sigma m(1,2)$ for the 1s of the function. The notation $\Sigma m(1,2)$ means $m_1 + m_2$, so $m_1 + m_2 = \overline{X}{\cdot}Y + X{\cdot}\overline{Y}$. Putting all the parts together, we can write the **XOR** Boolean function as $F_{XOR}(X, Y) = \Sigma m(1, 2) = m_1 + m_2 = \overline{X}{\cdot}Y + X{\cdot}\overline{Y}$. This is an algorithmic procedure that is usually memorized, and unfortunately, it is often applied without fully comprehending the derivation of a Boolean function for its 1s, as illustrated in Section 1.3.1. A function written in the form $F_{XOR}(X, Y) = \overline{X}{\cdot}Y + X{\cdot}\overline{Y}$ is referred to as the **canonical sum of products (CSOP) form** of the function.

Why is a lowercase m used as a minterm designator? In a truth table for the function $F(X,Y, \ldots) = m_i$ for $i = 0$, or $i = 1$, or $i = 2, \ldots$ the product term m_i is designated as a lowercase m, because there is a minimum number of 1s in the truth table for the function $F(X,Y, \ldots) = m_i$—that is, just 1.

Intermediate **sum terms** called **maxterms** may also be used to derive a Boolean function. A maxterm designator M_i is assigned to every row of a truth table where $i = 0, 1, 2, \ldots$, which represent the decimal values of the inputs. In the **XOR** function truth table in Table 1.6, the maxterms are uniquely defined as follows: $M_0 = X + Y$, $M_1 = X + \overline{Y}$, $M_2 = \overline{X} + Y$, and $M_3 = \overline{X} + \overline{Y}$. Observe that each maxterm expression is defined such that its value evaluates to 0 when the maxterm number i is applied to the expression, because 0s are replaced by an

uncomplemented variable and 1s are replaced by a complemented variable. Note: $M_1 = X + \overline{Y}$, where $X\,Y = 01$. **Memory Jogger**: For maxterms, place an overbar just over the 1s. For maxterm 1 (M_1), $X\,Y = 01$, so the maxterm expression is $X + \overline{Y}$ and the expression evaluates to 0 when 01 is substituted into the expression. The same process is used to obtain any maxterm for any number of variables. Example: For the three variables X, Y, and Z, $M_6 = \overline{X} + \overline{Y} + Z$, and for the four variables A, B, C, and D, $M_{10} = \overline{A} + B + \overline{C} + D$.

The **XOR** Boolean function can be written using the following **compact maxterm form**: $F_{XOR}(X,Y) = \Pi M(0,3)$ for the 0s of the function. The notation $\Pi M(0,3)$ means $M_0 \cdot M_3$, so $M_0 \cdot M_3 = (X + Y)\cdot(\overline{X} + \overline{Y})$. Putting all the parts together, we can write the **XOR** Boolean function as $F_{XOR}(X,Y) = \Pi M(0,3) = M_0 \cdot M_3 = (X + Y)\cdot(\overline{X} + \overline{Y})$. A function written in the form $F_{XOR}(X,Y) = (X + Y)\cdot(\overline{X} + \overline{Y})$ is referred to as the **canonical product of sums (CPOS) form** of the function.

Why is an uppercase M used as a maxterm designator? In a truth table for the function $F(X,Y,\ldots) = M_i$ for $i = 0$, or $i = 1$, or $i = 2, \ldots$ the sum term M_i is designated as an uppercase M, because there is a maximum number of 1s in the truth table for the function $F(X,Y,\ldots) = M_i$—that is, all except 1.

It is interesting to observe that minterms and maxterms for the same variables are complements of each other—that is, one can be obtained from the other by complementation. The relationship between a minterm and its corresponding maxterm is written as $m_0 = \overline{M_0}$, $m_1 = \overline{M_1}$, $m_2 = \overline{M_2}$, and $m_3 = \overline{M_3}$ or $m_i = \overline{M_i}$ where $i = 0, 1, 2, \ldots$. Compact minterm and maxterm forms represent a concise way to represent truth tables.

For the truth table shown in Table 1.7, write the compact minterm forms for the function $F1$.

TABLE 1.7 Truth table for function *F*1

(*Decimal*)	X	Y	Z	$F1$
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

The compact minterm forms for $F1$ are written as follows:

$F1(X,Y,Z) = \Sigma m(0,2,5,6,7)$ Explicit compact minterm form for the 1s of the function $F1$

$\overline{F1}(X,Y,Z) = \Sigma m(1,3,4)$ Explicit compact minterm form for the 0s of the function $F1$

The Boolean equation for either $F1$ or $\overline{F1}$ represents the truth table for the function $F1$. To write an implicit compact minterm form, simply leave off the m following the summation symbol. Example: $F1(X,Y,Z) = \Sigma(0,2,5,6,7)$ is the implicit compact minterm form for the 1s of the function $F1$, and $\overline{F1}(X,Y,Z) = \Sigma(1,3,4)$ is the implicit compact minterm form for the 0s of the function $F1$.

Using the compact minterm form for $F1$, we can write the CSOP form of the function $F1$ as follows:

$$F1(X,Y,Z) = m0 + m2 + m5 + m6 + m7$$
$$= \overline{X}\cdot\overline{Y}\cdot\overline{Z} + \overline{X}\cdot Y\cdot\overline{Z} + X\cdot\overline{Y}\cdot Z + X\cdot Y\cdot\overline{Z} + X\cdot Y\cdot Z$$

Using the compact minterm form for $\overline{F1}$, we can write the CSOP form of the function $\overline{F1}$ as follows:

$$\overline{F1}(X,Y,Z) = m1 + m3 + m4$$
$$= \overline{X}\cdot\overline{Y}\cdot Z + \overline{X}\cdot Y\cdot Z + X\cdot\overline{Y}\cdot\overline{Z}$$

For the truth table shown in Table 1.8, write the compact maxterm forms for the function $F2$.

TABLE 1.8 Truth table for function *F2*

(*Decimal*)	*X*	*Y*	*Z*	*F2*
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

The compact maxterm forms for $F2$ are written as follows:

$F2(X,Y,Z) = \Pi M(3,4,5)$ Explicit compact maxterm form for the 0s of the function $F2$

$\overline{F2}(X,Y,Z) = \Pi M(0,1,2,6,7)$ Explicit compact maxterm form for the 1s of the function $F2$

The Boolean equation for either $F2$ or $\overline{F2}$ represents the truth table for the function $F2$. To write an implicit compact maxterm form, simply leave off the M following the product symbol. Example:

$F2(X,Y,Z) = \Pi(3,4,5)$ is an implicit compact maxterm form for the 0s of the function $F2$, and $\overline{F2}(X,Y,Z) = \Pi(0,1,2,6,7)$ is an implicit compact maxterm form for the 1s of the function $F2$.

Using the compact maxterm form for $F2$, we can write the CPOS form of the function F2 as follows:

$$F2(X,Y,Z) = M3\cdot M4\cdot M5$$
$$= (X + \overline{Y} + \overline{Z})\cdot(\overline{X} + Y + Z)\cdot(\overline{X} + Y + \overline{Z})$$

Using the compact maxterm form for $\overline{F2}$, we can write the CPOS form of the function $\overline{F2}$ as follows:

$$\overline{F2}(X,Y,Z) = M0\cdot M1\cdot M2\cdot M6\cdot M7$$
$$= (X + Y + Z)\cdot(X + Y + \overline{Z})\cdot(X + \overline{Y} + Z)\cdot(\overline{X} + \overline{Y} + Z)\cdot(\overline{X} + \overline{Y} + \overline{Z})$$

The canonical sum of products forms and canonical product of sums forms of functions are unique because there is only one way to write them. When working by hand to design logic circuits, CSOP and CPOS forms of functions are seldom used to design logic circuits because they are not minimized Boolean functions. Boolean functions should be minimized so that circuits can be implemented with a minimum gate count. Chapter 2 introduces techniques for minimizing Boolean functions.

In the next section, we introduce VHDL. The VHDL software has a built-in program that minimizes Boolean functions so the CSOP and CPOS forms of functions can be used without first minimizing them.

1.4 WRITING VHDL DESIGNS FOR SIMPLE GATE FUNCTIONS

As we briefly mentioned earlier, VHDL is an acronym for Very High Speed Integrated Circuit Hardware Description Language. VHDL is a language for designing digital circuits. Once the VHDL code for a design is written, it can be simulated by a software program to determine if the design works properly. In this section, we will only discuss how to write VHDL code for designs using simple Boolean functions and show the waveform simulation diagrams for the designs.

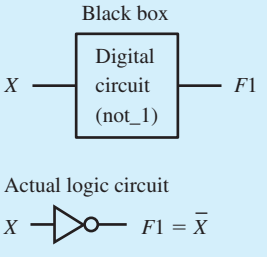
If you have access to a hardware board with a CPLD (complex programmable logic device) or a FPGA (field programmable gate array), you can download the required bits for the design and observe the circuit working on the hardware board. This can be done at home or in the laboratory as an experiment. Experiments are provided in Appendix A, and you are encouraged to use them.

We begin by presenting VHDL in the simplest form. The terms **entity** and **architecture** are the main sections of VHDL that you need to become familiar with. The entity contains the description for the inputs and outputs in a circuit just like the black box discussed earlier. The architecture contains the description of an actual logic circuit in terms of a Boolean function as discussed earlier.

1.4.1 VHDL Design for a NOT Function

Listing 1.1 shows a complete VHDL design for the Boolean function $F1 = \overline{X}$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity not_1 is port (
     X : in std_logic;
     F1 : out std_logic
     );
end not_1;

architecture Boolean_function of not_1 is
begin
     F1 <= not X;
end Boolean_function;
```



LISTING 1.1
Complete VHDL design for a NOT function and related graphics (Project: not_1).

Things you should notice about the VHDL design in Listing 1.1:

- A VHDL design consists of the three parts: **library part**, **entity declaration**, and **architecture declaration**.

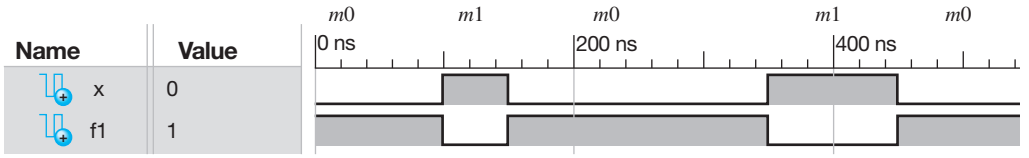
- The lines in the VHDL design that begin with **library** and **use** make up the **library part**. At this time, simply remember to include the library part in all your VHDL designs. We will discuss the library part in detail later in the book.
- The lines in the VHDL design that begin with **entity** and end with **end** entity_name (not_1 in this example) make up the **entity declaration**. The VHDL entity declaration represents the black box for the digital circuit that is shown to the right.
- The lines in the VHDL design that begin with **architecture** and end with **end** architecture_name (Boolean_function in this example) make up the **architecture declaration**. The VHDL architecture declaration represents the actual logic circuit that is shown to the right.
- The entity declaration and architecture declaration are textual descriptions in VHDL of the black box and the actual logic circuit, respectively.
- The syntax (the rules for writing a VHDL design) must be followed explicitly for the design to compile correctly. To learn how to write a VHDL design, we recommend that you copy the style and modify it as necessary to create new designs. With practice, you will soon be able to write your own VHDL design.
- For this design, the entity declaration has the label not_1. To end the entity declaration, you must type **end** not_1. The words in **bold type** are called keywords. Keywords are reserved words that cannot be used for signal names or labels. In the entity declaration, the keywords are **entity**, **is**, **port**, **in**, **out**, and **end**.
- The following is an alphabetical list of VHDL keywords for designing circuits:

  A **abs, all, alias, and, architecture, array, attribute**
  B **begin, block, body, buffer**
  C **case, component, configuration, constant**
  D **downto**
  E **else, elsif, end, entity, exit**
  F **for, function**
  G **generate, generic, group**
  I **if, in, inout, is**
  L **library, literal, loop**
  M **map, mod**
  N **nand, next, nor, not, null**
  O **of, or, others, out**
  P **package, port, procedure, process**
  R **range, record, rem, return, rol, ror**
  S **select, signal, sla, sll, sra, srl, subtype**
  T **then, to, type**
  U **until, use**
  V **variable**
  W **wait, when, while, with**
  X **xnor, xor**

- In the entity declaration, the external signal X is declared as an input via the keyword **in**, and the external signal $F1$ is declared as an output via the keyword **out**. Both signals X and $F1$ are declared as data type std_logic. We will discuss data types in more detail later in the book.
- In the architecture declaration, the Boolean expression **not** X is assigned to the signal $F1$ via the signal assignment symbol (<=); that is, $F1$ <= **not** X. The Boolean function $F1$ <= **not** X is placed in the architecture declaration between **begin** and **end** Boolean_function.

- Signal names (such as X and $F1$) and labels (such as not_1 and Boolean_function) have the following rules: (a) the first character must be a letter, (b) numbers may be included as well as the underscore character (_), (c) no adjacent underscore characters may be used, (d) an underscore character may not be used as the last character, and (e) spaces are not allowed. Signal names and labels are formally called **identifiers**.
- The label not_1 provides a meaningful description of the VHDL design. Observe that not_1 is used as the label for the entity declaration. The keyword "**not**" cannot be used for a label, because **not** is a keyword that represents the **NOT** operator in VHDL.
- VHDL is **not case sensitive**, which means that upper- or lowercase letters can be used for keywords, names and labels. VHDL also has a **free format**, which means that there is no formatting convention for spacing and indentations. We recommend that you use the formatting style that we use in our designs because our formatting style will become second nature to you as you read and study our VHDL designs.

Waveform 1.1 shows waveform diagrams for the VHDL design for the Boolean function $F1 = \overline{X}$.



WAVEFORM 1.1
Diagrams for the VHDL design for the Boolean function $F1 = \overline{X}$

Running a simulation for a VHDL design results in waveform diagrams like the ones shown in Waveform 1.1 and allows the designer to verify the correct functionality of a design. Observe that $F1$ is 1 when X is 0, and $F1$ is 0 when X is 1. This shows that the output waveform for $F1$ is the **complement** of the input waveform for X. Waveform 1.1 shows that the VHDL design in Listing 1.1 does, in fact, provide a correct design for the Boolean function $F1 = \overline{X}$. The values 0 and 1 that are listed in the second column of Waveform 1.1 are the values of X and $F1$, respectively, at the beginning of the simulation.

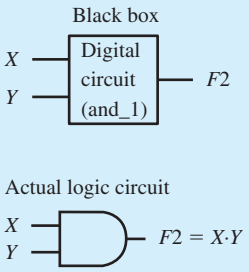
To identify the input, minterm designators were added to the waveform diagrams in Waveform 1.1. Because $F1(X) = \Sigma m(0) = \overline{X}$, you may observe in Waveform 1.1 that $F1 = 1$ for $m0$; else, $F1 = 0$.

NOTE: To learn how to use Xilinx ISE software to run your own VHDL design and simulation, see Experiments 1a in the Appendix A and Section B.2 in Appendix B.

1.4.2 VHDL Design for an AND Function

Listing 1.2 shows a complete VHDL design for the Boolean function $F2 = X{\cdot}Y$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_1 is port (
     x, y : in std_logic;
     f2 : out std_logic
     );
end and_1;

architecture Boolean_function of and_1 is
begin
     f2 <= x and y;
end Boolean_function;
```



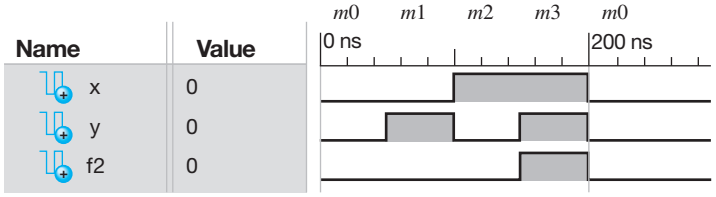
LISTING 1.2
Complete VHDL design for an AND function and related graphics (Project: and_1).

Things you should notice about the VHDL design in Listing 1.2:

- The VHDL entity declaration represents the black box for the digital circuit that is shown to the right.
- The VHDL architecture declaration represents the actual logic circuit that is shown to the right.
- Observe that the style for writing the VHDL design in Listing 1.2 is a modified copy of the VHDL design in Listing 1.1. When you first begin to write programs in a new programming language, it is a good habit to copy and modify existing code as we are doing.
- For this design, the entity declaration has the label and_1. To end the entity declaration, you must type **end** and_1. The label and_1 provides a meaningful description of the VHDL design. The keyword "**and**" cannot be used for a label, because **and** is a keyword that represents the **AND** operator in VHDL.
- In the entity declaration, the external signal *Y* is added to *X* as an input. Observe that you may use uppercase or lowercase because VHDL is case insensitive.
- In the entity declaration, the external signal *F*1 is changed to *F*2.
- Observe that the label and_1 must be used in the architecture declaration because it is the label that is used in entity declaration.
- In the architecture declaration, the Boolean expression *X* **and** *Y* is assigned to the signal *F*2 via the signal assignment symbol (<=); that is, *F*2 <= *X* **and** *Y*. The Boolean function *F*2 <= *X* **and** *Y* is placed in the architecture declaration between **begin** and **end** Boolean_function.

Waveform 1.2 shows waveform diagrams for the VHDL design for the Boolean function $F2 = X{\cdot}Y$.

WAVEFORM 1.2 Waveform diagrams for the VHDL design for the Boolean function $F2 = X{\cdot}Y$



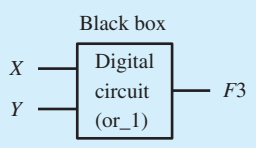
The waveform diagrams in Waveform 1.2 show the actual simulation of the VHDL design in Listing 1.2. Observe that *F*2 is only 1 when both *X* and *Y* are 1, and *F*2 is 0 for all other conditions of *X* and *Y*. This shows that the output waveform for *F*2 is the **AND**ing of the input waveforms for *X* and *Y*. Waveform 1.2 shows that the VHDL design in Listing 1.2 does, in fact, provide a correct design for the Boolean function $F2 = X{\cdot}Y$. To identify the inputs, minterm designators were added to the waveform diagrams in Waveform 1.2. Because $F2(X,Y) = \Sigma m(3)$, you may observe in Waveform 1.2 that $F2 = 1$ for $m3$; else, $F2 = 0$.

1.4.3 VHDL Design for an OR Function

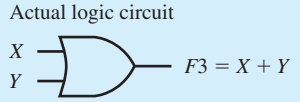
Listing 1.3 shows a complete VHDL design for the Boolean function $F3 = X + Y$.

LISTING 1.3 Complete VHDL design for an OR function and related graphics (Project: or_1)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity or_1 is port (
     x, y : in std_logic;
     f3 : out std_logic
     );
end or_1;

architecture Boolean_function of or_1 is
begin
     f3 <= x or y;
end Boolean_function;
```



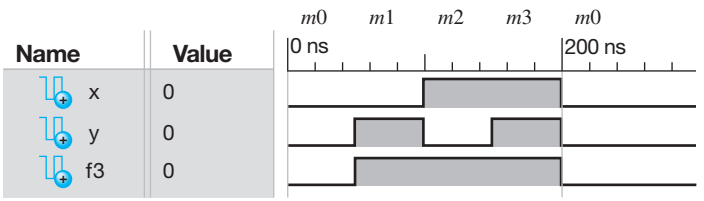
Actual logic circuit



Things you should notice about the VHDL design in Listing 1.3:

- Observe that the style for writing the VHDL design in Listing 1.3 is a modified copy of the VHDL design in Listing 1.2.
- For this design, the entity declaration has the label or_1. To end the entity declaration, you must type **end** or_1. The label or_1 provides a meaningful description of the VHDL design. The keyword "**or**" cannot be used for a label, because **or** is a keyword that represents the **OR** operator in VHDL.
- In the entity declaration, the external signal *F*2 is changed to *F*3.
- Observe that the label or_1 must be used in the architecture declaration because it is the label that is used in entity declaration.
- In the architecture declaration, the Boolean expression *X* **or** *Y* is assigned to the signal *F*3 via the signal assignment symbol (<=); that is, *F*3 <= *X* **or** *Y*. The Boolean function *F*3 <= *X* **or** *Y* is placed in the architecture declaration between **begin** and **end** Boolean_function.

Waveform 1.3 shows waveform diagrams for the VHDL design for the Boolean function $F3 = X + Y$.



WAVEFORM 1.3 Waveform diagrams for the VHDL design for the Boolean function $F3 = X + Y$

The waveform diagrams in Waveform 1.3 show the actual simulation of the VHDL design in Listing 1.3. Observe that *F*3 is 1 any time *X* is 1 or *Y* is 1, and *F*3 is 0 when both *X* and *Y* are 0. This shows that the output waveform for *F*3 is the **OR**ing of the input waveforms for *X* and *Y*. Waveform 1.3 shows that the VHDL design in Listing 1.3 does, in fact, provide a correct design for the Boolean function $F3 = X + Y$. To identify the inputs, minterm designators were added to the waveform diagrams in Waveform 1.3. Because $F3(X,Y) = \Sigma m(1,2,3)$, you may observe in Waveform 1.3 that $F3 = 1$ for *m*1, *m*2, and *m*3; else, $F3 = 0$.

1.4.4 VHDL Design for an XOR Function

Figure 1.9 shows a logic symbol summary for the XOR operation.

Boolean algebra expression	Name of expression	Distinctive-shape logic symbol	Logic name
$A \oplus B \oplus C \oplus \ldots$	XOR operation	A, B, C	$A \oplus B \oplus C \oplus \ldots$ XOR gate

FIGURE 1.9 Logic symbol summary for the XOR operation

Notice in Figure 1.9 that an XOR gate can have two or more inputs. For three input signals, $A \oplus B \oplus C \oplus$ is equivalent to *A* **XOR** *B* **XOR** *C* in VHDL.

The XOR function is an **odd function**, which has a value of 1 when the combination of all the inputs have an odd number of 1s (1, 3, 5, . . .). This is shown in Table 1.9, where F_{XOR} and F_{ODD} provide the same truth table values for the simplest case of just two inputs. Observe that 01 and 10 represent an odd number of 1s—that is, one 1. Using perfect induction it can be shown $F_{XOR} = F_{ODD}$ for more than two inputs.

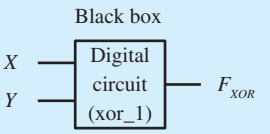
TABLE 1.9 Truth table for an XOR function and an ODD function with two inputs

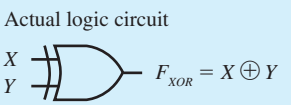
XOR function (ODD function)			
X	Y	F_{XOR}	F_{ODD}
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Listing 1.4 shows a complete VHDL design for the Boolean function $F_{XOR} = X \oplus Y$.

LISTING 1.4
Complete VHDL design for an XOR function and related graphics (Project: xor_1)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity xor_1 is port (
     x, y : in std_logic;
     fxor : out std_logic
     );
end xor_1;

architecture Boolean_function of xor_1 is
begin
     fxor <= x xor y;
--   fxor <= (not x and y) or (x and not y);
end Boolean_function;
```

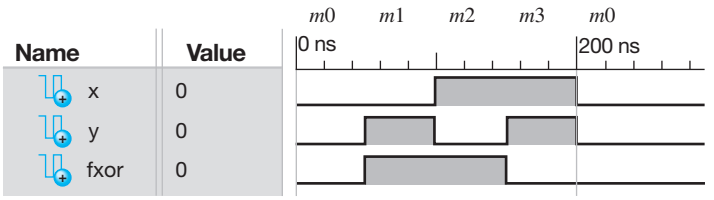




Things you should notice about the VHDL design in Listing 1.4:

- Observe that the style for writing the VHDL design in Listing 1.4 is a modified copy of the VHDL design in Listing 1.3.
- For this design, the entity declaration has the label xor_1. To end the entity declaration, you must type **end** xor_1. The label xor_1 provides a meaningful description of the VHDL design. The keyword "**xor**" cannot be used for a label, because **xor** is a keyword that represents the **XOR** operator in VHDL.
- In the entity declaration, the external signal *F*3 is changed to *Fxor*.
- Observe that the label xor_1 must be used in the architecture declaration because it is the label that is used in entity declaration.
- In the architecture declaration, the Boolean expression *X* **xor** *Y* is assigned to the signal *Fxor* via the signal assignment symbol (<=)—that is, *Fxor* <= *X* **xor** *Y*. The Boolean function is *Fxor* <= *X* **xor** *Y* placed in the architecture declaration between **begin** and **end** Boolean_function.
- Notice in listing 1.4 a comment may be placed in VHDL code by using two hyphens in series, that is, "--". The Boolean function *Fxor* <= (**not** *X* **and** *Y*) **or** (*X* **and not** *Y*) following the comment symbol can be used in place of the Boolean function *Fxor* <= *X* **xor** *Y*.
- The **AND** and **OR** binary operators have the same precedence in VHDL. This means that parentheses must be used to select the order of precedence of the binary operators in VHDL. No parentheses are required around the **NOT** operator, because it has a higher priority than any of the binary operators in VHDL.

Waveform 1.4 shows waveform diagrams for the VHDL design for the Boolean function $F_{XOR} = X \oplus Y$.



WAVEFORM 1.4 Waveform diagrams for the VHDL design for the Boolean function $F_{XOR} = X \oplus Y$

The waveform diagrams in Waveform 1.4 show the actual simulation of the VHDL design in Listing 1.4. Observe that F_{XOR} is 1 any time X has the opposite value of Y, and F_{XOR} is 0 when X has the same value of Y. This shows that the output waveform for F_{XOR} is the **XOR**ing of the input waveforms for X and Y. Waveform 1.4 shows that the VHDL design in Listing 1.4 does, in fact, provide a correct design for the Boolean function $F_{XOR} = X \oplus Y$. To identify the inputs, minterm designators were added to the waveform diagrams in Waveform 1.4. Because $F_{XOR}(X,Y) = \Sigma m(1,2)$ you may observe in Waveform 1.4 that $F_{XOR} = 1$ for $m1$ and $m2$; else, $F_{XOR} = 0$.

1.4.5 VHDL Design for a NAND Function

Table 1.10 shows a truth table for the **NAND** function F_{NAND}. When the input signal values are both 1 (for two inputs), or all 1s (if more than two inputs), the function F_{NAND} is 0; otherwise, the function is 1.

TABLE 1.10 Truth table for a NAND function with two inputs

NAND function		
X	Y	F_{NAND}
0	0	1
0	1	1
1	0	1
1	1	0

Using the 0s of the function, we can write the Boolean function as $\overline{F_{NAND}} = X \cdot Y$, so $F_{NAND} = \overline{X \cdot Y}$.

Figure 1.10 shows a logic symbol summary for the NAND operation.

Boolean algebra expression	Name of expression	Distinctive-shape logic symbol	Logic name
$\overline{A \cdot B \cdot C \cdot \ldots}$	NAND operation	A, B, C → $\overline{A \cdot B \cdot C \cdot \ldots}$	NAND gate

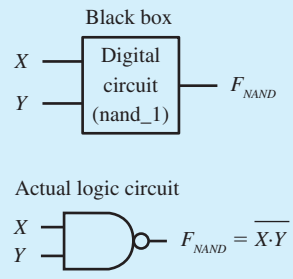
FIGURE 1.10 Logic symbol summary for the NAND operation

Notice in Figure 1.10 that a NAND gate can have two or more inputs. For three input signals, $\overline{X \cdot Y \cdot Z} = \overline{X} + \overline{Y} + \overline{Z}$ by DeMorgan's Theorem T6a and is equivalent to **NOT** X **OR NOT** Y **OR NOT** Z in VHDL. For three input signals, $\overline{X \cdot Y \cdot Z}$ is not equivalent to X **NAND** Y **NAND** Z. The expression X **NAND** Y **NAND** Z is an illegal expression in VHDL. In VHDL, **NOT** (X **AND** Y **AND** Z) is equivalent to the expression $\overline{X \cdot Y \cdot Z}$.

Listing 1.5 shows a complete VHDL design for the Boolean function $F_{NAND} = \overline{X \cdot Y}$.

LISTING 1.5
Complete VHDL design for a NAND function and related graphics (Project: nand_1)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity nand_1 is port (
     x, y : in std_logic;
     fnand : out std_logic
     );
end nand_1;

architecture Boolean_function of nand_1 is
begin
     fnand <= x nand y;
end Boolean_function;
```

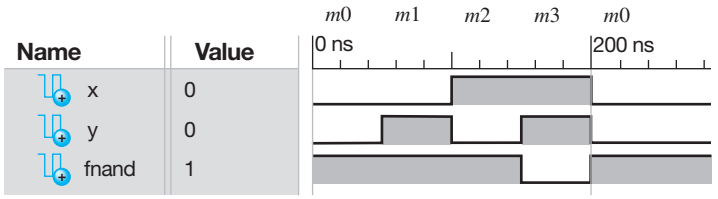


Things you should notice about the VHDL design in Listing 1.5:

- Observe that the style for writing the VHDL design in Listing 1.5 is a modified copy of the VHDL design in Listing 1.4.
- For this design, the entity declaration has the label nand_1. To end the entity declaration, you must type **end** nand_1. The label nand_1 provides a meaningful description of the VHDL design. The keyword "**nand**" cannot be used for a label, because **nand** is a keyword that represents the **NAND** operator in VHDL.
- In the entity declaration, the external signal *Fxor* is changed to *Fnand*.
- Observe that the label nand_1 must be used in the architecture declaration because it is the label that is used in entity declaration.
- In the architecture declaration, the Boolean expression *X* **nand** *Y* is assigned to the signal *Fnand* via the signal assignment symbol (<=)—that is, *Fnand* <= *X* **nand** *Y*. The Boolean function *Fnand* <= *X* **nand** *Y* is placed in the architecture declaration between **begin** and **end** Boolean_function.

Waveform 1.5 shows waveform diagrams for the VHDL design for the Boolean function $F_{NAND} = \overline{X \cdot Y}$.

WAVEFORM 1.5 Waveform diagrams for the VHDL design for the Boolean function $F_{NAND} = \overline{X \cdot Y}$



The waveform diagrams in Waveform 1.5 show the actual simulation of the VHDL design in Listing 1.5. Observe that F_{NAND} is 0 when both *X* and *Y* are 1, and F_{NAND} is 1 for all other conditions of *X* and *Y*. This shows that the output waveform for F_{NAND} is the **NAND**ing of the input waveforms for *X* and *Y*. Waveform 1.5 shows that the VHDL design in Listing 1.5 does, in fact, provide a correct design for the Boolean function $F_{NAND} = \overline{X \cdot Y}$. To identify the inputs, minterm designators were added to the waveform diagrams in Waveform 1.5. Because $F_{NAND}(X,Y) = \Sigma m(0,1,2)$ you may observe in Waveform 1.5 that $F_{NAND} = 1$ for *m*0, *m*1, and *m*2; else, $F_{NAND} = 0$.

1.4.6 VHDL Design for a NOR Function

Table 1.11 shows a truth table for the **NOR** function F_{NOR}. When the input signal values are both 0 (for two inputs), or all 0s (if more than two inputs), the function F_{NOR} is 1; otherwise, the function is 0.

NOR function		
X	Y	F_{NOR}
0	0	1
0	1	0
1	0	0
1	1	0

TABLE 1.11 Truth table for a NOR function with two inputs

Using the 1s of the function, we can write the Boolean function as $F_{NOR} = \overline{X} \cdot \overline{Y}$, but $\overline{X} \cdot \overline{Y} = \overline{X + Y}$ by DeMorgan's Theorem T5b, so $F_{NOR} = \overline{X + Y}$. Figure 1.11 shows a logic symbol summary for the NOR operation.

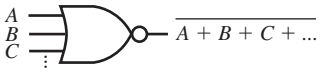
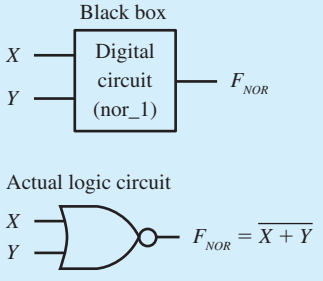
Boolean algebra expression	Name of expression	Distinctive-shape logic symbol	Logic name
$\overline{A + B + C + ...}$	NOR operation		NOR gate

FIGURE 1.11 Logic symbol summary for the NOR operation

Notice in Figure 1.11 that a NOR gate can have two or more inputs. For three input signals, $\overline{X + Y + Z} = \overline{X} \cdot \overline{Y} \cdot \overline{Z}$ by DeMorgan's Theorem T6b and is equivalent to **NOT** X **AND NOT** Y **AND NOT** Z in VHDL. For three input signals, $\overline{X + Y + Z}$ is not equivalent to X **NOR** Y **NOR** Z. The expression X **NOR** Y **NOR** Z is an illegal expression in VHDL. In VHDL, **NOT** (X **OR** Y **OR** Z) is equivalent to the expression $\overline{X + Y + Z}$.

Listing 1.6 shows a complete VHDL design for the Boolean function $F_{NOR} = \overline{X + Y}$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity nor_1 is port (
     x, y : in std_logic;
     fnor : out std_logic
     );
end nor_1;

architecture Boolean_function of nor_1 is
begin
     fnor <= x nor y;
end Boolean_function;
```



LISTING 1.6 Complete VHDL design for a NOR function and related graphics (Project: nor_1)

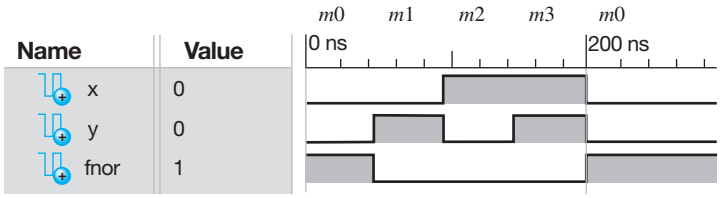
Things you should notice about the VHDL design in Listing 1.6:

- Observe that the style for writing the VHDL design in Listing 1.6 is a modified copy of the VHDL design in Listing 1.5.
- For this design, the entity declaration has the label nor_1. To end the entity declaration, you must type **end** nor_1. The label nor_1 provides a meaningful description of the VHDL design. The keyword "**nor**" cannot be used for a label, because **nor** is a keyword that represents the **NOR** operator in VHDL.
- In the entity declaration, the external signal *Fnand* is changed to *Fnor*.
- Observe that the label nor_1 must be used in the architecture declaration because it is the label that is used in entity declaration.

- In the architecture declaration, the Boolean expression *X* **nor** *Y* is assigned to the signal *Fnor* via the signal assignment symbol (<=)—that is, *Fnor* <= *X* **nor** *Y*. The Boolean function *Fnor* <= *X* **nor** *Y* is placed in the architecture declaration between **begin** and **end** Boolean_function.

Waveform 1.6 shows waveform diagrams for the VHDL design for the Boolean function $F_{NOR} = \overline{X + Y}$.

WAVEFORM 1.6 Waveform diagrams for the VHDL design for the Boolean function $F_{NOR} = \overline{X + Y}$



The waveform diagrams in Waveform 1.6 show the actual simulation of the VHDL design in Listing 1.6. Observe that F_{NOR} is 1 when both *X* and *Y* are 0, and F_{NOR} is 0 for all other conditions of *X* and *Y*. This shows that the output waveform for F_{NOR} is the **NOR**ing of the input waveforms for *X* and *Y*. Waveform 1.6 shows that the VHDL design in Listing 1.6 does, in fact, provide a correct design for the Boolean function $F_{NOR} = \overline{X + Y}$. To identify the inputs, minterm designators were added to the waveform diagrams in Waveform 1.6. Because $F_{NOR}(X,Y) = \Sigma m(0)$ you may observe in Waveform 1.6 that $F_{NOR} = 1$ for $m0$; else, $F_{NOR} = 0$.

1.4.7 VHDL Design for an XNOR Function

Table 1.12 shows a truth table for the **XNOR** function F_{XNOR}. When the input signal values are the same (both are 0, or both are 1), the function F_{XNOR} is 1; otherwise, the function is 0.

TABLE 1.12 Truth table for a XNOR function with two inputs

XNOR function		
X	*Y*	F_{XNOR}
0	0	1
0	1	0
1	0	0
1	1	1

Using the 1s of the function, we can write the Boolean function as $F_{XNOR} = \overline{X} \cdot \overline{Y} + X \cdot Y$. An alternate way of writing the XNOR function is $F_{XNOR} = \overline{X \oplus Y}$, which is the complement of the XOR function. The XNOR function with two inputs is often referred to as a **comparator** because the XNOR function is a 1 when both input signals are the same value and is a 0 when they are different values.

Figure 1.12 shows a logic symbol summary for the XNOR operation.

FIGURE 1.12 Logic symbol summary for the XNOR operation

Boolean algebra expression	**Name of expression**	**Distinctive-shape logic symbol**	**Logic name**
$\overline{A \oplus B \oplus C \oplus \ldots}$	XNOR operation	A, B, C $\rightarrow$ $\overline{A \oplus B \oplus C \oplus \ldots}$	XNOR gate

Notice in Figure 1.12 that an XNOR gate can have two or more inputs. For three input signals, $\overline{A \oplus B \oplus C}$ is equivalent to **NOT**(*A* **XOR** *B* **XOR** *C*) but not equivalent to *A* **XNOR** *B* **XNOR** *C* in VHDL.

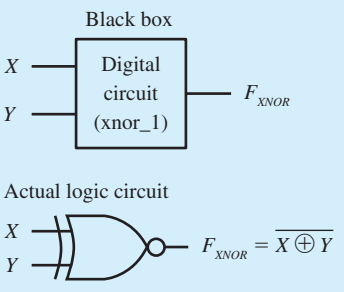
The XNOR function is an **even function**, which has a value of 1 when the combination of all the inputs has an even number of 1s (0, 2, 4, . . .). This is shown in Table 1.13, where F_{XNOR} and F_{EVEN} provide the same truth table values for the simplest case of just two inputs. Observe that 00 represents an even number of 1—that is, no 1s—and 11 represent an even number of 1s—that is, two 1s. Using perfect induction, it can be shown that $F_{XNOR} = F_{EVEN}$ for more than two inputs.

TABLE 1.13 Truth table for an XNOR function and an EVEN function with two inputs

XNOR function (**EVEN** function)			
X	*Y*	F_{XNOR}	F_{EVEN}
0	0	1	1
0	1	0	0
1	0	0	0
1	1	1	1

Listing 1.7 shows a complete VHDL design for the Boolean function $F_{XNOR} = \overline{X \oplus Y}$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity xnor_1 is port (
     x, y : in std_logic;
     fxnor : out std_logic
     );
end xnor_1;

architecture Boolean_function of xnor_1 is
begin
     fxnor <= x xnor y;
--   fxnor <= (not x and not y) or (x and y);
end Boolean_function;
```



LISTING 1.7
Complete VHDL design for an XNOR function and related graphics (Project: xnor_1)

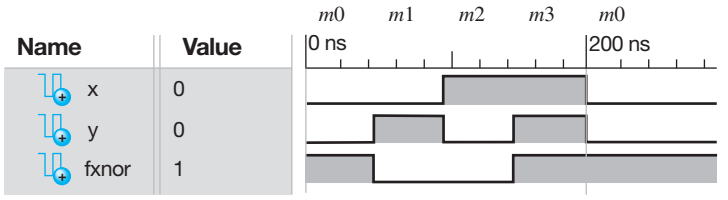
Things you should notice about the VHDL design in Listing 1.7:

- Observe that the style for writing the VHDL design in Listing 1.7 is a modified copy of the VHDL design in Listing 1.6.
- For this design, the entity declaration has the label xnor_1. To end the entity declaration, you must type **end** xnor_1. The label xnor_1 provides a meaningful description of the VHDL design. The keyword "**xnor**" cannot be used for a label, because **xnor** is a keyword that represents the **XNOR** operator in VHDL.
- In the entity declaration, the external signal *Fnor* is changed to *Fxnor.*
- Observe that the label xnor_1 must be used in the architecture declaration because it is the label that is used in entity declaration.
- In the architecture declaration, the Boolean expression *X* **xnor** *Y* is assigned to the signal *Fxnor* via the signal assignment symbol (<=)—that is, *Fxnor* <= *X* **xnor** *Y*. The Boolean function is placed in the architecture declaration between **begin** and **end** Boolean_function.

- Observe that the Boolean function *Fxnor* <= (**not** *X* **and not** *Y*) **or** (*X* **and** *Y*) may be used in place of the Boolean function *Fxnor* <= *X* **xnor** *Y*. Recall that any text that follows two hyphens in series, that is, "--", can be used as a comment in VHDL.

Waveform 1.7 shows waveform diagrams for the VHDL design for the Boolean function $F_{XNOR} = \overline{X \oplus Y}$.

WAVEFORM 1.7 Waveform diagrams for the VHDL design for the Boolean function $F_{XNOR} = \overline{X \oplus Y}$



The waveform diagrams in Waveform 1.7 show the actual simulation of the VHDL design in Listing 1.7. Observe that F_{XNOR} is 1 when both *X* and *Y* are 0 or when both *X* and *Y* are 1—that is, both inputs are the same value—and F_{XNOR} is 0 for all other conditions of *X* and *Y*—that is, when the inputs are different values. This shows that the output waveform for F_{XNOR} is the **XNOR**ing of the input waveforms for *X* and *Y*. Waveform 1.7 shows that the VHDL design in Listing 1.7 does, in fact, provide a correct design for the Boolean function $F_{XNOR} = \overline{X \oplus Y}$. To identify the inputs, minterm designators were added to the waveform diagrams in Waveform 1.7. Because $F_{XNOR}(X,Y) = \Sigma m(0,3)$ you may observe in Waveform 1.7 that $F_{XNOR} = 1$ for *m*0 and *m*3; else, $F_{XNOR} = 0$.

1.4.8 VHDL Design for a BUFFER Function

Table 1.14 shows a truth table for **BUFFER** function F_{BUF}. The **BUFFER** function seems like a do nothing function, because its output is logically equivalent to its input.

TABLE 1.14 Truth table for a BUFFER function

BUFFER function	
X	F_{BUF}
0	0
1	1

Using the 1s of the function, we can write the Boolean function $F_{BUF} = X$. Figure 1.13 shows a logic symbol summary for the BUFFER operation.

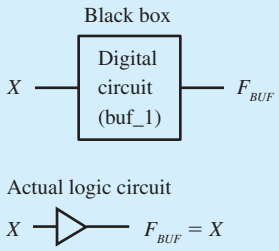
FIGURE 1.13 Logic symbol summary for the BUFFER operation

Boolean algebra expression	Name of expression	Distinctive-shape logic symbol	Logic symbol name
A	BUFFER operation	*A* ▷ $F_{BUF} = A$	BUFFER

The distinctive-shape symbol shown in Figure 1.13 has an **amplifier** symbol ▷ included. The amplifier symbol need not be included unless the output has amplification or additional "oomph" so that the output can drive a larger number of gates. A BUFFER can also be used in a design for the purpose of documentation so that different signal names can be used on the input and output of the BUFFER.

Listing 1.8 shows a complete VHDL design for the Boolean function $F_{BUF} = X$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity buf_1 is port (
     x : in std_logic;
     fbuf : out std_logic
     );
end buf_1;

architecture Boolean_function of buf_1 is
begin
     fbuf <= x;
end Boolean_function;
```

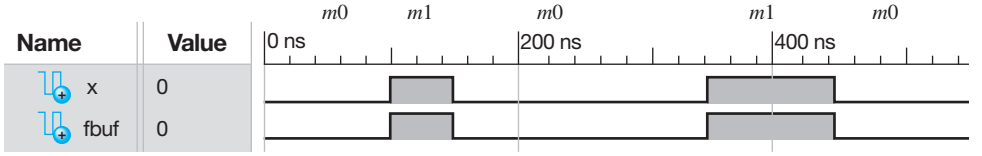


LISTING 1.8
Complete VHDL design for a BUFFER function and related graphics (Project: buf_1)

Things you should notice about the VHDL design in Listing 1.8:

- Observe that the style for writing the VHDL design in Listing 1.8 is a modified copy of the VHDL design in Listing 1.7.
- For this design, the entity declaration has the label buf_1. To end the entity declaration, you must type **end** buf_1. The label buf_1 provides a meaningful description of the VHDL design. The keyword "**buffer**" cannot be used for a label, because **buffer** is a keyword that represents the mode inout in VHDL.
- In the entity declaration, the external signal *Y* is removed and *Fxnor* is changed to *Fbuf*.
- Observe that the label buf_1 must be used in the architecture declaration because it is the label that is used in entity declaration.
- In the architecture declaration, the Boolean expression *X* is assigned to the signal *Fbuf* via the signal assignment symbol (<=)—that is, *Fbuf* <= *X*. The Boolean function is placed in the architecture declaration between **begin** and **end** Boolean_function.

Waveform 1.8 shows waveform diagrams for the VHDL design for the Boolean function $F_{BUF} = X$.



WAVEFORM 1.8
Waveform diagrams for the VHDL design for the Boolean function $F_{BUF} = X$

The waveform diagrams in Waveform 1.8 show the actual simulation of the VHDL design in Listing 1.8. Observe that F_{BUF} is 0 when *X* is 0, and F_{BUF} is 1 when *X* is 1. This shows that the output waveform for F_{BUF} is the **BUFFER**ing of the input waveform for *X*. Waveform 1.8 shows that the VHDL design in Listing 1.8 does, in fact, provide a correct design for the Boolean function $F_{BUF} = X$. To identify the inputs, minterm designators were added to the waveform diagrams in Waveform 1.8.

Because $F_{BUF}(X) = \Sigma m(1)$ you may observe in Waveform 1.8 that $F_{BUF} = 1$ for $m1$; else, $F_{BUF} = 0$.

1.4.9 VHDL Design for any Boolean Function Written in Canonical Form

A VHDL design can be obtained for any Boolean function written in canonical form. First, obtain a truth table for the function or a compact minterm or maxterm form of the function.

Next, write the function in a canonical form—that is, either in a canonical SOP (CSOP) form or a canonical POS (CPOS) form. Then, write the assignment statement for the Boolean function in VHDL.

To keep things simple, we will only use compact minterm forms and write CSOP forms for functions. If you prefer, you can use compact maxterm forms and write CPOS forms for functions.

For example, consider the truth table for a Two-1s function; that is, output F is 1 only when there are two 1s in each input combination $A\ B\ C$, as shown in Table 1.15.

TABLE 1.15 Truth table for a Two-1s function *F*

(Decimal)	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

We can write an assignment statement in VHDL for the function F in many different ways. First, write a compact minterm forms for the function using its 1s as:

$$F(A,B,C) = \Sigma m(3, 5, 6)$$

Then write the canonical SOP (CSOP) form as:

$$F(A,B,C) = \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + A\cdot B\cdot\overline{C}$$

The assignment statement for the 1s of the function F can now be written in CSOP form as:

F <= (**not** A **and** B **and** C) **or**
(A **and not** B **and** C) **or**
(A **and** B **and not** C)

Another way to write an assignment statement for the function F is to write a compact minterm form for the function using its 0s as:

$$\overline{F}(\text{A,B,C}) = \Sigma m(0,1,2,4,7)$$

The canonical SOP (CSOP) form is written as:

$$\overline{F}(A,B,C) = \overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C} + A\cdot\overline{B}\cdot\overline{C} + A\cdot B\cdot C$$

The assignment statement for the 0s of the function F can now be written in CSOP form as:

F <= **not** ((**not** A **and not** B **and not** C) **or**
(**not** A **and not** B **and** C) **or**
(**not** A **and** B **and not** C) **or**
(A **and not** B **and not** C) **or**
(A **and** B **and** C))

When writing an assignment statement for a function in VHDL, use the fewest number of 1s or 0s in the truth table for the function. The function written with the fewest number of 1s or 0s has fewer minterms and is easier to type. If the function has the same number of 1s and 0s, then choose the 1s, for less typing.

Listing 1.9 shows a complete VHDL design for the Two-1s function *F* in Table 1.15.

LISTING 1.9
Complete VHDL design for the Two-1s function *F* in Table 1.15 (Project: canonical_form)

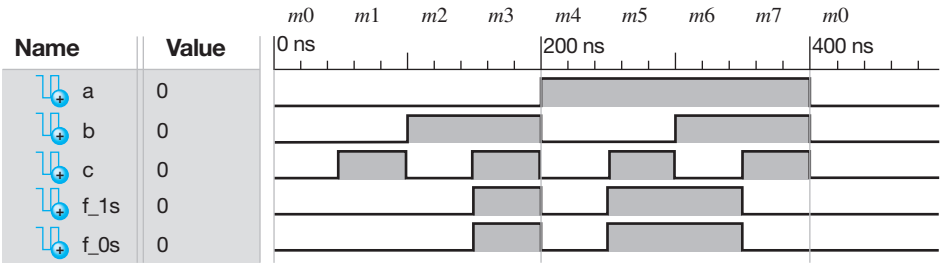
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity two_1s is port (
     a,b,c : in std_logic;
     f_1s,f_0s : out std_logic
     );
end two_1s;

architecture Boolean_function of two_1s is
begin

--the canonical SOP form for the 1s for the function f
     f_1s <= (not A and B and C) or
             (A and not B and C) or
             (A and B and not C);

--the canonical SOP form for the 0s for the function f
     f_0s <= not ((not A and not B and not C) or
                  (not A and not B and C) or
                  (not A and B and not C) or
                  (A and not B and not C) or
                  (A and B and C));

end Boolean_function;
```

Things you should notice about the VHDL design in Listing 1.9:

- Remember that VHDL code is not case sensitive—that is, upper- and lowercase names are interchangeable.
- Observe that the style for writing the VHDL design in Listing 1.9 is a modified copy of the VHDL design in Listing 1.7.
- For this design, the entity declaration has the label two_1s. To end the entity declaration, you must type **end** two_1s. The label two_1s provides a meaningful description of the VHDL design.
- In the entity declaration, the external input signals *X* and *Y* are changed to *A*, *B*, and *C*.
- In the entity declaration, the external output signal F_{XNOR} is changed to *F*_1*s*, *F*_0*s*.
- Observe that the label two_1s must be used in the architecture declaration because it is the label that is used in entity declaration.
- In the architecture declaration, the Boolean functions for the 1s of the function labeled (*F*_1*s*) and the Boolean function for the 0s of the function labeled (*F*_0*s*) are placed in the architecture declaration between **begin** and **end** Boolean_function.
- Observe the comments. Either function *F*_1*s* or function *F*_0*s* may be used to represent the Boolean function *F* in Table 1.15.
- The purpose of including both function *F*_1*s* and function *F*_0*s* is to verify that both functions generate the same output when we run a simulation. So, $F = F\_1s = F\_0s$ should be true.

Waveform 1.9 shows waveform diagrams for the VHDL design for the Two-1s function *F* in Table 1.15.

WAVEFORM 1.9
Waveform diagrams for the VHDL design for the Two-1s function *F* in Table 1.15



The waveform diagrams in Waveform 1.9 show the actual simulation of the VHDL design in Listing 1.9. Observe that both function *F_1s* and function *F_0s* in Waveform 1.9 provide the same output as function *F* in Table 1.15. This shows that the VHDL design in Listing 1.9 does, in fact, provide a correct design for Boolean function *F*—that is, $F = F\_1s$ or $F = F\_0s$ can be used to generate the function *F*. To identify the inputs, minterm designators were added to the waveform diagrams in Waveform 1.9. Because $F(A,B,C) = \Sigma m(3,5,6)$ you may observe in Waveform 1.9 that f_1*s* = 1 for *m*3, *m*5, and *m*6, else f_1*s* = 0. Because $\overline{F}(A,B,C) = \Sigma m(0,1,2,4,7)$, you may observe in Waveform 1.9 that f_0*s* = 0 for *m*0, *m*1, *m*2, *m*4, and *m*7; else, f_0*s* = 1.

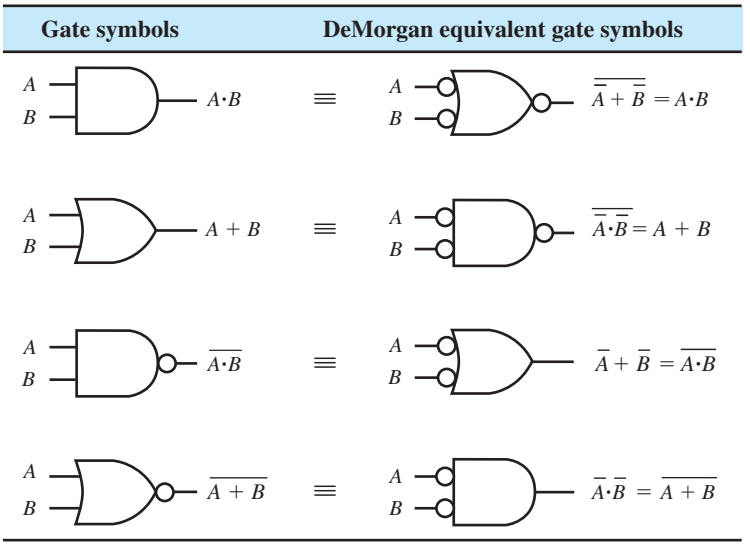
1.5 MORE ABOUT LOGIC GATES

In this section we summarize some additional information about gates.

1.5.1 Equivalent Gate Symbols

Logic gate symbols can be represented by different equivalent gate symbols called **DeMorgan equivalent gate symbols**. Figure 1.14 shows an equivalent gate symbols for the AND gate, OR gate, NAND gate, and NOR gate.

FIGURE 1.14 Equivalent logic gate symbols for the AND gate, OR gate, NAND gate, and NOR gate



The DeMorgan equivalent gate symbols are shown so that you may recognize that there is more than one way to represent a gate symbol and still provide the same output operation. An easy way to remember the equivalent gate symbols is to recognize that there is an AND form (the distinctive shape is the AND shape) and an OR form (the distinctive shape is the OR shape) for each logic gate symbol in Figure 1.14. To convert to the equivalent form, simply change to the

opposite distinctive shape, and at each input and output, simply remove a bubble if one is present or add a bubble if no bubble is present. You can see that this process works for each of the gates in Figure 1.14 from left to right or from right to left. **Memory Jogger**: To obtain a DeMorgan equivalent gate symbol for an AND, OR, NAND, or NOR gate, simply add bubbles to all inputs and outputs and change an AND symbol to an OR symbol or change an OR symbol to an AND symbol. Remember that two bubbles result in no bubble via Double Negation Theorem T7.

In Figure 1.14, notice that the Boolean output expressions for the DeMorgan equivalent gate symbols are obtained using DeMorgan's Theorems either directly or by complementing them. The output for the DeMorgan equivalent gate symbol for the AND gate is $\overline{\overline{A} + \overline{B}} = A \cdot B$. The output for the DeMorgan equivalent gate symbol for the OR gate is $\overline{\overline{A} \cdot \overline{B}} = A + B$. The output for the DeMorgan equivalent gate symbol for the NAND gate is $\overline{A} + \overline{B} = \overline{A \cdot B}$. The output for the DeMorgan equivalent gate symbol for the NOR gate is $\overline{A} \cdot \overline{B} = \overline{A + B}$.

1.5.2 Functionally Complete Gates

Using the three gates NOT, AND, and OR, we can design any logic circuit. The NAND gate is referred to as a functionally complete gate, because a set of just NAND gates can be used to design any logic circuit. Figure 1.15 shows NAND gate equivalent circuits for the NOT, AND, and OR gates.

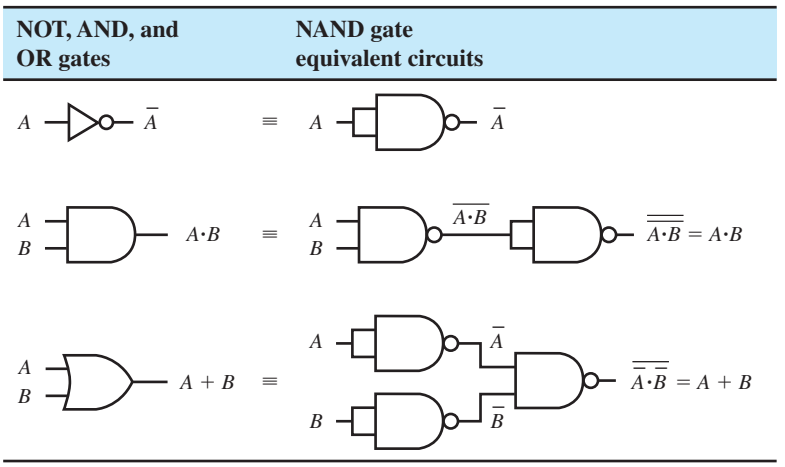


FIGURE 1.15 NAND gate equivalent circuits for the NOT, AND, and OR gates

The NOR gate is also a functionally complete gate, because a set of just NOR gates can be used to design any logic circuit. Figure 1.16 shows NOR gate equivalent circuits for the NOT, AND, and OR gates.

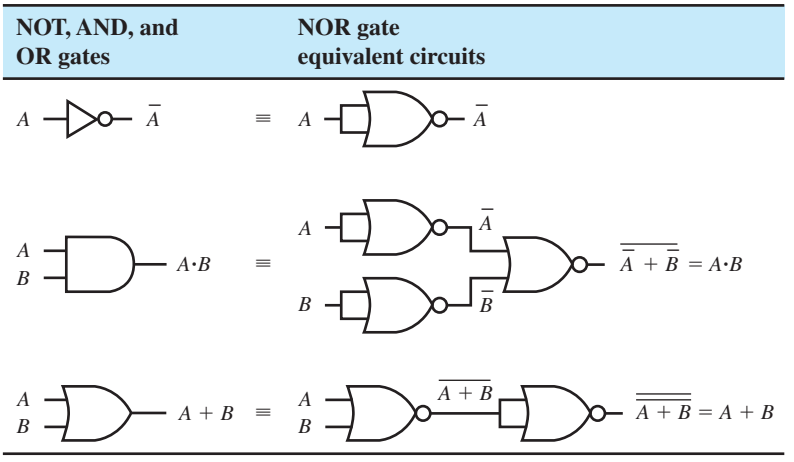


FIGURE 1.16 NOR gate equivalent circuits for the NOT, AND, and OR gates

1.5.3 Equivalent Gate Circuits

Logic gates that use bubbles at their inputs or outputs can be drawn in different but equivalent ways. Figure 1.17 shows a summary of several logic gates and their equivalent gate circuits.

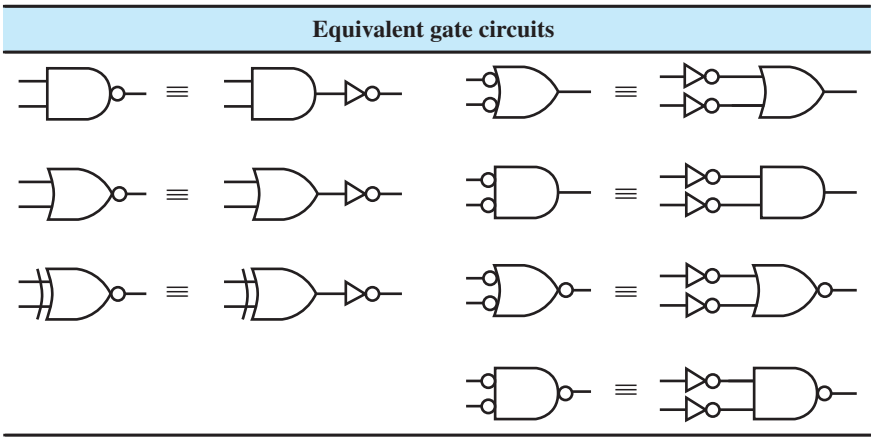


FIGURE 1.17 Equivalent gate circuits

In Figure 1.17, when a bubble is attached to a distinctive-shape gate symbol, the circuit denotes a built-in (or internal) NOT gate. When a bubble is attached to a triangle, the circuit denotes a separate (or external) NOT gate.

1.5.4 Compact Description Names for Gates

A graphic package is normally used to draw a circuit diagram on a computer. Rather than showing the logic symbols for the available gates in a graphic package, a list of compact description names is used. You must learn the compact description names to move the correct logic symbols to the screen. Figure 1.18 shows a summary of several logic symbols and their associated compact description names.

The notation AND2B1 stands for a 2 input AND gate with one bubble input, while the notation OR3B2 stands for a 3 input OR gate with two bubbles at the input. Bubbled inputs are used to make a circuit diagram simpler to draw. The bubble represents a complemented input or output, depending on where the bubble is placed.

A gate with a bubble input (or output) may be referred to as an **active low input** (or **active low output**). A gate that does not contain a bubble input (or output) may be referred to as an **active high input** (or **active high output**). The AND2 in Figure 1.18 has two active high inputs and an active high output. The NAND2 has two active high inputs and an active low output. The NOR2B1 has one active high input, one active low input, and an active low output. An active high input requires a 1 or a high-voltage input for the input to be active or asserted. An active high output provides a 1 or high-voltage output for the output to be active or asserted. An active low input requires a 0 or a low-voltage input for the input to be active or asserted. An active low output provides a 0 or low-voltage output for the output to be active or asserted.

1.5.5 International Logic Symbols for Gates

The rectangular-shape logic symbols shown in Figure 1.19 are IEEE (Institute of Electrical and Electronics Engineers) international standard logic symbols. You need to learn these symbols in order to read circuits diagrams in international books and journals.

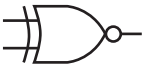
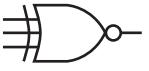
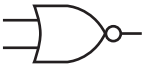
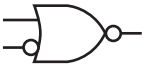
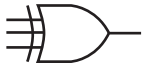
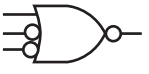
Logic symbols	Compact description names	Logic symbols	Compact description names
	AND2		XNOR2
	AND2B1		XNOR3
	AND3B2		NAND2
	OR2		NAND2B1
	OR2B1		NAND3B2
	OR3B2		NOR2
	XOR2		NOR2B1
	XOR3		NOR3B2

FIGURE 1.18 Logic symbols and their associated compact description names

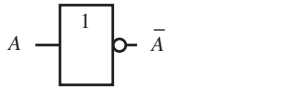
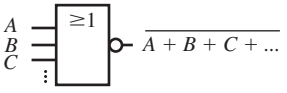
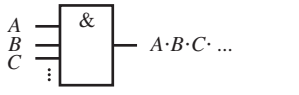
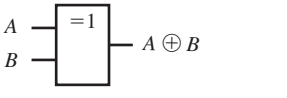
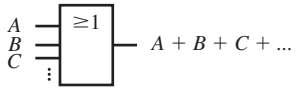
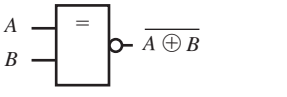
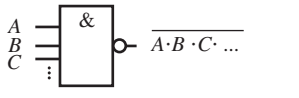
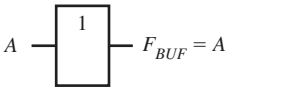
Rectangular shape logic symbols			
	Logic symbol name		Logic symbol name
	NOT gate		NOR gate
	AND gate		XOR gate (symbol for 2 inputs only)
	OR gate		XNOR gate (symbol for 2 inputs only)
	NAND gate		BUFFER

FIGURE 1.19 IEEE international standard logic symbols

The distinctive-shape logic symbols that we have used up until now are also IEEE standard logic symbols. Distinctive-shape logic symbols are predominantly used in the United States, so we will use the distinctive-shape logic symbols throughout the rest of this book.

PROBLEMS

Section 1.1 Introduction

1.1 What are three different ways to express Boolean functions, and what are their forms?
1.2 What are Boolean functions used to specify?
1.3 Name a few electronic devices that contain digital circuits.

Section 1.2 Basics of Boolean Algebra

1.4 What are the independent statements called in Boolean algebra that are assumed to be true without proof?
1.5 Name two early pioneers in the area of Boolean algebra.
1.6 What is the name of the binary operator "¯"?
1.7 What is the name of the binary operator "·"?
1.8 What is the name of the binary operator "+"?
1.9 Write a Boolean expression with a signal or signals for each of the binary operators NOT, AND, and OR.
1.10 What is a Venn diagram?
1.11 How are Boolean expressions represented on Venn diagrams?
1.12 If the area inside a border in a Venn diagram represents the Boolean expression $X + Y$, where is the area in the Venn diagram for the Boolean expression $\overline{X + Y}$?
1.13 What signals are shown on a black box for a Boolean function?
1.14 Does a black box show its digital circuit?
1.15 What are the input and output lines associated with a black box?
1.16 Show the logic symbols for the NOT or complement operation, the AND operation, and the OR operation using distinctive-shape symbols. For the AND operation and OR operation, show logic symbols with only two inputs.
1.17 For the following Boolean functions, draw the proper logic symbols and label them with their inputs and outputs: $F1 = P + Q + R$, $F2 = \overline{Y}$, $F3 = X \cdot Y \cdot Z$.
1.18 Write the Boolean function for each of the logic circuit diagrams in Figure P1.18.

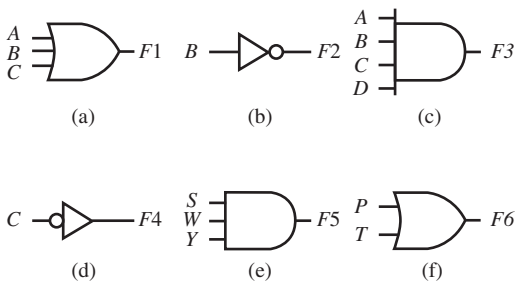


FIGURE P1.18

1.19 What are the values 1 and 0 called in a truth table?
1.20 Show the definitions with truth tables for the OR operator (use B and C as the input signals), the NOT operator (use A as the input signal), and the AND operator (use Y and Z as the input signals).
1.21 What are the values 1 and 0 called in Boolean algebra?
1.22 Show the definitions with all possible combinations of the identity elements (1 and 0) for the AND operator (use three identity elements), the OR operator (use two identity elements), and the NOT operator.
1.23 List the four names of the postulates or rules for Boolean algebra.
1.24 List the order of precedence of the binary operators.
1.25 One form of each of the four postulates or rules for Boolean algebra are provided as $A + 0 = A$ (variable dominant rule), $A + B = B + A$ (commutative rule), $A + B \cdot C = (A + B) \cdot (A + C)$ (distributive rule), and $A + \overline{A} = 1$ (complement rule). Write the dual form for each of the postulates.
1.26 Describe how to obtain the dual of a Boolean relationship, and show the dual of the Boolean relationship $X + Y \cdot Z \cdot W + A$.
1.27 Obtain the duals of the expressions $A + \overline{B} + C$, $X \cdot \overline{Y} + Z$, and $\overline{A} \cdot (B + \overline{C} \cdot D) + A \cdot D$.
1.28 What are theorems?
1.29 Show a logic circuit with a minimum gate count for the Boolean function $F = X \cdot (X + Y)$. Name and show the theorem you used to minimize the Boolean function prior to drawing the logic circuit.
1.30 Show a logic circuit with a minimum gate count for the Boolean function $F = X + \overline{X} \cdot Y$. Name and show the theorem you used to minimize the Boolean function prior to drawing the logic circuit.
1.31 Show a logic circuit with a minimum gate count for the Boolean function $F = A \cdot B + \overline{A} \cdot C + B \cdot C$. Name and show the theorem you used to minimize the Boolean function prior to drawing the logic circuit.
1.32 Prove that Idempotency Theorem T8a, $X \cdot X = X$, is true by the perfect induction method and the mathematical manipulation method.
1.33 Prove that DeMorgan's Theorem (with two variables) T5a, $\overline{X \cdot Y} = \overline{X} + \overline{Y}$, is true by the perfect induction method and the mathematical manipulation method.
1.34 Prove that Absorption Theorem T1a, $X \cdot (X + Y) = X$, is true by the perfect induction method and the mathematical manipulation method.

Section 1.3 Deriving Boolean Functions from Truth Tables

1.35 Obtain the CSOP forms for the following Boolean functions expressed in compact minterm form: (a) $F_1(X,Y) = \Sigma m(3)$, (b) $F_2(Y) = \Sigma m(0)$, (c) $F_3(A,B) = \Sigma m(1,2,3)$.
1.36 What is the memory jogger you can use when obtaining the expressions for minterms?
1.37 Obtain the truth tables for the following Boolean functions expressed in compact minterm form: (a) $F_1(X,Y) = \Sigma m(3)$, (b) $F_2(Y) = \Sigma m(0)$, (c) $F_3(A,B) = \Sigma m(1,2,3)$.

1.38 Obtain the CSOP forms for the following Boolean functions expressed in compact minterm form: (a) $\overline{F_1}(X,Y) = \Sigma m(2)$, (b) $\overline{F_2}(Y) = \Sigma m(1)$, (c) $\overline{F_3}(A,B) = \Sigma m(0,1,2)$.

1.39 Obtain the truth tables for the following Boolean functions expressed in compact minterm form: (a) $\overline{F_1}(X,Y) = \Sigma m(2)$, (b) $\overline{F_2}(Y) = \Sigma m(1)$, (c) $\overline{F_3}(A,B) = \Sigma m(0,1,2)$.

1.40 What is the memory jogger you can use when obtaining the expressions for maxterms?

1.41 Obtain the CPOS forms for the following Boolean functions expressed in compact maxterm form: (a) $F_1(X,Y) = \Pi M(0,1,2)$, (b) $F_2(Y) = \Pi M(0)$, (c) $F_3(A,B) = \Pi M(1,2)$.

1.42 Obtain the truth tables for the following Boolean functions expressed in compact maxterm form: (a) $F_1(X,Y) = \Pi M(0,1,2)$, (b) $F_2(Y) = \Pi M(0)$, (c) $F_3(A,B) = \Pi M(1,2)$.

1.43 Obtain the CPOS forms for the following Boolean functions expressed in compact maxterm form: (a) $\overline{F_1}(X,Y) = \Pi M(0,1,3)$, (b) $\overline{F_2}(Y) = \Pi M(1)$, (c) $\overline{F_3}(A,B) = \Pi M(1,3)$.

1.44 Obtain the truth tables for the following Boolean functions expressed in compact maxterm form: (a) $\overline{F_1}(X,Y) = \Pi M(0,1,3)$, (b) $\overline{F_2}(Y) = \Pi M(1)$, (c) $\overline{F_3}(A,B) = \Pi M(1,3)$.

Section 1.4 Writing VHDL Designs for Simple Gate Functions

1.45 What are the names for the acronyms CPLD and FPGA?

1.46 What are the two main sections of VHDL that you need to become familiar with?

1.47 Describe what the entity and the architecture contain in VHDL.

1.48 How many parts does a VHDL design consist of? Name them.

1.49 What are keywords in VHDL?

1.50 What are the keywords in the VHDL design in Listing P1.50?

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity larger_and is port (
     x, y, z : in std_logic;
     f : out std_logic
     );
end larger_and;

architecture Boolean_function of
larger_and is
begin
     f <= x and y and z;
end Boolean_function;
```

LISTING P1.50

1.51 How many inputs are there in the entity in Listing P1.50, and what are their names?

1.52 Show the black box for the entity in Listing P1.50.

1.53 Show the actual logic circuit for the architecture in Listing P1.50.

1.54 Show the black box for the entity in Listing P1.54.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity larger_or is port (
     w, x, y, z : in std_logic;
     f : out std_logic
     );
end larger_or;

architecture B_function of larger_or is
begin
     f <= w or x or y or z;
end B_function;
```

LISTING P1.54

1.55 Show the actual logic circuit for the architecture in Listing P1.54.

1.56 Show complete VHDL code for the Boolean function $F(A,B,C) = \Sigma m(2,3,6,7)$ for an entity named CSOP1 and the output signal *F*.

1.57 Show a complete and correct simulation for the VHDL design in problem 1.56. Identify the inputs using minterm designators.

1.58 Show complete VHDL code for the Boolean function $\overline{F}(A,B,C) = \Sigma m(1,5,7)$ for an entity named CSOP2 and the output signal *F*. Hint: Organize the function as $F(A,B,C) = \textbf{not}\ (\Sigma m(1,5,7))$.

1.59 Show a complete and correct simulation for the VHDL design in problem 1.58. Identify the inputs using minterm designators.

1.60 Show complete VHDL code for the Boolean function $F(W,X,Y,Z) = \Pi M(0,1,2,14)$ for an entity named CPOS1 and the output signal *F*.

1.61 Show a complete and correct simulation for the VHDL design in problem 1.60. Identify the inputs using maxterm designators.

1.62 Show complete VHDL code for the Boolean function $\overline{F}(X,Y,Z) = \Pi M(0,3,6)$ for an entity named CPOS2 and the output signal *F*. Hint: Organize the function as $F(X,Y,Z) = \textbf{not}\ (\Pi M(0,3,6))$.

1.63 Show a complete and correct simulation for the VHDL design in problem 1.62. Identify the inputs using maxterm designators.

Section 1.5 More about Logic Gates

1.64 Show the DeMorgan equivalent gate symbol for an AND gate.

1.65 Show the DeMorgan equivalent gate symbol for an OR gate.

1.66 Show the DeMorgan equivalent gate symbol for a NAND gate.

1.67 Show the DeMorgan equivalent gate symbol for a NOR gate.
1.68 Name two functionally complete gates, and show their logic symbols.
1.69 Show a NAND gate equivalent circuit for a NOT gate.
1.70 Show a NAND gate equivalent circuit for a NOR gate.
1.71 Show a NOR gate equivalent circuit for a NOT gate.
1.72 Show a NOR gate equivalent circuit for a NAND gate.
1.73 Show an equivalent gate circuit for the logic gate in Figure P1.73 using external NOT gates.

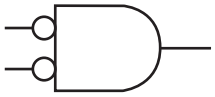


FIGURE P1.73

1.74 Show an equivalent gate circuit for the logic gate in Figure P1.74 using external NOT gates.

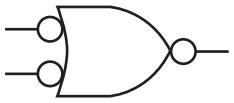


FIGURE P1.74

1.75 Show an equivalent gate circuit for the logic gate in Figure P1.75 using an external NOT gate.

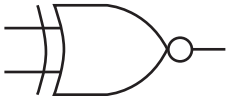


FIGURE P1.75

1.76 Show the logic symbol for the compact description name AND3B1.
1.77 Show the logic symbol for the compact description name NOR4.
1.78 Show the logic symbol for the compact description name OR3B1.
1.79 Show the logic symbol for the compact description name XNOR3.
1.80 Discuss how to identify an active low input for a gate and what an active low input means.
1.81 Discuss how to identify an active low output for a gate and what an active low output means.
1.82 Discuss how to identify an active high input for a gate and what an active high input means.
1.83 Discuss how to identify an active high output for a gate and what an active high output means.
1.84 Write the logic symbol names for the rectangular-shape logic symbols in Figure P1.84.

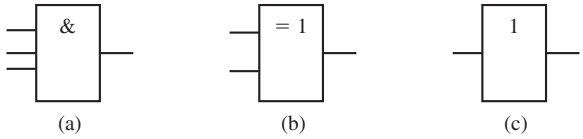


FIGURE P1.84

1.85 Write the logic symbol names for the rectangular-shape logic symbols in Figure P1.85.

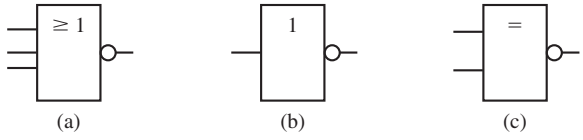


FIGURE P1.85

1.86 Write the Boolean expressions at the output for the rectangular-shape logic symbols in Figure P1.86.

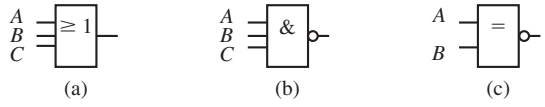


FIGURE P1.86

1.87 Write the Boolean expressions at the output for the rectangular-shape logic symbols in Figure P1.87.

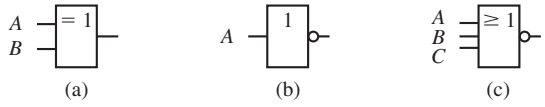


FIGURE P1.87

Chapter 2

Number Conversions, Codes, and Function Minimization

Chapter Outline

2.1 INTRODUCTION

In this chapter, you will learn the difference between two types of circuits—that is, digital circuits and analog circuits. You are introduced to binary numbers and their conversions to the decimal, octal, and hexadecimal number systems. You will also learn different types of binary codes and where they are used. You will learn how to design a small 7-segment display system using VHDL. You will learn how to minimize functions using a graphical technique called the Karnaugh map reduction method.

2.2 DIGITAL CIRCUITS VERSUS ANALOG CIRCUITS

There are two primary types of signals. These are **analog** or **continuous signals** and **discrete** or **digital signals**. Circuits that use analog signals are classified as **analog circuits**, while circuits that use digital signals are classified as **digital circuits**. Digital circuits operate with discrete signals, while analog circuits operate with continuous signals. In this text, we will only cover discrete signals, and thus we will only work with digital circuits.

2.2.1 Digitized Signal for the Human Heart

The majority of signals in nature are of the analog variety. The human heartbeat illustrated in Figure 2.1a is an example of an analog signal. Figure 2.1b shows a simple digitized version of the same signal.

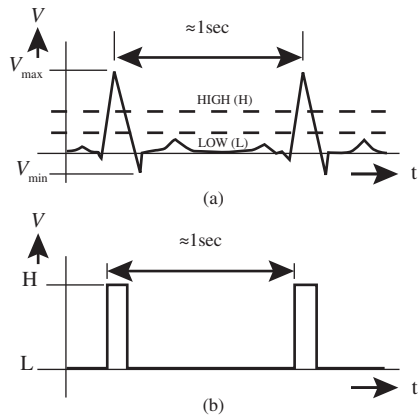


FIGURE 2.1 (a) Analog signal waveform or timing diagram for a human heartbeat; (b) digitized (digital) signal waveform or timing diagram for a human heartbeat

2.2.2 Discrete Signals versus Continuous Signals

Analog signals are continuous, while digital signals are noncontinuous; that is, discrete signals have distinct values. The mathematics (algebra and perhaps calculus) that you have learned up to this point in your career applies mostly to analog signals. The Boolean algebra that you learned back in Chapter 1 deals with two different discrete signal levels. The levels are high (or H) for high voltage and low (or L) for low voltage.

Notice that the analog signal shown in Figure 2.1a can be any value in the range of V_{min} (the minimum or lowest voltage) to V_{max} (the maximum or highest voltage) and is therefore a continuous signal. The digitized signal ideally has only two values **L or low** (a range of low-voltage values) and **H or high** (a range of high-voltage values). When H = 1 (1 or true) and L = 0 (0 or false), this protocol for identifying 1 and 0 is called the **positive logic convention**. When H = 0 (0 or false) and L = 1 (1 or true), then the protocol for identifying 1 and 0 is called the **negative logic convention**. We use the positive logic convention in this text.

The mathematics that deals with discrete signals or variables containing only two values is called Boolean algebra, switching algebra, or the algebra of logic. This is the algebra we used when we selected the values of 1 and 0 for the two values for the digital signals in Chapter 1.

2.3 BINARY NUMBER CONVERSIONS

In this section, we discuss number systems and conversions between the different number systems. The decimal number system, or base-10 number system, will be our reference system because we all know this one well. The binary number system, or base-2 number system, is a two-valued number system that is used in designing all types of digital systems. Two additional number systems will also be discussed. These number systems are the octal number system, or base-8 number system, and the hexadecimal number system, or base-16 number system. The octal and hexadecimal number systems are used to represent binary numbers in a more compact form.

2.3.1 Decimal, Binary, Octal, and Hexadecimal Numbers

The primary number systems used in digital systems are decimal, binary, octal, and hexadecimal. Consider the number 76. You may guess that the number is decimal because that is the number system you are most familiar with. A number without its **base** or **radix** identified is **implicitly written**. Usually when the base or radix is missing, a decimal (base-10) number is implied. To be more specific or exact, we can append a 10 to the number 76 like this $(76)_{10}$, to **explicitly show the base** or radix for the decimal number 76. Using this notation, there is now no doubt that the number 76 is a decimal number.

Recall that the decimal number system is a **weighted-positional number system** with only 10 digits to represent each digit position. The set of digits D contains only the following digits: D = {0,1,2,3,4,5,6,7,8,9}. All decimal numbers are built up from these digits. The **digit positions** are shown first in Example 2.1. Digit position 0 identifies the **least significant digit** (**LSD**), which is 4, and digit position 3 identifies the **most significant digit** (**MSD**), which is 6. The **weights of the digits** increase from the LSD on the right to the MSD on the left, as illustrated for the number $(6234)_{10}$.

3	2	1	0	Digit positions
10^3	10^2	10^1	10^0	Weights of the digits in powers of 10
1000	100	10	1	Weights of the digits
(6	2	3	$4)_{10}$	Decimal number
↑			↑	
MSD			LSD	

EXAMPLE 2.1 Weights of decimal digits

Like decimal numbers, binary numbers also have a base or radix. The base or radix is just 2 for binary numbers. Digital systems, including digital computers and microprocessors, only work with binary numbers. Decimal numbers are often used for human convenience, but all the internal math is done in the binary number system, not the decimal number system. To explicitly represent a binary number, we will append a 2 to the number like this: $(10110111)_2$. As we mentioned earlier, **binary digits** are called **bits** (short for **bi**nary digi**ts**). The set of digits or bits B for a binary number therefore contains only the following digits or bits: B = {0,1}. All binary numbers are built up from these digits. The digit positions or bits are shown first in Example 2.2. Bit position 0 identifies the least significant bit (LSB), which is 1, and digit position 7 identifies the most significant digit (MSB), which is also 1. The **weights of the bits** increase from the LSB on the right to the MSB on the left as illustrated for the number $(10110111)_2$.

7	6	5	4	3	2	1	0	Bit positions in decimal
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Weights of the bits in powers of 2
128	64	32	16	8	4	2	1	Weights of the bits in decimal
(1	0	1	1	0	1	1	$1)_2$	Binary number
↑							↑	
MSB							LSB	

EXAMPLE 2.2 Weights of binary digits or bits

All the number systems we will cover are weighted positional number systems. These number systems can always be represented using the digit notation in Example 2.3.

$$d_j \ldots d_5\, d_4\, d_3\, d_2\, d_1\, d_0$$

EXAMPLE 2.3 Digit notation for a positional number system

The value for each digit is dependent on the number system being used. A digit such as d_n can take on the values 0 through 9 in the decimal number system but only the values 0 and 1 in the binary number system. Notice that the subscripts for the digit notation are the digit positions. **Digit notation** is illustrated for the decimal number system in Example 2.4.

Weights	10^2	10^1	10^0
Decimal number	8	3	5
Digit notation	d_2	d_1	d_0

EXAMPLE 2.4 Digit notation for decimal numbers

In the binary number system, the base or radix is 2 instead of 10. Notice that digit notation for the binary number system and the decimal number system is the same as shown in Example 2.5.

Weights	2^2	2^1	2^0
Decimal number	1	0	1
Digit notation	d_2	d_1	d_0
Alternate notation (for binary numbers)	b_2	b_1	b_0

EXAMPLE 2.5 Digital notation for binary numbers

Notice that the least significant digit (LSD) or least significant bit (LSB) is d_0 or b_0, not d_1 or b_1, and always begins with the LSD or LSB for the number. Because the least significant bit begins with 0 and not 1, this is called **0 referencing** as opposed to **1 referencing**. When you first learned to count, you used 1 referencing. The first item counted was called 1, the second 2, and so on. In the binary number system, 0 referencing is used. This means that the first item counted is called 0, the second item 1, the third item 2, and so forth. If eight items are counted, they range from 0 through 7. For a count of n items the range is from 0 through $n - 1$, where d_{n-1} or b_{n-1} is the most significant digit (MSD) or the most significant bit (MSB).

2.3.2 Conversion Techniques

In this section, we discuss conversion techniques between each of the number systems: decimal, binary, octal, and hexadecimal. One way for humans to convert an integer decimal number to a binary number is to follow the conversion method referred to as the **subtract the weights method**. This step-by-step procedure or **algorithm** is listed as follows:

1. List the weights of the bits up to or beyond the value of the decimal number.
2. Working from the highest weight on the left to the smallest weight on the right, ask yourself the following question: Is the weight contained in the decimal number?
3. If the answer is no, then place a 0 for the bit value below the respective weight.
4. If the weight is contained in the decimal number, then place a 1 for the bit value below the respective weight and subtract the weight from the decimal number to obtain a reduced decimal number.
5. Repeat this process for the reduced decimal number for each lower bit positions.

When you finish, the weights with 1s under them should add up to the decimal number (this reverse procedure, called the **add the weights method**, can be used to convert an integer binary number to a decimal number). If the weights with the 1s under them do not add up to the decimal number, then check your work. This is a hand calculation method. You may also use a calculator to check your work. Most scientific calculators will perform base conversions of numbers. You may ask: Why must I learn to do base conversions when I have a calculator? The answer is simple. You may not have a calculator handy when you need it, or you may need to show that you can execute a technique for doing base conversions. The subtract the weights method is used to convert the decimal number 76 to its equivalent binary value in Example 2.6.

	76			12	4			
	−64			−8	−4			Weights in the binary number
	12			4	0			Reduced decimal number
128	64	32	16	8	4	2	1	Weights of the bits in decimal
(0	1	0	0	1	1	0	$0)_2$	Binary number = $(76)_{10}$

EXAMPLE 2.6 Subtract the weights method

Just as in decimal numbers, leading 0s are insignificant and may be dropped.

The reverse procedure, or add the weights method, can now be used to check your result. The algorithm is this: list the binary number and include the weights above the bits, then add the weights of all the bits for just the 1s in the binary number. The procedure is illustrated in Example 2.7.

64	32	16	8	4	2	1	Weights of the bits in decimal
(1	0	0	1	1	0	$0)_{10}$	Binary number

$(64 + 8 + 4)_{10} = (76)_{10}$

EXAMPLE 2.7 Add the weights method

Both octal and hexadecimal number systems are used for several reasons. These number systems offer a compression of the binary number system that is easier for humans to read, write, or remember. In addition, these two number systems can also be printed out in less space, which means that less printer ink is required. Long strings of binary numbers are not easy for humans to read, write, or remember. The number system being used may also be a function of the operating system used by the computer. A 12-bit binary number would be a string of 1s and 0s as shown in Example 2.8.

Binary number: 101100000001

EXAMPLE 2.8 12-bit binary number

One method of compressing this binary number is to obtain the equivalent number in the octal number system. **Each group of three bits** beginning at bit position 0 are assigned the weights 1, 2, 4 in sequence from right to left. When the weights are added via the add the weights method for each group of 3 bits, the numbers are octal numbers in the **octal number system** (base-8 or radix-8 number system). Because $2^3 = 8$, there are eight possible digits that range from 0 (000) to 7 (111). The set of digits O for an octal number contains only the following digits: O = {0,1,2,3,4,5,6,7}. The 12-bit number shown in Example 2.8 can be represented more simply in a compressed form in the octal number system as shown in Example 2.9.

$$\overset{421}{(101}\ \overset{421}{100}\ \overset{421}{000}\ \overset{421}{001)_2}$$
$$\underbrace{\quad}_{5}\ \underbrace{\quad}_{4}\ \underbrace{\quad}_{0}\ \underbrace{\quad}_{1}$$

so, $(101\ 100\ 000\ 001)_2 = (5401)_8$

EXAMPLE 2.9 Binary to octal conversion, called the groups of 3 method

For a **byte**, which is an 8-bit binary number, such as $(10111100)_2$, the conversion can easily be done by inspection, providing the result of $(274)_8$. For a **nibble**, which is a 4-bit binary number, $(1101)_2$, the octal equivalent number is $(15)_8$.

Converting the binary number $(111110)_2$ to octal yields 76. As you can see, you cannot always take for granted the base of a number that you think you know. When working with nondecimal numbers, you should use explicit notation, i.e., $(76)_8$.

To convert an octal number such as $(5673)_8$ to binary simply requires reversing the process, i.e., use the **groups of 3 method in reverse**. This is illustrated in Example 2.10.

$$\underbrace{(5}_{421}\ \underbrace{6}_{421}\ \underbrace{7}_{421}\ \underbrace{3)_8}_{421}$$
$$101\ 110\ 111\ 011$$

so, $(5673)_8 = (101\ 110\ 111\ 011)_2$

EXAMPLE 2.10 Groups of 3 method in reverse

An even more compact form for a binary number results when the number is converted to the hexadecimal number system. **Each group of four bits**, beginning at bit position 0, are assigned the weights 1, 2, 4, 8 in sequence from right to left When the weights are added via the add the weights method for each group of 4 bits, the numbers are decimal numbers. Substituting A, B, C, D, E, and F for the decimal numbers 10, 11, 12, 13, 14, and 15, respectively, converts the decimal numbers to hexadecimal numbers in the hexadecimal number system (base-16 or radix-16 number system). Because $2^4 = 16$, there are 16 possible digits that range from 0 to 15. The traditional symbols assigned to the numbers 10 to 15 are 10 = A, 11 = B, 12 = C, 13 = D, 14 = E, and 15 = F. The set of digits H for a hexadecimal number is H = {0,1,2,3,4,5,6,7,8,9,A,B, C,D,E,F}. The 12-bit number shown in Example 2.8 can be represented even more simply in a compressed form in the hexadecimal number system as shown in Example 2.11.

$$\begin{array}{cccc} & 8421 & 8421 & 8421 \\ (1011 & 0000 & 0001)_2 \\ \underbrace{} & \underbrace{} & \underbrace{} & \\ \mathrm{B} & 0 & 1 & \end{array}$$

so, $(1011\ 0000\ 0001)_2 = (\mathrm{B01})_{16}$

EXAMPLE 2.11 Binary to hexadecimal conversion, called the groups of 4 method

Bytes and nibbles such as $(10100101)_2$ and $(1110)_2$ in binary are easily converted to equivalent hexadecimal numbers using the **groups of 4 method** as $(\mathrm{A5})_{16}$ and $(\mathrm{E})_{16}$.

To convert a hexadecimal number such as EA to an equivalent binary number, simply reverse the process, using the **groups of 4 method in reverse**, as shown in Example 2.12.

$$\begin{array}{cc} (\mathrm{E} & \mathrm{A})_{16} \\ 14 & 10 \\ \overbrace{} & \overbrace{} \\ 8421 & 8421 \\ 1110 & 1010 \end{array}$$

so, $(\mathrm{EA})_{16} = (1110\ 1010)_2$

EXAMPLE 2.12 Groups of 4 method in reverse

Converting from hexadecimal to octal or from octal to hexadecimal is accomplished by simply regrouping the bits in the binary number, so $(\mathrm{C2})_{16} = (1100\ 0010)_2 = (11\ 000\ 010)_2 = (302)_8$ and $(452)_8 = (100\ 101\ 010)_2 = (0001\ 0010\ 1010)_2 = (\mathrm{12A})_{16}$.

In digital system design, we will only be concerned with the decimal, binary, octal, and hexadecimal number systems. To convert a number in the binary, octal, or hexadecimal number system to decimal, we can use the **polynomial function method**. The generalized form of the polynomial function is used to convert a number in any base or radix r to an equivalent number in the decimal number system and is written as follows:

$$\begin{aligned} (\text{Integer number in any number system})_r &= (d_j \ldots d_1 d_0)_r \\ &= (d_j r^j + \ldots + d_1 r^1 + d_0 r^0)_{10} \\ &= (\text{Decimal number})_{10} \end{aligned}$$

The binary number 100101 is converted to an equivalent decimal number using the polynomial function method as shown in Example 2.13.

$$
\begin{aligned}
(100101)_2 &= (d_5d_4d_3d_2d_1d_0)_2 \\
&= (d_5\times2^5 + d_4\times2^4 + d_3\times2^3 + d_2\times2^2 + d_1\times2^1 + d_0\times2^0)_{10} \\
&= (1\times32 + 0\times16 + 0\times8 + 1\times4 + 0\times2 + 1\times1)_{10} \\
&= (32 + 4 + 1)_{10} \qquad \text{weights of the 1s of the binary number} \\
&= (37)_{10}
\end{aligned}
$$

EXAMPLE 2.13 Binary number conversion to an equivalent decimal number via the polynomial function method

Notice that this conversion process is identical to the add the weights method discussed earlier for converting a binary number to its decimal equivalent.

To convert a number in any base to decimal, the polynomial function method can be programmed by you or a calculator manufacturer. The polynomial function can therefore be used to convert hexadecimal numbers to equivalent decimal numbers. The conversion of $(4BCE)_{16}$ to decimal is illustrated in Example 2.14.

$$
\begin{aligned}
(4BCE)_{16} &= (d_3d_2d_1d_0)_{16} \\
&= (d_3\times16^3 + d_2\times16^2 + d_1\times16^1 + d_0\times16^0)_{10} \\
&= (4\times4096 + 11\times256 + 12\times16 + 14\times1)_{10} \\
&= (16384 + 2816 + 192 + 14)_{10} \\
&= (19406)_{10}
\end{aligned}
$$

EXAMPLE 2.14 Hexadecimal number conversion to an equivalent decimal number via the polynomial function method

As an alternate method of converting a number in any base to a decimal number, we can use the **factored form (FF) method**. To illustrate this method, first observe the factored form of the polynomial function:

$$\text{Number in any base} = d_4r^4 + d_3r^3 + d_2r^2 + d_1r^1 + d_0r^0 = (((d_4r + d_3)r + d_2)r + d_1)r + d_0$$

Example 2.15 shows how the FF method is used to convert the binary number $(101011)_2$ to its equivalent decimal number.

$(1$	0	1	0	1	$1)_2$
$1 \times 2 = 2$	$(2 + 0) \times 2 = 4$	$(4 + 1) \times 2 = 10$	$(10 + 0) \times 2 = 20$	$(20 + 1) \times 2 = 42$	$42 + 1 = (43)_{10}$

EXAMPLE 2.15 Binary number conversion to an equivalent decimal number via the FF method

Example 2.16 shows how the FF method is used to convert the octal number $(314)_8$ to its equivalent decimal number.

$(3$	1	$4)_8$
$3 \times 8 = 24$	$(24 + 1) \times 8 = 200$	$(200 + 4) = (204)_{10}$

EXAMPLE 2.16 Octal number conversion to an equivalent decimal number via the FF method

Example 2.17 shows how the FF method is used to convert the hexadecimal number $(A4E)_{16}$ to its equivalent decimal number.

$(A$	4	$E)_{16}$
$10 \times 16 = 160$	$(160 + 4) \times 16 = 2{,}624$	$(2{,}624 + 14) = (2{,}638)_{10}$

EXAMPLE 2.17 Hexadecimal number conversion to an equivalent decimal number via the FF method

To convert a decimal number to any base, the **repeated radix division method** for integers can be used. We use this method to convert a decimal number to its equivalent binary number (sometimes called the **double-dabble method**), to convert a decimal number to its equivalent octal number (**octal-dabble method**), and to convert a decimal number to its equivalent hexadecimal number (**hex-dabble method**). The repeated radix division method can also be programmed by you or a calculator manufacturer. To illustrate this method, the following integer number will be used:

$$\begin{aligned}\text{Decimal number} &= d_3r^3 + d_2r^2 + d_1r^1 + d_0r^0\\ &= d_3r^3 + d_2r^2 + d_1r + d_0\end{aligned}$$

Dividing by the radix r (2 for binary, 8 for octal, or 16 for hexadecimal) results in

$$\frac{\text{Decimal number}}{r} = \frac{d_3r^3}{r} + \frac{d_2r^2}{r} + \frac{d_1r}{r} + \frac{d_0}{r}$$

When the division is carried out, the quotient is

$d_3r^2 + d_2r^1 + d_1$, with a remainder of d_0 (LSB)

Continuing with radix division of each remaining quotient, we obtain a set of remainders. These remainders are then used to represent the number in the required base by writing each remainder digit in its correct position as

$$\text{Decimal number} = (d_3d_2d_1d_0)_r \text{ for } r = 2 \text{ (binary)}, r = 8 \text{ (octal)}, \text{ or } r = 16 \text{ (hexadecimal)}$$

From our discussion, the least significant digit (d_0) is found first, and the most significant digit (d_3) is found last.

The conversion of the decimal number 182 to its equivalent binary number using the repeated radix division method (double-dabble method) is shown in Example 2.18.

182 / 2 = 91 with remainder (wr) = 0 = d_0 (LSB)
91 / 2 = 45 wr = 1 = d_1
45 / 2 = 22 wr = 1 = d_2
22 / 2 = 11 wr = 0 = d_3
11 / 2 = 5 wr = 1 = d_4
5 / 2 = 2 wr = 1 = d_5
2 / 2 = 1 wr = 0 = d_6
1 / 2 = 0 wr = 1 = d_7 (MSB)
so, $(182)_{10} = (d_7d_6d_5d_4d_3d_2d_1d_0)_2 = (10110110)_2$

EXAMPLE 2.18 Conversion of the decimal number 182 to its equivalent binary number using the repeated radix division method (double-dabble method)

The conversion of the decimal number 182 to its equivalent octal number using the repeated radix division method (octal-dabble method) is shown in Example 2.19.

182 / 8 = 22 wr = 6 = d_0 (LSB)
22 / 8 = 2 wr = 6 = d_1
2 / 8 = 0 wr = 2 = d_2 (MSB)
so, $(182)_{10} = (d_2d_1d_0)_8 = (266)_8$

EXAMPLE 2.19 Conversion of the decimal number 182 to its equivalent octal number using the repeated radix division method (octal-dabble method)

The conversion of the decimal number 182 to its equivalent hexadecimal number using the repeated radix division method (hex-dabble method) is shown in Example 2.20.

$182 / 16 = 11 \quad \text{wr} = 6 = d_0 \text{ (LSB)}$
$11 / 16 = 0 \quad \text{wr} = 11 = B = d_1 \text{ (MSB)}$
so, $(182)_{10} = (d_1 d_0)_{16} = (B6)_{16}$

EXAMPLE 2.20 Conversion of the decimal number 182 to its equivalent hexadecimal number using the repeated radix division method (hex-dabble method)

2.4 BINARY CODES

Binary codes are special strings of binary digits. The most common code is the **binary coded decimal** (**BCD**) code. Another very common code is used on your personal computer. This code is called the **American Standard Code for Information Interchange** (**ASCII**) and is pronounced AS-key. Other codes such as reflective Gray code, 2-out-of-5 code, and 7-segment code round out our discussion of codes.

2.4.1 Minimum Number of Bits for Keypads and Keyboards

Before you can begin to understand codes, it is important to realize how many bits it takes to represent a particular range of numbers. A decimal number is used to represent the keys on a keypad or keyboard. The formula that allows us to determine the minimum number of bits for any size keypad or keyboard can be written as follows:

$2^{\text{Number of bits}} \geq \text{Number of keys}$

To represent the decimal numbers or digits 0 through 9 on a keyboard or keypad requires 10 keys. How many bits are required to represent 10 keys? Because $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, and $2^4 = 16$, respectively, you can see that at least 4 bits are required to represent 10 keys. You can always use more bits than the minimum.

As you have observed, trial and error can be used to determine the smallest number of bits for a keyboard, but a calculator can also determine the smallest number of bits as shown in Example 2.21.

$$\begin{aligned}\text{Number of bits} &= \lceil \log_2(\text{Number of keys}) \rceil \quad \leftarrow \text{Ceiling function, smallest integer; i.e., } \geq \log_2(\text{Number of keys}) \\ &= \lceil \log_2 10 \rceil \\ &= \lceil \mathbf{ln}10/\mathbf{ln}2 \rceil = \lceil 3.32 \rceil = 4\end{aligned}$$

EXAMPLE 2.21 Calculation for the smallest number of bits for a keyboard using 10 keys

2.4.2 Commonly Used Codes: BCD, ASCII, and Others

When you type on the keyboard of a computer or enter commands on the keypad of a microwave oven, a TV remote control unit, a calculator, or a telephone, how do the numbers and letters designated on the push-button switches get mapped or transformed to binary numbers?

When you want to enter a character, you simply press the push button marked with the appropriate character. Each push button on a keyboard or a keypad is assigned a string of special binary digits called a **code**.

The decimal numbers 0 through 9 can be **encoded** (represented in a binary code) in several ways. The minimum number of bits required is only 4 bits, but some encoding schemes use more bits as shown in Table 2.1.

TABLE 2.1 Decimal numbers represented in various binary codes

Decimal	Binary code	8 4 2 1 or BCD code	Reflective Gray code	2-out-of-5 coded decimal code
0	0000	0000	0000	00011
1	0001	0001	0001	00101
2	0010	0010	0011	00110
3	0011	0011	0010	01001
4	0100	0100	0110	01010
5	0101	0101	0111	01100
6	0110	0110	0101	10001
7	0111	0111	0100	10010
8	1000	1000	1100	10100
9	1001	1001	1101	11000
10	1010	0001 0000	1111	00101 00011
25	11001	0010 0101	10101	00110 01100
37	100101	0011 0111	110111	01001 10010
98	1100010	1001 1000	1010011	11000 10100

BCD (binary coded decimal) is usually easier for humans to use than binary to represent decimal numbers with more than two digits. In this code, the numbers from 0000 or decimal 0 to 1001 or decimal 9 are represented in straight binary. For the decimal number with two or more digits, each digit is represented as a 4-bit code. Decimal 25 is represented as 0010 0101 in BCD, while decimal 437 is represented as 0100 0011 0111 in BCD.

Reflective Gray code (RGC) is important and is discussed in the next section.

The **2-out-of-5 coded decimal code** is an example of a very simple error detection code that can be used for data transmission applications. Every code word for the decimal digits 0 through 9 for this code always has two 1s and three 0s—that is, two ls out of the 5 bits. Digital circuitry counts the bits at the receiving end of a data transmission link utilizing this code. If the number of 1s is two at the receiving end, it is most likely that the data have arrived without errors (unless two bits were in error, which is highly unlikely although not impossible). If the number of 1s is not two at the receiving end, then it is most likely that the data have arrived with errors (probably due to a single bit error).

In 1950, Richard Hamming introduced a class of error detection/correction codes now called **Hamming codes**. This coding scheme allows practically any code to be modified by adding so-called **parity** or **check bits**. With enough parity or check bits added to the code, digital circuitry can detect as well as correct errors in the data being received. Error detection/correction codes are often utilized in the banking industry, where fortunes can be lost by a misplaced decimal point; in the Space Shuttle, where human lives are at risk; or whenever error-free data transmission is essential. Computers subject to continuous use often utilize error detection/correction code circuitry in their internal memory systems. You will be introduced to such codes in more advanced studies.

The alphanumeric character code that is found on almost all PCs today is extended 8-bit ASCII. This standard code is used to represent numbers, letters of the alphabet, and some special control and graphic characters. ASCII is the dominant keyboard code for alphanumeric characters today and is shown in Table 2.2 in the original (nonextended) form.

TABLE 2.2 7-bit ASCII character set code (nonextended form)

(a)

	$b_6\,b_5\,b_4$							
$b_3\,b_2\,b_1\,b_0$	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NULL	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	SYNC	&	6	F	V	f	v
0111	BELL	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

(b)

Control	**Characters:**	**Control**	**Characters:**	**Graphic**	**Characters:**
NULL	Null	DLE	Data link escape	’	Apostrophe
SOH	Start of heading	DC1	Device control 1	-	Hyphen
STX	Start of text	DC2	Device control 2	/	Forward slant
ETX	End of text	DC3	Device control 3	<	Less than
EOT	End of transmission	DC4	Device control 4 (stop)	>	Greater than
ENQ	Enquiry	NAK	Negative knowledge	[	Opening bracket
ACK	Acknowledge	SYNC	Synchronous idle	\	Reverse slant
BELL	Bell (audible Signal)	ETB	End of transmission block	]	Closing bracket
BS	Backspace	CAN	Cancel	^	Circumflex
HT	Horizontal tabulation	EM	End of medium	_	Underline
LF	Line feed	SUB	Substitute	`	Grave accent
VT	Vertical tabulation	ESC	Escape	{	Opening brace
FF	Form feed	FS	File separator	|	Vertical line
CR	Carriage return	GS	Group separator	}	Closing brace
SO	Shift out	RS	Record separator	~	Overline (tilde)
SI	Shift in	US	Unit separator		
		DEL	Delete		

The nonextended ASCII, or 7-bit ASCII, character set contains 128 different characters and thus 7 bits are required to represent all the characters ($2^7 = 128$). The extended ASCII character set uses the nonextended ASCII character set for the first 128 characters (with bit position $7 = 0$) followed by an additional 128 characters (with bit position $7 = 1$).

2.4.3 Modulo-2 Addition and Conversions between Binary and Reflective Gray Code

Another very common code, **reflective Gray code**, or **unit distance code**, has the property that only 1-bit changes as the code changes from its decimal equivalent value of i to $i + 1$; that is, for each pair of consecutive decimal numbers, the equivalent reflective Gray code numbers differ in only 1-bit position. Reflective Gray code numbers can be generated by the so-called **mirror method** as illustrated in Figure 2.2.

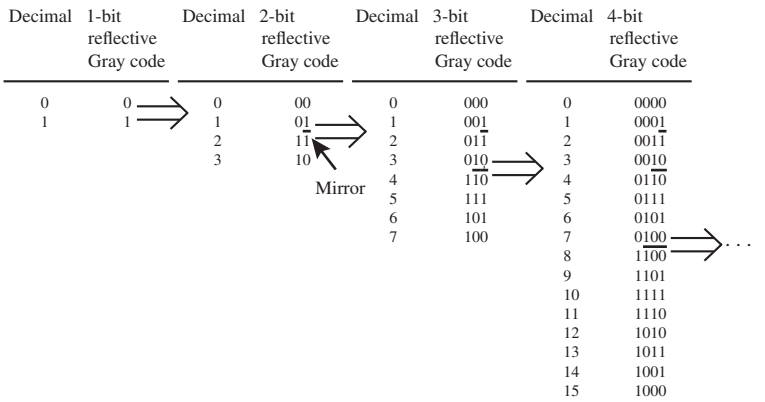


FIGURE 2.2 Decimal to reflective Gray code conversion using the mirror method

Notice in Figure 2.2 that the 4-bit reflective Gray code numbers for decimal 3 (0010) and decimal 4 (0110) differ only in bit position b_2. Likewise, the 4-bit reflective Gray code numbers for decimal 7 (0100) and decimal 8 (1100) differ only in bit position b_3. First write down all 1-bit reflective Gray codes (0 and 1), and draw a line under the last bit (1) as shown in Figure 2.2. Next write down 1 and 0 under the line with the line acting as a vertical mirror to reflect these bits from the ones written above the line. The 1 under the line appears closer to the line and the 0 appears further away. Finally, prefix the top bits over the line with 0s, and prefix the bottom bits under the line with 1s. This ends the process for obtaining all 2-bit reflective Gray code numbers.

Notice that one line or mirror is drawn to determine all 2-bit reflective Gray codes. To determine all 3-bit reflective Gray codes, begin with all 2-bit reflective Gray codes in decimal sequence and repeat the process as illustrated in Figure 2.2. Notice that the number of lines or mirrors that you draw in each case is always one less that the number of reflective Gray code bits.

The mirror method is interesting but time consuming for obtaining the equivalent reflective Gray code number for a single decimal number. Suppose that you wanted to obtain the equivalent reflective Gray code number for the decimal number 152 for 8 bits. You would need to repeat the mirror method 7 times.

Identify a single decimal number, and do the following to determine its equivalent reflective Gray code number:

1. Convert the decimal number to its equivalent binary number.
2. Apply the **binary to RGC conversion method** shown in Figure 2.3.

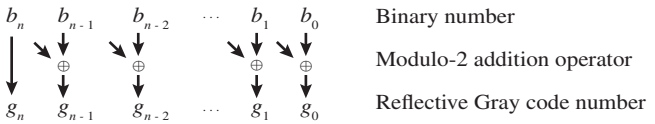


FIGURE 2.3 Binary to RGC conversion method

The symbol ⊕ represents the **modulo-2 addition operator**. Modulo-2 addition is the addition of 2 bits without a carry—that is, the carry bit is ignored. The modulo-2 addition operator is the same as the XOR operator. Table 2.3 shows the modulo-2 addition table.

TABLE 2.3 Modulo-2 addition table

0	0	1	1	} Two bits to be added by modulo-2 addition
⊕0	⊕1	⊕0	⊕1	
0	1	1	0	← Sum bits, ignore carry

In Example 2.22, the decimal numbers 27 and 76 are first converted to binary numbers and then converted to reflective Gray code numbers using the binary to RGC conversion method.

$$(27)_{10} = (1\ 1\ 0\ 1\ 1)_{2\ B}$$
$$= (1\ 0\ 1\ 1\ 0)_{2\ RGC}$$
$$(76)_{10} = (1\ 0\ 0\ 1\ 1\ 0\ 0)_{2\ B}$$
$$= (1\ 1\ 0\ 1\ 0\ 1\ 0)_{2\ RGC}$$

EXAMPLE 2.22 Conversions from decimal to binary to reflective Gray code

Figure 2.4 shows a **RGC to binary conversion method** for converting a reflective Gray code number back to a binary number, which can be, if desired, converted back to its decimal equivalent number.

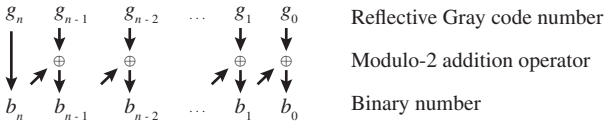


FIGURE 2.4 RGC to binary conversion method

An example of this conversion process is shown in Example 2.23 for reflective Gray code numbers $(10110)_{2\ RGC}$ and $(1101010)_{2\ RGC}$ obtained previously in Example 2.22. Both $(10110)_{2\ RGC}$ and $(1101010)_{2\ RGC}$ are converted back to binary numbers using the RGC to binary conversion method and then converted to equivalent decimal numbers.

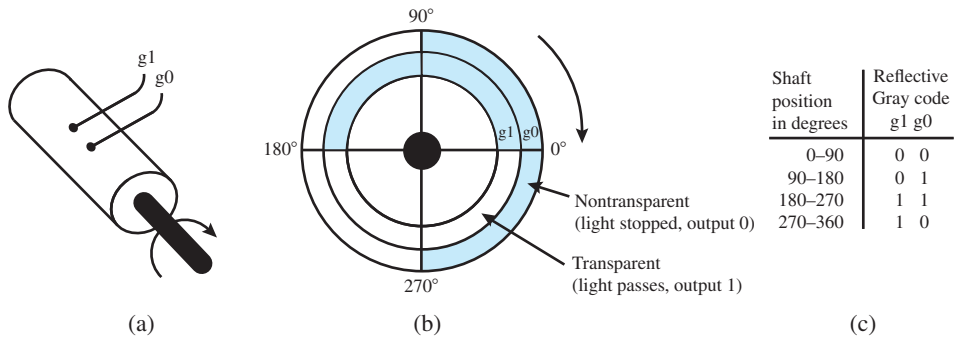
$$(1\ 0\ 1\ 1\ 0)_{2\ RGC}$$
$$= (1\ 1\ 0\ 1\ 1)_{2\ B}$$
$$= (27)_{10}$$

$$(1\ 1\ 0\ 1\ 0\ 1\ 0)_{2\ RGC}$$
$$= (1\ 0\ 0\ 1\ 1\ 0\ 0)_{2\ B}$$
$$= (76)_{10}$$

EXAMPLE 2.23 Conversions from reflective Gray code to binary to decimal

Modulo-2 addition is performed by an XOR gate. A circuit with XOR gates can be connected to convert binary code to reflective Gray code by simply arranging the XOR gates so that they perform the binary to RGC conversion method. Likewise, a circuit with XOR gates can be connected to convert reflective Gray code to binary code by arranging the XOR gates so that they perform the RGC to binary conversion method.

Reflective Gray code is often used in shaft-angle encoders as shown in Figure 2.5. This example shows a **binary-encoding device** that measures the angular position of a rotating shaft. To keep the example simple, we show a shaft-angle encoder with a two-bit output in Figure 2.5a. The output angular resolution in degrees for two output bits is only one-fourth of a full revolution (360 degrees), or 90 degrees. An encoded disk similar to the one illustrated in Figure 2.5b is mounted on the shaft inside the encoding device. As the shaft rotates the encoding device converts the angular position of the shaft (the four separate quadrants) into four different reflective Gray code numbers. The shaft position in degrees and the corresponding reflective Gray-code output for a two-bit shaft-angle encoder are shown in Figure 2.5b. The reflective Gray code values shown in the table in Figure 2.5c may then be used to provide input signals to a circuit to indicate the angular position of the shaft. A shaft-angle encoder with a larger number of bits provides a higher output angular resolution. For 3 bits, the output angular resolution in degrees would be 45 degrees and for n bits the output angular resolution would be $360/2^n$ degrees.



Shaft position in degrees	Reflective Gray code g1 g0
0–90	0 0
90–180	0 1
180–270	1 1
270–360	1 0

FIGURE 2.5 (a) Sketch of a shaft-angle encoder; (b) blown-up sketch of encoded disk using reflective Gray code; (c) table showing shaft position in degrees and corresponding output in reflective Gray code.

An encoded disk using a reflective Gray code allows only 1-bit to change between successive Gray code numbers. A code with this property is also called a unit distance code. If a Gray code was not used, area-shading misalignments that could occur in manufacturing the encoded disk would allow more than 1-bit to change between successive angular positions and thus cause momentary errors. A Gray code prevents these misalignment errors.

Two light sources and two photocells can be used with the encoded disk illustrated in Figure 2.5b. When light shines through a transparent area of the encoded disk, the bit associated with that output is a binary 1; otherwise, it is a binary 0. An alternate design could be a set of two brushes rubbing against a disk that has conducting and insulated areas such that a conducting area produces a binary 1 while an insulated area produces a binary 0. These examples serve to illustrate the principle involved in the encoding process.

The shaft-angle encoder is a mechanical form of **analog-to-digital converter**. The shaft position brought about by a continuous rotation of the shaft (an analog signal) is converted to a digital signal whose bit pattern is encoded using a reflective Gray code.

Another example of where the reflective Gray code sequence is used will be provided later when we discuss the graphical minimization technique that is referred to as a Karnaugh map or K-map.

2.4.4 7-Segment Code

A very common code that practically everyone encounters every day is the 7-segment type of code. This code is used for readouts for digital watches, voltmeters, blood pressure monitors, microwave ovens, toys, pedometers, and odometers in some new cars. An example of this code is shown in Table 2.4 for a 7-segment display.

TABLE 2.4 7-segment code for a 7-segment display

7-segment code	7-segment display			
$b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$				
0 1 1 1 1 1 1				
0 0 0 0 1 1 0				
1 0 1 1 0 1 1				
1 0 0 1 1 1 1				
1 1 0 0 1 1 0				
1 1 0 1 1 0 1				
1 1 1 1 1 0 0		or		With a flag $(b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0)$ (1 1 1 1 1 0 1)
0 0 0 0 1 1 1				
1 1 1 1 1 1 1				
1 1 0 0 1 1 1		or		With a flag $(b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0)$ (1 1 0 1 1 1 1)

Notice in Table 2.4 that bits $b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ represent the segments g, f, e, d, c, b, and a, respectively. This display requires a binary 1 or high signal to light a segment and a binary 0 or low signal to turn off a segment such that 0000110 is the code for the decimal digit 1. To change the decimal digit 1 to the decimal digit 7 only requires that the signal b_0 or segment a change from 0 to 1. This code is used to drive a 7-segment display device called a **common-cathode** LED (light-emitting diode) display.

A **common-anode** LED display requires the 1's complement of the code in Table 2.4. The 1's complement is simply the complement of each individual bit. For a common-anode LED display, a binary 0 or low signal lights a segment and a binary 1 or high signal turns off a segment. Both common-cathode and common-anode display devices provide the same output decimal digits 0 through 9. The input signals for a common-cathode display must be high to light the segments because the inputs are **active high**. The input signals for a common-anode display must be low to light the segments because the inputs are **active low**.

It is also possible to display the hexadecimal symbols A, b, c or C, d, E, and F when the correct codes are supplied to the segments. The code used for the lowercase letter b for a 7-segment common cathode display with the segments g, f, e, d, c, b, and a assigned as bits $b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ is 1111100. A lowercase letter b is used because an uppercase letter B would look just like the symbol for 8. It should be noted that when a 7-segment display is used to display hexadecimal symbols A, b, C, d, E, and F, a flag segment (segment a) is lighted for 6 to distinguish between the symbol 6 and the symbol b. The code used for the lowercase letter d for a 7-segment common-anode display with the segments g, f, e, d, c, b, and a assigned as bits $b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ is 0100001. A lowercase letter d is used because an uppercase D would look just like the symbol for 0.

2.4.5 VHDL Design for a Letter Display System

Figure 2.6 shows a schematic for a letter display system that displays an L (for low) on a 7-segment display, when a binary input of 0 is applied to the input of the letter display decoder via a slide switch, and displays an H (for high) when a binary input of 1 is applied at the input of the letter display decoder.

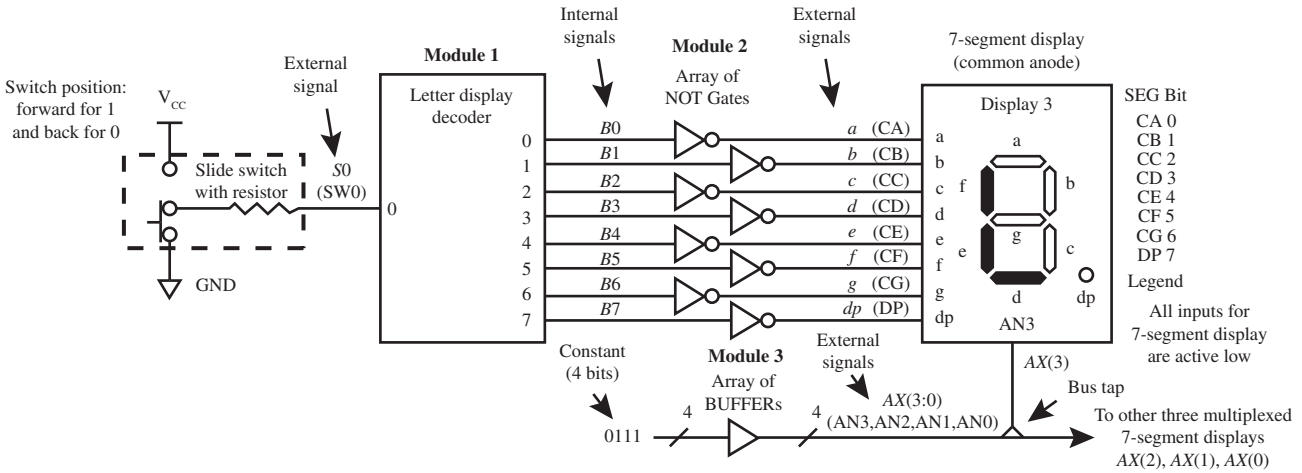


FIGURE 2.6 Schematic for a letter display system for displaying letters L and H on a 7-segment display

Things you should notice about the schematic for the letter display system (LDS) in Figure 2.6.

- There are three separate modules that make up the LDS. These are module 1, the letter display decoder circuit; module 2, the array of NOT gates; and module 3, the array of BUFFERs. The slide switch is an input device, and the 7-segment display is an output device.
- The slide switch that drives the letter display decoder is used to select the letter that will be lighted. When 0 is supplied to the input of the letter display decoder, the letter that will be lighted is an L, and when a 1 is supplied to the input the letter that will be lighted is an H. We have elected to make the outputs of the letter display decoder active high. This means that a 1 lights a segment, while a 0 turns off or blanks a segment.
- The 7-segment display has active low inputs, which means that the segments are lighted by a 0 and turned off by a 1. Because the outputs of the letter display decoder are active high and the inputs of the 7-segment display are active low, there is a mismatch. The outputs of the letter display decoder must be changed to provide the required active low input for the 7-segment display. The polarity mismatch is fixed by the array of NOT gates, because the outputs of the NOT gates complement (or invert) the outputs of the letter display decoder so that they agree with the inputs required by the 7-segment display.
- The 7-segment display is a multiplexed display that consists of four separate displays. Only display 3 is shown in Figure 2.6. Each display is turned on or off by applying a signal to the display enable inputs AN3, AN2, AN1, and AN0, respectively. The small triangle (called the bus tap) on the bus *AX*(3:0) represents a connection to an individual signal on the bus. Display 3 is turned ON via input AN3, display 2 is turned on via input AN2, etc. A display is turned on by applying a 0 and turned off by applying a 1 because these inputs are also

active low. The array of BUFFERs is supplied with the constant 0111 to turn display 3 on and to turn display 2, display 1, and display 0 off.

- Observe that all the wires or nets are labeled with a signal name. A schematic or circuit diagram with this feature is referred to as an annotated schematic or annotated circuit diagram.

Table 2.5 shows the truth table for the letter display decoder in Figure 2.6. Remember that the outputs are active high, so a 1 lights a segment and a 0 turns off or blanks a segment.

Using the letter display decoder truth table, we can write the output functions as follows: $B0 = 0$, $B1 = S0$, $B2 = S0$, $B3 = \overline{S0}$, $B4 = 1$, $B5 = 1$, $B6 = S0$, and $B7 = 0$. $B7$ provides an output for a decimal point, and it is set to 0 to turn the decimal point off.

TABLE 2.5 Truth table for letter display decoder in Figure 2.6

Letter display decoder (active high outputs)									7-segment display
$S0$	$B7$	$B6$	$B5$	$B4$	$B3$	$B2$	$B1$	$B0$	
0	0	0	1	1	1	0	0	0	L
1	0	1	1	1	0	1	1	0	H

Using the schematic for the letter display system, we can write the output equations for the array of NOT gates as follows: $a = \overline{B0}$, $b = \overline{B1}$, $c = \overline{B2}$, $d = \overline{B3}$, $e = \overline{B4}$, $f = \overline{B5}$, $g = \overline{B6}$, and $dp = \overline{B7}$.

Complete VHDL code for the letter display system is shown in Listing 2.1.

LISTING 2.1
Complete VHDL code for the letter display system (project: LDS_BASYS2 or LDS_NEXYS2)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity LDS is port (
     s0 : in std_logic;
     a,b,c,d,e,f,g,dp : out std_logic;
     ax :out std_logic_vector (3 downto 0)
     );
end LDS;

architecture Boolean_functions of LDS is
--Internal signals
     signal b0,b1,b2,b3,b4,b5,b6,b7: std_logic;
begin

--Letter Display Decoder
     b0 <= '0'; b1 <= s0; b2 <= s0; b3 <= not s0;
     b4 <= '1'; b5 <= '1'; b6 <= s0; b7 <= '0';
--Array of NOT gates
     a <= not b0; b <= not b1; c <= not b2; d <= not b3;
     e <= not b4; f <= not b5; g <= not b6; dp <= not b7;
--Array of BUFFERs to enable 7-segment display
     ax <= "0111";
end Boolean _ functions;
```

Things you should notice about the VHDL design in Listing 2.1:

- Keep in mind that VHDL is case insensitive.
- The port signals in the entity are the external signals. These are the input signal *S*0, and the output signals *a*, *b*, *c*, *d*, *e*, *f*, *g*, and *dp* from the array of NOT gates, and the output signals *AX* from the array of BUFFERs.
- The signals *S*0, *a*, *b*, *c*, *d*, *e*, *f*, *g*, and *dp* have the data type called std_logic, which means that they are **scalars**. A scalar is a signal that feeds a single wire. The signal *AX* has the data type called std_logic_vector. A **vector** is a signal that feeds a group of similar wires. On a schematic, a group of similar wires is called a **bus**. A vector must have a range to declare the number of signals in the group. The signal *AX* has the range 3 down to 0, written as (3 **downto** 0), which indicates there are four signals in the group. Downto is a keyword in VHDL and is one word.
- The signals coming out of the letter display decoder in Figure 2.6 and going into the array of NOT gates are internal signals. These are the signals *B*0, *B*1, *B*2, *B*3, *B*4, *B*5, *B*6, and *B*7. Internal signals in VHDL must be placed between architecture and the first begin as shown in Listing 2.1.
- The comments, such as -- Internal signals, and -- Letter Display Decoder, are used to help document the VHDL code; that is, they make it easier for someone to read and understand the VHDL code. Later on we will provide more details on a good documentation style.
- The design approach we used in writing the VHDL code is a **flat design approach**, because each module of the design is included within a single architecture. Another design approach is a **hierarchal design approach**, which will be covered later.
- In VHDL, single bits such as 0 and 1 in Listing 2.1 must be surrounded by single quotation marks, and a string (or series) of two or more bits such as 0111 in Listing 2.1 must be enclosed by double quotation marks.

2.5 KARNAUGH MAP REDUCTION METHOD

Boolean equations that make up circuits should be reduced prior to obtaining a hardware design to reduce the gate count. VHDL has built in reduction algorithms to reduce Boolean functions. An **algorithm** is a procedure for solving a problem in a finite number of steps. Fewer variables result in simpler digital circuits, so reduction is a good thing, because a reduced gate count may cost less and require less power to operate. The most common and fastest way to manually reduce or minimize a Boolean function is called the **Karnaugh map reduction method**.

A **literal** is defined as a variable or complement of a variable. The **literal count** of a Boolean expression is simply the sum of the single variables and the complement of single variables such as X, $\overline{X}$, Y, $\overline{Y}$ in a Boolean expression. Boolean expressions with a smaller literal count require a smaller number of gates to build digital circuits. The goal is to obtain Boolean functions with the fewest number of literals—that is, the smallest literal count.

Maurice Karnaugh, an engineer at Bell Labs, invented the Karnaugh map in 1953 ("The Map Method for Synthesis of Combinational Logic Circuits," *Trans. AIEE*, Communications and Electronics, 72, Part 1, November 1953, pages 593–599). **Combinational** or **combinatorial** logic circuits are circuits with outputs that depend only on the external inputs applied to the circuits. A **Karnaugh map**, or **K-map**, is a pictorial or graphical method used for reducing Boolean functions, which relies heavily on the ability of our minds to perceive patterns. As it turns out, our minds can handle this task remarkably well.

Given a truth table for a function or the equivalent compact minterm or maxterm form for the function we can

1. Draw a K-map for the function.
2. Fill in the K-map.
3. Write a reduced or minimum expression for the function by observing patterns in the K-map.

2.5.1 The Karnaugh Map Explorer

We will use a computer program to help you gain an understanding of how to reduce Boolean functions. From this textbook's website, download the program Karnaugh Map Explorer on your computer. Figure 2.7 shows the screen or graphical user interface (GUI) for KME (Karnaugh Map Explorer).

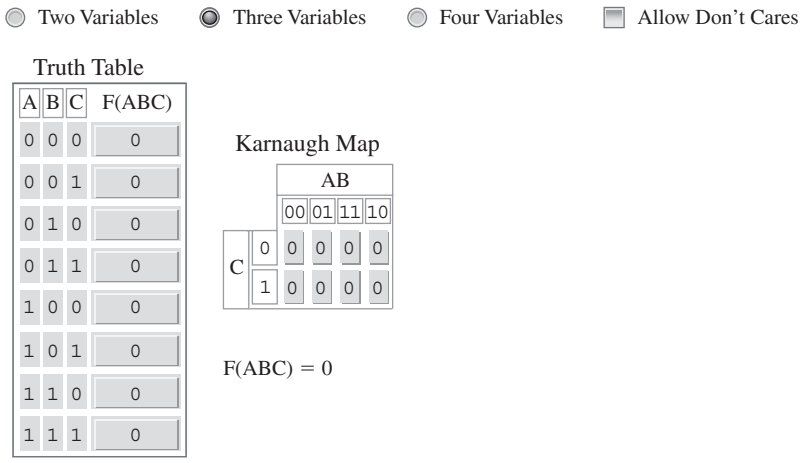


FIGURE 2.7 The screen (or GUI) for Karnaugh Map Explorer for three variables (refer to the color image in the back of the book)

The program always opens for three variables: *A*, *B*, and *C*. To change to two variables, click the radio button for Two Variables, or to change to four variables, click the radio button for Four Variables. Let's begin with two variables, so click the radio button for Two Variables. The screen for two variables is shown in Figure 2.8.

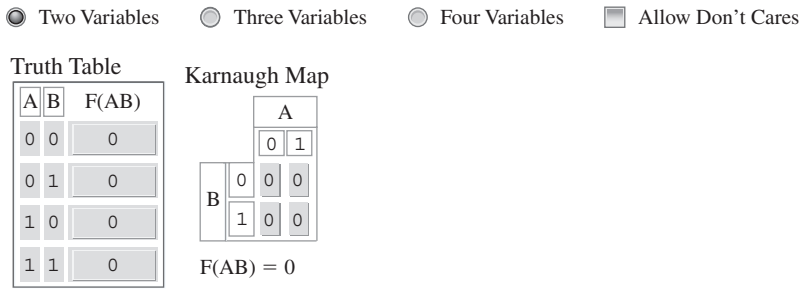


FIGURE 2.8 The screen (or GUI) for Karnaugh Map Explorer for two variables (refer to the color image in the back of the book)

Observe that Karnaugh Map Explorer has three major parts. These are the truth table, the K-map, and the function. When the radio button is selected for Two Variable, the program is designed to reduce any function that has two-input variables—that is, input variables *A* and *B*. The function that is shown in Figure 2.8 is $F(AB) = 0$. If you move the cursor over the 0 in the function, you will observe that the 0 is highlighted in yellow. You will also observe that all the rows in the truth table for the function that are 0 are also highlighted in yellow, and all the cells in the K-map that are 0 also highlighted in yellow. This provides an aid in learning how the values for the rows in a truth table and the values in the cells in a K-map are related to the value of the function.

2.5.2 Using a 2-Variable K-Map

Figure 2.9a shows a 2-variable K-map plotted (the cells in the map are filled in) for a 2-input AND gate. To plot the 2-variable K-map, simply select (click) the output $F(AB)$ for row $A\ B = 11$ (minterm 3) in the truth table to change the output to 1, and observe that the row is highlighted in red. Observe that the corresponding cell (minterm 3) in the K-map is changed to 1 and is also highlighted in red, and the function $F(AB) = A\ B$ is the result. If you move the cursor over $A\ B$ in the function, you will observe that $A\ B$ is highlighted in yellow, and the row for minterm 3 in the truth table and the cell for minterm 3 in the K-map are also highlighted in yellow as shown in Figure 2.9b.

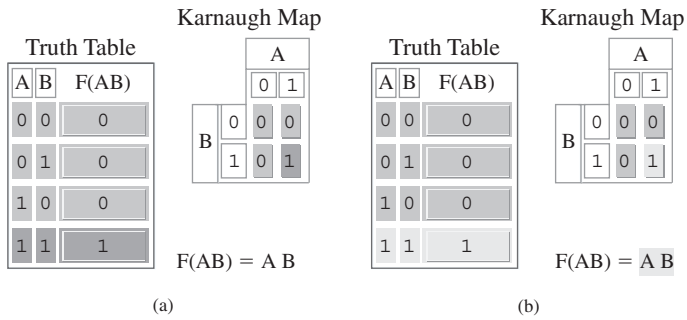


FIGURE 2.9 (a) The screen (or GUI) for Karnaugh Map Explorer for a 2-input AND gate; (b) highlighted areas in yellow in the truth table and K-map relate to the value of the function (refer to the color image in the back of the book)

Also observe in Figure 2.9 that $F(AB) = A\ B$ represents $F(A,B) = A \cdot B$. The program does not show a comma between the variables in a function, and the program uses implied ANDing.

Figure 2.10 shows a 2-variable K-map plotted for a 2-input OR gate. To plot the 2-variable K-map, simply select (click) the output $F(AB)$ for rows $A\ B = 01$, $A\ B = 10$, and $A\ B = 11$ to change the outputs to 1, and observe that the rows are highlighted in red. Observe that the corresponding cells in the K-map are changed to 1 and are highlighted in red, and the function $F(AB) = A + B$ is the result.

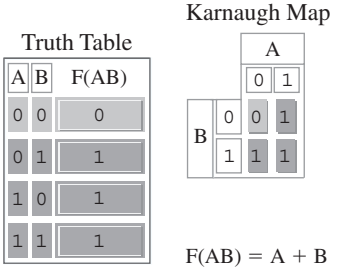


FIGURE 2.10 The screen (or GUI) for Karnaugh Map Explorer for a 2-input OR gate (refer to the color image in the back of the book)

Place the cursor over variable A in the function to observe the minterms that generate variable A in the truth table and the cells that generate variable A in the K-map. Variable A in the function, the minterms that generate variable A in the truth table, and the cells that generate variable A in the K-map are all highlighted in yellow as shown in Figure 2.11a. Likewise, place the cursor over variable B in the function to observe the minterms that generate variable B in the truth table and the cells that generate variable B in the K-map. Variable B in the function, the minterms that generate variable B in the truth table, and the cells that generate variable B in the K-map are highlighted in yellow, as shown in Figure 2.11b.

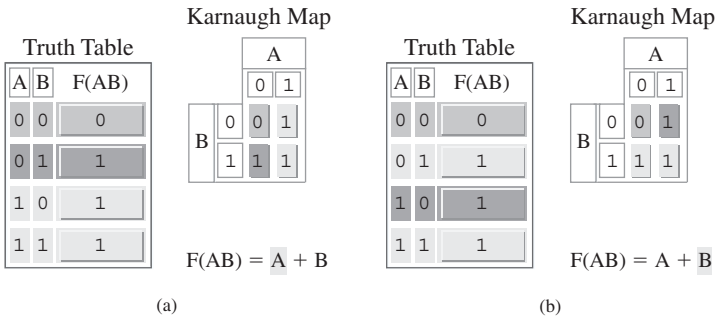


FIGURE 2.11 (a) Minterms that generate variable *A*; (b) minterms that generate variable *B* (refer to the color image in the back of the book)

The function for the OR gate can be represented in compact minterm form as $F(A,B) = \Sigma m(1,2,3)$ or in compact maxterm from as $F(A,B) = \Pi M(0)$. Using the same organization as Karnaugh Map Explorer, we can draw a manual K-map using paper and pencil as shown in Figure 2.12a. The minterm or maxterm numbers are labeled in each cell in the manual K-map. Using the manually drawn K-map, we can plot the minterms for the 1s of the function as shown in Figure 2.12b.

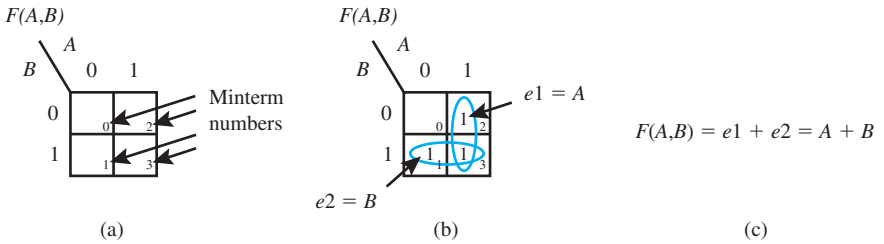


FIGURE 2.12 (a) Each cell identified in terms of its minterms; (b) minterms plotted and minterms grouped for the 1s of the function; (c) expression for each group and the reduced function

The expressions *e*1 and *e*2 are used to identify the groups of 1s that are circled in the K-map.

The values for the expressions are obtained by keeping the variable in each group that does not change over the area represented by the group. In Figure 2.12b, $e1 = A$, and because *B* has a 1-bit change in the group, *B* can be removed. In addition, $e2 = B$, and because *A* has a 1-bit change in the group, *A* can be removed. This can be verified by applying the Adjacency Theorem in each group so $e1 = (A \cdot \overline{B} + A \cdot B = A)$ and $e2 = (\overline{A} \cdot B + A \cdot B = B)$. A reduced function is obtained by ORing the expressions for each group as shown in Figure 2.12c.

Before moving on to 3- and 4-variable K-maps, you should fill in a truth table using Karnaugh Map Explorer for the NOR gate, the XOR gate, and the XNOR gate for practice. In each case, the function generated by the Karnaugh Map Explorer is written in sum of products (SOP) form in terms of expressions for the groups of 1s for the function as shown in Figure 2.13a, for a 2-input NAND gate.

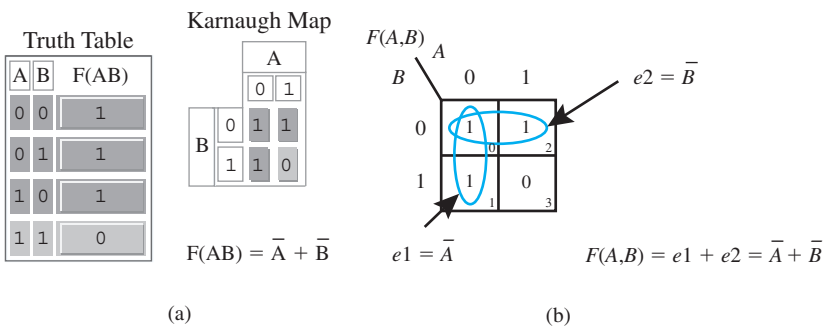


FIGURE 2.13 (a) The screen (or GUI) for Karnaugh Map Explorer for a 2-input NAND gate; (b) manual K-map reduction for a 2-input NAND gate (refer to the color image in the back of the book)

Figure 2.13b shows the manual K-map reduction for a 2-input NAND gate. Observe that $e1 = \overline{A}$, and because B has a 1-bit change in the group, B can be removed. Observe that $e2 = \overline{B}$, and because A has a 1-bit change in the group, A can be removed. Remember, when two cells with the same function values (1s in the case of Karnaugh Map Explorer) are physically adjacent to each other either vertically or horizontally, a reduced expression can always be obtained using the Adjacency Theorem ($e1 = (\overline{A}\cdot\overline{B} + \overline{A}\cdot B = \overline{A})$ and $e2 = (\overline{A}\cdot\overline{B} + A\cdot\overline{B} = \overline{B})$), because there is a 1-bit change from one cell to the next adjacent cell. A reduced function is obtained by ORing the expressions for each group as shown in Figure 2.13b.

2.5.3 Using a 3-Variable K-Map

A 3-variable K-map is shown in Figure 2.14 for a 3-input majority function such that the output is 1 any time two or more input variables are 1. The function can be written in compact minterm form as follows: $F(A,B,C) = \Sigma m(3,5,6,7)$.

FIGURE 2.14 The screen (or GUI) for Karnaugh Map Explorer for a 3-input majority function (refer to the color image in the back of the book)

Truth Table

A	B	C	F(ABC)
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

Karnaugh Map

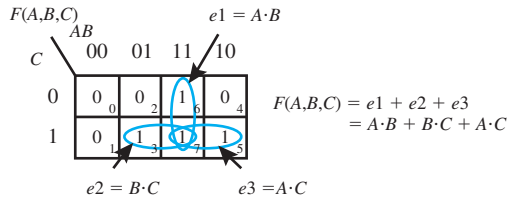
C \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

F(ABC) = A B + B C + A C

Look closely at the bit values for A B on the perimeter of the 3-variable K-map. Observe that the bits are arranged in Gray code counting sequence—that is, 00, 01, 11, 10. This sequence is necessary so that horizontal adjacent cells have a 1-bit change from one cell to the next adjacent cell, thus allowing the Adjacency Theorem to be applied. When grouping two adjacent cells that contain 1s, the variable that is removed by applying the Adjacency Theorem is the variable that has a 1-bit change. The remaining variables are ANDed together and represent the expression for the group. A reduced function is obtained by ORing the three expressions for each group. To identify each group of 1s in the function, move the cursor over each of the three ANDed expressions in the function one at a time, and observe the group that is highlighted in yellow.

Figure 2.15 shows a manual K-map reduction for a 3-input majority function.

FIGURE 2.15 Manual K-map reduction for a 3-input majority function



The expressions $e1$, $e2$, and $e3$ are used to identify the groups of 1s that are circled in the K-map. The values for the expressions are obtained by ANDing together the variables in each group that do not change over the area represented by the group. In Figure 2.15, $e1 = A\cdot B$, and because C has a 1-bit change in the group, C can be removed. In addition, $e2 = B\cdot C$, and because A has a 1-bit change in the group, A can be removed. Then, $e3 = A\cdot C$, and because B

has a 1-bit change in the group, B can be removed. A reduced function is obtained by ORing the expressions for each group as shown in Figure 2.15.

The 3-variable K-maps shown in Figures 2.16a and 16b have a group of two adjacent cells that contain 1s and also a group of four adjacent cells that contain 1s.

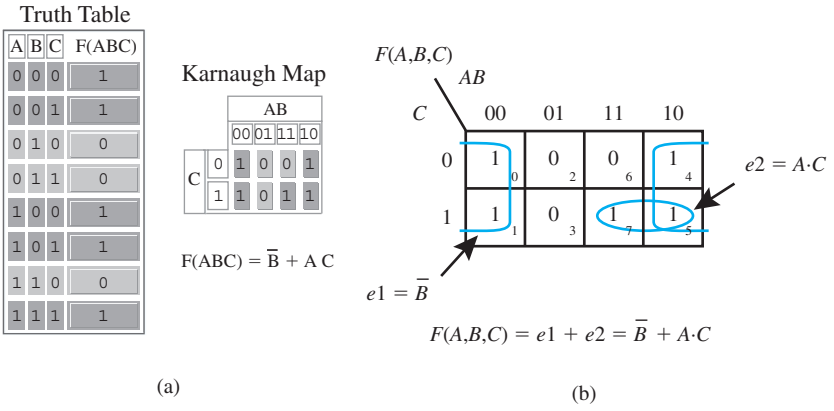


FIGURE 2.16 (a) The screen (or GUI) for Karnaugh Map Explorer with a group of four adjacent cells; (b) manual K-map reduction (refer to the color image in the back of the book)

In Figure 2.16b, observe that the cells 0, 1, 4, and 5 represent a group of four adjacent cells that contain 1s, because end-around adjacency is true for Karnaugh maps. The expression for cells 0, 1, 4, and 5 can be represented as $e1 = \overline{B}$, and because A and C each have a 1-bit change in the group, A and C can be removed. The cells for minterms 7 and 5 represent a group of two adjacent cells that contain 1s. The expression for cells 7 and 5 can be represented as $e2 = A{\cdot}C$, and because B has a 1-bit change in the group, B can be removed. The expressions for $e1$ and $e2$ are then ORed together. This example has shown us that we can also group four adjacent cells that contain 1s, as well as two adjacent cells that contain 1s, to obtain a reduced expression. A reduced function is obtained by ORing together the expressions for each group as shown in Figure 2.16b.

A 3-variable K-map is shown in Figure 2.17 for a 3-input XOR gate (or a 3-input ODD function).

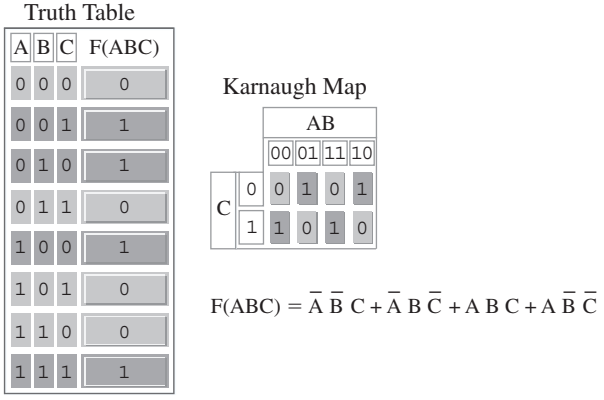


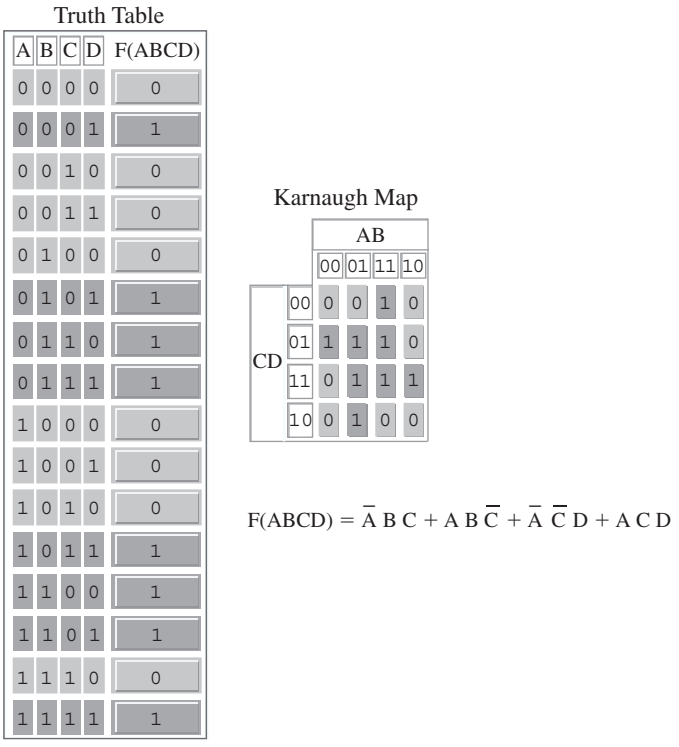
FIGURE 2.17 The screen (or GUI) for Karnaugh Map Explorer for a 3-input XOR gate (refer to the color image in the back of the book)

This function will not reduce to a simpler sum of products expression because none of the cells that contain a 1 are physically adjacent, so the Adjacency Theorem cannot be applied. Notice that an expression must be written for each cell that contains a 1, and then all the expressions must be ORed together to form the sum of products expression for the function. This shows us that each single isolated cell that contains a 1 must be included in a function, even though the cell does not provide a reduced expression. By using Boolean algebra, it can be shown that the function can be written using XOR operators as $F = A \oplus B \oplus C$.

2.5.4 Using a 4-Variable K-Map

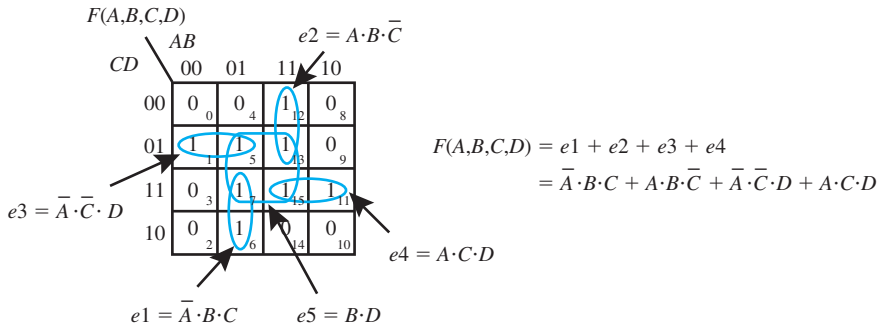
When you obtain a reduced function from a K-map, you may not obtain a minimum function unless you remember to cover the product terms for the smaller grouping of 1s first. Cover isolated cells, followed by two cells, followed by four cells, followed by eight cells that contain 1s. If a group of several smaller cells cover a larger group of larger cells, then it is not necessary to cover the larger group because the product term produced by the larger group would be redundant or unnecessary. The 4-variable K-map shown in Figure 2.18 illustrates the case of a redundant larger group of cells.

FIGURE 2.18 The screen (or GUI) for Karnaugh Map Explorer for a redundant larger group of cells (refer to the color image in the back of the book)



Notice in Figure 2.18 that the horizontal perimeter variables *AB* follow the Gray code counting sequence and the vertical perimeter variables *CD* also follow the Gray code counting sequence. This sequence is necessary so that both the horizontal adjacent cells and the vertical adjacent cells have a 1-bit change from one cell to the next adjacent cell, thus allowing the Adjacency Theorem to be applied. Figure 2.19 shows a manual K-map reduction with a redundant larger group of cells.

FIGURE 2.19 Manual K-map reduction for a 4-variable K-map with a redundant larger group of cells



In the manual K-map in Figure 2.19, the larger group of cell represented by the expression $e5$ is redundant because the four smaller groups of cells represented by the expressions $e1$ through $e4$ cover all the 1s in the larger group of cells. The redundant group is not included in the function because the function will contain more literals than necessary.

To obtain a minimum expression, the trick is to cover all the 1s using the largest possible groups of adjacent cells, while trying to do this with the fewest number of groups so that you obtain the function with the smallest literal count. It is important to keep in mind that occasionally there are alternate ways to cover the 1s and still obtain a minimum function.

2.5.5 Don't-Care Outputs

When a function has a specified output for all combinations of its inputs, the function has **completely** or **fully specified outputs**. When all the outputs of a function are not specified or are unimportant, the function has **incompletely specified outputs**. The output of an incompletely specified function may be given a third type of output value called a **don't-care output**. It is common practice to use the symbol × for a don't-care output in a truth table or a K-map.

The compact minterm form for a function with a don't-care output may be written as follows:

$F(A,B,C) = \Sigma m(0,1,6) + \Sigma md(2)$. The function has a 1 output for the minterms 0, 1, and 6, a don't care output for minterm 2, and a 0 output for all other minterms in the function.

The 3-variable K-map shown in Figure 2.20 is plotted for the function $F(A,B,C) = \Sigma m(0,1,6) + \Sigma md(2)$.

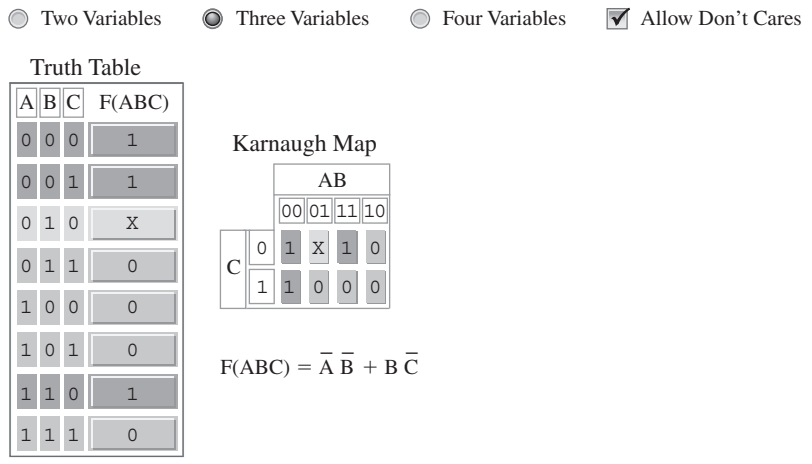


FIGURE 2.20 The screen (or GUI) for Karnaugh Map Explorer for a function with an incompletely specified, or don't-care, output (refer to the color image in the back of the book)

When plotting a function with an incompletely specified output you must check (click) the box Allow Don't Cares as shown in Figure 2.20. The don't-care output is obtained by clicking on minterm 2 in the truth table or on the corresponding K-map cell until an × appears. The × represents a don't-care output, which means its value may be a 0 or may be 1. Karnaugh Map Explorer chooses the value of × that provides the fewest literals for the function, which is what you also need to do if you are using a manual K-map to reduce a function. In Karnaugh Map Explorer, place the cursor on the expression $B \cdot \overline{C}$ to observe that × is grouped with the 1 at minterm 6. If × is not grouped with the 1 at minterm 6, the expression would be the same expression as minterm 6, or $A \cdot B \cdot \overline{C}$, which would increase the literal count. The proper use of × is to decrease the literal count, to minimize the function.

In Figure 2.21, a 4-variable K-map is shown plotted for a function that has several incompletely specified, or don't-care, outputs.

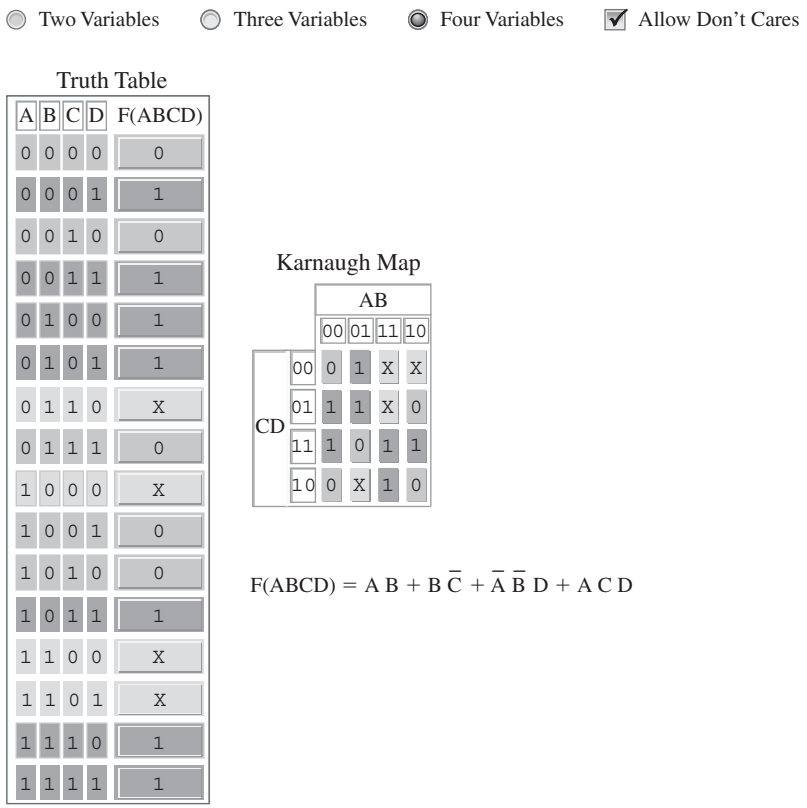


FIGURE 2.21 The screen (or GUI) for Karnaugh Map Explorer for a function with several incompletely specified, or don't-care, outputs (refer to the color image in the back of the book)

The compact minterm form for the function in Figure 2.21 may be written as $F(A,B,C,D) = \Sigma m(1,3,4,5,11,14,15) + \Sigma md(6,8,12,13)$. In Karnaugh Map Explorer, place the cursor on the expression $B \cdot \overline{C}$ to observe that the × at minterm 12 and the × at minterm 13 are both grouped with the 1 at minterm 4 and the 1 at minterm 5. The × at minterm 6 and the × at minterm 8 are not used to minimize the function, because they are not grouped with any 1s. This shows us that when we use a manual K-map to reduce a function, we should only use a don't-care output if it helps to decrease the literal count of the function.

When you use a manual K-map to reduce a function, **never cover don't-care outputs by themselves**, because a group of only don't-care outputs always represents a redundant expression or product term.

Karnaugh Map Explorer was written to group only the 1s to obtain a reduced function. In the manually drawn K-map shown in Figure 2.22, the 0s and the don't-care outputs for the function $F(A,B,C) = \Sigma m(0,1,6) + \Sigma md(5,7)$ are grouped to obtain a reduced function. The 1s and don't-care outputs are also grouped in Figure 2.22 so that you can see how the function is reduced in two different ways.

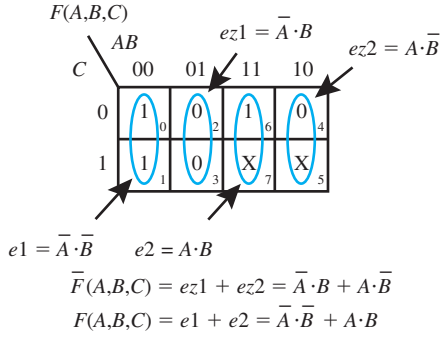


FIGURE 2.22 Grouping the 0s and don't-cares to obtain a reduced function and also grouping the 1s and don't-cares to obtain a reduced function

In Figure 2.22, each group of 0s is identified using *ez* (expression for zeros), while each group of 1s is identified using *e* (expression for ones). Observe that the function must be complemented when groups of 0s are ORed together to form the function, while the function is not complemented when groups of 1s are ORed together to form the function.

PROBLEMS

Section 2.2 Digital Circuits versus Analog Circuits

2.1 List the two primary types of signals.

2.2 Briefly describe the difference between analog and digital circuits. Would the wind be considered an analog or a digital measurement? Would a normal light switch in a room be considered an analog or digital control unit?

2.3 Explain the positive logic convention. Explain the negative logic convention. Which convention is used in this book?

Section 2.3 Binary Number Conversions

2.4 Name four number systems and their corresponding bases.

2.5 Write the value for the digit position in each of the following decimal numbers.
a. Digit position d_0 in 123
b. Digit position d_3 in 64087
c. Digit position d_2 in 39283

2.6 Write the values for the bit positions in the binary number 10001001.
a. Bit position b_0
b. Bit position b_3
c. Bit position b_6

2.7 What does 0 referencing mean? List the range for a count of 16 items using 0 referencing.

2.8 Use the subtract the weights method to convert the following decimal integers to binary.
a. 17
b. 37
c. 56

2.9 Convert the following binary numbers to equivalent octal numbers using the groups of 3 method.
a. 10001010
b. 11110000
c. 10000001

2.10 Convert the following octal numbers to equivalent binary numbers using the groups of 3 method in reverse.
a. 24
b. 754
c. 463

2.11 Convert the following binary numbers to equivalent hexadecimal numbers using the groups of 4 method.
a. 10001010
b. 11110000
c. 10000001

2.12 Convert the following hexadecimal numbers to equivalent binary numbers using the groups of 4 method in reverse.
a. E
b. 14
c. 234

2.13 Convert the following binary numbers to octal using the groups of 3 method and also to hexadecimal using the groups of 4 method.
a. 10001010
b. 11110000
c. 100000011

2.14 Construct a table with a column for hexadecimal numbers, and list the numbers 00 through 20. Make a column for binary numbers, and list the equivalent binary numbers.

2.15 Convert the following binary numbers to equivalent decimal numbers using the polynomial function method.
a. 11011
b. 110110
c. 1101101

2.16 Convert the following octal numbers to their decimal equivalents using the polynomial function method.
a. $(27)_8$
b. $(471)_8$
c. $(1234)_8$

2.17 Convert the following hexadecimal numbers to their decimal equivalents using the polynomial function method.
a. $(3A)_{16}$
b. $(D1B)_{16}$
c. $(E2B4)_{16}$

2.18 Convert the following binary numbers to their decimal equivalents using the factored form method.
a. $(10011)_2$
b. $(110010)_2$
c. $(1110001)_2$

2.19 Convert the following octal numbers to their decimal equivalents using the factored form method.
a. $(35)_8$
b. $(542)_8$
c. $(1361)_8$

2.20 Convert the following hexadecimal numbers to their decimal equivalents using the factored form method.
a. $(B3)_{16}$
b. $(12D)_{16}$
c. $(F2B4)_{16}$

2.21 Convert the following decimal numbers to their binary equivalents using the repeated radix division method (double-dabble method).
a. $(58)_{10}$
b. $(137)_{10}$
c. $(253)_{10}$

2.22 Convert the following decimal numbers to their octal equivalents using the repeated radix division method (octal-dabble method).
a. $(92)_{10}$
b. $(248)_{10}$
c. $(814)_{10}$

2.23 Convert the following decimal numbers to their hexadecimal equivalents using the repeated radix division method (hex-dabble method).
a. $(96)_{10}$
b. $(492)_{10}$
c. $(1678)_{10}$

Section 2.4 Binary Codes

2.24 Determine the minimum number of bits that are required to represent the following number of keys on a keypad or keyboard.
a. 27
b. 36
c. 132

2.25 Determine the number of keys that can be represented on a keypad or keyboard with the following number of bits.
a. 6
b. 7
c. 8

2.26 BCD is an abbreviation for binary coded decimal. In this code, straight binary is used to represent the decimal numbers 0 through 9 with a minimum number of bits (four). XS3 is an abbreviation for excess 3. In this code, straight binary + 0011 is used to represent the decimal numbers 0 through 9 with a minimum number of bits (again four). Make a table for the decimal numbers 0 through 9, the BCD representation for each decimal number, and the XS3 representation for each decimal number.

2.27 Which code in Table 2.1 (Section 2.4.2) in the text is considered easiest for humans to understand? Explain your answer.

2.28 Which code in Table 2.1 (Section 2.4.2) in the text is an error-detecting code? Explain your answer.

2.29 Which character code is the most commonly used code for PCs? In the extended version of the code, how many bits does the code require?

2.30 Look in the ASCII table (Table 2.2) in Section 2.4.2, and record the ASCII character for each of the following binary bit patterns.
a. 0011111
b. 1101010
c. 0110000

2.31 Look in the ASCII table (Table 2.2) in Section 2.4.2, and write the binary code and its hexadecimal equivalent for the following ASCII characters.
a. A
b. >
c. \

2.32 Write the code sequence that is being sent to the computer in hexadecimal when an ASCII keyboard is used to type the following words. Hint: Use Table 2.2.
a. Having
b. Fun
c. Yet?

2.33 Convert the following decimal numbers to binary numbers to reflective Gray code numbers. Write the results in explicit notation with the binary numbers specified as B and the reflective Gray code numbers specified as RGC.
a. 10
b. 19
c. 25

2.34 Convert each of the following binary numbers to equivalent reflective Gray code numbers. Write the results in explicit notation with the binary numbers specified as B and the reflective Gray code numbers specified as RGC.
a. 101
b. 0110
c. 10111

2.35 Write the Boolean functions for the reflective Gray code bits g_3, g_2, g_1, and g_0 in terms of the binary bits b_3, b_2, b_1, and b_0 for the following binary numbers.
a. $(b_1\ b_0)_{2\,\mathrm{B}}$
b. $(b_2\ b_1\ b_0)_{2\,\mathrm{B}}$
c. $(b_3\ b_2\ b_1\ b_0)_{2\,\mathrm{B}}$

2.36 Convert each of the following reflective Gray code numbers to equivalent binary numbers.
a. 011
b. 1011
c. 01100

2.37 Write the Boolean functions for the binary bits b_3, b_2, b_1, and b_0 in terms of the reflective Gray code bits g_3, g_2, g_1, and g_0 for the following reflective Gray code numbers.
a. $(g_1\ g_0)_{2\,\mathrm{RGC}}$
b. $(g_2\ g_1\ g_0)_{2\,\mathrm{RGC}}$
c. $(g_3\ g_2\ g_1\ g_0)_{2\,\mathrm{RGC}}$

2.38 Show the circuit for a group of XOR gates that will convert any binary number with 8 bits to a reflective Gray code number. First write the Boolean functions for the reflective Gray code bits g_7, $g_6 \ldots$, g_0 in terms of the binary bits b_7, $b_6 \ldots$, b_0, then draw the circuit using the Boolean functions.

2.39 Show the circuit for a group of XOR gates that will convert any reflective Gray code number with 5 bits to a binary number. First write the Boolean functions for the binary bits b_4, $b_3 \ldots$, b_0 in terms of the reflective Gray code bits g_4, $g_3 \ldots$, g_0, then draw the circuit using the Boolean functions.

2.40 What is the minimum number of output bits required for each shaft-angle encoder with the following output angular resolution in degrees?
a. 22.50 degrees
b. 11.25 degrees
c. 2.8125 degrees

2.41 Determine the 7-segment code for a common-cathode display with the segments *g*, *f*, *e*, *d*, *c*, *b*, and *a* assigned as bits $b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$, where a segment is lighted when a code bit is 1, for the following symbols.
a. Uppercase letter E
b. Lowercase letter b
c. Uppercase letter C

2.42 Determine the 7-segment code for a common-anode display with the segments *g*, *f*, *e*, *d*, *c*, *b*, and *a* assigned as bits $b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$, where a segment is lighted when a code bit is 0, for the following symbols.
a. Uppercase letter A
b. Lowercase letter b
c. Lowercase letter c

2.43 In the schematic for the letter display system in the text, which module generates the code for the 7-segment display?

2.44 In the schematic for the letter display system in the text, the letter display decoder has active high outputs. Explain what this means.

2.45 In the schematic for the letter display system in the text, the 7-segment display has active low inputs. Explain what this means.

2.46 In the schematic for the letter display system in the text, why is an array of NOT gates used to invert the outputs from the letter display decoder?

2.47 How many separate displays does the 7-segment display contain in the letter display system discussed in the text?

2.48 Describe how display 0 can be turned on in the letter display system discussed in the text.

2.49 Describe how display 1 can be turned off in the letter display system discussed in the text.

2.50 Describe how display 2 can be turned off in the letter display system discussed in the text.

2.51 What does the term *annotated schematics* or *annotated circuit diagram* mean?

2.52 The letter display system in the text must be changed. Table P2.52 shows a truth table for a new letter display decoder that generates the letter n (for negative) when a 0 is supplied to the input of the letter display decoder. When a 1 is supplied to the input, the letter that will be lighted is P (for positive). Write the output functions for the letter display decoder in Table P2.52.

Letter display decoder (active high outputs)

*S*0	*B*7	*B*6	*B*5	*B*4	*B*3	*B*2	*B*1	*B*0
0	0	1	0	1	0	1	0	0
1	0	1	1	1	0	0	1	1

7-segment display

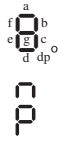


TABLE P2.52

2.53 The letter display system in the text must be changed. The new letter display decoder must generate the letter b (for back) when a 0 is supplied to its input. When a 1 is supplied to its input, the letter that must be lighted is F (for forward). To simplify the design, write the truth table for the new letter display decoder using active low outputs so the array of NOT gates can be removed. After you obtain the truth table, write the output functions for the new letter display system.

2.54 Suppose we wanted to change the letter display system in the text to a digit display system. When 0 is supplied to the input of the digit display decoder, the digit that will be lighted is 0, and when a 1 is supplied to the input the digit that will be lighted is a 1. Write the truth table for the digit display decoder using active high outputs.

2.55 Suppose we wanted to change the letter display system to a simpler digit display system. When 0 is supplied to the input of the digit display decoder, the digit that will be lighted is 0, and when a 1 is supplied to the input the digit that will be lighted is a 1. Write the truth table for the digit display decoder using active low outputs. How does this change simplify the design compared to the letter display system it will replace?

2.56 Write complete VHDL code for a simpler letter display system. Table P2.56 shows the truth table for the new letter display decoder. Hint: The array of NOT gates must be removed.

Letter display decoder (active low outputs)

*S*0	*B*7	*B*6	*B*5	*B*4	*B*3	*B*2	*B*1	*B*0
0	1	0	1	0	0	0	0	1
1	1	1	0	0	0	0	0	1

7-segment display

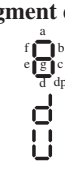


TABLE P2.56

2.57 What is a scalar in VHDL?

2.58 What is a vector in VHDL? What is another name for a vector?

2.59 How is the number of wires specified in a vector? Provide the vector specification in VHDL for five wires.

2.60 What design approach was use in Listing 2.1 in the text for the letter display system? How does this design approach handle each module of the design.

2.61 Write the correct VHDL code for the following Boolean constants: $F1 = 0$, $F2 = 1$, $F3 = 01$, and $F4 = 101$

Section 2.5 Karnaugh Map Reduction Method

2.62 Why is it important to minimize or reduce Boolean equations that make up digital circuits?

2.63 What is an algorithm?

2.64 Name the method that is used in this chapter to reduce Boolean functions.

2.65 What is a literal?

2.66 Why is it important to reduce the literal count of Boolean expressions?

2.67 What is a K-map?

2.68 What is the name of the theorem that applies when grouping adjacent 1s in a K-map?

2.69 In what form is a function generated by Karnaugh Map Explorer?

2.70 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake.
a. $F(A,B) = \Sigma m(2,3)$
b. $F(A,B) = \Pi M(1,3)$

2.71 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake.
a. $F(A,B) = \Sigma m(0,2,3)$
b. $F(A,B) = \Pi M(1,2,3)$

2.72 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake.
a. $F(A,B) = \Sigma m(0,3)$
b. $F(A,B) = \Sigma m(1,2)$

2.73 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake.
a. $F(A,B,C) = \Sigma m(0,1,4,5)$
b. $F(A,B,C) = \Sigma m(0,2,3,6,7)$
c. $F(A,B,C) = \Pi M(2,3)$

2.74 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake.
a. $F(A,B,C) = \Pi M(0,2,3)$
b. $F(A,B,C) = \Pi M(1,3,5)$
c. $F(A,B,C) = \Sigma m(1,3,4,6)$

2.75 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake. It is important to keep in mind that occasionally there are alternate ways to cover the 1s and still obtain a minimum function.
a. $F(A,B,C,D) = \Sigma m(2,3,6,7,8,9,12,13)$
b. $F(A,B,C,D) = \Pi M(2,4,6,8,10)$
c. $F(A,B,C,D) = \Pi M(0,2,5,8,10,13)$

2.76 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake.
a. $F(A,B,C,D) = \Pi M(0,1,4,5,9,10,12,13,14,15)$
b. $F(A,B,C,D) = \Sigma m(0,1,2,4,5,8,9,12,13,14,15)$
c. $F(A,B,C,D) = \Sigma m(0,2,5,7,8,10,13,15)$

2.77 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake.
a. $F(A,B,C) = \Sigma m(0,7) + \Sigma md(1)$
b. $F(A,B,C) = \Sigma m(1,5) + \Sigma md(0,2,4)$
c. $F(A,B,C) = \Pi M(4,6)\cdot\Pi Md(1,5)$

2.78 Plot each of the following functions on a manual K-map. Circle each adjacent group of 1s to identify each expression. Write the reduced functions in SOP form. Then check your result using Karnaugh Map Explorer. If you make an error, learn from your mistake.
a. $F(A,B,C,D) = \Sigma m(2,3,4,7,11) + \Sigma md(5,15)$
b. $F(A,B,C,D) = \Pi M(4,10)\cdot\Pi Md(2,6,8)$
c. $F(A,B,C,D) = \Sigma m(0,2,5,7,8,10,13) + \Sigma md(4,6,14)$

2.79 Plot each of the following functions on a manual K-map. Circle each adjacent group of 0s to identify each expression. Write the reduced functions in SOP form.
a. $F(A,B,C) = \Sigma m(0,7) + \Sigma md(1)$
b. $F(A,B,C) = \Sigma m(1,5) + \Sigma md(0,2,4)$
c. $F(A,B,C) = \Pi M(4,6)\cdot\Pi Md(1,5)$

2.80 Plot each of the following functions on a manual K-map. Circle each adjacent group of 0s to identify each expression. Write the reduced functions in SOP form.
a. $F(A,B,C,D) = \Pi M(0,1,6,8,9,10,12,13,14)\cdot\Pi Md(5,15)$
b. $F(A,B,C,D) = \Pi M(4,10)\cdot\Pi Md(2,6,8)$
c. $F(A,B,C,D) = \Sigma m(0,2,5,7,8,10,13) + \Sigma md(4,6,14)$

Chapter 3

Introduction to Logic Circuit Analysis and Design

Chapter Outline

3.1 INTRODUCTION

In this chapter, you will learn about integrated circuit devices, how to analyze and design logic circuits, and how to generate detailed schematic diagrams. You will learn how to manually design circuits in AND/OR form, OR/AND form, NAND/NAND form, and NOR/NOR form. All logic circuits have a delay time, so we will discuss how to determine the worst-case delay time through a circuit. Two very important logic devices—decoders and multiplexers—are introduced. Simple procedures are presented for manually designing digital circuits with decoders and also with multiplexers. Function hazards and logic hazards that generate glitches that can cause circuits to fail are covered. The design of most of the circuits is followed by a listing that shows the complete VHDL design of the circuits using Boolean equations.

3.2 INTEGRATED CIRCUIT DEVICES

There are many manufacturers that provide physical hardware devices called **integrated circuits (ICs)** that are capable of carrying out two-valued Boolean functions. These devices can contain tens to literally millions of transistors on a small silicon semiconductor crystal called a die or chip. Because the circuitry contains mainly transistors, diodes, and resistors, which are all interconnected inside the chip, power consumption can be quite low and reliability quite high. The die is constructed and then welded to a frame as illustrated in Figure 3.1. Its input

and output leads are connected by thin gold wires to the package's leads or pins. The unit is encapsulated using glass, ceramic, or plastic. Finally, the unit is hermetically sealed. ICs that are hermetically sealed guard against die contamination in many different environments.

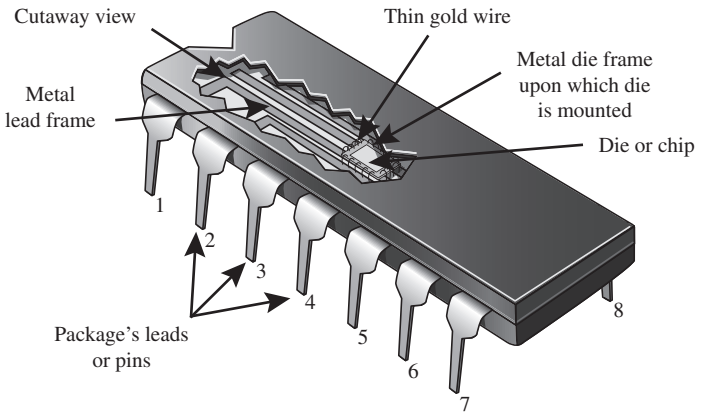


FIGURE 3.1 Cutaway view of an IC package showing the die or chip, the die frame, the gold wire, the lead frame, the package's leads or pins, and the pin numbers or pin outs

Five different types of integrated circuit packages are shown in Figure 3.2.

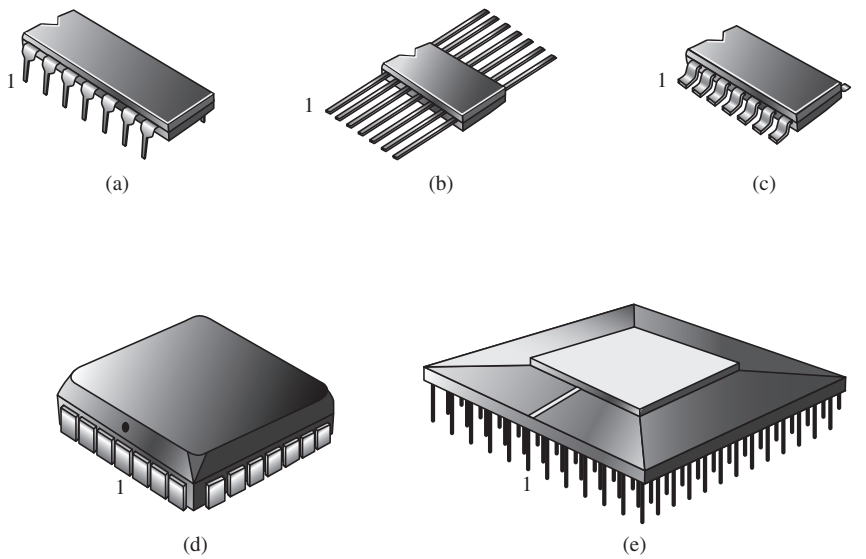


FIGURE 3.2 Packages for integrated circuits: (a) dual-in-line package; (b) flat package; (c) surface mounting package; (d) plastic leaded chip carrier package; (e) pin grid array package

The package shown in Figure 3.2a is the common **dual-in-line (DIP) package**. The packages shown in Figures 2b and c are the **flat package** (**flat pack**) and the **surface mount (small outline) package**. These packages are generally used in applications in which real estate on a **printed circuit board** (**PCB**) is critical and/or a lower cost must be achieved for high volume application. The packages shown in Figures 2d and e are the **plastic leaded chip carrier** (**PLCC**) **package** and the **pin grid array** (**PGA**) **package**, respectively, which are used for very large IC designs especially when the pin count—that is, the package inputs and outputs—for the designs become very large. Note the location of pin 1 for each package type. The integrated circuit packages shown in Figure 3.2 are only a few among many different types of packages available. For very large integrated circuit devices, a newer package is available called a **ball grid array (BGA) package** (the BGA package is not shown in Figure 3.2). The BGA has balls of solder on its pins that are soldered directly to a PC board. Manufacturers have a website that provides the data sheets for their parts. The data sheets provide a list of IC packages available, so engineers can choose the ones they prefer to use.

3.3 ANALYZING AND DESIGNING LOGIC CIRCUITS

We call **circuit analysis** the process of obtaining a Boolean function for a schematic or a circuit diagram. We call **circuit design** or **synthesis** the process of obtaining a schematic or a circuit diagram for a Boolean function. Keep in mind that the schematics or circuit diagrams we will cover are combinational or combinatorial logic circuits; that is, the outputs of these circuits depend only on the external inputs applied to the circuits. Combinational logic circuits do not have feedback (the outputs are never fed back as inputs), and they do not have memory capability.

3.3.1 Analyzing and Designing Relay Logic Circuits

Before we introduce the analysis/design process using IC logic circuits, let's first analyze a logic switching circuit that uses relays. Switching circuits of this type are used in heavy power equipment. Figure 3.3a shows a physical representation for relay contacts that are normally closed (n.c.) and its corresponding symbol. Figure 3.3b shows a physical representation for relay contacts that are normally open (n.o.) and its corresponding symbol.

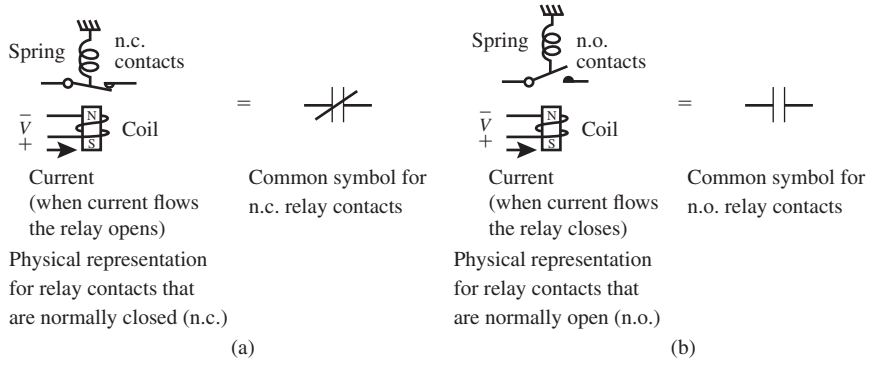


FIGURE 3.3 Relay contacts: (a) physical representation and common symbol for normally closed (n.c.) relay contacts and (b) physical representation and common symbol for normally open (n.o.) relay contacts

A logic switching circuit that uses relays connected up to a drive a motor is shown in Figure 3.4.

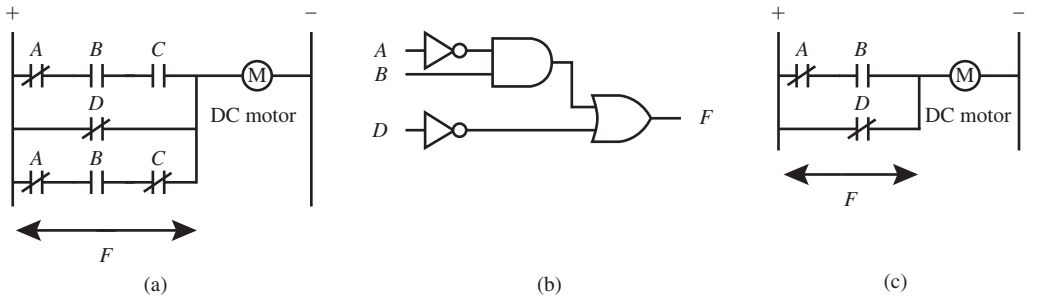


FIGURE 3.4 Relay logic switching circuits: (a) original relay circuit; (b) minimized symbolic logic circuit; (c) minimized relay circuit

Logic switching circuits used in power applications are generally referred to as **ladder logic circuits**. This term is used because it refers to the physical layout of the circuit (it looks like a ladder that may be climbed). In the switching circuit in Figure 3.4a, when F is 1, the motor M is turned on. Otherwise, the motor is turned off—that is, $F = 0$. To analyze relay logic circuits, one must remember that relays or switches connected in series provide an AND operation, while relays or switches connected in parallel provide an OR operation. The same series and parallel principles apply to transistor circuits, which you will study in a later course—that is, transistors connected in series provide an AND operation while transistors connected in parallel provide an OR operation.

Analyzing the circuit in Figure 3.4a, we obtain the following Boolean equation for the circuits: $F = \overline{A}\cdot B\cdot C + \overline{D} + \overline{A}\cdot B\cdot\overline{C}$. The expression $\overline{A}\cdot B\cdot C$ indicates that signal A must be 0 AND signal B must be 1 AND signal C must be 1, so $F = 1$ to turn the motor on; OR the expression $\overline{D}$

indicates that the signal D must be 0, so $F = 1$ to turn the motor on; OR the expression $\overline{A}\cdot B\cdot\overline{C}$ indicates that the signal A must be 0 AND the signal B must be 1 AND the signal C must be 0, so $F = 1$ to turn the motor on.

Applying Boolean algebra or using a K-map, the function F can be reduced to the following minimum form: $F = \overline{A}\cdot B + \overline{D}$. Using the minimum form of the function, we can design the circuit in symbolic form as shown in Figure 3.4b. The relay logic circuit drawn for the minimum form of the function is shown in Figure 3.4c. In general, logic circuits that use ICs may not be hefty enough to drive high-powered motors or electric light bulbs; however, logic circuits that use relays can be used for these applications.

To manually design a logic circuit using relays as illustrated in the last example, first obtain a minimum Boolean equation for the function. Next, substitute the appropriate relay type (either normally open or normally closed) for each input, and connect each relay either in series or in parallel depending on the form of the function. Connect the resulting relay circuit in series with an output device such as a motor, a lamp, or a control relay (CR). A control relay is represented by CR with a circle around it and signifies the coil of a relay. When a control relay is energized, its associated contacts close, thus allowing the CR to control another circuit. The voltage for a ladder logic circuit is applied across the vertical lines labeled + and −.

3.3.2 Analyzing IC Logic Circuits

Now let's analyze the symbolic logic circuits shown in Figure 3.5.

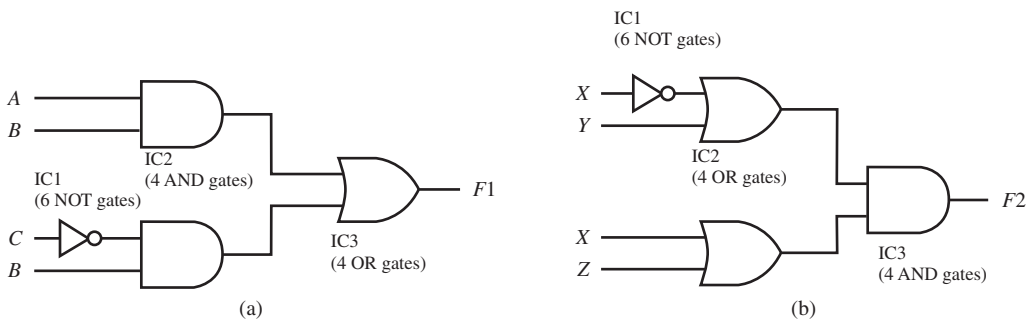


FIGURE 3.5 Circuits to be analyzed to obtain their Boolean functions

The circuit in Figure 3.5a is drawn in AND/OR form because AND gates are feeding into an OR gate. The circuit in Figure 3.5b is drawn in OR/AND form because OR gates are feeding into an AND gate. To analyze these circuits, we need to obtain the Boolean functions for $F1$ and $F2$. These circuits could represent relay logic circuits, or they could represent logic circuits constructed with ICs. In this section, we will assume that the circuits are constructed with ICs.

In Figure 3.5a, the function $F1$ is written by obtaining the outputs of IC2 (an AND gate with output $A\cdot B$) and IC2 (another AND gate with output $\overline{C}\cdot B$) and then obtaining the output of IC3 (an OR gate with output $F1 = A\cdot B + C\cdot B$ or $\overline{F1} = (\overline{A} + \overline{B})\cdot(C + \overline{B})$ via DeMorgan's Theorem). The function $F1 = A\cdot B + \overline{C}\cdot B$ is written in SOP form, while the function $\overline{F1} = (\overline{A} + \overline{B})\cdot(C + \overline{B})$ is written in POS form.

In Figure 3.5b, the function $F2$ is written by obtaining the outputs of IC2 (an OR gate with output $\overline{X} + Y$) and IC2 (another OR gate with output $X + Z$) and then obtaining the output of IC3 (an AND gate with output $F2 = (\overline{X} + Y)\cdot(X + Z)$ or $\overline{F2} = X\cdot\overline{Y} + \overline{X}\cdot\overline{Z}$ via DeMorgan's Theorem). The function $F2 = (\overline{X} + Y)\cdot(X + Z)$ is written in POS (product of sums) form, while the function $\overline{F2} = X\cdot\overline{Y} + \overline{X}\cdot\overline{Z}$ is written in SOP (sum of products) form.

As you can see from these examples, the analysis of small IC (or relay) logic circuits is fairly simple. All one needs to do is to write the output function of the circuit in terms of the input variables.

In practice, an IC number is assigned to each IC package on a printed circuit board (PCB). An IC package can contain more than one gate, as shown in Figure 3.5. **Small-scale integration (SSI)** packages are ICs packages that contain only a few gates. For example, each package may contain six Inverters or NOT gates, four 2-input OR gates, and four 2-input AND gates, respectively. An IC number is also assigned to large-scale integration packages such as a **complex programmable logic device (CPLD)** or a **field programmable gate array (FPGA)**. CPLDs contain hundreds to thousands of gates, while FPGAs contain thousands to millions of gates.

3.3.3 Designing IC Logic Circuits

When you manually design a circuit for a Boolean function, it is a good idea to first reduce the Boolean function. The easiest way to manually design an IC logic circuit is write the Boolean function in reduced SOP form and then draw the circuit in AND/OR form, or to write the Boolean function in reduced POS form and then draw the circuit in OR/AND form. The following four steps may help you draw the circuit:

Step 1: Draw the AND and OR gates for the reduced Boolean function.
Step 2: Show all signals as noncomplemented signal names.
Step 3: Interconnect the gates, including NOT gates where necessary.
Step 4: Cleanup or reduce the number of NOT gates.

Figure 3.6 shows the manual design of an IC logic circuit for the reduced SOP function $F1 = \overline{A} \cdot B \cdot \overline{C} + \overline{B} \cdot C + A \cdot \overline{B}$.

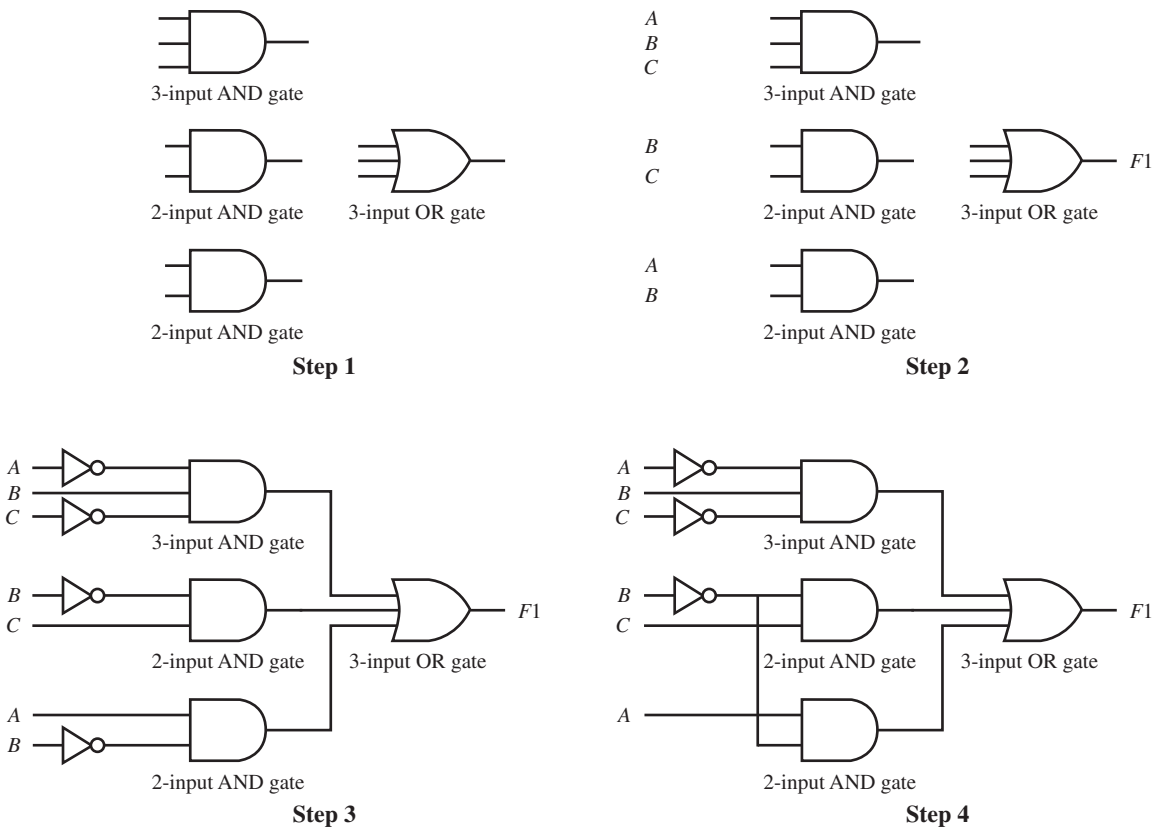


FIGURE 3.6 Manual design of an IC logic circuit for a reduced SOP function

Step 1: The AND and OR gates are drawn for the reduced function.
Step 2: All the input and output signals are shown as noncomplemented signal names.
Step 3: The gates are interconnected, including NOT gates where they were necessary.
Step 4: Cleanup was performed to remove one NOT gate.

Each IC for a particular logic family such as TTL (transistor transistor logic) and CMOS (complementary metal-oxide semiconductor) has a **fan-out**. The fan-out is the maximum number of inputs to which the IC output can be connected without electrically loading down the output. As long as the fan-out is not exceeded, the IC will function properly. The fan-out for the low-power Schottky (LS) TTL family is 20 (or 20 inputs). The original or standard TTL family has a fan-out of only 10 (or 10 inputs). Another name you should know is **fan-in**. Fan-in is the name used to describe the number of gate inputs—that is, the number of inputs that a gate has.

A **signal line** is a line drawn to an input line of a gate symbol, or a line drawn from an output line of a gate symbol. A **net** is the name used to describe signal lines that are connected together to carry the same signal. In Figure 3.6, signal lines with the same signal names are considered to be connected—that is, they belong to the same net. Be careful to label each signal line with only one name. If you were to wire up the circuit in Figure 3.6 in the laboratory, you would need to connect all signal lines together that have the same name.

In Figure 3.7, we show two alternate solutions for function $F1 = \overline{A}\cdot B\cdot\overline{C} + \overline{B}\cdot C + A\cdot\overline{B}$. Both solutions use a connection scheme for the input signal lines called a vertical-input scheme. Circuits drawn using a vertical-input scheme are usually more organized. With this scheme, one can draw large circuit designs in AND/OR form or OR/AND form quite rapidly. The vertical-input scheme eliminates Step 4 (cleanup) to reduce the number of NOT gates.

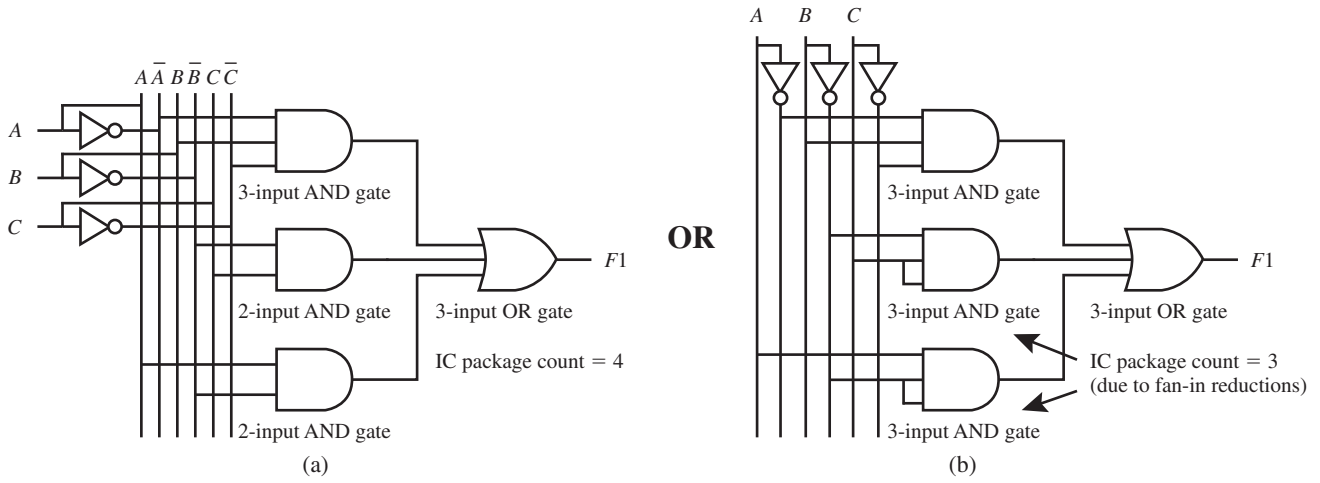


FIGURE 3.7 IC logic circuit designs for a minimum SOP form of a function using a vertical-input scheme

Notice in Figure 3.7a that the IC package count is four, but in Figure 3.7b the IC package count is only three, because a single IC package of three 3-input AND gates is used in the same package. After obtaining a circuit design for a minimum function, it is sometimes possible to build a smaller implementation of the circuit by using fewer ICs. This is done by connecting unused gate inputs to used gate inputs to reduce the fan-in of a gate, as shown in Figure 3.7b. This technique may be referred to as **fan-in reduction**. By using fan-in reduction, the 2-input AND gate IC package is not required.

Manually designing logic circuits is somewhat tedious. Using a hardware description language such as VHDL is a more efficient way to design logic circuits. Listing 3.1 shows a complete VHDL design for the function $F1 = \overline{A}\cdot B\cdot\overline{C} + \overline{B}\cdot C + A\cdot\overline{B}$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb1 is port (
     A, B, C : in std_logic;
     F1 : out std_logic
     );
end comb1;

architecture Boolean_function of comb1 is
begin
     F1 <= (not A and B and not C) or (not B and C) or (A and not B);
end Boolean_function;
```

LISTING 3.1 Complete VHDL design for the function $F1 = \overline{A}\cdot B\cdot\overline{C} + \overline{B}\cdot C + A\cdot\overline{B}$ (project: comb1)

For manual designs, OR gates with more than two inputs for some off-the-shelf logic families (such as TTL) are not available in ICs. This may pose a problem with the circuit designs in Figure 3.6 and Figure 3.7, which required a 3-input OR gate. Connecting (cascading) two 2-input OR gates in series to obtain a 3-input OR gate easily solves this problem as shown in Figure 3.8a.

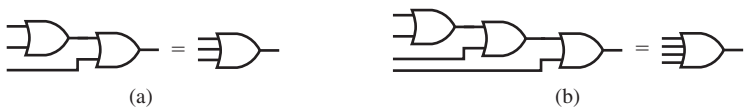


FIGURE 3.8 Cascading IC gates (a) 3-input OR gate (b) 4-input OR gate

Connecting three 2-input OR gates in series provides us with a 4-input OR gate as shown in Figure 3.8b. This solution also has a problem. The resulting cascaded circuit provides an output that responds more slowly to input changes as the number of cascaded stages is increased. We will consider this phenomenon a little later.

Consider the function $F2$ written in a minimum POS form as $F2 = (\overline{X} + Y)\cdot(X + \overline{Y})\cdot(X + \overline{Z})$. A circuit design for this function is shown in Figure 3.9 using a vertical-input scheme.

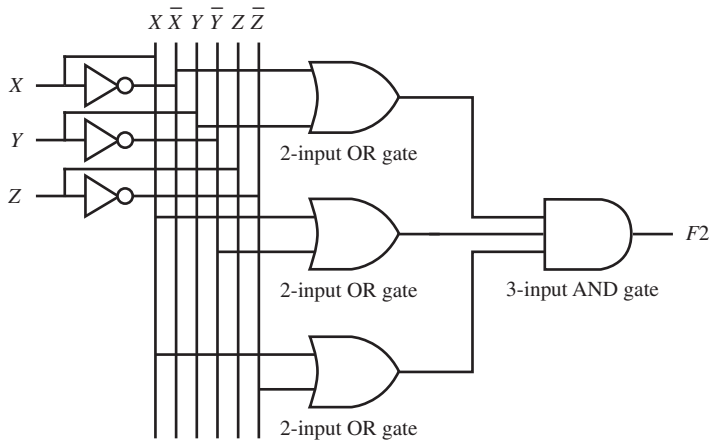


FIGURE 3.9 IC logic circuit design for a minimum POS form of a function using a vertical-input scheme

The main difference between the design of an SOP form (AND/OR form) of circuit and a POS form (OR/AND form) of circuit is the placement of the AND and the OR gates. For an SOP form of circuit, the AND gates feed into an OR gate—hence the name AND/OR form. For a POS form of circuit, the OR gates feed into an AND gate—hence the name OR/AND form.

Here is a more efficient way to design a logic circuit for function $F2$. Listing 3.2 shows a complete VHDL design for the function $F2 = (\overline{X} + Y)\cdot(X + \overline{Y})\cdot(X + \overline{Z})$.

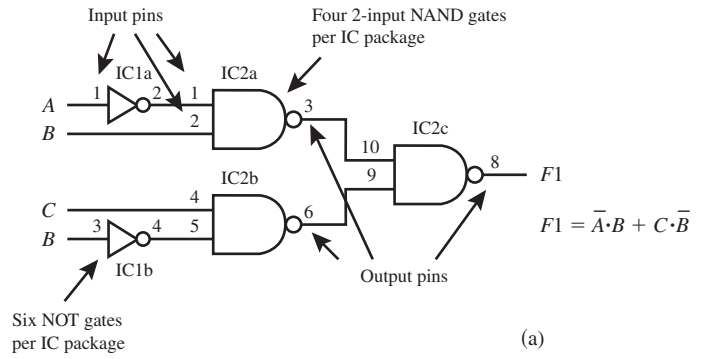
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb2 is port (
     X, Y, Z : in std_logic;
     F2 : out std_logic
     );
end comb2;

architecture Boolean_function of comb2 is
begin
     F2 <= (not X or Y) and (X or not Y) and (X or not Z);
end Boolean_function;
```

LISTING 3.2 Complete VHDL design for the function $F2 = (\overline{X} + Y)\cdot(X + \overline{Y})\cdot(X + \overline{Z})$ (project: comb2)

3.4 GENERATING DETAILED SCHEMATICS

All the circuits that we have drawn up to now are functional logic diagrams; that is, they are functionally correct but lack the details necessary to show the actual IC connections (or wiring) required to build a circuit on a PC board or in the lab. The circuit shown in Figure 3.10a is an example of a **detailed schematic** for the function $F1 = \overline{A}\cdot B + C\cdot\overline{B}$ using off-the-shelf advanced CMOS (complementary metal-oxide semiconductor) devices. The circuit shown in Figure 3.10a is in NAND/NAND form, which will be covered in the next section. Datasheets for Texas Instruments logic devices are available online at http://ti.com. When the ti window opens, select Logic, User Guides, GO; then click on Logic. Choose a logic family such as AC (Advanced CMOS); and select GO. When all the devices for that family appear, click on the device you want, then click on the Datasheet Icon. The datasheet provides the input pins, output pins, and power pins (V_{CC} and GND) for the package.

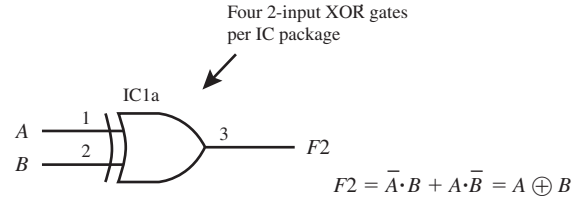
FIGURE 3.10
Detailed schematics: (a) circuit using multiple IC devices; (b) circuit using a single gate in an IC device



(a)

Reference designator	Part value	Pin number	
		V_{CC}*	GND
IC1	CD74AC04	14	7
IC2	CD74AC00	14	7

* + 5V



(b)

Reference designator	Part value	Pin number	
		V_{CC}*	GND
IC1	CD74AC86	14	7

* + 5V

In Figure 3.10b, we show a detailed schematic for the function $F2 = \overline{A}\cdot B + A\cdot\overline{B} = A \oplus B$. This IC has four XOR gates in the same package, and we are only using one. The following important items are necessary when drawing a detailed schematic:

1. Identify the part number for each IC in the circuit.
2. Show the pin numbers (also referred to as pin assignments) for all ICs in the circuit.
3. Show the power connections (V_{CC} and GND) for all the ICs in the circuit.

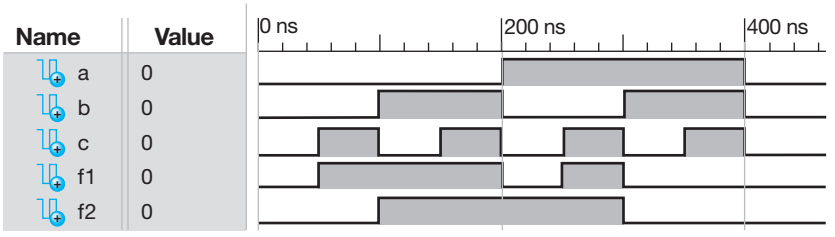
Many companies require their designers to provide detailed schematics so that an accurate record can be kept for each design. In addition, they require a written record to explain how the circuit works. This information is archived by companies so that they can keep complete and accurate records of their designs.

Listing 3.3 shows a complete VHDL design for the functions $F1 = \overline{A}\cdot B + C\cdot\overline{B}$ and $F2 = A \oplus B$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb3 is port (
     A, B, C : in std_logic;
     F1, F2 : out std_logic
     );
end comb3;

architecture Boolean_funtions of comb3 is
begin
     F1 <= (not A and B) or (C and not B);
     F2 <= A xor B;
end Boolean_funtions;
```

LISTING 3.3 Complete VHDL design for the functions $F1 = \overline{A}\cdot B + C\cdot\overline{B}$ and $F2 = A \oplus B$ (project: comb3)

Waveform 3.1 shows waveform diagrams for the VHDL design for the Boolean functions $F1 = \overline{A}\cdot B + C\cdot\overline{B}$ and $F2 = A \oplus B$.



WAVEFORM 3.1 Waveform diagrams for the VHDL design for the Boolean functions $F1 = \overline{A}\cdot B + C\cdot\overline{B}$ and $F2 = A \oplus B$

Note that function $F1$ is 1 when A is 0 and B is 1, $F1$ is also 1 when C is 1 and B is 0, and $F1$ is 0 for all other combinations of A, B, and C. Note that the function $F2$ is only 1 when A is not equal to B otherwise $F2$ is 0. The simulation shows that the VHDL design functions correctly.

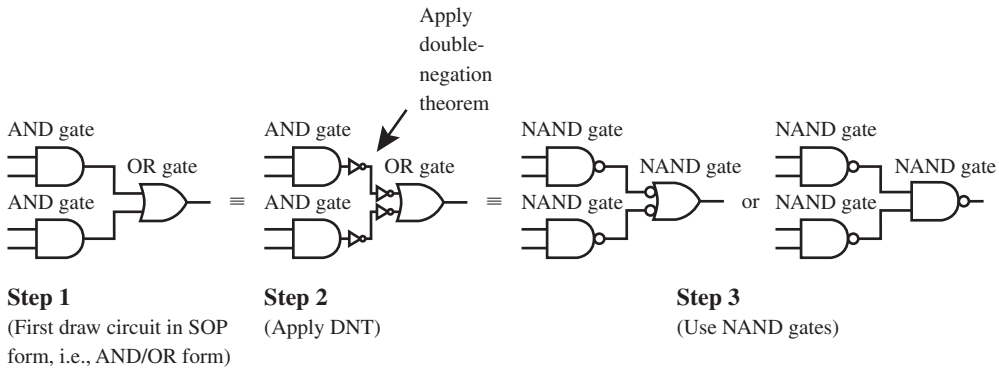
One of the nice things about using a hardware description language such as VHDL is that you do not have to draw detailed schematics to obtain a circuit on a system board such as the BASYS 2 board or the NEXYS 2 board, because the circuits are connected up internally via the bit pattern that is generated by the software. Only the external pin connections for each of the signals in the entity have to be declared. None of the pin connections for power have to be declared. Just apply power to the system board via the USB connector to the computer.

3.5 DESIGNING CIRCUITS IN NAND/NAND AND NOR/NOR FORM

In manual designs using discrete (or separate) IC devices, NAND and NOR gates are preferable to AND and OR gates. There are three reasons this is true: (1) NAND gates are generally faster than AND gates, and NOR gate are generally faster than OR gates in the same logic family; (2) NAND gates and NOR gates are available with a larger variety of fan-ins (gate inputs) to choose from than AND gates and OR gates; and (3) fewer IC packages are required to design circuits that use NAND gates and NOR gates because they are functionally complete gates, as discussed in Chapter 1.

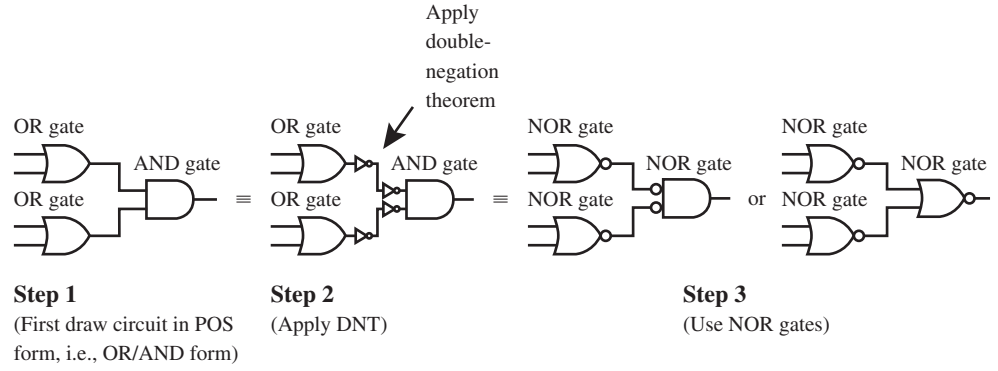
A procedure for manually designing a logic circuit in NAND/NAND form is shown in Figure 3.11. First, design the circuit in AND/OR form, and then convert the circuit into NAND/NAND form, as shown in Figure 3.11 using the **graphical design method for NAND/NAND form**.

FIGURE 3.11
Graphical Design Method for NAND/NAND form



A procedure for manually designing a logic circuit in NOR/NOR form is shown in Figure 3.12. First, design the circuit in OR/AND form, and then convert the circuit into NOR/NOR form, as shown in Figure 3.12 using the **graphical design method for NOR/NOR form**.

FIGURE 3.12
Graphical design method for NOR/NOR form



Consider the manual design of a circuit to implement the reduced Boolean function $F1 = \overline{A}\cdot\overline{B}\cdot\overline{C} + B\cdot C + A\cdot C$ in NAND/NAND form using the graphical design method. Since the function is already expressed in SOP form, we just have to draw the circuit in AND/OR form then convert the circuit to NAND/NAND form as shown in Figure 3.13.

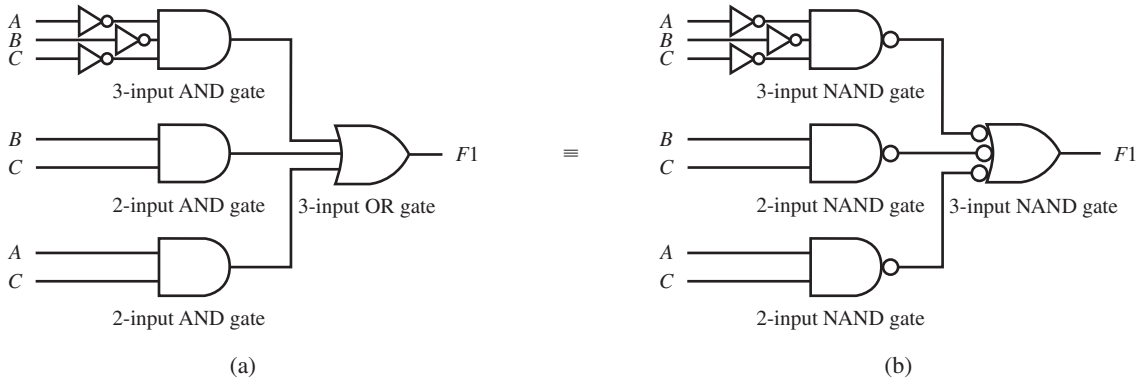


FIGURE 3.13 Converting a circuit from AND/OR form to NAND/NAND form

The application of the double-negation theorem (DNT) is not shown in Figure 3.13 because this can be done mentally without drawing all the NOT gate pairs.

Listing 3.4 shows a complete VHDL design for the function $F1 = \overline{A}\cdot\overline{B}\cdot\overline{C} + B\cdot C + A\cdot C$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb4 is port (
     A, B, C : in std_logic;
     F1 : out std_logic
     );
end comb4;

architecture Boolean_function of comb4 is
begin
     F1 <= (not A and not B and not C) or (B and C) or (A and C);
end Boolean_function;
```

LISTING 3.4 Complete VHDL design for the function $F1 = \overline{A}\cdot\overline{B}\cdot\overline{C} + B\cdot C + A\cdot C$ (project: comb4)

Consider the manual design of a circuit to implement the reduced Boolean function $F2 = \overline{X}\cdot\overline{Z} + X\cdot Z + Y$ in NOR/NOR form using the graphical design method. To follow the graphical design method, we must first express the function $F2$ in POS form, which is $\overline{F2} = (X + Z)\cdot(\overline{X} + \overline{Z})\cdot\overline{Y}$. Now the circuit must be drawn in OR/AND form and then converted to NOR/NOR form as shown in Figure 3.14.

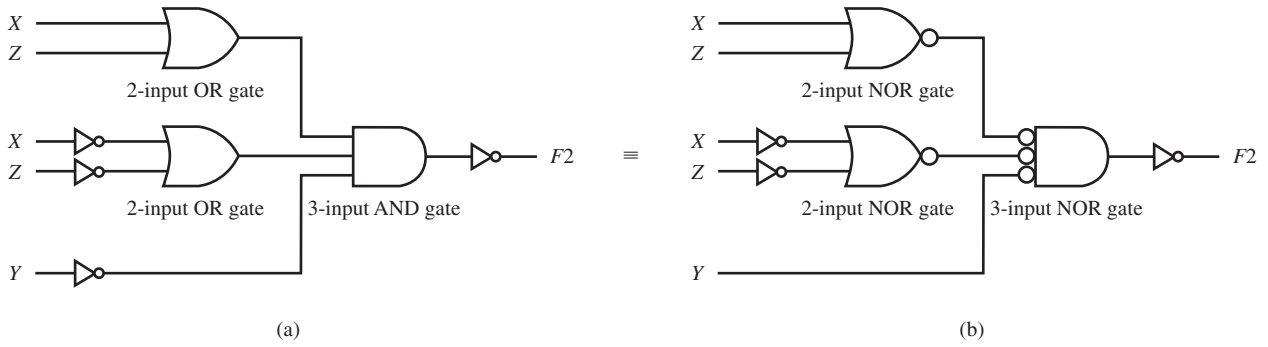


FIGURE 3.14 Converting a circuit from OR/AND form to NOR/NOR form

The application of the DNT is not shown in Figure 3.14 because this can be done mentally without drawing all the NOT gate pairs.

Listing 3.5 shows a complete VHDL design for the function $\overline{F2} = (X + Z)\cdot(\overline{X} + \overline{Z})\cdot\overline{Y}$.

LISTING 3.5
Complete VHDL design for the function $\overline{F2} = (X + Z)\cdot(\overline{X} + \overline{Z})\cdot\overline{Y}$ (project: comb5)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb5 is po.5rt (
     X, Y, Z : in std_logic;
     F2 : out std_logic
     );
end comb5;

architecture Boolean_function of comb5 is
begin
     F2 <= not ((X or Z) and (not X or not Z) and not Y);
               --POS form for F2
     --F2 <= (not X and not Z) or (X and Z) or Y;
               --This is an alternate description for F2
               --i.e., an SOP form for F2
end Boolean_function;
```

Reminder: A comment may be placed in VHDL code by using two hyphens in series, that is, --, as shown in Listing 3.5.

3.6 PROPAGATION DELAY TIME

Circuit delays are caused by signals passing through the components that make up the circuit. Worst-case delay is caused by a signal passing through the slowest delay path in the circuit. Each wire (or connector) and each gate has a **propagation delay time**, which is the time it takes a signal applied at its input to travel from the input to the output. The propagation delay time of a wire is dependent on its length and its cross-sectional area. It is also dependent on the material that the wire is made out of, such as copper, silver, and gold. Gold has the best conductivity, followed by silver and then copper. In terms of just the physical dimensions, a longer wire has a longer propagation delay time than a shorter wire, and a wire with a smaller cross-sectional area has a longer propagation delay time than a wire with a larger cross-sectional area. Remember that even wires have a propagation delay time that usually cannot be ignored in a circuit, if the circuit is operated at a very high frequency.

The abbreviation t_p is used as a relative measure of the time it takes for a signal to propagate through a gate. Figure 3.15 shows a NOT gate, an AND gate, and an OR gate with their symbols, their Boolean functions, and their propagation delay times. The propagation delay times are generally different because they are all different circuits. The propagation delay times for single-gate circuits is in the order of only a few nanoseconds. A nanosecond is one billionth of a second (or 10^{-9} seconds) and is the time it takes for electricity to travel through a length of wire approximately 1 foot or about $\frac{1}{3}$ meter.

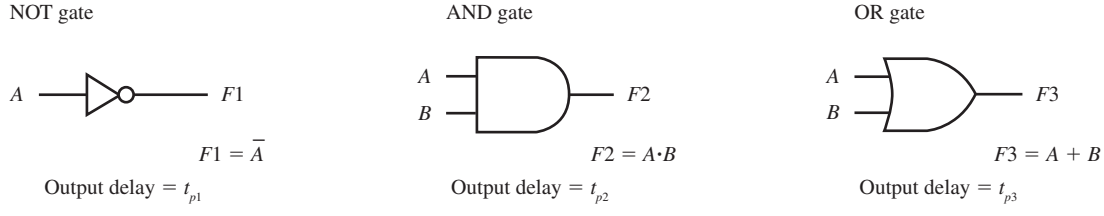
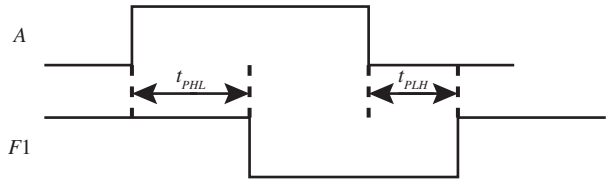


FIGURE 3.15 A NOT gate, an AND gate, and an OR gate with their symbols, their Boolean functions, and their propagation delay times

The propagation delay time, t_p, is the average of t_{PLH} and t_{PHL}, which are specified in the data sheets for the ICs. The propagation delay time (high-to-low-level output, or t_{PHL}) is the delay time through a gate when the output changes from a high (H) value to a low (L) value. The propagation delay time (low-to-high-level output, or t_{PLH}) is the delay time through a gate when the output changes from a low (L) value to a high (H) value. Waveform 3.2 shows a waveform diagram that illustrates both t_{PHL} and t_{PLH} for the NOT gate in Figure 3.15.



WAVEFORM 3.2 Waveform diagram illustrating both t_{PLH} and t_{PHL} for the NOT gate in Figure 3.15

Delays add up. For example, three similar NOT gates connected in cascade or in series (where one output feeds into the next) cause a propagation delay time of three times the propagation delay time of one of the NOT gates, or $3t_{p1}$. Sometimes NOT gates are used to slow down or delay a signal through a circuit.

Delays also add up for all gate types, including NOT gates, AND gates, and OR gates as shown in Figure 3.16 for the function $F4 = A \cdot \overline{C} + \overline{A} \cdot C + \overline{B}$.

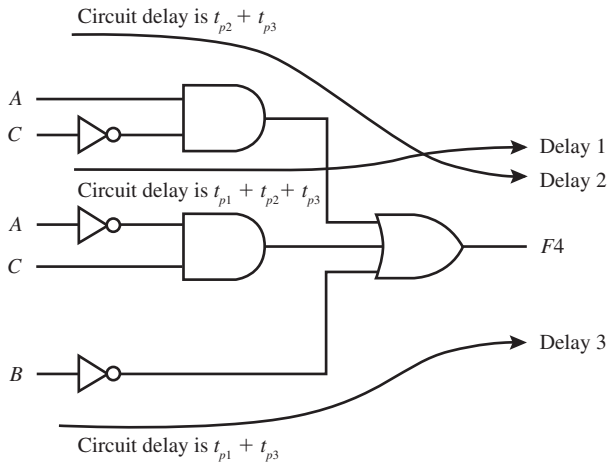


FIGURE 3.16 Worst-case delay time through the circuit

The worst-case delay time through the circuit in Figure 3.16 is the path from the input to the output that has the longest delay time—that is, delay 1 in the circuit in Figure 3.16. The delay times through the wires in the circuit in Figure 3.16 were ignored, but they would increase the overall delay time through the circuit slightly. So,

$$\text{Worst-case delay time} = t_{p1} + t_{p2} + t_{p3} = t_{pNOT\ gate} + t_{pAND\ gate} + t_{pOR\ gate}$$

In general, faster circuits have shorter delay times. Also, faster circuits have the fewest number of cascaded components from the input to the output of the circuit.

3.7 DECODERS

Figure 3.17a shows a very useful circuit called a **decoder** that utilizes NOT gates and AND gates. A circuit that converts a binary code applied to n input lines to one of 2^n different output lines is called an n-to-2^n line decoder. A decoder with n input lines can convert 2^n different binary codes applied to its input lines into 2^n mutually exclusive outputs. Each code applied to

its input is converted to a corresponding single bit on the output. Figure 3.17a shows the circuit diagram for a 2-line to 4-line decoder, which we will just call a 2-to-4 decoder.

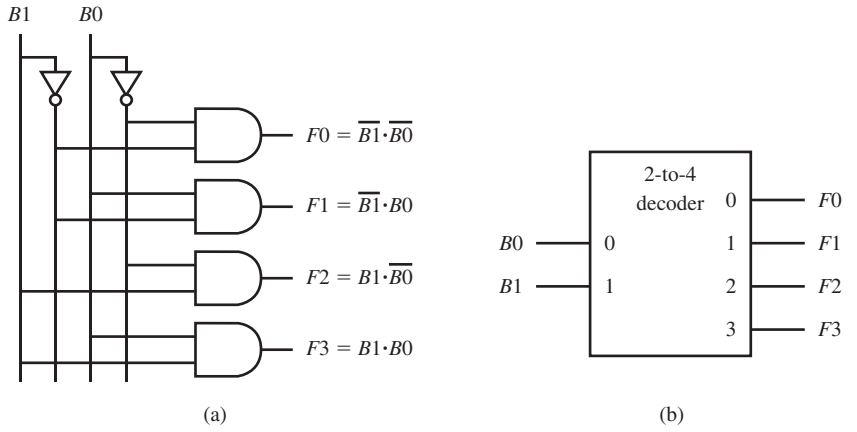


FIGURE 3.17 2-to-4 decoder: (a) discrete IC circuit diagram; (b) logic symbol

Figure 3.17b shows a logic symbol for a 2-to-4 Decoder. A decoder can also be thought of as a minterm generator because it generates the minterms at its outputs for each of the binary values applied to its inputs. For the inputs $B1$ $B0$ in Figure 3.17a, observe that the outputs are $F0(B1,B0) = \overline{B1}\cdot\overline{B0} = m0$, $F1(B1,B0) = \overline{B1}\cdot B0 = m1$, $F2(B1,B0) = B1\cdot\overline{B0} = m2$, and $F3(B1,B0) = B1\cdot B0 = m3$. Using this fact, it is very easy to manually design combination logic circuits for Boolean functions using a decoder with discrete ICs gates, as we will show in the next section.

In Figure 3.17a, when the binary input $B1B0$ is 00, output $F0$ evaluates to 1 and outputs $F1$, $F2$, and $F3$ evaluate to 0. When the binary input $B1B0$ is 01, output $F1$ evaluates to 1 and outputs $F0$, $F2$, and $F3$ evaluate to 0. When the binary input $B1B0$ is 10, output $F2$ evaluates to 1 and outputs $F0$, $F1$, and $F3$ evaluate to 0. When the binary input $B1B0$ is 11, output $F3$ evaluates to 1 and outputs $F0$, $F1$, and $F2$ evaluate to 0. This explanation is represented by the truth table for the 2-to-4 decoder shown in Table 3.1.

TABLE 3.1 Truth table for the 2-to-4 decoder in Figure 3.17

Select inputs		Outputs			
$B1$	$B0$	$F0$	$F1$	$F2$	$F3$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

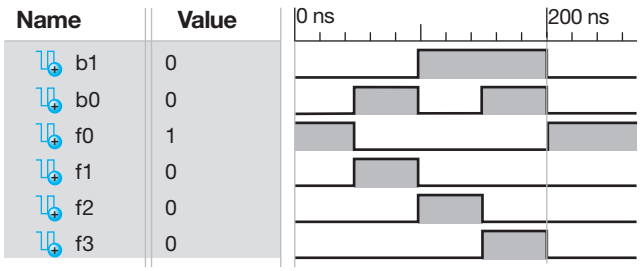
Outputs $F0$ through $F3$ in Table 3.1 are active-high outputs; that is, each decoded input results in *just one* 1 on the outputs. If we change the 1s to 0s and the 0s to 1s for the outputs in Table 3.1, we obtain a 2-to-4 decoder with active-low outputs; that is, each decoded input results in *just one* 0 on the outputs. Larger decoders (3-to-8, 4-to-16, 5-to-32, etc.) have similar truth tables that operate in a similar manner. Decoder circuits are available as discrete off-the-shelf IC devices. Decoders are often used in microprocessor or microcontroller systems as an address decoder that **selects a specific device** in the system such as a **RAM (random-access memory)**, a **ROM (read-only memory)**, or an **I/O (input/output) device** via the outputs of the decoder.

Listing 3.6 shows a complete VHDL design for the 2-to-4 decoder.

LISTING 3.6
Complete VHDL design for the 2-to-4 decoder (project: comb6)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb6 is port (
     B1, B0 : in std_logic;
     F0, F1, F2, F3 : out std_logic
     );
end comb6;

architecture Boolean_functions of comb6 is
begin
     F0 <= not B1 and not B0;
     F1 <= not B1 and B0;
     F2 <= B1 and not B0;
     F3 <= B1 and B0;
end Boolean_functions;
```

Waveform 3.3 shows waveform diagrams for the VHDL design for the 2-to-4 decoder.



WAVEFORM 3.3 Waveform diagrams for the VHDL design for the 2-to-4 decoder

Off-the-shelf decoders are usually equipped with one or more enable inputs—some active high and some active low. A decoder with an enable input is also called a **demultiplexer**. Table 3.2 shows the truth table for a 3-to-8 decoder with an active high enable input G1, an active low enable input G2, and active-low outputs. The 3-to-8 decoder shown in Table 3.2 is very similar to the off-the-shelf Texas Instruments CD74AC138 3-line to 8-line decoder/demultiplexer. As discussed earlier, logic products for Texas Instruments and data sheets are available online at http://ti.com.

TABLE 3.2 Truth table for the 3-to-8 decoder/demultiplexer

Enable inputs		Select inputs			Outputs							
*G*1	*G*2	*B*2	*B*1	*B*0	*F*0	*F*1	*F*2	*F*3	*F*4	*F*5	*F*6	*F*7
0	×	×	×	×	1	1	1	1	1	1	1	1
×	1	×	×	×	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1	1	1
1	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	1	1	0	1	1
1	0	1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	1	1	0

Using only the 0 in the column of the output function $F0$, we can write the Boolean function for $F0$ as $\overline{F0} = G1 \cdot \overline{G2} \cdot \overline{B2} \cdot \overline{B1} \cdot \overline{B0}$ or as $F0 = \overline{G1 \cdot \overline{G2} \cdot \overline{B2} \cdot \overline{B1} \cdot \overline{B0}}$. In VHDL, the latter form is written as

F0 <= **not** (G1 **and not** G2 **and not** B2 **and not** B1 **and not** B0).

Listing 3.7 shows a complete VHDL design for the 3-to-8 decoder/demultiplexer in Table 3.2.

LISTING 3.7 Complete VHDL design for the 3-to-8 decoder (project: comb7)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb7 is port (
     G1, G2, B2, B1, B0 : in std_logic;
     F0, F1, F2, F3, F4, F5, F6, F7 : out std_logic
     );
end comb7;

architecture Boolean_functions of comb7 is
begin
     F0 <= not (G1 and not G2 and not B2 and not B1 and not B0);
     F1 <= not (G1 and not G2 and not B2 and not B1 and B0);
     F2 <= not (G1 and not G2 and not B2 and B1 and not B0);
     F3 <= not (G1 and not G2 and not B2 and B1 and B0);
     F4 <= not (G1 and not G2 and B2 and not B1 and not B0);
     F5 <= not (G1 and not G2 and B2 and not B1 and B0);
     F6 <= not (G1 and not G2 and B2 and B1 and not B0);
     F7 <= not (G1 and not G2 and B2 and B1 and B0);
end Boolean_functions;
```

3.7.1 Designing Logic Circuits with Decoders and Single Gates

It is rather easy to manually design a logic circuit using a decoder and a single gate (AND gate, OR gate, NAND gate, or NOR gate). The design technique utilizes the fact that a decoder generates all possible minterms for the input variables. ORing the required minterms for the 1s of the function is the job of an OR gate when designing with a decoder that has active high outputs. If a decoder has active low outputs, then ORing the minterms is the job of a NAND gate drawn as an OR form—that is, its DeMorgan equivalent gate symbol.

ORing the required minterms for the 0s of the function is the job of a NOR gate when designing with a decoder that has active high outputs. If a decoder has active low outputs, then ORing the minterms is the job of an AND gate drawn as an OR form—that is, its DeMorgan equivalent gate symbol.

When designing with a decoder and a gate, the function does not have to be reduced. Table 3.3 shows the truth table for the function $F1(A,B,C) = \Sigma m(2,3,5,7)$.

TABLE 3.3 Truth table for function *F*1

A	B	C	$F1$
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

Figure 3.18a shows a design for the function $F1(A,B,C) = \Sigma m(2,3,5,7)$ in Table 3.3 using a 3-to-8 decoder with an OR gate.

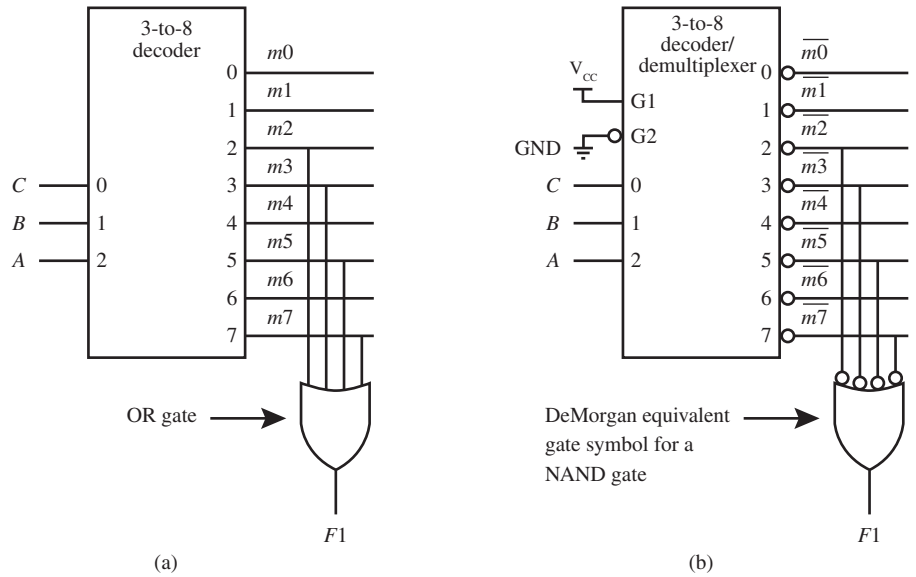


FIGURE 3.18 Design of the function $F1(A,B,C) = \Sigma m(2,3,5,7)$ (a) using a 3-to-8 decoder with active high outputs for the 1s of the function $F1$ and (b) using a 3-to-8 decoder/demultiplexer with active low outputs for the 1s of the function $F1$

Figure 3.18b shows an equivalent design for the function $F1(A,B,C) = \Sigma m(2,3,5,7)$ in Table 3.3 using a 3-to-8 decoder/demultiplexer with a NAND gate. Notice that both designs use the function expressed as the minterms of the 1s of the function $F1$.

Listing 3.8 shows a complete VHDL design for the function $F1(A,B,C) = \Sigma m(2,3,5,7)$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb8 is port (
     A, B, C : in std_logic;
     F1 : out std_logic
     );
end comb8;

architecture Boolean_function of comb8 is
begin
     F1 <= (not A and B and not C) or (not A and B and C) or
                    --minterms 2 and 3
           (A and not B and C) or (A and B and C);
                    --minterms 5 and 7
end Boolean_function;
```

LISTING 3.8 Complete VHDL design for the function $F1(A,B,C) = \Sigma m(2,3,5,7)$ (project: comb8)

The function $F1$ expressed as the minterms of the 0s of the function is written as $\overline{F1}(A,B,C) = \Sigma m(0,1,4,6)$. Figure 3.19a shows a design for the function $\overline{F1}(A,B,C) = \Sigma m(0,1,4,6)$ using a 3-to-8 decoder with a NOR gate, which ORs the minterms and complements the result.

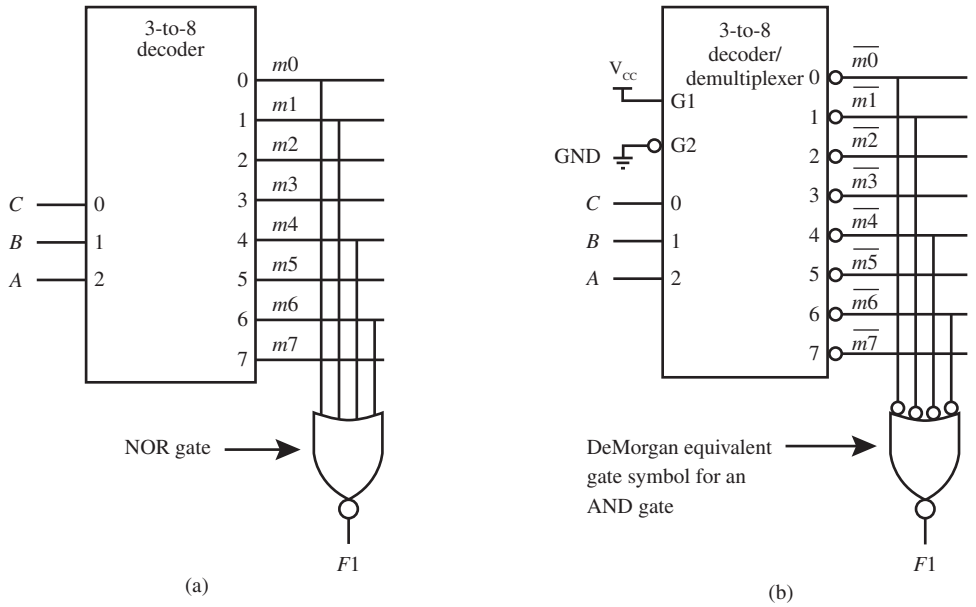


FIGURE 3.19 Design for the function $\overline{F1}(A,B,C) = \Sigma m(0,1,4,6)$: (a) using a 3-to-8 decoder with active high outputs for the 0s of the function $F1$; (b) using a 3-to-8 decoder/demultiplexer with active low outputs for the 0s of the function $F1$

Figure 3.19b shows an equivalent design for the function $\overline{F1}(A,B,C) = \Sigma m(0,1,4,6)$ using the 3-to-8 decoder/demultiplexer with an AND gate. If you are confused with the names associated with the DeMorgan equivalent gate symbols, now is a good time to review the DeMorgan equivalent gate symbols back in Chapter 1, Section 1.5.1.

When implementing a function with a decoder, it is best to use the fewest number of 1s or 0s to make up the function so that the fan-in of the gate is as small as possible. If there are fewer 1s in the function, use the compact minterm form for the 1s of the function to obtain the design; however, if there are fewer 0s in the function, use the compact minterm form for the 0s of the function. Additional gates can also be added to provide for additional outputs, thus allowing more than a single function to be implemented with a decoder.

Listing 3.9 shows a complete VHDL design for the function $\overline{F1}(A,B,C) = \Sigma m(0,1,4,6)$.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb9 is port (
     A, B, C : in std_logic;
     F1 : out std_logic
     );
end comb9;

architecture Boolean_function of comb9 is
begin
     F1 <= not ((not A and not B and not C) or (not A and not B and C) or
                    --minterms 0 and 1
               (A and not B and not C) or (A and B and not C));
                    --minterms 4 and 6
end Boolean_function;
```

LISTING 3.9 Complete VHDL Design for the function $\overline{F1}(A,B,C) = \Sigma m(0,1,4,6)$ (project: comb9)

3.8 MULTIPLEXERS

Figure 3.20a shows a very versatile circuit called a **multiplexer** or **MUX** that utilizes a NOT gate, a couple of AND gates, and an OR gate shown in AND/OR form. A MUX is a circuit that is used to direct one of 2^n data inputs to a single output. Because n select lines are used to select each of the 2^n data input signals and direct it to the output, a MUX is also called a **data selector**. Figure 3.20a shows the circuit diagram for a 2-line to 1-line MUX, which we will just call a 2-to-1 MUX.

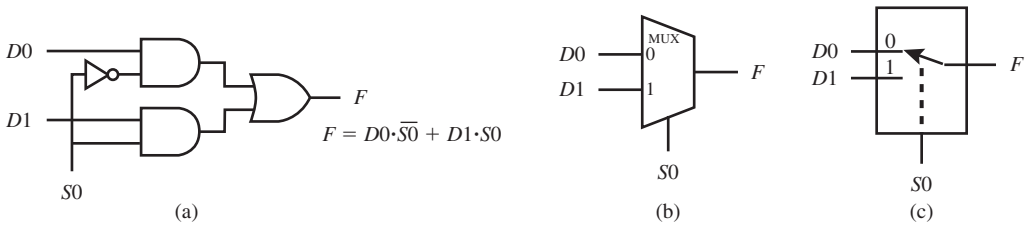


FIGURE 3.20
Multiplexer: (a) gate-level circuit diagram; (b) logic symbol; (c) switch representation

To observe how the MUX works, look at the switch representation in Figure 3.20c. When the select input $S0$ is 0, output F is $D0$, and when $S0$ is 1, output F is $D1$. The equation of the MUX in Figure 3.20a provides the same result when $S0$ is 0 and when $S0$ is 1 as shown as follows:

$F = D0 \cdot \overline{S0} + D1 \cdot S0 = D0$ when $S0 = 0$

$F = D0 \cdot \overline{S0} + D1 \cdot S0 = D1$ when $S0 = 1$

The logic symbol for the MUX in Figure 3.20b implies that $F = D0$ when $S0 = 0$ and $F = D1$ when $S0 = 1$. The truth table for the 2-to-1 MUX is shown in Table 3.4.

TABLE 3.4 Truth table for the 2-to-1 MUX in Figure 3.20

Inputs			Output
$S0$	$D1$	$D0$	F
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

Notice in the truth table that output F follows (is the same as) input $D0$ when $S0$ is 0, but output F follows (is the same as) input $D1$ when $S0$ is 1. Based on this observation, we can write a compact or compressed form of the truth table for the 2-to-1 MUX as shown in Table 3.5.

TABLE 3.5 Compressed truth table for the 2-to-1 MUX in Figure 3.20

$S0$	F
0	$D0$
1	$D1$

Listing 3.10 shows a complete VHDL design for the function $F = D0 \cdot \overline{S0} + D1 \cdot S0$ for the 2-to-1 MUX.

LISTING 3.10
Complete VHDL design for the function $F = D0 \cdot \overline{S0} + D1 \cdot S0$ for the 2-to-1 MUX (project: comb10)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb10 is port (
     D1, D0, S0 : in std_logic;
     F : out std_logic
     );
end comb10;

architecture Boolean_function of comb10 is
begin
     F <= (D0 and not S0) or (D1 and S0);
end Boolean_function;
```

Larger MUXs or data selectors (4-to-1, 8-to-1, 16-to-1, etc.) have similar truth tables and operate in a similar manner with more inputs. Table 3.6 shows the compressed truth table for a 2-to-1 MUX with an active low strobe input *G*. When the strobe input is 1 the output is 0, and when the strobe input is 0 the output is selected from one of the two data inputs and is routed to the output. The 2-to-1 MUX shown in Table 3.6 performs the same as one-fourth of an off-the-shelf Texas Instruments CD74AC157 Quadruple 2-line to 1-line data selector/multiplexer. CD stands for compliant device and is a lead-free device. When discarded, these devices do not pollute the world by contributing to lead contamination because they are lead free.

TABLE 3.6 Compressed truth table for the 2-to-1 MUX with an active low strobe input *G*

*G*0	*S*0	*F*
1	×	0
0	0	*D*0
0	1	*D*1

Figure 3.21a shows the logic symbol and output function for the 2-to-1 MUX in Table 3.6, and Figure 3.21b shows the logic symbol and output function for a 4-to-1 MUX without a strobe input.

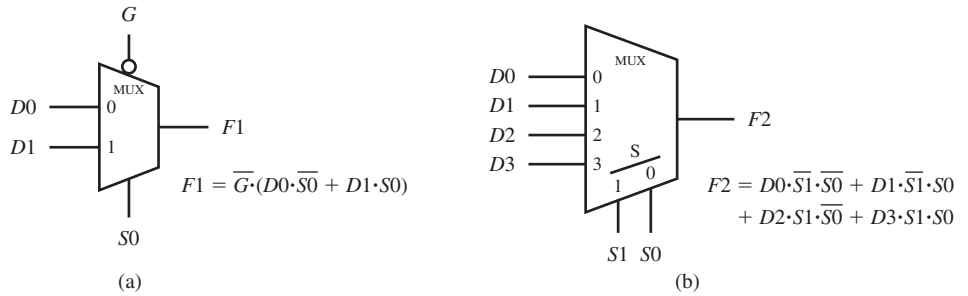


FIGURE 3.21 (a) Logic symbol and output function for the 2-to-1 MUX with an active low strobe input *G*; (b) Logic symbol and output function for a 4-to-1 MUX without a strobe input

MUXs are used to *implement designs for logic functions* and to *provide data-flow paths* between circuits by using MUXs as steering or routing circuits. We discuss the implementation of logic functions with MUXs in the following section.

Listing 3.11 shows a complete VHDL design for the 2-to-1 MUX and the 4-to-1 MUX in Figure 3.21.

LISTING 3.11
Complete VHDL design for the 2-to-1 MUX and the 4-to-1 MUX in Figure 3.21 (project: comb11)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb11 is port (
     D3, D2, D1, D0, S1, S0, G : in std_logic;
     F1, F2 : out std_logic
     );
end comb11;

architecture Boolean_functions of comb11 is
begin
     F1 <=  not G and ((D0 and not S0) or (D1 and S0));
     F2 <=  (D0 and not S1 and not S0) or (D1 and not S1 and S0) or
            (D2 and S1 and not S0) or (D3 and S1 and S0);
end Boolean_functions;
```

3.8.1 Designing Logic Circuits with MUXs

To obtain a MUX design for a logic function, we use the compact minterm form of the function. To implement the function, connect the data inputs of the MUX to the function values (V_{CC} for 1 and GND for 0). Connect the select lines of the MUX to the input variables. To generate the design for a function, simply write the compact minterm form for the function; that is, the function does not have to be reduced.

Figure 3.22a shows a MUX design for the function $F1(X,Y,Z) = \Sigma m(1,2,5,7)$ or the function $\overline{F1}(X,Y,Z) = \Sigma m(0, 3, 4, 6)$. Simply connect the data inputs of the MUX to the values of the function $F1$. If you wish, you may write the truth table for the function and then use the function values in the truth table, or you can simply make the connections for the 1s and by default the rest of the connections are for the 0s, or vice versa. The select inputs $S3$, $S2$, and $S1$ are then connected to the input X, Y, and Z respectively. The MUX design shown in Figure 3.22a uses an 8-to-1 MUX, such as a Texas Instruments CD74AC151, with an active low strobe input. The MUX design in Figure 3.22b does not have a strobe input.

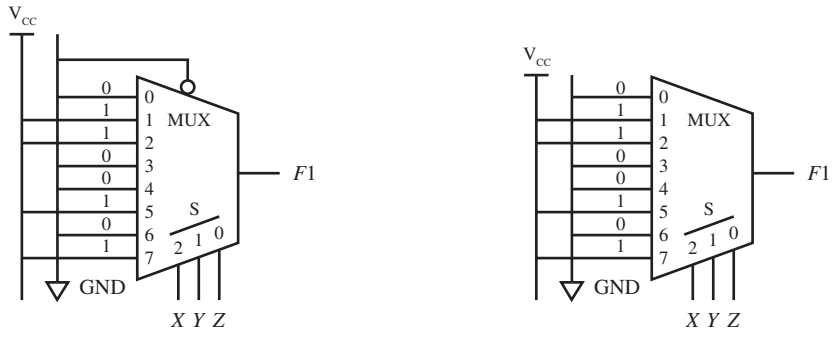


FIGURE 3.22 MUX design: (a) with a strobe input; (b) without a strobe input

With this manual procedure, you can obtain a MUX design for any 2-variable function using a 4-to-1 MUX, or you can obtain a MUX design for any 3-variable function using an 8-to-1 MUX. For a MUX design you need a 2^n-to-1 MUX for any n-variable function. Notice that you do not have to reduce a function to obtain a MUX design, but you must obtain the truth table of the function or obtain the function in compact minterm form for its 1s or 0s.

Listing 3.12 shows a complete VHDL design for the function $F1(X,Y,Z) = \Sigma m(1,2,5,7)$.

LISTING 3.12
Complete VHDL design for the function $F1(X,Y,Z) = \Sigma m(1,2,5,7)$ (project: comb12)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity comb12 is port (
     X, Y, Z : in std_logic;
     F1 : out std_logic
     );
end comb12;

architecture Boolean_function of comb12 is
begin
     F1 <= (not X and not Y and Z) or (not X and Y and not Z) or
                --minterms 1 and 2
           (X and not Y and Z) or (X and Y and Z);
                --minterms 5 and 7
end Boolean_function;
```

As you can see, the manual MUX design methods that we presented are rather easy to use and understand, but you still must obtain a detailed logic diagram for the circuit so that the circuit can be wired up on a breadboard or on a PC board. The modern way is to write VHDL code for the design and then download the bit pattern into a CPLD or an FPGA. The software automatically wires up the circuit on the CPLD or FPGA chip. You must remember to assign the external package pins for each of the signals that are placed in the entity.

3.9 HAZARDS

Hazards are classified as either **function hazards** or **logic hazards**. In this section, we present a brief introduction to these two different types of hazards. A hazard can cause a logic **glitch**, where a glitch is an undesired momentary pulse that occurs at the output of a circuit. In some cases, glitches in a circuit can cause a circuit to fail.

3.9.1 Function Hazards

A hazard that can cause a glitch in the output signal of a combinational logic function implemented with gates, when two or more input signals are changed at the same time in the circuit due to the way the function is defined, is called a **function hazard**. A function hazard can be spotted by plotting the function in a K-map. If two or more input signals are changed in the function to produce the output, a logic glitch may occur. In a combinational logic circuit, the designer has no control over function hazards. Figure 3.23 illustrates the occurrence of static and dynamic function hazards and their corresponding glitches for the function $F1$.

The function in Figure 3.23 is a 3-input XOR gate. All hardware implementations of this function will contain the function hazards, so showing the circuit is not necessary. The directed lines in the K-map show the transitions that cause each of the function hazards and their corresponding glitches—that is, a static 1 function hazard, a static 0 function hazard, a dynamic 1-to-0 function hazard, and a dynamic 0-to-1 function hazard. Notice in Figure 3.23 that a **runt pulse** can also occur. A runt pulse is a pulse with small amplitude.

Function hazards cannot be eliminated; however, the output signals from circuits that contain function hazards may be used by simply waiting until the function hazards settle (die out). After the output signals become stable or the function hazards settle, the signals may be used. This concept is the basis of synchronous circuits that are introduced in Chapters 6 and 9, where settling occurs between clock ticks.

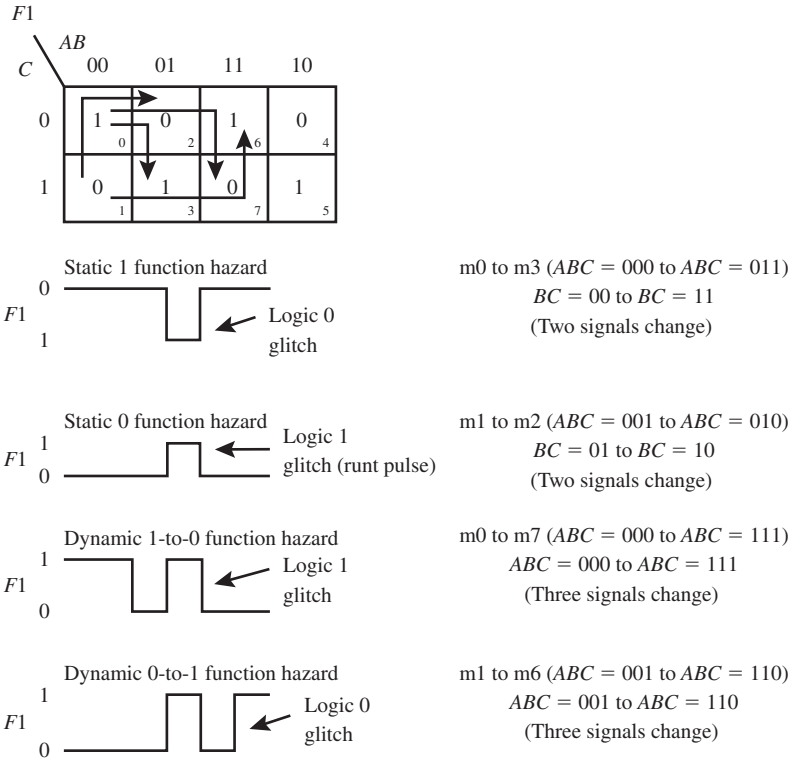


FIGURE 3.23 Function hazards—hazards that result from two or more input signals changing at the same time

3.9.2 Logic Hazards

A hazard that can cause a glitch in the output signal of a combinational logic function implemented with gates, when only one input signal is changed due to delays in the particular circuit used to implement the function, is called a **logic hazard**. Both static and dynamic logic hazards can be eliminated by adding additional product terms in the Boolean equation implemented by the circuit (this requires adding more gates in the implementation). To eliminate a logic hazard, a designer must recognize that a logic hazard may occur and add the necessary circuitry to prevent the logic hazard. Some functions do not contain logic hazards. Figure 3.24 shows a design specification with a single static 1 logic hazard. The single static 1 logic hazard represented in Figure 3.24a exists for any realization of the minimized logic function. Figure 3.24b shows a circuit for the function and a plausible explanation for the single static 1 logic hazard to be present.

Figure 3.25a shows how the single static 1 logic hazard can be eliminated, and Figure 3.25b shows a circuit for the function that eliminates the single static 1 logic hazard.

Things you should know about eliminating static and dynamic logic hazards:

- Logic hazards *may occur* for a minimized function implemented for the 1s or 0s of the function. If the product terms for the function are linked to each other, as shown for the K-map for the 1s of the function in Figure 3.25a, then the function contains no logic hazards.
- For the minimized 1s or 0s of a function, one can add logic hazard cover terms *if required*, which are consensus terms, to eliminate static and dynamic logic hazards.
- Cover terms are nonessential product terms that are used to link each product term in the minimized form in the K-map for the function.

- In some cases, product terms for a minimized form of the function are linked to each other and require no additional cover terms.
- By chain linking all the minimized product terms in a K-map for the function, you will obtain the required cover terms to add to the minimized form of the function to eliminate all the static and dynamic logic hazards for the function.
- A function that has all of its products terms for the 1s or 0s of the function linked to each other does not have static or dynamic logic hazards and is called a **logic hazard-free function**.

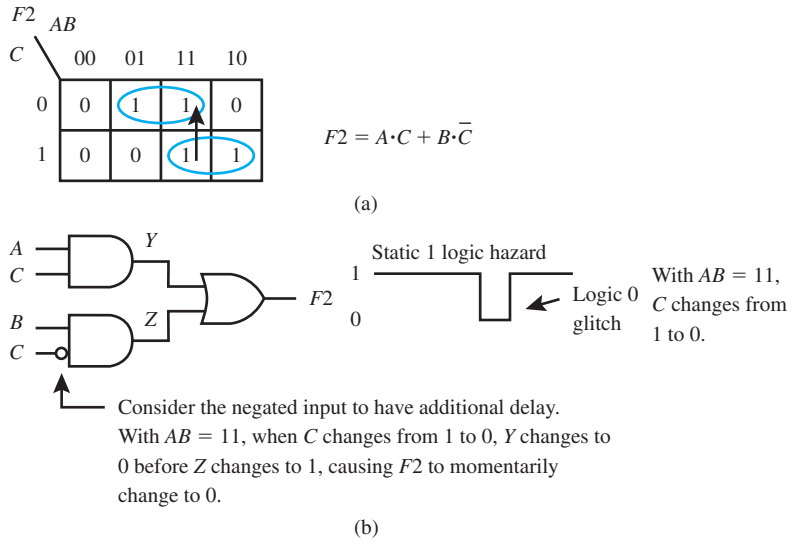


FIGURE 3.24 (a) Design specification with a single static 1 logic hazard; (b) circuit for the function and a plausible explanation for the single static 1 logic hazard to be present

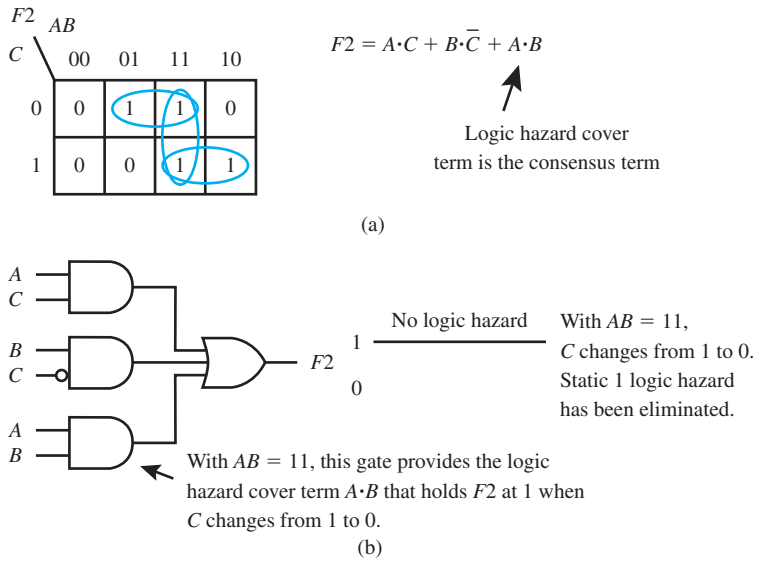


FIGURE 3.25 (a) How a single static 1 logic hazard is eliminated; (b) circuit for the function that eliminates the single static 1 logic hazard

Logic hazards may be eliminated; however, the output signals from circuits that contain logic hazards may also be used by simply waiting until the logic hazards settle (die out). After the output signals become stable or the logic hazards settle, the signals may be used. This concept is the basis of synchronous circuits that are introduced in Chapters 6 and 9, where settling occurs between clock ticks.

PROBLEMS

Section 3.2 Integrated Circuit Devices

3.1 What can physical hardware devices called integrated circuits (ICs) do?

3.2 What type of wire is used to connect together the die and the package leads or pins inside an IC?

3.3 What does an IC with a hermetical seal provide?

3.4 Name a few different types of integrated circuit packages.

3.5 Which IC package type has balls of solder on its pins that are soldered directly to a PC board?

Section 3.3 Analyzing and Designing Logic Circuits

3.6 What is the process of circuit analysis?

3.7 What is the process of circuit design or synthesis?

3.8 Show a common symbol for normally closed relay contacts.

3.9 Show a common symbol for normally open relay contacts.

3.10 Relays or switches connected in series provide what logic operation?

3.11 Relays or switches connected in parallel provide what logic operation?

3.12 What is the name that is used for logic switching circuit in power applications?

3.13 Analyze the logic switching circuit shown in Figure P3.13 to obtain its function *F*.

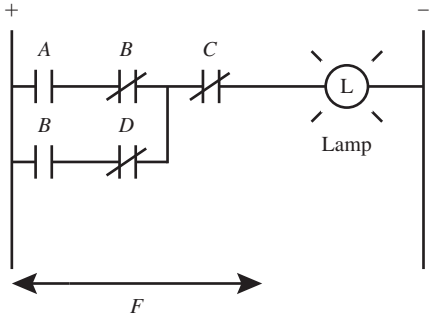


FIGURE P3.13

3.14 Show the design for an OR gate ladder logic circuit driving a lamp.

3.15 Show the design for an AND gate ladder logic circuit driving a control relay (CR).

3.16 Show the design for an NAND gate ladder logic circuit driving a lamp.

3.17 Analyze the logic circuit shown in Figure P3.17 to obtain its function *F* in SOP form and its truth table.

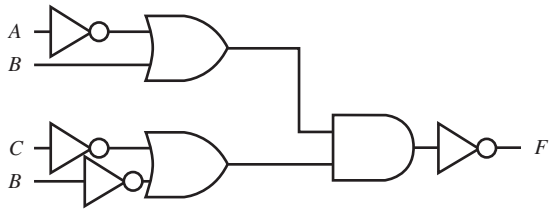


FIGURE P3.17

3.18 Analyze the logic circuit shown in Figure P3.18 to obtain its function *F* in SOP form and its truth table.

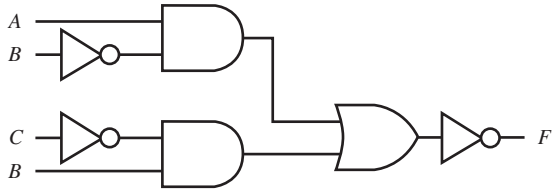


FIGURE P3.18

3.19 Analyze each of the following circuits in Figure P3.19 to obtain their Boolean function in SOP form and their truth table.

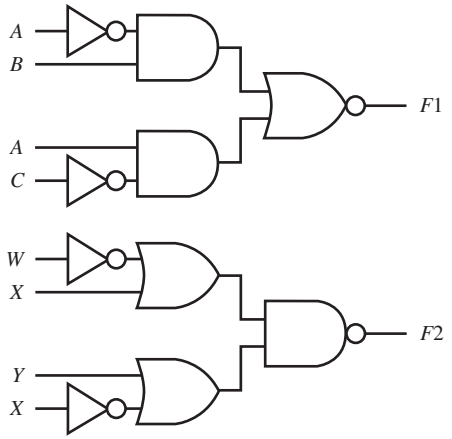


FIGURE P3.19

3.20 Design a circuit that provides the 1's complement at its output for each 4-bit binary number applied at its input. Use the input signals *IN*3, *IN*2, *IN*1, and *IN*0 and the

corresponding output signals *OUT*3, *OUT*2, *OUT*1, and *OUT*0. Hint: The 1's complement of a binary number is simply the complement of each individual bit.

3.21 Show a complete VHDL design for a circuit that provides the 1's complement at its output for each 4-bit binary number applied at its input. Use the input signals *IN*3, *IN*2, *IN*1, and *IN*0 and the corresponding output signals *OUT*3, *OUT*2, *OUT*1, and *OUT*0. Hint: The 1's complement of a binary number is simply the complement of each individual bit.

3.22 Design a circuit that provides the 2's complement at its output for each 3-bit binary number applied at its input. Use the input signals *X*, *Y*, and *Z* and the corresponding output signals *F*1, *F*2, and *F*3, and only AND, OR, and NOT gates. Hint: The 1's complement of a binary number is simply the complement of each individual bit. The 2's complement of a binary number is the 1's complement of the binary number + 1 (i.e., the 1 is added to the least significant bit).

3.23 Show a complete VHDL design for a circuit that provides the 2's complement at its output for each 3-bit binary number applied at its input. Use the input signals *X*, *Y*, and *Z* and the corresponding output signals *F*1, *F*2, and *F*3. Hint: The 1's complement of a binary number is simply the complement of each individual bit. The 2's complement of a binary number is the 1's complement of the binary number + 1 (i.e., the 1 is added to the least significant bit).

3.24 Design a majority of 1s circuit such that the output signal *F* is 1 when a majority of the input signals *X*, *Y*, and *Z* are 1. Only use OR, AND, and NOT gates.

3.25 Use VHDL to design a majority of 1s circuit such that the output signal *F* is 1 when a majority of the input signals *X*, *Y*, and *Z* are 1.

Section 3.4 Generating Detailed Schematics

3.26 What is the difference between a functional logic or schematic diagram and a detailed logic or schematic diagram?

3.27 List a good online source for obtaining datasheets for IC devices as mentioned in the text.

3.28 How many gates are contained in the IC device IC1 in Figure 3.10a in the text? Name the gates.

3.29 Which pin must be connected to V_{CC} and which pin must be connected to GND for IC device IC1 in Figure 3.10a in the text?

3.30 How many gates are contained in the IC device IC2 in Figure 3.10a in the text? Name the gates.

3.31 Which pin must be connected to V_{CC} and which pin must be connected to GND for IC device IC2 in Figure 3.10a in the text?

3.32 How many gates are contained in the IC device IC1 in Figure 3.10b in the text? Name the gates.

3.33 Which pin must be connected to V_{CC} and which pin must be connected to GND for IC device IC1 in Figure 3.10b in the text?

3.34 List the important items that are necessary when drawing a detailed schematic.

3.35 Why do you need to know how to provide detailed schematics of your designs?

3.36 Show a complete VHDL design for the functions $F1 = A \cdot B + \overline{A} \cdot \overline{B}$ and $\overline{F2} = \overline{A} \cdot B$.

3.37 What is the purpose of running a simulation on a VHDL design?

3.38 To obtain a circuit on a system board, why is a detailed schematic not required when using a hardware description language such as VHDL?

Section 3.5 Designing Circuits in NAND/NAND and NOR/NOR Form

3.39 Design a two-1s-out-of-four event detector for input signals *A*, *B*, *C*, *D* and output signal *F*. Use a vertical-input scheme and fan-in reduction if possible using NOT gates (six in a package), 4-input NAND gates (two in a package), and an 8-input NAND gate (one in a package). Use the graphical design method.

3.40 Show a complete VHDL design for a two-1s-out-of-four event detector for input signals *A*, *B*, *C*, *D* and output signal *F*.

3.41 Design a circuit using the 1s of the function $F(X,Y,Z) = \Sigma m(1,2,4)$. If possible, reduce the function. Draw the circuit for the function using just NAND gates in NAND/NAND form and NOT gates. Use the graphical design method.

3.42 Show a complete VHDL design for a circuit using the 1s of the function $F(X,Y,Z) = \Sigma m(1,2,4)$.

3.43 Design a circuit using the 1s of the function $F(X,Y,Z) = \Sigma m(1,2,4)$. If possible, reduce the function. Draw the circuit for the function using just NOR gates in NOR/NOR form and NOT gates. Use the graphical design method.

3.44 Design a circuit using the 1s of the function $F(W,X,Y,Z) = \Sigma m(0,2,5,7,8,10) + \Sigma md(12, 13)$. If possible, reduce the function. Draw the circuit for the function using just NAND gates in NAND/NAND form and NOT gates. Use the graphical design method.

3.45 Show a complete VHDL design for a circuit using the 1s of the function $F(W,X,Y,Z) = \Sigma m(0,2,5,7,8,10) + \Sigma md(12,13)$. Hint: Choose 0s for the don't cares, to reduce the number of minterms in the VHDL expression for *F*.

3.46 Design a circuit using the 1s of the function $F(W,X,Y,Z) = \Sigma m(0,2,5,7,8,10) + \Sigma md(12,13)$. If possible, reduce the function. Draw the circuit for the function using just NOR gates in NOR/NOR form and NOT gates.

Section 3.6 Propagation Delay Time

3.47 Which metal has the best conductivity—copper, gold, or silver? Which has the second best conductivity? Which has the third best conductivity?

3.48 Provide an equation for t_p in terms of t_{PHL} and t_{PLH}.

3.49 Describe what is meant by the term *worst-case delay time through a circuit*.

3.50 Draw a circuit for a function *FDELAY* = *A* made up of four cascaded NOT gates. If each NOT gate has a delay

of t_p, what is the output delay of the circuit from its input to its output?

3.51 Draw a circuit for the function $\overline{FD1} = A \cdot \overline{B}$ implemented with an AND gate and NOT gates. Determine the worst-case output delay for the circuit, assuming each gate has a delay of t_p.

3.52 Draw a circuit for the function $FD2 = \overline{\overline{X} + Y}$ implemented with an OR gate and NOT gates. Determine the worst-case output delay for the circuit, assuming each gate has a delay of t_p.

Section 3.7 Decoders

3.53 Draw and label a gate level circuit for a 3-to-8 decoder with active low outputs. Also draw and label a logic symbol for the 3-to-8 decoder.

3.54 Design a circuit for the function $F(X,Y,Z) = \Sigma m(0,2,7)$ with a 3-to-8 decoder with active high outputs. Use an appropriate gate to provide the smallest possible fan-in to implement the function.

3.55 Show a complete VHDL design for the function $F(X,Y,Z) = \Sigma m(0,2,7)$.

3.56 Design a circuit for the function $F(X,Y,Z) = \Sigma m(0,1,5,6,7)$ with a 3-to-8 decoder/demultiplexer that has an active low enable input and active low outputs. Use an appropriate gate to provide the smallest possible fan-in to implement the function.

3.57 Show a complete VHDL design for the function $F(X,Y,Z) = \Sigma m(0,1,5,6,7)$. To simplify the function in VHDL, rewrite the function with a minimum number of minterms.

Section 3.8 Multiplexers

3.58 Write the equation for a 4-to-1 MUX (data selector); then draw and label the circuit. Use $D3$ down to $D0$ as the data inputs, $S1$ down to $S0$ as the select inputs, and F as the output. Draw a logic symbol for the circuit, and then show its truth table in compressed form.

3.59 Show a complete VHDL design for a 4-to-1 MUX (data selector). Use $D3$ down to $D0$ as the data inputs, $S1$ down to $S0$ as the select inputs, and F as the output.

3.60 Obtain a MUX design for the AND function $F(A,B) = A \cdot B$ using an off-the-shelf MUX without a strobe input.

3.61 Show a complete VHDL design for the AND function $F(A, B) = A \cdot B$.

3.62 Obtain a MUX design for the NOR function $F(A,B) = \overline{A + B}$ using an off-the-shelf MUX without a strobe input.

3.63 Show a complete VHDL design for the NOR function $F(A,B) = \overline{A + B}$.

3.64 Obtain a MUX design for the XOR function $F(X,Y) = X \cdot \overline{Y} + \overline{X} \cdot Y$ using an off-the-shelf MUX with an active low strobe input.

3.65 Show a complete VHDL design for the XOR function $F(X,Y) = X \cdot \overline{Y} + \overline{X} \cdot Y$.

3.66 Obtain a MUX design for the function $F(X,Y,Z) = \Sigma m(0,1,2,3,5,7)$ using an off-the-shelf MUX with an active low strobe input.

3.67 Show a complete VHDL design for the function $F(X,Y,Z) = \Sigma m(0,1,2,3,5,7)$. To simplify the function in VHDL, rewrite the function with a minimum number of minterms.

Section 3.9 Hazards

3.68 Name the two classifications of hazards.

3.69 What can a hazard cause at the output of a circuit?

3.70 List the reason a glitch can occur in a combinational logic circuit as a result of a function hazard.

3.71 List the four types of function hazards covered in the book.

3.72 List the two types of glitches covered in the book.

3.73 What is a runt pulse?

3.74 Can function hazards be eliminated? Describe how the outputs of circuits that contain function hazards can be used.

3.75 List the reason a glitch can occur in a combinational logic circuit as a result of a logic hazard.

3.76 Can logic hazards be eliminated? If so, describe how.

3.77 Eliminate all logic hazards that can result for the 1s of the Boolean function $F1(A,B,C) = \Sigma m(1,3,4,5)$. Show the K-maps and the equations for the minimum Boolean function and the logic hazard-free Boolean function.

3.78 Eliminate all logic hazards that can result for the 0s of the Boolean function $F1(A,B,C) = \Sigma m(1,3,4,5)$. Show the K-maps and the equations for the minimum Boolean function and the logic hazard-free Boolean function.

3.79 Eliminate all logic hazards that can result for the 1s of the Boolean function $F2(A,B,C) = \Sigma(1,2,3,6)$. Show the K-maps and the equations for the minimum Boolean function and the logic hazard-free Boolean function.

3.80 Eliminate all logic hazards that can result for the 0s of the Boolean function $F2(A,B,C) = \Sigma(1,2,3,6)$. Show the K-maps and the equations for the minimum Boolean function and the logic hazard-free Boolean function.

3.81 Eliminate all logic hazards that can result for the 1s of the Boolean function $F3(A,B,C) = \Sigma(0,2,3,4,6)$. Show the K-maps and the equations for the minimum Boolean function and the logic hazard-free Boolean function.

3.82 Eliminate all logic hazards that can result for the 0s of the Boolean function $F3(A,B,C) = \Sigma(0,2,3,4,6)$. Show the K-maps and the equations for the minimum Boolean function and the logic hazard-free Boolean function.

3.83 Determine the number of logic hazards that the circuit shown in Figure P3.83 could contain. What product terms are necessary to eliminate these logic hazards? Write a logic hazard-free function for the circuits.

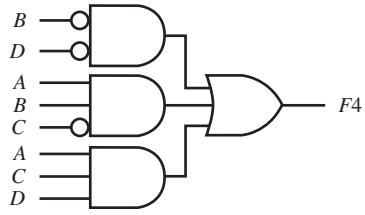


FIGURE P3.83

Chapter 4

Combinational Logic Circuit Design with VHDL

Chapter Outline

4.1 INTRODUCTION

Up to now we have only introduced VHDL code using Boolean functions. In this chapter, you will learn many different ways to write VHDL to **synthesize** or create digital hardware for combinational logic circuits. Combinational logic circuits are digital circuits *without feedback*, which means that the outputs are totally dependent on the external inputs to the circuits.

4.2 VHDL

To write VHDL code to create digital hardware, you must create a **design entity**. An entity is defined by *The American Heritage Dictionary* as "Something that exists as a particular and discrete unit." A design entity is therefore a complete definition for a **block of hardware**. A design entity in VHDL has **three parts** called the **library part**, **entity declaration**, and **architecture declaration**.

In the hardware description language VHDL, a design entity may be visualized as consisting of the parts shown in Figure 4.1.

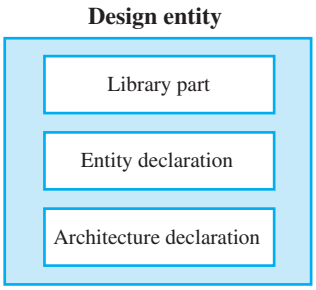


FIGURE 4.1 Visualization of a design entity in the hardware description language VHDL

The library part consists of **a standard logic library** with its **data type definitions**, **functions**, **and procedures**. The entity declaration (ED) specifies **the interface** or the external inputs and outputs of a digital circuit. The architecture declaration (AD) specifies **the functional composition** or **functionality** of a digital circuit.

4.3 THE LIBRARY PART

To make a complete VHDL design, we need to place a library part that consists of a **library clause** and a **use clause** within each design entity so that we can use a standard logic library with its data type definitions, functions, and procedures. Once you write a VHDL design, you must analyze, compile, or synthesize the code with a vendor's design tool. We will use the terms *analyze*, *compile*, and *synthesize* interchangeably. The synthesize process identifies **syntactical errors**—that is, those errors governing the rules of the language. The **library** is a **storage place** that contains **packages** that supply information for your design. The library also provides storage for your compiled VHDL code. Figure 4.2 shows three types of libraries that are available for VHDL designs. These are the main libraries we will use.

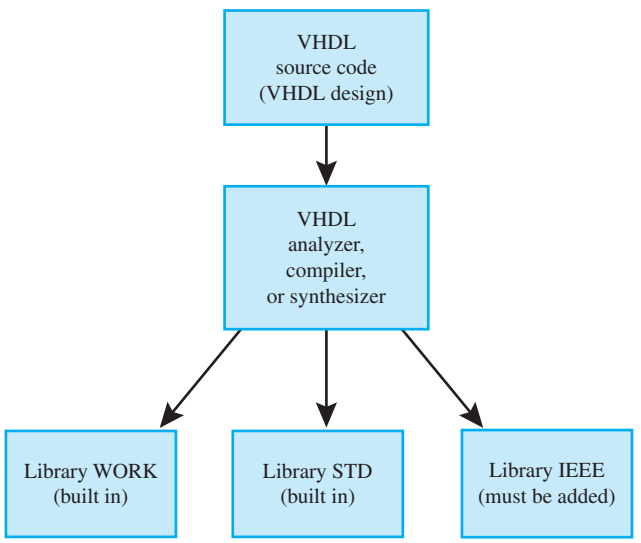


FIGURE 4.2
Available library units

The WORK and STD libraries in Figure 4.2 are implicitly declared (built in), but IEEE library is not and must be added to a VHDL design via a library clause to make it visible to

the design. The WORK library is the default library, and this is the storage place for the current design you are working with and where that design is placed after it is compiled. The STD library is the storage place for the miscellaneous and logical operators such as **not**, **and**, and **nor**. The STD library also contains relational operators such as =, >, <. Refer to Section 4.10, VHDL Examples (at the end of this chapter), for a list of all the supported operators. The IEEE library is the storage place for 9-value data types called std_logic in the **package** IEEE. STD_LOGIC_1164. The main reason for using the IEEE library is for design portability—that is, you can use your VHDL source code with many different software vendors.

Listing 4.1 shows a **library clause** and a **use clause** for a VHDL design.

```
--These are the library and use clauses (The Library Part)
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

LISTING 4.1 Making the IEEE library and IEEE.STD_LOGIC_1164 package visible to a design

The **library** IEEE clause makes the IEEE library visible to a design. A library contains packages. The IEEE.STD_LOGIC_1164 package is specified by the **use clause** so that data type definitions, functions, and procedures that reside in the package are visible to the design. A specific component name can be used inside the package, but the wild card **ALL** is used to indicate that all the declarations inside the package can be used.

4.4 THE ENTITY DECLARATION

Think of a **black box** and inputs *A*, *B*, *C*, *D* (or terminal signals) and an output *F* as shown in Figure 4.3.

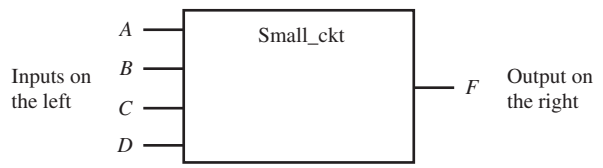


FIGURE 4.3 Black box for a design entity named Small_ckt

The term *black box* is used, as in circuit theory, to specify only the terminal or external signals for a design entity and not its contents. What is inside the box or the contents of the black box are not visible. A black box is equivalent to a symbol in a schematic. The terminal signals may represent the pins on an integrated circuit chip or the terminal signals of an embedded hardware block within a larger design. Small_ckt (short for Small_circuit) is simply the name or label that is used to identify a specific design entity.

Listing 4.2 shows the ED (entity declaration) for the black box in Figure 4.3.

LISTING 4.2 Entity declaration for black box named Small_ckt in Figure 4.3

```
--This is the entity declaration for design entity Small_ckt
entity Small_ckt is port (
    A, B, C, D : in std_logic;
    F : out std_logic
    );
end Small_ckt;
```

Things you should notice about the VHDL entity declaration in Listing 4.2:

- Comments start with --.
- **Keywords** (reserved words) are shown in **boldface type** in the VHDL code. Refer to Section 4.10, VHDL Examples (at the end of this chapter), for a list of all keywords that are supported for synthesis.
- An ED must start with the keyword **entity**.
- VHDL is not case sensitive and has a free format, which means that there is no formatting convention for spacing and indentations.
- Signal names (such as A, B, etc.) and labels (such as Small_ckt) have the following rules: (a) the first character must be a letter, (b) numbers may be included as well as the underscore character (_), (c) no adjacent underscore characters may be used, (d) an underscore character may not be used as the last character, and (e) spaces are not allowed. Keywords (reserved words) cannot be used as signal names or labels. Signal names and labels are formally called **identifiers**.
- Input signals are mode **in**.
- Std_logic is the **data type** used in most digital designs [this is a 9-value system of which only the values 0, L (weak 0—pull-down resistor), 1, H (weak 1—pull-up resistor), Z (high impedance—tri-state), and - (don't care) are supported for synthesis, i.e., hardware implementations]. Note: Values L, H, and Z are case sensitive; that is, they must be uppercase. Just for your information, the values U (uninitialized), W (weak unknown), and X (forcing unknown) are not supported by synthesis. This leaves only six values that are supported for synthesis. The values U, W, and X are only supported for simulations of designs on a computer but not for synthesizing designs via hardware components. The Xilinx compiler supports the value - (don't care), but not all compilers do.
- Output signals are mode **out**.
- Observe where semicolons are required and where a semicolon is not allowed, that is, before ");" in the **entity**.
- A port must open with a left parenthesis and end with a right parenthesis.
- The entity must terminate with the keyword **end**.

Data Type Bit

A signal can be assigned the data type "bit." The data type "bit" can only take on the values of 1 and 0. This limits the usefulness of this data type, because it cannot have the values L, H, Z, or -, which are the additional data types used by data type "std_logic." Also, VHDL has strict data type checking so that one data type cannot be assigned to a different data type. Data type "std_logic" is the preferred data type for most designs because it has six values that are supported for synthesis rather than the data type "bit," which only has two values.

4.5 THE ARCHITECTURE DECLARATION

The contents of a black box represent the actual digital circuit for the design. In VHDL, the actual digital circuit is expressed in an **architecture declaration**. Figure 4.4 shows the contents of the black box labeled Small_ckt.

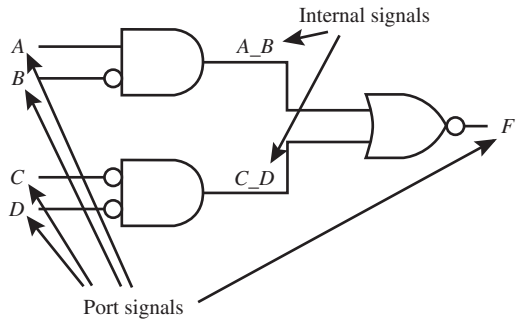


FIGURE 4.4 Digital circuit: contents of the black box for design entity named Small_ckt in Figure 4.3

VHDL has three design styles available for use in an architecture declaration. We will use each of these design styles for the digital circuit in Figure 4.4 to help you to understand their syntax. The three design styles are listed as follows:

1. Dataflow design style.
2. Behavioral design style.
3. Structural design style.

4.5.1 Comments about a Dataflow Design Style

Only (1) *Boolean equations*, (2) *conditional signal assignments (CSAs)*, and (3) *selected signal assignments (SSAs)* can be used in a dataflow design style. These signal assignments are **concurrent statements (CSs)** because they are evaluated by the VHDL compiler concurrently or at the same time. The order in which they are written is not important. The simplest form of a dataflow design style is a Boolean equation. The equation implies how the hardware should be created; therefore, **implication** is used to create the hardware required for a circuit when using a dataflow design style.

4.5.2 Comments about a Behavioral Design Style

Only (1) *Boolean equations*, (2) *if statements*, and (3) *case statements* can be used in a behavioral design style. These equations and statements must be placed inside a **process**. The complete **process** is a **concurrent statement**; however, the statements inside the **process** are evaluated sequentially by the VHDL compiler—that is, in the order in which they are written in the process. A behavioral architecture declaration creates the structure for a circuit by **inference** (creating the circuit from deduction by reasoning from the general to the specific).

4.5.3 Comments about a Structural Design Style

Hardware blocks or components are used in a structural design style. An **annotated circuit** or **schematic** must be provided to use this design style—that is, a schematic with all the input, output, and internal signals clearly labeled. The schematic is separated into hardware blocks or components and then simply connected together just like wiring a digital circuit using gates. The installation or placement of the hardware blocks or components are referred to in VHDL as **instantiation** and their interconnections are referred to as **port mapping**. Instantiation and port-mapping statements are **concurrent statements** because they are evaluated at the same time. The order in which they are written is not important. A structural architecture declaration creates the structure for a circuit by the way you **wire it up or connect the components**.

As you will see later, a combination of design styles can also be used together in an architecture declaration to generate a hardware circuit for a **system**. We will refer to a collection of modules or components that form a hardware circuit in VHDL as a system. When we use a combination of design styles, we will call the name of the architecture **mixed** to indicate that a mixture of design styles is being used.

4.6 DATAFLOW DESIGN STYLE

Listing 4.3 shows an AD (architecture declaration) with a dataflow design style for the digital circuit in Figure 4.4—that is, the design entity named Small_ckt.

LISTING 4.3
Dataflow architecture declaration with a Boolean equation

```
--This is a dataflow AD for Small_ckt
architecture dataflow of Small_ckt is
begin
    F <= (A and not B) nor (not C and not D);
end dataflow;
```

Things you should notice about the VHDL architecture declaration in Listing 4.3:

- The architecture declaration must start with the keyword **architecture**.
- The architecture declaration must reference the name of the design entity—that is, Small_ckt for this example.
- The keyword **begin** is required before the **simple signal assignment statement** for *F*—that is, a **Boolean equation**. A single Boolean equation is sufficient; however, multiple Boolean equations could be specified if we add a declaration for the internal signals *A_B* and *C_D*.
- A Boolean expression is assigned to the signal *F* via the signal assignment symbol (<=).
- A semicolon is required to terminate the **simple signal assignment statement**.
- VHDL has no order of precedence for logic operators with two signals (or operands) such as **and** and **nor;** therefore, parentheses must be used to establish precedence for these logic operators. **Not** has the highest precedence of all the operators in VHDL and therefore does not need parentheses—that is, the result for **not** *B* and **not** (*B*) are the same.
- The dataflow AD must terminate with the keyword **end**.

If there is more than one signal assignment statement between **begin** and **end**, all of the signal assignment statements are evaluated at the same time.

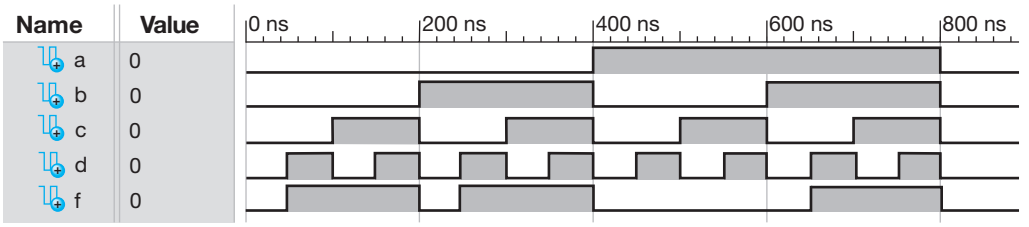
Complete VHDL code for the design entity Small_ckt is obtained by combining Listings 4.1, 4.2, and 4.3 as shown in Listing 4.4.

LISTING 4.4
Complete VHDL design entity for Small_ckt using a dataflow architecture declaration with a Boolean equation (project: Small_ckt_Bool)

```
--These are the library and use clauses
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--This is the entity declaration for Small_ckt
entity Small_ckt is port (
    A, B, C, D : in std_logic;
    F : out std_logic
    );
end Small_ckt;

--This is a dataflow architecture declaration for Small_ckt
architecture dataflow of Small_ckt is
begin
    F <= (A and not B) nor (not C and not D);
end dataflow;
```

To check for VHDL design correctness means to check for correct functionality. Waveform 4.1 shows a simulation waveform diagram with the correct functionality of design entity Small_ckt. This can easily be confirmed by creating a truth table for function F and then checking the waveform diagram with the truth table output.

WAVEFORM 4.1 Simulation for the correct functionality of design entity Small_ckt



A different architecture declaration can be used with the same entity declaration to implement the same hardware block. Rather than a **Boolean equation**, two other types of signal assignment statements may be used in a dataflow architecture declaration. A **conditional signal assignment (CSA)** is one type. Another type is a **selected signal assignment (SSA)**.

Listing 4.5 shows a complete VHDL design entity for Small_ckt using a conditional signal assignment in an architecture declaration.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Small_ckt is port (
     A, B, C, D : in std_logic;
     F : out std_logic
     );
end Small_ckt;

architecture dataflow of Small_ckt is
begin
     --writing F in terms of its 1s
     F <= '1' when ((A and not B) nor (not C and not D)) = '1' else
          '0';
--or one of the following:
     --writing F in terms of its 0s, observe that nor must be changed to or
--   F <= '0' when ((A and not B) or (not C and not D)) = '1' else
--        '1';
     --writing F in terms of its 1s
--   F <= '1' when (A and not B) = '1' nor (not C and not D) = '1' else
--        '0';
     --writing F in terms of its 1s
--   F <= '1' when (A = '1' and B = '0') nor (C = '0' and D = '0') else
--        '0';
end dataflow;
```

LISTING 4.5 Complete VHDL design entity for Small_ckt using a dataflow architecture declaration with a conditional signal assignment (project: Small_ckt_CSA)

Things you should notice about the VHDL design in Listing 4.5:

- Only one signal assignment symbol (<=) is used for a conditional signal assignment.
- A single bit is represented as '1' or '0'—that is, using single quotation marks.
- A Boolean expression or signal may be compared to a bit using the "=" relational operator. Parentheses must be used around a Boolean expression when it is compared to a bit using a relational operator.
- Parentheses determine the order of the precedence for logic operators.
- A semicolon is required to terminate a conditional signal assignment.
- The **when** conditions in a conditional signal assignment are prioritized. The first output statement listed (*F* <= '1') has the highest priority and will be executed first, if the condi-

tion is true. If the condition is not true, the first output statement is skipped and the second output statement (*F* <= '0') will be executed next, and so on (for additional conditions).

- If a conditional signal assignment does not have the last **else** (the **else** after the last **when**), a bad thing can happen: the circuit is not implemented correctly, because one or more inputs are not used. The output of the circuit is tied to GND or V_{CC} making the circuit nonfunctional or useless. For combinational circuits, be sure to include the last **else** (the **else** after the last **when)** so that latches or nonfunctional circuits are not inadvertently created on the outputs of the circuits.
- The simulation for the VHDL design in Listing 4.5 is the same as Waveform 4.1 shown earlier.

Listing 4.6 shows a complete VHDL design entity for Small_ckt using a selected signal assignment (SSA).

LISTING 4.6
Complete VHDL design entity for Small_ckt using a dataflow architecture declaration with a selected signal assignment (project: Small_ckt_SSA)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Small_ckt is port (
     A, B, C, D : in std_logic;
     F : out std_logic
     );
end Small_ckt;

architecture dataflow of Small_ckt is
begin
     with (A and not B) nor (not C and not D) select
          F <= '1' when '1',
               '0' when '0',
               '0' when others;
end dataflow;
```

Things you should notice about the VHDL design in Listing 4.6:

- The evaluation of a Boolean expression provides the selection value.
- Parentheses determine the order of precedence for logic operators.
- Only one signal assignment symbol (<=) is used for a selected signal assignment.
- A single bit is represented as '1' or '0'—that is, using single quotation marks.
- The selected signal assignment examines the value of the expression and executes only the signal assignment that matches the **when** value, and all the others signal assignments are skipped.
- **When others**; is required to terminate a selected signal assignment statement. **When others** is used to ensure that all possible select values for the expression (*A* **and not** *B*) **nor** (**not** *C* **and not** *D*) are included in the choice list after **when**. The std_logic values that must be included for the expression are 0, L, 1, H, Z, and -. Note that the single dash "-" represents a don't care in VHDL.
- Observe where commas are required in a selected signal assignment.
- The conditions for a selected signal assignment are not prioritized like a conditional signal assignment and thus require conditions that do not overlap or are mutually exclusive values.
- Order is not important, and the only output statement that is executed is the one that meets the condition.
- The simulation for the VHDL design in Listing 4.6 is the same as Waveform 4.1 shown earlier.

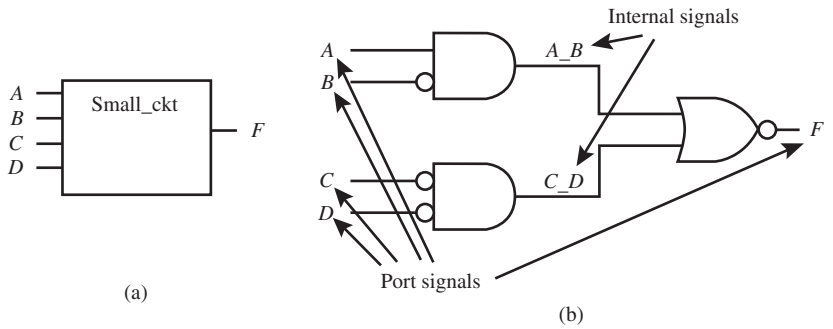
Listing 4.4, 4.5, or 4.6 may be used for the hardware design of the digital circuit in Figure 4.4 using a dataflow architecture declaration. You should now be very familiar with how to create VHDL design entities using a dataflow architecture declaration.

4.7 BEHAVIORAL DESIGN STYLE

Statements *within* a process—that is, in the body of the process (between **begin** and **end process**)—are evaluated in the order they are written (one after the other or sequentially), which is similar to the evaluation of statements in normal software programming languages such as Pascal, C, Java, Perl, and Ruby. Even though a process contains statements that are evaluated sequentially, a complete **process statement**, from **process** through **end process**, is a concurrent statement. This means that a complete process statement is evaluated concurrently with another complete process statement or with other concurrent statements in an architecture declaration. The fact that a complete process statement is a concurrent statement might not seem important to you at this time, but it will be very important later, so keep this in mind.

We will now show how to implement the design entity named Small_ckt using behavioral architecture declarations. To refresh your memory, Figures 4.3 and 4.4 are combined in Figure 4.5 to provide you with a handy reference.

FIGURE 4.5 Design entity Small_ckt (a) black box, and (b) contents of black box



The VHDL design entity for a behavioral architecture declaration must have the library clause, the use clause, and the entity declaration as shown in Listing 4.7 for design entity Small_ckt.

LISTING 4.7 Library clause, use clause, and entity declaration for Small_ckt

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Small_ckt is port (
    A, B, C, D : in std_logic;
    F : out std_logic
    );
end Small_ckt;
```

Listing 4.8 shows a behavioral architecture declaration using a process with Boolean equations, for design entity Small_ckt.

LISTING 4.8 Behavioral architecture declaration for design entity Small_ckt using a process with Boolean equations

```
architecture behavioral of Small_ckt is
    --internal signal declarations for A_B and C_D
    signal A_B, C_D: std_logic;
begin
example: process (A, B, C, D, A_B, C_D)
begin
    A_B <= A and not B;
    C_D <= not C and not D;
    F <= A_B nor C_D;
end process example;
end behavioral;
```

Things you should notice about the VHDL design in Listing 4.8:

- Internal signals (such as *A_B* and *C_D*) must be declared before the first **begin**, because it is illegal in VHDL to declare signals in a process. These internal signals are only visible inside the architecture.
- A process statement is used after the first **begin**. An optional name such as "example:" (notice that it must be followed by a colon) may be used to name a process. The optional name may or may not be included following **end process** (this is your choice).
- The process statement contains a list of signals called the **sensitivity list**. The process wakes up or executes when an event occurs for any one of the signals *A*, *B*, *C*, *D*, *A_B*, and *C_D* in the sensitivity list—that is, when a signal in the sensitivity list changes from 1 to 0 or from 0 to 1. When a process wakes up, it executes once and then suspends (stops or does nothing) and waits for the next event to occur.
- All the inputs must be included in the sensitivity list, including the internal signals, or the process will not work properly.
- The keyword **is** may be used after the sensitivity list but is practically never used because it is optional.
- A process has a **begin** and an **end** just like an architecture.
- Each statement in a process ends with a semicolon.
- Each statement in a process is executed in the order that it is written in the process. You may consider that the process executes in zero time each time a signal in the sensitivity list changes. Writing the statements in a different order may result in a different design functionality. A simulation run can be made to verify proper design functionality.
- Using intermediate simple signal assignments in a process tends to show the circuit layout.

Note that Boolean equations *can* be used inside a process; however, conditional signal assignments (CSAs) and selected signal assignments (SSAs) *cannot* be used inside a process.

Complete VHDL code for the design entity Small_ckt using a behavioral architecture declaration with Boolean equations is obtained by combining the Listings for 4.7 and 4.8 (project: Small_ckt_Proc_Bool). The simulation for the VHDL design obtained by combining Listings 4.7 and 4.8 is the same as Waveform 4.1 shown earlier.

Listing 4.9 shows a behavioral architecture declaration using a process with an **if–then–else statement** for design entity Small_ckt.

```
architecture behavioral of Small_ckt is
begin
process (A, B, C, D)
begin
     if (A = '1' and B = '0') nor (C = '0' and D = '0') then F <= '1';
     else F <= '0';
     end if;
end process;
end behavioral;
```

LISTING 4.9 Behavioral architecture declaration for design entity Small_ckt using a process with an if–then–else statement

Things you should notice about the VHDL design in Listing 4.9:

- A process statement is used after the first **begin**. Notice that the process was not named as it was in Listing 4.8. Remember, naming a process is optional.
- The process statement contains a list of signals called the sensitivity list. The process wakes up or executes when an event occurs—that is, when a signal in the sensitivity list changes

from 1 to 0 or from 0 to 1 (any one of the signals *A*, *B*, *C*, and *D* in the list). When a process wakes up, it executes once and then suspends and waits for the next event to occur.

- A process declaration has a **begin** and an **end** just like an architecture declaration.
- Semicolons are required after the simple signal assignment statements *F* <= '1', *F* <= '0' and also after **end if** in the process.
- The conditions in the if–then–else statements are prioritized. The first output statement listed (*F* <= '1') has the highest priority and will be executed first, if the condition is true. If the condition is not true, the first output statement is skipped and the second output statement (*F* <= '0') will be executed next, and so on (for additional conditions).
- If an if–then–else statement does not have the last **else** (the **else** after the last **then**), a couple of bad things can happen: the circuit is not combinational, because the output is latched, or the circuit is not implemented correctly, because one or more inputs are not used. In the first case, the output now contains a memory and is no longer a combinational circuit. In the second case, the output of the circuit is tied to GND or V_{CC}, making the circuit nonfunctional or useless. For combinational circuits, be sure to include the last **else** (the **else** after the last **then**) so that latches or nonfunctional circuits are not inadvertently created for the outputs of the circuits.
- An if–then–else statement must be placed in a process between **begin** and **end process**.
- You may observe that the if–then–else statement is similar to the conditional signal assignment (CSA).

Complete VHDL code for the design entity Small_ckt using a behavioral architecture declaration with a process and an if–then–else statement, is obtained by combining Listings 4.7 and 4.9 (project: Small_ckt_Proc_if). The simulation for the VHDL design obtained by combining Listings 4.7 and 4.9 is the same as Waveform 4.1 shown earlier.

To handle multiple conditions, if–then–else statements can be nested (a statement within a statement). Listing 4.10 shows the code for two conditions using nested if–then–else statements.

LISTING 4.10
Nested if–then–else statements (project: example_Proc_if)

```
if x = '1' then f <= a;
else if y = '1' then f <= b;
     else f <= c;
     end if;
end if;
```

Output *F* is prioritized such that its output is *A* if *X* is 1, *B* if *X* is 0 and *Y* is 1, and *C* if *X* is 0 and *Y* is 0. The condition for signal *X* has priority over the signal *Y*. When both signal *X* and signal *Y* are inactive (or 0), then *F* is *C*.

Things you should notice about the VHDL if statements in Listing 4.10:

- Nested if–then–else statements require an **end if** to terminate each **if**. If six conditions were required then six **end ifs** would be required.
- For readability, line up each **if**, **else**, and **end if**.

The truth table for Listing 4.10 is shown in Table 4.1.

TABLE 4.1 Truth table for Listing 4.10

X	*Y*	*F*	
1	1	*A*	Highest priority
1	0	*A*	↓
0	1	*B*	
0	0	*C*	Lowest priority

To simplify nesting, it is more efficient to use an **if–then–elsif statement**. Listing 4.11 shows equivalent code as Listing 4.10 using an if–then–elsif statement.

```
if x = '1' then f <= a;
elsif y = '1' then f <= b;
else f <= c;
end if;
```

LISTING 4.11
Equivalent code as in Listing 4.10 using an if–then–elsif statement (project: example_Proc_elsif)

Things you should notice about the VHDL if statement in Listing 4.11:

- Using an if–then–elsif statement results in fewer lines of code (only one less line for this simple case).
- Only one **end if** is required to terminate an if–then–elsif statement. If six conditions were required then only one **end if** would be required when **elsif** is used rather than **else if** for each condition after the first **if**.
- For readability, line up the single **if**, all the **elsif**s, the final **else**, and the single **end if**. Be sure to use the final **else** to make the circuit combinational and not sequential. If you leave off the final **else** and compile the design by running Synthesize – XST, you will get the warning: "Found 1-bit latch for signal <f>". When you generate a latch in this manner, your circuit may have timing problems. So if you get this warning, fix your VHDL code by writing complete if statements.

A **case statement** is often preferred to nested **if statements**. This is because nested if statements and elsif statements can result in more logic gates, because the conditions are prioritized. Listing 4.12 shows a behavioral architecture declaration with a process and a case statement for design entity Small_ckt.

```
architecture behavioral of Small_ckt is
begin
process (A, B, C, D)
begin
    case (A and not B) nor (not C and not D) is
        when '1' => F <= '1';
        when '0' => F <= '0';
        when others => null;
    end case;
end process;
end behavioral;
```

LISTING 4.12
Behavioral architecture declaration for design entity Small_ckt with a process and a case statement

Things you should notice about the VHDL architecture declaration in Listing 4.12:

- A case statement must be placed in a process between **begin** and **end process**.
- The case statement examines the value of the signal that is listed after **case**—for example, (*A* **and not** *B*) **nor** (**not** *C* **and not** *D*)—and only executes the output statement to the right of the symbol => that matches the **when** value for the signal and all the other output statements are skipped.
- The conditions for a case statement are not prioritized like an if statement and thus require conditions that do not overlap or are mutually exclusive values.
- **When others** is used to ensure that all possible select values for the signal listed after **case** are included in the choice list after **when**. The std_logic values that must be included for the signal are 0, L, 1, H, Z, and -. Note that the single dash "-" represents a don't care in VHDL. The keyword **null** means do nothing—that is, perform no action.

- Order is not important, and the only output statement that is executed is the one that meets the condition.
- You may observe that the case statement is similar to the selected signal assignment (SSA).
- Observe that a semicolon is required after each signal assignment and also after the keyword **null** to terminate the case statement.

Listing 4.13 is alternate code for design entity Small_ckt with a behavioral architecture declaration with a process, Boolean equation, and a case statement:

LISTING 4.13
Behavioral architecture declaration for design entity Small_ckt with a process, a Boolean equation, and a case statement

```
architecture behavioral of Small_ckt is
    --internal signal declaration for sel (or select)
    signal sel: std_logic;
begin
process (A, B, C, D, sel)
begin
    sel <= (A and not B) nor (not C and not D);
    case sel is
        when '1' => F <= '1';
        when '0' => F <= '0';
        when others => null;
    end case;
end process;
end behavioral;
```

Things you should notice about the VHDL architecture declaration in Listing 4.13:

- The signal declaration statement for sel (or select) must be placed in the signal declaration part of the architecture, which is between **architecture** and the first **begin**. This is an internal signal, so a mode (such as **in**, **out**, or **inout**) must not be assigned.
- To use *SEL* in the process, it must be included in the sensitivity list.
- The internal signal *SEL* is assigned its value in the process after **begin** using a simple signal assignment (Boolean equation).

Complete VHDL code for the design entity Small_ckt using a behavioral architecture declaration with a case statement is obtained by combining Listings 4.7 and 4.12 (project: Small_ckt_Proc_case) or Listings 4.7 and 4.13 (project: Small_ckt_Proc_case_alt). The simulations for the VHDL designs obtained by combining Listings 4.7 and 4.12 and obtained by combining Listings 4.7 and 4.13 are the same as Waveform 4.1 shown earlier.

4.8 STRUCTURAL DESIGN STYLE

A structural architecture declaration is often referred to as a **hierarchal design approach**. The procedure we use to obtain a structural design is listed as follows:

Step 1: Partition the design into manageable hardware blocks called components.
Step 2: Write complete VHDL code to define each component with a dataflow design style or a behavioral design style.
Step 3: Write the library part and the entity declaration for the top level of the structural design.
Step 4: Declare all the internal signals and all the components between **architecture** and the first **begin** in the architecture declaration for the top level of the structural design.
Step 5: Instantiate the components (or place and connect the components) after the first **begin** in the architecture declaration for the top level of the structural design.

Step 1: Figure 4.6 shows the design entity Small_ckt partitioned or subdivided into smaller units called components.

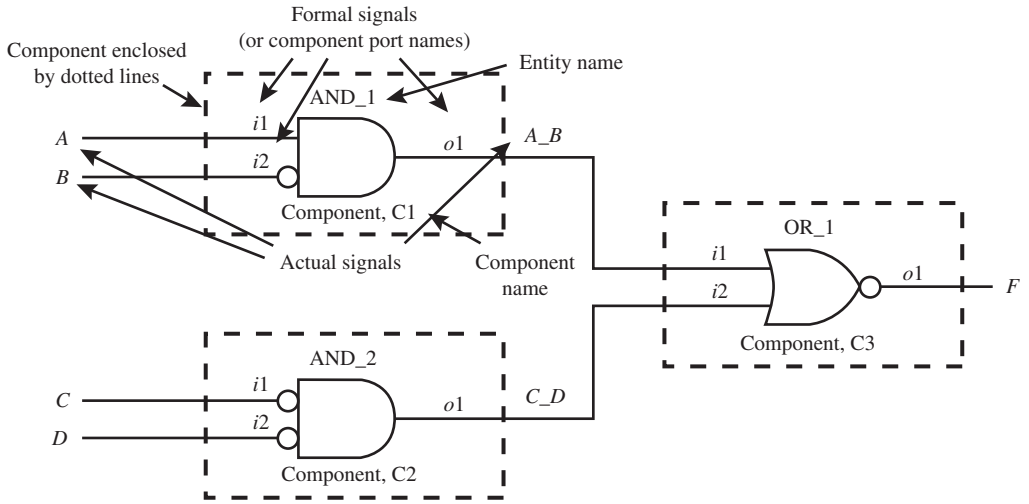


FIGURE 4.6 Design entity Small_ckt partitioned or subdivided into smaller units called components

Things you should notice about the design entity Small_ckt in Figure 4.6:

- The components are enclosed by dotted lines.
- The components are given the arbitrary entity names AND_1, AND_2, and OR_1. These names (or identifiers) are used as the names of the entities when writing the definitions for the components in step 2.
- The input and output signals of the components (component port names) are called **formal signals** and are given the arbitrary signal names *i*1, *i*2, and *o*1. Formal signals must be used when writing the definitions for the components in step 2.
- The components are given the arbitrary component names C1, C2, and C3. These names (or identifiers) are used when instantiating the components in step 5.

Step 2: Listing 4.14 shows complete VHDL code to define the components AND_1, AND_2, and OR_1. Each design should be simulated to verify correct functionality.

LISTING 4.14
Component definitions for AND_1, AND_2, and OR_1

```
--Component definition for AND_1
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_1 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end and_1;

architecture dataflow of and_1 is
begin
     o1 <= i1 and not i2;
end dataflow;

--Component definition for AND_2
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

(Continued)

```
entity and_2 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end and_2;

architecture dataflow of and_2 is
begin
     o1 <= not i1 and not i2;
end dataflow;

--Component definition for OR_1
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity or_1 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end or_1;

architecture dataflow of or_1 is
begin
     o1 <= i1 nor i2;
end dataflow;
```

Things you should notice about the VHDL component definitions in Listing 4.14:

- Complete VHDL code is written for each component definition.
- The entity name for each component is AND_1, AND_2, and OR_1, respectively.
- The formal signals (or component port names) *i*1 (short for input1), *i*2, and *o*1 (short for output1) must be used to define the components.

Step 3: Listing 4.15 shows the library part and the entity declaration for the top level of the structural design for design entity Small_ckt.

LISTING 4.15
Library part and entity declaration for design entity Small_ckt

```
--Structural Design (top level)
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Small_ckt is port (
     A, B, C, D : in std_logic;
     F : out std_logic
     );
end Small_ckt;
```

Things you should notice about the VHDL code for design entity Small_ckt in Listing 4.15:

- The name of the entity is Small_ckt, which represents the complete design or complete system.
- The signals in the entity Small_ckt are the external input signals and the external output signals for the complete design or complete system.

Steps 4 and 5: Listing 4.16 shows the **internal signal declarations** and the **component declarations**, which are placed between **architecture** and the first **begin** in the architecture declaration for the structural design—that is, design entity Small_ckt. Listing 4.16 also shows the **component instantiations**, which are placed after the first **begin**.

```
architecture structural of Small_ckt is
--internal signal declarations for A_B and C_D
    signal A_B, C_D: std_logic;
--Component declaration for and_1
component and_1 is port (
    i1, i2: in std_logic;
    o1: out std_logic
    );
end component;
--Component declaration for and_2
component and_2 is port (
    i1, i2: in std_logic;
    o1: out std_logic
    );
end component;
--Component declaration for or_1
component or_1 is port (
    i1, i2: in std_logic;
    o1: out std_logic
    );
end component;
begin
--Component placement and connections (formally called component instantiations)
    C1: and_1 port map (i1 => A, i2 => B, o1 => A_B);
    C2: and_2 port map (i1 => C, i2 => D, o1 => C_D);
    C3: or_1 port map (i1 => A_B, i2 => C_D, o1 => F);
end structural;
```

LISTING 4.16 Structural architecture declaration for design entity Small_ckt

Things you should notice about the VHDL architecture declaration in Listing 4.16:

- The signal declaration **signal** *A_B*, *C_D*: std_logic; is required to declare two internal signals *A_B* and *C_D*, which are the output signals of the two AND gates in Figure 4.6. Signals *A_B* and *C_D* are only visible inside the architecture declaration of Small_ckt. A mode (**in** or **out**) is not required for internal signals. Internal signal declarations must always be placed between **architecture** and the first **begin**.
- The component declarations are placed between **architecture** and the first **begin**.
- Notice that a component declaration has exactly the same information as the entity declaration for the component only it starts with the word **component** and is terminated by **end component**.
- Each component must be placed and connected in the architecture of the structural design. This is done via the **instantiation statement** C1: and_1 **port map** (*i*1 => *A*, *i*2 => *B*, *o*1 => *A_B*); for component 1, C2: and_2 **port map** (*i*1 => *C*, *i*2 => *D*, *o*1 => *C_D*); for component 2, and C3: or_1 **port map** (*i*1 => *A_B*, *i*2 => *C_D*, *o*1 => *F*); for component 3. This method for connecting the components is called **name association**, and has the form **formal signal** => **actual signal**.
- An alternate method for connecting components is called **positional association**. Positional association for C1 is written as C1: and_1 **port map** (*A*, *B*, *A_B*). The actual signals must be placed in *exactly* the same positions as they are listed in the component declaration. Name association is highly recommended over positional association, because it is easier to make mistakes when using positional association.

- The order of writing the instantiation statements is not important because these are concurrent statements.
- The structural architecture declaration must be terminated by the keyword **end**.

It should be noted that instantiation statements (or component instantiations) *cannot* be placed inside a process.

Complete VHDL code for the design entity Small_ckt using a structural architecture declaration is obtained by combining Listings 4.14, 4.15, and 4.16. The VHDL code for the component definitions must be placed in the same project as the VHDL code for the structural design—that is, design entity Small_ckt—to allow the VHDL code for the structural design to correctly compile. Listing 4.17 shows the complete VHDL code for the design entity Small_ckt, including the definitions of the components.

LISTING 4.17
Complete VHDL code for the design entity Small_ckt using a structural architecture declaration, including the definitions of the components (project: Small_ckt_structural).

```
--Component definition for AND_1
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity and_1 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end and_1;
architecture dataflow of and_1 is
begin
     o1 <= i1 and not i2;
end dataflow;
--Component definition for AND_2
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity and_2 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end and_2;
architecture dataflow of and_2 is
begin
     o1 <= not i1 and not i2;
end dataflow;
--Component definition for OR_1
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity or_1 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end or_1;
architecture dataflow of or_1 is
begin
     o1 <= i1 nor i2;
end dataflow;
--Structural Design (top level)
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

```
entity Small_ckt is port (
     A, B, C, D : in std_logic;
     F : out std_logic
     );
end Small_ckt;
architecture structural of Small_ckt is
     signal A_B, C_D: std_logic;
component and_1 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end component;
component and_2 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end component;
component or_1 is port (
     i1, i2: in std_logic;
     o1: out std_logic
     );
end component;
begin
     C1: and_1 port map (i1 => A, i2 => B, o1 => A_B);
     C2: and_2 port map (i1 => C, i2 => D, o1 => C_D);
     C3: or_1 port map (i1 => A_B, i2 => C_D, o1 => F);
end structural;
```

The simulation for the VHDL design in Listing 4.17 is the same as Waveform 4.1 shown earlier.

As we have pointed out, the structural design style can be treated as a hierarchal design approach by considering the components as the lower levels and the architecture declaration as the top level. In practice, we partition the design into the desired components (the lower levels of the hierarchy), write the definitions for the components, simulate each component to verify that it works as expected, and then write the architecture declaration for the structure (the top level of the hierarchy). The top level can then be simulated or run in hardware to verify that it works as expected.

In Chapter 2 you were introduced to a **flat design approach**, where each of the modules in the system was included within a single architecture declaration. Renaming component C1 as module 1, component C2 as module 2, and component C3 as module 3 in the circuit in Figure 4.6, we can write VHDL code for a flat design approach as shown in Listing 4.18 using a dataflow design style for each module.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Small_ckt_flat is port (
     a,b,c,d : in std_logic;
     f : out std_logic
     );
end Small_ckt_flat;
```

(Continued)

LISTING 4.18 Flat design approach for the circuit in Figure 4.6 using a dataflow design style for each module (project: Small_ckt_flat)

```
architecture dataflow of Small_ckt_flat is
    signal a_b, c_d : std_logic;
begin
--Module 1, AND_1
    a_b <= a and not b;
--Module 2, AND_2
    c_d <= not c and not d;
--Module 3, OR_1
    f <= a_b nor c_d;
end dataflow;
```

The simulation for the VHDL design in Listing 4.18 is the same as Waveform 4.1 shown earlier.

Observe how much simpler it is to use a flat design approach compared to a hierarchal design approach (or structural design style) for the simple circuit in Figure 4.6. By simpler, we mean fewer lines of code. For medium design projects, one may elect to use a hierarchal or a flat design approach. For a very large design project, a hierarchal design approach is very important because this approach allows individuals in a group to work on a portion of the design in a real-world situation.

In a very large design project, a flat design approach suffers because it may be hard to understand conceptually and possibly hard to modify.

4.9 IMPLEMENTING WITH WIRES AND BUSES

A **wire** and a **bus** are hardware terms. A wire carries a single bit of information. A bus, which represents a collection of wires, carries multiple bits of information. A signal on a wire and on a bus is shown in Figure 4.7.

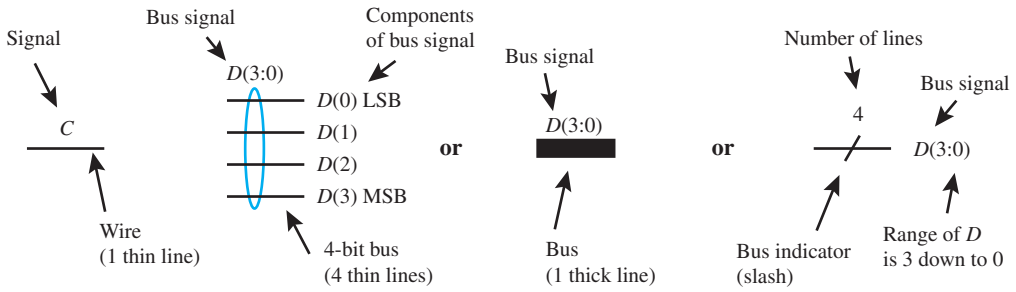


FIGURE 4.7 Signal on a wire and on a bus

In VHDL, signal *C* has a **data type** called std_logic that we have been using throughout this chapter. A nonbus signal or a signal for a single wire or line such as signal *C* may be referred to as a **scalar**. Bus signal *D*(3:0) has a **data type** called std_logic_vector (3 **downto** 0), where *D*(3) is the MSB (most significant bit) and *D*(0) is the LSB (least significant bit). The components of a bus signal *D*(3:0) in VHDL may not be written as *D*3, *D*2, *D*1, and *D*0. A bus signal or a signal for a number of lines such as signal *D*(3:0), with the range 3 down to 0, may be referred to as a **vector**.

If the bus signal in Figure 4.7 were rewritten as *D*(0:3), then *D*(0) is the MSB and *D*(3) is the LSB. In VHDL, the bus signal *D*(0:3) would be the std_logic_vector (0 **to** 3) data type. When writing VHDL code to represent hardware blocks or design entities, you must choose the correct data type for each wire and bus in the design.

A black box for a 3-to-8 decoder is shown in Figure 4.8, drawn in two different but equivalent ways. The first drawing uses thin lines to represent the individual bus wires. The second drawing uses slashes with the number of lines to represent the buses.

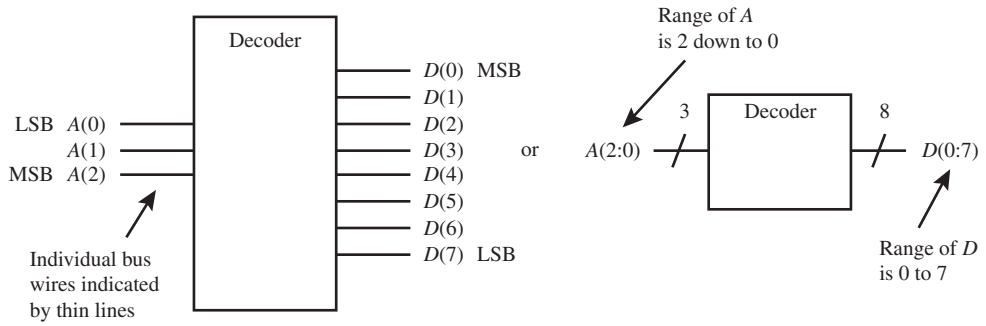


FIGURE 4.8 Black box for 3-to-8 decoder

Table 4.2 shows the truth table for the decoder.

TABLE 4.2 Truth table for 3-to-8 decoder

A(2)	*A*(1)	*A*(0)	*D*(0)	*D*(1)	*D*(2)	*D*(3)	*D*(4)	*D*(5)	*D*(6)	*D*(7)
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

A behavioral architecture declaration using a case statement is shown in Listing 4.19 for the 3-to-8 decoder. The bit pattern (or code) for bus *A*(2:0) is binary while the bit pattern for *D*(0:7) is one-hot. For a one-hot output code, only one output bit is **hot** or **active**—that is, 1—while all other output bits are inactive or 0.

LISTING 4.19
Behavioral architecture declaration for design entity Decoder using a case statement

```
architecture behavioral of Decoder is
begin
process (A)
begin
    case A is
        when "000" => D <= "10000000";
        when "001" => D <= "01000000";
        when "010" => D <= "00100000";
        when "011" => D <= "00010000";
        when "100" => D <= "00001000";
        when "101" => D <= "00000100";
        when "110" => D <= "00000010";
        when "111" => D <= "00000001";
        when others => null;
    end case;
end process;
end behavioral;
```

Things you should notice about the VHDL architecture declaration in Listing 4.19:

- Signal *A* must be read by the process, and so signal *A* must be in the sensitivity list of the process.
- Signal *A* represents a single signal consisting of 3 bits (a bus). A string of bits—that is, more than one bit—must be included in double quotation marks. Recall that a single bit is included in single quotation marks. The data type for bus signal *A* is std_logic_vector (2 **downto** 0). This must be the data type listed in the entity declaration for the signal *A*.
- Signal *D* represents a single signal consisting of 8 bits (a bus). A string of bits—that is, more than one bit—must be included in double quotation marks. The data type for bus signal *D* is std_logic_vector (0 **to** 7). This must be the data type listed in the entity declaration for the signal *D*.
- A **when others clause** is required at the end of the choice list of the case statement. **When others** is used to ensure that all possible select values for signal *A* are included in the choice list after **when**. The std_logic values that must be included for signal *A* are 0, L, 1, H, Z, and -. Note that the single dash "-" represents a don't care in VHDL. The keyword **null** means do nothing—that is, perform no action.

The library clause, use clause, and entity declaration for the design entity Decoder are shown in Listing 4.20.

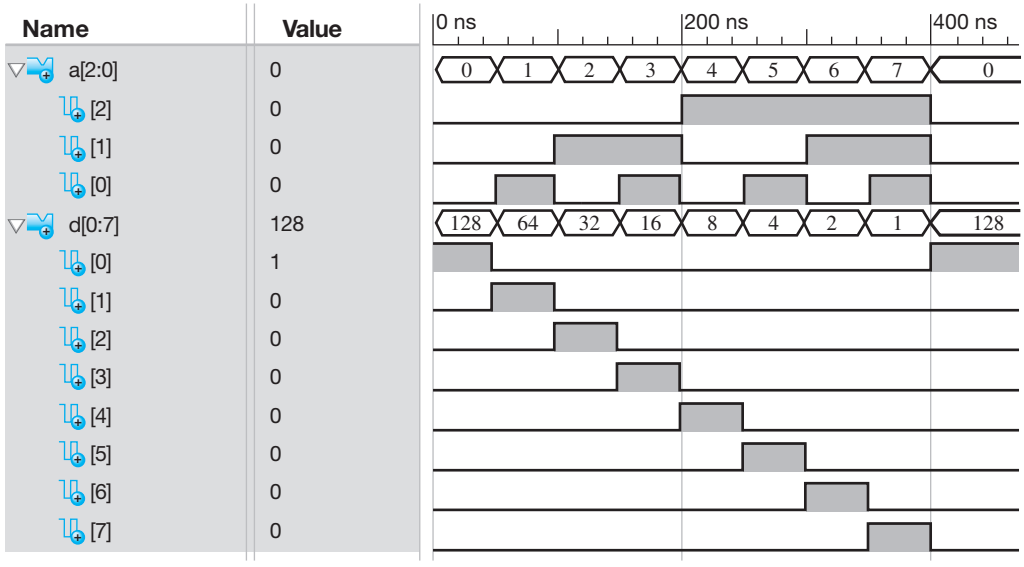
LISTING 4.20
Library clause, use clause, and entity declaration for design entity Decoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Decoder is port (
    A : in std_logic_vector (2 downto 0);
    D : out std_logic_vector (0 to 7)
    );
end Decoder;
```

To obtain complete VHDL code for the design entity Decoder, combine the listings for 4.19 and 4.20 (project: Decoder_3t8_Proc_case).

The waveforms shown in Waveform 4.2 represent the correct functionality of design entity Decoder. This can easily be confirmed by comparing Waveform 4.2 with the truth table for the decoder in Table 4.2.

WAVEFORM 4.2
Simulation for the correct functionality of design entity Decoder (project: Decoder_3t8_Proc_case)



Rather than declaring a bus in the entity for signal *A* as shown in Listing 4.20, individual signals *A*2, *A*1, and *A*0 can be declared and then grouped together to form a bus using an expression called an **aggregate**. An aggregate is a collection or group of elements. The VHDL code in Listing 4.21 shows how to form a bus signal *A* with the aggregate (*A*2, *A*1, *A*0). The bus signal *A* is needed for the case statement in the design.

LISTING 4.21
Forming a bus with an aggregate (project: Decoder_3t8_Proc_case_agg)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Decoder is port (
     A2,A1,A0 : in std_logic;
     D : out std_logic_vector (0 to 7)
     );
end Decoder;

architecture behavioral of Decoder is
     signal A: std_logic_vector (2 downto 0);

begin
     A <= (A2,A1,A0);
process (A)

begin
     case A is
          when "000" => D <= "10000000";
          when "001" => D <= "01000000";
          when "010" => D <= "00100000";
          when "011" => D <= "00010000";
          when "100" => D <= "00001000";
          when "101" => D <= "00000100";
          when "110" => D <= "00000010";
          when "111" => D <= "00000001";
          when others => null;
     end case;
end process;
end behavioral;
```

Things you should notice about the VHDL code in Listing 4.21:

- The data type for signals *A*2, *A*1, and *A*0 is std_logic as declared in the entity.
- An internal signal *A* is declared as std_logic_vector (2 downto 0) and placed between **architecture** and the first **begin**, because a bus signal is needed for the case statement.
- An aggregate (*A*2, *A*1, *A*0) is assigned to the internal signal *A* and placed in the architecture after the first **begin**.
- The bus signal *A* is now available for use in the case statement.

The simulation for the VHDL design in Listing 4.21 is the same as Waveform 4.2 shown earlier.

Forming a bus with an aggregate is a handy concept to know. The placement of the elements (or the order of the elements) in the aggregate is very important. The left-most element in the list is the MSB, while the right-most element in the list is the LSB. In Listing 4.21, making the assignment *A* <= (*A*0, *A*1, *A*2) would be an error for the logic circuit, because *A* is defined as a std_logic_vector (2 downto 0), assuming that *A*2(MSB), *A*1, *A*0(LSB).

An aggregate can also be used to form a larger bus using the individual elements of the bus to form the aggregate. Do not use a vector in an aggregate; however, you may use the individual elements to form a larger bus with a larger range. Suppose we wanted to form the bus *A*(4:0). If

some of the values of the individual elements of the bus [say, *A*(4) and *A*(3)] must be added to *A*(2 downto 0) to form the larger bus then the correct way to write the aggregate is *A* <= (*A*(4), *A*(3), *A*(2), *A*(1), *A*(0)). Writing the aggregate as *A* <= (*A*(4), *A*(3), *A*(2 **downto** 0)) would be a syntax error. If you want to form a larger bus with vectors you may use the concatenation operator "&" as follows: *A* <= (*A*(4) & *A*(3) & *A*(2 **downto** 0)). Just like the aggregate, the order in which the elements are placed in the list is very important—that is, the left-most element in the list is the MSB and the right-most element in the list is the LSB.

4.10 VHDL EXAMPLES

Figure 4.9 shows an alphabetical list of **keywords** that support synthesis.

FIGURE 4.9 An alphabetical list of keywords that support synthesis

A	**abs, all, alias, and, architecture, array, attribute**
B	**begin, block, body, buffer**
C	**case, component, configuration, constant**
D	**downto**
E	**else, elsif, end, entity, exit**
F	**for, function**
G	**generate, generic, group**
I	**if, in, inout, is**
L	**library, literal, loop**
M	**map, mod**
N	**nand, next, nor, not, null**
O	**of, or, others, out**
P	**package, port, procedure, process**
R	**range, record, rem, return, rol, ror**
S	**select, signal, sla, sll, sra, srl, subtype**
T	**then, to, type**
U	**until, use**
V	**variable**
W	**wait, when, while, with**
X	**xnor, xor**

Figure 4.10 shows a list of **supported operators**.

FIGURE 4.10 A list of supported operators

Group	Supported Operators
1. Logical	(**and, or, nand, nor, xor, xnor**)
2. Relational	(=, /=, <, <=, >, >=)
3. Shifting	(**sll, srl, sla, sra, rol, ror**)
4. Adding	(+, −, &)
5. Unary signing	(+, −)
6. Multiplying	(*, /, **mod, rem**)
7. Miscellaneous	(**, **abs, not**)

Note: Without parentheses, operators in group 7 have the highest precedence, followed by group 6, and so on. down to the lowest precedence, or group 1. Within each group, there is no operator precedence, and precedence must be established by parentheses. In an expression without parentheses, operators are applied left to right.

4.10.1 Design with Scalar Inputs and Outputs

Figure 4.11 shows the design entity for an OR gate called OR_2. This design uses scalar inputs and outputs.

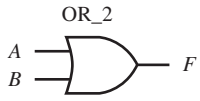


FIGURE 4.11 Design entity for an OR gate called OR_2

The library part, entity declaration, and partial architecture declaration for design entity OR_2 are shown in Listing 4.22.

LISTING 4.22 The library part, entity declaration, and partial architecture declaration for design entity OR_2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OR_2 is port (
     A, B : in std_logic;
     F : out std_logic
     );
end OR_2;

architecture design_style of OR_2 is
begin
     <Architecture body>
end design_style;
```

Any one of the architecture design styles presented in Listing 4.23 can be used to complete the partial architecture declaration in Listing 4.22. These design styles use scalar inputs and outputs.

LISTING 4.23 Architecture design styles with scalar inputs and outputs

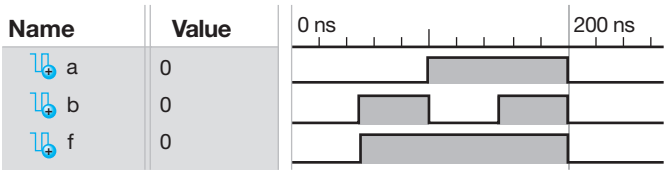
```
Boolean equation:
   --place the Boolean equation between begin and end process for
   --a behavioral design
      F <= A or B;

Conditional signal assignment (CSA):
   --Note: parentheses are required around A or B
      F <= '1' when (A or B) = '1' else
           '0';
   Note: CSA is the concurrent equivalent of the if statement.

Selected signal assignment (SSA):
      with A or B select
         F <= '1' when '1',
              '0' when '0',
              '0' when others;
   Note: SSA is the concurrent equivalent of the case statement.

If statement:
   --place the if statement between begin and end process
   --Note: parentheses are required around A or B
      if (A or B) = '1' then F <= '1';
      else F <= '0';
      end if;
   Note: if statement is the sequential equivalent of the CSA.
```

(Continued)

```
Case statement:
   --place the case statement between begin and end process
      case A or B is
         when '1' => F <= '1';
         when '0' => F <= '0';
         when others => null;
      end case;
   Note: case statement is the sequential equivalent of the SSA.
```

The waveform diagram shown in Waveform 4.3 represents the correct functionality of design entity OR_2.

WAVEFORM 4.3 Simulation for the correct functionality of design entity OR_2 (project: or_2_Bool)

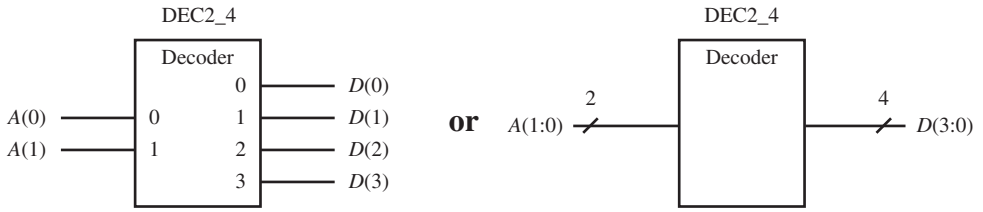


All of the architecture design styles presented in Listing 4.23 provide the same simulation as shown in Waveform 4.3.

4.10.2 Design with Vector Inputs and Outputs

Figure 4.12 shows the design entity for a 2-to-4 decoder called DEC2_4. This design uses vector inputs and outputs.

FIGURE 4.12 Design entity for a 2-to-4 decoder called DEC2_4



The library part, entity declaration, and partial architecture declaration for design entity DEC2_4 are shown in Listing 4.24.

LISTING 4.24 The library part, entity declaration, and partial architecture declaration for design entity DEC2_4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DEC2_4 is port (
     A : in std_logic_vector (1 downto 0);
     D : out std_logic_vector (3 downto 0)
     );
end DEC2_4;

architecture design_style of DEC2_4 is
begin
     <Architecture body>
end design_style;
```

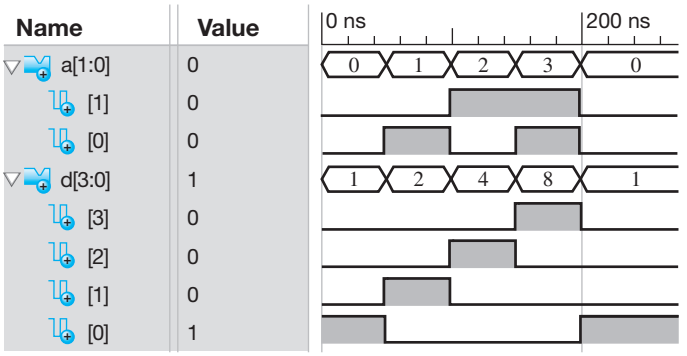
Any one of the architecture design styles presented in Listing 4.25 can be used to complete the partial architecture declaration in Listing 4.24. These design styles use vector inputs and outputs.

LISTING 4.25
Architecture design styles with vector inputs and outputs

```
Boolean equations:
      D(0) <= not A(1) and not A(0);
      D(1) <= not A(1) and A(0);
      D(2) <= A(1) and not A(0);
      D(3) <= A(1) and A(0);

Conditional signal assignment (CSA):
      D <= "0001" when A = "00" else
           "0010" when A = "01" else
           "0100" when A = "10" else
           "1000";
   Note: CSA is the concurrent equivalent of the if statement.

Selected signal assignment (SSA):
      with A select
         D <= "0001" when "00",
              "0010" when "01",
              "0100" when "10",
              "1000" when "11",
              "0001" when others;
   Note: SSA is the concurrent equivalent of the case statement.

If statement:
   --place the if statement between begin and end process
      if    A = "00" then  D <= "0001";
      elsif A = "01" then  D <= "0010";
      elsif A = "10" then  D <= "0100";
      else                 D <= "1000";
      end if;
   Note: if statement is the sequential equivalent of the CSA.

Case statement:
   --place the case statement between begin and end process
      case A is
         when "00" => D <= "0001";
         when "01" => D <= "0010";
         when "10" => D <= "0100";
         when "11" => D <= "1000";
         when others => null;
      end case;
   Note: case statement is the sequential equivalent of the SSA.
```

The waveform diagram shown in Waveform 4.4 represents the correct functionality of design entity DEC2_4.

WAVEFORM 4.4 Simulation for the correct functionality of design entity DEC2_4 (project: DEC2_4_Bool)



All of the architecture design styles presented in Listing 4.25 provide the same simulation as shown in Waveform 4.4.

4.10.3 Common VHDL Constructs

Listing 4.26 is a brief list of common VHDL constructs that we have used in this chapter. Other constructs exist and are available by clicking on the light bulb icon (language templates) in Xilinx ISE Projector Navigator.

LISTING 4.26 A brief list of common VHDL constructs

```
   Library Part:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

   Entity Declaration:
entity <Entity name> is port (
       <Port name> : <Mode> <Type>;
       <Other ports>. . .
       );
end <Entity name>;

   Architecture Declaration:
architecture <Architecture name> of <Entity name> is
   --Signal_and_component declarations (internal signal
   --declarations, component declarations)
begin
    <Architecture body>
end <Architecture name>;

   Boolean equation:
<Signal name> <= <Expression>;

   Conditional signal assignment (CSA):
<Signal name> <= <Expression> when <Condition> else
                 <Expression> when <Condition> else
                 <Expression>;

   Selected signal assignment (SSA):
with <Choice expression> select
     <Signal name> <= <Expression> when <Choices>,
                      <Expression> when <Choices>,
                      <Expression> when others;
```

```
   Process Declaration:
process (<All input signals separated by commas>)
begin
   <Boolean equation, if statement, case statement >;
end process;

   If statement:
if <Condition> then <Statement>;
elsif <Condition> then <Statement>;
else <Statement>;
end if;

   Case statement:
case (<Signal name>) is
   when ". . .00" => <Statement>;
   when ". . .01" => <Statement>;
   when ". . .10" => <Statement>;
   when ". . .11" => <Statement>;
   . . .
   when others => <Statement>;
end case;

   Component declaration:
component <Component name> port (
   <Port name> : <Mode> <Type>;
   <Other ports>. . .
   );
end component;

   Component instantiation:
<Instance name> : <Component name> port map (<Port name> =>
                                              <Signal name>,
                                              <Other ports>. . .);
```

PROBLEMS

Section 4.2 VHDL

4.1 What is a design entity?

4.2 What are the names of the three parts of a design entity in VHDL?

4.3 What part of a design entity for VHDL has data type definitions, functions, and procedures?

4.4 What part of a design entity for VHDL specifies the interface or the external inputs and outputs of a digital circuit?

4.5 What part of a design entity for VHDL specifies the functional composition or functionality of a digital circuit?

Section 4.3 The Library Part

4.6 What two clauses are required for the library part of a design entity in VHDL?

4.7 What two libraries are implicitly declared (built in) for VHDL designs?

4.8 What library must be added to a VHDL design to make it visible to the design?

4.9 Where is your design placed after it is compiled?

4.10 What library stores the logical and relational operators?

4.11 What library stores the 9-value data type called std_logic?

4.12 What is the main reason for using the IEEE library?

Section 4.4 The Entity Declaration

4.13 What can an entity declaration be thought of, in terms of circuit theory?

4.14 What are the three main features of a black box?

4.15 When writing VHDL, how do you insert a comment in the code?

4.16 What is the definition of the term *synthesis* in VHDL?
4.17 Given the identifiers, Bq_1, P_w_, B 1, and 2B_3, which ones are incorrect? Why they are incorrect?
4.18 What is the data type for a signal using the standard logic 1164 package?
4.19 List the values for the data type std_logic that support synthesis.
4.20 What is the mode for an output signal?
4.21 List the values for the data type bit.

Section 4.5 The Architecture Declaration

4.22 What part of a design entity deals with the contents of a black box?
4.23 What are the three design styles in VHDL that can be used in an architecture declaration?
4.24 What are the three signal assignments that can be used in a dataflow design style?
4.25 What are signal assignments called that are evaluated at the same time in VHDL?
4.26 What design style uses a process in an architecture declaration?
4.27 What are the equations and statements that can be used in a behavioral design style?
4.28 What is an annotated circuit or schematic?
4.29 What design style uses components in an architecture declaration?
4.30 How does a structural architecture declaration create the structure for a circuit?
4.31 What is a system in VHDL?

Section 4.6 Dataflow Design Style

4.32 What is the signal assignment symbol?
4.33 How is a simple signal assignment statement terminated in VHDL?
4.34 Write the VHDL signal assignment for each of the following Boolean functions:
a. $F1 = A + B \cdot \overline{C}$
b. $F2 = \overline{A} \cdot B + \overline{C} \cdot D$
c. $F3 = (A \oplus \overline{B}) \cdot C + D$
4.35 Write the Boolean function for each of the following VHDL signal assignments:
a. *F*1 <= *A* **and** *B* **or** *C*
b. *F*2 <= *A* **or** *B* **and not** *C*
c. *F*3 <= *A* **nand** *B* **or** *C* **xor not** *D*
4.36 Describe how to check for VHDL design correctness for a function.
4.37 What are the keywords used in a conditional signal assignment?
4.38 What are the keywords used in a selected signal assignment?
4.39 A single bit in VHDL must be surrounded by what?
4.40 What must be used around a Boolean expression when it is compared to a bit using a relational operator?
4.41 How is a conditional signal assignment (CSA) terminated in VHDL?
4.42 How is a selected signal assignment (SSA) terminated in VHDL?

Section 4.7 Behavioral Design Style

4.43 In what order are assignment statements evaluated in a process?
4.44 Is the following statement true or false? A complete process is a concurrent statement that is evaluated at the same time as another complete process or at the same time as other concurrent statements.
4.45 When does a process wake up or execute?
4.46 What does a process do while it is waiting for an event to occur?
4.47 Is the following statement true or false? It is correct to use a conditional signal assignment or a selected signal assignment inside a process.
4.48 Is it necessary to name a process?
4.49 Where is an if–then–else statement placed in a process?
4.50 How many **end if**s are required when **if–then–else statements** are nested?
4.51 How many **end if**s are required to terminate an **if–then–elsif statement**?
4.52 Which is usually preferred: a case statement or nested if statements? Give one reason this may be true.
4.53 Where is a case statement placed in a process?
4.54 What does null mean in VHDL?
4.55 Is a mode required for an internal signal?
4.56 A signal name or identifier placed between **case** and **is** in a cast statement must also appear another placed in the VHDL code for the case statement to run. Where?

Section 4.8 Structural Design Style

4.57 What is another name for a structural design style?
4.58 What is the first step in the procedure we use to obtain a structural design?
4.59 How are components defined in a structural design?
4.60 Where are the component declarations placed in a structural design?
4.61 Where are the components placed and connected in a structural design?
4.62 What four items are used to identify each component?
4.63 What signals must be used in the entity declaration of a structural design?
4.64 Where are internal signals placed in a structural design?
4.65 Describe a component declaration in relationship to an entity for the definition of a component.
4.66 Given a formal signal *I*0, which must be connected to an actual signal *X*0, select the correct port map statement.
a. *X*0 => *I*0
b. *I*0 => *X*0
4.67 What method for connecting components is recommended in the text?
4.68 What is the order for writing instantiation statements?
4.69 Can instantiation statements (or component instantiations) be placed inside a process?
4.70 Where do you have to place the component definitions for a structural design so that the structural design will compile correctly?

Section 4.9 Implementing with Wires and Buses

4.71 How much information does a single wire carry?

4.72 What is another name for a signal that carries a single bit of information?

4.73 How much information does a bus carry?

4.74 What is a collection of wires called that carries multiple bits of information?

4.75 What is another name for a signal that carries multiple bits of information?

4.76 Write all the components for the signal *A*(5:0).

4.77 Write the data type for the bus signal *B*(2:0) in VHDL code.

4.78 Write the data type for the bus signal *C*(0:7) in VHDL code.

4.79 What is a one-hot code for a decoder?

4.80 What is an aggregate?

4.81 Use the individual signals *X*, *Y*, and *Z*, which range from MSB down to LSB, to write an aggregate to form a bus *M*(2:0) in VHDL code.

4.82 Use the individual signals *P*, *Q*, *R*, and *S*, which range from LSB to MSB, to write an aggregate to form a bus *N*(0:3) in VHDL code.

4.83 Use the signals *C*(4) and *C*(3:0), which range from MSB down to LSB, to write an aggregate to form a bus *O*(4:0) in VHDL code.

4.84 Use the concatenation operator to form a bus *M*(2:0) for the individual signals *X*, *Y*, and *Z*, which range from MSB down to LSB.

4.85 Use the concatenation operator to form a bus *N*(0:3) for the individual signals *P*, *Q*, *R*, and *S*, which range from LSB to MSB.

4.86 Use the concatenation operator to form a bus *O*(4:0) for the signals *C*(4) and *C*(3:0), which range from MSB down to LSB.

Section 4.10 VHDL Examples

4.87 Write an entity declaration for the black box in Figure P4.87.

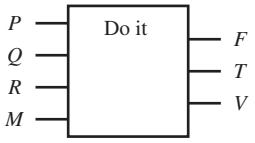


FIGURE P4.87

4.88 Draw the black box for the entity declaration in Listing P4.88.

```
entity hardware is port (
     X, Y, Z: in std_logic;
     F0, F1: out std_logic
     );
end hardware;
```

LISTING P4.88

4.89 Write a dataflow architecture declaration using a Boolean equation for the 2-bit comparator circuit in Figure P4.89, which compares *D*1 *D*0 with *B*1 *B*0 to see if they have the same values, if so $F = 1$, else $F = 0$. Use the name Comparator as the design entity.

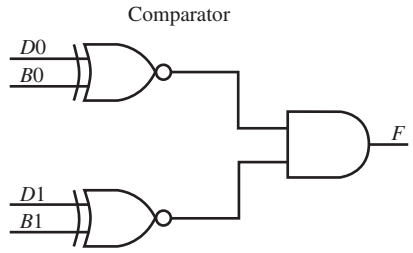


FIGURE P4.89

4.90 Write a dataflow architecture declaration using a conditional signal assignment (CSA) for design entity Comparator in problem 4.89.

4.91 Write a dataflow architecture declaration using a selected signal assignment (SSA) statement for design entity Comparator in problem 4.89.

4.92 Write a behavioral architecture declaration using an if statement for design entity Comparator in problem 4.89.

4.93 Write a behavioral architecture declaration using a case statement for design entity Comparator in problem 4.89.

4.94 Write an entity declaration for design entity Comparator in problem 4.89.

4.95 Write the library and use clauses for the package IEEE .STD_LOGIC_1164 for design entity Comparator in problem 4.89.

4.96 Show a functionally correct simulation for the complete VHDL design for design entity Comparator in problem 4.89. You can use any of the design styles presented in the text to obtain the simulation.

4.97 Write a dataflow architecture declaration using Boolean equations for the 2-to-4 decoder/demultiplexer called DMUX2_4 in Figure P4.97 (project: DMUX2_4).

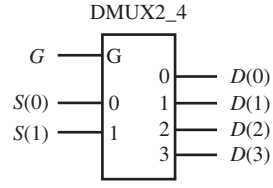


FIGURE P4.97

4.98 Write a dataflow architecture declaration using a conditional signal assignment (CSA) for design entity DMUX2_4 in problem 4.97.

4.99 Write a dataflow architecture declaration using a selected signal assignment (SSA) for design entity DMUX2_4 in problem 4.97.

4.100 Write a behavioral architecture declaration using an if statement for design entity DMUX2_4 in problem 4.97.

4.101 Write a behavioral architecture declaration using a case statement for design entity DMUX2_4 in problem 4.97.

4.102 Write an entity declaration for design entity DMUX2_4 in problem 4.97.

4.103 Write the library and use clauses for the package IEEE .STD_LOGIC_1164 for design entity DMUX2_4 in problem 4.97.

4.104 Show a functionally correct simulation for the complete VHDL design for design entity DMUX2_4 in problem 4.97. You can use any of the design styles presented in the text to obtain the simulation.

4.105 Write a dataflow architecture declaration using a Boolean equation for the 4-to-1 multiplexer with an enable input called MUX_E4_1 in Figure P4.105 (project: MUX_E4_1).

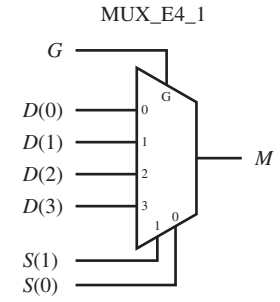


FIGURE P4.105

4.106 Write a dataflow architecture declaration using a conditional signal assignment (CSA) for design entity MUX_E4_1 in problem 4.105.

4.107 Write a dataflow architecture declaration using a selected signal assignment (SSA) for design entity MUX_E4_1 in problem 4.105.

4.108 Write a behavioral architecture declaration using an if statement for design entity MUX_E4_1 in problem 4.105.

4.109 Write a behavioral architecture declaration using a case statement for design entity MUX_E4_1 in problem 4.105.

4.110 Write an entity declaration for design entity MUX_E4_1 in problem 4.105.

4.111 Write the library and use clauses for the package IEEE. STD_LOGIC_1164 for design entity MUX_E4_1 in problem 4.105.

4.112 Show a functionally correct simulation for the complete VHDL design for design entity MUX_E4_1 in problem 4.105. You can use any of the design styles presented in the text to obtain the simulation.

4.113 Write the definitions for the XNOR_1 and AND_1 components for the design entity Comparator_struc in Figure P4.113 using a dataflow design style with Boolean equations.

4.114 Show a functionally correct simulation for the XNOR_1 and AND_1 components for design entity Comparator_struc in problem 4.113.

4.115 Write the library and use clauses for the package IEEE. STD_LOGIC_1164 for design entity Comparator_struc in problem 4.113.

4.116 Write an entity declaration for design entity Comparator_struc in problem 4.113.

4.117 Write an architecture declaration for design entity Comparator_struc in problem 4.113.

4.118 Show a functionally correct simulation for the complete VHDL design for design entity Comparator_struc in problem 4.113. Use the structural design styles presented in the text to obtain the simulation.

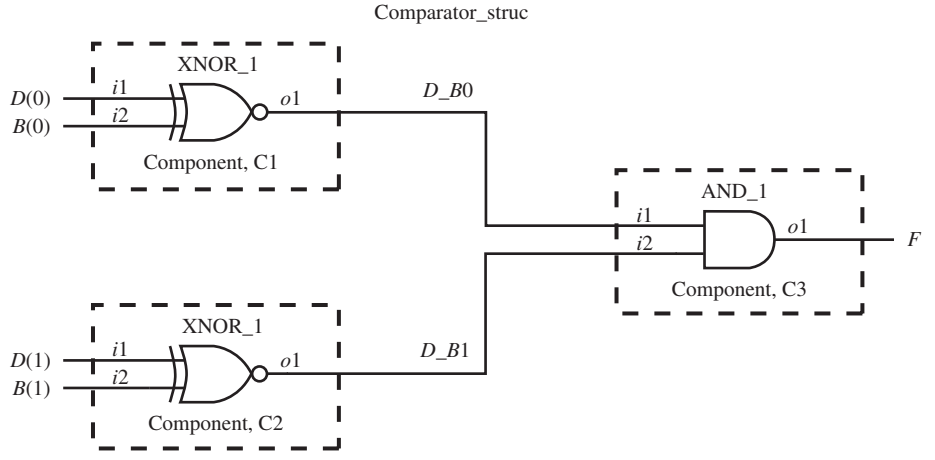


FIGURE P4.113

Chapter 5

Bistable Memory Device Design with VHDL

Chapter Outline

5.1 INTRODUCTION

In this chapter, we present the design of **bistable memory devices**. The term *bistable* means that there are two stable states, allowing these devices to store, save, or capture the value for a logic 1 or a logic 0. We discuss the design and analysis of various bistable devices including **set–reset (S-R) latches**, **D latches**, and **edge-triggered D flip-flops**, and present their operating characteristics. The design of a simple clock is also presented. The techniques and devices that are presented in this chapter are used in the next chapter to design **synchronous sequential logic circuits**.

5.2 ANALYZING AN S-R NOR LATCH

A **register** is a digital component that can temporarily store single or multiple bits. A **latch** is the simplest circuit form of a single-bit register. Two cross-coupled NOR gates or two cross-coupled NAND gates can be used to form a set–reset (S-R) latch. First, we cover a light switch, because the operation of a light switch is analogous to the operation of a latch with cross-coupled NOR gates.

5.2.1 Simple On/Off Light Switch

Think of a simple light switch on the wall. Consider that there are two signals: one signal named S to set or turn the light on, and the other signal named R to reset or turn the light off. Each of the two signals S and R has two values. We can push the switch lever in one direction to set the light to on ($S = 1$), and then stop pushing the lever and the light stays on ($S = 0$). We can push

the switch lever in the opposite direction to reset the light to off ($R = 1$), and then stop pushing the lever and the light stays off (R = 0). Not pushing the switch lever to set the light to on ($S = 0$) and not pushing the switch lever to reset the light to off ($R = 0$) maintains the status. The concept is simple. Try it out yourself dynamically using the light switch on the wall of your room. Figure 5.1 shows five different conditions for a simple light switch (mechanical latch) with the values marked for S and R to turn the light on if the switch lever is pushed up or to turn the light off if the switch lever is pushed down.

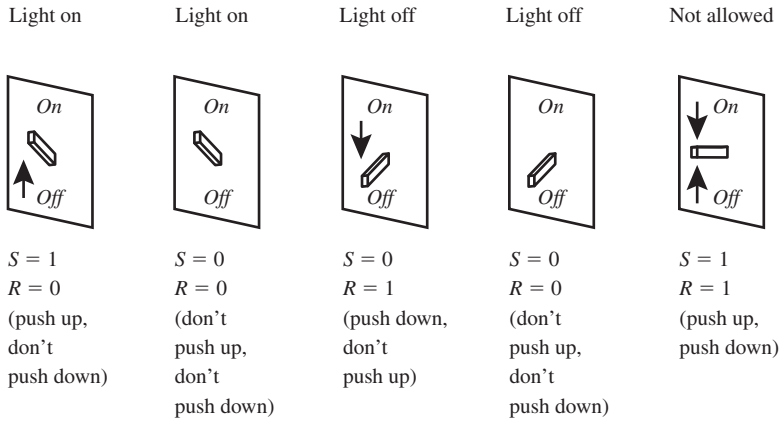


FIGURE 5.1 Five different conditions for a simple light switch

An arrow in the figure represents both the direction and the force being applied to change the switch position. If no arrow is present, then no force is being applied. Once a force is applied, the switch stores the set or reset position, and the force can then be removed. This type of switch has a memory because it remembers the last force that was applied. A momentary push-button switch does not have a memory, but a light switch has a memory, so we don't have to stand near it and continuously press the switch to light the room.

The light switch in Figure 5.1 snaps and stores the on position so we can stop pushing and the light stays on. Likewise, the light switch snaps and stores the off position so we can stop pushing and the light stays off.

Table 5.1 shows the **characteristic table** for a light switch.

TABLE 5.1 Characteristic table for a light switch

S	R	Light condition
0	0	Hold (on or off)
0	1	Off
1	0	On
1	1	Not allowed

A light switch and two cross-coupled NOR gates, which is a digital circuit called an **S-R NOR latch**, have similar characteristic tables. Both a light switch and an S-R NOR latch have a memory—that is, they store a value.

Up to this point, we have only covered **combinational logic circuits**, which are circuits that do not contain feedback and therefore do not have memory capability. Circuits with feedback are called **sequential logic circuits** and have memory capability. Sequential logic circuits are

more difficult to analyze and design because of feedback. A few different techniques for analyzing a basic S-R NOR latch are present as follows:

1. Circuit delay model.
2. Characteristic table.
3. Characteristic equation.
4. PS/NS (present-state/next-state) table.
5. Timing diagram.

5.2.2 Circuit Delay Model for an S-R NOR Latch

The gate-level circuit for an S-R NOR latch is shown in Figure 5.2a. Figure 5.2b shows the logic symbol for an S-R NOR latch.

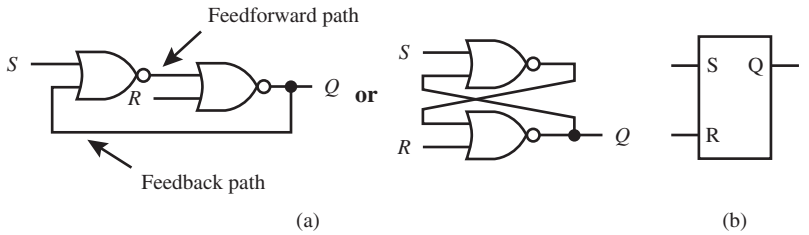


FIGURE 5.2 S-R NOR latch: (a) gate-level circuit; (b) logic symbol

Figure 5.3 shows a circuit delay model for an S-R NOR latch. The gate-level circuit for an S-R NOR latch is a digital circuit that has its **present-state output *Q*** fed back to its input. This is the secret of retaining data or storing data for an S-R NOR latch.

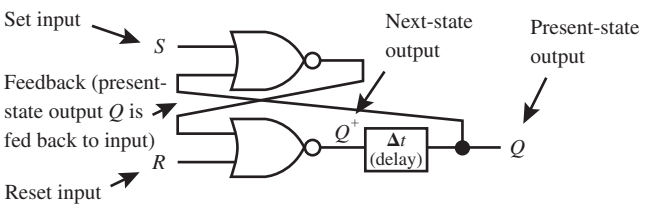


FIGURE 5.3 Circuit delay model for the S-R NOR latch

If an S-R NOR latch is set, its output Q goes to 1, and if it is reset, its output Q goes to 0. Each NOR gate has a propagation delay time. The worst-case delay time for the circuit is Δt and represents the longest delay time from the input S to the output Q or $2t_p$, assuming that each gate has a delay of t_p. In our delay model, we simply show the worst-case delay of the circuit in the delay box. Now, the gates are considered to have no delay because the delay box shows the delay for the circuit. Treating circuits as though they have lumped parameters (in this case, the parameter is delay time) is usually easier to explain than treating circuits with distributed parameters, as they really exist.

In our delay model, if the value of either S or R is changed to 1, the **next-state output Q^+** changes immediately to reflect the change caused by S or R. The present-state output Q is the value of the output of the circuit before S or R changes. The present-state output changes to a new present-state output that is equal to the values of Q^+ only after delay time Δt has expired, or $Q = Q^+$ after Δt. In our light switch analogy, the bulb takes time to go from off to on or glow (just like Q takes time Δt to go to a 1), or to go from on to off or stop glowing (just like Q takes time Δt to go to a 0). The circuits a through e in Figure 5.4 represents the same conditions as those shown for the simple light switch in Figure 5.1.

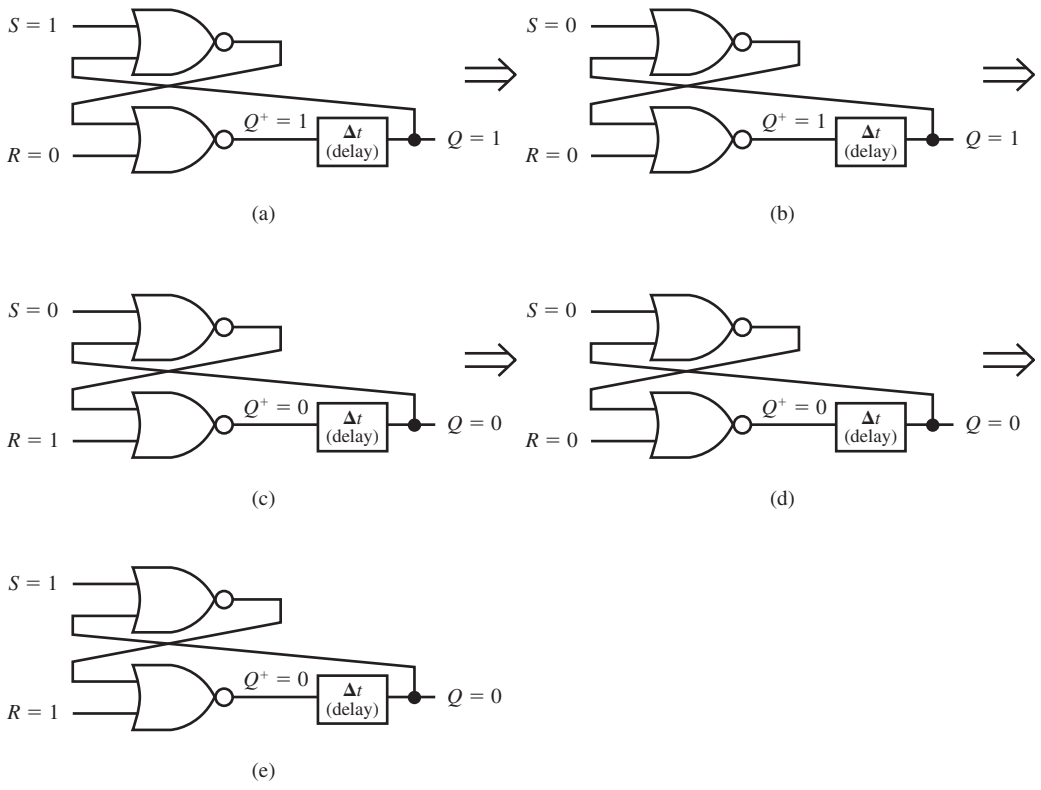


FIGURE 5.4 Five different conditions for an S-R NOR latch

The condition shown in Figure 5.4e is normally not allowed because the output cannot be both set and reset at the same time. If you try to set and reset an S-R NOR latch at the same time, the output of the circuit goes to the reset state ($Q = 0$); that is, the reset input overrides the set input. Because the reset input overrides the set input, an S-R NOR latch circuit is referred to as **reset dominant**.

At startup, when power is first applied to an S-R NOR latch circuit and the S and R inputs are at 0, the Q output of the circuit is momentarily unknown, because the Q output may settle at a 1 or a 0.

5.2.3 Characteristic Table for an S-R NOR Latch

An S-R NOR latch has active high inputs. For the positive logic convention, 1 = high (H) and 0 = low (L). If $S = 1$, then Q goes to 1 or high, provided $R = 0$. If $R = 1$, then Q goes to 0 or low, provided $S = 0$. Table 5.2 shows the **characteristic table** for an S-R NOR latch.

TABLE 5.2 Characteristic table for an S-R NOR latch

			Comment
S	R	Q^+	Next state
0	0	Q	Present state (high or low)
0	1	0	Low
1	0	1	High
1	1	0	Reset dominant (normally not allowed)

Observe that if you try to set and reset the S-R NOR latch at the same time, the output of the circuit goes to the reset state, because the circuit is reset dominant.

5.2.4 Characteristic Equation for an S-R NOR Latch

The Boolean equation for a latch circuit is called its **characteristic equation** because it describes in equation form how the latch circuit is constructed. Using the delay model for an S-R NOR latch, we can write the characteristic equation from the circuit as shown in Figure 5.5.

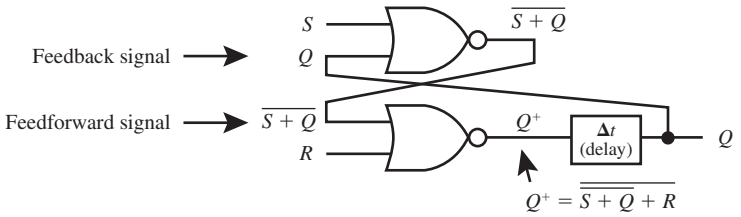


FIGURE 5.5 Characteristic equation for an S-R NOR latch

From analyzing the circuit, you can see that $Q^+ = \overline{\overline{S + Q} + R}$. The variable Q on the right side of the equation is the present-state output, while the variable Q^+ on the left side is the next-state output.

A combinational logic equation always has the form $F(A,B,C, \ldots)$, which shows that the function is only dependent on the inputs $A,B,C, \ldots$. A **sequential logic equation** (the equation for a circuit that has a memory) has the form $F^+(F,A,B,C, \ldots)$, which shows that the function is dependent on its own output, as an input, and other inputs. The function Q^+ has the form $Q^+(Q,S,R)$, which shows that the function is dependent on its own output, as an input, and the inputs S and R. When the output of an S-R NOR latch becomes stable, the present-state output signal Q catches up or becomes Q^+ after the delay time Δt, or $Q = Q^+$ after Δt.

5.2.5 PS/NS Table for an S-R NOR Latch

A truth table for a sequential logic circuit is called a **present-state/next-state table** or **PS/NS table**. The table gets its name from the fact that the present-state (PS) output signal Q is listed as an input (feedback input) followed by the next-state (NS) output signal Q^+. The PS/NS table for an S-R NOR latch can be obtained in two ways: by expanding the characteristic table for the S-R NOR latch or by filling in the table using the SOP form of the characteristic equation. We can rewrite the characteristic equation in SOP form (AND/OR form) as follows:

$$Q^+ = \overline{\overline{Q + S} + R} = \overline{\overline{Q + S} \cdot \overline{R}} = (Q + S) \cdot \overline{R} = Q \cdot \overline{R} + S \cdot \overline{R}$$

The characteristic equation shows us that $Q^+ = 1$ only when $Q \cdot \overline{R} = 1$ OR $S \cdot \overline{R} = 1$; otherwise, $Q^+ = 0$.

Table 5.3 shows the PS/NS table for an S-R NOR latch.

TABLE 5.3 PS/NS table for an S-R NOR latch

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

⟹

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	0
1 1 1	0

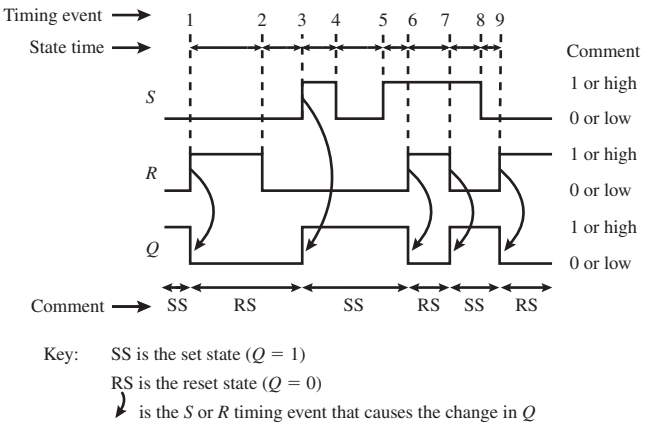
5.2.5 Timing Diagram for an S-R NOR Latch

Events that may cause the output of a sequential logic circuit to change are called **timing events**. A change in either external input *S* or *R* provides a timing event for an S-R NOR latch. **Ideal timing diagrams**, or waveform diagrams, show the logic values of signals associated with a circuit in terms of 1s and 0s. An ideal timing diagram does not show **rise times** and **fall times**, and each gate is assumed to have a 0 gate delay. This is the simplest way to draw a timing diagram.

The rise time or fall time of a signal is usually defined as the time it takes the signal to change between the 10% and 90% values of the transition. The **pulse width** of a positive pulse is usually defined as the time it takes the signal to go from its 50% value on the rising edge of the pulse to its 50% value on the falling edge of the pulse. The pulse width for a negative pulse is defined in a similar way. To simplify timing diagrams, gate delays may be represented by an average propagation delay time $t_p = (t_{PHL} + t_{PLH})/2$, or in the simplest case, 0 gate delays may be used. The delay t_{PHL} represents the delay time through a gate when the output changes from a high (H) value to a low (L) value, while the delay t_{PLH} represents the delay time through a gate when the output changes from a low (L) value to a high (H) value. These delay times are not always equal.

Waveform 5.1 shows an ideal timing diagram for an S-R NOR latch circuit.

WAVEFORM 5.1 Ideal timing diagram for an S-R NOR latch



In Waveform 5.1 we list the inputs *S* and *R* first, followed by the output *Q*. The *Q* output must follow the input signals according to the characteristic table for the S-R NOR latch. Time progresses from left to right. A timing event occurs each time an external input changes, either from a 1 to a 0 or changes from a 0 to a 1. The **state time** is the time between each timing event.

Listing 5.1a shows a complete VHDL design for an S-R NOR latch.

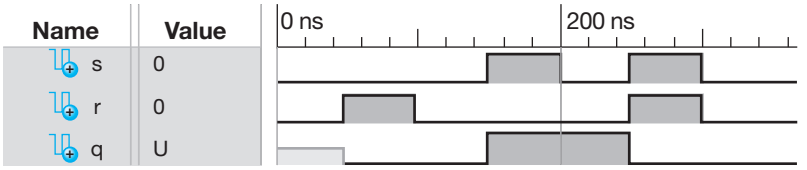
LISTING 5.1A Complete VHDL design for an S-R NOR latch (project: S_R_NOR_LATCH_Bool)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity S_R_NOR_LATCH is port (
     S, R : in std_logic;
     Q : inout std_logic
     );
end S_R_NOR_LATCH;

architecture dataflow of S_R_NOR_LATCH is
begin
     Q <= (Q and not R) or (S and not R);
end dataflow;
```

Things you should notice about the VHDL design in Listing 5.1a:

- The output signal *Q* is specified as mode **inout** in the entity.
- Output signal *Q* must be read because it appears on the right side of the assignment statement; hence it must be mode **in**. Because output signal *Q* also appears on the left side of the assignment statement, it must also be mode **out**. The keyword **inout** is used to specify both an input and an output mode. The keyword **buffer** is an alternate name that can be used for the keyword **inout** for signals that are not bidirectional.
- A dataflow design style is used with a Boolean equation. The Boolean equation is the characteristic equation for an S-R NOR latch.

Waveform 5.2 shows a correct simulation (a behavioral simulation) of design entity S_R_NOR_LATCH.



WAVEFORM 5.2 Correct simulation of design entity S_R_NOR_LATCH

Remember: At startup, if *S* or *R* is not asserted (i.e., is not 1), then the *Q* may be momentarily undefined (its output is between a 0 and a 1) as shown in the Waveform diagram for *Q*. Observe that the simulation in Waveform 5.2 follows the characteristic table for the S-R NOR latch shown in Table 5.2. This proves that the VHDL code for the design is correct.

Listing 5.1b shows an alternate way to design an S-R NOR latch.

LISTING 5.1B Alternate way to design an S-R NOR latch (project: S_R_NOR_LATCH_Bool_int)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity S_R_NOR_LATCH_int is port (
    S, R : in std_logic;
    Q : out std_logic
    );
end S_R_NOR_LATCH_int;

architecture dataflow of S_R_NOR_LATCH_int is
    signal Q_int: std_logic;
begin
    Q_int <= (Q_int and not R) or (S and not R);
    Q <= Q_int;
end dataflow;
```

Things you should notice about the VHDL design in Listing 5.1b:

- The output signal *Q* is specified as mode **out** in the entity.
- Without using the mode **inout**, we can substitute a dummy signal *Q_int* (which stands for *Q_internal*) for the signal *Q* in the Boolean equation for the latch. The dummy signal is an internal signal and must be specified in the design between **architecture** and the first **begin** by writing **Signal** Q_int: std_logic;.
- The output signal *Q* is generated by the Boolean equation *Q* <= Q_int;.

Listing 5.1b provided the same simulation waveforms as shown in Waveform 5.2. The circuits produced by the Xilinx ISE software for Listings 5.1a and 5.1b are also the same. To keep our VHDL code as simple as possible, we have elected to use mode **inout** throughout the text to write our VHDL code for outputs that must also be feed back as inputs, because this results in fewer lines of code to type and also requires fewer variables.

5.3 ANALYZING AN S-R NAND LATCH

In this section, the properties of a basic S-R NAND latch are presented. You should observe that a basic S-R NAND latch has active low inputs, which is opposite to a basic S-R NOR latch, which has active high inputs. You should also observe that a basic S-R NAND latch is set dominant rather than reset dominant like a basic S-R NOR latch.

5.3.1 Circuit Delay Model for an S-R NAND Latch

The gate-level circuit for an S-R NAND latch and its logic symbol is shown in Figures 5.6a and 5.6b with multiple *S* and *R* inputs. We used bubbled input OR gates to represent the NAND gates because these equivalent gate forms help remind us that the inputs to an S-R NAND latch are active low inputs. This type of latch is sometimes referred to as an $\overline{S}$-$\overline{R}$ NAND latch as a reminder. The logic symbol in Figure 5.6b is used later in this chapter to draw the circuit for a D flip-flop. Observe that the feedforward path provides $\overline{Q}$.

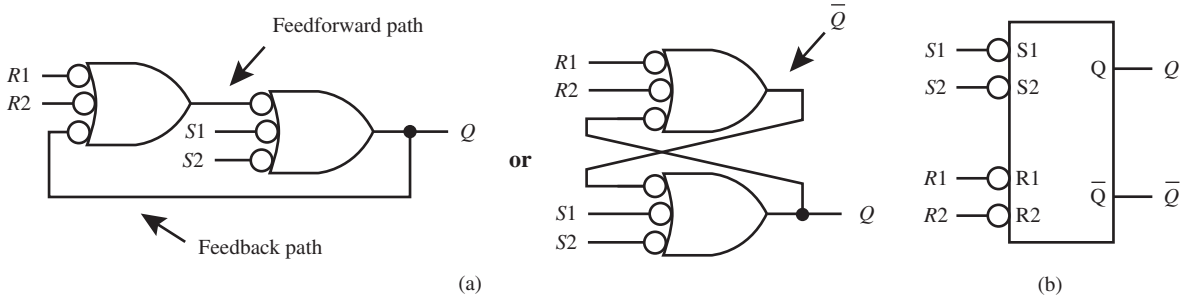
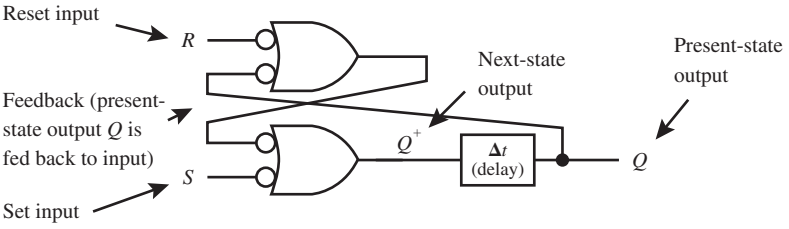


FIGURE 5.6 S-R NAND latch: (a) gate-level circuit; (b) logic symbol

Figure 5.7 shows a circuit delay model of an S-R NAND latch.

FIGURE 5.7 Circuit delay model for an S-R NAND latch



At startup, when power is first applied to an S-R NAND latch circuit and the *S* and *R* inputs are at 1, the *Q* output of the circuit is momentarily unknown, because the *Q* output may settle at a 1 or a 0.

5.3.2 Characteristic Table for an S-R NAND Latch

Table 5.4 shows the characteristic table for an S-R NAND latch.

TABLE 5.4 Characteristic table for an S-R NAND latch

			Comment
S	*R*	Q^+	Next state
0	0	1	Set dominant (normally not allowed)
0	1	1	High
1	0	0	Low
1	1	*Q*	Present state (high or low)

Observe that if you try to set and reset the S-R NAND latch at the same time, the output of the circuit goes to the set state, because the circuit is set dominant.

5.3.3 Characteristic Equation for an S-R NAND Latch

Using the delay model for the S-R NAND latch in Figure 5.7, one can write the characteristic equation from the circuit as $Q^+ = \overline{\overline{R} + \overline{Q}} + \overline{S}$. The variable Q on the right side of the equation is the present-state output, while the variable Q^+ on the left side is the next-state output.

5.3.4 PS/NS Table for an S-R NAND Latch

Table 5.5 shows the PS/NS table for an S-R NAND latch, which can be written from its characteristic table or from the SOP form of its characteristic equation.

TABLE 5.5 PS/NS table for an S-R NAND latch

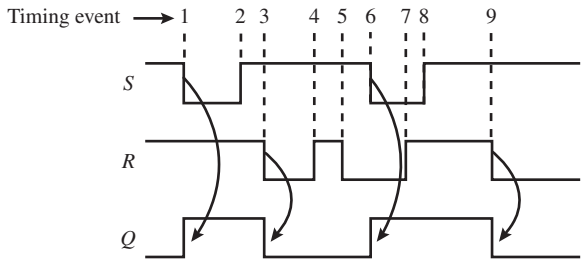
S	R	Q^+
0	0	1
0	1	1
1	0	0
1	1	Q

⇒

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

5.3.5 Timing Diagram for an S-R NAND Latch

Waveform 5.3 shows an ideal timing diagram for an S-R NAND latch.



WAVEFORM 5.3 Ideal timing diagram for an S-R NAND latch

Listing 5.2 shows a complete VHDL design for an S-R NAND latch.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity S_R_NAND_LATCH is port (
     S, R : in std_logic;
     Q : inout std_logic
     );
end S_R_NAND_LATCH;

architecture dataflow of S_R_NAND_LATCH is
begin
     Q <= (R and Q) or not S;
end dataflow;
```

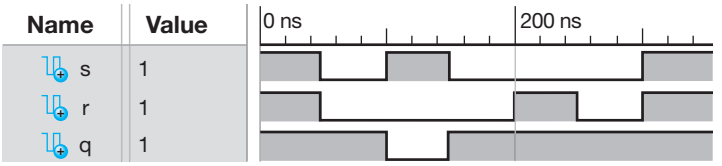
LISTING 5.2 Complete VHDL design for an S-R NAND latch (project: S_R_NAND_LATCH_Bool)

Things you should notice about the VHDL design in Listing 5.2:

- The output signal Q is listed as mode **inout** in the entity because this allows Q to be read as an input and also an output.
- A dataflow design style is used with a Boolean equation. The Boolean equation is the characteristic equation for an S-R NAND latch.

Waveform 5.4 shows a correct simulation of design entity S_R_NAND_LATCH.

WAVEFORM 5.4 Correct simulation of design entity S_R_NAND_LATCH



Observe that the waveform diagram in Waveform 5.4 follows the characteristic table for the S-R NAND latch shown in Table 5.4. This proves that the VHDL code for the design is correct.

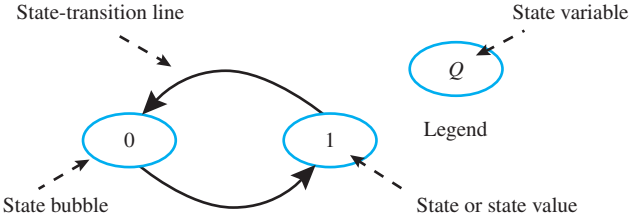
5.4 DESIGNING A SIMPLE CLOCK

In this section, we present the design of a simple circuit called a **clock**. A clock provides a sequence of pulses at its output. Clocks are used in the design of synchronous sequential logic systems. For example, your PC (personal computer) uses a clock to operate the digital logic circuits inside the computer in a synchronous manner—that is, synchronized with the clock. The output of a clock is used in the operation of D latches and flip-flops, which will be covered later. The control input to a D Latch and a D flip-flop is driven by a system clock.

A state diagram is a graphical way of describing a sequential logic circuit—that is, a logic circuit with feedback. A circle or oval is used to represent each state of the circuit. A state-transition line (a directed line segment) is used to represent the flow of the circuit when it changes from one state to the next state. The four major parts of a state diagram are the state bubbles, state-transition lines, state variables, and states or state values.

The state diagram for a simple clock is shown in Figure 5.8.

FIGURE 5.8 State diagram for a clock



If the clock is in the reset state ($Q = 0$) it goes to the set state ($Q = 1$) after a brief delay time. It then goes back to the reset state and continues in this manner as long as power is supplied to the circuit. Because the clock switches from 0 to 1 to 0, a clock is sometimes called an **oscillator**. Notice that there are no external inputs. If there were external input signals, they would be placed beside the state-transition lines. What does a clock circuit look like? First, we need to obtain the equation for the clock to see how to design it.

It is common in digital design to describe a logic element by its state diagram or its PS/NS table. The state diagram, or PS/NS table, can then be used to obtain the equation or equations

for the logic element. By observing the state diagram, we can write the PS/NS table for the clock as shown in Table 5.6.

Q	Q^+
0	1
1	0

TABLE 5.6 PS/NS table for the clock (oscillator)

From the PS/NS table, we can write the characteristic equations for the clock as

$Q^+ = \Sigma m(0) = m0 = \overline{Q}$.

Figure 5.9 shows the gate-level circuit design for the clock using the 1s of the function Q^+—that is, $Q^+ = \overline{Q}$.

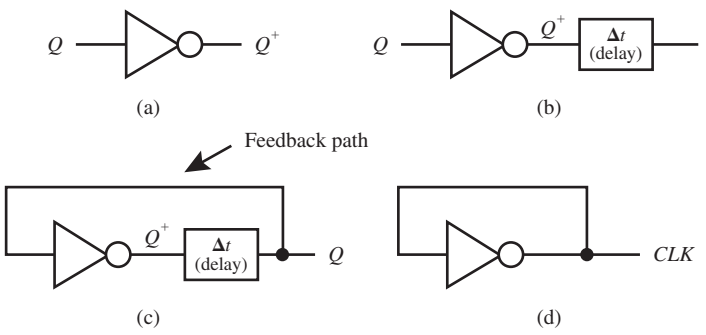
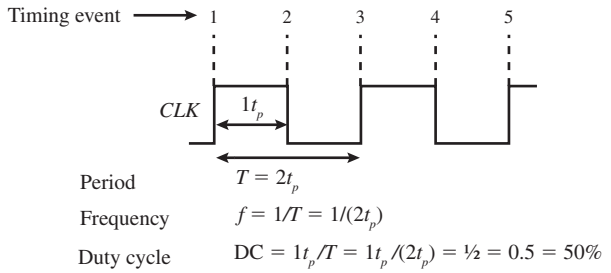


FIGURE 5.9 Gate-level circuit design for the clock

The output signal of a clock circuit is usually called *CLK*. The circuit delay, $1t_p$, is the propagation delay time for the NOT gate used in the circuit. The **period** is $T = 2t_p$, the **frequency of oscillation** is $f = 1/T = 1/(2t_p)$, and the **duty cycle** is DC = 50%, as shown in the timing diagram in Waveform 5.5.



WAVEFORM 5.5 Ideal timing diagram for the clock circuit

The duty cycle of a periodic square waveform is defined as the on time (waveform is 1 or high) divided by "the period of the waveform." For the timing diagram of the clock circuit shown in Waveform 5.5, the duty cycle is 50% because the on time is one-half the period of the waveform.

The propagation delay time t_p of the NOT gate in the clock circuit determines the period and thus the frequency of the circuit. Providing a longer delay time increases the period and thus provides a lower frequency. Adding buffers in cascade with the NOT gate will provide a longer delay time. Using an odd number of NOT gates will also provide a longer delay time to ensure that the clock circuit will oscillate at a lower frequency, when constructed with hardware.

Adding an RC (resistor capacitor) circuit as shown in Figure 5.10 allows for frequency adjustment.

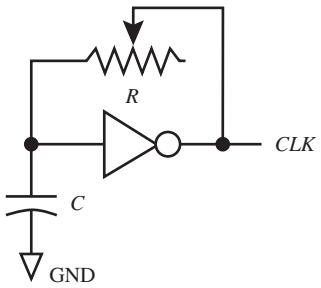


FIGURE 5.10 Clock circuit with an RC circuit added for frequency adjustment

The variable resistor *R* and a fixed capacitor *C* allow us to adjust the delay time and thus raise or lower the frequency of the output signal *CLK*.

Listing 5.3 shows a complete VHDL design for a clock circuit with a *DISABLE* input signal. When *DISABLE* is 1, the output of the clock circuit *CLK* is 0, and when *DISABLE* is 0, the output of the clock circuit *CLK* oscillates at a very high unknown frequency. The circuit for this clock is simply a NOR gate with feedback. In practice, you cannot create a clock with just a NOR gate as we are doing here, because the frequency is too high and is not predictable or controllable.

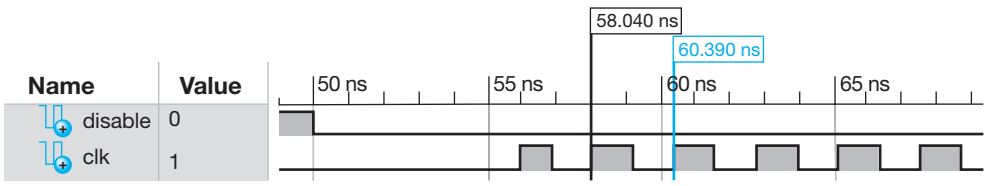
LISTING 5.3 Complete VHDL design for a clock circuit with a *DISABLE* input signal (project: CLOCK)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CLOCK is port (
     DISABLE : in std_logic;
     CLK : inout std_logic
     );
end CLOCK;

architecture dataflow of CLOCK is
begin
--Clock Generator
     CLK <= CLK nor DISABLE;
end dataflow;
```

Things you should notice about the VHDL design in Listing 5.3:

- The output signal *CLK* is listed as mode **inout** in the entity because this allows *CLK* to be read as an input and also an output.
- A dataflow design style is used with a Boolean equation. The Boolean equation is for a NOR gate with feedback from its output back to one of its inputs.

Waveform 5.6 shows a correct **post-route simulation** of design entity CLOCK.



WAVEFORM 5.6 Post-route simulation for the correct functionality of design entity CLOCK

Observe that the period for one cycle of the clock frequency in Waveform 5.6 is 2.35 ns which is a frequency $f = 1/T = 1/(2.35 \times 10^{-9}\text{s}) \approx 0.4255$ GHz. This frequency is extremely high, compared to the operating frequency of most FPGAs, and may not be usable by the circuits on the FPGA chip. A behavioral simulation uses VHDL code to obtain the simulation for a circuit.

A behavioral simulation is just a functional simulation and has no timing information for device delays. When a behavioral simulation fails to provide an output, try a post-route simulation.

A post-route simulation first creates the placed and routed circuit using the VHDL code and then obtains the simulation. A post-route simulation is a functional simulation with complete timing information (device and internal wiring delays).

The best possible clock to use is a **crystal clock oscillator** that vibrates at the desired frequency when voltage is applied across the crystal. Crystal clock oscillators are much more accurate than the clock circuits in Figures 5.9 and 5.10. Five-place accuracy is common for crystal clock oscillators. Less accurate programmable silicon oscillators provide about three-place accuracy. Crystal clock oscillators and programmable silicon oscillators provide relatively high frequencies of oscillation. We will use a clock oscillator that provides 50 MHz for many of the lab experiments. A divider circuit is used to lower the frequency to observe outputs on LEDs. We will present divider circuits later.

5.5 DESIGNING A D LATCH

A **gated latch** circuit can be created by adding a control input C to an S-R latch circuit. The control input is used to retain or hold the present-state output value. A gated latch circuit is **level sensitive**, because its output is dependent on the logic level that is applied to the control input. Once we design a gated latch circuit, we can add an additional NOT gate and turn the circuit into a **gated D latch** circuit, which is simply referred to as a **D latch**.

5.5.1 Gated S-R Latch Circuit Design

To retain or hold the present-state output Q at either 0 or 1 in an S-R NOR latch when a control input called C is 0 (or low) is easy to do. Simply add two AND gates at the inputs to the S-R NOR latch as shown in Figure 5.11a. To retain or hold the present-state output Q at either 0 or 1 in an S-R NAND latch when a control input called C is 0 (or low) is also easy to do. Simply add two NAND gates at the inputs to the S-R NAND latch as shown in Figure 5.11b. Figure 5.11c shows the logic symbol for a gated S-R latch.

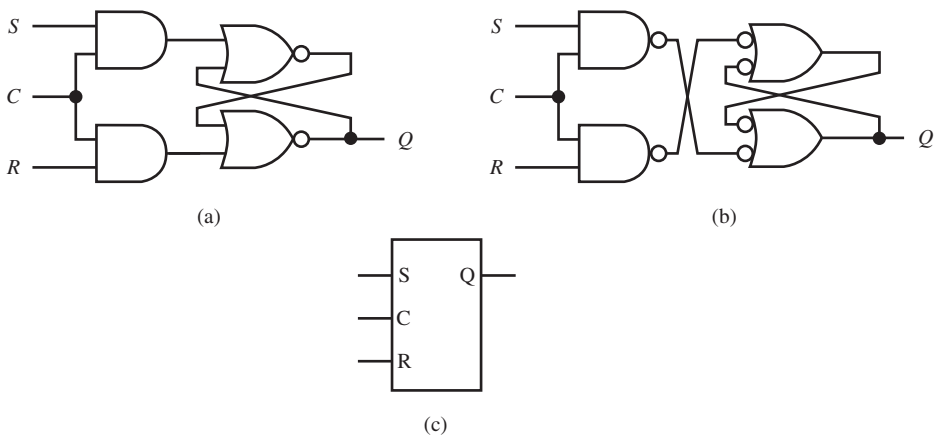


FIGURE 5.11 Gated S-R latch: (a) implemented with an S-R NOR latch; (b) implemented with an S-R NAND latch; (c) logic symbol

In the circuit in Figure 5.11a and 5.11b, output Q retains or holds the present-state output value when C is 0. Output Q goes to 1 when C is 1, S is 1, and R is 0. Output Q goes to 0 when C is 1, S is 0, and R is 1.Output Q does not change when C is 1, S is 0, and R is 0.

S and R should not be 1 at the same time C is 1. Each circuit acts differently when C is 1, S is 1, and R is 1. This is not a good situation, because the circuits act the same in all other respects.

These circuits only act the same provided we restrict the operation of the circuits by never allowing the condition C is 1, S is 1, and R is 1 to occur.

The operation for a gated S-R latch implemented with an S-R NOR latch is shown in the characteristic table in Table 5.7a. The operation for the gated S-R latch implemented with an S-R NAND latch is shown in the characteristic table in Table 5.7b. Observe that the two circuits act the same except in the very last line of their characteristic tables. In the last line of the characteristic table in Table 5.7a, Q is 0, because an S-R NOR latch is reset dominant. In the last line of the characteristic table in Table 5.7b, Q is 1, because the S-R NAND latch is set dominant.

TABLE 5.7 Characteristic table for a gated S-R latch: (a) implemented with an S-R NOR latch; (b) implemented with an S-R NAND latch

Gated S-R latch (using an S-R NOR latch)

C	S	R	Q^+
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	0

(a)

Gated S-R latch (using an S-R NAND latch)

C	S	R	Q^+
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	1

(b)

The best solution to this dilemma (the circuits do not act the same in all cases) is to redesign the circuit such that the condition S and R are both 1 when C is 1 can never occur. The redesigned circuit is called a gated D latch or simply a D latch.

5.5.2 D Latch Circuit Design with S-R Latches

When a NOT gate is added to the circuits in Figures 5.11a and 5.11b, as shown in Figures 5.12a and 5.12b, the condition S and R are both 1 when C is 1 can never occur because of the NOT gate. The S input is labeled D, which represents the *DATA* input, and the R input is supplied by the NOT gate. Figure 5.12*c* shows the logic symbol for a D latch.

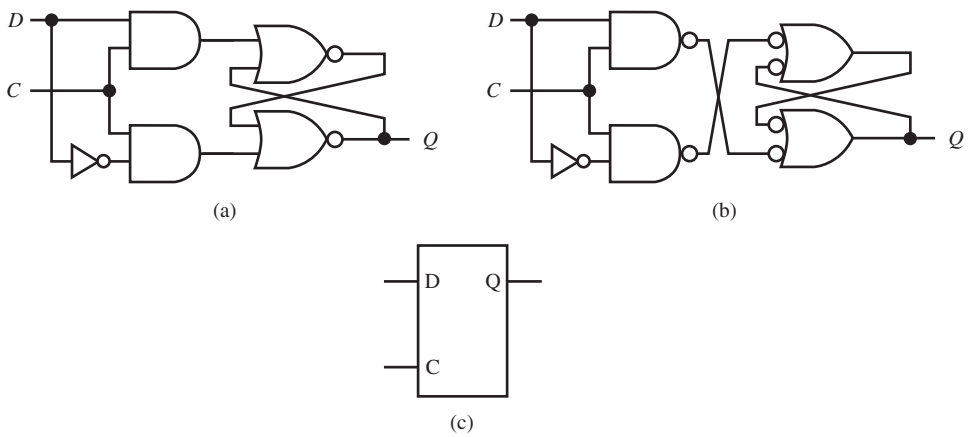


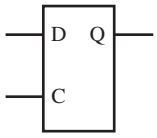
FIGURE 5.12 D latch: (a) implemented with an S-R NOR latch; (b) implemented with an S-R NAND latch; (c) logic symbol

The characteristic table for the D latch implemented with an S-R NOR latch or implemented with an S-R NAND latch is shown in Table 5.8.

TABLE 5.8 Characteristic table for a D latch

C	D	Q^+	Comment (level sensitive)
0	0	Q	No change
0	1	Q	No change
1	0	0	Q^+ follows D
1	1	1	Q^+ follows D

for



In Table 5.8, when C goes to 0, output Q^+ stores the last value of D. This is called the **storage mode**. When C is 1, the output is enabled, so output Q^+ follows input D. This latter case is called the **transparent mode** or **see-through mode** of the D latch. A D latch is **level sensitive** because its output is dependent on the logic level that is applied to the control input.

5.5.3 D Latch Circuit Design via the Characteristic Table for a D Latch

Table 5.9 shows the PS/NS table for a D Latch.

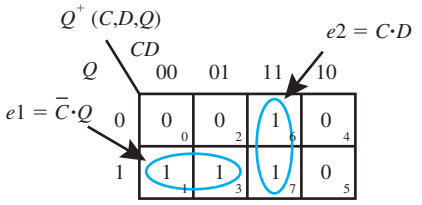
TABLE 5.9 PS/NS table for a D latch

C	D	Q^+
0	0	Q
0	1	Q
1	0	0
1	1	1

$\Rightarrow$

C	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

Figure 5.13 shows the manual K-reduction for the next state output Q^+ for a D latch.



$$Q^+(C,D,Q) = e1 + e2$$
$$= \overline{C}\cdot Q + C\cdot D$$

FIGURE 5.13 Manual K-map reduction for the next state output Q^+ for a D latch

Figure 5.14 shows an AND/OR circuit for a D latch using the reduced next-state output Q^+ written in SOP form. The reduced AND/OR circuit for the D latch requires fewer gates than a D latch implemented with an S-R latch.

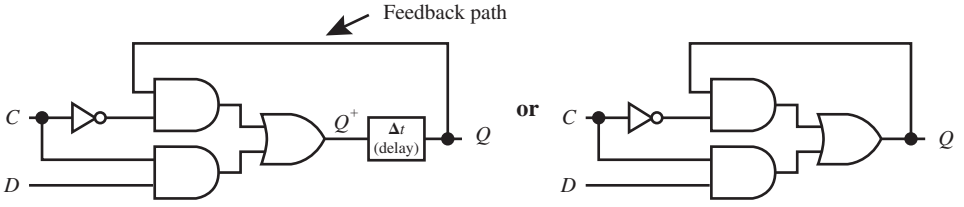


FIGURE 5.14 Reduced AND/OR circuit for a D latch

It might not be obvious until you look a little closer, but the output $Q^+ = \overline{C}\cdot Q + C\cdot D$ contains the Boolean expression for a 2-to-1 MUX. Figure 5.15 shows a D latch using or implemented with a 2-to-1 MUX.

FIGURE 5.15 D Latch using a 2-to-1 MUX

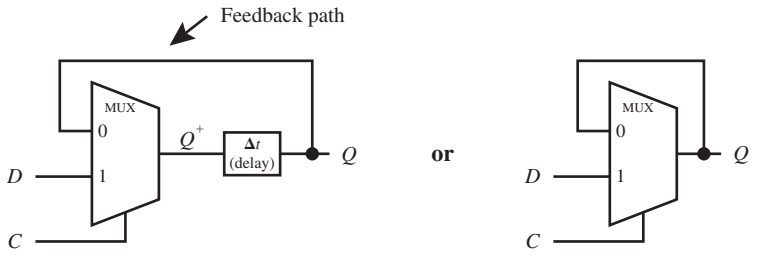
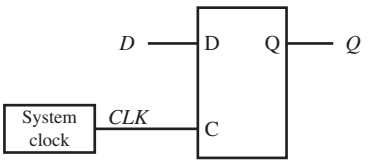


Figure 5.16 shows the logic symbol for a D latch driven by a system clock.

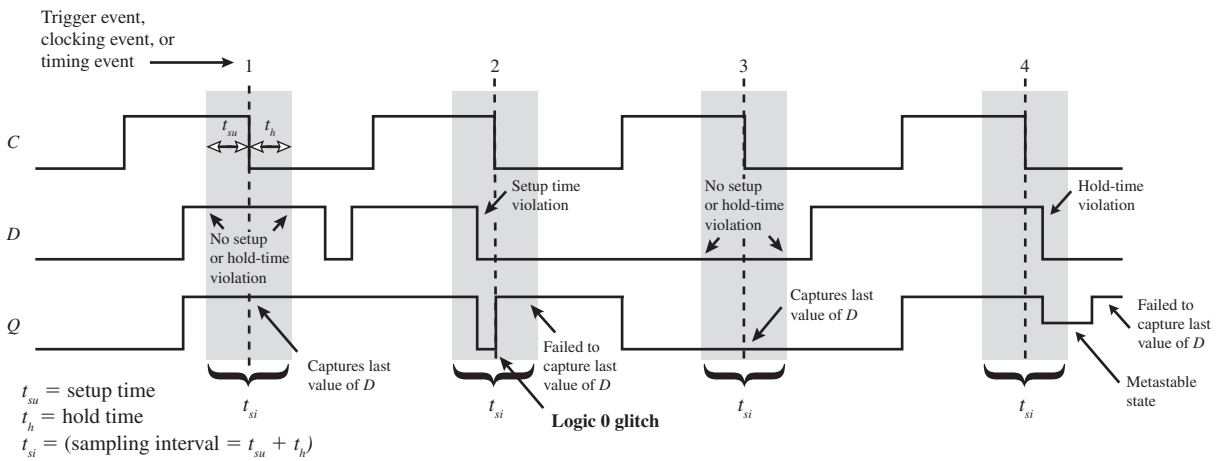
FIGURE 5.16 Logic symbol for a D latch driven by a system clock



A D latch circuit is primarily used for **temporary data storage**. A D latch circuit is also a single bit **Register**.

5.5.4 Timing Diagram for a D Latch

Waveform 5.7 shows a timing diagram or waveform diagram for a D latch.



WAVEFORM 5.7 Timing diagram for D latch

Device manufacturers generally specify a **setup time**, t_{su}, and **hold-time**, t_h, requirement for parts that contain a control input. To reduce the possibility of circuit malfunction, the input value *D* must be constant during the entire **sampling interval**, t_{si}. If the C input and the D input are changed nearly simultaneously, a bistable device may generate a **logic 0 glitch** as shown in Waveform 5.7 or a logic 1 glitch, which is not shown. A glitch is simply an undesired momentary pulse that occurs at the output of a circuit as shown in Waveform 5.7. Another possibility is the circuit may fail to store the last value of *D* or go into a temporarily unstable state, called a **metastable state**, and stay in the metastable state for a short but undetermined time. A metastable state is not a 1 or a 0 but a state in-between, as shown in Waveform 5.7. The setup time is the time that the *D* input must be held stable—that is, unchanging or a constant value—prior to the *C* input going to 0 to store the data value at the *D* input. The hold time is the time that the *D* input to the D latch must be held stable after the *C* input goes to 0.

At trigger events 1 and 3 in Waveform 5.7, there is no setup or hold-time violation, and output *Q* stores 1 at event 1 and stores 0 at event 3, which is the last value of *D* when *C* goes to 0. At trigger event 2, there is a setup time violation, which causes output *Q* to fail to store the last value of *D* momentarily, which results in a logic 0 glitch. At timing event 4, there is another violation. This is a hold-time violation, which causes output *Q* to go into a metastable state for a brief period of time, then to a high (or 1 in this case). Although metastability is not a frequent event, it can cause the circuit to malfunction when it occurs. In the case shown at timing event 4, the D latch fails to store the last value of *D*.

Listing 5.4 shows a complete VHDL design for a D latch.

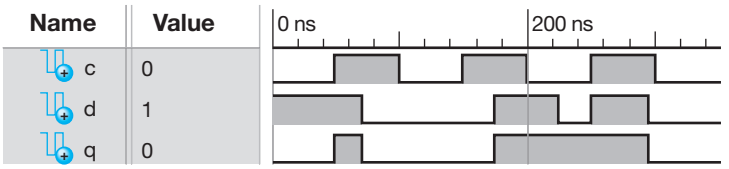
LISTING 5.4
Complete VHDL design for a D latch (project: D_Latch)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_Latch is port (
     C, D : in std_logic;
     Q : inout std_logic
     );
end D_Latch;

architecture dataflow of D_Latch is
begin
     Q <= '1' when ((not C and Q) or (C and D)) = '1' else
          '0';
--   Q <= (not C and Q) or (C and D); --alternate way
end dataflow;
```

Things you should notice about the VHDL design in Listing 5.4:

- The output signal *Q* is listed as mode **inout** in the entity because this allows *Q* to be read as an input and also an output.
- A dataflow design style is used with a conditional signal assignment (CSA). An alternate way to write the assignment statement for *Q* via a Boolean equation is shown by the comment.

Waveform 5.8 shows a correct simulation of design entity D_Latch.



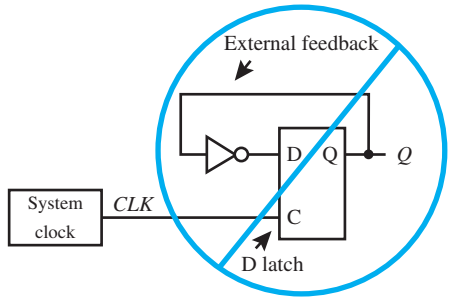
WAVEFORM 5.8 Correct simulation of design entity D_Latch

Observe that the waveform diagram in Waveform 5.8 follows the characteristic table for the D latch shown in Table 5.8. This proves that the VHDL code for the design is correct.

5.5.5 Creating a Clock via a D Latch

As we mentioned in the last section, a D latch is used for temporary data storage. That should be the only time a D latch is used. The last value of the *D* input is stored at the output when *C* is 0. If feedback is added externally to a D latch, the circuit may break into oscillations at the output when *C* is 1 or when the latch goes to its transparent or see-through mode. To prevent this possibility from happening, *never use external feedback around a D latch*, as shown in Figure 5.17.

FIGURE 5.17 Example of a bad circuit design



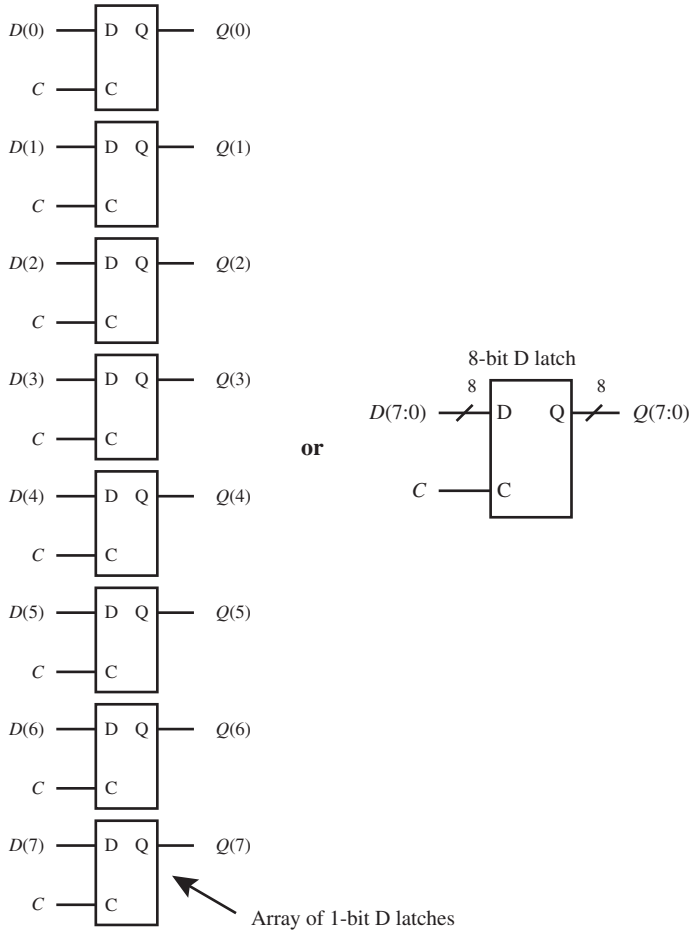
The circuit in Figure 5.17 is a bad circuit design because it will oscillate when *C* is 1. This circuit provides a reminder that you should never use external feedback around a D latch or connect the output of a D latch back to a gate or gates that feed into the D input of a D latch.

If you are trying to create a clock, you can connect the C input of the D latch in Figure 5.17 to V_{CC} and the circuit will oscillate. The frequency of this circuit is too high to be useful and also the frequency is not predictable or controllable.

5.5.6 Creating an 8-bit D Latch

Figure 5.18 shows an array of 1-bit D latches and a simplified symbol for the array—that is, an 8-bit D latch. An 8-bit D latch is also an 8-bit register. Latches are only used to store temporary data and should never be used with external feedback around them.

FIGURE 5.18 An array of 1-bit D latches and a simplified logic symbol for the array, an 8-bit D latch



5.6 DESIGNING D FLIP-FLOP CIRCUITS

A flip-flop is specifically designed so that its output will not break into oscillation when feedback is added externally around it because it is **edge triggered**, as opposed to the D latch, which is level sensitive. Edge triggering is required to allow feedback to be added externally. As we will see later, D flip-flops are used to design counters. Counters have external feedback and are found in practically every digital device produced today.

5.6.1 Designing Master–Slave D Flip-Flop Circuits

Figure 5.19a shows a positive edge-triggered D flip-flop (DFF) with a *CLR* (CLEAR) input implemented with two D latches and a NOT gate. This type of circuit is called a **master–slave D flip-flop circuit**. The master D latch and the slave D latch in the circuit each have an additional input called *CLR*, which allows the user to clear the D latches and the resulting D flip-flop. Figure 5.19b shows the logic symbol for a positive edge-triggered D flip-flop with a *CLR* input.

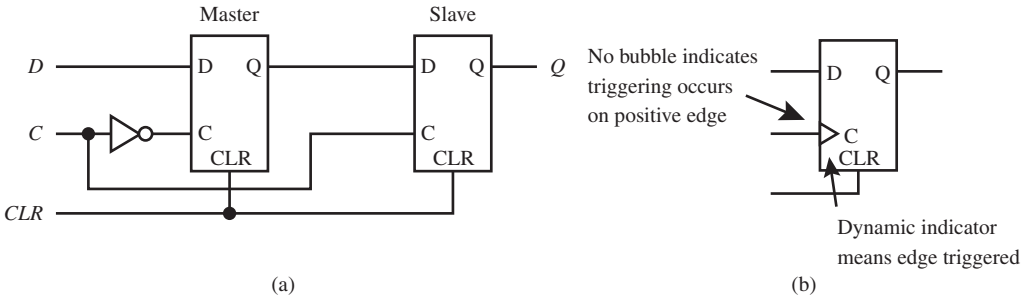


FIGURE 5.19 Positive edge-triggered D flip-flop with a *CLR* input: (a) implemented with two D latches and a NOT gate; (b) logic symbol

There is no transparent mode or see-through mode from the *D* input to the *Q* output for a D flip-flop due to edge triggering. When the master D latch is in its transparent mode, the slave D latch is in its storage mode, and when the master D latch is in its storage mode, the slave D latch is in its transparent mode. External feedback can thus be used around a D flip-flop, and the circuit will not break into oscillation.

The master D latch must be logic hazard free so that the circuit cannot produce a logic glitch that will be passed on to the slave D latch and result in a circuit failure. When the 1s (or 0s) are grouped in a K-map and the groups are not linked—that is, groups are not connected—a logic hazard may occur and produce a logic glitch. A logic 0 glitch may occur when groups of 1s are not linked, while a logic 1 glitch may occur when groups of 0s are not linked. A glitch is simply an undesired momentary pulse that occurs at the output of the circuit. The K-map in Figure 5.13 is repeated in Figure 5.20 to help with this discussion.

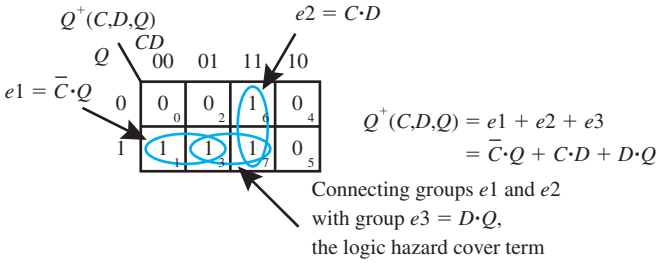


FIGURE 5.20 Manual K-map reduction for the next state output Q^+ for a D latch

The logic hazard cover term *e*3 that links the groups e1 and e2 shown in Figure 5.20 provides a logic hazard-free function for Q^+ so that the circuit cannot produce a logic glitch. The reduced function for Q^+ has a logic hazard that may result in a logic 0 glitch. The function $Q^+ = \overline{C}{\cdot}Q + C{\cdot}D + D{\cdot}Q$ with the logic hazard cover term included—that is, $D{\cdot}Q$—is a **logic hazard-free function** that effectively removes the logic glitch.

Figure 5.21a shows the final circuit for the D latch with a *CLR* input implemented with the logic hazard free function in AND/OR circuit form. Figure 5.21b shows the logic symbol for a D latch with a *CLR* input.

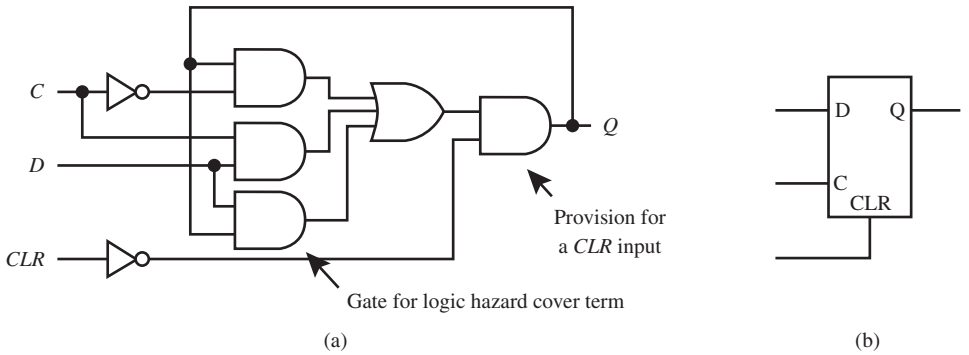


FIGURE 5.21 D latch with a *CLR* input: (a) implemented with a logic hazard-free function in AND/OR circuit form; (b) logic symbol

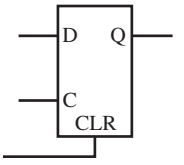
The circuit shown in Figure 5.21a can be used for the master D latch in the D flip-flop circuit with a *CLR* input shown in Figure 5.19a. The circuit shown in Figure 5.21a can also be used for the slave D latch with or without the gate for the logic hazard cover term.

The characteristic table for a D latch with a *CLR* input is shown in Table 5.10.

TABLE 5.10 Characteristic table for a D latch with a *CLR* input

CLR	*C*	*D*	Q^+	Comment (level sensitive)
1	x	x	0	Asynchronous *CLR*
0	0	0	*Q*	No change
0	0	1	*Q*	No change
0	1	0	0	Q^+ follows *D*
0	1	1	1	Q^+ follows *D*

for



An S-R NOR latch and an S-R NAND latch are logic hazard-free circuits, because the groups of 1s (or 0s) for these functions are linked, as you can observe by drawing K-maps for the functions. Therefore, these circuits do not create a glitch due to a logic hazard. Each of the D latch circuits shown in Figure 5.22 can also be used for the master or the slave D latch in the D flip-flop circuit with a *CLR* input shown in Figure 5.19a.

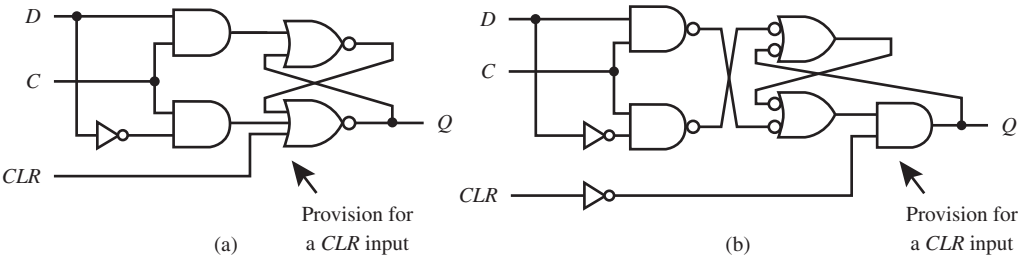


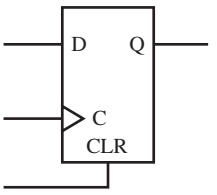
FIGURE 5.22 D latch with a *CLR* input: (a) implemented with an S-R NOR latch; (b) implemented with an S-R NAND latch

The characteristic table for a positive edge-triggered D flip-flop with a *CLR* input is shown in Table 5.11.

TABLE 5.11 Characteristic table for a positive edge-triggered D flip-flop with a *CLR* input

CLR	*C*	*D*	Q^+	Comment (positive edge-triggered)
1	x	x	0	Asynchronous *CLR*
0	0	x	*Q*	No change
0	1	x	*Q*	No change
0	↑	0	0	Q^+ stores last value of *D*
0	↑	1	1	Q^+ stores last value of *D*

for



Notice in the characteristic table that Q^+ stores the last value of D *only* when C is ↑. The **arrow up symbol** in the characteristic table means that the C input goes from a 0 to a 1—that is, in the positive direction from low to high; hence, this flip-flop type is a positive (or rising) edge-triggered device. Observe that the asynchronous clear input overrides the control input C and immediately clears the flip-flop. The asynchronous clear input allows the D flip-flop circuit to function normally only when *CLR* is 0. When the asynchronous clear input *CLR* is active or 1, it forces the Q output of the D flip-flop circuit to go to 0 immediately.

To convert a D flip-flop (or a D latch) circuit with an active high *CLR* input to an active low *CLR* input, place a NOT gate on the *CLR* input or remove a NOT gate if one is currently there. Keep in mind that the addition or deletion of the NOT gate will modify the characteristic table and the logic symbol. The values in the characteristic table for *CLR* must be complemented, and a bubble must be added to the *CLR* input of the logic symbol to indicate an active low input.

Figure 5.23a shows a positive edge-triggered D flip-flop (DFF) with a *PRE* (*PRESET*) input implemented with two D latches and a NOT gate. In this **master–slave D flip-flop circuit**, the master D latch and the slave D latch each have an additional input called *PRE*, which allows the user to set the D latches and the resulting D flip-flop. Figure 5.23b shows the logic symbol for a positive edge-triggered D flip-flop with a *PRE* input.

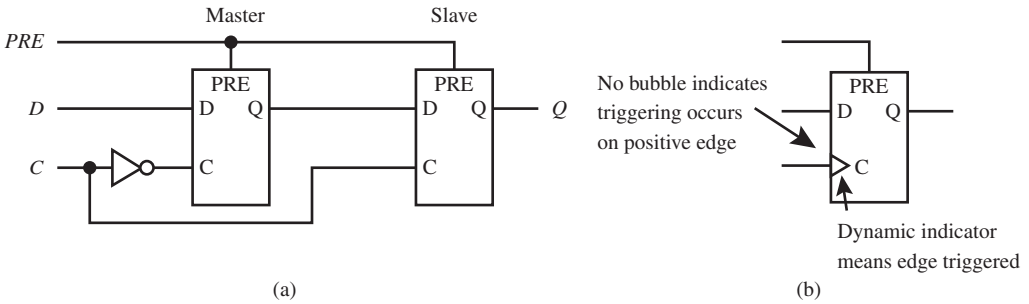


FIGURE 5.23 Positive edge-triggered D flip-flop with a *PRE* input: (a) implemented with two D latches and a NOT gate; (b) logic symbol

Figure 5.24 shows several logic hazard-free circuits for a D latch circuit with a *PRE* input.

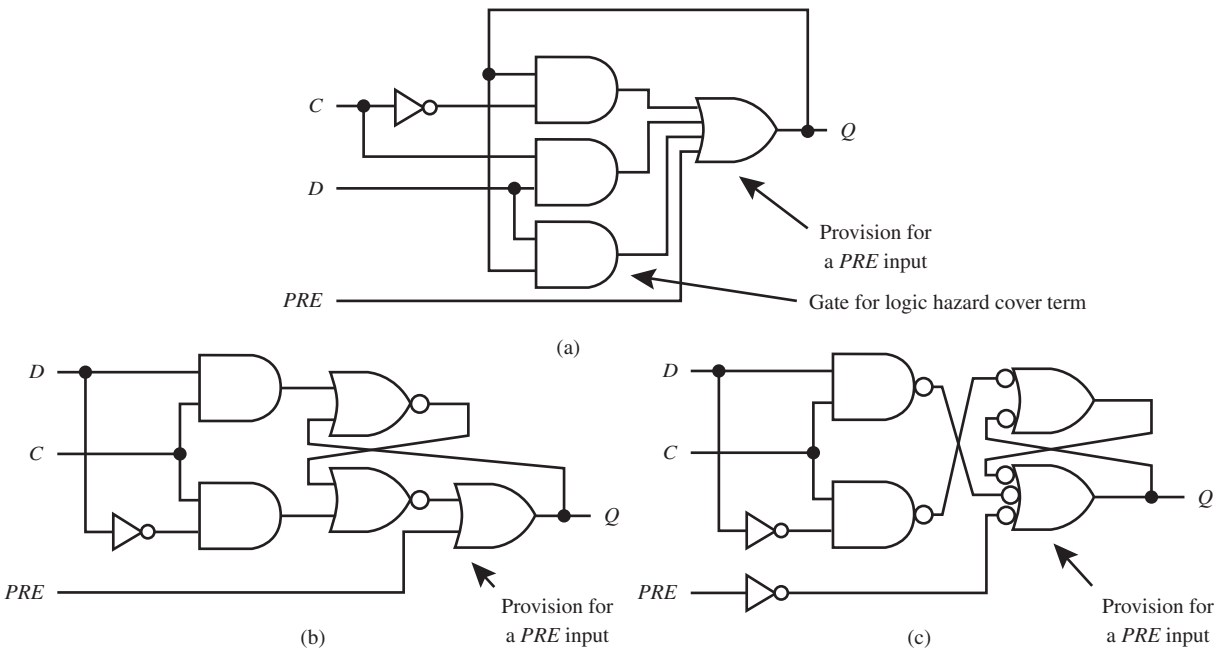


FIGURE 5.24 D latch implemented with a *PRE* input: (a) implemented with a logic hazard-free function in AND/OR circuit form; (b) implemented with an S-R NOR latch; (c) implemented with an S-R NAND latch

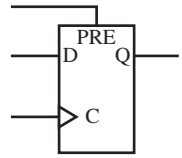
Each of the circuits in Figure 5.24 may be used for the master or the slave D latch in the D flip-flop circuit with a *PRE* input in Figure 5.23a.

The characteristic table for a positive edge-triggered D flip-flop with a *PRE* input is shown in Table 5.12.

TABLE 5.12 Characteristic table for a positive edge-triggered D flip-flop with a *PRE* input

PRE	*C*	*D*	Q^+	Comment (positive edge-triggered)
1	x	x	1	Asynchronous *PRE*
0	0	x	*Q*	No change
0	1	x	*Q*	No change
0	↑	0	0	Q^+ stores last value of *D*
0	↑	1	1	Q^+ stores last value of *D*

for



Observe that the asynchronous preset input overrides the control input *C*. The asynchronous preset input allows the D flip-flop circuit to function normally only when *PRE* is 0. When the asynchronous clear input *PRE* is 1, it forces the *Q* output of the D flip-flop circuit to go to 0 immediately.

To convert a D flip-flop (or a D latch) circuit with an active high *PRE* input to an active low *PRE* input, place a NOT gate on the *PRE* input or remove a NOT gate if one is currently there. Keep in mind that the addition or deletion of the NOT gate will modify the characteristic table and the logic symbol. The values in the characteristic table for *PRE* must be complemented, and a bubble must be added to the PRE input of the logic symbol to indicate an active low input.

5.6.2 Designing D Flip-Flop Circuits with S-R NAND Latches

An alternate way to implement a positive edge-triggered D flip-flop (DFF) with a *CLR* input is with three S-R NAND latches and a NOT gate as shown in Figure 5.25a. Figure 5.25b shows the logic symbol for a positive edge-triggered D flip-flop with a *CLR* input.

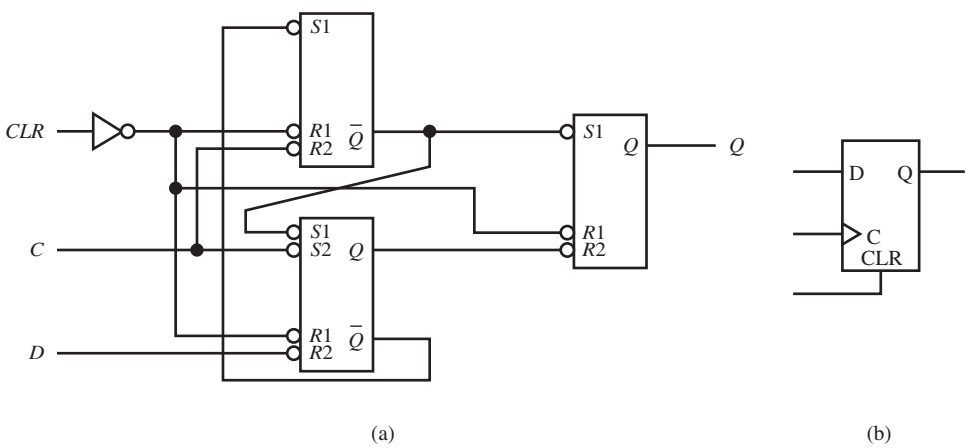


FIGURE 5.25 Positive edge-triggered D flip-flop circuit with a *CLR* input: (a) implemented with three S-R NAND latches and a NOT gate; (b) logic symbol

Figure 5.26 shows an **annotated gate-level circuit** for the positive edge-triggered D flip-flop with a *CLR* input implemented with a three S-R NAND latches and a NOT gate. The internal signals E through J can be used to write the VHDL code for the circuit as shown in the next section. This D flip-flop implementation is preferred over a master–slave D flip-flop with a *CLR* input because it requires fewer gates.

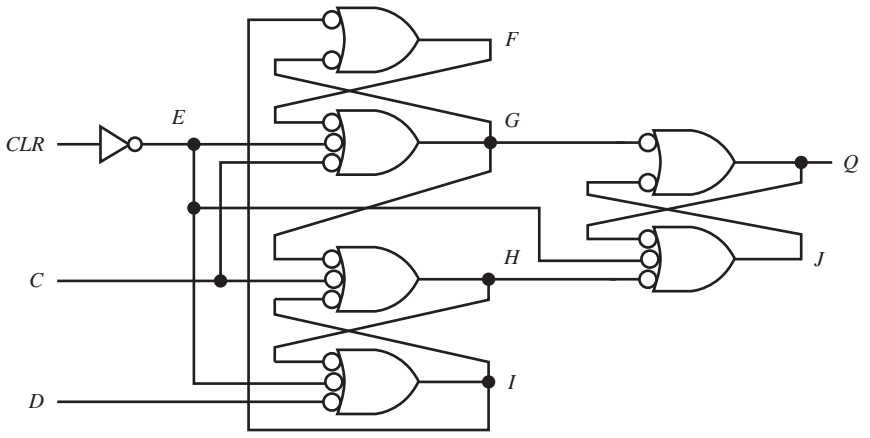


FIGURE 5.26 Annotated gate-level circuit for a positive edge-triggered D flip-flop with a *CLR* input implemented with three S-R NAND latches and a NOT gate

The characteristic table for the D flip-flop circuit in Figure 5.26 is shown in Table 5.11.

Figure 5.27 shows a **gate-level circuit** for a positive edge-triggered D flip-flop (DFF) with a *PRE* input implemented with three S-R NAND latches and a NOT gate. This D flip-flop implementation is preferred over a master–slave D flip-flop with a *PRE* input because it requires fewer gates.

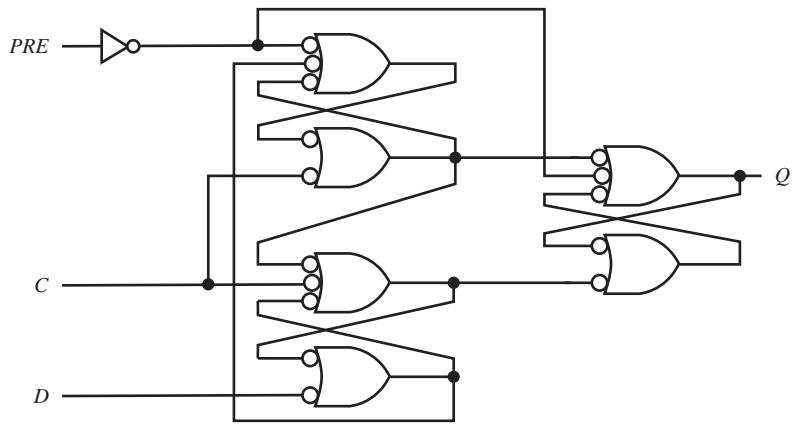


FIGURE 5.27 Gate-level circuit for a positive edge-triggered D flip-flop with a *PRE* input implemented with three S-R NAND latches and a NOT gate

The characteristic table for the D flip-flop circuit in Figure 5.27 is shown in Table 5.12.

Figure 5.28a shows the logic symbol for a negative edge-triggered D flip-flop with a *CLR* input, and Figure 5.28b shows the logic symbol for a negative edge-triggered D flip-flop with a *PRE* input.

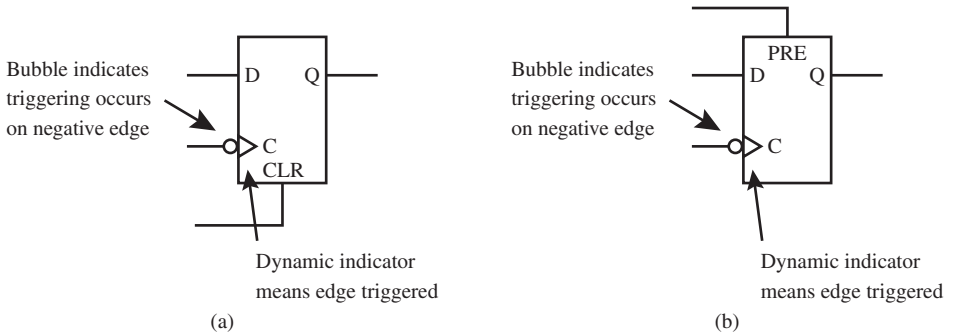


FIGURE 5.28 Logic symbol for negative edge-triggered D flip-flop: (a) *CLR* input; (b) *PRE* input

The characteristic tables for the negative edge-triggered D flip-flops in Figure 5.28 are shown in Table 5.13.

TABLE 5.13 Characteristic table for negative edge-triggered D flip-flop: (a) *CLR* input; (b) *PRE* input

CLR	*C*	*D*	Q^+	Comment (negative edge-triggered)
1	x	x	0	Asynchronous *CLR*
0	0	x	*Q*	No change
0	1	x	*Q*	No change
0	↓	0	0	Q^+ stores last value of *D*
0	↓	1	1	Q^+ stores last value of *D*

(a)

PRE	*C*	*D*	Q^+	Comment (negative edge-triggered)
1	x	x	1	Asynchronous *PRE*
0	0	x	*Q*	No change
0	1	x	*Q*	No change
0	↓	0	0	Q^+ stores last value of *D*
0	↓	1	1	Q^+ stores last value of *D*

(b)

Notice in the characteristic table that Q^+ stores the last value of *D only* when *C* is ↓. The **arrow down symbol** in the characteristic table means that the *C* input goes from a 1 to a 0—that is, in the negative direction from high to low; hence, this flip-flop type is a negative (or falling) edge-triggered device.

As you can observe from the characteristic tables, a positive edge-triggered D flip-flop and a negative edge-triggered D flip-flop acts the same except for the control input. A positive edge-triggered D flip-flop stores the value of *D* on the rising edge of the control input signal, while a negative edge-triggered D flip-flop stores the value of *D* on the falling edge of the control input signal.

To convert a positive edge-triggered D flip-flop circuit to a negative edge-triggered D flip-flop, place a NOT gate on the control input or remove a NOT gate if one is currently there. Keep in mind that the addition or deletion of the NOT gate will modify the characteristic table and the logic symbol. The values in the characteristic table for C input must be complemented, and a bubble must be added to the control input of the logic symbol to indicate a negative edge-triggered input.

The control input *C* is often called the clock input because the signal supplied to the control input is usually the system clock or a slower running clock derived from the system clock. The logic symbols for the two most commonly used D flip-flops in this text are shown in Figure 5.29 with signals applied at their inputs. These single bit D flip-flops are also single-bit registers.

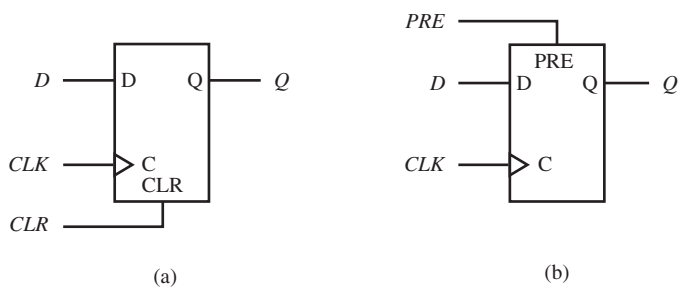
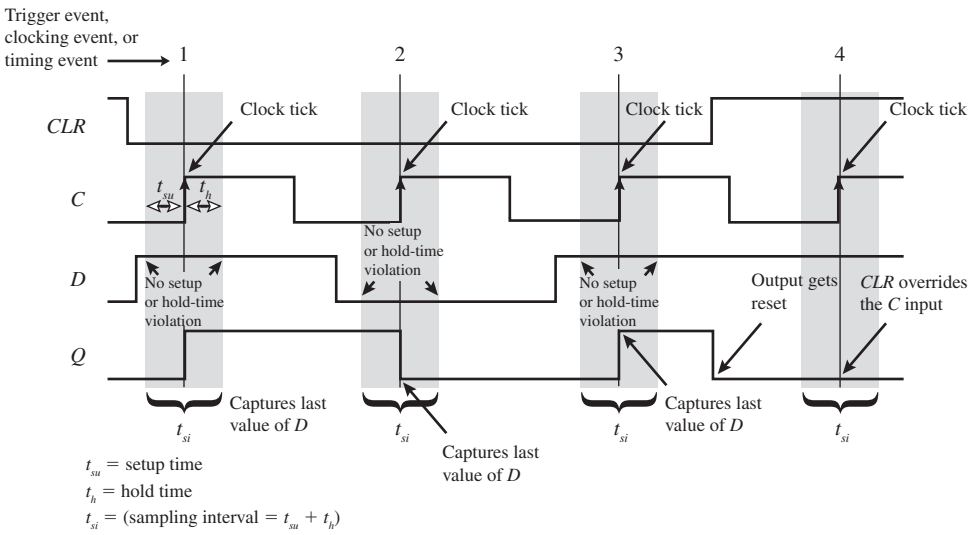


FIGURE 5.29 Two most commonly used D flip-flops in this text: (a) positive edge-triggered D flip-flop with a *CLR* input; (b) positive edge-triggered D flip-flop with a *PRE* input

The *D* input to a D flip-flop is called the **excitation input** because it excites the output and thus allows the output to change to a new value. The *D* input is also referred to as a **synchronous input**, because it must wait for the next **clock tick** to cause the output to change. We use the term *clock tick* to mean a rising-edge timing event for a positive edge-triggered device or falling-edge timing event for a negative edge-triggered device.

5.6.3 Timing Diagram for Positive Edge-Triggered D Flip-Flop

Waveform 5.9 shows a timing diagram for a positive edge-triggered D flip-flop.



WAVEFORM 5.9 Timing diagram for a positive edge-triggered D flip-flop

All devices with a controlled input have a specified setup time, t_{su}, and hold time, t_h, requirement, including flip-flops. To reduce the possibility of circuit malfunction, the input value D must be constant during the entire sampling interval, t_{si}. If input C and input D are changed nearly simultaneously, a bistable device may either fail to store the correct value or go into a temporarily unstable state, called a metastable state, and stay in the metastable state for an undetermined time. The timing diagram in Waveform 5.9 does not show examples of setup time or hold-time violations, because they are an exception, not the norm. An example of a metastable state is shown in Waveform 5.7 in the timing diagram for a D latch. Observe in Waveform 5.9 that the Q output stores the correct value of D only at the rising edge of C, assuming that D is constant during the entire sampling interval, t_{si}. *CLR* overrides C and resets Q at any time.

A timing diagram for a negative edge-triggered D flip-flop operates in exactly the same manner as a positive edge-triggered D flip-flop, only the Q output stores the correct value of D at the falling edge of C, assuming that D is constant during the entire sampling interval, t_{si}.

The annotated gate-level circuit for the positive edge-triggered flip-flop in Figure 5.26 is repeated in Figure 5.30 as a handy reference for writing the VHDL code for the flip-flop.

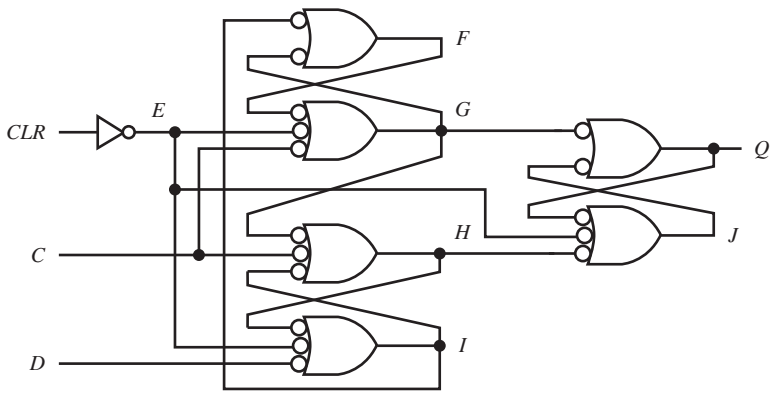


FIGURE 5.30 Annotated gate-level circuit for a positive edge-triggered D flip-flop with a *CLR* input implemented with three S-R NAND latches and a NOT gate

Listing 5.5 shows a complete VHDL design for the positive edge-triggered D flip-flop with a CLR input shown in Figure 5.30.

LISTING 5.5
Complete VHDL design for a positive edge-triggered D flip-flop with a CLR input (project: DFF_W_CLR_Bool)

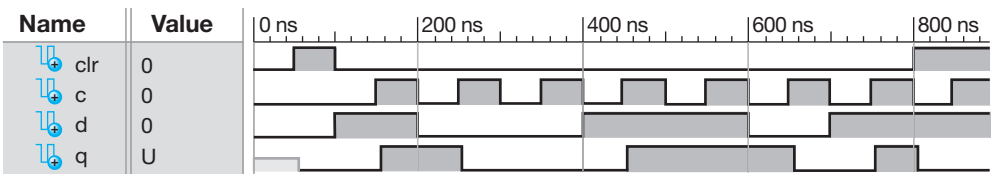
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DFF_W_CLR is port (
     CLR,C,D : in std_logic;
     Q : inout std_logic
     );
end DFF_W_CLR;

architecture dataflow of DFF_W_CLR is
     signal E,F,G,H,I,J : std_logic;
begin
     E <= not CLR;
     F <= I nand G;
     G <= not (F and E and C);
     H <= not (G and C and I);
     I <= not (H and E and D);
     J <= not (Q and E and H);
     Q <= G nand J;
end dataflow;
```

Things you should notice about the VHDL design in Listing 5.5:

- The output signal *Q* is listed as mode **inout** in the entity because this allows *Q* to be read as an input and also an output.
- The internal signals E, F, G, H, I, and J must be declared between **architecture** and the first **begin**. Remember that internal signals are not assigned a mode—that is, in, out, or inout.
- A dataflow design style is used with Boolean equations.
- It is simpler to use NAND (or NOR) gate assignments when working with gate-level designs for latches and flip-flops because this takes less typing, but you may also use DeMorgan equivalent gates if you prefer.

Waveform 5.10 shows a correct simulation of design entity DFF_W_CLR.

WAVEFORM 5.10
Correct simulation of design entity DFF_W_CLR



A clock tick is a rising edge timing event at the *C* input for this positive edge-triggered D flip-flop. To emphasize adequate setup and hold time for the D flip-flop, the *D* input is changed on the falling edge of the clock. Observe that the waveform diagram in Waveform 5.10 follows the characteristic table for the positive edge-triggered D flip-flop with a *CLR* input shown in Table 5.11. This proves that the VHDL code for the design is correct.

PROBLEMS

Section 5.2 Analyzing an S-R NOR Latch

5.1 What is the simplest form of a single-bit register?
5.2 Describe two ways to form an S-R latch with gates.
5.3 What do S and R stand for when discussing S-R latches?
5.4 For a simple light switch (mechanical latch), what does *set* mean?
5.5 For a simple light switch (mechanical latch), what does *reset* mean?

5.6 Write the characteristic table for a light switch.
5.7 What digital circuit has a similar characteristic table to a light switch?
5.8 Does a combinational logic circuit have a memory?
5.9 What does a sequential logic circuit have that a combinational logic circuit does not?
5.10 Draw and label the circuit for an S-R NOR latch.
5.11 Draw the logic symbol for an S-R NOR latch.
5.12 Write the signal name used for the present-state output of a latch.
5.13 Write the signal name used for the next-state output of a latch.
5.14 Write an expression for the present-state output for a latch in terms of the next-state output of the latch. Explain what the expression means.
5.15 Is an S-R NOR latch set dominant or reset dominant? What do the terms mean?
5.16 Write the characteristic table for an S-R NOR latch.
5.17 Draw the circuit delay model for an S-R NOR latch, and write the characteristic equation for the latch by analyzing the circuit.
5.18 Expand the characteristic table for an S-R NOR latch to obtain the PS/NS table.
5.19 Complete the timing diagram in Figure P5.19 by drawing the waveform for the Q output for an S-R NOR latch.

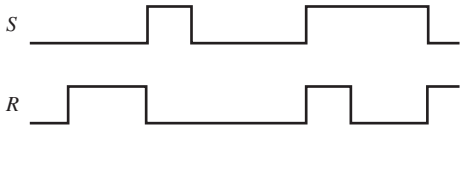


FIGURE P5.19

5.20 Write complete VHDL code for the S-R NOR latch shown in Figure P5.20. Use two Boolean equations—that is, one Boolean equation for signal E and another Boolean equation for output Q.

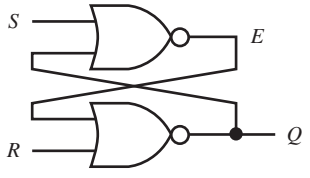


FIGURE P5.20

5.21 Obtain a simulation for the S-R NOR latch in problem 5.20. The simulation must agree with the characteristic table for the S-R NOR latch shown in Table 5.2 in the text to be correct.
5.22 Write complete VHDL code for the latch circuit shown in Figure P5.22. Use three Boolean equations—that is, one Boolean equation for signal E, one Boolean equation for signal F, and one Boolean equation for output Q.

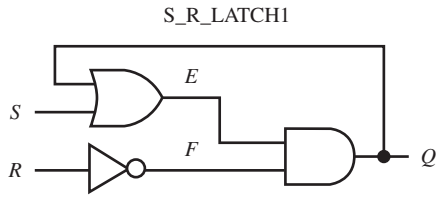


FIGURE P5.22

5.23 Obtain a simulation for the S-R NOR latch in problem 5.22. The simulation must agree with the characteristic table for the S-R NOR latch shown in Table 5.2 in the text to be correct.

Section 5.3 Analyzing an S-R NAND Latch

5.24 Draw and label the circuit for an S-R NAND latch.
5.25 Draw the logic symbol for an S-R NAND latch.
5.26 Is an S-R NAND latch set dominant or reset dominant? What do the terms mean?
5.27 Write the characteristic table for an S-R NAND latch.
5.28 Draw the circuit delay model for an S-R NAND latch, and write the characteristic equation for the latch by analyzing the circuit.
5.29 Expand the characteristic table for an S-R NAND latch to obtain the PS/NS table.
5.30 Complete the timing diagram in Figure P5.30 by drawing the waveform for the Q output for an S-R NAND latch.

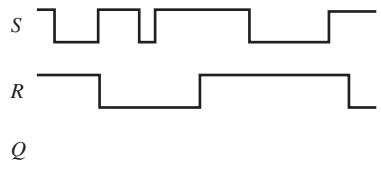


FIGURE P5.30

5.31 Write complete VHDL code for the S-R NAND latch shown in Figure P5.31. Use two Boolean equations—that is, one Boolean equation for signal E and another Boolean equation for output Q.

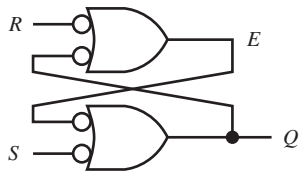


FIGURE P5.31

5.32 Obtain a simulation for the S-R NAND latch in problem 5.31. The simulation must agree with the characteristic table for the S-R NAND latch shown in Table 5.4 in the text to be correct.

5.33 Write complete VHDL code for the latch circuit shown in Figure P5.33. Use three Boolean equations—that is, one Boolean equation for signal *E*, one Boolean equation for signal *F*, and one Boolean equation for output *Q*.

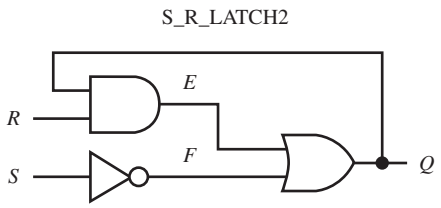


FIGURE P5.33

5.34 Obtain a simulation for the S-R NAND latch in problem 5.33. The simulation must agree with the characteristic table for the S-R NAND latch shown in Table 5.4 in the text to be correct.

Section 5.4 Designing a Simple Clock

5.35 Draw and label a state diagram for a simple clock.
5.36 List the four major parts of a state diagram.
5.37 What is another name that is used for a clock?
5.38 Write the PS/NS table for a simple clock.
5.39 Calculate the frequency of a simple clock. The NOT gate for the clock circuit has a propagation delay time, t_p, of 10 ns. Write an expression for the frequency of the clock, and then solve for the frequency.
5.40 Write an expression for the definition of the term *duty cycle* (DC). If the on time is 10 ns and the period of the Waveform is 30 ns, calculate the duty cycle as a percentage.
5.41 What is the most accurate clock to use in a digital circuit?
5.42 For the VHDL design for the design entity CLOCK in the text, what is the mode of the CLK? What is the alternate name that can be used for the mode?

Section 5.5 Designing a D Latch

5.43 What is another name for a D latch?
5.44 Draw and label a circuit for a D latch that uses an S-R NOR latch.
5.45 Draw and label a circuit for a D latch that uses an S-R NAND latch.
5.46 Draw the logic symbol for a D latch.
5.47 Write the characteristic table for a D latch.
5.48 For a D Latch, what is the value of C for the storage mode?
5.49 For a D latch, what is the value of C for the transparent mode or see-through mode?
5.50 Expand the characteristic table for the D latch to obtain the PS/NS table.
5.51 Obtain a reduced equation in POS form for the next state output Q^+ for a D latch. Hint: Group the 0s in the K-map in Figure 5.13 in the text. Use the equation to draw and label the D latch circuit.
5.52 When is a D latch used in a digital circuit?
5.53 Why is a D latch referred to as level sensitive?
5.54 What is a metastable state, and how can a D latch arrive in a metastable state?
5.55 Write complete VHDL code for the D Latch circuit shown in Figure P5.55. Use five Boolean equations—that is, one Boolean equation for signal *E*, and one Boolean equation for each NAND gate outputs.

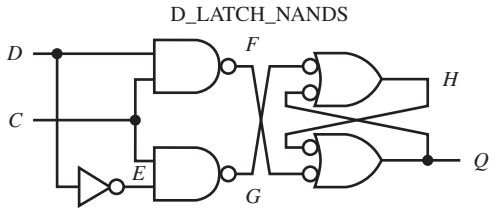


FIGURE P5.55

5.56 Obtain a simulation for the D latch in problem 5.55. The simulation must agree with the characteristic table for the D latch shown in Table 5.8 in the text to be correct.
5.57 What can happen if you provide external feedback around a D latch circuit?
5.58 Draw the circuit for a 2-bit register using an array of D latches.
5.59 Draw a simplified logic symbol for a 2-bit register using an array of D latches.

Section 5.6 Designing D Flip-Flop Circuits

5.60 Explain why external feedback can be added to a D flip-flop but not to a D latch circuit.
5.61 Draw a circuit for a master–slave positive edge-triggered D flip-flop with a *CLR* input using logic symbols for D latches and a NOT gate. Also draw a logic symbol for the circuit.
5.62 Write the characteristic table for a D latch with a *CLR* input and draw the logic symbol.
5.63 What is the purpose of a gate for a logic hazard cover term in the design of a D latch? Why is it important to include a gate for a logic hazard cover term in the master D latch for a master–slave edge-triggered D flip-flop design?
5.64 Can a D latch implemented with an S-R NOR latch or with an S-R NAND latch generate a glitch due to a logic hazard? Provide an explanation for your answer.
5.65 Write complete VHDL code for the D latch circuit with a *CLR* input shown in Figure P5.65. Use five Boolean equations—that is, one Boolean equation for signal *E*, one Boolean equation for each AND gate output, and one Boolean equation for each NOR gate output.

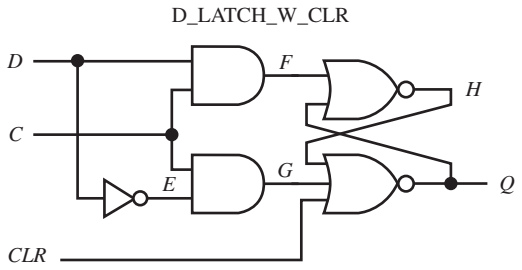


FIGURE P5.65

5.66 Obtain a simulation for the D latch in problem 5.65. The simulation must agree with the characteristic table for the D latch shown in Table 5.10 in the text to be correct.

5.67 Write the characteristic table for a positive edge-triggered D flip-flop with a *CLR* input and draw the logic symbol.

5.68 In Figure 5.19 in the text, explain what the *CLR* input does to the output *Q* of the master–slave D flip-flop when it is active or 1.

5.69 What does the symbol ↑ mean?

5.70 Does an asynchronous clear wait on the control input to a D flip-flop to clear the flip-flop, or does it override the control input and clear the flip-flop immediately?

5.71 Explain how a D flip-flop (or a D latch) with an active high *CLR* input is converted to a D flip-flop (or a D latch) with an active low *CLR* input.

5.72 Write the characteristic table for a positive edge-triggered D flip-flop with an active low *CLR* input, and draw the logic symbol.

5.73 Draw a circuit for a master–slave positive edge-triggered D flip-flop with a *PRE* input using logic symbols for D latches and a NOT gate. Also draw a logic symbol for the circuit.

5.74 In the D latch circuit in Figure P5.74, what is eliminated by the gate for the logic hazard cover term?

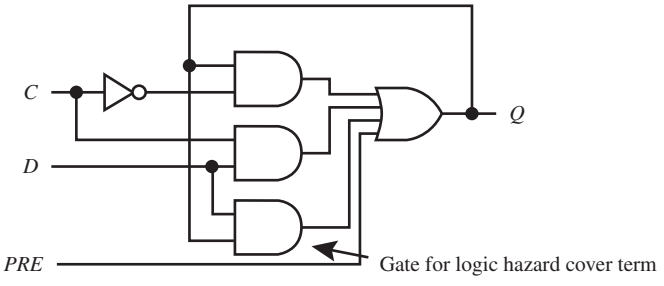


FIGURE P5.74

5.75 Write the characteristic table for a D latch with a *PRE* input, and draw the logic symbol.

5.76 Explain how a D flip-flop (or a D latch) with an active high *PRE* input, is converted to a D flip-flop (or a D latch) with an active low *PRE* input.

5.77 Write the characteristic table for a positive edge-triggered D flip-flop with an active low *PRE* input, and draw the logic symbol.

5.78 Draw the gate-level circuit for the S-R NAND latch shown in Figure P5.78, which is used in the D flip-flop with a *CLR* input implemented with three S-R NAND latches in Figure 5.25.

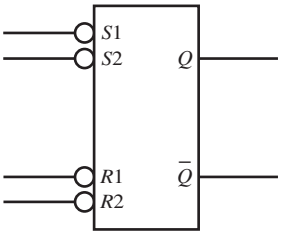


FIGURE P5.78

5.79 Draw a circuit for a positive edge-triggered D flip-flop with a *PRE* input, using logic symbols for S-R NAND latches and a NOT gate. Also draw a logic symbol for the circuit. Hint: Use the circuit in Figure 5.27 in the text as a guide.

5.80 Write the characteristic table for a negative edge-triggered D flip-flop with a *CLR* input, and draw the logic symbol.

5.81 Draw the logic symbol for a negative edge-triggered D flip-flop with a *PRE* input.

5.82 What does the symbol ↓ mean?

5.83 Explain how you can convert a positive edge-triggered flip-flop to a negative edge-triggered flip-flop.

5.84 Is the D flip-flop circuit in Figure P5.84 a positive edge-triggered flip-flop or a negative edge-triggered flip-flop? Discuss how you arrived at your answer.

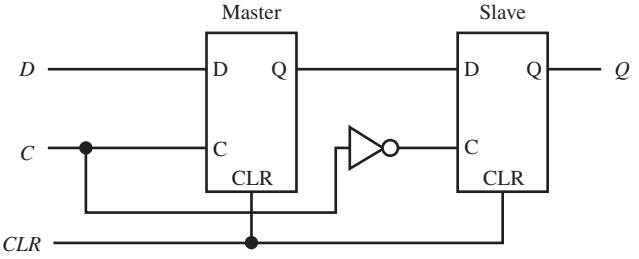


FIGURE P5.84

5.85 Is the D flip-flop in Figure P5.85 a positive edge-triggered flip-flop or a negative-edge triggered flip-flop? Discuss how you arrived at your answer.

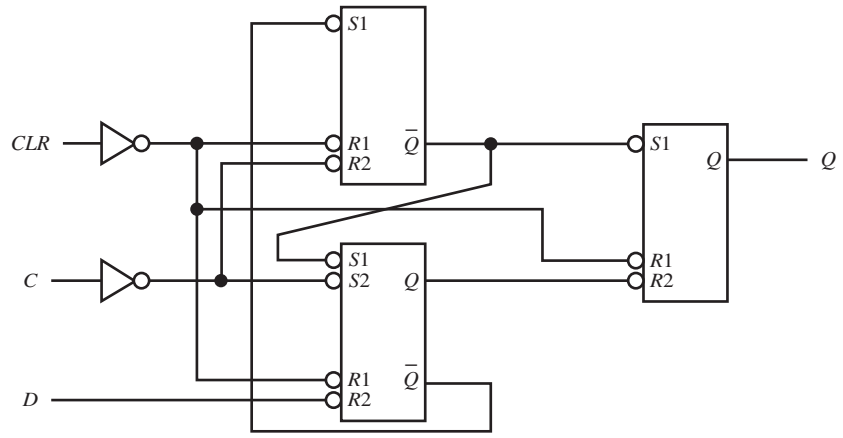


FIGURE P5.85

5.86 What is the excitation input to a D flip-flop?
5.87 Why is the *D* input to a D flip-flop referred to as a synchronous input?
5.88 What does the term *clock tick* mean?
5.89 What is the name used for a temporarily unstable state for a bistable device?
5.90 Complete the timing diagram in Figure P5.90 by drawing the waveform for the *Q* output for a positive edge-triggered D flip-flop.

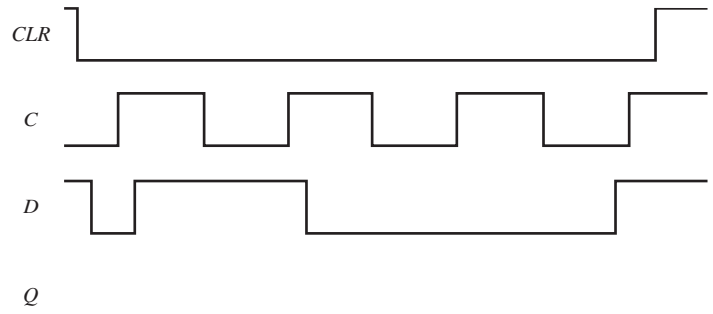


FIGURE P5.90

5.91 Complete the timing diagram in Figure P5.91 by drawing the Waveform for the *Q* output for a negative edge-triggered D flip-flop.

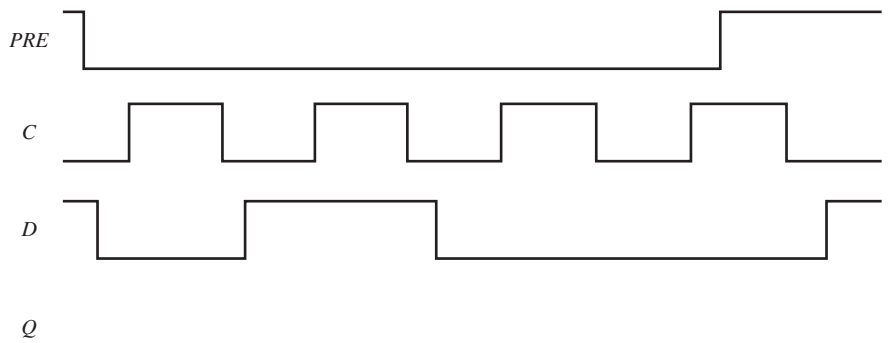


FIGURE P5.91

5.92 Write complete VHDL code for the D flip-flop circuit with a *CLR* input shown in Figure P5.92. Use the following Boolean equations:

$E = \overline{CLR}$

$H = (\overline{\overline{H} + \overline{E} + \overline{D}}) + \overline{C} + \overline{G}$

$G = (\overline{G} + (\overline{\overline{H} + \overline{E} + \overline{D}}) + \overline{E} + \overline{C}$

$Q = (\overline{G} + (\overline{Q} + \overline{E} + \overline{H})$

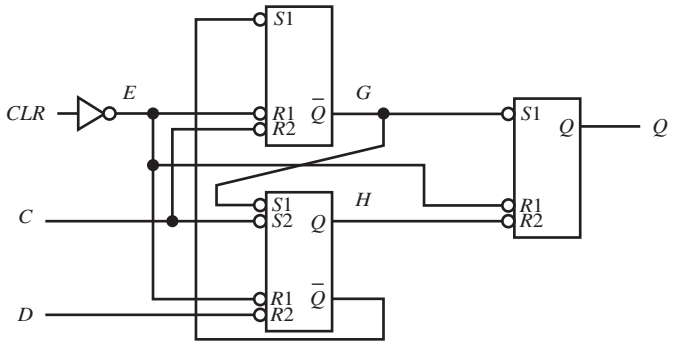


FIGURE P5.92

5.93 Obtain a simulation for the D flip-flop circuit in problem 5.92. The simulation must agree with the characteristic table for a positive edge-triggered D flip-flop with a *CLR* input shown in Table 5.11 in the text to be correct.

5.94 Write complete VHDL code for the master–slave D flip-flop circuit with a *CLR* input shown in Figure P5.94. Use the following Boolean equations:

$E = (E \cdot \overline{C} + C \cdot D + D \cdot E) \cdot \overline{CLR}$

$F = \overline{C}$

$Q = (Q \cdot \overline{F} + F \cdot E) \cdot \overline{CLR}$

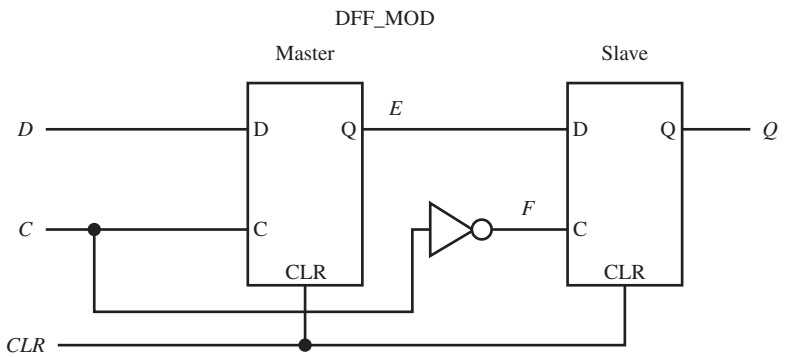


FIGURE P5.94

5.95 Obtain a simulation for the D flip-flop circuit in problem 5.94. The simulation must agree with the characteristic table for a negative edge-triggered D flip-flop with a *CLR* input shown in Table 5.13a in the text to be correct.

6 Chapter

Simple Finite State Machine Design with VHDL

Chapter Outline

6.1 INTRODUCTION

In this chapter, you will learn many ways to write VHDL to create digital hardware for synchronous sequential logic circuits called **counters**. These types of circuits have **feedback**—that is, the present-state flip-flop outputs are fed back as inputs to the circuits. In this chapter, we will only present **simple counters**.

6.2 SYNCHRONOUS CIRCUITS

Our emphasis in this chapter is placed on designing clock mode (or clocked) counter circuits. These circuits are **synchronous circuits** because their outputs change state in step with a particular input signal called the **clock**. A synchronous circuit stores a value (either a single bit or a string of bits) at the rising (or falling) edge of the clock and remembers that value until the next rising (or falling) edge of the next clock cycle. Latches are sometimes created due to our carelessness in writing VHDL code. We will concentrate on synchronous circuits that use data-type flip-flops (DFFs), because they are predominantly used in modern programmable devices—that is, CPLDs (complex programmable logic devices) and FPGAs (field programmable gate arrays). Other types of flip-flops such as the SR-type flip-flop, JK-type flip-flop, and T-type flip-flops exist but are no longer used for new design, because CPLDs and FPGAs only contain data-type flip-flops, which are also called D flip-flops or DFFs.

6.3 CREATING D-TYPE FLIP-FLOPS IN VHDL

A D-type flip-flop (DFF) is rather useless in many applications unless its output value can either be **cleared** (Q output goes to 0 when CLR input is 1) or **preset** (Q output goes to 1 when PRE input is 1) to a known state or value either at startup (or power on) or at any other desired time. Figure 6.1a shows a standard DFF with a CLR input while Figure 6.1b shows a standard DFF with a PRE input.

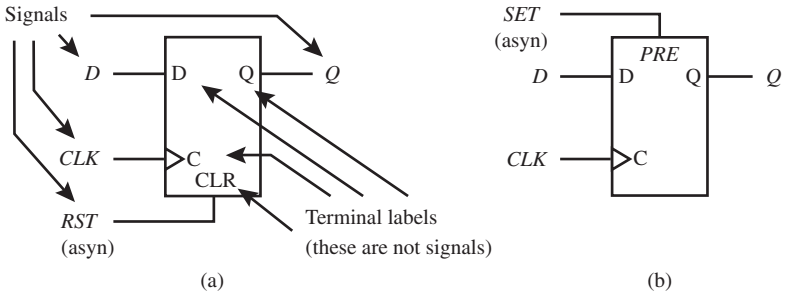


FIGURE 6.1 Logic symbol for a standard DFF (D-type flip-flop): (a) CLR is a clock-independent input for clearing the flip-flop; (b) PRE is a clock-independent input for setting the flip-flop

In Figure 6.1a when the CLR input is asserted or pulled high to logic 1, output Q is cleared to 0, independent of the control input C. As long as the CLR input is 1, Q remains at 0. When the CLR input is changed to 0, the DFF behaves in its normal fashion and loads the value of D at the next rising edge of the clock signal *CLK*, which is applied to the control input C.

In Figure 6.1b, when the PRE input is asserted or pulled high to logic 1, output Q is set to 1, independent of the control input C. As long as the PRE input is 1, Q remains at 1. When the PRE input is changed to 0, the DFF behaves in its normal fashion and loads the value of D at the next rising edge of clock signal *CLK*, which is applied to the control input C.

Signals are used to write VHDL code, not terminal labels. In Figures 1a and 1b, signal *D* and input D have the same name, and signal *Q* and output Q also have the same name. Signal *CLK* and input C are named differently. The signal *RST* and input CLR are named differently. The signal *SET* and input PRE are also named differently. The signal *CLK* is generally connected to an external clock and hence is named *CLK*, which is short for *CLOCK*. The signal *RST* is short for *RESET*, which is an alternate name for CLEAR. The signal *SET* is an alternate name for *PRESET*. We added (asyn) which is short for asynchronous to the signals *RST* and *SET* to indicate that the signals *RST* and *SET* are clock independent. An asynchronous signal is a signal that can happen at any time. When the signal *RST* (asyn) in Figure 6.1a is asserted, it overrides the clock signal *CLK* and immediately clears the D flip-flop. When the signal *SET* (asyn) in Figure 6.1b is asserted, it overrides the clock signal *CLK* and immediately sets the D flip-flop.

Listing 6.1 shows a complete VHDL design for the DFF in Figure 6.1a using a dataflow architecture declaration with a conditional signal assignment (CSA).

LISTING 6.1
Complete VHDL design for the DFF (project: DFF)

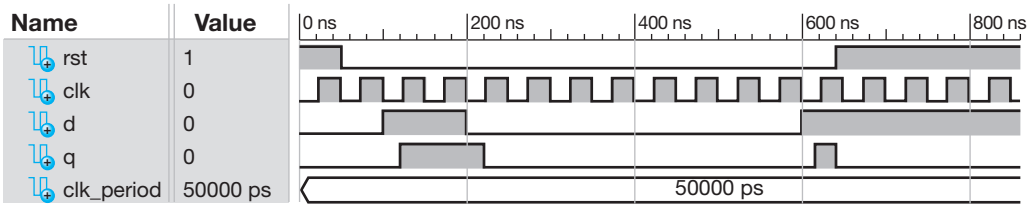
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity DFF is port (
     rst, clk, d: in std_logic;
     q: inout std_logic
     );
end DFF;
architecture dataflow of DFF is
begin
     q <= '0' when rst = '1' else
            d  when rising_edge(clk) else
            q;
                 --note: "else q" is inferred (so it can be removed)
end dataflow;
```

Things you should notice about the VHDL design in Listing 6.1:

- A library clause, a use clause, and an entity declaration are required to make the design complete.
- The output signal *Q* is listed as mode **inout** in the entity because this allows *Q* to be read as an input and also an output.
- Because the *RST* input precedes rising_edge(clk), the *RST* input is a clock-independent input.
- Rising_edge(*CLK*) represents the event caused by *CLK* changing from 0 to 1. Another way of representing the event is by using *CLK*' event **and** *CLK* = '1' in place of rising_edge(*CLK*). Note: falling_edge(C*LK)* represents the event caused by *CLK* changing from 1 to 0. In either case, a D flip-flop is created.
- As indicated in the comment, **else** *Q* is inferred. When **else** *Q* is left off of this conditional signal assignment, the present-state output value *Q* is simply retained or stored. Leaving out **else** *Q* is an advantage because this results in less typing. This can cause problems by creating latches by inference, when there is no clock tick event specified—that is, rising_edge(*CLK*) or falling_edge(C*LK)*. In general, when no clock tick event is specified, be sure to include the last **else** (the **else** after the last **when**) in a conditional signal assignment (CSA) and also include the last **else** (the **else** after the last **then**) in an if-then-else statement or if-then-elsif statement unless you really intend to create an inferred latch.
- Recommendation: Do not use rising_edge(*CLK*) embedded in a selected signal assignment (SSA) because this construct is not supported in VHDL.

Waveform 6.1 shows the simulation with the correct functionality of design entity DFF.

WAVEFORM 6.1
Simulation with the correct functionality of design entity DFF



To understand how the simulation of the design entity DFF was obtained, see Appendix B, Section B.3.

The VHDL design in Listing 6.1 can easily be modified to create a flip-flop with an active low *RST* input, an active low *SET* input, and also a clock tick event for a falling edge of *CLK*. Remember to place the *RST* input or the *SET* input prior to the clock tick event (before rising_edge(*CLK*) in the VHDL code) if you want the *RST* input or the *SET* input to be a clock-independent input and thus override the clock input.

To make a flip-flop with a clock-dependent or synchronous *RST* (*SET*) input, simply place the *RST* (*SET*) input after the clock tick event (following rising_edge(*CLK*) in the VHDL code). The *RST* (*SET*) input will be clock dependent and only be able to clear (set) the flip-flop after the rising_edge(*CLK*). If you use a process to create the flip-flop, do not include *RST* (*SET*) in the sensitivity list when the *RST* (*SET*) input is clock dependent. A DFF with a synchronous *RST* (*SET*) input may be labeled *RST* (syn) (*SET* (syn)) as a reminder that the input is synchronous.

6.4 DESIGNING SIMPLE SYNCHRONOUS CIRCUITS

Simple counters have a **fixed counting sequence** and therefore do not have external inputs to change the counting sequence. Complex counters have external inputs that allow the counting

sequence to be changed. We will cover **complex counters** or **complex state machines** later in Chapter 9.

Simple counters are often named by the sequence as well as the direction in which they count. The sequence can be a straight binary sequence, a Gray code sequence, or any other sequence in addition to counting forward (or up) or counting in reverse (or down). The sequence can be a one-hot sequence such that all bits are off except one that is on; hence, the term one-hot. A one-cold sequence can be used as well. Counters by definition are **state machines**, or **finite state machines**, because they contain only a finite number of states in their counting sequence. A simple counter is a **simple state machine** (SSM) or **simple finite state machine** (SFSM).

We will cover the following three methods for designing simple counters suitable for VHDL:

1. Algorithmic equation method.
2. Arithmetic method.
3. PS/NS (present-state/next-state) tabular method.

The first method requires a full understanding of D-type flip-flops as well as how to draw a logic circuit for a design. The last two methods require less hardware understanding, but a little more detailed understanding of VHDL. The following list is a brief description of the three methods we will cover:

- When we use the algorithmic equation method, we use predefined steps to obtain the flip-flop excitation equations and then draw the schematic. The schematic is then used to write proper VHDL code for the design.
- When we use the arithmetic method, we enter the counting sequence via an arithmetic "+" or "−" operator in VHDL.
- When we use the PS/NS tabular method, we enter the counting sequence via an if statement and a case statement in VHDL.

6.5 COUNTER DESIGN USING THE ALGORITHMIC EQUATION METHOD

The **algorithmic equation method** (AE method) is partially a manual method and partially a coding method because you must manually obtain the excitation equations for the flip-flops, manually draw the circuit, and then write the VHDL code for the circuit. This method requires that you have a complete understanding of the entire design process that must be performed to obtain a counter design.

Figure 6.2a shows a **counting sequence diagram** (or **state sequence diagram**) for a binary up counter (2 bits) with a clock independent reset input *RST* for design entity Counter1. An equivalent state diagram for the counter is shown in Figure 6.2b.

When the asynchronous input *RST* is asserted or equal to 1, the counter goes to state 00. When the asynchronous input *RST* is not asserted or is 0, the counter responds to the clock. Each time the clock ticks, the counter changes from its present-state value to its next-state value—that is, 00 to 01, then 01 to 10, then 10 to 11, then 11 back to 00—following the state-transition lines. The clock signal is not shown in the counting sequence diagram or in a state diagram; that is, it is implied.

Figure 6.2 shows an asynchronous reset signal *RST*, which resets the counter to 00 independent of the clock. From this information, we can draw the clouds-of-logic circuit for the binary up counter as shown in Figure 6.3. Each state represents the output of a D flip-flop, so two flip-flops are drawn with the outputs $Q0$ and $Q1$. D flip-flop $Q0$ has a D input signal labeled $D0$, and D flip-flop $Q1$ has a D input signal labeled $D1$. The D inputs provide the next-state values for the D flip-flops.

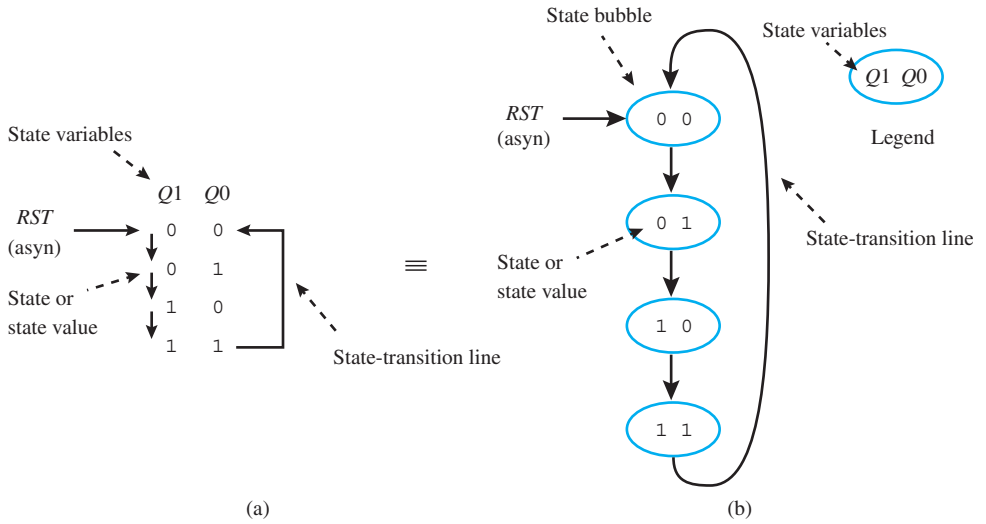


FIGURE 6.2 Binary up counter (2 bits) for design entity Counter1: (a) counting sequence diagram; (b) equivalent state diagram

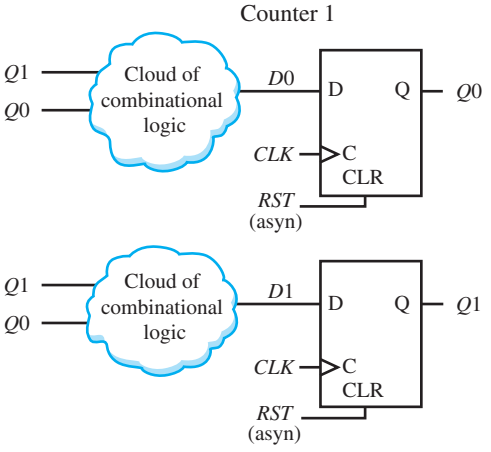


FIGURE 6.3 Binary up counter (2 bits) with clouds of combinational logic to be determined

Our task is to obtain the D excitation equations $D0$ and $D1$ for the flip-flops $Q0$ and $Q1$ so that we can fill in the clouds of combinational logic.

An algorithm, or step-by-step procedure, will be developed so we can write the excitation equations for the D inputs for each of the two flip-flops $Q0$ and $Q1$. After we obtain the D excitation equations, we can draw the schematic for the circuit and write the proper VHDL code for the design.

The excitation equation for a single D flip-flop depends on the output transitions made by the flip-flop from its present-state output Q(PS) to its next-state output Q(NS), or $Q(\text{PS}) \rightarrow Q(\text{NS})$. Table 6.1 summarizes all possible output transitions for a single D flip-flop. Notice in Table 6.1 that the D column is the equivalent to the next-state value, or $D = Q^+ = Q(NS)$.

TABLE 6.1 Flip-flop output transitions and required *D* input

Q(PS)	Q(NS)	Comment	D
0	0	Hold 0 transition	0
0	1	Set transition	1
1	0	Clear transition	0
1	1	Hold 1 transition	1

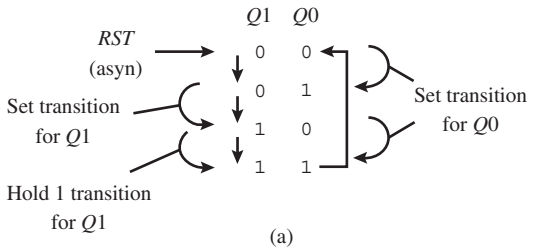
The equation for the function D can be written in terms of its 1s or its 0s. We chose to obtain the D excitation equation for the 1s of the function D. This requires using just two of the transitions in Table 6.1—that is, the Set transition (Q(PS) = 0) → (Q(NS) = 1) or simply the 0 → 1 transition, and the Hold 1 transition (Q(PS) = 1) → (Q(NS) = 1) or simply the 1 → 1 transition. Using these two transitions, we can write the equations for simple counters that use D flip-flops by inspection using just a counting sequence diagram, or a state diagram. For now we will just work with a counting sequence diagram.

Because only Set transitions and Hold 1 transitions are required to obtain each D input excitation equation, we refer to this method as the **Set OR Hold 1 method**. The algorithm for the method can be expressed by the following **Set OR Hold 1 equation**:

D = Σ(PS expression for a set transition)
+ Σ(PS expression for a hold 1 transition)

The summation symbol Σ represents a logical summation—that is, OR the PS expression together for all the Set transitions and all the Hold 1 transitions. The Set OR Hold 1 equation is applied in turn to obtain the D input excitation equations for each of the D flip-flops making up the counter. This method works for *any size design* and the equations do not have to be minimized because VHDL will reduce the equations. K-maps do not have to be drawn and minimized when using the Set OR Hold 1 method.

Figure 6.4a shows the procedure for using the Set OR Hold 1 method via a counting sequence diagram. Figure 6.4b shows the classical procedure for using a PS/NS (present-state/next-state) table to obtain the same D input excitation equations for the binary up counter, which we will refer to as the **conventional method**.



≡

RST (asyn)	PS Q1	PS Q0	NS Q1+	NS Q0+
1	x	x	0	0
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0

(b)

FIGURE 6.4 (a) Counting sequence diagram showing all set and hold 1 transitions for the binary up counter; (b) equivalent PS/NS table

To use the Set OR Hold 1 method, you first need to understand how to obtain the present-state expressions for each state. The PS expression for state 00 is $\overline{Q1}\cdot\overline{Q0}$, for state 01 is $\overline{Q1}\cdot Q0$, for state 10 is $Q1\cdot\overline{Q0}$, and for state 11 is $Q1\cdot Q0$. Notice that the PS expressions are simply the minterms written for the two state variables $Q1$ and $Q0$.

Always remember when using the Set OR Hold 1 equation that the D in the equation represents the next-state value, which is Q^+, or $Q(NS)$.

From the counting sequence diagram shown in Figure 6.4a, we can write the excitation equation for the $D1$ input by using the Set OR Hold 1 equation as follows:

$D1$ = *PS* expression for a 0 → 1 transition from state 1 to state 2
+ PS expression for a 1 → 1 transition from state 2 to state 3

which results in $D1 = \overline{Q1}\cdot Q0 + Q1\cdot\overline{Q0}$ (observe that $D1$ is an XOR function). Notice in Figure 6.4a that there is one Set transition and one Hold 1 transition for $Q1$.

From the counting sequence diagram shown in Figure 6.4a we can write the excitation equation for the $D0$ input by using the Set OR Hold 1 equation as follows:

$D0$ = PS expression for a 0 → 1 transition from state 0 to state 1
+ PS expression for a 0 → 1 transition from state 2 to state 3

which results in $D0 = \overline{Q1}\cdot\overline{Q0} + Q1\cdot\overline{Q0} = \overline{Q0}$.

To obtain the *D* input excitation equations using the conventional method, you must first remember that $D1 = Q1^+$ and $D0 = Q0^+$. Next you must obtain the next-state outputs equations for $Q1^+$ and $Q0^+$ from the PS/NS table and then assign *D*1 and *D*0 to those expressions.

From the equivalent PS/NS table shown in Figure 6.4b, we can write *D*1 as follows:
$D1 = Q1^+ = \overline{Q1}\cdot Q0 + Q1\cdot\overline{Q0}$ (observe that *D*1 is an XOR function).
From the equivalent PS/NS table shown in Figure 6.4b, we can write *D*0 as follows:
$D0 = Q0^+ = \overline{Q1}\cdot\overline{Q0} + Q1\cdot\overline{Q0} = \overline{Q0}$.

You must remember to only use the next-state values that result from clock ticks in Figure 6.4b; do not use the row in the table for any asynchronous input signals—that is, *RST*, *SET*, or *INIT*.

Observe that the resulting *D* equations obtained using the conventional method are the same as the *D* equations obtained using the Set OR Hold 1 method. Keep in mind that the conventional method works fine for small designs, but it does not work well for large designs—that is, designs with many states. K-maps are generally required when using the conventional method, which adds additional time and effort to obtain the *D* excitation equations. Minimized equations are not required, because VHDL will reduce the equations. The size required for the K-maps is a limiting factor when using the conventional method.

The excitation equations for the *D* inputs are used to fill in the clouds of combinational logic for the binary up counter as shown in the Figure 6.5.

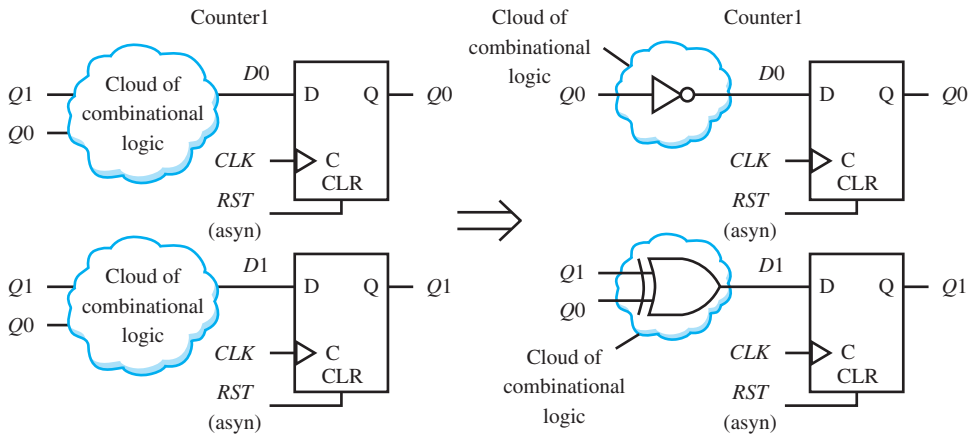


FIGURE 6.5 Schematic for a binary up counter (2 bits) with wire connections implied by signal name association

Wires with the same signal name form a **net**, which is a group of wires that are all connected together. Notice that there is a *Q*0 net, a *Q*1 net, a *CLK* net, and a *RST* net. The wiring for each net is implied by **signal name association**.

Figure 6.6 represents the same schematic as Figure 6.5 with the *Q*0 net, the *Q*1 net, the *CLK* net, and the *RST* net shown with the actual **physical wiring connections**. Sometimes a combination of both wiring connection schemes is used.

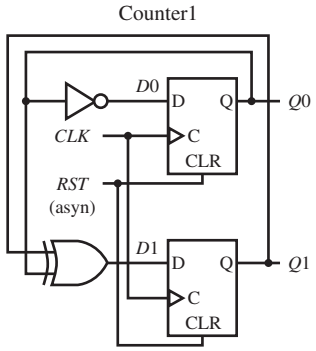


FIGURE 6.6 Equivalent schematic for a binary up counter (2 bits) with the signals shown with physical wiring connections

Note: Some graphic tools allow crossing wires that are not connected to be drawn as shown here:

The physical (or point-to-point) wiring connections shown in Figure 6.6 are usually harder to draw because crossing wires (no dots at their intersections) and connected wires (dots at their connection points) must be clearly indicated. The next step in the algorithmic equation method is to use the schematic to write proper VHDL.

Listing 6.2 shows a complete VHDL design for the binary up counter (2 bits) in Figure 6.5 or 6.6—that is, design entity Counter1.

LISTING 6.2
Complete VHDL design for design entity Counter1 using a mixed architecture declaration with dataflow and behavioral design styles (project: Counter1)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Counter1 is port (
   rst, clk : in std_logic;
   q1, q0 : inout std_logic
   );
end Counter1;

architecture mixed of Counter1 is
   signal d1, d0: std_logic;
begin
   d1 <= q1 xor q0;
   d0 <= not q0;
q1_output:  process (clk, rst)
            begin
               if rst = '1' then q1 <= '0';
               elsif rising_edge(clk) then q1 <= d1;
                                      --or q1 <= q1 xor q0
               end if;
            end process;

q0_output:  process (clk, rst)
            begin
               if rst = '1' then q0 <= '0';
               elsif rising_edge(clk) then q0 <= d0;
                                       --or q0 <= not q0
               end if;
            end process;
end mixed;
```

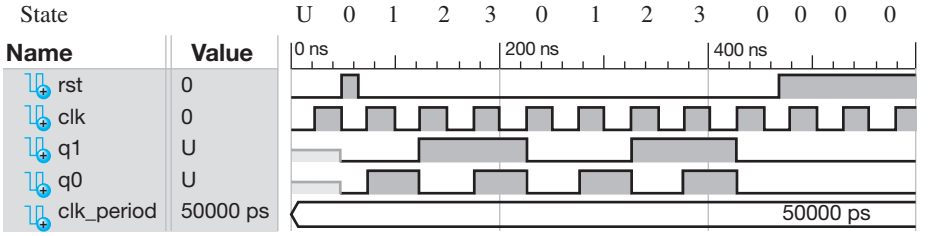
Things you should notice about the VHDL design in Listing 6.2:

- The mode for *Q*1 and *Q*0 is **inout** because the VHDL code must read *Q*1 and *Q*0 (mode in) and write *Q*1 and *Q*0 (mode out).
- Because the signals *D*1 and *D*0 are **internal signals** used inside the architecture, they are declared between **architecture** and the first **begin**. Internal signals do not require a mode. If a mode is supplied, this will be an error in your code.
- All the signals in this design are scalars.
- The excitation equations for *D*1 and *D*0 are assigned after the first **begin**. The right-hand side (or expression) for each excitation equation can be assigned directly to *Q*1—that is, *Q*1 <= *Q*1 **xor** *Q*0—and *Q*0—that is, *Q*0 <= **not** *Q*0. This eliminates the need for the signal declarations for *D*1 and *D*0 and also the assignments for *D*1 and *D*0.
- The **process** requires in its sensitivity list (i.e., the list following **process**) all the signals that must be read to start the process, which are *CLK* and *RST* for this design.
- The label *Q*1_output shows the beginning of the process for the DFF output *Q*1.
- The label *Q*0_output shows the beginning of the process for the DFF output *Q*0.

- The signal *RST* is a clock-independent reset signal because it is placed before rising_edge(*CLK*) in the VHDL code. When the signal *RST* is active or 1, it overrides the clock and clears the circuit to state 00.
- All complete process statements, from **process** through **end process**, are inherently concurrent statements within this mixed architecture declaration, even though they contain sequential statements. This allows multiple process statements to be used in a design.
- A single process can be used to simplify the VHDL code, because a single clock tick event can load all the *D* inputs. This design will work correctly with one process that contains the following **if statement**: **if** *RST* = '1' **then** *Q*1 <= '0'; *Q*0 <= '0'; **elsif** rising_edge(*CLK*) **then** *Q*1 <= *Q*1 **xor** *Q*0; *Q*0 <= **not** *Q*0; **end if**;.

Waveform 6.2 shows the simulation with the correct functionality of design entity Counter1.

WAVEFORM 6.2
Simulation with the correct functionality of design entity Counter1



The **state** of Counter1 is shown above Name in Waveform 6.2 in decimal. U represents an unknown state at the beginning of the simulation. Observe that signal *RST* is used to place the counter in state 0. The numbers 0, 1, 2, 3 are specified in decimal, where state 0 represents q1 q0 = 00, state 1 represents q1 q0 = 01, state 2 represents q1 q0 = 10, and state 3 represents q1 q0 = 11. Each state has a state time of one period of the clock from the rising edge to *CLK* the next rising edge of *CLK*, or 50 ns as observed in Waveform 2.

The counting sequence diagram and equivalent state diagram for a one-hot up counter (4 bits) for design entity Counter2, are shown in Figures 6.7a and 6.7b, respectively. Each flip-flop output that is turned on represents a state, for a one-hot counter.

FIGURE 6.7 One-hot up counter (4 bits): (a) counting sequence diagram; (b) equivalent state diagram

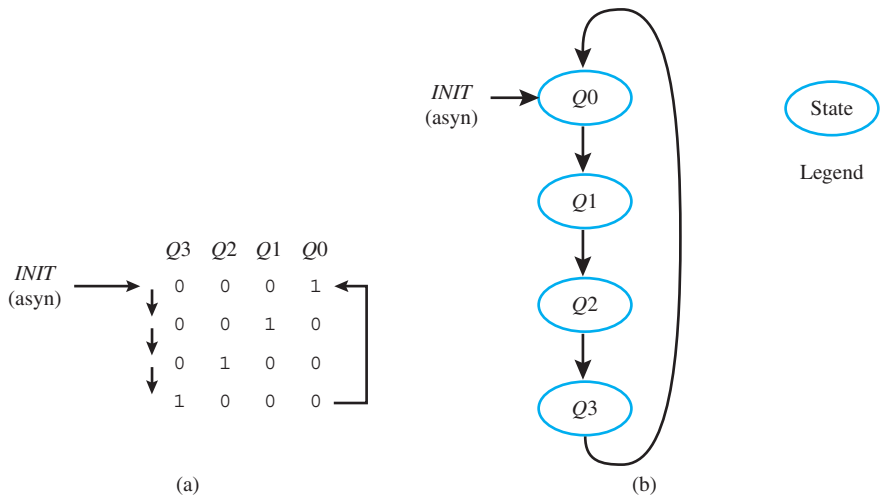


Figure 6.7 shows an asynchronous signal INIT (INITialize), which initializes the counter to 0001 (one of its one-hot states) independent of the clock. Using the Set OR Hold 1 equation, we can write the excitation equations for the *D* inputs for each of the four flip-flops *Q*3, *Q*2, *Q*1, and *Q*0 that are required for the design. After we obtain the excitation equations for the *D* inputs, we

can fill in the clouds of combinational logic for the D flip-flops and write the proper VHDL for the design.

From Figure 6.7a or 7b, we can write the excitation equations for the *D* inputs by inspection using the Set OR Hold 1 equation as shown here:

- $D3 = \overline{Q3}\cdot Q2\cdot\overline{Q1}\cdot\overline{Q0} = Q2$ because there is only one Set transition and the PS expression for the $0 \rightarrow 1$ transition of $Q3$ is $\overline{Q3}\cdot Q2\cdot\overline{Q1}\cdot\overline{Q0}$, which reduces to $Q2$ because of the one-hot definition for the present state $Q2$—that is, $(Q3 = Q1 = Q0 = 0)$.
- $D2 = \overline{Q3}\cdot\overline{Q2}\cdot Q1\cdot\overline{Q0} = Q1$ because there is only one Set transition and the PS expression for the $0 \rightarrow 1$ transition of $Q2$ is $\overline{Q3}\cdot\overline{Q2}\cdot Q1\cdot\overline{Q0}$, which reduces to $Q1$ because of the one-hot definition for the present state $Q1$—that is, $(Q3 = Q2 = Q0 = 0)$.
- $D1 = \overline{Q3}\cdot\overline{Q2}\cdot\overline{Q1}\cdot Q0 = Q0$ because there is only one Set transition and the PS expression for the $0 \rightarrow 1$ transition of $Q1$ is $\overline{Q3}\cdot\overline{Q2}\cdot\overline{Q1}\cdot Q0$, which reduces to $Q0$ because of the one-hot definition for the present state $Q0$—that is, $(Q3 = Q2 = Q1 = 0)$.
- $D0 = Q3\cdot\overline{Q2}\cdot\overline{Q1}\cdot\overline{Q0} = Q3$ because there is only one Set transition and the PS expression for the $0 \rightarrow 1$ transition of $Q0$ is $Q3\cdot\overline{Q2}\cdot\overline{Q1}\cdot\overline{Q0}$, which reduces to $Q3$ because of the one-hot definition for the present state $Q3$—that is, $(Q2 = Q1 = Q0 = 0)$.

So, $D3 = Q2$, $D2 = Q1$, $D1 = Q0$, and $D0 = Q3$. As you can see, it is easy to write the D excitation input equations for this simple one-hot up counter.

The *D* excitation equations allow us to draw the schematic shown in Figure 6.8.

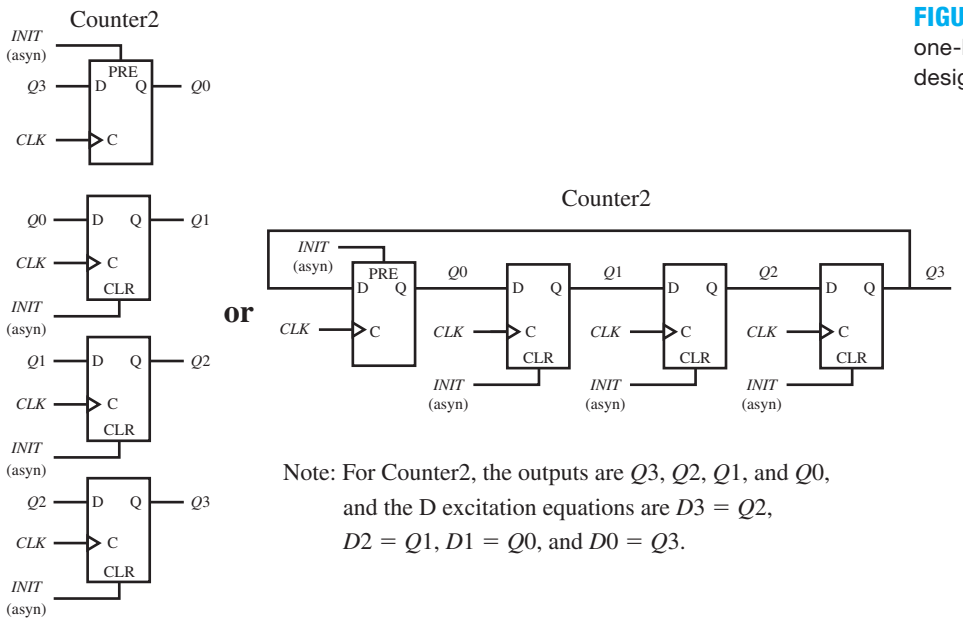


FIGURE 6.8 Schematic for a one-hot up counter (4 bits) for design entity Counter2

Notice that the clouds of combinational logic represent single-wire connections for this one-hot design. Also notice that the asynchronous *INIT* signal must set $Q0$ and reset $Q1$, $Q2$, and $Q3$ when it is asserted. This requires a different type of D flip-flop for $Q0$ compared to the D flip-flops for $Q1$ through $Q3$. The D flip-flop for $Q0$ requires an asynchronous PRE (Preset) input, while all the other D flip-flops require an asynchronous CLR (Clear) input.

The schematic for the one-hot up counter shown in Figure 6.8 is a special form of a shift register counter called a **ring counter**. Each flip-flop derives its *D* input from the output of the previous flip-flop thus forming a ring.

Listing 6.3 shows the VHDL design for the one-hot up counter (4 bits) in Figure 6.8 for design entity Counter2.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Counter2 is port (
     init, clk : in std_logic;
     q3, q2, q1, q0 : inout std_logic
     );
end Counter2;

architecture behavioral of Counter2 is
begin
process (init, clk)
begin
     if init = '1' then q3 <= '0'; q2 <= '0'; q1 <= '0'; q0 <= '1';
     elsif rising_edge(clk) then q3 <= q2; q2 <= q1; q1 <= q0; q0 <= q3;
     end if;
end process;
end behavioral;
```

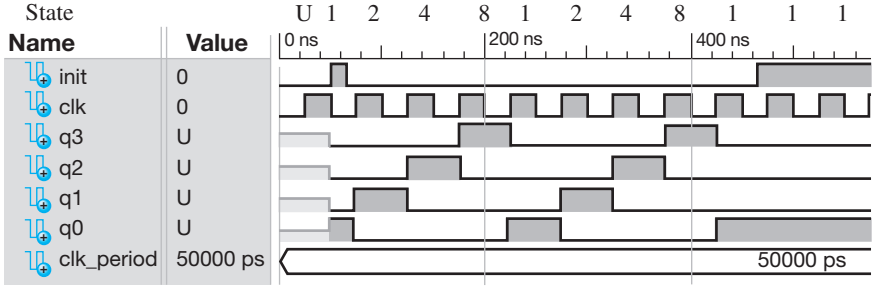
LISTING 6.3 Complete VHDL design for design entity Counter2 using a behavioral architecture declaration with a single process and if statement (project: Counter2)

Things you should notice about the VHDL design in Listing 6.3:

- All the signals in this design are scalars.
- The mode for *Q*3 through *Q*0 is **inout** because the VHDL code must read *Q*3 through *Q*0 (mode in) and write *Q*3 through *Q*0 (mode out).
- The **process** requires in its sensitivity list (i.e., the list following **process**) all the signals that must be read to start the process, which are *INIT* and *CLK* for this design.
- When *INIT* = '1' the state of the flip-flops for *Q*3 through *Q*0 is 0001, respectively. The *INIT* signal is asynchronous because it is placed before rising_edge(C*LK)*. When the init signal is active or 1, this signal overrides the clock and forces the circuit to one of its one-hot states—that is, *Q*0. The circuit cannot be cleared because this would place the circuit in a state that is not in its normal counting sequence.
- The design can be accomplished with just one process.

Waveform 6.3 shows the simulation with the correct functionality for design entity Counter2.

WAVEFORM 6.3 Simulation with the correct functionality of design entity Counter2



The state of Counter2 is shown above Name in Waveform 6.3 in decimal. U represents an unknown state at the beginning of the simulation. Observe that signal *INIT* is used to place the counter in state 1. The numbers 1, 2, 4, and 8 are specified in decimal, where state 1 represents q3 q2 q1 q0 = 0001, state 2 represents q3 q2 q1 q0 = 0010, state 4 represents q3 q2 q1 q0 = 0100, and state 8 represents q3 q2 q1 q0 = 1000.

6.6 NONCONVENTIONAL COUNTER DESIGN USING THE ALGORITHMIC EQUATION METHOD

Most simple counters are **conventional counters**, because they do not have repeating states. Each state in a conventional counter has a different state value. The simple counters we discussed in the last section were conventional counters. A simple counter that has repeating states (states with the same state value) in its counting sequence is a **nonconventional (NC) counter**. It is necessary to add additional flip-flops to a nonconventional counter to differentiate between the repeating states; that is, the additional flip-flops are used to convert a nonconventional counter to a conventional counter. Only one additional flip-flop is required for two repeating states. For additional repeating states, you must determine the number of additional flip-flops to add so that each state has a different state value—that is, remove all repeating states values so there are no repeating states.

Figure 6.9a shows a nonconventional counter (2 bits) with the repeating state value 10. Adding state variable $Q2$ (an additional D flip-flop $Q2$) removes the repeating state value 10 as shown in Figure 6.9b. The states in Figure 6.9b are now 001, 110, and 010. Figure 6.9c shows Figure 6.9b written in the format of a PS/NS table. Figures 6.9b and 6.9c do not have repeating states because $Q2$ has been added to remove the repeating states in Figure 6.9a. In other words, an additional D flip-flop $Q2$ was added to the counting sequence diagram to differentiate between the repeating states of $Q1$ $Q0$.

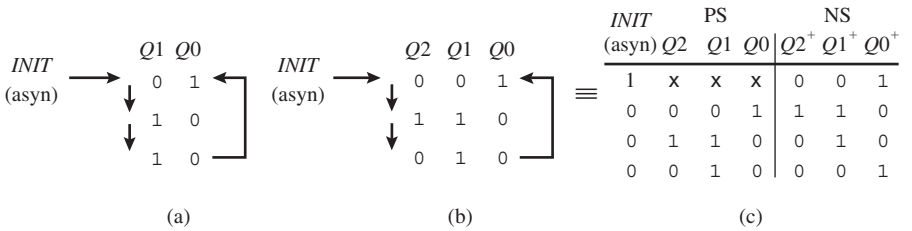


FIGURE 6.9
Nonconventional counter: (a) counting sequence diagram for NC counter (2 bits); (b) additional D flip-flop Q2 to differentiate between repeating states of Q1Q0; (c) equivalent PS/NS table

When the asynchronous input *INIT* is asserted or pulled high or to a 1, the counter goes to state 001. When the asynchronous input *INIT* is not asserted the counter responds to the clock. Each time the clock ticks, the counter changes from its present-state value to its next-state value—that is, 001 to 110, then 110 to 010, then 010 back to 001, and so on. The counting sequence diagram and the PS/NS table clearly show these transitions.

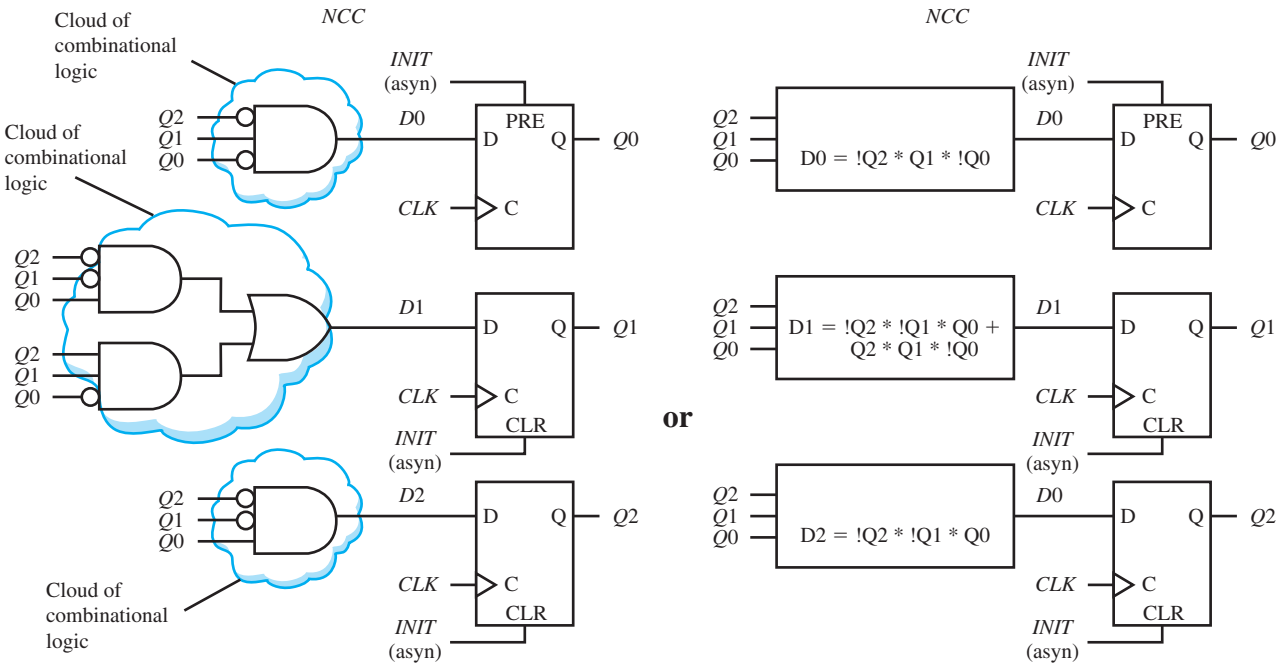
Using the Set OR Hold 1 equation, we can write the $D2$, $D1$, and $D0$ excitation equations from Figure 6.9b as follows:

$$D2 = \overline{Q2}\cdot\overline{Q1}\cdot Q0$$
$$D1 = \overline{Q2}\cdot\overline{Q1}\cdot Q0 + Q2\cdot Q1\cdot\overline{Q0}$$
$$D0 = \overline{Q2}\cdot Q1\cdot\overline{Q0}$$

Again you should notice how easy it is to write the D excitation equations by inspection via Figure 6.9b.

The D excitation equations allow us to draw the schematic shown in Figure 6.10a and fill in the clouds of combinational logic.

To simplify drawings, it is sometimes convenient to represent each combinational logic circuit as a rectangular box as shown in Figure 6.10b. The D excitation equation for each rectangular box may be listed separately or inside each box as shown in Figure 6.10b (NOTE: ! is NOT, * is AND, and + is OR).



Note: The counter outputs are $Q2$, $Q1$, and $Q0$ where $Q2$ is used to differentiate between repeating states of $Q1$ and $Q0$.

FIGURE 6.10 Schematic for nonconventional counter: (a) with gates and DFFs; (b) with rectangular boxes and DFFs

Listing 6.4 shows the VHDL design for the NCC (nonconventional counter) in Figure 6.10—that is, design entity NCC.

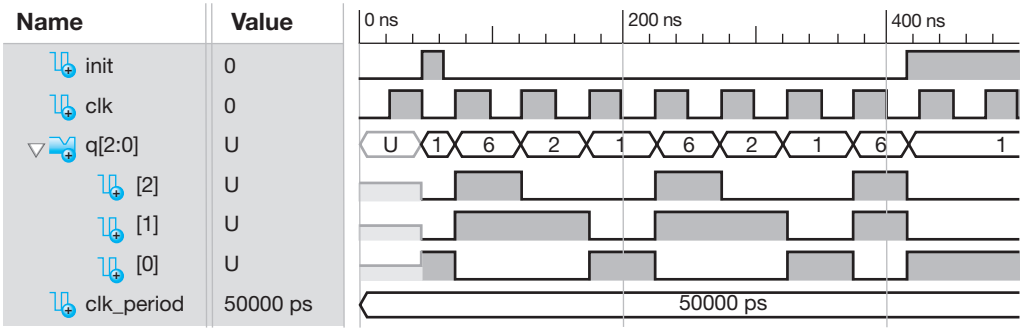
LISTING 6.4
Complete VHDL design for design entity NCC using a dataflow architecture declaration (project: NCC)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity NCC is port (
    init, clk : in std_logic;
    q : inout std_logic_vector (2 downto 0)
    );
end NCC;

architecture dataflow of NCC is
    signal d2,d1,d0: std_logic;
begin
    d2 <= (not q(2) and not q(1) and     q(0));
    d1 <= (not q(2) and not q(1) and     q(0)) or
          (    q(2) and     q(1) and not q(0));
    d0 <= (not q(2) and     q(1) and not q(0));
    q <= "001"      when init = '1' else
         (d2,d1,d0) when rising_edge(clk);
end dataflow;
```

Things you should notice about the VHDL design in Listing 6.4:

- All the signals in this design are scalars except *Q*, which is a vector.
- The mode for *Q* is **inout** because the VHDL code must read *Q*(2), *Q*(1), and *Q*(0)—that is, mode in—and write *Q*(2), *Q*(1), and *Q*(0)—that is, mode out.
- Simple signal assignments are used to assign the *D* inputs for the flip-flops. Observe that *Q* is assigned the *D*2, *D*1, and *D*0 values for the NC counter via the **aggregate** (*D*2, *D*1, *D*0). The order in which these signals are assigned is important because *Q* was declared as a std_logic_vector (2 **downto** 0). *Q* may also be assigned the *D*2, *D*1, and *D*0 values for the NC counter via **concatenation operators** as *D*2& *D*1& *D*0 **when** rising-edge (CLK).

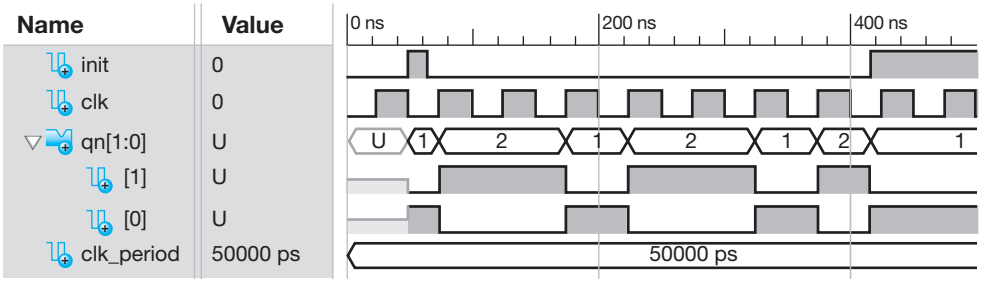
Waveform 6.4 shows the simulation with the correct functionality of design entity NCC.



WAVEFORM 6.4
Simulation with the correct functionality of design entity NCC

The state of NCC is shown inside the Waveform signal q(2:0) in Waveform 6.4. At the beginning of the simulation for q(2:0), U represents an unknown state. Observe that the waveforms for q(1) q(0) follow the repeating state sequence 01 10 10, and waveform q(2) provides the necessary distinction between waveform q(1) and waveform q(0).

As you can observe in Waveform 6.4, the simulation provides an output of all the port signals in the entity. To provide only the nonconventional counter outputs in the simulation, rename the nonconventional counter outputs qn(1) and qn(0), and place the port signal declaration qn: **out** std_logic_vector (1 **downto** 0) in the entity. Remove the port declaration q : **inout** std_logic_vector (2 **downto** 0) from the entity, and place the internal signal declaration **signal** q: std_logic_vector (2 **downto** 0) between **architecture** and the first **begin**. Place the assignment statement qn <= q (1 **downto** 0) in the body of the **architecture** after the first **begin**. Waveform 6.5 shows the simulation that result from these changes. Remember: A simulation only displays the signals in the entity declaration—that is, the external signals not the internal signals.



WAVEFORM 6.5
Simulation with the correct functionality of design entity NCC_MOD with changes (project: NCC_MOD)

The state of NCC_MOD is shown inside the waveform signal q(1:0) in Waveform 6.5. Observe that the waveforms for qn(1) qn(0) follow the repeating state sequence 01 10 10.

6.7 COUNTER DESIGN USING THE ARITHMETIC METHOD

The **arithmetic method (AM)** is perhaps the most concise method, but it is limited to counting up or counting down by a fixed integer value, i.e., 1, 2, 3, . . . , etc. When using this method you do not have to obtain the excitation equations for the flip-flops, and you do not have to draw the circuit. You must know how to use a conditional signal assignment or a process with an if statement to enter the counting sequence via an arithmetic "+" or "−" operator in VHDL and then simply let the VHDL software do the work. You can always observe the circuit that is produced using either **View RTL Schematic** or **View Technology Schematic** when using Xilinx software. Other software vendors may have a similar option to allow you to view the resulting schematic.

Listing 6.5 shows a complete VHDL design for the binary up counter (2 bits) with a clock-independent reset using an arithmetic "**+**" operator for design entity Counter1_AM. See Section 6.5, Figure 6.2, for the counting sequence diagram and the equivalent state diagram.

LISTING 6.5
Complete VHDL design for design entity Counter1_AM using a behavioral architecture declaration with an arithmetic "+" operator (project: Counter1_AM)

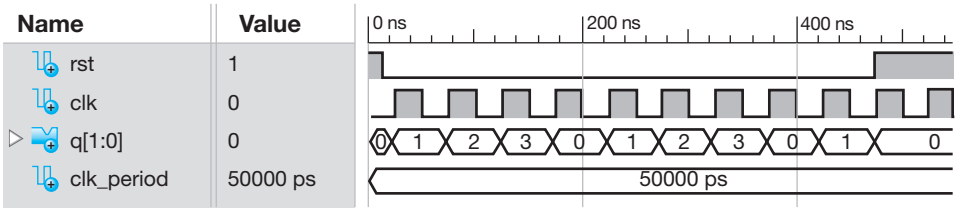
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter1_AM is port (
    rst, clk : in std_logic;
    q : inout std_logic_vector (1 downto 0)
    );
end Counter1_AM;

architecture behavioral of Counter1_AM is
begin
process (clk, rst)
begin
    if rst = '1' then q <= "00";
    elsif rising_edge(clk) then q <= q + 1;
    end if;
end process;
end behavioral;
```

Things you should notice about the VHDL design in Listing 6.5:

- A new use clause must be added to the design in order to use the arithmetic "**+**" operator. The package IEEE.STD_LOGIC_UNSIGNED specified in **use** IEEE.STD_LOGIC_UNSIGNED.**all**; is required because it contains the definition for the arithmetic "+" operator.
- Output *Q* is written in bus notation as std_logic_vector (1 **downto** 0), because *Q*(1) and *Q*(0) need to be specified as a single unit for performing the "+" or arithmetic addition operation.
- The process requires in its sensitivity list (i.e., the list following **process**) all the signals that must be read to start the process, which are *CLK* and *RST* for this design.
- The mode for *Q* is **inout** because the VHDL code must read *Q* (mode in) and write *Q* (mode out) as specified by the signal assignment statement *Q* <= *Q* + 1. Note: To count down, use the arithmetic "−" operator.
- The signal *RST* is a clock-independent asynchronous reset signal because it is placed before rising_edge(C*LK)* in the VHDL code. When the signal *RST* is active or 1, it overrides the clock and clears the counter to state 00.

- A string of bits—that is, more than one bit—must be included in double quotation marks. Recall that a single bit is included in single quotation marks.

Waveform 6.6 shows the simulation with the correct functionality of design entity Counter1_AM.



WAVEFORM 6.6 Simulation with the correct functionality of design entity Counter1_AM

The state of Counter1_AM is shown inside the waveform signal q(1:0) in Waveform 6.6. The numbers 0, 1, 2, and 3 are specified in decimal, where state 0 represents q(1:0) = 00, state 1 represents q(1:0) = 01, State 2 represents q(1:0) = 10, and state 3 represents q(1:0) = 11.

6.8 FREQUENCY DIVISION (SLOWING DOWN A FAST CLOCK FREQUENCY)

Sometimes it may be desirable to operate a **state machine** at a much lower or slower frequency than the frequency provided by the **crystal clock oscillator** on a **hardware board**. This would be the case if we were to implement a state machine and wanted to see it operating via LEDs (light emitting diodes) flashing on a hardware board that has a high-frequency crystal clock oscillator. We would definitely need to provide a slower clock frequency to the state machine. Frequency division can be used to effectively (not literally) slow down the frequency of the crystal clock oscillator on the board.

Suppose our hardware board came with a crystal clock oscillator frequency of just 1000 cycles per second or 1000 Hz [most clock oscillators operate in the megahertz (MHz) range, which is much faster than this simple hypothetical example]. A frequency of 1000 Hz is still much too fast to see our state machine operating LEDs on the board. We would like to operate our state machine at approximately 1 Hz so we can see the LEDs flashing on and off in the proper sequence. This can be done by designing a binary counter that divides the crystal clock oscillator frequency by 1000. Because counters operate in binary, the closest integer to 1000 that is a power of 2 is 2^{10}, or 1024. That means that we need to design a 10-bit counter and use the highest output bit position (the ninth output bit position) as the clock input to our state machine.

The arithmetic method is the easiest way to generate a very large binary up or binary down counter. Using the ninth bit of a 10-bit counter, COUNT(9) divides the clock frequency *CLK* by 2^{10}, or 1024, thus providing a signal *SLOW_CLK* that has a frequency that is approximately 1 Hz (actually, 1000 Hz/ 2^{10} = 0.9766 Hz).

Figure 6.11 shows a design entity FD_SM, which represents a frequency divider and a state machine.

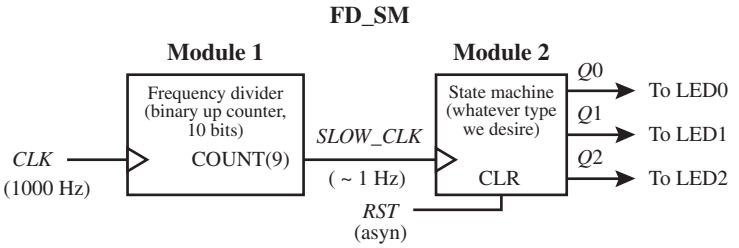


FIGURE 6.11 Annotated schematic diagram for a frequency divider and a state machine

Module 1, or the block labeled Frequency divider in Figure 6.11, shows *SLOW_CLK* driving module 2, or the block labeled State machine.

Listing 6.6 shows a VHDL design for design entity FD_SM using a flat design approach. Observe that the arithmetic method is used for the frequency divider design. The process for the state machine in Listing 6.6 is left blank because the state machine is only partially specified in Figure 6.11 via its inputs and outputs.

LISTING 6.6 VHDL design for design entity FD_SM (frequency divider and state machine) using a flat design approach

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FD_SM is port (
     rst, clk : in std_logic;
     q : inout std_logic_vector (2 downto 0)
     );
end FD_SM;

architecture mixed of FD_SM is
     signal COUNT: std_logic_vector (9 downto 0);
     signal SLOW_CLK: std_logic;
begin

--Module 1, Frequency Divider
     COUNT <= COUNT + 1 when rising_edge(clk);
     SLOW_CLK <= COUNT(9);

--Module 2, State Machine
state_machine: process (rst, SLOW_CLK) begin
          .
          .
          .
end process state_machine;
end mixed;
```

Things you should notice about the VHDL for design entity FD_SM in Listing 6.6:

- The VHDL code for module 1 and module 2 are included within a single architecture declaration, which is the format for a flat design approach.
- The **signal** *COUNT* and the **signal** *SLOW_CLK* are internal signals and must be declared between **architecture** and the first **begin**. Because a counter with 10 bits is needed, the data type for *COUNT* is std_logic_vector (9 **downto** 0). *SLOW_CLK* is a single signal with a std_logic data type.
- Mixed is used as the architecture name to indicate that two different design styles are used for the design entity FD_SM. A dataflow design style is used for the module (Frequency Divider) while a behavioral design style is used for the module (State Machine).
- A conditional signal assignment is used to create a counter, and a Boolean equation is used to assign signal *COUNT*(9) to the signal *SLOW_CLK*.
- *SLOW_CLK* must be used in the sensitivity list as the clock in the process named state_machine.

The frequency of the programmable silicon oscillator on a hardware board such as the BASYS 2 board (manufacture by Digilent; go to digilentinc.com) has three different clock settings: 25 MHz, 50 MHz, and 100 MHz with an accuracy of about ±0.5%. The frequency of the crystal clock oscillator on a hardware board such as the NEXYS 2 board has a fixed frequency of 50 MHz with an accuracy of about ±0.01%. To see a state machine operating via LEDs at

any of these frequencies would require a slower clock frequency. To accurately divide 50 MHz to obtain a frequency of 1 Hz, the arithmetic method can be used to count the number of clock cycles (25,000,000 clock cycles) that must occur for half the period of 1 Hz, which is $T/2 = (1/f)/2 = (1/(1\text{ Hz}))/2 = 0.5$ s. A signal called *SLOW_CLK* can be toggled between 1 and 0 at 0.5-s intervals to produce a frequency of 1 Hz.

Listing 6.7 shows the complete VHDL design for design entity ACCURATE_CLK_1HZ.

```
libraryIEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ACCURATE_CLK_1HZ is port (
     clk : in std_logic;   --50 MHz clock
     slow_clk : inout std_logic
     );
end ACCURATE_CLK_1HZ;

architecture behavioral of ACCURATE_CLK_1HZ is
     signal count : integer;
     constant max_count: integer := 25000000;
          --25M clock cycles for half the period (0.5 s)
begin
process (clk)
begin
     if rising_edge(clk) then count <= count + 1;
          if count = max_count then slow_clk <= not slow_clk; count <= 0;
          end if;
     end if;
end process;
end behavioral;
```

LISTING 6.7 Complete VHDL design for design entity ACCURATE_CLK_1HZ (project: ACCURATE_CLK_IHZ)

Things you should notice about the VHDL for design entity ACCURATE_CLK_1HZ in Listing 6.7:

- The **signal** *COUNT* is an internal signal and must be declared between **architecture** and the first **begin**. The data type for *COUNT* is an integer and does not have to be bounded or assigned a range.
- The keyword **constant** is used to declare a fixed value named max_count as a data-type integer with the assigned value of 25,000,000. Constants are used for easy readability. We could have simply removed the constant declaration and substituted 25000000 for the value of max_count inside the second if statement. Data-type integers do not have commas or quotation marks around the values of the integers—that is, 25000000 not "25,000,000."
- Only input *CLK* is required in the sensitivity list for the frequency divider process.
- Data type integers do not have quotation marks—that is, count <= 0 not count <= '0'.
- A single process with two if statements and the assignment *COUNT* <= *COUNT* +1 is used to create a counter. *SLOW_CLK* is toggled between 1 and 0 precisely when count = max_count, which is at 0.5-s intervals, thus producing an output frequency of 1 Hz.
- This type of design is only required where you need an exact frequency division as for a digital stop watch or a digital clock.

6.9 COUNTER DESIGN USING THE PS/NS TABULAR METHOD

The PS/NS tabular method is perhaps one of the easiest methods to use and understand. When using this method you do not have to obtain the excitation equations for the flip-flops, and you do not have to draw the circuit. You must know how to use a process with an if statement and a case statement to enter the counting sequence in VHDL, and then simply let the VHDL software do the work. You can always observe the circuit that is produced using either **View RTL Schematic** or **View Technology Schematic** which is under **Synthesize**, when using Xilinx software. Other software vendors have a similar option to allow you to view the resulting schematic.

As a handy reference, the counting sequence diagram (or state sequence diagram) for the binary up counter (2 bits) with a clock-independent reset input and the equivalent state diagram is repeated in Figures 6.12a and 6.12b. Figure 6.12c also shows the equivalent PS/NS Table for the binary up counter.

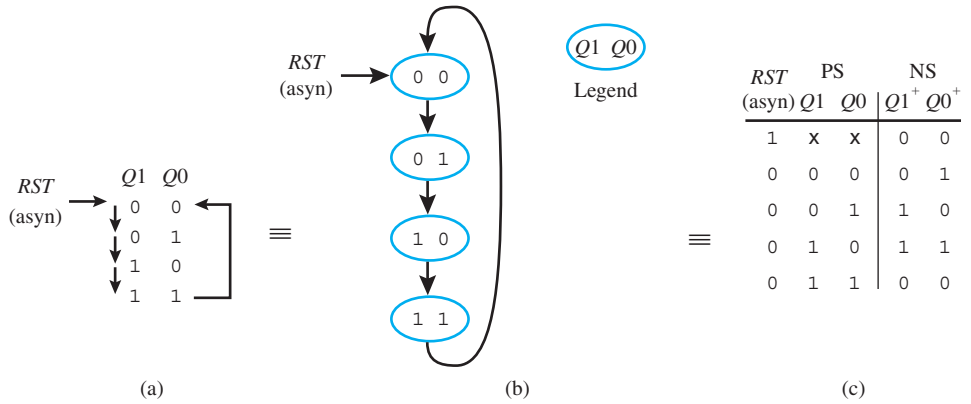


FIGURE 6.12 Binary up counter (2 bits): (a) Counting sequence diagram; (b) equivalent state diagram; (c) equivalent PS/NS table

The counting sequence diagram, state diagram, and the PS/NS Table in Figure 6.12 each represent a binary up counter in a slightly different format. Remember, when the asynchronous input *RST* is asserted, the counter goes to state 00. When the asynchronous input *RST* is not asserted, the counter responds to the clock. Each time the clock ticks, the counter changes from its present-state value to its next-state value—that is, 00 to 01, then 01 to 10, then 10 to 11, then 11 back to 00, and so on.

Listing 6.8 shows a complete VHDL design for the binary up counter (2 bits) with an if statement and a case statement using the PS/NS tabular method for design entity Counter1_TM.

LISTING 6.8 Complete VHDL design for design entity Counter1_TM using a behavioral architecture declaration with an if statement and a case statement (project: Counter1_TM)

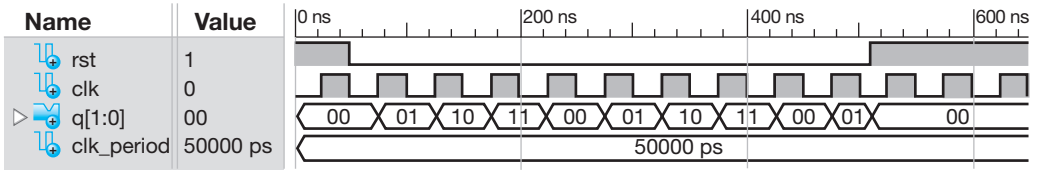
```
libraryIEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Counter1_TM is port (
     rst, clk : in std_logic;
     q : inout std_logic_vector (1 downto 0)
     );
end Counter1_TM;

architecture behavioral of Counter1_TM is
begin
process (rst, clk)
begin
     if rst = '1' then q <= "00";
     elsif rising_edge(clk) then
```

```
        case q is
            when "00" => q <= "01";
            when "01" => q <= "10";
            when "10" => q <= "11";
            when "11" => q <= "00";
            when others => null;
        end case;
    end if;
end process;
end behavioral;
```

Things you should notice about the VHDL design Listing 6.8:

- Output *Q* is written in bus notation as **inout** std_logic_vector (1 **downto** 0), because *Q*(1) and *Q*(0) need to be specified as a single unit for the case statement—that is, "00" for the PS (present state) and "01" for the NS (next state) as represented by **when** "00" => *Q* <= "01";.
- A **when others** clause is required at the end of the choice list of the case statement to ensure that all possible select values for signal *Q* are included in the choice list after **when**. The value **null** means do nothing. The std_logic values that must be included for signal *Q* are 0, L, 1, H, Z, and -. Note that the single dash "-" represents a don't care in VHDL.

Waveform 6.7 shows the simulation with the correct functionality of design entity Counter1_TM.



WAVEFORM 6.7 Simulation with the correct functionality of design entity Counter1_TM

The state of the counter is shown inside the Waveform signal q(1:0) in Waveform 6.7 in binary.

Waveforms 6.2, 6.6, and 6.7 each provide the correct functionality for a binary up counter (2 bits). This shows that the algorithmic equation method, the arithmetic method, and the PS/NS tabular method are simply three different design methodologies that provide the same result.

Figures 6.13a and 6.13b show two equivalent state diagrams for a one-hot up counter (3 bits) for design entity OHUC_3B.

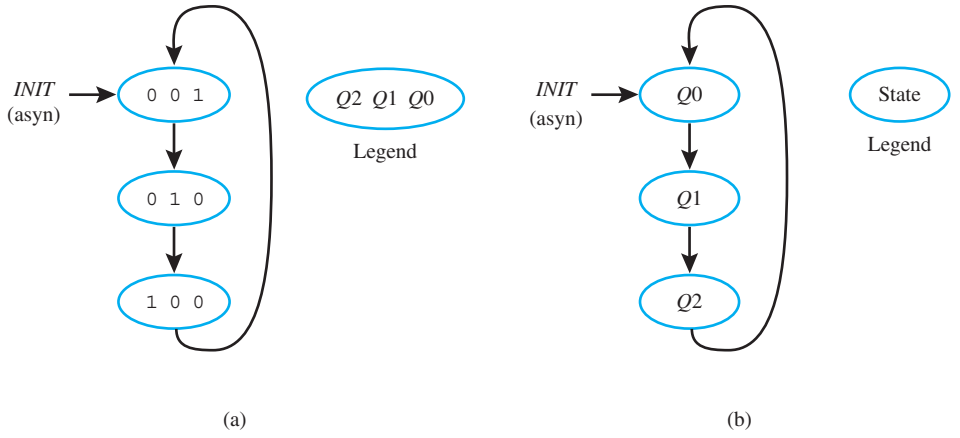


FIGURE 6.13 Equivalent state diagrams for a one-hot up counter (3 bits): (a) traditional; (b) simplified.

Listing 6.9 shows the complete VHDL design for design entity OHUC_3B using the PS/NS tabular method for design entity OHUC_3B.

LISTING 6.9
Complete VHDL design for design entity OHUC_3B using the PS/NS tabular method (project: OHUC_3B)

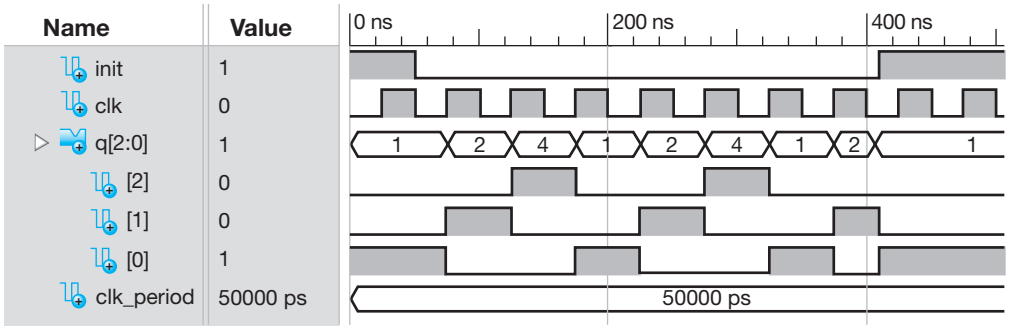
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OHUC_3B is port (
     init,clk : in std_logic;
     q : inout std_logic_vector (2 downto 0)
     );
end OHUC_3B;

architecture behavioral of OHUC_3B is
begin
process (init,clk)
begin
     if init = '1' then q <= "001";
     elsif rising_edge(clk) then
          case q is
               when "001" => q <= "010";
               when "010" => q <= "100";
               when "100" => q <= "001";
               when others => null;
          end case;
     end if;
end process;
end behavioral;
```

Things you should notice about the VHDL design in Listing 6.9:

- The process requires in its sensitivity list (i.e., the list following **process**) all the signals that must be read to start the process, which are *CLK* and *INIT* for this design.
- The signal *INIT* is a clock-independent asynchronous initialization signal because it is placed before rising_edge(C*LK)* in the VHDL code. When the signal *INIT* is active or 1, it overrides the clock and initializes the counter to state 001.

Waveform 6.8 shows the simulation with the correct functionality of design entity OHUC_3B.

WAVEFORM 6.8
Simulation with the correct functionality of design entity OHUC_3B



The state of the counter is shown inside the Waveform signal q(2:0) in Waveform 6.8.

6.10 NONCONVENTIONAL COUNTER DESIGN USING THE PS/NS TABULAR METHOD

As we mentioned earlier, a simple counter that has repeating states in its counting sequence is a **nonconventional counter**. This type of simple counter needs additional flip-flops to differentiate between the repeating states.

The nonconventional counter (2 bits) with the repeating state value 10 that was presented earlier is repeated in Figure 6.14a. The additional flip-flop that is used to differentiate between the repeating states is shown in Figures 6.14b and 6.14c.

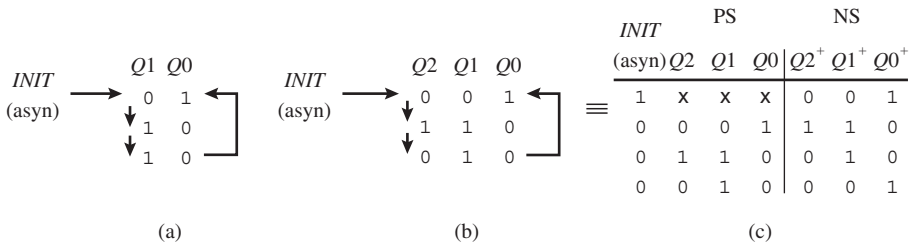


FIGURE 6.14
Nonconventional Counter: (a) counting sequence diagram for nonconventional counter (2 bits); (b) additional D flip-flop Q2 to differentiate between repeating states of Q1Q0; (c) equivalent PS/NS table

Listing 6.10 shows a complete VHDL design for the nonconventional counter (2 bits) with an if statement and a case statement using the PS/NS tabular method. The output of the nonconventional counter is represented in the design as *QN*. The output of the counter with the added state is represented as *Q* in the design in Listing 6.10, *QN* is used as the port signal for the output, while *Q* is used as an internal signal.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity NCC_TM is port (
     init, clk : in std_logic;
     qn : inout std_logic_vector (1 downto 0)
     );
end NCC_TM;

architecture behavioral of NCC_TM is
     signal q : std_logic_vector (2 downto 0);
begin
process (init, clk, q)
begin
     if init = '1' then q <= "001";
     elsif rising_edge(clk) then
          case q is
               when "001" => q <= "110";
               when "110" => q <= "010";
               when "010" => q <= "001";
               when others => null;
          end case;
     end if;
     qn <= q (1 downto 0);
end process;
end behavioral;
```

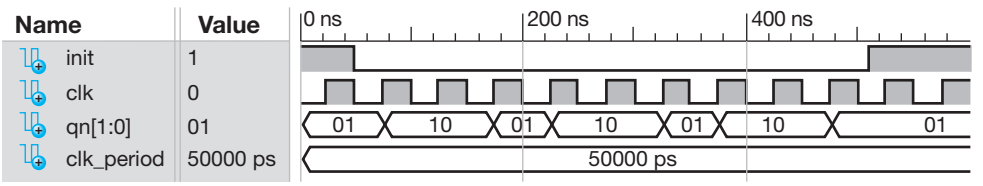
LISTING 6.10
Complete VHDL design for design entity for NCC_TM with an if statement and a case statement using the PS/NS tabular method (project: NCC_TM)

Things you should notice about the VHDL design in Listing 6.10:

- All the signals in this design are scalars except *QN* and *Q*, which are vectors.
- Signal *QN* is declared as a port signal and written in bus notation as std_logic (1 downto 0).
- Signal *Q* is declared as an internal signal and written in bus notation as std_logic_vector (2 **downto** 0), because *Q*(2), *Q*(1), and *Q*(0) need to be specified as a single unit for the case statement—that is, "001" for the PS (present state) and "110" for the NS (next state) as represented by **when** "001" => *Q* <= "110";.
- No mode is required for signal *Q*, because it is an internal signal.
- The process requires in its sensitivity list (i.e., the list following **process**) all the signals that must be read in the process, which are *INIT*, *CLK*, and *Q*. If *Q* is not included in the list, *QN* would not be assigned the value of *Q* until the clock goes low, which would delay the output *QN* by half a clock cycle, which would result in an incorrect design.
- The signal *INIT* is a clock-independent asynchronous reset signal because it is placed before rising_edge(C*LK)* in the VHDL code. When the signal *INIT* is active or 1, it overrides the clock and initializes the counter to state 001.
- The assignment *QN* <= *Q* (1 **downto** 0) is made to generate only the outputs *QN*(1) and *QN* (0)—that is, *Q*(2) is not output because it is an internal signal.

Waveform 6.9 shows the simulation with the correct functionality of design entity NCC_TM.

WAVEFORM 6.9
Simulation with the correct functionality of design entity NCC_TM



The state of the counter is shown inside the waveform signal q(1:0) in Waveform 6.9 in binary. Observe that qn(1:0) maintains the same value of 10 for two clock ticks as it should, as shown in the counting sequence diagram in Figure 6.14a.

PROBLEMS

Section 6.2 Synchronous Circuits

6.1 What is the emphasis of this chapter?

6.2 Circuits that change state in step with a particular input signal called a clock are called what types of circuits?

6.3 Name two types of logic circuits that store a single bit.

6.4 From the following list of flip-flops, which type is predominantly used in modern programmable devices?
a. Data-type flip-flops
b. SR-type flip-flops
c. JK-type flip-flops
d. T-type flip-flops

Section 6.3 Creating D-type Flip-Flops in VHDL

6.5 The logic symbol for a standard DFF with a CLR input is shown in Figure P6.5. Complete the following sentence: When the CLR input is asserted or pulled high or to a logic 1,

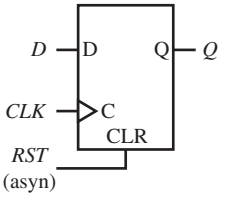


FIGURE P6.5

6.6 Explain what "(asyn)" indicates in Figure P6.5.

6.7 The logic symbol for a standard DFF with a PRE input is shown in Figure P6.7. Complete the following sentence: When the PRE input is asserted or pulled high or to a logic 1,

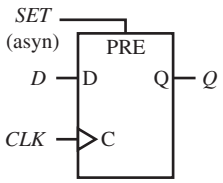


FIGURE P6.7

6.8 Explain what "(asyn)" indicates in Figure P6.7.

6.9 Labels are used inside logic symbols. Signals are assigned to inputs and outputs of logic symbols. Is VHDL code written with labels or is VHDL code written with signals?

6.10 Is the logic symbol in Figure P6.5 a positive edge-triggered or a negative edge-triggered D flip-flop? Provide a reason for your answer.

6.11 How is the clock signal *CLK* specified in VHDL code for a positive edge-triggered flip-flop?

6.12 How is the clock signal *CLK* specified in VHDL code for a negative edge-triggered flip-flop?

6.13 What is another mode for inout for a signal declared in the entity declaration when a signal must be read (mode in) as well as written (mode out)?

6.14 Which event does falling_edge(clk) represent? Recall that rising_edge(clk) represents the event caused by *CLK* changing from 0 to 1.

6.15 Write a behavioral architecture declaration for the DFF shown in Figure P6.15. Use an if statement and an elsif statement. The signal *SET* (asyn) is clock independent—that is, it overrides the clock.

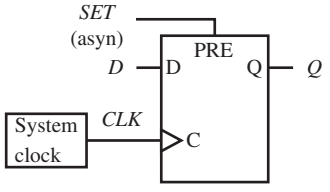


FIGURE P6.15

6.16 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the DFF in problem 6.15.

6.17 Combine your code for problems 6.15 and 6.16 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

6.18 Write a dataflow architecture declaration for the D flip-flop in Figure P6.18 using a conditional signal assignment. The signal *SET* (asyn) is clock independent—that is, it overrides the clock.

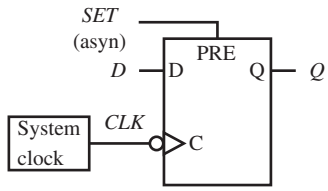


FIGURE P6.18

6.19 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the DFF in problem 6.18.

6.20 Combine your code for problems 6.18 and 6.19 to form a complete VHDL design. Obtain a Waveform diagram that shows correct functionality for the complete VHDL design.

6.21 Draw a logic symbol that represents the VHDL code with the following conditional signal assignment as the only assignment in the architecture:

```
Q <= '0' when RST = '1' else
     D when rising_edge(clk);
```

6.22 Draw a logic symbol that represents the VHDL code with the following if statement as the only assignment in the architecture:

```
if rising_edge(clk) then
    if RST = '1' then Q <= '0';
    else Q <= D;
    end if;
end if;
```

6.23 Draw a logic symbol that represents the VHDL code with the following if statement as the only assignment in the architecture:

```
if SET = '1' then Q <= '1';
elsif rising_edge(clk) then Q <= D;
end if;
```

6.24 Draw a logic symbol that represents the VHDL code with the following if statement as the only assignment in the architecture:

```
if rising_edge(clk) then
    if SET = '1' then Q <= '1';
    else Q <= D;
    end if;
end if;
```

Section 6.4 Designing Simple Synchronous Circuits

6.25 What is our definition of a simple counter?

6.26 What are the three methods presented in this chapter for designing simple counters suitable for VHDL?

6.27 What is another name for a state machine?

6.28 What is the name of the method for the design of a simple counter for VHDL that requires the user to draw a logic circuit for a design?

6.29 What is the name of the method for the design of a simple counter for VHDL that requires the user to use the "+" or "−" operator?

6.30 What is the name of the method for the design of a simple counter for VHDL that requires the user to use an if statement and a case statement?

Section 6.5 Counter Design Using the Algorithmic Equation Method

6.31 Verify that the counter in Figure P6.31a can be represented by the circuit shown in Figure P6.31b by using the Set OR Hold 1 equation to obtain the D excitation equations for the circuit. What would be an appropriate name for this counter?

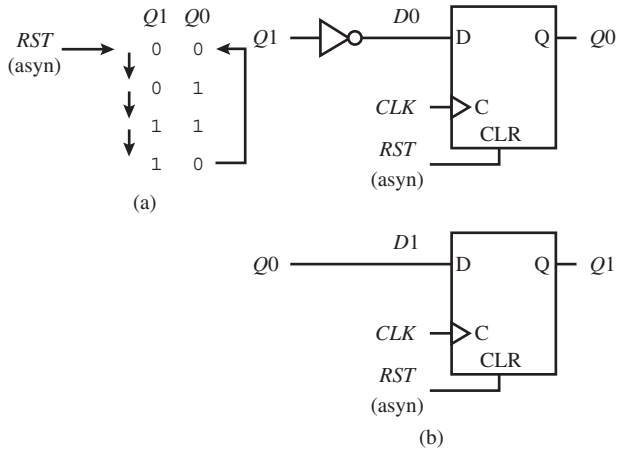


FIGURE P6.31

6.32 Verify that the counter in Figure P6.32a can be represented by the circuit shown in Figure P6.32b by using the Set OR Hold 1 equation to obtain the D excitation equations for the circuit. What would be an appropriate name for this counter?

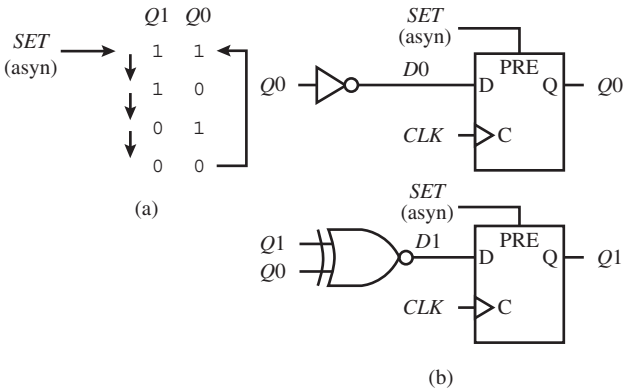


FIGURE P6.32

6.33 Use the algorithmic equation method to obtain a schematic for the counter shown in Figure P6.33 using DFFs. Hint: For one-hot counters, the current states can be represented as simply $Q2$, $Q1$, and $Q0$. This is only true for one-hot counters.

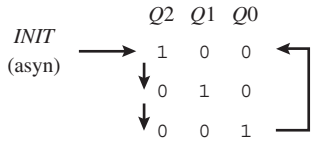


FIGURE P6.33

6.34 What is the definition of a net in a circuit?

6.35 Write a behavioral architecture declaration for the counter circuit in Figure P6.35. Use two processes with if statements. The signal *RST* (asyn) is clock independent—that is, it overrides the clock. Can a single process be used for this counter?

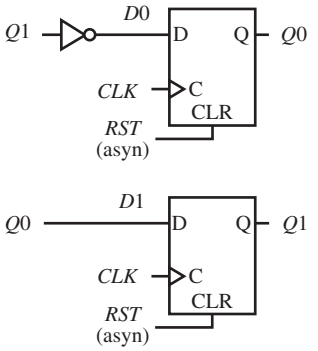


FIGURE P6.35

6.36 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter circuit in problem 6.35.

6.37 Combine your code for problems 6.35 and 6.36 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

6.38 Write a dataflow architecture declaration for the counter circuit in Figure P6.38. Use two conditional signal assignments. Input signal *SET* (asyn) is clock independent—that is, it overrides the clock.

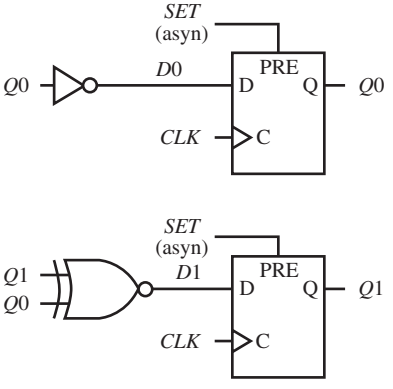


FIGURE P6.38

6.39 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter circuit in problem 6.38.

6.40 Combine your code for problems 6.38 and 6.39 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

6.41 Repeat problem 6.38. Use a single conditional signal assignment statement with an aggregate. Hint: Make *Q* a vector.

6.42 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter in problem 6.41.

6.43 Combine your code for problems 6.41 and 6.42 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

6.44 Write a dataflow architecture declaration for the counter circuit in Figure P6.44. Use three conditional signal assignments. The signal *INIT* (asyn) is clock independent—that is, it overrides the clock.

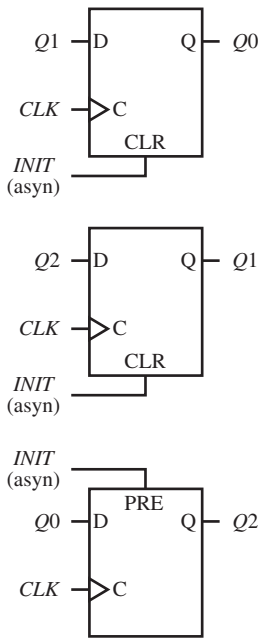


FIGURE P6.44

6.45 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter circuit in problem 6.44.

6.46 Combine your code for problems 6.44 and 6.45 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

Section 6.6 Nonconventional Counter Design Using the Algorithmic Equation Method

6.47 What is a simple conventional counter?

6.48 What is a simple nonconventional counter?

6.49 To design a simple nonconventional counter, what must be done to the nonconventional counter?

6.50 Explain how a simple counter can be design that sequences through the states values 100 100 100 001 each time the clock ticks?

6.51 Explain how a simple counter can be designed that sequences through the states values 00 01 00 01 10 11 10 11 each time the clock ticks.

6.52 A design specification for a simple nonconventional counter is shown in the counting sequence diagram in Figure P6.52. Add the required number of FFs to the counting sequence diagram so that the counter does not have repeating state values.

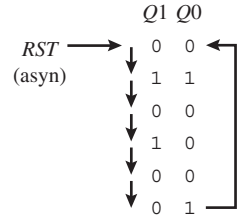


FIGURE P6.52

6.53 A design specification for a simple nonconventional counter is shown in the counting sequence diagram in Figure P6.53a. One FF has been added to the counting sequence diagram to remove the repeating state value (see Figure P6.53b). Use the algorithmic equation method to design the simple nonconventional counter. First write the D excitation equations for the design using the Set OR Hold 1 method. Next, draw an annotated schematic for the design using D type FFs with the nonconventional counter outputs named *QN*1 and *QN*0. Then, write complete VHDL code for the design using a mixed architecture declaration with Boolean equations and a single process. Run a simulation to verify correct functionality, and include the waveform diagram as part of your solution.

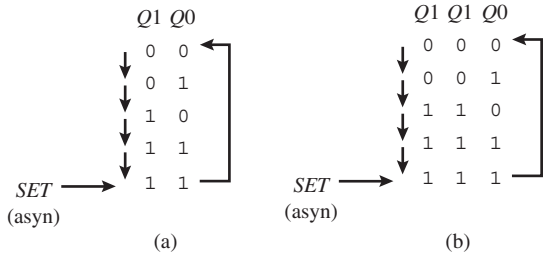


FIGURE P6.53

6.54 A design specification for a simple nonconventional counter is shown in the counting sequence diagram in Figure P6.54a. Two FFs have been added to the counting sequence diagram to remove repeating state values (see Figure P6.54b). Use the algorithmic equation method to design the simple nonconventional counter. First write the D excitation equations for the design using the Set OR Hold 1 method. Next, draw an annotated schematic for the design using D type FFs with the nonconventional counter outputs named *QN*3, *QN*2, *QN*1, and *QN*0. Then, write complete VHDL code for the design using a mixed architecture declaration with Boolean equations and a single conditional signal assignment statement. Run a simulation to verify correct functionality, and include the waveform diagram as part of your solution.

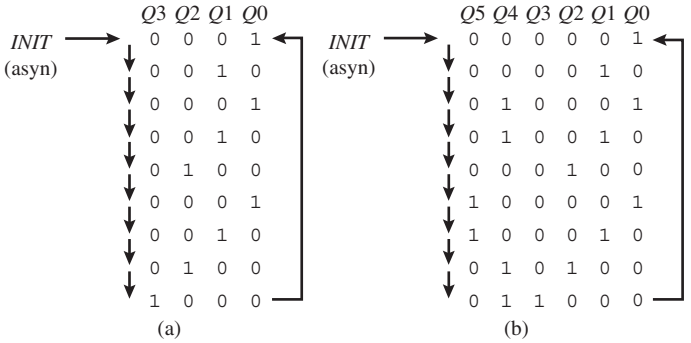


FIGURE P6.54

Section 6.7 Counter Design Using the Arithmetic Method

6.55 The arithmetic method is perhaps the most concise of the three design methods presented in this chapter. Name one advantage of using the arithmetic method. Name one limitation of using the arithmetic method. Write the use clause that must be included in the VHDL code when you use the arithmetic method.

6.56 Use the arithmetic method to write a dataflow architecture declaration with a conditional signal assignment for the binary down counter shown in Figure P6.56. The signal *SET* (asyn) is clock independent—that is, it overrides the clock.

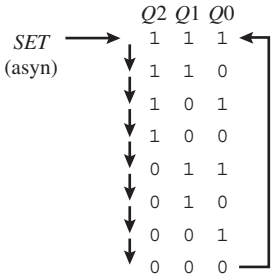


FIGURE P6.56

6.57 Write the required library clause, use clauses, and entity declaration for the counter in problem 6.56.

6.58 Combine your code for problems 6.56 and 6.57 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

6.59 Use the arithmetic method to write a behavioral architecture declaration with a process for a binary up counter that increment by 2 as shown in Figure P6.59. The signal *RST* (asyn) is clock independent—that is, it overrides the clock.

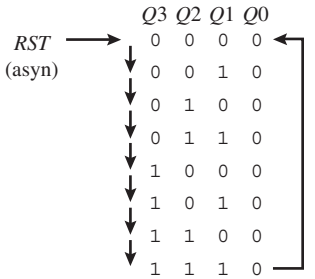


FIGURE P6.59

6.60 Write the required library clause, use clauses, and entity declaration for the counter in problem 6.59.

6.61 Combine your code for problems 6.59 and 6.60 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

Section 6.8 Frequency Division (Slowing Down a Fast Clock Frequency)

6.62 Assume that a crystal clock oscillator on a hardware board operates at 25 MHz. We would like to operate a state machine on the board at a much slower frequency. What would be the minimum number of bits or FFs for a binary down counter that could be used to slow this clock oscillator frequency down to slightly less than 1 Hz by using frequency division via a power of 2? What signal assignment must be made to signal *SLOW_CLK* to operate the state machine on the board at the slower frequency?

6.63 When using the operators + and − in statements such as A <= A + 1 and B <= B − 1 in an architecture declaration, what packages are required in the IEEE library? Write the library clause and the use clauses that specifies the require packages.

6.64 How much is the frequency of a crystal clock oscillator divided by a binary up counter with one flip-flop, two flip-flops, three flip-flops, and six flip-flops if the output is taken from the highest bit in the counter in each case?

6.65 What is the frequency division provided by a 16-bit counter at the following FF outputs that has 0 as the least significant bit: FF output 0, FF output 2, FF output 3, and FF output 9?

6.66 Explain how the arithmetic method can be used to generate an accurate division of the frequency of a crystal clock oscillator.

6.67 For a crystal clock oscillator frequency of 100 MHz, how many clock cycles must be counted for each half-

period to generate an accurate frequency of 10 Hz and also an accurate frequency of 4 Hz?

6.68 What data type is used in the text for counting clock cycles for the signal *COUNT*? Show how signal *COUNT* is declared.

6.69 What declaration can be used to specify a constant named small_value? Show how 36 is assigned to small_value.

Section 6.9 Counter Design Using the PS/NS Tabular Method

6.70 Use the PS/NS tabular method to write a behavioral architecture declaration for the counter shown in Figure P6.70. The signal *SET* (syn) is clock dependent—that is, it doesn't take effect until the clock ticks.

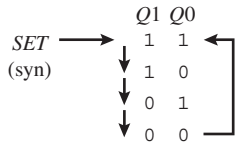


FIGURE P6.70

6.71 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter in problem 6.70.

6.72 Combine your code for problems 6.70 and 6.71 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

6.73 Use the PS/NS tabular method to write a behavioral architecture declaration for the one-hot down counter shown in Figure P6.73. The signal *INIT* (asyn) is clock independent—that is, it overrides the clock.

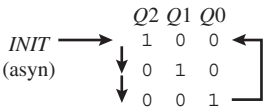


FIGURE P6.73

6.74 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter in problem 6.73.

6.75 Combine your code for problems 6.73 and 6.74 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

Section 6.10 Nonconventional Counter Design Using the PS/NS Tabular Method

6.76 A design specification for a simple nonconventional counter named NCC4 is shown in the counting sequence diagram in Figure P6.76a. One FF has been added to the counting sequence diagram to remove the repeating state value (see Figure P6.76b). Use the PS/NS tabular method to write a behavioral architecture declaration for the simple nonconventional counter using an if statement and a case statement. Use *QN* as a port signal for the output of the counter and *Q* as an internal signal. Run a simulation to verify correct functionality, and include the waveform diagram as part of your solution.

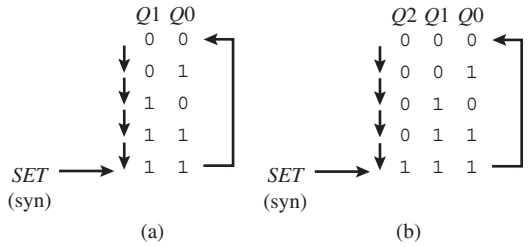


FIGURE P6.76

6.77 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter in problem 6.76.

6.78 Combine your code for problems 6.76 and 6.77 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

6.79 A design specification for a simple nonconventional counter named NCC5 is shown in the counting sequence diagram in Figure P6.79a. Two FFs have been added to the counting sequence diagram to remove repeating state values (see Figure P6.79b). Use the PS/NS tabular method to write a behavioral architecture declaration for the simple nonconventional counter using an if statement and a case statement. Use *QN* as a port signal for the output of the counter and *Q* as an internal signal. Run a simulation to verify correct functionality, and include the waveform diagram as part of your solution.

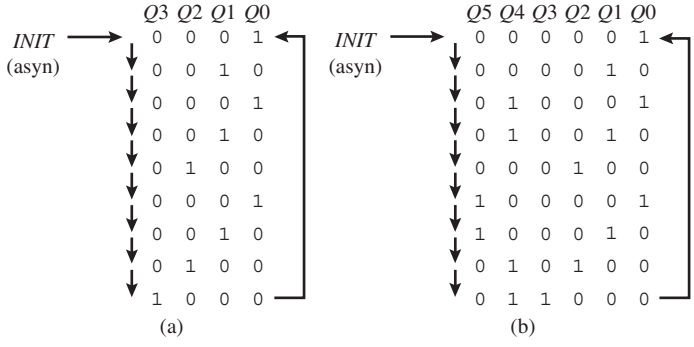


FIGURE P6.79

6.80 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter in problem 6.79.

6.81 Combine your code for problems 6.79 and 6.80 to form a complete VHDL design. Obtain a waveform diagram that shows correct functionality for the complete VHDL design.

7 Chapter

Computer Circuits

Chapter Outline

7.1 INTRODUCTION

In this chapter, you will learn about three-state outputs and the disconnected state. You are introduced to data bus sharing for a microcomputer system. XOR and XNOR functions are used to design a simple error detection system, and comparators and greater than circuits are covered via a modular design technique. Adder design is presented first in the form of the half adder and the full adder. These adders are then combined to form the ripple-carry adders and carry look-ahead adders.

7.2 THREE-STATE OUTPUTS AND THE DISCONNECTED STATE

Each of the integrated circuits (ICs) shown in Figure 7.1 has an output enable (*OE*) input that allows the ICs output signal to be **tri-stated**. This third-state output condition is a **high impedance state** in which the output acts as though it is disconnected. Circuits with three-state output are combinational logic circuits and do not have storage capability. In the positive logic system, a high output is logic 1, and a low output is logic 0. The disconnected or high impedance state is represented by the symbol Z.

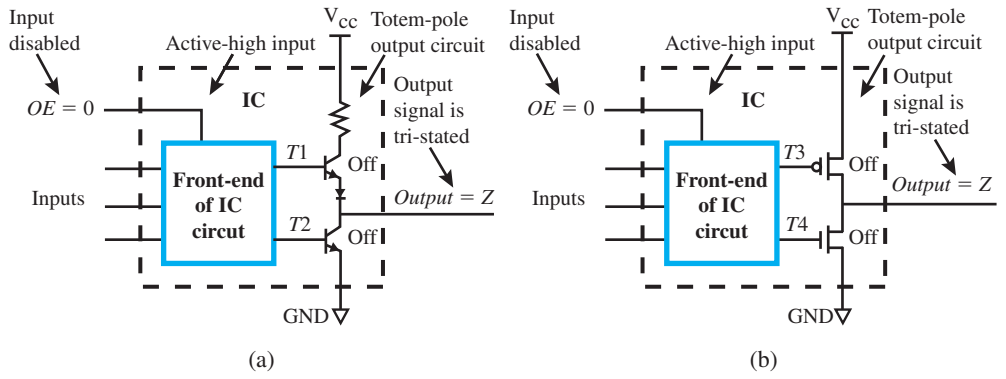


FIGURE 7.1 Condition for obtaining a tri-stated output signal with an active-high output enable input

Circuits with this type of control input are designed to act normally—that is, as 2-state output devices when the control input is active; however, when the control input is not active (inactive or disabled), the output (or outputs), becomes disconnected or floating. A device with a 3-state output is created by simply turning off both transistors in the totem-pole output as shown in Figure 7.1. Both transistors are turned off in a totem-pole output by disabling the *OE* input by pulling it low or to 0. The term Z is borrowed from circuit theory, where it is defined as impedance. When the Z value occurs, the output signal is said to be tri-stated or in its high impedance state. A device with a 3-state output has the output values high, low, and Z.

In Figure 7.1a when OE = 0, $T1 = T2 = 0$ and this shuts off both transistors and provides a Z output. In Figure 7.1b when OE = 0, $T3 = 1$ and $T4 = 0$ and this shuts off both transistors and provides a Z output. In both Figure 7.1a and b when OE = 1, the *T* signals only allow a single transistor to turn on and provide a high or low output.

Figure 7.2 shows an example of a 3-state buffer with an OE input and a 3-state output.

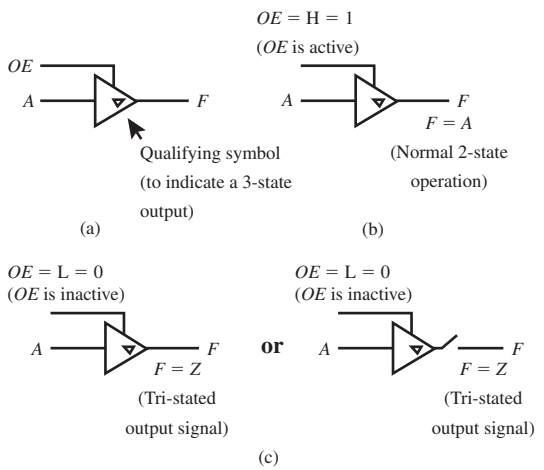


FIGURE 7.2 3-state buffer with an *OE* input: (a) logic symbol; (b) normal 2-state operation; (c) output in disconnected state or open switch representation of tri-stated output signal

Notice in Figure 7.2a that the circuit has two inputs. The normal input to the 3-state buffer is labeled signal *A* and the output is labeled signal *F*. The control input is labeled signal *OE*, which stands for output enable. When *OE* is active or goes high (or to a 1), output $F = A$ (normal 2-state operation) as shown in Figure 7.2b; however, when *OE* is inactive or goes low (or to a 0), output *F* is disconnected and is assigned the value of Z as shown in Figure 7.2c. The inverted triangle shown at the output of the 3-state buffer is the **3-state qualifying symbol** used to specify

a device with a 3-state output (or outputs). When $F = Z$, we say the output signal is tri-stated, disconnected, or floating. In VHDL, the value for a tri-stated output signal or the disconnected state is an uppercase Z.

Table 7.1 shows a compressed truth table for the 3-state buffer in Figure 7.2.

TABLE 7.1 Truth table for the 3-state buffer in Figure 7.2

OE	*F*
0	*Z*
1	*A*

When the signal *OE* is inactive or 0, the output signal is in the disconnected state (or tri-stated) and the output signal value is Z, or $F = Z$. When the signal *OE* is active or 1, the output acts normally and is equal to the input *A*, or $F = A$.

Listing 7.1 shows a complete VHDL design for the 3-state buffer shown in Figure 7.2a.

LISTING 7.1
Complete VHDL design for a 3-state buffer (project: Three_State_Buffer)

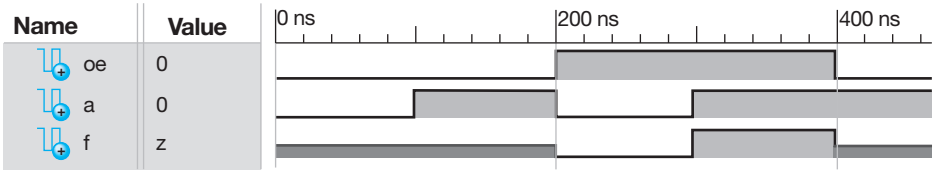
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Three_State_Buffer is port (
    oe, a : in std_logic;
    f : out std_logic
    );
end Three_State_Buffer;

architecture Dataflow of Three_State_Buffer is
begin
    f <= 'Z' when oe = '0' else
         a;
end Dataflow;
```

Things you should notice about the VHDL design in Listing 7.1:

- This design uses a dataflow design style, because a CSA is used to write the function *F*.
- When *OE* is a 0, output *F* is in the disconnected state, which must be represented by an uppercase Z in the VHDL design.
- Uppercase Z is a value—that is, it is not a signal. Signals can be made upper- or lowercase in VHDL, but values cannot be made lowercase.

Waveform 7.1 shows the simulation with the correct functionality of design entity Three_State_Buffer.

WAVEFORM 7.1
Simulation with the correct functionality of design entity Three_State_Buffer



Things you should notice about the waveforms in Waveform 7.1:

- When oe is a 0, notice that output *f* is in the disconnected state, which is represented by the Waveform diagram as a line down the middle of the waveform, which indicates the output is not a high or a 1 and is not a low or a 0.
- When oe is a 1, notice that output *f* follows input *a*—that is, when input *a* is 1—then *f* is a 1 and when *a* is a 0 *f* is a 0.

Figure 7.3a shows four 3-state buffers with 3-state outputs connected together so the signals *A*, *B*, *C*, and *D* can share a common data line.

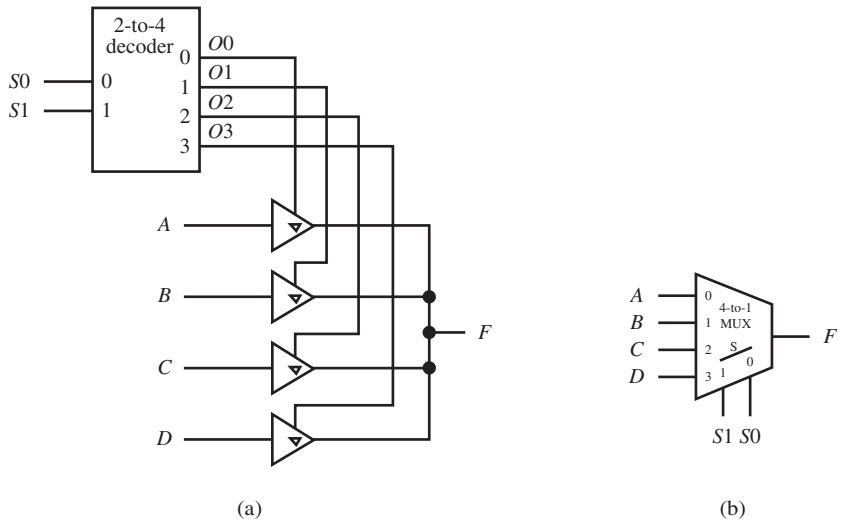


FIGURE 7.3 Sharing a common data line: (a) four 3-state buffers with 3-state outputs connected together to share a common data line; (b) a 4-to-1 MUX used to share a common data line

An alternate and somewhat simpler solution is to use a 4-to-1 MUX to share a common data line. Notice that the four buffers require a decoder to provide mutually exclusive signals to the *OE* inputs, because only one of the four buffers may be enabled at a time.

Table 7.2 shows a compressed truth table for the circuits in Figure 7.3.

S1	*S0*	*F*
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*

TABLE 7.2 Truth Table for the circuits in Figure 7.3

7.3 DATA BUS SHARING FOR A MICROCOMPUTER SYSTEM

Logic devices with 3-state outputs provide a disconnected state. Devices with disconnected states allow different signals to share common **data lines** in a **microcomputer system**. A microcomputer system is a system that contains either a microprocessor or a microcontroller. Data lines are the logic lines that are used to transport data from memories such as **random-access memory (RAM)** and **read only memory (ROM)** and from **input/output (I/O)** devices such as keyboards and monitors. Figure 7.4 shows a block diagram for a microcomputer system.

A **bus** is a set of logic lines. Many microcomputer systems are designed using three buses to move information. These buses are called the **address bus**, the **control bus**, and the **data bus**. This is illustrated in Figure 7.4 for the simple case of a data bus of 8 bits. The address bus provides the specific location for either getting data (reading data) or putting data (writing data). The control bus provides the specific operation to be performed. This might be a read (*RD*) or write (*WR*) operation. The data bus provides the specific data that is either read or written. Both the address bus and the control bus are unidirectional because they send information in only one direction. The data bus is bidirectional—that is, it operates in two directions.

In addition to selecting the specific location for getting (reading) or putting (writing) data, certain address bits are also used to decode the location of the specific device (RAM, ROM, or

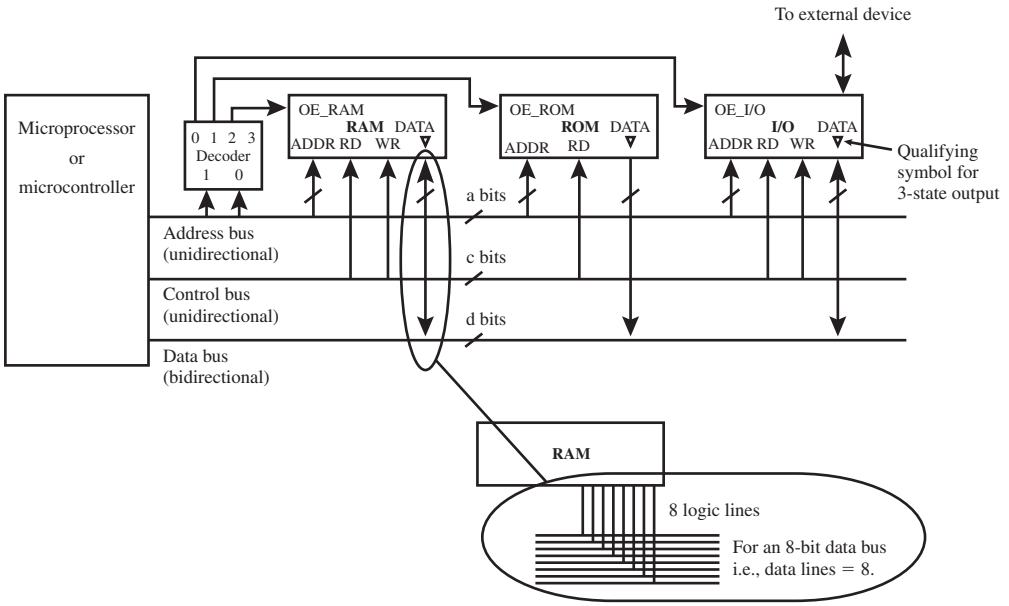


FIGURE 7.4 Block diagram for a microcomputer system

I/O) that gets to use or share the data bus. The output of the decoder shown in Figure 7.4 provides the signals *OE_RAM*, *OE_ROM*, and *OE_I/O* to enable the RAM, the ROM, or the I/O to share the data bus, respectively. *OE* stands for output enable (sometimes called chip select or *CS*). If more than one device is enabled at the same time, **logic line contention** or a **driver fight** can occur. A driver fight is another commonly used name for logic line contention. Never connect normal or totem-pole outputs together because a driver fight will occur during certain input combinations and possibly damage the devices. Do not allow this to happen. Logic line contention occurs when two or more signals are trying to drive the same logic line: one signal tries to drive the line high while another signal tries to drive the line low. The microcomputer system in Figure 7.4 uses a decoder to prevent **bus contention** of the logic lines on the data bus; otherwise, incorrect data may be read by the microcomputer due to driver fights.

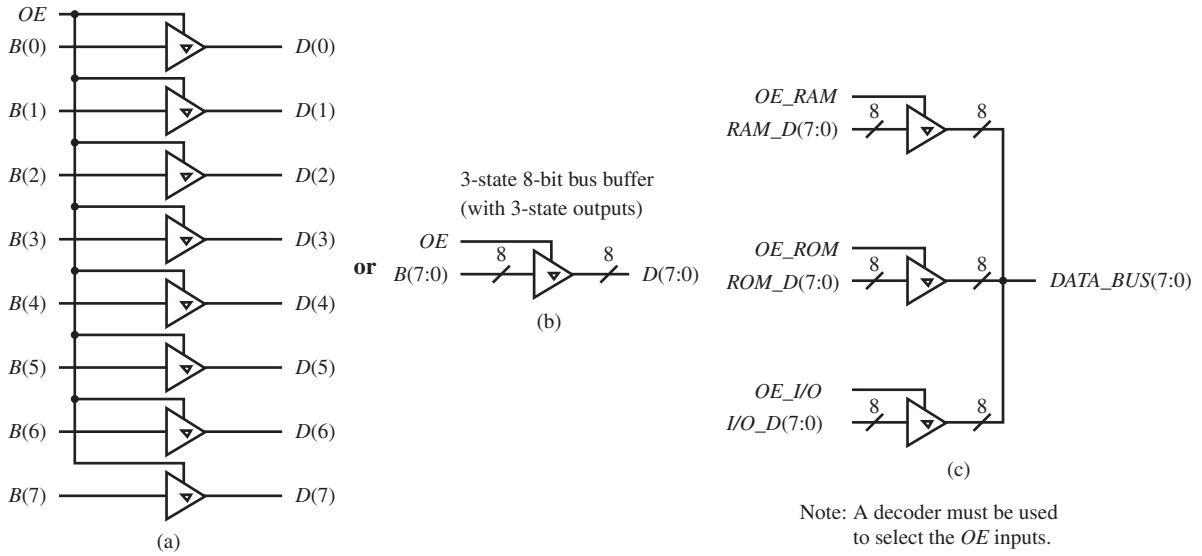


FIGURE 7.5 3-state 8-bit bus buffer with 3-state outputs: (a) schematic; (b) simplified schematic; (c) connection of three 3-state 8-bit bus buffers with 3-state outputs to provide data from three different sources to a common data bus

Using a separate data bus for the RAM, ROM and the I/O in Figure 7.4 is too complex and expensive for normal use. Devices with 3-state outputs can share a common data bus. Using a common data bus reduces design complexity. The 3-state outputs can either be built into the RAM, ROM and the I/O devices as shown in Figure 7.4, or they can be added externally via three separate 3-state bus buffers with 3-state outputs. Figure 7.5a shows the schematic diagram for a 3-state 8-bit bus buffer with 3-state outputs. Figure 7.5b shows a simplified schematic diagram for a 3-state 8-bit bus buffer with 3-state outputs. Simplified schematics are used to save time when drawing larger systems. Figure 7.5c shows three separate 3-state 8-bit bus buffers connected together to provide data from either a RAM, a ROM, or an I/O device to a common data bus.

MUXs can be used to select different data sources—that is, provide **data paths** for the RAM, ROM, or I/O in Figure 7.4—rather than using devices with 3-state outputs or using three 3-state 8-bit bus buffer circuits with 3-state outputs. To use MUXs, we simply construct an array of 4-to-1 MUXs in parallel—that is, one MUX for each bit position on the data bus—and tie the common select lines together. Figure 7.6 shows a 4-to-1 MUX array for selecting three different data sources (RAM, ROM, and I/O) for an 8-bit data bus.

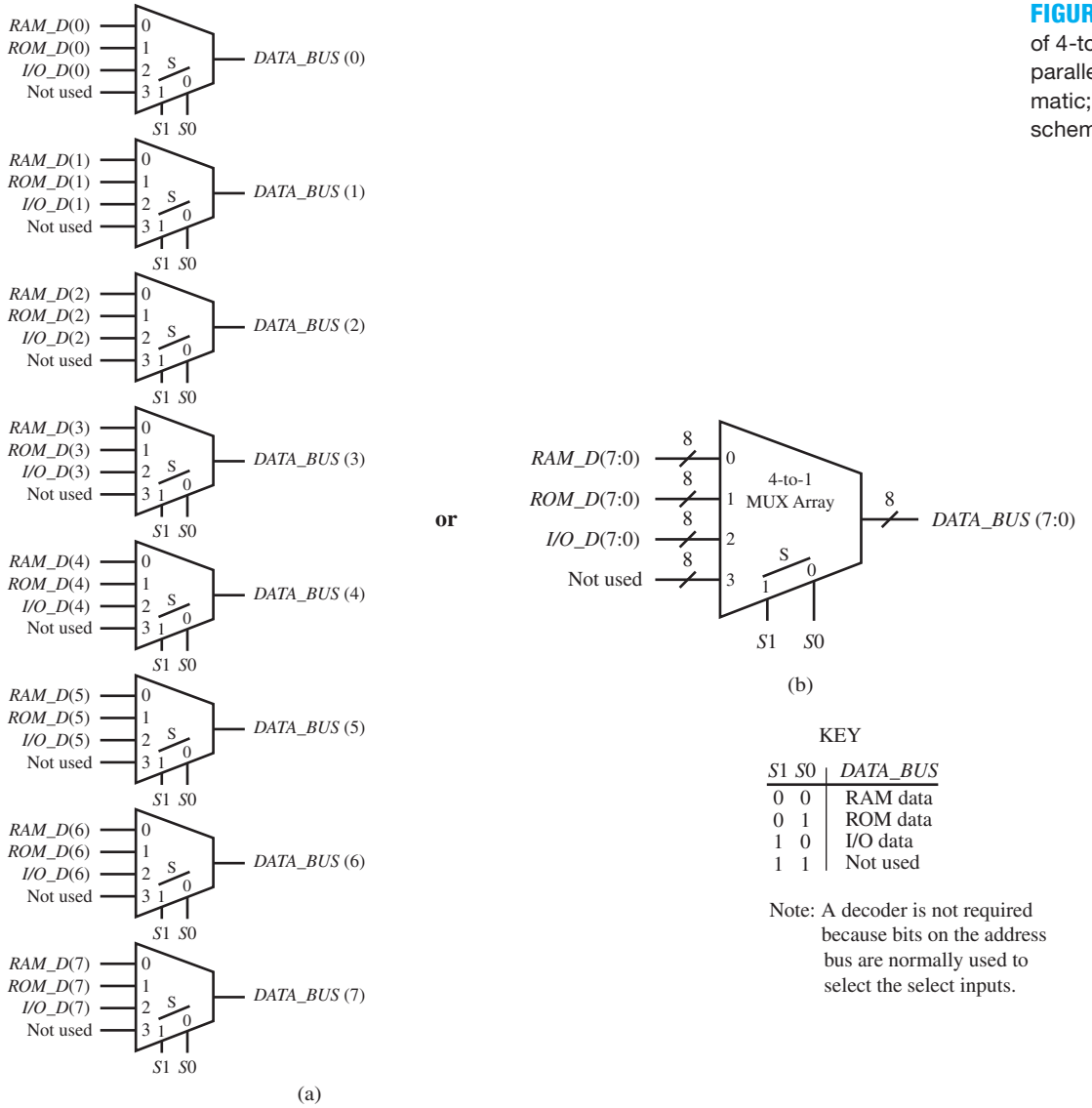


FIGURE 7.6 Array of 4-to-1 MUXs in parallel: (a) schematic; (b) simplified schematic

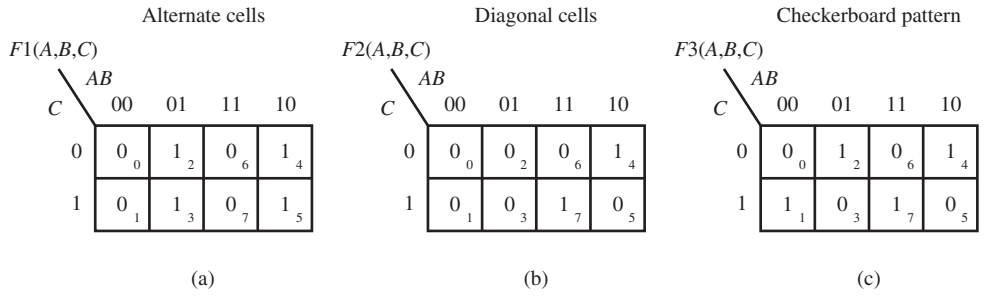
The 4-to-1 MUX array in Figure 7.6 can be used instead of using three 3-state 8-bit bus buffers with 3-state outputs shown in Figure 7.5c to provide data to the data bus of a microprocessor or microcomputer.

Either bus buffers with 3-state outputs or MUX arrays can be used to provide data paths in microcomputer systems.

7.4 MORE ABOUT XOR AND XNOR SYMBOLS AND FUNCTIONS

Alternate cells, diagonal cells, and a checkerboard pattern of 1s on a K-map are signs of XOR or XNOR functions as shown in the K-maps in K-map 7.1. We can represent such functions in terms of the XOR operator ($\oplus$).

K-MAP 7.1 XOR or XNOR functions: (a) alternate cells; (b) diagonal cells; (c) checkerboard pattern



For the K-map with the alternate cells in K-map 7.1a, the function $F1$ is

$$\begin{aligned} F1 &= \overline{A}\cdot B + A\cdot\overline{B} \\ &= A \oplus B \end{aligned}$$

For the K-map with the diagonal cells in K-map 7.1b, the function $F2$ is

$$\begin{aligned} F2 &= A\cdot\overline{B}\cdot\overline{C} + A\cdot B\cdot C \\ &= A\cdot(\overline{B}\cdot\overline{C} + B\cdot C) \\ &= A\cdot(\overline{B\oplus C}) \end{aligned}$$

For the K-map with the checkerboard pattern in K-map 7.1c, the function $F3$ is

$$\begin{aligned} F3 &= \overline{C}\cdot(A\oplus B) + C\cdot(\overline{A\oplus B}) \\ &= A\oplus B\oplus C \end{aligned}$$

In the preceding examples, the XOR operator ($\oplus$) is both **commutative**, $A\oplus B\oplus C = C\oplus A\oplus B$, and **associative**, $(A\oplus B)\oplus C = A\oplus(B\oplus C)$, just like the AND operator ($\cdot$) and the OR operator ($+$).

It can be demonstrated using perfect induction that an overbar associated with an XOR operator can be **rubber banded**—that is, stretched longer or shorter—to cover a single variable and still provide an equivalent XOR expression.

For example: $A\oplus B = \overline{A\oplus B} = \overline{A}\oplus B = A\oplus\overline{B} = \overline{A}\oplus\overline{B}$ provides an even number of overbars, and each expression represents an equivalent XOR gate symbol as shown in Figure 7.7.

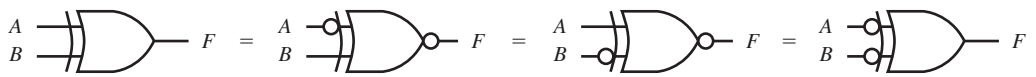


FIGURE 7.7 Equivalent XOR gates symbols

We obtain equivalent XOR gate symbols using an even number of bubbles. This may be remembered as the **XOR even number of bubbles rule**, even though XOR has an odd number of letters.

The following expressions are also equivalent: $\overline{A \oplus B} = \overline{A} \oplus B = A \oplus \overline{B} = \overline{\overline{A} \oplus \overline{B}} = \overline{A \oplus \overline{B}}$; each expression has an odd number of overbars. The equivalent XNOR gate symbol for each expression is shown in Figure 7.8.

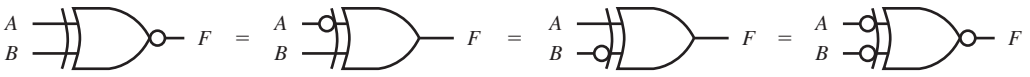


FIGURE 7.8 Equivalent XNOR gates symbols

We obtain equivalent XNOR gate symbols using an odd number of bubbles. This may be remembered as the **XNOR odd number of bubbles rule**, even though XNOR has an even number of letters.

7.4.1 Odd and Even Functions

Table 7.3 shows examples of odd functions.

TABLE 7.3 Odd functions for 2 inputs and 3 inputs

A1	*B0*	*F2odd*
0	0	0
0	1	1
1	0	1
1	1	0

A1	*B0*	*C*	*F3odd*
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

An **odd function** has a value of 1 when the input string has an odd number of 1s (1,3,5, . . .), else it has a value of 0. If the function *F2odd* is plotted in a K-map, the 1s occupy diagonal cells. The function that results is the XOR function of two variables, or $F2odd = A \oplus B$. If the function *F3odd* is plotted in a K-map, the 1s occupy cells resulting in a checkerboard pattern. The function that results is the XOR function of three variables, or $F3odd = A \oplus B \oplus C$.

Table 7.4 shows examples of even functions.

TABLE 7.4 Even functions for 2 inputs and 3 inputs

A1	*B0*	*F2even*
0	0	1
0	1	0
1	0	0
1	1	1

A1	*B0*	*C*	*F3even*
0	0	0	1
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	0

An **even function** has a value of 1 when the input string has an even number of 1s (0,2,4, . . .), else it has a value of 0. If the function *F2even* is plotted in a K-map, the 1s occupy diagonal cells.

The function that results is the XNOR function of two variables, or $F2even = \overline{A \oplus B}$. If the function *F3even* is plotted in a K-map, the 1s occupy cells resulting in a checkerboard pattern. The function that results is the XNOR function of three variables, or $F3even = \overline{A \oplus B \oplus C}$. The diagonal cells and checkerboard patterns in the K-maps for odd functions and even functions are complements of each other.

You should confirm these statements by plotting the K-maps and reading the maps. You can then use Boolean algebra to rewrite the expressions using the XOR operator ($\oplus$).

Is the function $FX = \overline{A \oplus B} \oplus C \oplus D$ an even or an odd function? Stretching the shorter overbar to cover the full length of the expression results in $FX = \overline{A \oplus B} \oplus C \oplus D = \overline{A \oplus B \oplus C \oplus D} = A \oplus B \oplus C \oplus D$, which is an odd function (i.e., XORs with **omitted** overbar). Remember that an odd function has all the input variables XORed together without a complementation, while an even function has all the input variables XORed together with one complementation.

7.4.2 Single-Bit Error Detection System

When transmitting data bits from a source to a destination, there is a need for error detection because errors can occur. Odd and even functions are used in the error detection system shown in Figure 7.9 to detect errors.

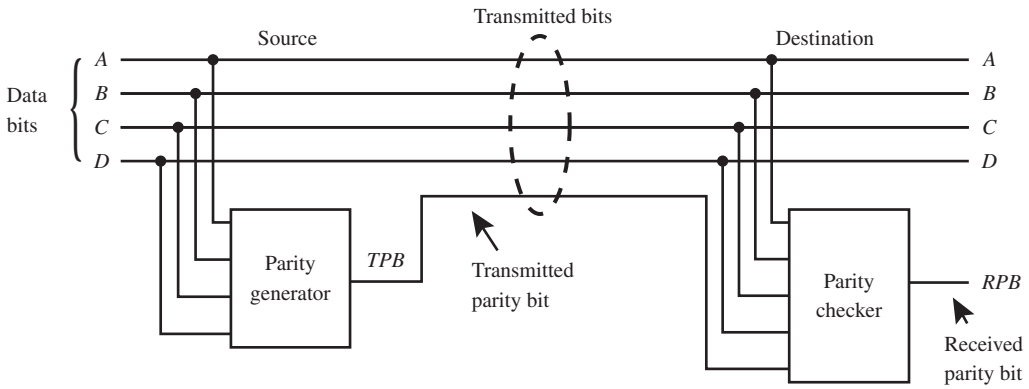


FIGURE 7.9 Block diagram for a single-bit error detection system

The error detection system shown in Figure 7.9 detects via the received parity bit (*RPB*) any single-bit errors in the transmitted bits received at the destination. The error detection system also detects any odd number of bit errors in the transmitted bits received at the destination, including the transmitted parity bit (*TPB*). Such errors, although not frequent, are due to random noise that change transmitted bits from a 1 to a 0 or from 0 to a 1. Random noise changing two transmitted bits or an even number of transmitted bits will not be detected by this system.

In Figure 7.9, the **parity generator** interrogates the data bits and provides a transmitted parity bit *TPB* that is included in the transmitted bits sent to the destination. A **parity checker** interrogates the transmitted bits received at the destination and provides a received parity bit *RPB*. If even parity is sent, then even parity should be received; otherwise, an error occurred in the system. Likewise, if odd parity is sent, then odd parity should be received; otherwise, an error occurred in the system. Using this system, when an error is detected, the data must be retransmitted.

Even parity means an even number of 1s in a given set of bits. To transmit even parity, the parity generator's output *TPB* must be an odd function, as shown in Table 7.5a in the simple case of just two data bits. To receive even parity, the parity checker's output *RPB* must also be an odd function as shown in Table 7.5b, if $RPB = 0$ is used to indicate that no single-bit error is detected. Note: Any number of data bits can be used, but we just used two data bits in Table 7.5 to keep the example simple.

TABLE 7.5 To transmit and receive even parity: (a) to transmit even parity at the source, make *TPB* an odd function; (b) to check for even parity at the destination, make *RPB* an odd function so that *RPB* = 0 indicates that no single-bit error is detected

To transmit even parity at the source (make *TPB* an odd function)

A	*B*	*TPB*	
0	0	0	Transmitted bits are *A*, *B*, and *TPB*
0	1	1	
1	0	1	
1	1	0	

(a)

To check for even parity at the destination (make *RPB* an odd function)

A	*B*	*TPB*	*RPB*	Comment
0	0	0	0	No error
0	0	1	1	Error
0	1	0	1	Error
0	1	1	0	No error
1	0	0	1	Error
1	0	1	0	No error
1	1	0	0	No error
1	1	1	1	Error

(b)

To transmit and receive even parity, with $RPB = 0$ indicating that no single-bit error is detected, output $TPB = A \oplus B$ and output $RPB = A \oplus B \oplus TPB$—that is, both are odd functions. If output $RPB = 1$, then an error occurred during transmission.

Odd parity means an odd number of 1s in a given set of bits. To transmit odd parity, the parity generator's output *TPB* must be an even function, as shown in Table 7.6a in the simple case of just two data bits. To receive odd parity, the parity checker's output *RPB* must also be an even function as shown in Table 7.6b, if $RPB = 0$ is used to indicate that no single-bit error is detected.

TABLE 7.6 To transmit and receive odd parity: (a) to transmit odd parity at the source, make *TPB* an even function; (b) to check for odd parity at the destination, make *RPB* an even function so that *RPB* = 0 indicates that no single-bit error is detected

To transmit even parity at the source (make *TPB* an even function)

A	*B*	*TPB*	
0	0	1	Transmitted bits are *A*, *B*, and *TPB*
0	1	0	
1	0	0	
1	1	1	

(a)

To check for even parity at the destination (make *RPB* an even function)

A	*B*	*TPB*	*RPB*	Comment
0	0	0	1	Error
0	0	1	0	No error
0	1	0	0	No error
0	1	1	1	Error
1	0	0	0	No error
1	0	1	1	Error
1	1	0	1	Error
1	1	1	0	No error

(b)

To transmit and receive odd parity, with $RPB = 0$ indicating that no single-bit error is detected, output $TPB = \overline{A \oplus B}$ and output $RPB = \overline{A \oplus B \oplus TPB}$—that is, both are even functions. If output $RPB = 1$, then an error occurred during transmission.

In Figure 7.9, to transmit and receive even parity for four data bits, with $RPB = 0$ indicating that no single-bit error is detected, output $TPB = A \oplus B \oplus C \oplus D$ and output $RPB = A \oplus B \oplus C \oplus D \oplus TPB$—that is, both are odd functions. To transmit and receive odd parity for four data bits, with $RPB = 0$ indicating that no single-bit error is detected, output $TPB = \overline{A \oplus B \oplus C \oplus D}$ and output $RPB = \overline{A \oplus B \oplus C \oplus D \oplus TPB}$—that is, both are even functions.

We have shown how to detect single-bit errors but not how to correct them. Hamming codes can detect and correct single-bit errors. They can also detect but not correct double-bit errors. Correction schemes for single-bit errors are not trivial. Advanced memory systems have the capability to detect and correct single-bit errors. This capability requires much more circuitry than the simple error detection system shown in Figure 7.9 that only detects single-bit errors but cannot correct them.

7.4.3 Comparators and Greater Than Circuits

An application of an XNOR gate is used to compare two bits applied to its input. If the bits are equal, then the output of the XNOR gate will be 1; else, it will be 0. **Comparators** utilize the property of XNOR gates to compare the magnitudes of binary numbers for equality. For multiple-bit operands such as operand $A = A_{n-1} \ldots A_2\, A_1\, A_0$ and operand $B = B_{n-1} \ldots B_2\, B_1\, B_0$, a comparator is a circuit that produces an output of 1 when $\boldsymbol{A} = \boldsymbol{B}$. Figure 7.10 shows a logic symbol for a 4-bit comparator.

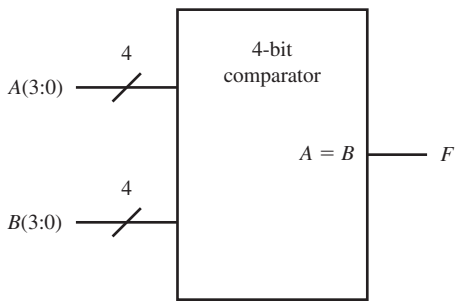


FIGURE 7.10 Logic symbol for a 4-bit comparator

In Figure 7.10, the notation $A(3{:}0)$ represents four signals lines with the signals $A_3\, A_2\, A_1\, A_0$ applied at the input. Likewise the notation $B(3{:}0)$ represents four signals lines with the signals $B_3\, B_2\, B_1\, B_0$ applied at the input.

A truth table for a 4-bit comparator with 8 inputs (4 bits for operand $\boldsymbol{A}$ and 4 bits for operand $\boldsymbol{B}$) requires $2^8 = 256$ rows. Solving the problem in this manner would be by brute force, which is difficult.

To write the function for a Comparator with two operands with multiple bits only requires a simple truth table for bit position i for a function $A_i = B_i$, as shown in Table 7.7.

TABLE 7.7 Truth Table for bit position *i* for the function $\boldsymbol{A_i = B_i}$

A_i	B_i	$A_i = B_i$
0	0	1
0	1	0
1	0	0
1	1	1

For the 4 bit comparator shown in Figure 7.10, our analysis will start with the most significant bits ($A3$ and $B3$) of the operands and proceed to the least significant bits ($A0$ and $B0$).

The only time that operands $A3\ A2\ A1\ A0$ and $B3\ B2\ B1\ B0$ are equivalent is when ($A3 = B3$) AND ($A2 = B2$) AND ($A1 = B1$) AND ($A0 = B0$). The Boolean equation for a comparator with the two operands $A3\ A2\ A1\ A0$ and $B3\ B2\ B1\ B0$ is

$$F = (\overline{A3 \oplus B3}) \cdot (\overline{A2 \oplus B2}) \cdot (\overline{A1 \oplus B1}) \cdot (\overline{A0 \oplus B0})$$

Figure 7.11 shows a circuit design for a 4-bit comparator using XNOR gates. This is a **modular design technique** (or a **divide-and-conquer technique**) that is easy to understand and can be extended to any number of bits.

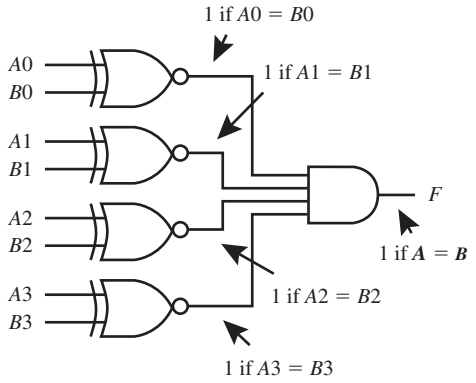


FIGURE 7.11 Circuit design for a 4-bit comparator using XNOR gates

Listing 7.2 shows a complete VHDL design for the 4-bit Comparator shown in Figure 7.11.

LISTING 7.2
Complete VHDL design for a 4-bit comparator (project: Four_bit_Comparator)

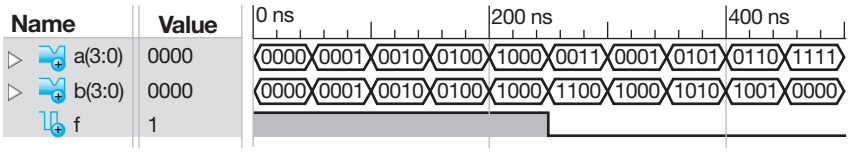
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Four_bit_Comparator is port (
   a, b : in std_logic_vector (3 downto 0);
   f : out std_logic
   );
end Four_bit_Comparator;

architecture Dataflow of Four_bit_Comparator is
begin
   f <= (a(3) xnor b(3)) and (a(2) xnor b(2)) and
        (a(1) xnor b(1)) and (a(0) xnor b(0));
end Dataflow;
```

Things you should notice about the VHDL design in Listing 7.2:

- Vector notation was used for inputs A and B to provide less typing when writing the VHDL code.
- This design uses a dataflow design style, because a Boolean equation is used to write the function F.
- Because the order of precedence of the binary operators **xnor** and **and** are the same in VHDL, parentheses must be placed around each **xnor** operation to established the required order of precedence for the Boolean equation for the function F.

Waveform 7.2 shows the simulation with the correct functionality of design entity Four_bit_Comparator.

WAVEFORM 7.2 Simulation with the correct functionality of design entity Four_bit_Comparator



Things you should notice about the waveforms in Waveform 7.2:

- Observe that for the first five random-bit patterns that were chosen, where $a = b$, output $f = 1$—as it should be for a correct design.
- Observe that for the second five random-bit patterns that were chosen, where $a \neq b$, output $f = 0$—as it should be for a correct design.
- The simulation was not an exhaustive simulation run, but it does show that the output function f is correct for the selected inputs.

The XNOR gate is also used in the design for a circuit that compares two operands to see if operand ***A*** is larger or greater than operand ***B***. A truth table for bit position i for a function $A_i > B_i$ is shown in Table 7.8.

TABLE 7.8 Truth Table for bit position *i* for the function $A_i > B_i$

A_i	B_i	$A_i > B_i$
0	0	0
0	1	0
1	0	1
1	1	0

To obtain a 4-bit greater than function with an output of 1 when $\boldsymbol{A} > \boldsymbol{B}$, we can write the function in the following format and translate it to a Boolean equation as follows:

For our analysis, start with the most significant bits of the operands and proceed to the least significant bits.

Most significant bits ($i = 3$):

$(A3 > B3)$

Next least significant bits ($i = 2$):

$(A3 = B3)$ AND $(A2 > B2)$

Next least significant bits ($i = 1$):

$(A3 = B3)$ AND $(A2 = B2)$ AND $(A1 > B1)$

Least significant bits ($i = 0$);

$(A3 = B3)$ AND $(A2 = B2)$ AND $(A1 = B1)$ AND $(A0 > B0)$

Putting the pieces together, we can write the function greater than (FGT) as

$$\begin{aligned} FGT = {} & (A3 > B3) \text{ OR} \\ & (A3 = B3) \text{ AND } (A2 > B2) \text{ OR} \\ & (A3 = B3) \text{ AND } (A2 = B2) \text{ AND } (A1 > B1) \text{ OR} \\ & (A3 = B3) \text{ AND } (A2 = B2) \text{ AND } (A1 = B1) \text{ AND } (A0 > B0) \end{aligned}$$

Using Tables 7.7 and 7.8, we can write the Boolean equation as

$$\begin{aligned} FGT = {} & (A3 \cdot \overline{B3}) + \\ & (\overline{A3 \oplus B3}) \cdot (A2 \cdot \overline{B2}) + \\ & (\overline{A3 \oplus B3}) \cdot (\overline{A2 \oplus B2}) \cdot (A1 \cdot \overline{B1}) + \\ & (\overline{A3 \oplus B3}) \cdot (\overline{A2 \oplus B2}) \cdot (\overline{A1 \oplus B1}) \cdot (A0 \cdot \overline{B0}) \end{aligned}$$

This is also a modular design technique that can be extended to any number of bits.

Figure 7.12a shows a logic symbol for the function *FGT* for 4-bit operands—that is, a 4-bit greater than circuit. A gate-level circuit design for the function *FGT* is shown in Figure 7.12b.

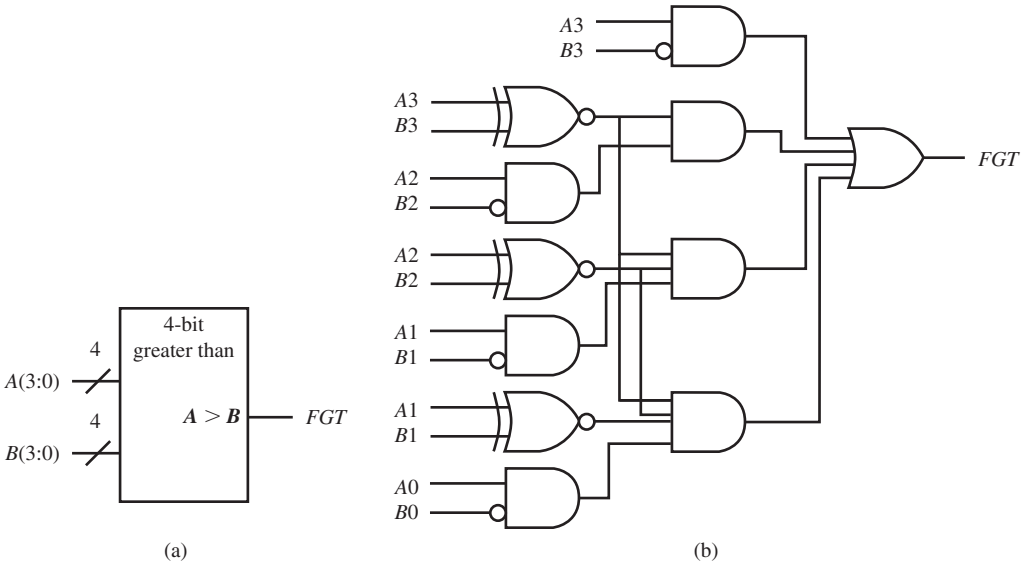


FIGURE 7.12 4-bit greater than circuit: (a) logic symbol for 4-bit operands; (b) gate-level circuit design

7.5 ADDER DESIGN

In this section, we show how to design a half adder and a full adder circuit at the gate level. The design of larger arithmetic circuits at the gate level is too tedious because of the large number of inputs required. To make the design process easier we use a **modular design technique**. We first obtain the gate-level circuits (small modules) that perform small parts of the overall design by analyzing bit slices of the design. Once the small modules are designed, they are copied (repeated or iterated) to construct larger adders and/or subtractors.

7.5.1 Designing a Half Adder Module

A **half adder (HA)** is the simplest form of adder circuit. Figure 7.13 shows a logic symbol for a half adder.

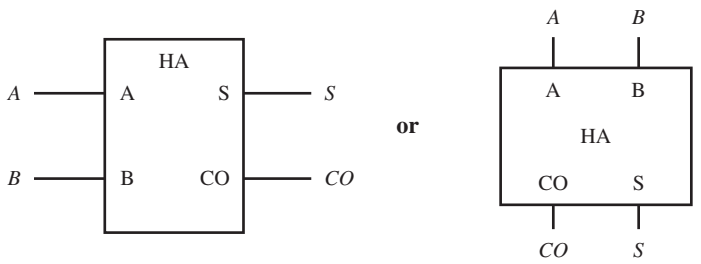


FIGURE 7.13 Logic symbol for a half adder

The half adder shown in Figure 7.13, has two operand bits A and B, a sum bit S, and a carry-out bit CO as shown by the following addition operation:

$$\begin{array}{r} A \\ +\ B \\ \hline CO\ S \end{array}$$

Table 9 shows the truth table for a half adder.

TABLE 7.9 Truth table for a half adder

A	*B*	*CO*	*S*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

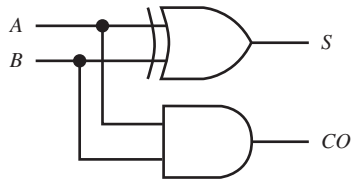
Because the truth table for the half adder in Table 7.9 contains only four rows, we can easily write the set of minimized equations for the 1s of the functions by inspection as follows:

$$S = \overline{A} \cdot B + A \cdot \overline{B} = A \oplus B$$

$$CO = A \cdot B$$

Figure 7.14 shows a gate level circuit design for a half adder.

FIGURE 7.14 Gate level circuit design for a half adder



7.5.2 Designing a Full Adder Module

A versatile circuit called a **full adder (FA)** is a single-bit slice of an n-bit adder. Consider operand ***A*** with bits $A3\ A2\ A1\ A0$ and operand ***B*** with bits $B3\ B2\ B1\ B0$. The **sum bits** are $S3\ S2\ S1\ S0$ with a **carry-out bit** $CO4$ as shown by the following addition operation for a 4-bit adder:

$$\begin{array}{rcccc} & A3 & A2 & A1 & A0 \\ + & B3 & B2 & B1 & B0 \\ \hline CO4 & S3 & S2 & S1 & S0 \end{array}$$

The truth table for a 4-bit adder with two 4-bit inputs (4 inputs for operand ***A*** and 4 inputs for operand ***B***) requires $2^8 = 256$ rows. If we tried to obtain reduced equations by hand for the sum bits $S3\ S2\ S1\ S0$ and the carry-out bit $CO4$, this would require five 8-variable K-maps. To design a 4-bit adder, we will use a modular design technique, which requires much less work.

To show the carry-out bits in the 4-bit adder, we can represent the addition operation as follows:

$$\begin{array}{rcccc} & CO3 & CO2 & CO1 & \\ & A3 & A2 & A1 & A0 \\ + & B3 & B2 & B1 & B0 \\ \hline CO4 & S3 & S2 & S1 & S0 \end{array}$$

Notice that each bit position except the 0 position requires the same addition operation, which is the operation of a full adder. For bit positions 1, 2, and 3, the carry-in bit to the next full adder is the carry-out bit from the previous adder. The operation for bit position 0 requires only a half adder. Bit positions 1, 2, and 3 require a full adder.

Figure 7.15 shows a logic symbol for a full adder.

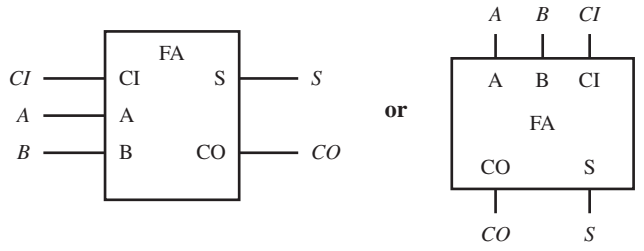


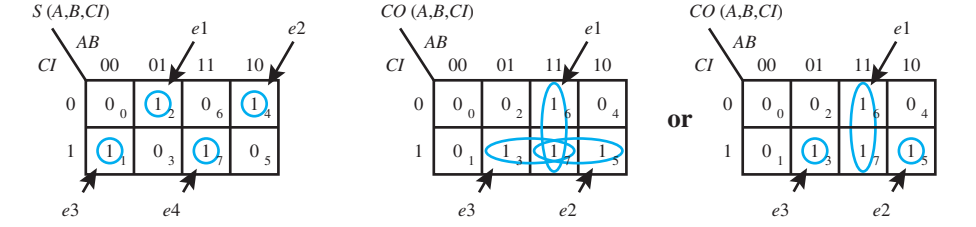
FIGURE 7.15 Logic symbol for a full adder

Notice that the full adder has a carry input (CI), which is missing in the half adder. Table 7.10 shows the truth table for a full adder.

TABLE 7.10
Truth table for a full adder

A	*B*	*CI*	*CO*	*S*
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

Minimized equations for *S* and *CO* are obtained from the K-maps shown in K-map 7.2.



$S = e1 + e2 + e3 + e4$
$= \overline{A}\cdot B\cdot\overline{CI} + A\cdot\overline{B}\cdot\overline{CI} + A\cdot B\cdot CI + \overline{A}\cdot\overline{B}\cdot CI$
$= (A \oplus B)\cdot\overline{CI} + \overline{(A \oplus B)}\cdot CI$
$= A \oplus B \oplus CI$

$CO = e1 + e2 + e3$
$= A\cdot B + A\cdot CI + B\cdot CI$

$CO = e1 + e2 + e3$
$= A\cdot B + A\cdot\overline{B}\cdot CI + \overline{A}\cdot B\cdot CI$
$= A\cdot B + CI\cdot(A \oplus B)$

K-MAP 7.2 K-maps plotted for the functions *S* and *CO* for the full adder truth table in Table 7.10

Figure 7.16 shows two different gate-level circuit designs for a full adder. The first design in Figure 7.16a uses *CO* in SOP form, while the second design in Figure 7.16b uses *CO* in factored form, with an XOR expression.

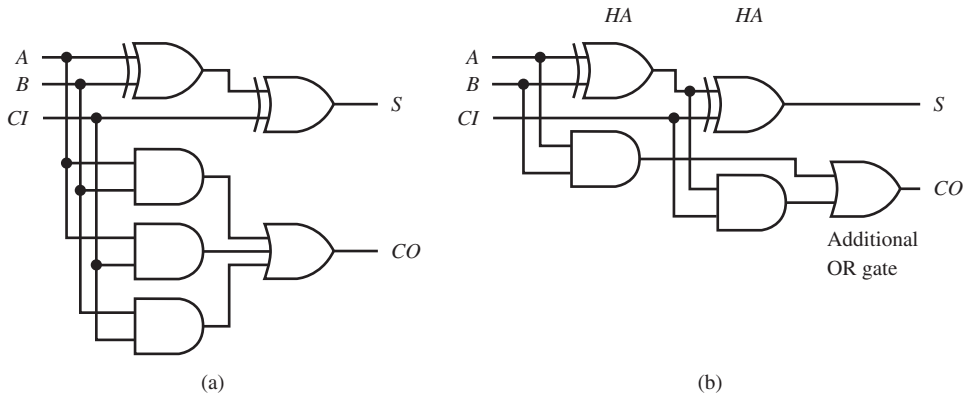


FIGURE 7.16 Gate-level circuit designs for a full adder: (a) using *CO* in SOP form; (b) using a factored form of *CO* and including an additional OR gate to form a full adder with two half adders

The factored form illustrates that a full adder may also be designed using two half adders and one additional OR gate. The design in Figure 7.16b may be a little slower than the design in Figure 7.16a because of the three gate delays from the input to the output for the *CO* circuit. To speed up the full adder circuit, it would be better to use an SOP form rather than a factored form to generate the output *CO*.

7.6 DESIGNING AND USING RIPPLE-CARRY ADDERS AND SUBTRACTORS

Figure 7.17 shows a circuit design for a 4-bit adder using a modular design technique.

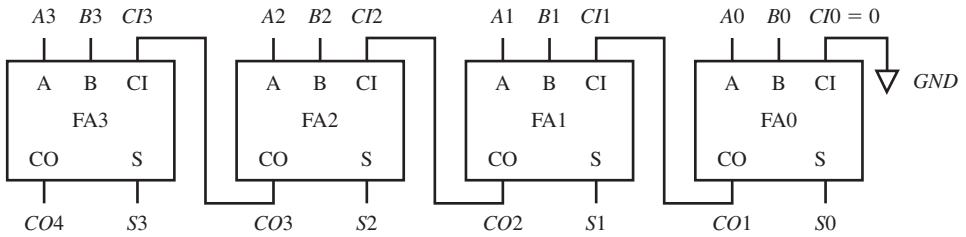


FIGURE 7.17 Circuit design for a 4-bit adder using a modular design technique

In the design in Figure 7.17, FA3 down to FA0 are all full adder modules. The carry-input signal to FA0 is tied to ground (GND) so that CI0 = 0. Notice that the carry-out signal *CO*1 from module FA0 provided as the carry-in signal *CI*1 to the module FA1. The carry-out signal from each adder module is supplied to the next adder module in the chain such that $CI_i = CO_i$ for $i = 1, 2, \ldots$.

An adder designed in this manner is called a **ripple-carry adder (RCA)**, because the carry output of the first adder module ripples through all the other adder modules via the circuit connections shown in Figure 7.17. Designing a ripple-carry adder is the easiest way to design an adder circuit with two or more bits for the input operands.

Additional full adder modules can be added to build even larger adders using this process. The carry-out signal from module FA0 must propagate through all the additional full adder modules before the result can settle to its final binary value. Ripple-carry adders are considered to be slow adders. The drawback of a ripple-carry adder is its **settling time**. Settling time is the time (propagation delay time) it takes for the result to be computed—that is, for the adder output to become stable—after the inputs are applied. **Carry look-ahead** circuitry, which we will discuss a little later, is often used to speed up the addition operation.

Subtractor circuits are not generally used in computers because subtraction can be performed with an adder circuit using **indirect subtraction by addition**. The modular logic circuit in Figure 7.18 shows a circuit design for a 2-bit adder–subtractor using indirect subtraction by addition.

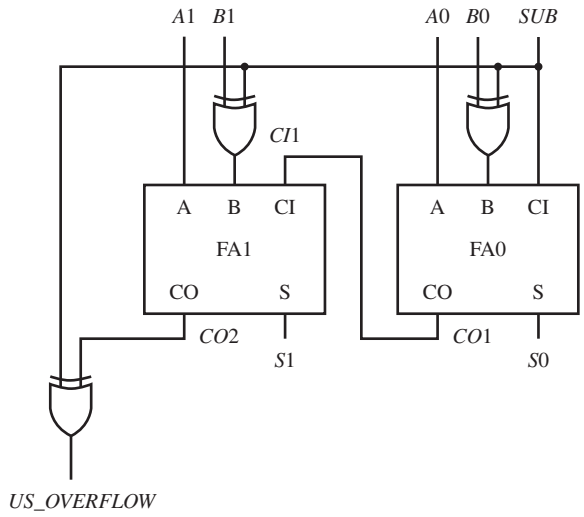


FIGURE 7.18
Circuit design for a 2-bit adder–subtractor using indirect subtraction by addition

When the signal *SUB* is 0, the adder–subtractor circuit adds the operands at its inputs providing the result $\boldsymbol{A} + \boldsymbol{B} = A1\ A0 + B1\ B0$. The two XOR gates act as Buffers when $SUB = 0$—that is, $(B0 \oplus SUB = B0 \oplus 0 = B0$ and $B1 \oplus SUB = B1 \oplus 0 = B1)$—and just pass the $\boldsymbol{B}$ operand to the adder. *US_OVERFLOW* is the unsigned overflow bit. When adding two 2-bit operands, *US_OVERFLOW* is 0 if the 2-bit result *S*1 *S*0 is correct, and *US_OVERFLOW* is 1 if the 2-bit result *S*1 *S*0 is incorrect, because the result will not fit in just 2 bits.

How does indirect subtraction work? It works using the 2's complement of an operand. The following relationship shows the 2's complement concept:

$-B =$ (2's complement of B) = (1's complement of B) $+ 1 = \overline{B} + 1$, so

$$A - B = A + (-B) = A + (\overline{B} + 1) = A + \overline{B} + 1$$

When the signal *SUB* is 1, the circuit performs indirect subtraction by addition which provides the result $\boldsymbol{A - B = A + (\overline{B} + 1) = A1\ A0 + \overline{B1}\ \overline{B0} + 1}$. The two XOR gates act as NOT gates when $SUB = 1$—that is, $(B0 \oplus 1 = \overline{B0}$ and $B1 \oplus 1 = \overline{B1})$—to provide the 1's complement of $\boldsymbol{B}$ or $\overline{\boldsymbol{B}}$ to the adder. Complementing each bit of operand $\boldsymbol{B}$ to obtain $\overline{\boldsymbol{B}}$ provides the 1's complement of $\boldsymbol{B}$. When 1 is added to $\overline{\boldsymbol{B}}$ via input CI of FA0, the result is the 2's complement of $\boldsymbol{B}$. Adding the 2's complement of $\boldsymbol{B}$ to $\boldsymbol{A}$ provides the result $\boldsymbol{A - B}$. When subtracting two 2-bit operands, $\boldsymbol{A - B}$ for this design, *US_OVERFLOW* is 0 if the 2-bit result *S*1 *S*0 is correct, and *US_OVERFLOW* is 1 if the 2-bit result *S*1 *S*0 is incorrect, because the result will not fit in just 2 bits.

The design in Figure 7.18 can be expanded to any number of bits. The limiting factor is the speed of the circuit—that is, its settling time. Carry look-ahead circuitry can be used to speed up the addition operation.

Circuits that perform an **increment operation** and **decrement operation** are important in the design of digital circuits and digital computers. In digital computers, these operations are sometimes referred to as **micro-operations**. The increment operation can be represented by the arithmetic formula $\boldsymbol{F = A + 1}$. An adder circuit with a carry-in bit to bit position 0 can be used to perform this arithmetic operation. Simply set all the $\boldsymbol{B}$ bits of the adder to 0, and set the carry-in bit at bit position 0 to a 1. An adder circuit without a carry-in bit to the bit position 0 can also

be used to perform the arithmetic operation. Simply set the ***B*** bits of the adder to 1. For a 4-bit adder, this means to set the ***B*** bits to 0001.

The decrement operation can be represented by the arithmetic formula $\boldsymbol{F = A - 1}$. To perform this operation with an adder for any word size (i.e., number of bit for **A**), we can write the decrement operation as

$\boldsymbol{F = A - 1 = A +}$ (all 1s for the word size)

For an adder with just 4 bits, -1 = 2's complement of $(1)_{10}$ = 2's complement of $(0001)_2$ = $\overline{0}\,\overline{0}\,\overline{0}\,\overline{1} + 1 = 1110 + 1 = 1111$ = all 1s for the word size of 4 bits. An adder circuit with a carry-in bit to bit position 0 can be used to perform this arithmetic operation. Simply set all the ***B*** bits of the adder to 1, and set the carry-in bit at bit position 0 to a 0. An adder circuit without a carry-in bit to the bit position 0 can also be used to perform the arithmetic operation. Simply set all the ***B*** bits of the adder to 1. For a 4-bit adder, this means to set the ***B*** bits to 1111.

A circuit that performs the increment operation ($\boldsymbol{F = A + 1}$) can be used to design a binary up counter. To do this, simply feed the outputs of the adder to a set of D flip-flops with the outputs of the D flip-flops fed back into the ***A*** bits of the adder. Figure 7.19 shows the circuit for an increment operation combined with D flip-flops for a binary up counter with 3 bits. This is a modular design technique and is used later on in the book to design the running program counter for VBC1 (Very Basic Computer 1).

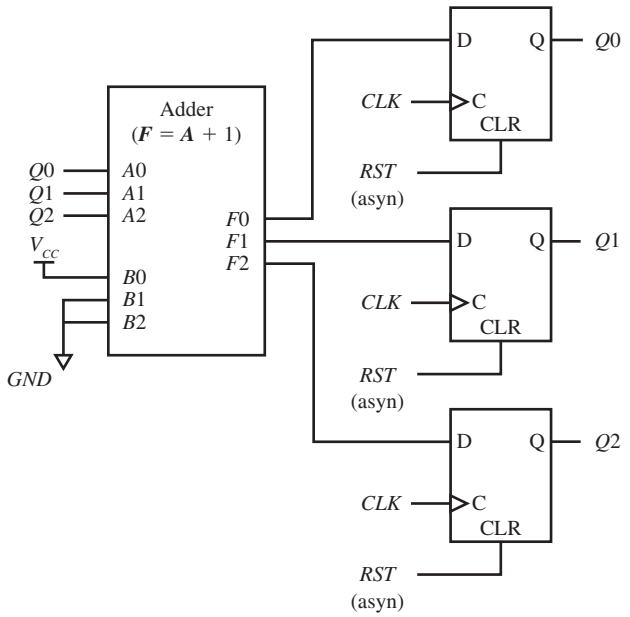


FIGURE 7.19 Circuit for an increment operation combined with D flip-flops for a binary up counter with 3 bits

Notice in Figure 7.19 that the data supplied to the ***B*** input by V_{CC} and *GND* is 001, which adds 1 to ***A***, to provide the increment operation. Also observe that the adder in Figure 7.19 does not have a carry-in bit to bit position 0. If the adder had a carry-in bit to bit position 0, then it could be set to 1, and all the bits to the ***B*** input would then be set to 0 to perform the increment operation.

Likewise, a circuit that performs the decrement operation ($\boldsymbol{F = A - 1}$) can be used to design a binary down counter by simply feeding the outputs of the adder to a set of D flip-flops with the outputs of the D flip-flops fed back into the ***A*** bits of the adder. All the bits for the ***B*** input must be connected to V_{CC}, so that the decrement operation will be performed by the adder. If the adder you are using has a carry-in bit at bit position 0, then set that bit to 0.

7.7 PROPAGATION DELAY TIME FOR RIPPLE-CARRY ADDERS

In Figure 7.20, we show a circuit design for a 4-bit ripple-carry adder in gate-level form for easy analysis. To keep things simple, assume that each gate has a propagation delay time of $1t_p$.

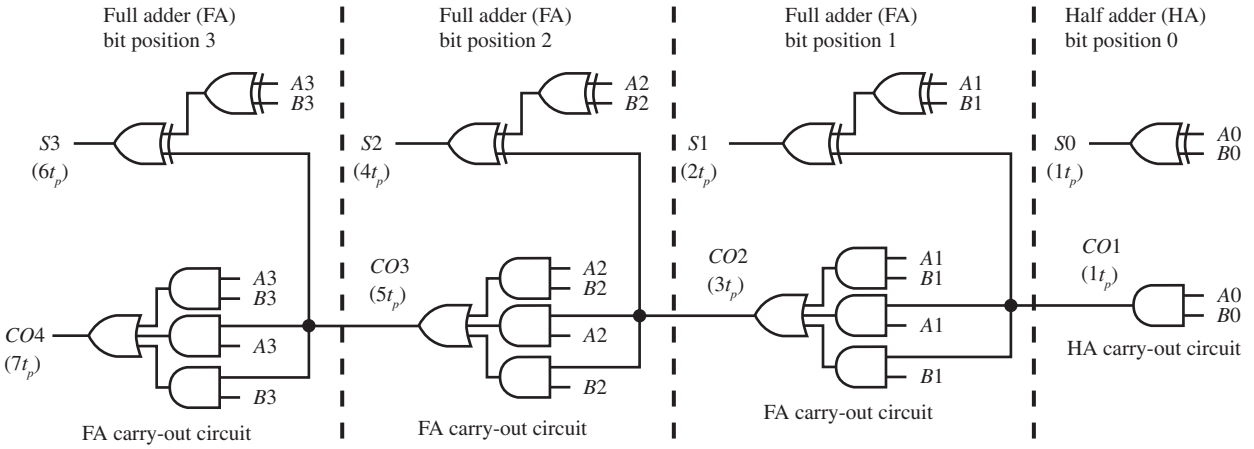


FIGURE 7.20 Circuit design for a 4-bit ripple-carry adder in gate-level form

The propagation delay times are shown in parentheses for each of the sum bits and the carry-out bits. For larger adders, the delay time gets progressively larger as more full adder modules are added to the design. For the 4-bit ripple-carry adder in Figure 7.20, the worst-case delay time is $7t_p$, which is the time it takes the carry-out bit $CO4$ to settle or become static (observe that $S3$ settles in $6t_p$, which is slightly faster). A general relationship for **settling time (ST)**—that is, the worst-case delay time—for a ripple-carry adder is $ST = 1t_p + (n - 1) \times 2t_p$, where n is the number of adder modules. In a ripple-carry adder, the settling time is a function of the accumulative propagation delay times of the carry-out circuits. Adding more full adders to make a larger adder results in a linear increase in the propagation delay time.

Modern computers are relatively fast machines. For a 64-bit ripple-carry adder like the adder in Figure 7.20, the settling time for the adder would be $ST = 1t_p + (64 - 1) \times 2t_p = 127t_p$, which is a relatively long time compared to the propagation delay time of a single gate, which we are assuming is $1t_p$. In the next section, we will show how to speed up the settling time.

The advantage of a ripple-carry adder is simply that it is easy to design. The disadvantage of a ripple-carry adder is its slow settling time, which gets worse as more full adder modules are added to make a larger adder. Remember that we assumed that each gate in the circuit in Figure 7.20 only has a gate delay of $1t_p$, which was used to simplify our calculations. In practice, the true delay time for each gate type must be used to determine the actual settling time of a ripple-carry adder. In general, the actual settling time will be worse than the simplified calculations.

7.8 DESIGNING CARRY LOOK-AHEAD ADDERS

The purpose of a **carry look-ahead adder (CLAA)** is to speed up the adding process by adding more gates in each carry-out circuit to decrease the settling time. This is done by simply generating each individual carry-out circuit independent of all previous carry-out outputs except the first one ($CO1$), which speeds up the addition process. The only place each carry-out output is used is for the carry-in input to the sum circuitry of the next modified full adder.

A carry look-ahead adder contains **carry generate terms** (terms of the form $G_i = A_i \cdot B_i$) and **carry propagate terms** (terms of the form $P_i = A_i + B_i$).

Starting with a half adder, we can write the carry-out CO_{i+1} as follows:

$CO_{i+1} = A_i \cdot B_i = G_i$, or $CO1 = G0$

Plotting the carry-out bit CO_{i+1} for a full adder in a 3-variable K-map and minimizing the function, we can write:

$CO_{i+1} = A_i \cdot B_i + CI_i \cdot A_i + CI_i \cdot B_i = A_i \cdot B_i + CI_i \cdot (A_i + B_i)$

The carry-out bit CO_{i+1} contains one carry generate term $A_i \cdot B_i$ or G_i and one carry propagate term $A_i + B_i$ or P_i, so, CO_{i+1} can be expressed as $CO_{i+1} = G_i + CI_i \cdot P_i$ for each full adder.

Because the carry-in bit of the next full adder module is supplied by the carry-out bit of the previous full adder module, $CI_i = CO_i$ and we can write $CO_{i+1} = G_i + CO_i \cdot P_i$ for each full adder.

The carry-out bits for a carry look-ahead adder can be written as shown in Figure 7.21 for each bit position.

FIGURE 7.21 The carry-out bits for a carry look-ahead adder

Starting with a half adder, we can write:

Bit position 0:
$CO1 = G0$ — Carry-out bit for a half adder

Bit position 1:
$CO2 = G1 + CO1 \cdot P1$ — Substitute previous result for $CO1$
$= G1 + G0 \cdot P1$

Bit position 2:
$CO3 = G2 + CO2 \cdot P2$ — Substitute previous result for $CO2$
$= G2 + (G1 + G0 \cdot P1) \cdot P2$
$= G2 + G1 \cdot P2 + G0 \cdot P1 \cdot P2$

Bit position 3:
$CO4 = G3 + CO3 \cdot P3$ — Substitute previous result for $CO3$
$= G3 + (G2 + G1 \cdot P2 + G0 \cdot P1 \cdot P2) \cdot P3$
$= G3 + G2 \cdot P3 + G1 \cdot P2 \cdot P3 + G0 \cdot P1 \cdot P2 \cdot P3$

Bit position 4:
$CO5 = G4 + CO4 \cdot P4$ — Substitute previous result for $CO4$
$= G4 + (G3 + G2 \cdot P3 + G1 \cdot P2 \cdot P3 + G0 \cdot P1 \cdot P2 \cdot P3) \cdot P4$
$= G4 + G3 \cdot P4 + G2 \cdot P3 \cdot P4 + G1 \cdot P2 \cdot P3 \cdot P4 + G0 \cdot P1 \cdot P2 \cdot P3 \cdot P4$

Bit position i : for $i = 0, 1, 2 \ldots$. This is the general expression for the function CO_{i+1}
$CO_{i+1} = G_i + G_{i-1} \cdot P_i + G_{i-2} \cdot P_{i-1} \cdot P_i + G_{i-3} \cdot P_{i-2} \cdot P_{i-1} \cdot P_i + \ldots$

When you write CO_{i+1} with the general expression, continue with the pattern until the carry generate term in the last product term is $G0$. Observe for bit position 2 that the last product term for $CO_{i+1} = CO3$ would be $G0 \cdot P1 \cdot P2$. Also observe for bit position 6 that the last product term for $CO_{i+1} = CO7$ would be $G0 \cdot P1 \cdot P2 \cdot P3 \cdot P4 \cdot P5 \cdot P6$. Once you understand the pattern, you can write the all the terms for any of the carry-out bits of a carry look-ahead adder in a straightforward manner.

In Figure 7.22, we show a circuit design for a 4-bit carry look-ahead adder in gate-level form. To observe the carry generate and carry propagate terms in the 4-bit carry look-ahead adder design, look closely at the modified full adder carry-out circuits. Remember that each individual carry-out circuit is independent of all previous carry-out outputs except $CO1$.

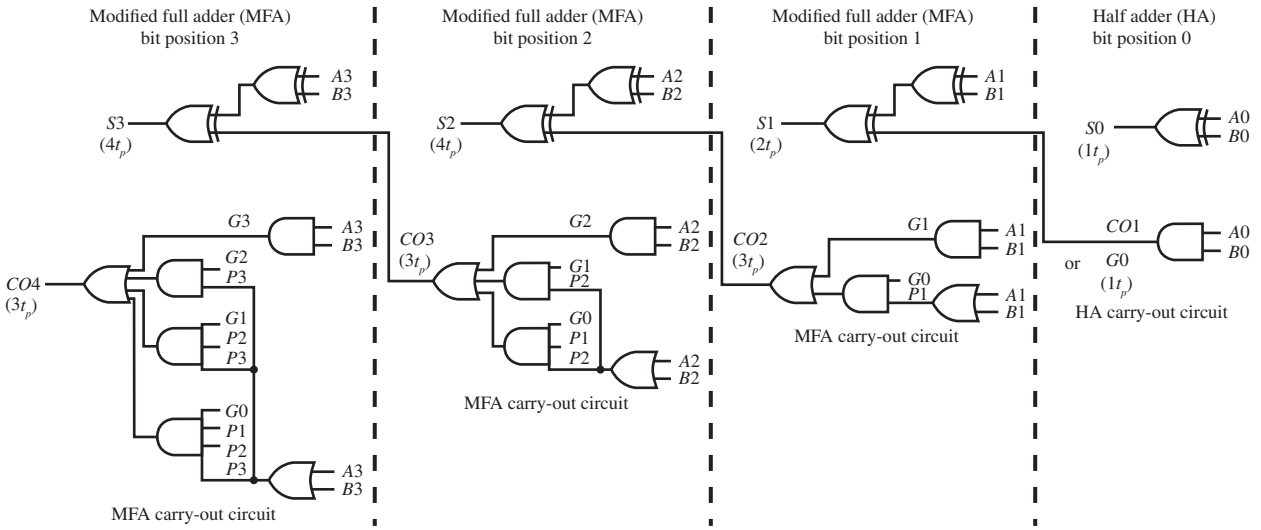


FIGURE 7.22 Circuit design for a 4-bit carry look-ahead adder in gate-level form

Figure 7.23 shows the circuit for the modified full adder module for bit position 4 for a 5-bit carry look-ahead adder.

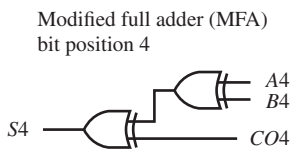
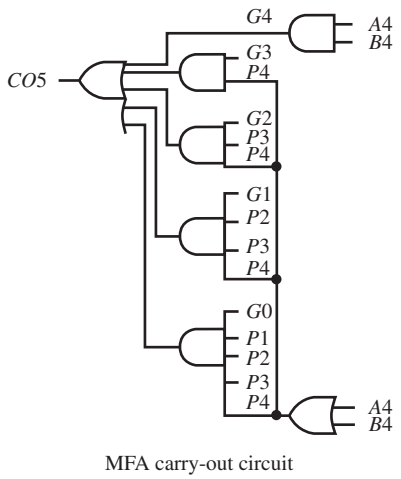


FIGURE 7.23 Circuit for the modified full adder module for bit position 4 for a 5-bit carry look-ahead adder



The limitation of using carry-look ahead for adders with many bits is the **fan-in requirements** of the gates—that is, the number of gate inputs—and the **fan-out requirements** of the gates—that is, the number of gates an output can electrically drive. If cascading gates must be

added to meet the fan-in requirements or buffers must be added to meet the fan-out requirements for the design, then the settling time for the circuit will be increased, thus slowing down the addition operation.

7.9 PROPAGATION DELAY TIME FOR CARRY LOOK-AHEAD ADDERS

To keep things simple, in Figure 7.22 we assumed that each gate has a propagation delay time of $1t_p$. The propagation delay times shown in parentheses for each of the sum bits and the carry-out bits for the carry look-ahead adder are substantially smaller than that of a ripple-carry adder when the number of modified full adder (MFAs) is increased. Also observe that each modified full adder module is slightly different because of the additional carry-out circuitry.

For the 4-bit carry look-ahead adder in Figure 7.22, the worst-case delay time is $4t_p$, which is the time it takes the sum bit $S3$ to settle or become static (observe that $CO4$ settles in $3t_p$, which is slightly faster). The propagation delay time for each carry-out circuit in a carry look-ahead adder is the same (always $3t_p$) at the expense of additional gates for the carry-out circuitry. Carry look-ahead adders are mainly used for adders with a large number of bits. The settling time for the circuit design in Figure 7.22 is $ST = 3t_p + 1t_p = 4t_p$ for three or more bits.

Assuming there were no fan-in and fan-out requirement limitations, a 64-bit carry look-ahead adder like the adder in Figure 7.22 would have a settling time of $ST = 4t_p$, which is a great speed improvement over the settling time of the ripple-carry adder in Figure 7.20, which was $ST = 127t_p$. Don't forget that we assumed that each gate in the circuits in Figures 7.20 and 7.22 only has a gate delay of $1t_p$. In practice, the true delay time for each gate type must be used to determine the actual settling time.

PROBLEMS

Section 7.2 Three-State Outputs and the Disconnected State

7.1 What is the output signal value of a 3-state output circuit that is tri-stated?

7.2 What value is used in VHDL to represent a tri-stated output signal?

7.3 In Figure P7.3a, what is the input signal value that must be applied to the output enable input of the 3-state output circuit when the output signal of the circuit is not being tri-stated? Answer the same question for Figure P7.3b.

7.4 What values are available at the output of a 3-state output circuit when the input signal that is applied to the output enable input is active?

7.5 From a circuit point of view, what does a tri-stated output signal act like?

Section 7.3 Data Bus Sharing for a Microcomputer System

7.6 What is the main purpose of using logic devices with 3-state outputs?

7.7 Three-state bus buffers with 3-state outputs can be connected together to share a common data bus. What is another way that a common data bus can be shared?

7.8 When using 3-state bus buffers with 3-state outputs to share a common data bus with different devices, is a decoder required to enable the *OE* inputs of the bus buffers?

7.9 When using a MUX array to share a common data bus with different devices, is a decoder required to enable the select inputs of the MUX array?

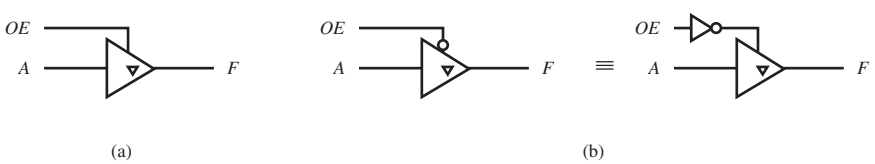


FIGURE P7.3

Section 7.4 More about XOR and XNOR Symbols and Functions

7.10 What are the signs of XOR or XNOR functions plotted on a K-map?

7.11 An XOR gate can be drawn using a different number of bubbles. What is a simple rule for remembering how to draw equivalent XOR gate symbols?

7.12 An XNOR gate can be drawn using a different number of bubbles. What is a simple rule for remembering how to draw equivalent XNOR gate symbols?

7.13 In Figure P7.13, is the logic symbol an XOR gate or an XNOR gate? Write the Boolean function F for the gate.

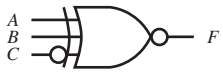


FIGURE P7.13

7.14 In Figure P7.14, is the logic symbol an XOR gate or an XNOR gate? Write the Boolean function F for the gate.

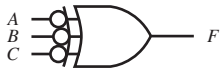


FIGURE P7.14

7.15 An odd function has a value of 1 when the input string has an odd number of 1s. Which gate type (a 2-input XOR gate or a 2-input XNOR gate) is an odd function? Prove your answer by writing the truth table for the gate type you select, and show that the input string has an odd number of 1s when the value of the function is 1.

7.16 An even function has a value of 1 when the input string has an even number of 1s. Which gate type (a 2-input XOR gate or a 2-input XNOR gate) is an even function? Prove your answer by writing the truth table for the gate type you select, and show that the input string has an even number of 1s when the value of the function is 1.

7.17 Obtain the output functions for a parity generator and a parity checker that can be used to send two data bits to a destination using the single-bit error detection system illustrated in the textbook to transmit and receive odd parity. Assume that A and B are the data bits, TPB is the output of the parity generator, and RPB is the output of the parity checker. Also assume that $RPB = 0$ indicates that no single-bit error is detected.

7.18 Obtain the output functions for a parity generator and a parity checker that can be used to send three data bits to a destination using the single-bit error detection system illustrated in the textbook to transmit and receive even parity. Assume that A, B, and C are the data bits, TPB is the output of the parity generator, and RPB is the output of the parity checker. Also assume that $RPB = 0$ indicates that no single-bit error is detected.

7.19 Obtain the output functions for a parity generator and a parity checker that can be used to send eight data bits to a destination using the single-bit error detection system illustrated in the textbook to transmit and receive odd parity. Assume that $D7$, $D6$, $D5$, $D4$, $D3$, $D2$, $D1$, and $D0$ are the data bits, TPB is the output of the parity generator, and RPB is the output of the parity checker. Also assume that $RPB = 0$ indicates that no single-bit error is detected.

7.20 Obtain the output function for a comparator with an output F that will detect the equality of two operands that contain 3 bits each—that is, $F = 1$ when $\boldsymbol{A} = \boldsymbol{B}$. Assume that $A2$, $A1$, and $A0$ are the operand bits for one operand and $B2$, $B1$, and $B0$ are the operand bits for the second operand. Use the modular design technique presented in the textbook. Draw and label the gate-level circuit.

7.21 Obtain the output function for a Comparator with an output F that will detect the equality of two operands that contain 5 bits each—that is, $F = 1$ when $\boldsymbol{A} = \boldsymbol{B}$. Assume that $A(4:0)$ are the operand bits for one operand and $B(4:0)$ are the operand bits for the second operand. Use the modular design technique presented in the textbook. Draw and label the gate-level circuit.

7.22 Obtain the output function for a greater than circuit with an output F that will detect the inequality of two operands that contain 2 bits each—that is, $F = 1$ when $\boldsymbol{A} > \boldsymbol{B}$. Assume that $A1$ and $A0$ are the operand bits for one operand and $B1$ and $B0$ are the operand bits for the second operand. Use the modular design technique presented in the textbook. Draw and label the gate-level circuit.

7.23 Obtain the output function for a less than circuit with an output F that will detect the inequality of two operands that contain 4 bits each—that is, $F = 1$ when $\boldsymbol{A} < \boldsymbol{B}$. Assume that $A3$, $A2$, $A1$, and $A0$ are the operand bits for one operand and $B3$, $B2$, $B1$, and $B0$ are the operand bits for the second operand. Use the modular design technique presented in the textbook. Draw and label the gate-level circuit.

Section 7.5 Adder Design

7.24 Does a half adder have a carry input signal?

7.25 Where is a half adder used in a larger adder?

7.26 Write the truth table for a half adder with inputs $X0$ and $Y0$ and outputs $S0$ and CO. Write the simplest Boolean functions for $S0$ and CO.

7.27 Write the truth table for a full adder with inputs A, B, and CI and outputs S and CO. Obtain the circuit for a full adder for function S and CO using MUX designs for each output.

7.28 Where is a full adder used in a larger adder?

Section 7.6 Designing and Using Ripple-Carry Adders and Subtractors

7.29 What is the easiest way to design an adder circuit with two or more bits for the input operands?

7.30 How are full adders connected together to form a ripple-carry adder?

7.31 How is the carry-in signal handled in a 4-bit ripple-carry adder that uses all full adders in the chain?

7.32 Use the modular design technique to design a 2-bit ripple-carry adder using just full adders. Explain how one of the output signals can be used to indicate that an unsigned overflow has occurred in the addition process.

7.33 What is the drawback of a ripple-carry adder? What can be used to speed up the addition operation?

7.34 How can subtraction be performed with an adder? Provide a formula for doing subtraction with an adder.

7.35 Use the modular design technique to design a 3-bit ripple carry adder–subtractor that uses indirect subtraction by addition. Make the design with full adders and XOR gates. Add an XOR gate to indicate when an unsigned overflow has occurred.

7.36 In the adder–subtractor circuit in the book, what does the *US_OVERFLOW* output indicate?

7.37 In the expression $-\boldsymbol{OP} = +(\overline{\boldsymbol{OP}} + 1)$, what does $\overline{\boldsymbol{OP}}$ represent and what does $\overline{\boldsymbol{OP}} + 1$ represent? How is the expression used to perform subtraction? Show that this is true for performing indirect subtraction by addition $\boldsymbol{A} - \boldsymbol{B}$ for 4 bits, if $\boldsymbol{A} = 7$ (0111 in binary) and $\boldsymbol{B} = 3$ (0011 in binary).

7.38 Use the modular design technique to design a logic circuit that will increment any 3-bit binary number applied to its input. Let the input to the circuit be *A*2 *A*1 *A*0 and the output be *R*2 *R*1 *R*0. Use only full adders. Example: Apply 5 (or 101) and the result is 6 (or 110).

7.39 Use the modular design technique to design a logic circuit that will decrement any 3-bit binary number applied to its input. Let the input to the circuit be *A*2 *A*1 *A*0 and the output be *R*2 *R*1 *R*0. Use only full adders. Example: Apply 5 (or 101) and the result is 4 (or 100).

7.40 Use the modular design technique to design a binary up counter with 4 bits. Use the circuit for an increment operation combined with D flip-flops. Use D flip-flops with a CLR input so the binary up counter can be reset at any time. Use an adder that has a carry-in input to its least significant bit position *CI*0.

7.41 Use the modular design technique to design a binary down counter with 3 bits. Use the circuit for a decrement operation combined with D flip-flops. Use D flip-flops with a CLR input so the binary down counter can be reset at any time. Use an adder that has a carry-in input to its least significant bit position *CI*0.

Section 7.7 Propagation Delay Time for Ripple-Carry Adders

7.42 In a ripple-carry adder, what is the worst-case delay time for the HA carry-out circuit, assuming all gates have a propagation delay time of $1t_p$?

7.43 In a ripple-carry adder, what is the worst-case delay time for each FA carry-out circuit, assuming all gates have a propagation delay time of $1t_p$?

7.44 Write the relationship for the settling time (ST) for a ripple-carry adder circuit, assuming all gates have a propagation delay time of $1t_p$ and n is the number of adder modules.

7.45 What is the worst-case delay time for a 3-bit ripple-carry adder, assuming all gates have a propagation delay time of $1t_p$?

7.46 What is the worst-case delay time for a 7-bit ripple-carry adder, assuming all gates have a propagation delay time of $1t_p$?

7.47 What is the worst-case delay time for a 32-bit ripple-carry adder, assuming all gates have a propagation delay time of $1t_p$?

7.48 Which bit in a 10-bit ripple-carry adder settles first or becomes static—the carry-out bit *CO*11 or the sum bit *S*10, assuming all gates have a propagation delay time of $1t_p$?

7.49 Which bit in a 12-bit ripple-carry adder settles last or becomes static—the carry-out bit *CO*13 or the sum bit *S*12, assuming all gates have a propagation delay time of $1t_p$?

7.50 List the advantage and the disadvantage of a ripple-carry adder as provided in the book.

Section 7.8 Designing Carry Look-Ahead Adders

7.51 What is the purpose of a carry look-ahead adder?

7.52 How does a carry look-ahead adder decrease the settling time of each individual carry-out circuit?

7.53 List all the carry generate terms for a 3-bit carry look-ahead adder with inputs *A*2, *A*1, *A*0 and *B*3, *B*2, *B*0.

7.54 List all the carry propagate terms for a 4-bit carry look-ahead adder with inputs *A*3, *A*2, *A*1, *A*0 and *B*4, *B*3, *B*2, *B*0.

7.55 Express the carry-output bit CO_{i+1} or *CO*1 for a half adder in a carry look-ahead adder in terms of its carry generated term G_i or *G*0.

7.56 Express the carry-output bit CO_{i+1} for each full adder in a carry look-ahead adder in terms of its carry generate term G_i and its carry propagate term P_i.

7.57 Write the carry-out bit *CO*3 (for bit position 2) for a 3-bit carry look-ahead adder in terms of its carry generate terms (*G* terms) and its carry propagate terms (*P* terms).

7.58 Show the circuit for the modified full adder module for bit position 2 for a 3-bit carry look-ahead adder.

7.59 Write the carry-out bit *CO*6 (for bit position 5) for a 6-bit carry look-ahead adder in terms of its carry generate terms (*G* terms) and its carry propagate terms (*P* terms).

7.60 Show the circuit for the modified full adder module for bit position 5 for a 6-bit carry look-ahead adder.

7.61 Write the carry-out bit *CO*7 (for bit position 6) for a 7-bit carry look-ahead adder in terms of its carry generate terms (*G* terms) and its carry propagate terms (*P* terms).

7.62 Show the circuit for the modified full adder module for bit position 6 for a 7-bit carry look-ahead adder.

Section 7.9 Propagation Delay Time for Carry Look-Ahead Adders

7.63 In a carry look-ahead adder, what is the worst-case delay time for the HA carry-out circuit, assuming all gates have a propagation delay time of $1t_p$?

7.64 In a carry look-ahead adder, what is the worst-case delay time for each MFA carry-out circuit, assuming all gates have a propagation delay time of $1t_p$?

7.65 Which bit in a 8-bit carry-look-ahead adder settles first or becomes static—the carry-out bit $CO9$ or the sum bit $S8$, assuming all gates have a propagation delay time of $1t_p$?

7.66 Which bit in a 20-bit carry-look-ahead adder settles last or becomes static—the carry-out bit $CO21$ or the sum bit $S20$, assuming all gates have a propagation delay time of $1t_p$?

7.67 For a 9-bit carry look-ahead adder with no fan-in and fan-out requirement limitations, what would be the worst-case delay time, assuming all gates have a propagation delay time of $1t_p$?

7.68 For a 16-bit carry look-ahead adder with no fan-in and fan-out requirement limitations, what would be the worst-case delay time, assuming all gates have a propagation delay time of $1t_p$?

8 Chapter

Circuit Implementation Techniques

Chapter Outline

8.1 INTRODUCTION

In this chapter, you will learn about the following programmable logic devices: PROMs, PLAs, PALs and GALs, and LUTs. You are introduced to the direct polarity indication system for specifying signal name and equivalent circuits. A simple MUX/DMUX circuit is shown for a data transmission scheme. The design of larger MUX and DMUX circuits, called trees, is presented.

8.2 PROGRAMMABLE LOGIC DEVICES

Industry only wired up so many circuits in laboratory situations before it developed smarter ways to implement designs with logic gates. It is not known for sure who invented the **PROM (programmable read only memory)**. The PROM was introduced back in the early 1960s. The **PLA (programmable logic array)** was invented at Signetics in 1975. PLA is a registered trademark of Signetics. The **PAL (programmable array logic)** was invented in 1978 by John Birkner at MMI (Monolithic Memories Inc.). Another name for a PAL is a **GAL (generic array logic)**. Both acronyms PAL and GAL are registered trademarks of Lattice Semiconductor Corporation. The **FPGA (field programmable gate array)**, which contains **LUTs (look-up tables)**, was invented at Xilinx in 1985. The PROM, PLA, PAL or GAL, and LUTs in FPGA are all **PLDs (programmable logic devices)**. Larger versions of PLA, PAL, or GAL architectures are referred to as **CPLDs (complex programmable logic devices)**. Programmable logic devices primarily use the AND/OR gate form discussed earlier. Each of these devices can implement multiple functions. The PROM, LUT, and PLA each allow product-term expression sharing; however, the PAL or GAL do not allow product-term expression sharing.

You can implement any function with these architectures provided the architectures have enough gates. Figure 8.1 illustrates a template or generalized format for PLDs.

When PLDs were first introduced they only had a few gates—that is, a 100 or so. Small functions would fit into the architecture, while larger functions with more inputs or more product terms would not fit. Designers had to be aware of the size of the PLDs they elected to use

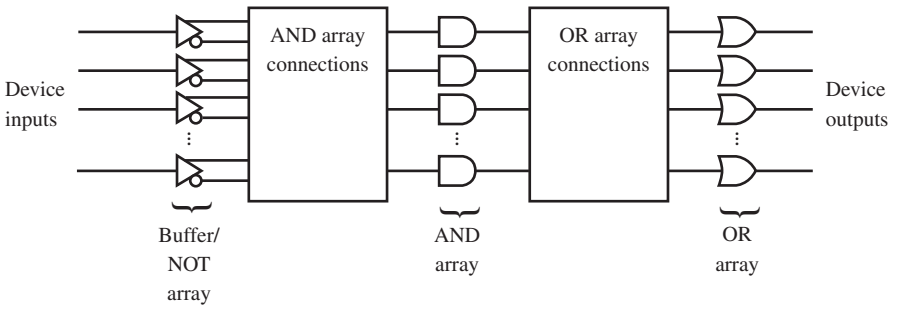


FIGURE 8.1 Template or generalized AND/OR gate form architecture for PLDs

to be sure the functions they wanted to implement would fit into the architecture of the PLDs they chose to use. To satisfy designers, industry has continued to manufacture larger and larger PLDs. Larger PLDs are ones with an increase in the number of gates.

Table 8.1 shows the classification of the different types of PLDs. The connections to the AND array, connections to the OR array, and volatility of the circuit connections determine the type of PLD. Nonvolatile connection indicated that the connections are not lost (not broken) when the power is turned off, while volatile connections are lost (effectively broken) when power is turned off.

TABLE 8.1 Classification of the different types of PLDs

Device type	AND array connections	OR array connections	Volatility of the connections
PROM	Fixed	Programmable	Nonvolatile
PLA	Programmable	Programmable	Nonvolatile
PAL or GAL	Programmable	Fixed	Nonvolatile
FPGA with LUTs	Fixed	Programmable	Volatile

A brief symbology summary for the different types of PLDs is shown in Figure 8.2.

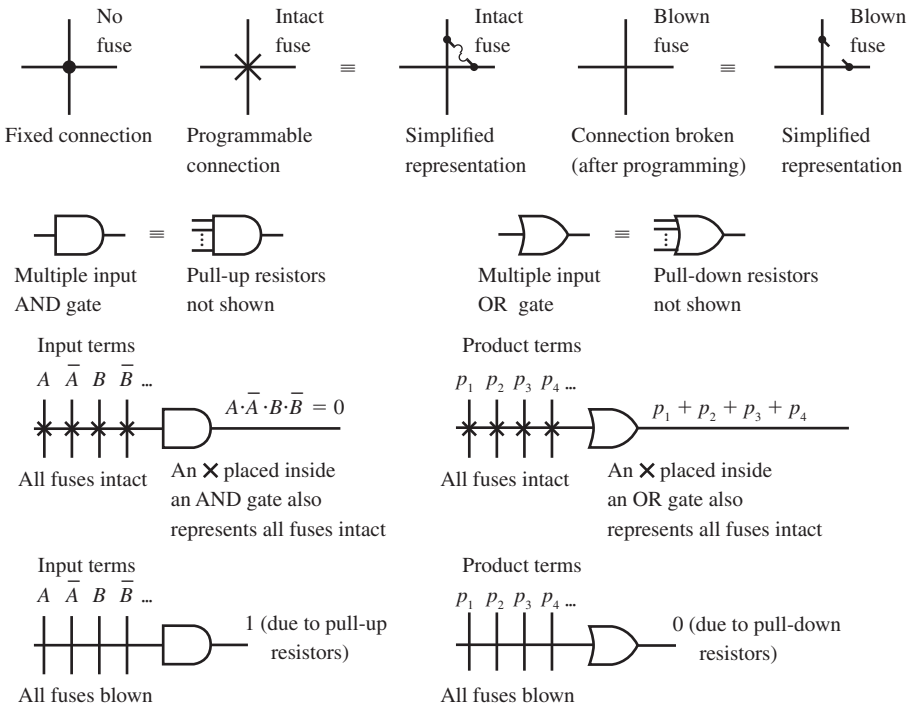


FIGURE 8.2 Symbology summary for the different types of PLDs

In the summary, you should note that an "×" is used as a programmable connection (an intact fuse), not a don't care. Actual fuses may be used or they may be emulated by a transistor circuit that is reprogrammable. If a transistor circuit is emulating a fuse connection, the circuit may be **volatile** (such as the RAM storage in your PC; when the power is turned off the connection is effectively broken) or **nonvolatile**. If the circuit is nonvolatile, the connection is not broken when power is turned off. Nonvolatile circuits may be of the old ultraviolet erasable type or of the electrically erasable type. Erasing a PLD reestablishes all fuse connections. Once a reprogrammable device is programmed, it needs to be erased prior to programming it again. The old ultraviolet erasable type requires a window in the top of package to expose the circuit to ultraviolet light when it needs to be erased. The electrically erasable type is state-of-the-art technology, and it may be erased more quickly by simply applying pulses to the correct device pins. Devices using electrically erasable technology can be erased and reprogrammed even after the device has been mounted on a PC board.

A device called a universal programming unit can be used to erase and program programmable devices. Several different companies manufacture these units. To use the universal programming unit, you must first generate a map of the fuses that need to be blown. The fuses that are not blown are the connections you want to keep, and the connections that are blown are the ones you don't want to keep. A map of the fuses is called a **fuse map** and is generated automatically by specialized industrial software packages. A standard has been devised by the Joint Electronic Devices Engineering Council—the JEDEC standard fuse map file—for certain types of PLDs. Once the device is programmed with the proper connections for the required Boolean functions, the PLD is just a special-purpose circuit. For devices that emulate fuses, the circuit is semipermanent, and the emulated fuse connections can last for several years. If actual fuses are used, the fuses are permanent and cannot be reprogrammed.

8.2.1 PROMs and LUTs

Let's look a little closer to see how the AND/OR gate circuits are connected for the different PLDs. Figure 8.3 shows a PROM simplified circuit representation or a LUT (look-up table) simplified circuit representation in an FPGA with just 2 inputs and 4 outputs.

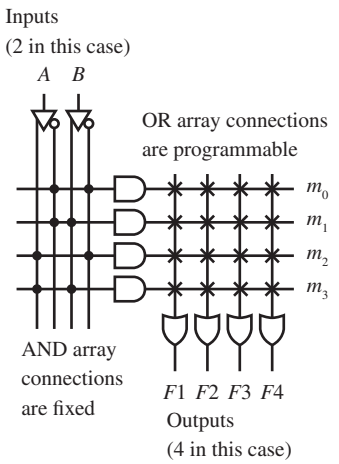


FIGURE 8.3 PROM or LUT simplified circuit representation

The PROM or LUT circuit in Figure 8.3 consists of four 4-input AND gates and 4 outputs. Much larger PROM circuits are available with many more inputs and outputs. LUTs in FPGAs are generally limited to 4 to 8 inputs and just a single output. For the simple PROM or LUT circuit shown in Figure 8.3, each additional input that is added causes the AND array to double

in size—that is, for 2 inputs there are four AND gates, for 3 inputs there are eight AND gates, and so on (for n-inputs there are 2^n AND gates).

Observe that the AND array is a minterm generator circuit (or decoder circuit). For each combination of inputs, only one AND gate output goes high at one time. For example, for $A\ B = 00$, the top AND gate is pulled to a 1 or high and all other AND gates are pulled to a 0 or low. For $A\ B = 01$, the second AND gate down is pulled to a 1 or high and all other AND gates are pulled to a 0 or low. A circuit that performs in this manner is called a decoder because it selectively decodes at its outputs each binary combination applied to its inputs. Observe that the minterm outputs are shared by each of the OR gates.

A nonprogrammable **read-only memory (ROM)** is **mask programmed** one time at the factory and may not be altered. Metal connections are used for the mask, and these are permanent connections. Mask-programmed ROMs are much more economical and are used after designs are finalized. PROMs are used primarily in the development stage of a project when designs are not fixed and may be changed. Remember that PROMs are nonvolatile. FPGAs with LUTs use the same circuitry as the PROM circuit shown in Figure 8.3, but the LUTs usually have only one output and the circuit connections are volatile.

8.2.2 PLAs

Figure 8.4 shows a PLA simplified circuit representation with just 3 inputs and 3 outputs. The PLA circuit in Figure 8.4 consists of five 6-input AND gates and three 5-input OR gates.

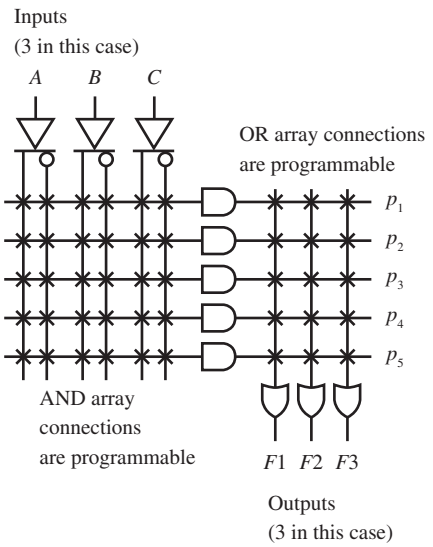


FIGURE 8.4 PLA simplified circuit representation

It seems natural to look back in history and see why Signetics invented the PLA. When the decoder in the PROM circuit is replaced by another programmable array—the AND array—the PLA can have more inputs without increasing the size of its circuit. The PLA circuit is quite versatile. All the product-term expressions generated at the outputs of the AND array are shared by all the ORs in the OR array just like the minterm expressions are shared in a PROM circuit.

Notice that the PLA circuit has two arrays that must be programmed compared to only one array for the PROM circuit. Two fuse maps must be generated for a PLA circuit.

8.2.3 PALs or GALs

Figure 8.5 shows a PAL simplified circuit representation with just 3 inputs and 2 outputs. The PAL or GAL circuit in Figure 8.5 consists of four 6-input AND gates and two 2-input OR gates.

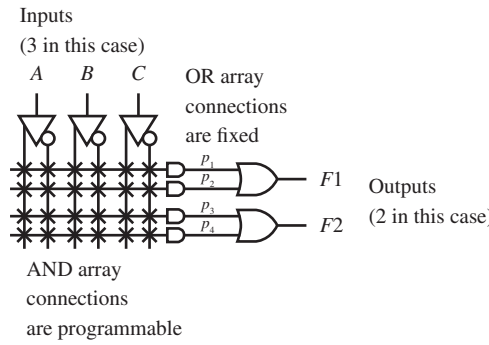


FIGURE 8.5 PAL or GAL simplified circuit representation

Like the PLA circuit, increasing the number of inputs for the PAL circuit does not cause the AND array to double in size for each additional input as it does for the PROM circuit.

When PALs were first introduced, they became the programmable device workhorse of the industry for small designs because of their lower cost, higher speed, and ease of use. When several PALs or GALs are included on the same chip or several PLAs are included on the same chip, industry refers to theses devices as CPLDs (complex programmable logic devices). Today, the programmable device workhorses are CPLDs (for small to medium-size designs) and FPGAs (for medium- to very large-size designs).

In the PAL circuit in Figure 8.5, there is no product-term expression sharing because each AND gate output cannot be used by more than one OR gate input. This is the only drawback of the PAL compared to the PLA which has product-term expression sharing.

One might ask, "Why use PLDs to design circuits when you can always build circuits in gate form?" It is desirable to use programmable devices for the following reasons: (1) to shorten design time—PLD designs *provide rapid prototyping* because fuses serve as connections or no connections based on the fuse map, making all manual wiring of the gates unnecessary; (2) to *allow rapid design changes*—changing the fuse map or fuse connections changes the PLD design; (3) to *decrease PC board real estate*—PLD designs in a single package are less costly and bulky than multiple packages for gate-level designs; and (4) to *improve reliability*—PLD designs require fewer packages and thus fewer interconnections. The first two items in the list are the most important for students. They provide rapid prototyping and allow rapid design change.

8.2.4 Designing with PROMs or LUTs

The design techniques we present here are automated and, therefore, are provided automatically by manufactures' software. Our designs simply illustrate what a software program must do when programming a device.

Table 8.2 shows the truth table for a NOT function, an OR function, a NAND function, and an XOR function.

TABLE 8.2 Truth table for a NOT function, an OR function, a NAND function, and an XOR function

A	*B*	*F*1	*F*2	*F*3	*F*4
0	0	1	0	1	0
0	1	1	1	1	1
1	0	0	1	1	1
1	1	0	1	0	0

Figure 8.6 shows how the functions in Table 8.2 are implements with a PROM or a LUT.

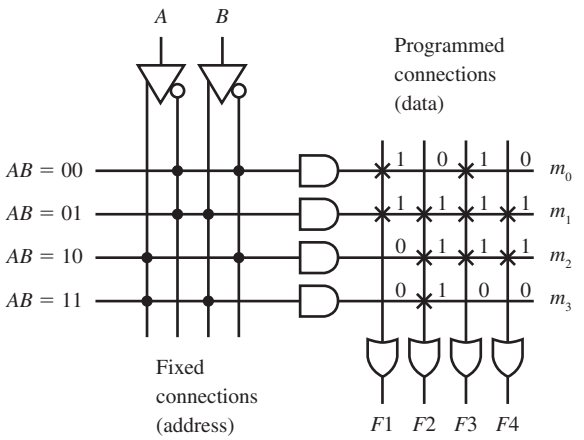


FIGURE 8.6 Implementation for a NOT function, an OR function, a NAND function, and an XOR function with a PROM or LUT

In Figure 8.6, observe that $F1(A,B) = \Sigma m(0,1)$, $F2(A,B) = \Sigma m(1,2,3)$, $F3(A,B) = \Sigma m(0,1,2)$, and $F4(A,B) = \Sigma m(1,2)$.

Because a PROM or LUT has a decoder front end that generates all the minterms for each of the functions, think of the inputs to a PROM or LUT as an address and its outputs as data at each address. A 1 in each column of a function in the table represents a fuse connection that has not been blown (an intact fuse). A 0 in each column of a function in the table represents a fuse connection that has been blown (a blown fuse). Given the circuit for a PROM or LUT without the fuses, we can rapidly draw the intact fuses as shown in Figure 8.6. A universal programmer does just the opposite because it actually blows the fuses where the connections need to be removed because each new device comes with all fuses intact.

The fuse map consists of the addresses 00B, 01B, 10B, 11B (where the B stands for binary). Addresses can also be represented in decimal, octal, or hexadecimal. The corresponding data listed in binary are 1010B, 1111B, 0111B, and 0100B, respectively. In hexadecimal, the *ADDRESS* and *DATA* are listed in Table 8.3.

ADDRESS (Hexidecimal)	*DATA* (Hexidecimal)
0	A
1	F
2	7
3	4

TABLE 8.3 Hexadecimal listing of the address and data for the PROM or LUT implementation shown in Figure 8.6

To change the fuse map we simply need to change the data for the PROM or LUT. If we decide later to complement input *B* rather than input *A* for function *F*1, then the address and data sequence would be changed to address (data) as follows: 0(A), 1(7), 2(F), 3(4). This is a small change (software change) to the fuse map that initiates a hardware change (reprogramming of fuses) inside the device.

8.2.5 Designing with PLAs

A PLA requires two fuse maps to implement a function. One fuse map is for the AND array connections and a different fuse map is for the OR array connections. Because there is a fuse

map for the OR array, product-term expressions are shared. Figure 8.7 shows how the functions in Table 8.2 can be implements with a PLA.

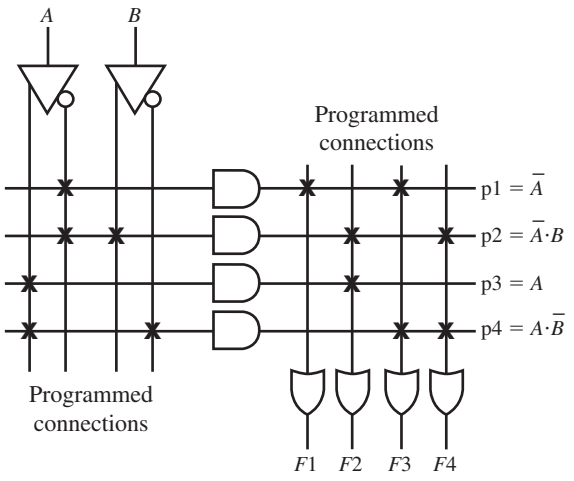


FIGURE 8.7 Implementation for a NOT function, an OR function, a NAND function, and an XOR function with a PLA

PLA implementations are usually made by software designed to allow product-term expression sharing to occur. Notice in the K-maps in Figure 8.8 that the product-term expressions are chosen for the AND array connections and then shared in the OR array connections.

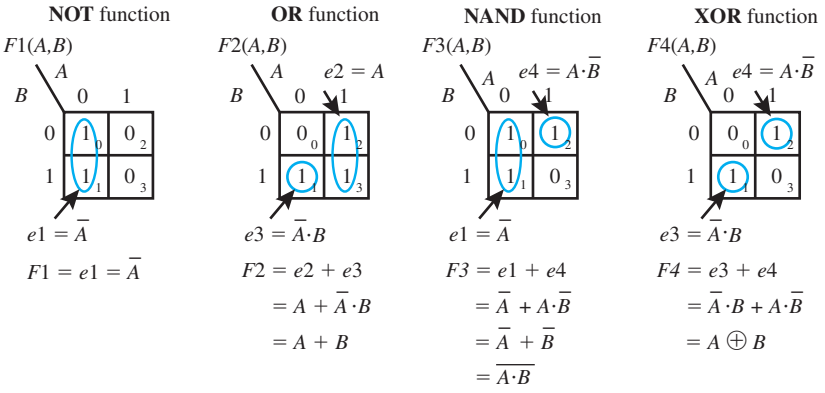


FIGURE 8.8 Product-term expression sharing for PLA implementation

8.2.6 Designing with PALs or GALs

The circuit for a PAL or GAL has a programmable AND array and a fixed OR array. As a result, only one fuse map is required for the AND array. To use a minimum number of gates on the chip, functions are generally minimized prior to creating the fuse map. To create the fuse map, the NOT, OR, NAND, and XOR functions in Table 8.2 are written in the following minimized forms:

$F1 = \overline{A}$, $F2 = A + B$, $F3 = \overline{A} + \overline{B}$, and $F4 = \overline{A}\cdot B + A\cdot\overline{B}$

Figure 8.9 shows how the functions in Table 8.2 are implemented with a PAL or GAL. To provide a product term of 0, we simply keep all fuses intact (not blown) in the AND array connections.

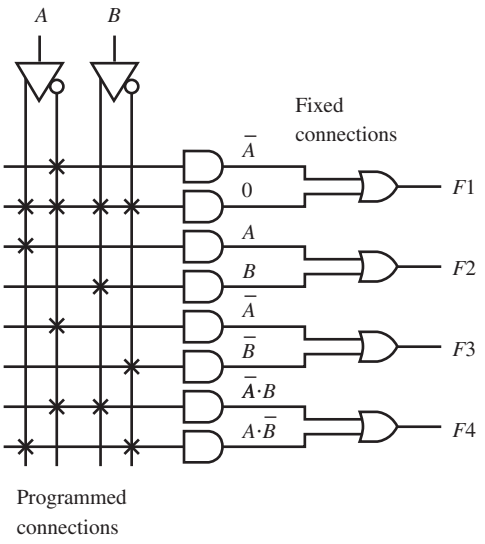


FIGURE 8.9
Implementation for the NOT, OR, NAND, and XOR functions with a PAL or GAL

With these examples in mind, you should be able to create a fuse map for any type of PLD to implement Boolean functions.

8.3 POSITIVE LOGIC CONVENTION AND DIRECT POLARITY INDICATION

Up until now we have only used logic symbols and signal names that correspond to the **positive logic convention (PLC)** system. Only positive logic signals are used in the PLC system. In the PLC system, the logic level H represents the logic state of 1 and the logic level L represents the logic state of 0, or H = 1 and L = 0.

Some books and data sheets provided by manufacturers use a different signal naming notation and logic symbol notation. This system is referred to as the **direct polarity indication (DPI)** system. In the DPI system, each signal name is assigned a suffix of either H or L to indicate the logic convention chosen for the signal name. These signal names are called **polarized signals**. A suffix with H indicates a positive logic signal, while a suffix with L indicates a negative logic signal. Both positive logic signals and negative logic signals exist in the DPI system. For negative logic signals, L = 1 and H = 0.

Engineers who prefer to work with 1s and 0s usually choose the PLC system while those who prefer to work with voltage levels usually choose the DPI system. We prefer to work with 1s and 0s and therefore have chosen to use the PLC system throughout this text, except in this section of the text.

Knowledge of both the PLC system and the DPI system will provide you with a better understanding when reading technical literature or when conversing with other engineers who use the DPI system.

8.3.1 Signal Names

Polarized signals have two parts, the **expression part** and the **suffix part**. In the polarized signal name *X*(H), the expression part is *X* and the suffix part is H. In the polarized signal name

$\overline{Y}$(H), the expression part is $\overline{Y}$ and the suffix part is H. In the polarized signal name $A{\cdot}B$(L) the expression part is $A{\cdot}B$ and the suffix part is L.

Table 8.4 shows a summary of the signals used in the PLC system and in the DPI system.

TABLE 8.4 Summary of the signals used in the PLC system and in the DPI system

Signal name in the PLC system	Equivalent polarized signal name in the DPI system
$X = X$(H)	X(H) $= \overline{X}$(L)
$\overline{Y} = \overline{Y}$(H)	$\overline{Y}$(H) $= Y$(L)
$\overline{A{\cdot}B} = \overline{A{\cdot}B}$(H)	$\overline{A{\cdot}B}$(H) $= A{\cdot}B$(L)

Double complementation can be used to obtain equivalent signal names for polarized signals in the DPI system as follows:

$$X(\mathrm{H}) = \overline{\overline{X(\mathrm{H})}^{1}}^{2} = \overline{X}^{1}\,\overline{(\mathrm{H})}^{2} = \overline{X}(\mathrm{L})$$

$$\overline{Y}(\mathrm{H}) = \overline{\overline{\overline{Y}(\mathrm{H})}^{1}}^{2} = \overline{\overline{Y}}^{1}\,\overline{(\mathrm{H})}^{2} = Y(\mathrm{L})$$

$$\overline{A{\cdot}B}(\mathrm{H}) = \overline{\overline{A{\cdot}B}(\mathrm{H})}^{1}{}^{2} = \overline{\overline{A{\cdot}B}}^{1}\,\overline{(\mathrm{H})}^{2} = A{\cdot}B(\mathrm{L})$$

There is a slight difference in graphics notation for each system. The PLC system uses a **bubble** or **negation symbol** in a circuit to indicate complementation. The presence or absence of a bubble in the PLC system is a **negation indicator**.

The DPI system uses a **wedge** or **polarity symbol** in a circuit. If a wedge or polarity symbol is present at an input or output, that input or output is active low. If a wedge or polarity symbol is not present at an input or output, that input or output is active high. The presence or absence of a wedge in the DPI system is a **polarity indicator**.

8.3.2 Analyzing Equivalent Circuits for the PLC and the DPI Systems

Figure 8.10 shows a circuit using the PLC system in NAND/NAND form.

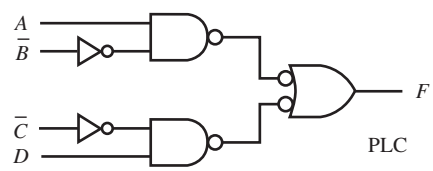


FIGURE 8.10 Circuit using the PLC system in NAND/NAND form

Things you should notice about the circuit in Figure 8.10:

- The circuit is drawn using the PLC system.
- Signals A and D are normal signals, while signals B and C are complemented signals. Signal F is a normal signal.
- Negation indicators matching along internal signal lines are not important.

The Boolean function for the circuit in Figure 8.10 is $F = A{\cdot}B + C{\cdot}D$. A simple way to specify the available signals that are applied to the inputs in the circuit and the available signal

at the output of the circuit is by a list called the signal list (SL). The signal list for the circuit in Figure 8.10 is SL: $A, \overline{B}, \overline{C}, D, F$. Combining the Boolean function and the signal list provide a complete description of the circuit in Figure 8.10 as $F = A{\cdot}B + C{\cdot}D$; SL: $A, \overline{B}, \overline{C}, D, F$.

You can convert a circuit drawn in the PLC system to the DPI system by simply changing all bubbles to wedges. You must also write the equivalent polarized signals at the inputs and output of the circuit. **Polarity/indicator (P/I) matching** at the beginning and ending of each internal signal line is sometimes recommended. This is helpful but not really necessary.

Figure 8.11 shows the same circuit in Figure 8.10 using the DPI system in NAND/NAND form. Equivalent signal names for the inputs and the outputs are shown in the circuit in Figure 8.11, and we have provided polarity/indicator (P/I) matching for this example.

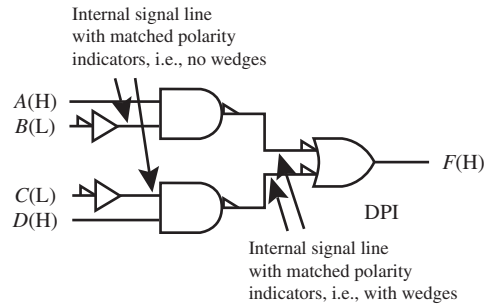


FIGURE 8.11 Circuit using the DPI system in NAND/NAND form

Figure 8.12 shows how we obtain the Boolean function for the circuit in Figure 8.11.

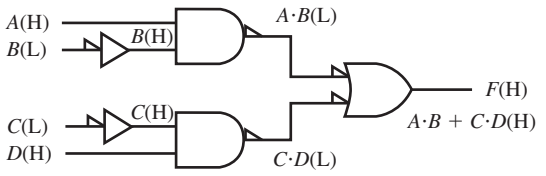


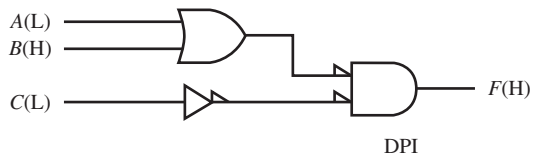
FIGURE 8.12 Obtaining the Boolean function for the circuit in Figure 8.11

The Boolean function for the circuit in Figure 8.12 is $F(H) = A{\cdot}B + C{\cdot}D(H)$ and the signal list is SL: $A(H)$, $B(L)$, $C(L)$, $D(H)$, $F(H)$.

To analyze the circuit in Figure 8.12 and obtain its Boolean function, we used a concept called **signal/indicator (S/I) matching**. S/I matching exists throughout the circuit in Figure 8.12, which made it easy to analyze. This was done on purpose. If the polarity indicator on a logic line contains a wedge or polarity symbol, a signal with the suffix (L) is required to provide an S/I match. If the polarity indicator on a logic line does not contain a wedge or polarity symbol, a signal with the suffix (H) is required to provide an S/I match. If a signal's suffix on a logic line does not match the polarity indicator on that logic line, then there is a mismatch. To analyze a circuit, S/I matching should be provided throughout the circuit (as it does in Figure 8.12). To provide a match, if one does not occur, simply use the equivalent polarized signal name to obtain a match of the suffix with the polarity indicator. After matching all the suffixes with the polarity indicators in the circuit, the Boolean function for the circuit is simply written using the expression parts of the polarized signal names.

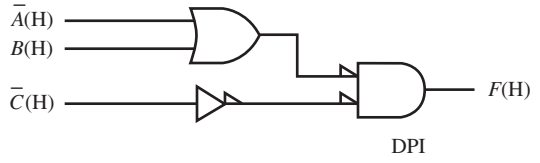
Figure 8.13 shows a circuit using DPI. Observe in this example that polarity/indicator (P/I) matching does not occur on the signal line between the output of the OR gate, which does not contain a wedge, and the input to the AND gate, which does have a wedge.

FIGURE 8.13 Circuit using DPI



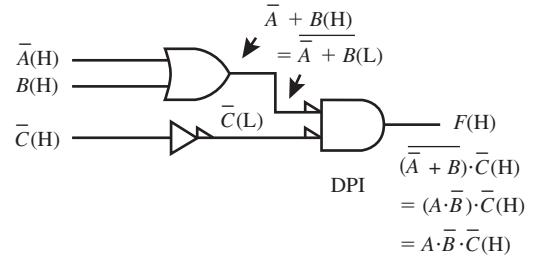
To analyze the circuit, first provide signal/indicator (S/I) matching on all the inputs and outputs in the circuit. Figure 8.14 shows the circuit in Figure 8.13 with S/I matching at the inputs and output of the circuit.

FIGURE 8.14
Circuit using DPI with S/I matching at the inputs and output of the circuit



Next, write the Boolean signals throughout the circuit using S/I matching. The key to analyzing a circuit in the DPI system is to mentally think of (or write down) the required polarized signals to obtain S/I matching on each signal line throughout the circuit, as shown in Figure 8.15.

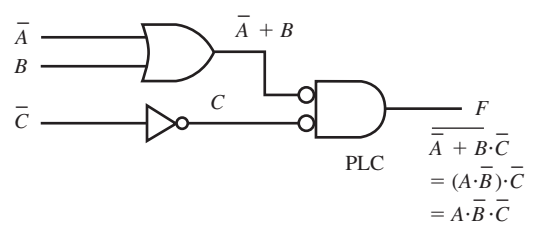
FIGURE 8.15 Circuit using DPI with S/I matching throughout the circuit and the resulting Boolean signals for the circuit



The *Boolean function* for the circuit in Figure 8.15 is $F(H) = A \cdot \overline{B} \cdot \overline{C}(H)$. The signal list for the circuit is *SL*: *A*(*L*), *B*(*H*), *C*(*L*), *F*(*H*). Notice that all of the signals in the signal list are written in their **simplest form**—that is, as *noncomplemented polarized signals*.

To check our analysis, let's convert the circuit in Figure 8.13 that uses the DPI system to a circuit that uses the PLC system. You can convert a circuit drawn in the DPI system to the PLC system by simply changing all wedges to bubbles. You must also write the equivalent positive logic signals at the inputs and output of the circuit. Matching or not matching negation indicators is not important in the PLC system. Equivalent signal names for the inputs and the outputs are shown in the circuit in Figure 8.16 in the PLC system.

FIGURE 8.16
Circuit using PLC in Figure 8.13 drawn using PLC system



The *Boolean function* for the circuit in Figure 8.16 is $F = A \cdot \overline{B} \cdot \overline{C}$. The signal list for the circuit is $SL: \overline{A}, B, \overline{C}, F$. Observe that the Boolean functions and the signal lists that we obtained for the circuits in Figure 8.13 and Figure 8.16 are equivalent.

If you read a book or a data sheet that uses the DPI system you should know how to analyze the circuit. Given a circuit in the DPI system, we prefer to convert the circuit to the PLC system and then analyze the circuit in the PLC system.

8.4 MORE ABOUT MUXS AND DMUXS

Sometimes it is important to know how to design a circuit that reduces the number of signal lines when routing signals from one location (the source) to another location (the destination). Figure 8.17 shows a **data transmission routing scheme** for this type of application that uses an 8-to-1 MUX (**multiplexer**) and a 3-to-8 DMUX (**demultiplexer**). Remember that a demultiplexer is a decoder with an enable input. At any one time for this data transmission routing scheme, only one data source (*DS*) signal gets routed to a data destination (*DD*) signal at the output; *DSi* effectively gets routed to *DDi*, for $i = 0$ to 7. This circuit provides a form of time-division multiplexing.

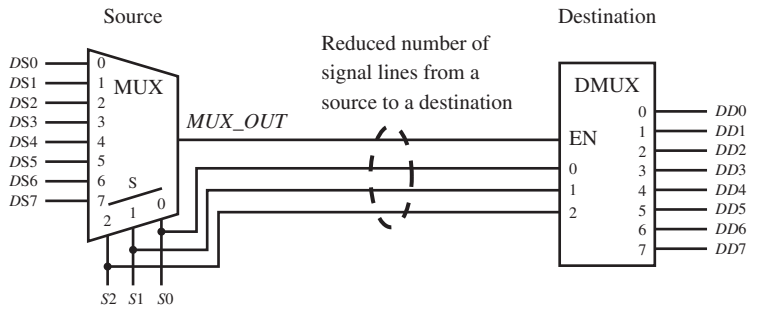


FIGURE 8.17 Data transmission routing scheme using MUX and DMUX circuits

This type of data transmission routing scheme can be used across an IC (integrated circuit), across a PC (printed circuit) board, or even across an office or a longer distance to reduce the number of signal lines from the source to the destination. In Figure 8.17, only four signal lines (*MUX_OUT*, *S2*, *S1*, and *S0*) are required, rather than eight signal lines (*DS*7 down to *DS*0).

Listing 8.1 (on the next page) shows a complete VHDL design for the data transmission routing scheme (DTRS) using MUX and DMUX circuits as shown in Figure 8.17.

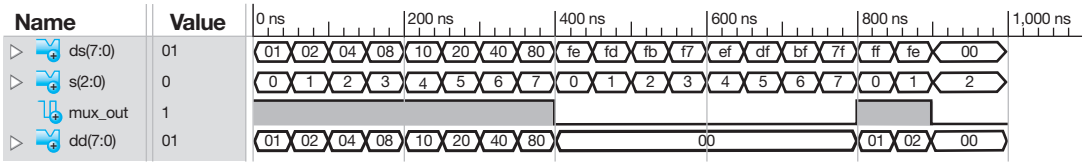
Things you should notice about the VHDL design in Listing 8.1:

- Vector notation was used for the inputs to provide less typing when writing the VHDL code.
- Internal signal *MUX_OUT* is placed in the entity with mode **inout** so it will be shown in the simulation.
- Because a CSA and a process were used in the architecture of the design, we called this a mixed design style.
- A concatenation operator was use in the **if statement** to form the signal *S&MUX_OUT*, which consist of 4 bits—that is, the 4 bits required for the inputs to the DMUX circuit.

Waveform 8.1 shows the simulation with the correct functionality of design entity DTRS.

LISTING 8.1
Complete VHDL design for a data transmission routing scheme (project: DTRS)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DTRS is port (
     ds : in std_logic_vector(7 downto 0);
     s : in std_logic_vector(2 downto 0);
     mux_out : inout std_logic;
     dd : out std_logic_vector(7 downto 0)
     );
end DTRS;

architecture Mixed of DTRS is
begin
     mux_out <= ds(0) when s <= "000" else
                ds(1) when s <= "001" else
                ds(2) when s <= "010" else
                ds(3) when s <= "011" else
                ds(4) when s <= "100" else
                ds(5) when s <= "101" else
                ds(6) when s <= "110" else
                ds(7);

process (s,mux_out)
begin
     if    s&mux_out = "0001" then dd <= "00000001";
     elsif s&mux_out = "0011" then dd <= "00000010";
     elsif s&mux_out = "0101" then dd <= "00000100";
     elsif s&mux_out = "0111" then dd <= "00001000";
     elsif s&mux_out = "1001" then dd <= "00010000";
     elsif s&mux_out = "1011" then dd <= "00100000";
     elsif s&mux_out = "1101" then dd <= "01000000";
     elsif s&mux_out = "1111" then dd <= "10000000";
     else                          dd <= "00000000";
     end if;
end process;
end Mixed;
```



WAVEFORM 8.1 Simulation with the correct functionality of design entity DTRS

Things you should notice about the Waveforms in Waveform 8.1:

- All the vector signals are displayed in hexadecimal values.
- Observe that signal *ds*[*i*] effectively gets routed to the output signal *dd*[*i*] for $i = 0$ to 7, for each value of the select input *s* when *mux_out* = 1. When *mux_out* = 0, output signal *dd*[*i*] = 00 for $i = 0$ to 7, for each value of the select input *s*.
- The simulation was not an exhaustive simulation run, but it does show that the data transmission routing scheme is correct for the selected inputs.

It is also important in some cases to be able to design larger MUXs and DMUXs. The following sections cover an introduction to designing MUX trees and DMUX trees via a **modular design technique**.

8.4.1 Designing MUX Trees

Figure 8.18 shows a design for a 4-to-1 MUX tree that uses three 2-to-1 MUXs to form the tree. The 4-to-1 MUX tree is equivalent to a 4-to-1 MUX with data inputs *D*3:*D*0—that is, *D*3, *D*2, *D*1, *D*0, select inputs *A*(MSB), *B*(LSB), and output F.

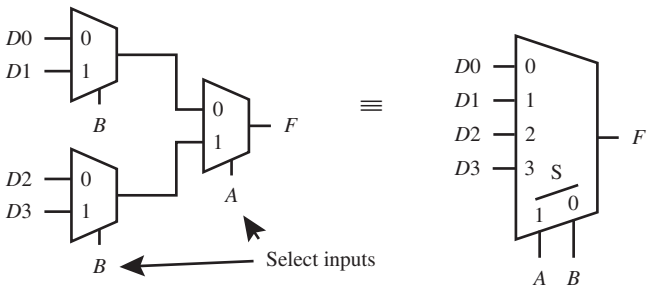


FIGURE 8.18 4-to-1 MUX tree using three 2-to-1 MUXs

Larger MUX trees can be also be designed using this modular design technique.

The propagation delay time for the signal to propagate through this MUX tree is twice the propagation delay time as a single MUX design that performs the same function. This is the disadvantage of a MUX tree design. In general, you should only design a MUX tree when you can't find a MUX for your design requirement or when your design tools do not provide a MUX that is large enough for your design requirement.

8.4.2 Designing DMUX Trees

Figure 8.19 shows a gate-level circuit for a 1-to-2 DMUX with its corresponding logic symbol.

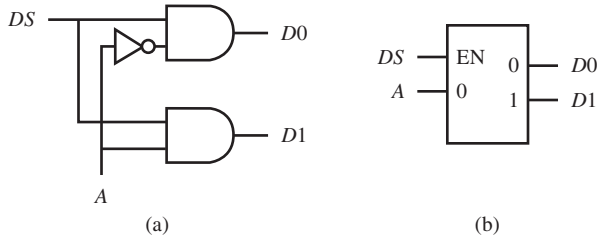


FIGURE 8.19 1-to-2 DMUX: (a) gate-level circuit; (b) logic symbol

The modular design technique for designing a DMUX tree is very similar to the design technique for designing MUX trees.

Figure 8.20 shows a 2-to-4 DMUX tree using three 1-to-2 DMUXs to form the tree. The 2-to-4 DMUX tree is equivalent to a 2-to-4 DMUX with input DS (data source), select inputs *A*(MSB), *B*(LSB), and outputs *D*3:*D*0—that is, *D*3, *D*2, *D*1, *D*0.

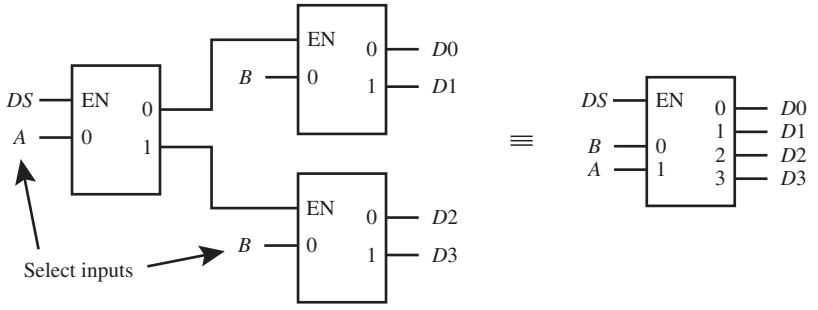


FIGURE 8.20 2-to-4 DMUX tree using three 1-to-2 DMUXs

Larger DMUX trees can be also be designed using this modular design technique.

The propagation delay time for the signal to propagate through this DMUX tree is twice the propagation delay time as a single DMUX design that performs the same function. This is the disadvantage of a DMUX tree design. In general, you should only design a DMUX tree when you can't find a DMUX for your design requirement or when your design tools do not provide a DMUX that is large enough for your design requirement.

PROBLEMS

Section 8.2 Programmable Logic Devices

8.1 Which gate form of architecture does a PLD have? Can equations written in POS form directly fit into this architecture?

8.2 What PLD has programmable AND array connections and also programmable OR array connections? Discuss the advantages and disadvantages of having two programmable array connections.

8.3 When several PLDs are included on the same chip, the resulting device is called a CPLD. What types of PLDs are used to make a CPLD?

8.4 Describe the form of the equations that can be fit into a PAL or GAL. (Hint: Equations are in the form of sum of products or in the form of product of sums.)

8.5 Use the same PROM architecture shown in Figure P8.5 to design a circuit for the following functions. Show the correct intact fuse connections with ×s—that is, provide the fuse map connections, for (a) an XOR function using inputs X and Z for $F1$; (b) a majority of 0s function using inputs X, Y, and Z for $F2$; and (c) an AND function using inputs X, Y, and Z for $F3$.

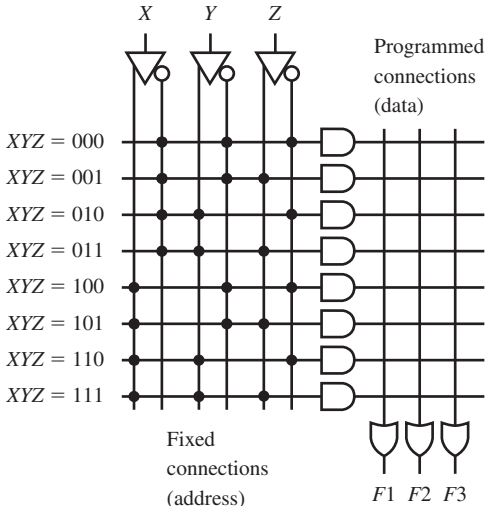


FIGURE P8.5

8.6 Design a circuit for the following functions using the small PAL in Figure P8.6. Only equations for the 0s of functions can be implemented with this PAL. Show the correct intact fuse connections with ×s—that is, provide the fuse map connections, for (a) an XNOR function using inputs Y and Z for $F1$; (b) an odd number of 1s function using inputs X, Y, and Z for $F2$; and (c) an OR function using inputs X, Y, and Z for $F3$. Observe that the simple PAL has active low outputs—that is, the outputs of the OR gates are inverted.

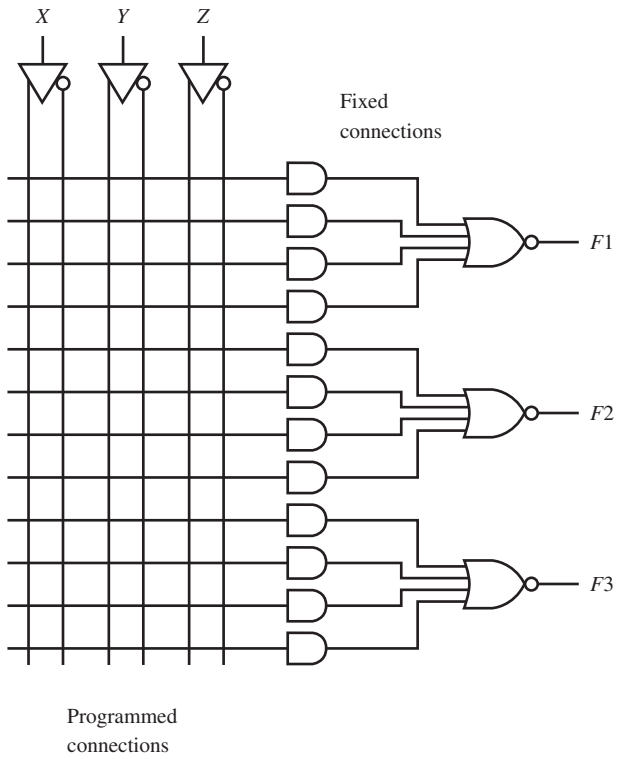


FIGURE P8.6

8.7 Show a design for the following Boolean functions using the simple 4-input, 4-output PAL shown in Figure P8.7. Observe that the simple PAL has active high outputs—that is, the outputs of the OR gates are not inverted.

$$F1(A,B,C,D) = \Sigma m(6,7,9,11,12,13)$$
$$F2(A,B,C,D) = \Sigma m(0,2,3,4,5,10,11,13,15)$$
$$F3(A,B,C,D) = \Sigma m(2,3,6,7,10,11,14,15)$$

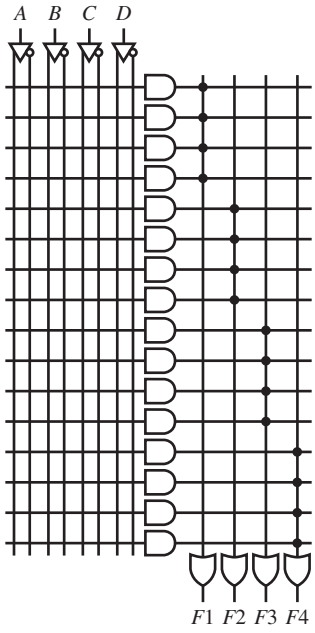


FIGURE P8.7

8.8 Repeat problem 8.7 using the simple 4-input, 4-output PAL shown in Figure P8.8. Observe that the simple PAL has active low outputs—that is, the outputs of the OR gates are inverted. (Hint: Use 0s of functions.)

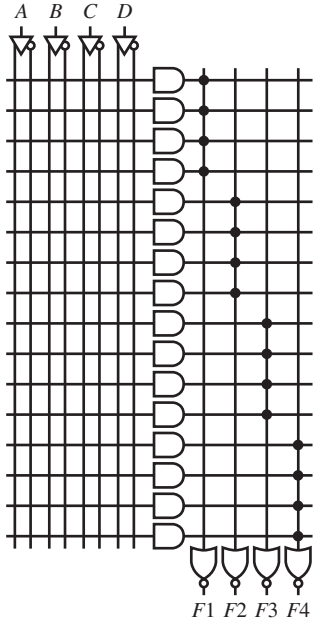


FIGURE P8.8

8.9 PALs are available today that have the capability to program the polarity of the output. This means that the software can choose to add or not to add an inverter on each OR gate output. Which type of equation must have an inverter added at the output of the OR gate (an equation for the 1s of the function or for the 0s of the function)? Provide an example that supports your answer.

8.10 Which type of output (inverted or noninverted) is required when implementing a function for its 0s in a PAL? Show why, using a simple example.

8.11 Which type of output (inverted or noninverted) is required when implementing a function for its 1s in a PAL? Use a simple example to show your reasoning.

Section 8.3 Positive Logic Convention and Direct Polarity Indication

8.12 Only positive logic signals are used in the PLC system. True or False?

8.13 What do the logic levels H and L represent in the PLC system?

8.14 Both positive logic signals and negative logic signals exist in the DPI system. True or False?

8.15 What is a signal name called that is written with a suffix of either H or L?

8.16 What type of signal in the DPI system is written with the suffix H? Define H and L.

8.17 What type of signal in the DPI system is written with the suffix L? Define L and H.

8.18 Name the two parts of a polarized signal.

8.19 Use double complementation to obtain the equivalent signal names for the following polarized signals:

a. A(L)

b. $A + B$(H)

c. $C \cdot D$(L)

d. $A \cdot \overline{B} + \overline{C}$(H)

8.20 What symbol is used as the negation indicator in the PLC system in a circuit?

8.21 What symbol is used as the polarity indicator in the DPI system in a circuit? What does a wedge or polarity symbol represent in a circuit?

8.22 When a wedge or polarity symbol is not present in the DPI system in a circuit, what does that represent?

8.23 Convert the circuit in Figure P8.23 into a circuit in the DPI system. Use polarized signals for the DPI system. Analyze the converted circuit to obtain its function and its corresponding signal list.

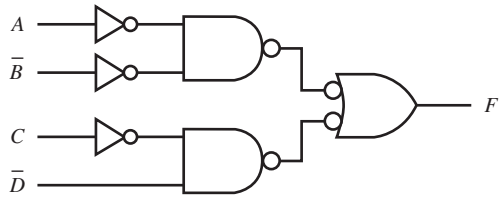


FIGURE P8.23

8.24 Convert the circuit in Figure P8.24 into a circuit in the DPI system. Use polarized signals for the DPI system. Analyze the converted circuit to obtain its function and its corresponding signal list.

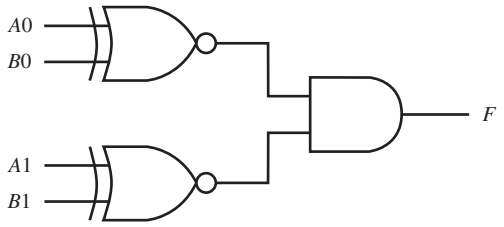


FIGURE P8.24

8.25 Convert the circuit in Figure P8.25 into a circuit in the DPI system. Use polarized signals for the DPI system. Analyze the converted circuit to obtain its function and its corresponding signal list.

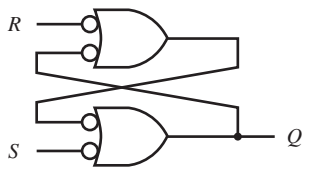


FIGURE P8.25

8.26 Convert the circuit in Figure P8.26 into a circuit in the PLC system. Write the signals for the circuit using the PLC system. Analyze the converted circuit to obtain its function and its corresponding signal list.

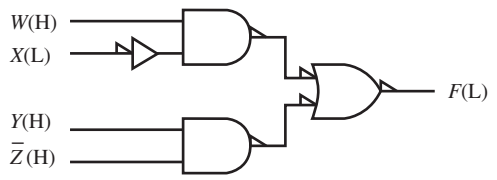


FIGURE P8.26

8.27 Convert the circuit in Figure P8.27 into a circuit in the PLC system. Write the signals for the circuit using the PLC system. Analyze the converted circuit to obtain its function and its corresponding signal list.

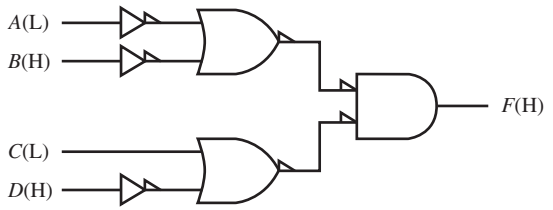


FIGURE P8.27

Section 8.4 More about MUXs and DMUXs

8.28 Draw a data transmission scheme to reduce the number of signal lines from 16 down to 5 for data that must be moved from one side of a PC board to the other side of the board. (Hint: Show the circuit that uses a 16-to-1 MUX and a 4-to-16 DMUX to provide the data transmission scheme.)

8.29 Describe the difference between a decoder and a demultiplexer.

8.30 Draw and label the circuit for an 8-to-1 MUX tree design that uses two 4-to-1 MUXs and one 2-to-1 MUX. Make the data inputs *D*7:*D*0; the select inputs *X*(MSB), *Y*, *Z*(LSB); and the output *F*. (Hint: Use the modular design technique discussed in the book.)

8.31 Draw and label the circuit for an 8-to-1 MUX tree design that uses four 2-to-1 MUXs and one 4-to-1 MUX. Make the data inputs *D*7:*D*0; the select inputs *X*(MSB), *Y*, *Z*(LSB); and the output *F*. (Hint: Use the modular design technique discussed in the book.)

8.32 Draw and label the circuit for a 16-to-1 MUX tree design that uses two 8-to-1 MUXs and one 2-to-1 MUX. Make the data inputs *D*16:*D*0; the select inputs *W*(MSB), *X*, *Y*, *Z*(LSB); and the output *F*. (Hint: Use the modular design technique discussed in the book.)

8.33 Draw and label the circuit for a 3-to-8 DMUX tree design that uses one 1-to-2 DMUX and two 2-to-4 DMUXs. Make the input *DS*; the select inputs *X*(MSB), *Y*, *Z*(LSB); and the outputs *D*7:*D*0. (Hint: Use the modular design technique discussed in the book.)

8.34 Draw and label the circuit for a 3-to-8 DMUX tree design that uses one 2-to-4 DMUX and four 1-to-2 DMUXs. Make the input *DS*; the select inputs *X*(MSB), *Y*, *Z*(LSB); and the outputs *D*7:*D*0. (Hint: Use the modular design technique discussed in the book.)

8.35 Draw and label the circuit for a 4-to-16 DMUX tree design that uses one 1-to-2 DMUX and two 3-to-8 DMUXs. Make the input *DS*; the select inputs *W*(MSB), *X*, *Y*, *Z*(LSB); and the outputs *D*15:*D*0. (Hint: Use the modular design technique discussed in the book.)

Chapter 9

Complex Finite State Machine Design with VHDL

Chapter Outline

9.1 INTRODUCTION

In this chapter, you will learn how to apply VHDL to design **complex finite state machines** (CFSMs) with the **two-process PS/NS method**. Our definition of a complex finite state machine is a state machine that has external inputs to change the state sequence, *excluding* the SET, RESET, and INITIALIZE inputs. The following description is an example of a complex state machine: If a state sequence requires a binary up counter at one time and a binary down counter at a different time, then an external input called *UP* can be included in the design to allow switching between the two state sequences. Complex finite state machines, or **complex state machines** (CSMs), and **simple state machines** are sometimes called **controllers**, because they are often used to control other circuits. In addition to flip-flop outputs, Moore and Mealy outputs are presented for complex state machines.

Synchronizers are introduced to improve the reliability of complex state machines. To complete our the discussion, we present two additional state machine design methods: the two-assignment PS/NS method and the hybrid PS/SN method. You can implement and download the last complex state machine design example, to observe it working in hardware, if you so desire.

9.2 DESIGNING WITH THE TWO-PROCESS PS/NS METHOD

Throughout this chapter, we will place our emphasis on designing complex state machines that are driven by a clock—that is, synchronous or clock mode digital circuits. The two-process PS/NS method is very versatile and can be used to design either simple or complex state machines. When you use this method, you do not have to obtain the excitation equations and draw the circuit. You simply use the method and let the VHDL software do the work. You can always observe the circuit that is produced using either **View RTL Schematic** or **View Technology Schematic** if you are using Xilinx software. Other software vendors have a similar option to allow you to view the resulting schematic.

Figure 9.1 shows a circuit model for the two-process PS/NS method. The two-process PS/NS method uses a behavioral design style with two processes. The first process is called the synchronous process (sync_proc), and the second process is called the combinational process (comb_proc). The synchronous process generates D flip-flops for the design, while the combinational process decodes the next-state (NS) functions for the D flip-flop inputs and also provides the flip-flop outputs as shown in Figure 9.1.

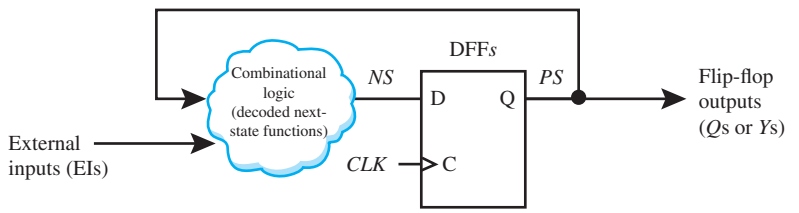


FIGURE 9.1 Circuit model for the two-process PS/NS method

A *SET*, *RST*, or *INIT* input is not shown for the flip-flops in Figure 9.1, because this state machine model may be designed with a *SET* input, a *RST* input, or *INIT* input—that is, the required input is provided in the design specification.

The **synchronous process** generates the D flip-flops for the state machine design and the following signals:

1. The present-state (*PS*) value after SET, RESET, or INITIALIZATION (a *SET*, *RST*, or *INIT* input is not shown in Figure 9.1).
2. The present-state (*PS*) value after the next rising edge (or falling edge) of the clock—that is, the value of the next state, or *PS* <= *NS*.

The **combinational process** generates the following signals:

1. The next-state (*NS*) value—that is, the decoded next state functions—based on the present-state (*PS*) value and the external inputs (EIs) via the cloud of combinational logic.
2. The flip-flop output values for *Q*s or *Y*s.
3. The Moore and Mealy output values (covered later).

The two-process PS/NS method requires less hardware understanding, but a more detailed understanding of VHDL. The two-process PS/NS method is considered the preferred VHDL coding style for complex state machine designs. By following this coding style, someone can easily understand the code that you write, and you can understand someone else's code.

Figure 9.2 shows a complex state machine named binary up/down counter (2 bits) with an external input *UP* that is used to change the state sequence. "State name" is shortened to "Name" in Figure 9.2. The state names are a, b, c, and d.

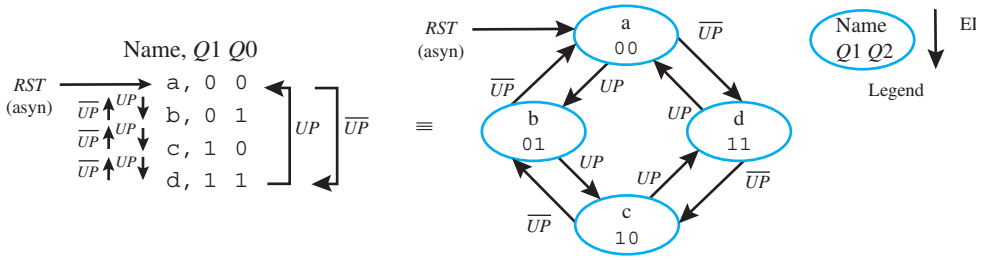


FIGURE 9.2 State sequence diagram in tabular form and equivalent state diagram for binary up/down counter (2 bits)

A complex state machine can be represented either as a **state sequence diagram** (or **counting sequence diagram**) in tabular form or as an equivalent **state diagram**, as illustrated in Figure 9.2. It is a good idea to obtain either a state sequence diagram or a state diagram for a complex state machine design prior to writing the VHDL code for the complex state machine.

The VHDL code for the binary up/down (BUD) counter in Figure 9.2 is shown in Listing 9.1 using the two-process PS/NS method. The two-process PS/NS method is a specific coding style. Other coding styles are possible, but we will predominantly use this coding style for complex state machine designs.

LISTING 9.1
Complete VHDL design entity for BUD_counter (project: BUD_counter)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity BUD_counter is port (
     rst,clk, up : in std_logic;
     q : out std_logic_vector (1 downto 0)
     );
end BUD_counter;
architecture behavioral of BUD_counter is
     type state_type is (a,b,c,d);
     signal ps, ns: state_type;
begin

sync_proc:
process (rst,clk)
begin
     if rst = '1' then ps <= a;
     elsif rising_edge (clk) then ps <= ns;
     end if;
end process;

comb_proc:
process (ps,up)
begin
     case ps is
          when a >= q <= "00"; if up = '1' then ns <= b;
                               else ns <= d;
                               end if;
          when b >= q <= "01"; if up = '1' then ns <= c;
                               else ns <= a;
                               end if;
                                                    (Continued)
```

```
        when c >= q <= "10"; if up = '1' then ns <= d;
                             else ns <= b;
                             end if;
        when d >= q <= "11"; if up = '1' then ns <= a;
                             else ns <= c;
                             end if;
    end case;
end process;
end behavioral;
```

Things you should notice about the VHDL design in Listing 9.1:

- The flip-flop outputs *Q*(1) and *Q*(0) in the entity declaration are specified as a std_logic_vector *Q*. This requires less writing to enter the flip-flop output values. The mode for *Q* is **out**, because *Q* only needs to be written (assigned a value), but not read.
- An **enumerated data type** called state_type is used to define the states a through d. Using an enumerated type removes the requirement of obtaining excitation equations for the design. It also allows the state sequence to be easily changed—that is, from a binary up/down counter to a binary Gray code up/down counter, or from one state sequence to any other state sequence. The signals present state (*PS*) and next state (*NS*) are internal signals and are declared to be of **type** state_type. The declaration for the **type** state_type and the internal signals *PS* and *NS* must appear between **architecture** and the first **begin**. The signals *PS* and *NS* can only be assigned the values of state_type—that is, these values are a, b, c, and d. The signal *PS* can also be assigned the signal *NS* because their types are the same.
- Only the inputs *RST* and *CLK* are required in the sensitivity list of the synchronous process (sync_proc), because the process only generates D flip-flops, which are completely controlled just by these two inputs.
- The synchronous process generates a clock-independent reset input that resets the state machine to state a via *PS* <= a, which is the reset state. Recall that the reset input *RST* must be placed before rising_edge (*CLK*) so that *RST* will be clock independent. The synchronous process is also used to assign a new value to the present state—that is, the value of the next state, or *PS* <= *NS*, after the next rising edge of the clock.
- Observe that the last **else** in the if statement in the synchronous process is inferred and therefore is not present. The inference means that *PS* <= *PS* is implied between clock ticks, so the flip-flop outputs maintain their current values until the next clock tick.
- The combinational process (comb_proc) contains all the inputs in its sensitivity list that must be read—that is, (*PS*, *UP*), in the combinational process.
- The major statement (the first statement in this case) in the combinational process (comb_proc) is the case statement with the select input *PS*. An if statement could be used as the major statement, but this generally results in more logic gates because an if statement has prioritized conditions, while a case statement only selects one condition; hence, a case statement with the select input *PS* is preferred as the major statement rather than an if statement.
- The combinational process (comb_proc) is used to assign the flip-flop outputs—that is, the *Q*(1) and *Q*(0) outputs—and the next-state (*NS*) outputs based on the present state (*PS*) and the synchronous external input *UP*. Therefore, the case statement (with accompanying if statements) in the combinational process is written to precisely follow the description specified by the state sequence.

- Observe that a **when others clause** is not required as the last choice value in the case statement because the data type of the select signal *PS* is state_type, and state_type only has the values of a, b, c and d, and no other values.
- The order for listing the two processes is not important, because process statements in VHDL execute concurrently.

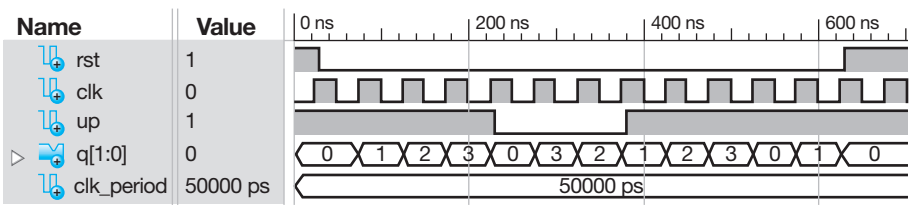
Each state in a state diagram has a distinct **state name** that is different from all other states in the state diagram. In VHDL, state names can be defined using an **enumerated data type** as shown in Listing 9.1 and repeated in the following clauses:

```
type state_type is (a,b,c,d);
signal PS, NS: state_type;
```

In the first clause, state_type (the name placed between **type** and **is**) is declared as an enumerated data type, and the state names for state_type are listed as a, b, c, and d (these represent a set of ordered values; i.e., the values have a default ordering beginning on the left and ending on the right). The data types used the most in VHDL are std_logic and std_logic_vector. For our simple examples, we did not name the states with intuitive or meaningful names. To use meaningful state names, we could use reset instead of a, one instead of b, two instead of c, and three instead of d. Using meaningful state names helps make our designs more readable and reusable.

In the second clause, the signals *PS* and *NS* are declared to be of the type state_type. This means that the signals *PS* and *NS* can only have the values that are assigned to state_type—that is, a, b, c, and d, and no other values.

Waveform 9.1 shows the correct functionality of design entity BUD_counter (binary up/down counter, 2 bits) for Listing 9.1.



WAVEFORM 9.1
Simulation for the correct functionality of design entity BUD_counter for Listing 9.1

The **state** of the counter is shown inside the waveform signal q(1:0) in Waveform 9.1.

The counter starts in state 0 because the signal rst is asserted (rst = 1). Pay attention to the direction of the counting sequence provided by output q(1:0) in relationship to the signal up. When rst is not asserted (rst = 0), the counter begins counting up through the sequence (1, 2, 3, 0), because signal up is asserted (up = 1). When up is not asserted (up = 0), the counter begins to count down through the sequence (3, 2, 1). When up is asserted again, the counter begins to count up through the sequence (2, 3, 0, 1). When rst is asserted at the end of the simulation, the counter goes to state 0.

9.3 EXPLANATION OF CPLDS AND FPGAS AND STATE MACHINE ENCODING STYLES

Figure 9.3 shows a simplified representation of part of a CPLD (complex programmable logic device).

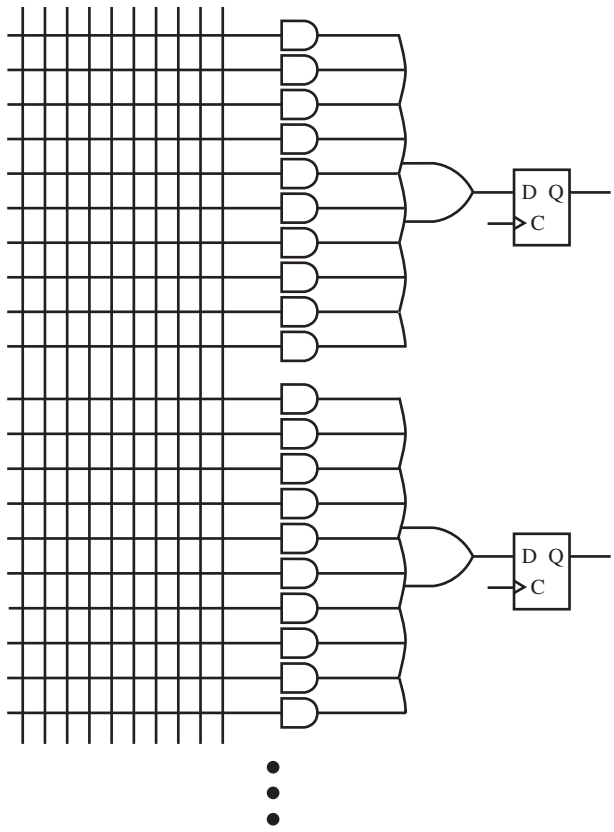


FIGURE 9.3
Simplified representation of part of a CPLD

Notice in Figure 9.3 that there are a large number of inputs and a large number of gates to provide the combinational logic for each D flip-flop. The interconnections are made via special transistors that store a charge to allow the interconnections to remain when power is removed and reapplied to the device. CPLDs of this type are classified as **nonvolatile** because the interconnections for a design remain when power is removed and reapplied. These special transistors have a large area requirement and thus limit the amount of total logic gates and D flip-flops that can be contained on the device chip.

The advantage of CPLDs is their very low power requirements and their wide gating or large number of input gates. This is provided at the expense of a small number of D flip-flops compared to FPGAs with the same chip size.

Figure 9.4 (on the facing page) shows a simplified representation of part of an FPGA (field programmable gate array).

Notice in Figure 9.4 that there are just a few inputs to provide the logic for each D flip-flop. However, the circuit is replicated many, many times so there are a very large number of D flip-flops. In the FPGA, the interconnections as well as the combinational logic (called LUTs or look-up tables) are made via SRAM (static random-access memory). SRAM has a much smaller area requirement and thus provides more space for D flip-flops compared to CPLDs with the same chip size. Because interconnections are not maintained when power is removed and reapplied to the device, SRAM-type FPGAs are classified as **volatile**.

To minimize the number of flip-flops required to implement a design, a **compact encoding style** (or **full encoding style**) can be used. This type of encoding style is considered best for CPLDs (complex programmable logic devices) that have a limited number of D flip-flops (or registers) and usually have very fast predictable timing paths through the PLD. The relationship $2^{\#FFs} \geq \#\ States$ can be used to determine the minimum number of state variables or flip-flops

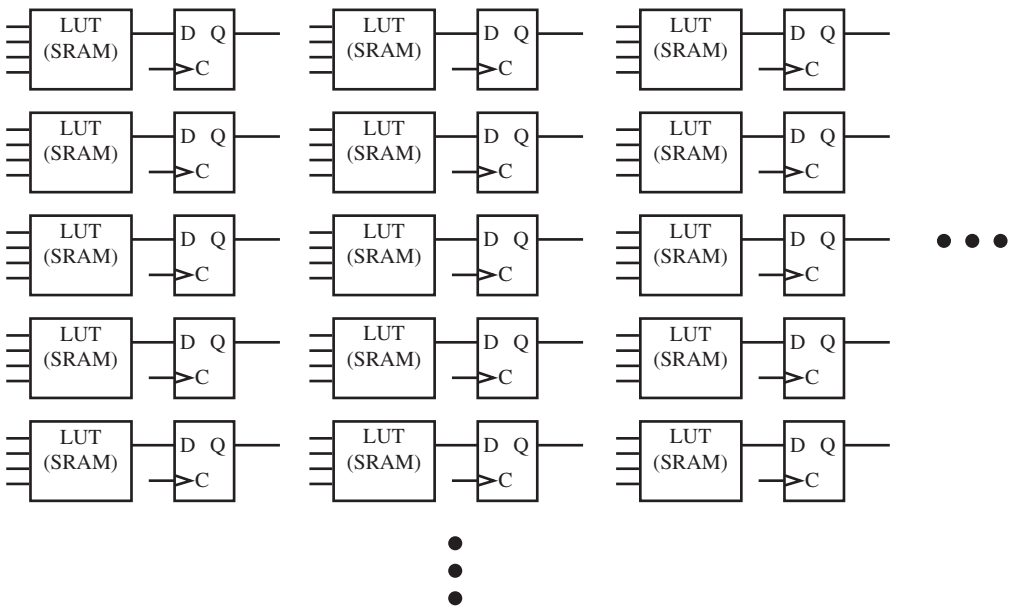


FIGURE 9.4
Simplified representation of part of an FPGA

(#FFs) required for a specified number of states (*# States*) for a compact encoded state machine. The relationship can also be used to determine the number of states given the number of state variables or flip-flops for a compact encoded state machine.

A state machine that has a dedicated flip-flop for each state is **one-hot encoded**. The relationship for the required number of state variables or flip-flops and the number of states for a one-hot encoded state machine is *# FFs = # States*. This type of encoding style is considered best for FPGAs that have an abundance of D flip-flops (or are register rich). With one dedicated flip-flop for each state, decoding logic is reduced for both the next-state functions and the output functions, and this encoding style can increase performance compared to using a compact or full encoding style for FPGAs.

Compact-encoded state machines generally require a minimum number of flip-flops and can require many gates. One-hot encoded state machines generally require a maximum number of flip-flops and just a few gates. A compact encoding style was used in the design of the binary counter in Chapter 6 and the binary up/down counter in Chapter 9, Figure 9.2, because only two flip-flops were used for the four states. A one-hot encoding style was used in the design of the one-hot counter in Chapter 6, because four flip-flops were used for the four states.

CAD (computer-aided design) tools from manufactures such as Xilinx®, Synplify®, Exemplar®, Synopsys®, and others include different types of encoding styles such as **Auto**, **Compact**, **Sequential/binary**, **One-hot**, **Gray**, **Zero One-hot**, **Johnson**, **Two-hot**, **User**, and **Random**. These tools allow a user to specify an encoding style for the states of a state machine via a GUI (graphical user interface) option provided by the tool. You need to learn the method for selecting the encoding style for the CAD tool you are using. For the Xilinx ISE software tool, right click on **Synthesis → Process Properties → HDL Options → FSM Encoding Algorithm**, then select **Compact** for compact encoding or **One-Hot** for one-hot encoding. The default is **Auto**, which "Selects the needed optimization algorithms during the synthesis process."

To determine which encoding style Synthesize—XST chose for the BUD_counter design in Listing 9.1 for the default encoding style Auto, click **Project → Σ Design Summary/Reports**, or click on the icon Σ and look at the logic utilization summary table and observe the row labeled "Number of Slice Flip Flops." The row lists 2 Used and 9,312 Available for the Spartan 3E on the NEXYS 2 board that we used to run out test. Because there are four states and only two flip-flops used, the encoding style is compact.

To automatically change the BUD_counter design in Listing 9.1 to a one-hot encoding style via the Xilinx ISE software tool, right click on **Synthesis → Process Properties → HDL Options → FSM Encoding Algorithm**, then select **One-Hot**. Rerun Synthesize—XST, and observe the row labeled "Number of Slice Flip Flops." The row lists 4 Used and 9,312 Available for the Spartan 3E on the NEXYS 2 board that we used to run out test. Because there are four states and four flip-flops used, the encoding style is one-hot. Change the encoding style back to Auto if you elected to run the test.

The logic utilization summary table also shows the number of LUTs (Look-up tables) used in a design. The LUTs provide the logic gates for FPGA designs.

The Σ **Design Summary/Reports** allows a user to compare the utilization of the programmable logic area that will be used on the chip called the **fabric** for different encoding styles.

To observe the circuit diagram for the design in Listing 9.1, either click **View RTL (Register Transfer Level) Schematic**, or click **View Technology Schematic**, which is under Synthesize—XST, and follow the instructions on the screen.

9.4 SUMMARY OF FINITE STATE MACHINE MODELS

Figure 9.5 shows a summary of six different finite state machine models. A *SET*, *RST*, or *INIT* input is not shown for the flip-flops in Figure 9.5 because each state machine model may be designed with a *SET* input, a *RST* input, or *INIT* input—that is, the required input is provided in the design specification.

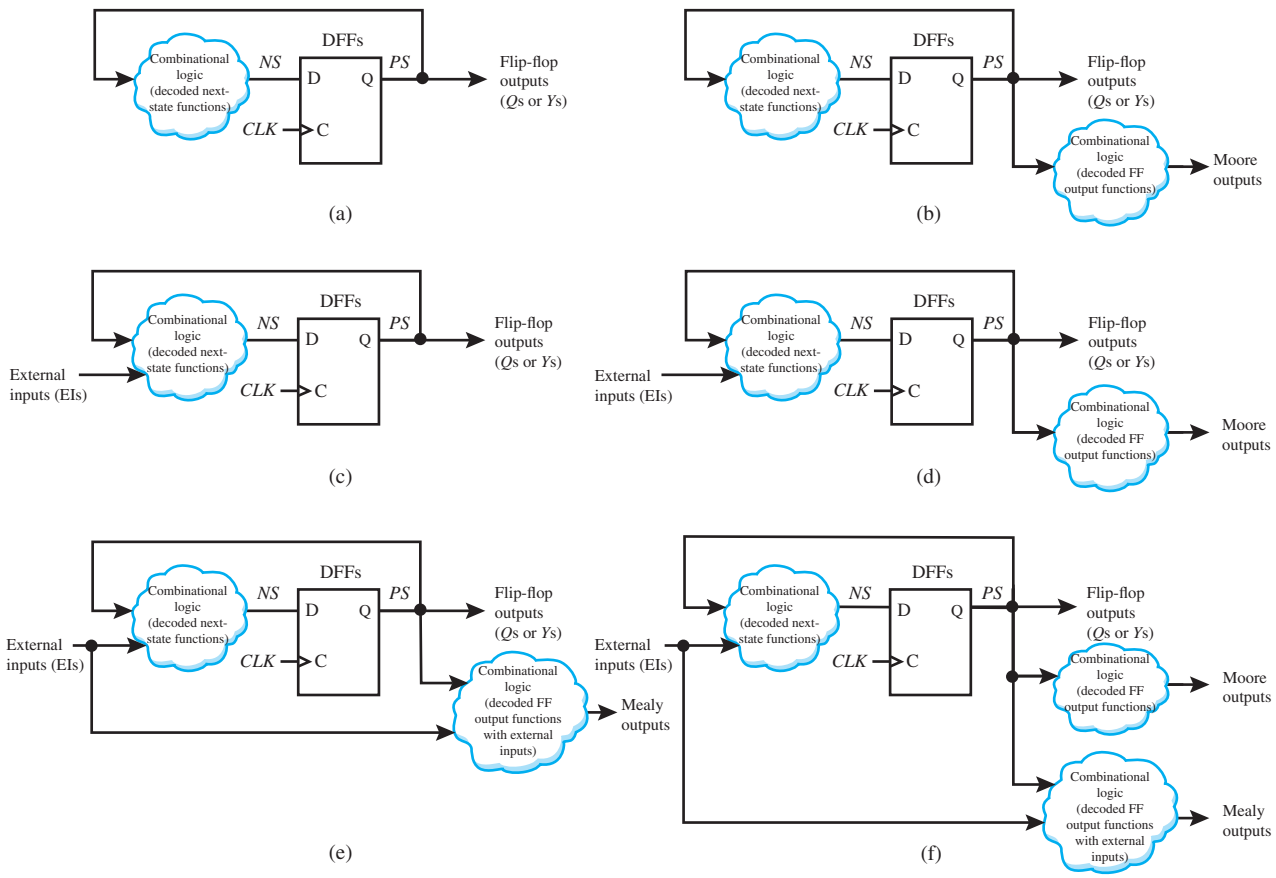


FIGURE 9.5 Finite state machine models: (a) simple FSM; (b) simple FSM with Moore Outputs; (c) complex FSM; (d) complex FSM with Moore outputs; (e) complex FSM with Mealy outputs; (f) complex FSM with Moore and Mealy outputs

The state machine models in Figures 9.5a and 9.5c only have flip-flop outputs. State machine models with Moore outputs (see Figures 9.5b and 9.5d) are referred to as Moore-type state machines in honor of Edward Moore, a famous pioneer in sequential design in the 1950s. Notice that the Moore outputs are decoded functions of only the flip-flop outputs—that is, Moore outputs are state dependent. For example, a Moore output signal $Z_{Moore} = Q1 \cdot \overline{Q2}$ would be a valid output for a state machine with 2 or more bits.

The state machine model with Mealy outputs (see Figure 9.5e) is referred to as a Mealy-type state machine in honor of G. H. Mealy, another famous pioneer in sequential design, also in the 1950s. Notice that the Mealy outputs are decoded functions of both the flip-flop outputs as well as the external inputs. For example, a Mealy output signal $Z_{Mealy} = Q1 \cdot \overline{Q2} \cdot UP$ would be a valid output for a state machine with 2 or more bits with an external input signal called *UP*; Mealy outputs are transition dependent.

The state machine model in Figure 9.5f contains both Moore and Mealy outputs.

9.5 DESIGNING COMPACT ENCODED STATE MACHINES WITH MOORE OUTPUTS

Figure 9.6 shows the state sequence diagram (or counting sequence diagram) and an equivalent state diagram for a Moore type complex state machine (a compact encoded stoppable Johnson counter (2 bits)). The counting sequence for a Johnson counter is marching 1s, as shown in the state sequence diagram and also in the state diagram. The external input *STOP* is used to change the state sequence. Suppose we wish to know when the machine is in state a and also when the machine is in state c. Moore outputs *Z*1 and *Z*2 are the signals used to provide that information. *Z*1 is the decoded FF output that indicates when the machine is in state c, while *Z*2 is the decoded FF output that indicates when the machine is in state a. Keep in mind that Moore outputs are always shown inside the state bubbles in a state diagram.

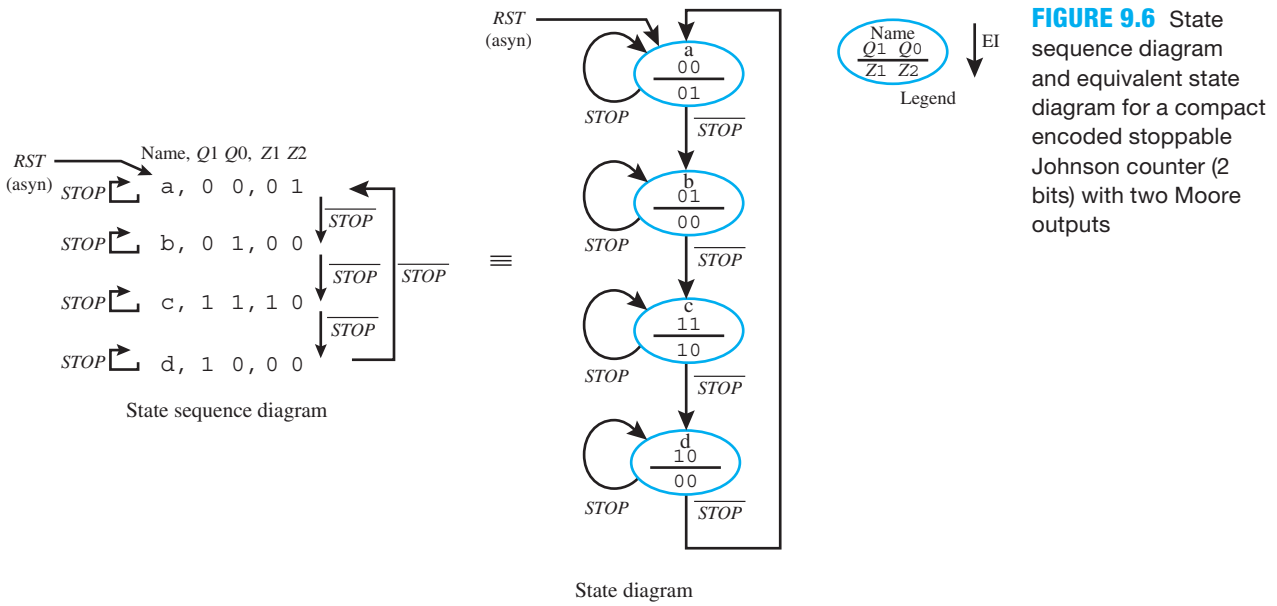


FIGURE 9.6 State sequence diagram and equivalent state diagram for a compact encoded stoppable Johnson counter (2 bits) with two Moore outputs

You should observe that the counting sequence of a Johnson counter (2 bits) is exactly the same as a binary Gray code up counter (2 bits). Do you think the counting sequence is the same for these two counters with a higher number of bits, say, 3 bits?

The VHDL code for the compact encoded stoppable Johnson counter (2 bits) represented by the state sequence diagram or state diagram in Figure 9.6 is shown in Listing 9.2 using the two-process PS/NS method.

LISTING 9.2
Complete VHDL design entity for Johnson_counter (compact encoded stoppable Johnson counter, 2 bits) (project: Johnson_counter)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Johnson_counter is port (
   rst,clk,stop : in std_logic;
   q : out std_logic_vector (1 downto 0);
   z : out std_logic_vector (1 to 2)
   );
end Johnson_counter;

architecture behavioral of Johnson_counter is
   type state_type is (a,b,c,d);
   signal ps, ns: state_type;
begin

sync_proc:
process (rst,clk)
begin
   if rst = '1' then ps <= a;
   elsif rising_edge (clk) then ps <= ns;
   end if;
end process;

comb_proc:
process (ps,stop)
begin
   z <= "00";--default values for Moore outputs for case statement
   case ps is
      when a => q <= "00"; z <= "01"; if stop = '1' then ns <= a;
                                      else ns <= b;
                                      end if;
      when b => q <= "01"; if stop = '1' then ns <= b;
                           else ns <= c;
                           end if;
      when c => q <= "11"; z <= "10"; if stop = '1' then ns <= c;
                                      else ns <= d;
                                      end if;
      when d => q <= "10"; if stop = '1' then ns <= d;
                           else ns <= a;
                           end if;
   end case;
end process;
end behavioral;
```

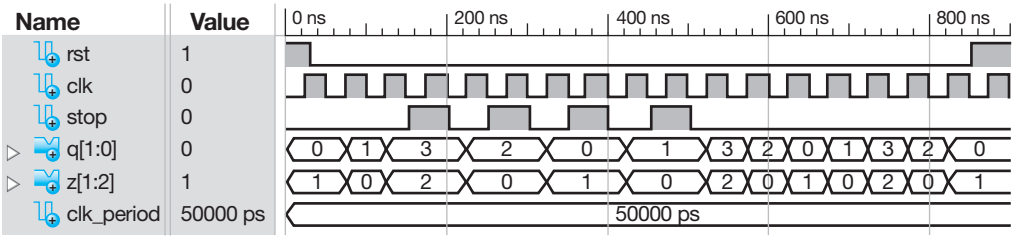
Things you should notice about the VHDL design in Listing 9.2:

- The Moore outputs $Z(1)$ and $Z(2)$ are declared in the entity declaration as mode **out** because Z only needs to be written (assigned a value), but not read. The Moore outputs are also listed as data type std_logic_vector (1 **to** 2).
- The combinational process (comb_proc) contains all the inputs in its sensitivity list that must be read—that is, (*PS*, *STOP*), in the combinational process.
- Get in the habit of including **default values** for Moore and/or Mealy outputs that are used in the case statement for the combinational process. The default values are placed before

the case statement in the combinational process (comb_proc). Default values help ensure that inferred latches will not be generated for these outputs if you write incomplete case statements or if statements.

- The case statement with the select input *PS* in the combinational process (comb_proc) is written to precisely follow the description specified by the state sequence diagram or state diagram. Either if statements or additional case statements can be used to describe the actions required by the state diagram.
- The flip-flop output values for *Q* and the Moore output values for *Z* are assigned for each state a through d in the case statement. Observe that flip-flop outputs and Moore outputs *are not placed within an if statement*.
- The *NS* (next state) is assigned its state value based on the external input signal *STOP*—that is, if STOP is true then *NS* maintains the same state, but if *STOP* is false then *NS* is assigned the next-state value in the state sequence diagram or state diagram.

Waveform 9.2 shows the correct functionality of design entity Johnson_counter (compact encoded stoppable Johnson counter, 2 bits) for Listing 9.2.



WAVEFORM 9.2
Simulation for the correct functionality of design entity Johnson_counter for Listing 9.2

An alternate method for designing the state machine in Figure 9.6 is to design it as a one-hot encoded state machine, then add two additional Moore outputs to provide the required marching 1s counting sequence.

9.6 DESIGNING ONE-HOT ENCODED STATE MACHINES WITH MOORE OUTPUTS

Figure 9.7 shows a one-hot encoded version of the state machine in Figure 9.6. The flip-flop outputs are *Y*3, *Y*2, *Y*1, and *Y*0. The Moore outputs *Z*1 and *Z*2 are the same as in Figure 9.6. Two additional Moore outputs *Q*1 and *Q*0 have been added to provide the required marching 1s counting sequence.

The VHDL code for the one-hot encoded stoppable Johnson counter (4 bits) represented by the state diagram in Figure 9.7 is shown in Listing 9.3 using the two-process PS/NS method.

Things you should notice about the VHDL design in Listing 9.3:

- The following changes were made in the entity declaration: (1) the entity name was changed to OH_JC; (2) the signal *RST* was changed to *INIT* (initialize); and (3) the state variable *Y* is declared as mode **out** with a data type of std_logic_vector (3 downto 0). Output *Q* was the state variable for the compact encoded counter, but for this one-hot encoded counter it provides the Moore outputs—that is, the decoded FF outputs.
- In the synchronous process (sync_proc), the input *RST* (reset) was changed to *INIT* (initialize) in the sensitivity list. Only the inputs *INIT* and *CLK* are required in the sensitivity list of the synchronous process (sync_proc), because the process only generates D flip-flops, which are completely controlled by just these two inputs.
- In the synchronous process (sync_proc), the signal *INIT* (initialize) is used to asynchronously initialize the counter to state a.

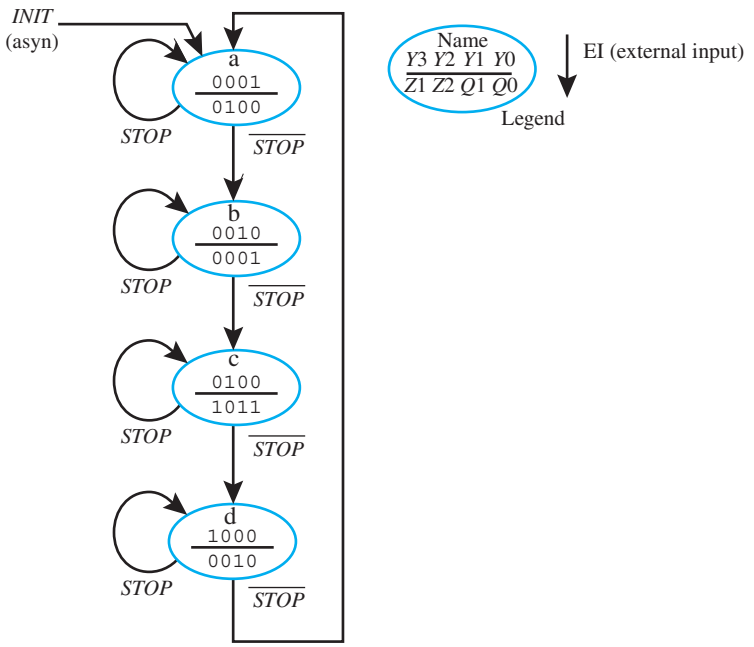


FIGURE 9.7 State diagram for a one-hot encoded stoppable Johnson counter (4 bits) with four Moore outputs

LISTING 9.3
Complete VHDL design entity for OH_JC (one-hot encoded stoppable Johnson counter, 4 bits) (project: OH_JC)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OH_JC is port (
   init,clk,stop : in std_logic;
   y : out std_logic_vector (3 downto 0);
   z : out std_logic_vector (1 to 2);
   q : out std_logic_vector (1 downto 0)
   );
end OH_JC;

architecture behavioral of OH_JC is
   type state_type is (a,b,c,d);
   signal ps, ns: state_type;
begin

sync_proc:
process (init,clk)
begin
   if init = '1' then ps <= a;
   elsif rising_edge (clk) then ps <= ns;
   end if;
end process;

comb_proc:
process (ps,stop)
begin
   z <= "00"; q <= "00"; --default values for Moore outputs for
                          case statement
```

```
   case ps is
      when a => y <= "0001"; z <= "01"; if stop = '1' then ns <= a;
                                        else ns <= b;
                                        end if;
      when b => y <= "0010"; q <= "01"; if stop = '1' then ns <= b;
                                        else ns <= c;
                                        end if;
      when c => y <= "0100"; q <= "11"; z <= "10";
                                        if stop = '1' then ns <= c;
                                        else ns <= d;
                                        end if;
      when d => y <= "1000"; q <= "10"; if stop = '1' then ns <= d;
                                        else ns <= a;
                                        end if;
   end case;
end process;
end behavioral;
```

- In the combinational process (comb_proc), the state *Y* is assigned the one-hot code shown in the state diagram in Figure 9.7. Observe that the output value of *Q* provides the marching 1s counting sequence.
- No other changes were required—that is, the rest of the code is identical to the VHDL code in Listing 9.2.

The reason the designs in Listings 9.2 and 9.3 are only slightly different is because the two-process PS/NS method uses an enumerated type for the state names. An enumerated type allows the binary values of the states to be easily changed from compact encoding to one-hot encoding.

The purpose of the design in Figure 9.7 was to provide the same identical outputs as those provided in Figure 9.6 with a one-hot encoding style rather than a full encoding style. The major difference was the addition of the flip-flop outputs *Y*3 through *Y*0. The flip-flop outputs can be removed from the combinational process and provided via a selected signal assignment (SSA) as shown in Listing 9.4.

LISTING 9.4
Complete VHDL design entity for OH_JC_MOD (one-hot encoded stoppable Johnson counter, 4 bits, with a selected signal assignment) (project: OH_JC_MOD)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity OH_JC_MOD is port (
   init,clk,stop : in std_logic;
   y : out std_logic_vector (3 downto 0);
   z : out std_logic_vector (1 to 2);
   q : out std_logic_vector (1 downto 0)
   );
end OH_JC_MOD;

architecture mixed of OH_JC_MOD is
   type state_type is (a,b,c,d);
   signal ps, ns : state_type;
begin
sync_proc:
process (init,clk)
```

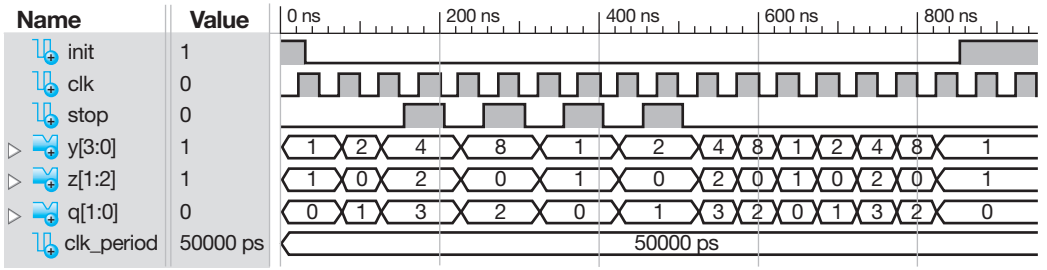
(Continued)

```
begin
   if init = '1' then ps <= a;
   elsif rising_edge (clk) then ps <= ns;
   end if;
end process;
comb_proc:
process (ps,stop)
begin
   z <= "00"; q <= "00"; --default values for Moore outputs for
                           case statement
   case ps is
      when a => z <= "01"; if stop = '1' then ns <= a;
                           else ns <= b;
                           end if;
      when b => q <= "01"; if stop = '1' then ns <= b;
                           else ns <= c;
                           end if;
      when c => q <= "11"; z <= "10"; if stop = '1' then ns <= c;
                                      else ns <= d;
                                      end if;
      when d => q <= "10"; if stop = '1' then ns <= d;
                           else ns <= a;
                           end if;
   end case;
end process;

with ps select
   y <= "0001" when a,
        "0010" when b,
        "0100" when c,
        "1000" when d;
end mixed;
```

Things you should notice about the VHDL design in Listing 9.4:

- The state variable *Y* or flip-flop outputs have been removed from the case statement in the combinational process (comb_proc). This tends to make the combinational process easier to write because the flip-flop output values need not be considered until later.
- The state variable *Y* or flip-flop outputs are provided in the selected signal assignment (SSA) after the combinational process. This tends to make the flip-flop output values easier to write because they are kept together in one selected signal assignment statement. Observe that a **when others** clause is not required as the last choice value in the SSA because the data type of the select signal ps is state_type, and state_type only has the values of a, b, c, and d and no other values.
- Observe that this design is a mixed architecture description because it contains both a behavioral design style and a dataflow design style in the same architecture description.
- This is an alternate way of providing the state variable (or flip-flop) output values. If we decide later that we do not want to provide the state variable output values, we can simply comment out the selected signal assignment (SSA) without altering the combinational process (comb_proc).

Waveform 9.3 shows the correct functionality of design entities OH_JC and OH_JC_MOD for Listings 9.3 and 9.4.



WAVEFORM 9.3 Simulation for the correct functionality of design entities OH_JC and OH_JC_MOD for Listings 9.3 and 9.4

The simulations are the same for the VHDL code in Listings 9.3 and 9.4. Observe that the *Y* outputs follow a one-hot sequence; the *Z*1 and *Z*2 outputs are active only in states c and a, respectively; and the *Q* outputs follow the Johnson counter marching 1s sequence—that is, marching from right to left, as shown in the state diagram in Figure 9.7.

9.7 DESIGNING COMPACT ENCODED STATE MACHINES WITH MOORE AND MEALY OUTPUTS

Figure 9.8 shows the state diagram for a complex state machine that contains both Moore and Mealy outputs (a compact encoded shortened binary down [SBD] counter [2 bits]). The synchronized external input signal *SHORTEN* is used to change the counting sequence from 11, 10, 01, 00 to just 11, 10, 01. Suppose we wish to know when the sequence is being shortened. Mealy output *Z*1 is the signal used to provide that information. Keep in mind that Mealy outputs are always shown with the external inputs beside the transition lines in a state diagram. Suppose we also wish to know when the machine uses the entire sequence. Moore output *Z*2 is the signal used to provide that information.

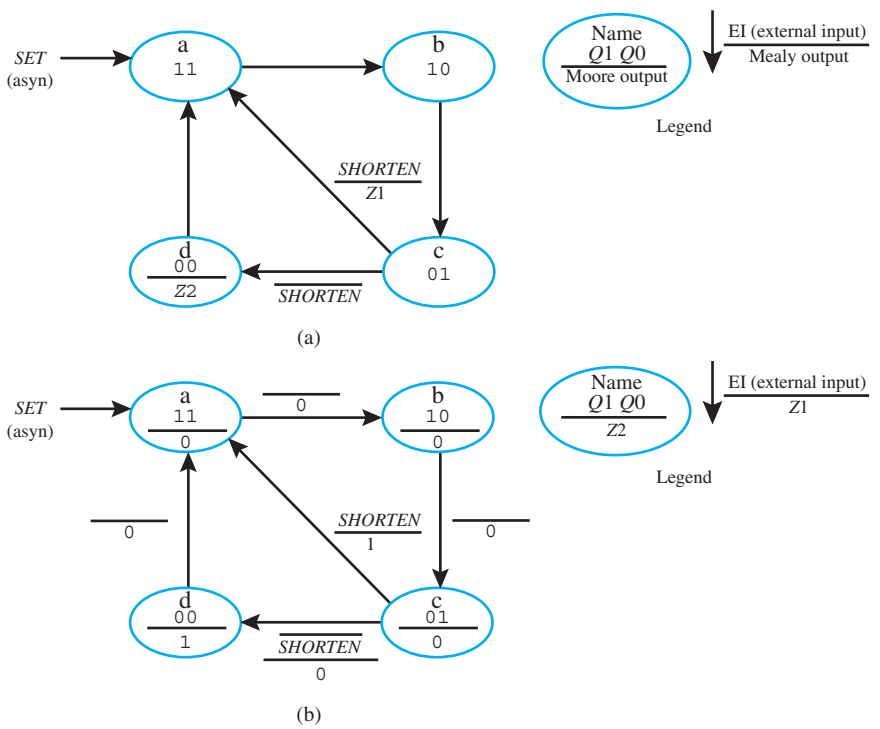


FIGURE 9.8 State diagram for a compact encoded shortened binary down (SBD) counter (2 bits) with both Moore and Mealy outputs; (a) using the Show Where True signal convention for Moore and Mealy outputs; (b) using the Show All Values signal convention for Moore and Mealy outputs

In Figure 9.8a, Mealy output $Z1$ is true—that is, $Z1 = 1$, when the machine is in state c and *SHORTEN* is true. For all other transitions $Z1$ is false—that is, $Z1 = 0$. Moore output $Z2$ is true—that is, $Z2 = 1$—in state d and false—that is, $Z2 = 0$—in all other states. We refer to this signal convention as the **Show Where True** signal convention. This convention helps simplify a state diagram because the Mealy and Moore outputs for all the false values need not be shown. Signals without overbars in state diagrams are active high signals, while signals with overbars are active low signals. $Z1$ is an active high signal because it does not have an overbar. $Z2$ is also an active high signal because it does not have an overbar.

An active low output signal $\overline{Z}_{(Moore)}$ can be used in a state diagram. Everywhere the output $\overline{Z}_{(Moore)}$ would be shown in the state diagram, $\overline{Z}_{(Moore)} = 1$ would be true, and for all other states it would be false—that is, $\overline{Z}_{(Moore)} = 0$. An active low output signal $\overline{Z}_{(Mealy)}$ can also be used in a state diagram. Everywhere the output $\overline{Z}_{(Mealy)}$ would be shown in the state diagram, $\overline{Z}_{(Mealy)} = 1$ would be true, and for all other states it would be false—that is, $\overline{Z}_{(Mealy)} = 0$. Recall that $\overline{X} = 1$ means that $X = 0$ via Boolean algebra, and $\overline{X} = 0$ means that $X = 1$.

Figure 9.8b shows an alternate signal convention for listing the Moore and Mealy output values in terms of 1s and 0s. We refer to the alternate signal convention as **Show All Values** signal convention. This signal convention shows all signals at a glance but is more detailed.

The VHDL code for the compact encoded SBD counter (2 bits) in Figure 9.8 is shown in Listing 9.5 using the two-process PS/NS method.

LISTING 9.5
Complete VHDL design entity for SBD_counter (compact encoded shortened binary down counter, 2 bits) (project: SBD_counter)

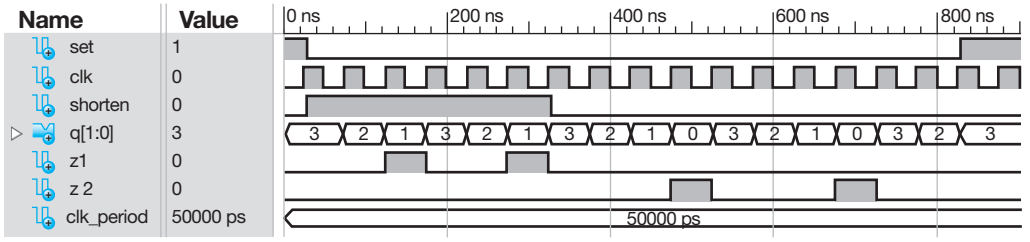
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SBD_counter is port (
   set,clk, shorten : in std_logic;
   q : out std_logic_vector (1 downto 0);
   z1, z2 : out std_logic
   );
end SBD_counter;

architecture behavioral of SBD_counter is
   type state_type is (a,b,c,d);
   signal ps, ns: state_type;
begin
sync_proc:
process (set,clk)
begin
   if set = '1' then ps <= a;
   elsif rising_edge (clk) then ps <= ns;
   end if;
end process;
comb_proc:
process (ps,shorten)
begin
   z1 <= '0'; z2 <= '0'; --default values for Moore and Mealy
                            outputs for case statement
   case ps is
      when a => q <= "11"; ns <= b;
      when b => q <= "10"; ns <= c;
```

```
      when c => q <= "01"; if shorten = '1' then z1 <= '1';
                                                 ns <= a;
                           else ns <= d;
                           end if;
      when d => q <= "00"; z2 <= '1'; ns <= a;
   end case;
end process;
end behavioral;
```

Things you should notice about the VHDL design in Listing 9.5:

- State variable *Q* of data type std_logic_vector (1 downto 0) provides the flip-flop outputs.
- Output *Z*1 of data type std_logic provides the Mealy output—that is, the decoded FF output with the external input *SHORTEN*.
- Output *Z*2 of data type std_logic provides the Moore output—that is, the decoded FF output in state d.
- Because the *SET* input is asynchronous, **if** set = '1' **then** *PS* <= a is placed before rising_edge (clk) in the synchronous process (sync_proc).
- The combinational process (comb_proc) contains all the inputs in its sensitivity list that must be read—that is, (*PS*, *SHORTEN*), in the combinational process.
- The case statement (with accompanying if statements) in the combinational process (comb_proc) is written to precisely follow the description specified by the state diagram.
- Observe that the Mealy output *Z*1 is placed within or inside an if statement, because the Mealy output is a function of the external input *SHORTEN*, which is tested by an if statement.

Waveform 9.4 shows the correct functionality of design entity SBD_counter (compact encoded shortened binary down counter, 2 bits) for Listing 9.5.



WAVEFORM 9.4
Simulation for the correct functionality of design entity SBD_counter for Listing 9.5

Observe that the counter follows the state diagram in Figure 9.8 as expected. When *SHORTEN* is active or 1, *Q* follows the sequence 3, 2, 1, 3, 2, 1 . . . , and when *SHORTEN* is inactive or 0, *Q* follows the sequence 3, 2, 1, 0, 3, 2, 1, 0 . . . . *Z*1 is only active when *SHORTEN* is active and the counter is in state c. *Z*2 is only active when the counter is in state d, as shown in Figure 9.8.

9.8 DESIGNING ONE-HOT ENCODED STATE MACHINES WITH MOORE AND MEALY OUTPUTS

Figure 9.9 shows a one-hot version of the state machine in Figure 9.8. The flip-flop outputs are *Y*3 *Y*2 *Y*1 and *Y*0. The Mealy output *Z*1 and Moore output *Z*2 are the same as in Figure 9.8. Two additional Moore outputs *Q*1 and *Q*0 have been added to provide the binary down counting sequence.

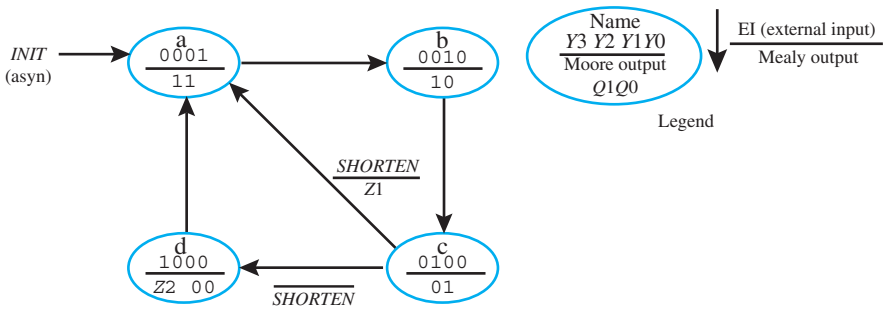


FIGURE 9.9 State diagram for a one-hot shortened binary down counter (4 bits) with both Moore and Mealy outputs

The VHDL code for the one-hot shortened binary down counter (OH_SBD_ counter) (4 bits) in Figure 9.9 is shown in Listing 9.6 using the two-process PS/NS method.

LISTING 9.6 Complete VHDL design entity for OH_SBD_counter (one-hot shortened binary down counter, 4 bits) (project: OH_SBD_counter)

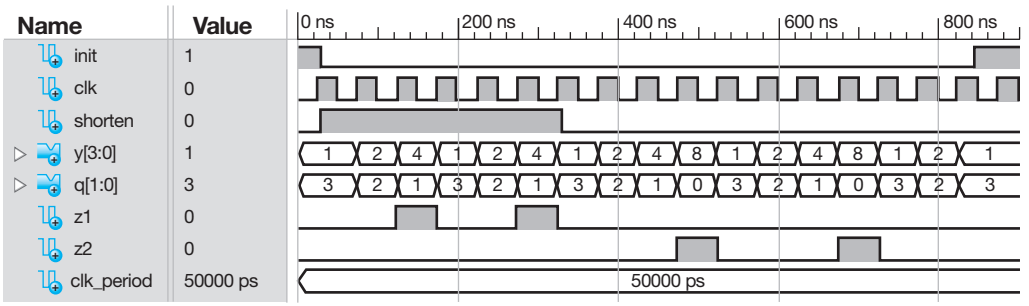
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OH_SBD_counter is port (
   init,clk,shorten : in std_logic;
   y : out std_logic_vector (3 downto 0);
   q : out std_logic_vector (1 downto 0);
   z1, z2 : out std_logic
   );
end OH_SBD_counter;

architecture mixed of OH_SBD_counter is
   type state_type is (a,b,c,d);
   signal ps, ns : state_type;
begin

sync_proc:
process (init,clk)
begin
   if init = '1' then ps <= a;
   elsif rising_edge (clk) then ps <= ns;
   end if;
end process;

comb_proc:
process (ps,shorten)
begin
   z1 <= '0'; z2 <= '0'; --default values for Moore and Mealy
                          outputs for case statement
   case ps is
      when a => q <= "11"; ns <= b;
      when b => q <= "10"; ns <= c;
      when c => q <= "01"; if shorten = '1' then z1 <= '1'; ns <= a;
                           else ns <= d;
                           end if;
      when d => q <= "00"; z2 <= '1'; ns <= a;
   end case;
end process;
```

```
with ps select
   y <= "0001" when a,
        "0010" when b,
        "0100" when c,
        "1000" when d;
end mixed;
```

Things you should notice about the VHDL design in Listing 9.6:

- The signal *SET* was changed to *INIT* (initialize) in the entity declaration in Listing 9.5.
- The state variable *Y* or flip-flop outputs are declared as external signals with a data type of std_logic_vector (3 downto 0). Only external signals show up in a simulation of the design.
- In the synchronous process (sync_proc), the signal *SET* was changed to *INIT* in the sensitivity list. Only the inputs *INIT* and *CLK* are required in the sensitivity list of the synchronous process (sync_proc), because the process only generates D flip-flops, which are completely controlled by just these two inputs.
- In the synchronous process (sync_proc), the signal *INIT* (initialize) is used to asynchronously initialize the counter to State a.
- No changes were made in the combinational process (comb_proc).
- The value of *Y*, which is the state variable, is assigned the one-hot code shown in the state diagram in Figure 9.9 in a selected signal assignment.

Waveform 9.5 shows the correct functionality of design entity OH_SBD_counter (one-hot shortened binary down counter, 4 bits) for Listing 9.6.



WAVEFORM 9.5
Simulation for the correct functionality of design entity OH_SBD_counter for Listing 9.6

If signal *Y*(3:0) is removed from the timing diagram in Waveform 9.5, then the remaining waveform signals provide the correct waveform for Figure 9.8, which uses compact encoding for the flip-flop outputs with the exception of the signal *INIT*. The signal *INIT* was used to initialize the one-hot encoded counter to state 0001, while the signal *SET* was used to initialize the compact encoded counter to state 11.

9.9 USING THE ALGORITHMIC EQUATION METHOD TO DESIGN COMPLEX STATE MACHINES

You should be able to draw the D flip-flops and the clouds of combinational logic for the decoded next-state functions for the circuit of a complex state machine using the **algorithmic equation (AE) method**. The clouds of combinational logic can easily be obtained by using a modified form of the Set OR Hold 1 equation that includes the external inputs for the design.

The algorithm for the **Set OR Hold 1 method** for a complex state machine can be expressed by the following **Set OR Hold 1 equation**:

$D = \Sigma$(PS expression for a set transition)·(External input condition)
$+ \Sigma$(PS expression for a hold 1 transition)·(External input condition)

Always remember when using the Set OR Hold 1 equation that the D in the equation represents the next-state value, which is Q^+ or $Q(NS)$.

Observe that the Set OR Hold 1 equation has been modified to include the external input condition that may occur for each set or hold 1 transition. If an external input condition occurs for a set or hold 1 transition, the condition must be ANDed, hence the "·," with the PS expression. If an external input condition such as $SHORTEN$ or $\overline{SHORTEN}$ does not appear beside a transition line in a state sequence or state diagram, then the external input condition in the Set OR Hold 1 equation is simply ignored, because $SHORTEN + \overline{SHORTEN} = 1$, which indicates that no external input condition is required to change to the next state.

The D excitation equations can be written by inspection using the state sequence diagram or the state diagram for the design.

Once the D excitation equations are obtained, you can draw the complete circuit diagram for the complex state machine. You can also use the D excitation equations to write the VHDL code for the design. If the state diagram has Moore or Mealy outputs, you must also write the equations for these outputs and include the equations in the circuit diagram and the VHDL code.

The disadvantage of using the algorithmic equation method for D flip-flops compared to using the two-process PS/NS method is the hassle of obtaining the correct D excitation equations and drawing the circuit diagram.

Figure 9.10 shows a modified form of the state diagram in Figure 9.9. Observe that the flip-flop output signals rather than the flip-flop output signal values are place in each state, which provides a more compact state machine design description—especially for larger one-hot encoded designs.

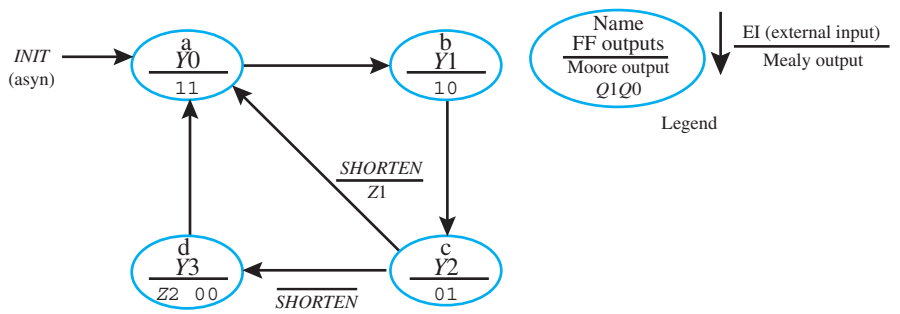


FIGURE 9.10 State diagram for a one-hot shortened binary down counter (4 bits) with both Moore and Mealy outputs

Using the Set OR Hold 1 equation for a complex state machine, we can write the excitation equation for the $D3$ input as follows:

$D3 =$ (PS expression for a $0 \rightarrow 1$ transition from state c to state d)·$\overline{SHORTEN}$

which results in $D3 = Y2 \cdot \overline{SHORTEN}$.

The excitation equation for the $D2$ is written as follows:

$D2 =$ (PS expression for a $0 \rightarrow 1$ transition from state b to state c)

which results in $D2 = Y1$

The excitation equation for the $D1$ is written as follows:

$D1 =$ (PS expression for a $0 \rightarrow 1$ transition from state a to state b)

which results in $D1 = Y0$

The excitation equation for the $D0$ is written as follows:

$D0 =$ (*PS* expression for a $0 \rightarrow 1$ transition from state d to state a)
$+$ (*PS* expression for a $0 \rightarrow 1$ transition from state c to state a)$\cdot$*SHORTEN*

which results in $D0 = Y3 + Y2 \cdot SHORTEN$

You might observe that flip-flop output signals (*Y3*, *Y2*, *Y1*, and *Y0*) in each of the *D* excitation equations are always noninverted—that is, they do not contain overbars. If you remember this fact, this may help you catch errors when you write the *D* excitation equations. Also observe that each PS expression for one-hot encoding only requires one state variable, for each Set transition or each Hold 1 transition.

The equations for the Moore outputs for $Q1$ and $Q0$ can be written by inspection from the state diagram:

$Q1 = Y0 + Y1$
$Q0 = Y0 + Y2$

The Mealy output Z1 and the Moore output Z2 can also be written by inspection from the state diagram:

$Z1 = Y2 \cdot SHORTEN$
$Z2 = Y3$

Figure 9.11a shows the circuit model for the one-hot shortened binary down counter, while Figure 9.11b shows the final circuit for the one-hot shortened binary down (4 bits). The *D* excitation equations in the final circuit provide the signals for the next state (NS) shown in the circuit model.

The VHDL code for the one-hot shortened binary down counter in Figure 9.11b is shown in Listing 9.7 using a dataflow design style.

LISTING 9.7
Complete VHDL design entity for OH_SBD_counter_AEM (4 bits) using a dataflow design style (project: OH_SBD_counter_AEM)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OH_SBD_counter_AEM is port (
     init,clk,shorten : in std_logic;
     y : inout std_logic_vector (3 downto 0);
     q : out std_logic_vector (1 downto 0);
     z1,z2 : out std_logic
     );
end OH_SBD_counter_AEM;

architecture dataflow of OH_SBD_counter_AEM is
     signal d3,d2,d1,d0: std_logic;
begin
     d3 <= y(2) and not shorten;
     d2 <= y(1);
     d1 <= y(0);
     d0 <= y(3) or (y(2) and shorten);

     y <= "0001"       when init = '1' else
          (d3&d2&d1&d0) when rising_edge (clk);

     q(1) <= y(0) or y(1);
     q(0) <= y(0) or y(2);
```

(Continued)

```
    z1 <= y(2) and shorten;
    z2 <= y(3);
end dataflow;
```

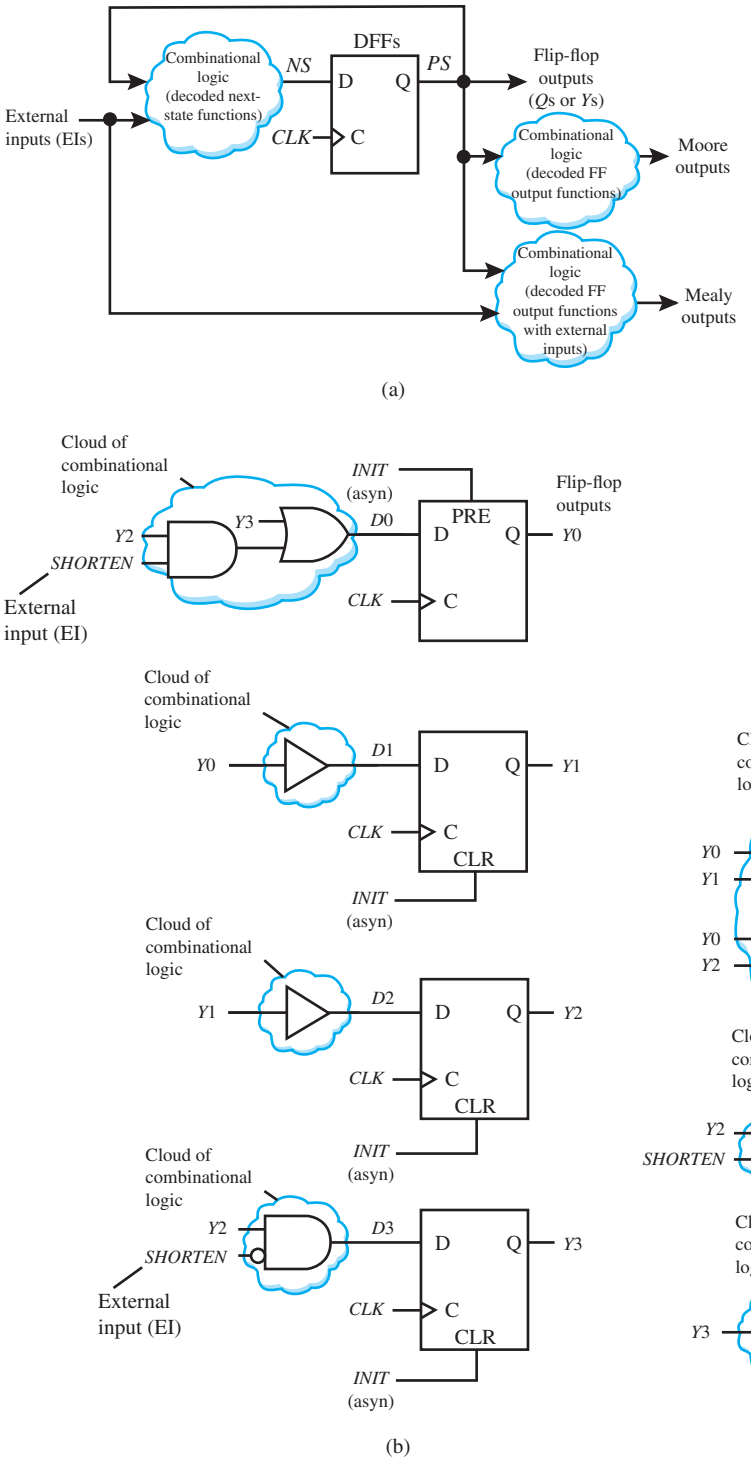


FIGURE 9.11 (a) Circuit model for the one-hot shortened binary down counter; (b) the final circuit for the one-hot shortened binary down counter (4 bits)

Things you should notice about the VHDL design in Listing 9.7:

- *Y* must be assigned the mode **inout**.
- Signals *D*3, *D*2, *D*1, and *D*0 are internal signals and are declared between **architecture** and the first **begin**.
- The excitation inputs *D*3, *D*2, *D*1, and *D*0 are written using Boolean equations.
- A dataflow design style is used in the architecture declaration.
- The concatenation operator "&"is used to assign the *D* excitation signals as shown in the expression (*D*3&*D*2&*D*21&*D*0) to *Y* in the conditional signal assignment. The order of the elements in the concatenation expression is very important. The most significant element must be placed on the left, because the elements for *Y* are declared as *Y*(3 downto 0). An alternate way to handle the *D* excitation signals is to declare *D* as a std_logic_vector with the range (3 downto 0). This will change the individual equations from *D*3 to *D*(3), *D*2 to *D*(2), and so on.
- Notice that the VHDL code has fewer lines than the VHDL code required for the two-process PS/NS method. For large complex state machine designs, it is generally much easier to use the two-process PS/NS method than the algorithmic equation method due to the hassle of obtaining the *D* excitation equations and drawing the circuit diagram prior to writing the VHDL code.

The simulation for Listing 9.7 is the same as Waveform 9.5 obtained earlier. This proves that the algorithmic equation method and the two-process PS/NS method provide the same result.

Figure 9.12 shows the state diagram for the binary up/down counter shown earlier in Figure 9.2.

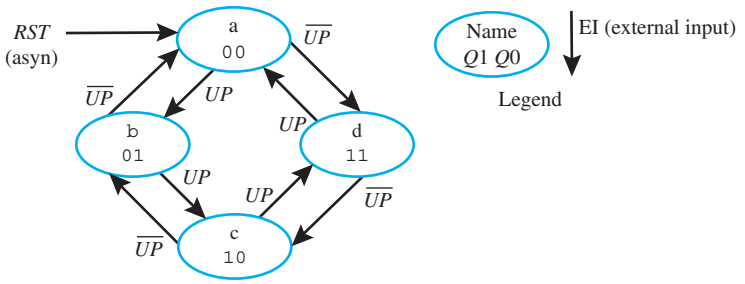


FIGURE 9.12 State diagram for the binary up/down counter (2 bits)

Using the Set OR Hold 1 equation for a complex state machine, we can write the excitation equation for the *D*1 input as follows:

$$\begin{aligned} D1 = {} & (PS \text{ expression for a } 0 \rightarrow 1 \text{ transition from state a to state d}) \cdot \overline{UP} \\ & + (PS \text{ expression for a } 1 \rightarrow 1 \text{ transition from state d to state c}) \cdot \overline{UP} \\ & + (PS \text{ expression for a } 1 \rightarrow 1 \text{ transition from state c to state d}) \cdot UP \\ & + (PS \text{ expression for a } 0 \rightarrow 1 \text{ transition from state b to state c}) \cdot UP \end{aligned}$$

which results in

$$\begin{aligned} D1 = {} & \overline{Q1} \cdot \overline{Q0} \cdot \overline{UP} + Q1 \cdot Q0 \cdot \overline{UP} \\ & + Q1 \cdot \overline{Q0} \cdot UP + \overline{Q1} \cdot Q0 \cdot UP \\ = {} & \overline{Q1 \oplus Q0} \cdot \overline{UP} + Q1 \oplus Q0 \cdot UP = \overline{Q1 \oplus Q0 \oplus UP} \end{aligned}$$

Although the *D*1 excitation equation cannot be minimized via Boolean algebra, it can be organized in terms of XOR operators (⊕ s) via a three-variable K-map. To see how this is done, first fill in an odd function in a three-variable K-map and observe how the function can be organized in terms of XOR, AND, and OR operators. Next fill in an even function in a three-variable K-map and observe how the function can be organized in terms of XOR, AND, and OR operators.

$D1$ is an even function. With a little practice, you can write odd and even functions just in terms of XOR operators.

The excitation equation for the $D0$ input can be written as follows:

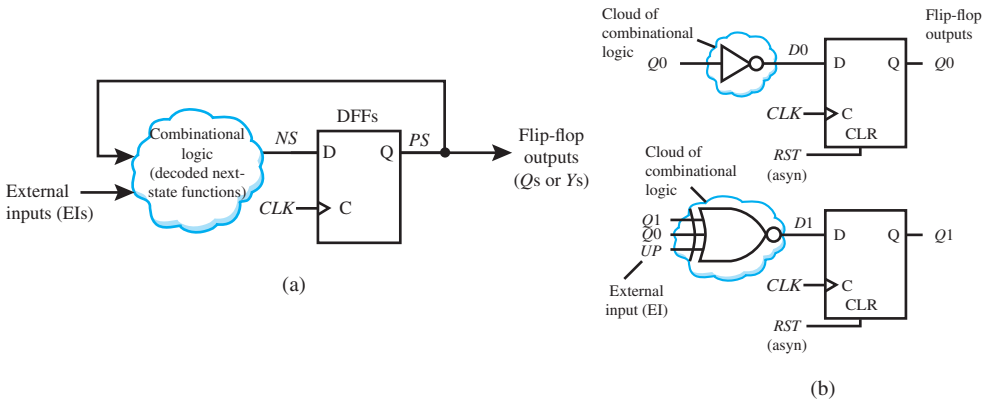
$$\begin{aligned} D0 = {} & (PS \text{ expression for a } 0 \rightarrow 1 \text{ transition from state a to state d}) \cdot \overline{UP} \\ & + (PS \text{ expression for a } 0 \rightarrow 1 \text{ transition from state c to state d}) \cdot UP \\ & + (PS \text{ expression for a } 0 \rightarrow 1 \text{ transition from state c to state b}) \cdot \overline{UP} \\ & + (PS \text{ expression for a } 0 \rightarrow 1 \text{ transition from state a to state b}) \cdot UP \end{aligned}$$

which results in

$$\begin{aligned} D0 = {} & \overline{Q1} \cdot \overline{Q0} \cdot \overline{UP} + Q1 \cdot \overline{Q0} \cdot UP \\ & + Q1 \cdot \overline{Q0} \cdot \overline{UP} + \overline{Q1} \cdot \overline{Q0} \cdot UP \\ = {} & \overline{Q0} \cdot \overline{UP} + \overline{Q0} \cdot UP = \overline{Q0} \end{aligned}$$

Figure 9.13a shows the circuit model for the binary up/down counter, while Figure 9.13b shows the final circuit for the binary up/down counter (2 bits). The D excitation equations in the final circuit provide the signals for the next state (NS) shown in the circuit model.

FIGURE 9.13
(a) Circuit model for the binary up/down counter; (b) the final circuit for the binary up/down counter (2 bits)



The VHDL code for the binary up/down counter in Figure 9.13b is shown in Listing 9.8 using a dataflow design style.

LISTING 9.8
Complete VHDL design entity for BUD_counter_AEM (2 bits) using a dataflow design style (project: BUD_counter_AEM)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity BUD_counter_AEM is port (
     rst,clk,up : in std_logic;
     q : inout std_logic_vector (1 downto 0)
     );
end BUD_counter_AEM;

architecture dataflow of BUD_counter_AEM is
     signal d0,d1: std_logic;
begin
     d1 <= not (q(1) xor q(0) xor up);
     d0 <= not q(0);
     q  <= "00" when rst = '1' else
          (d1,d0) when rising_edge (clk);
end dataflow;
```

Things you should notice about the VHDL design in Listing 9.8:

- *Q* must be assigned the mode **inout**.
- Signals *D*0 and *D*1 are internal signals and are declared between **architecture** and **begin**.
- The excitation inputs *D*0 and *D*1 are written using Boolean equations.
- A dataflow design style is used in the architecture declaration.
- The aggregate (*D*1,*D*0) is used to assign the *D* excitation signals to *Q* in the conditional signal assignment. The order of the elements in the aggregate is very important. The most significant element must be placed on the left, because the elements for *Q* are declared as *Q*(1 downto 0). An alternate way to handle the *D* excitation signals is to declare *D* as a std_logic_vector with the range (1 downto 0). This will change the individual equations from *D*1 to *D*(1) and *D*0 to *D*(0).
- Notice that the VHDL code has fewer lines than the VHDL code required for the two-process PS/NS method. For large complex state machine designs, it is generally much easier to use the two-process PS/NS method than the algorithmic equation method due to the hassle of obtaining the *D* excitation equations and drawing the circuit diagram prior to writing the VHDL code.

Using the algorithmic equation method, the *D* excitation equations are usually easier to write for one-hot encoded state machines than for compact encoded state machines, as you have observed in these last two examples. Compact encoded (or full encoded) designs have fewer *D* excitation equations, but they also contain more than one flip-flop output variable for each Set or each Hold 1 transition, which can either be inverted or noninverted. This makes the equations a little harder to write for compact encoded designs.

The simulation for Listing 9.8 is the same as Waveform 9.1 obtained earlier. This proves that the algorithmic equation method and the two-process PS/NS method provide the same result.

9.10 IMPROVING THE RELIABILITY OF COMPLEX STATE MACHINE DESIGNS

The output of a flip-flop has a high-voltage range, which is referred to as 1 or High, and a low-voltage range, which is referred to as 0 or Low. When the setup or hold time of a flip-flop is not met, the output of the flip-flop can go to a mid-voltage range between the high- and low-voltage range, which is undesirable. When the flip-flop is in the mid-voltage range, it is unstable or in a metastable state. The output of the flip-flop that is in a metastable state will eventually go to the stable state of 1 or to the stable state of 0. If the stable state it goes to is not the desired state, then the circuit will fail to perform as expected. The reliability of a circuit is improved by helping to keep flip-flops from going into a metastable state. To be a little more specific, operating a D flip-flop by changing its *D* and *CLK* inputs too close together can cause in an internal function hazard to occur that creates a runt pulse (a glitch), which can place the D flip-flop output in a metastable state.

Figure 9.14 illustrates how the *Q* output of a D flip-flop can go into a metastable state—and what happens afterwards—for a setup time violation.

When a flip-flop in a state machine goes into a metastable state, the circuit may fail to operate properly if its state sequence (or counting sequence) is interrupted. Metastability can occur in a complex state machine when a *D* input, of one of its internal D flip-flops, does not meet setup or hold time (or the *D* input changes during the data sampling time), thus forcing the flip-flop into a metastable state, as shown in Figure 9.14 for a setup time violation. The interruption occurs for the period of time t_m that the flip-flop is in the metastable state.

It is not possible to prevent metastability from occurring, but it is possible to reduce metastability. Metastability is an event that doesn't happen very often, but when it does the circuit

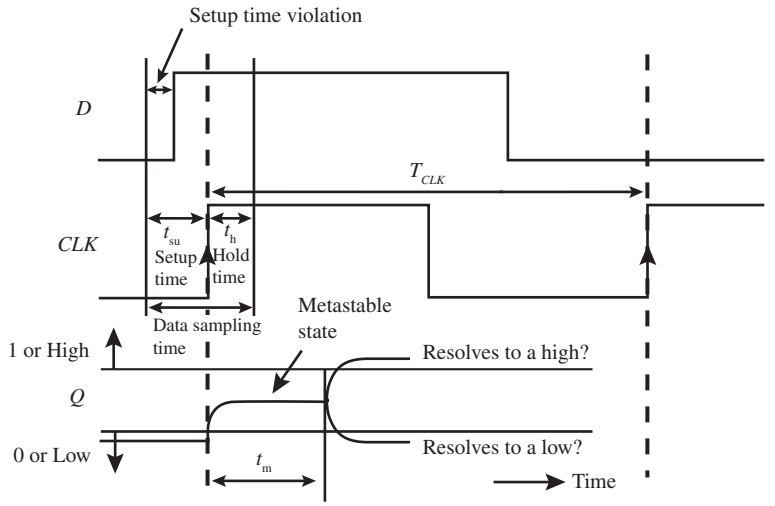


FIGURE 9.14 *Q* output of D flip-flops going into a metastable state

fails to perform correctly. The time that it takes for a circuit to resolve itself has been shown to decrease exponentially with time t—that is, $k_1 e^{-k_2 t}$, where k_1 and k_2 are empirically determined.

For the setup time violation shown in Figure 9.14, if $t_m < T_{CLK} - t_{su}$ a problem occurs only if the D flip-flop resolves itself to a 0 after the metastability state—that is, it fails to detect the 1 due to the setup time violation. This may cause the D flip-flop to miss one of its counting states, thus causing the circuit to fail. If the flip-flop resolves itself to a 1, then it detected the 1 even though it violated setup time. If $t_m > T_{CLK} - t_{su}$, the D flip-flop will probably not have enough time to resolve itself and may miss one or more of its counting states, causing the circuit to fail.

Mean time between failures (MTBF) indicates the time between metastable state failures. MTBF has been shown to be inversely proportional to both the flip-flop clock frequency and the data frequency—that is, $1/(f_{CLK} \cdot f_d)$. At low clock frequencies, which we will be working with in this book, the MTBF can be measured in years, but you never know when metastability will happen. At very high clock frequencies, the MTBF can be measured in hours or even a few seconds. Designers of high-frequency digital circuits need to seriously investigate and study the problem of metastablilty in much more depth than what we briefly cover in this section.

To help prevent the internal flip-flops in a complex state machine from going into a metastable state, it is common practice to use the synchronizer circuit shown in Figure 9.15. This circuit provides *partial isolation* for a single asynchronous external input from the combinational logic cloud that provides the decoded next-state functions (NS) for the flip-flops. If a complex state machine design has n asynchronous external inputs, then n synchronizer circuits are required. The logic symbol for the synchronizer circuit shown in Figure 9.15 can be used to simplify drawings.

If the first basic synchronizer's output in Figure 9.15 is forced into a metastable state, the second basic synchronizer's output will most likely not become metastable itself. Cascading two basic synchronizers—that is, two D flip-flops as shown in Figure 9.15—to synchronize *each* asynchronous external input is considered a good way to reduce metastability of a complex state machine's internal flip-flops and thus improve reliability. Synchronous external inputs—that is, inputs that have been synchronized with the clock—do not cause metastability, but asynchronous external inputs can cause metastability. State machines should be driven by synchronous signals, *not* asynchronous signals. Synchronizing each asynchronous external input supplied to a complex state machine via the synchronizer circuit in Figure 9.15 greatly improves the state machine's reliability by helping to keep its internal flip-flops from going into a metastable state. Do not use a synchronizer circuit to synchronize clock independent inputs such as *RST*, *SET*, or *INIT*.

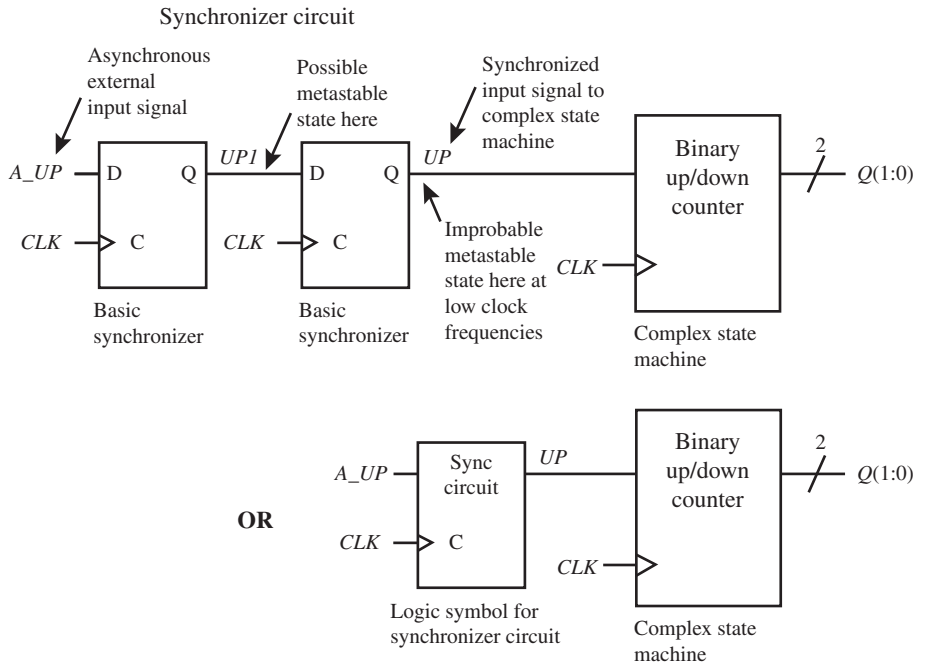


FIGURE 9.15
Synchronizer circuit, two basic synchronizers cascaded

For the synchronizer circuit in Figure 9.15, it is important to remember that an asynchronous input that is supplied to a state machine is delayed by three clock ticks. Because the synchronized input signal occurs *at the second clock tick* after an asynchronous input event, the synchronized input—that is, the output of the synchronizer circuit—will be applied to the state machine that it is driving *at the third clock tick*.

After the asynchronous input signal is synchronized, it will most likely not cause metastability of the state machine's internal flip-flops. If the second flip-flop in the synchronizer circuit goes into a metastable state, which is highly unlikely, this may force an internal flip-flop in the state machine to go into a metastable state. If this should happen, it may be necessary to reset the state machine.

Up to this point, we have assumed that all external input signals that are used to change the state sequence of a state machine are synchronous signals. If these input signals are asynchronous, we use the notation "A_" (or "a_") in front of the signal names. To synchronize an asynchronous input, use the synchronizer circuit in Figure 9.15. For consistency, we use the following signal labeling scheme: A_<signal name> is the name used for the asynchronous input signal, <signal name>1 is the name used for the output signal of the first basic synchronizer, and <signal name> is the name used for the output signal of the second basic synchronizer as shown in Figure 9.15. Example: If *UP* is an asynchronous signal, then *A_UP* is the input signal to the first basic synchronizer, *UP*1 is the output signal of the first basic synchronizer, and *UP* is the output signal of the second basic synchronizer (the synchronous input signal or synchronized input signal). If we wanted to synchronize an output signal with a synchronizer circuit, we would use the notation <signal name>_*S* as the synchronous output signal from the second basic synchronizer. This notation is used later in the book in two of the experiments (Experiments 17L and 25L).

Listing 9.9 shows how to add the synchronizer circuit in Figure 9.15 to synchronize an asynchronous input signal *UP* (*A_UP*) for the binary up/down counter (2 bits) in Figure 9.12.

LISTING 9.9
Complete VHDL design entity for BUD_counter_A_UP (binary up/down counter, 2 bits) with a synchronizer circuit to synchronize signal *A_UP* (project: BUD_counter_A_UP)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity BUD_counter_A_UP is port (
   rst, clk, a_up : in std_logic;
   q : out std_logic_vector (1 downto 0)
   );
end BUD_counter_A_UP;

architecture behavioral of BUD_counter_A_UP is
   signal up,up1: std_logic;
   type state_type is (a,b,c,d);
   signal ps, ns: state_type;
begin

Synchronize_proc: process (clk)
begin
   if rising_edge (clk) then up <= up1; up1 <= a_up;
   end if;
end process;

sync_proc: process (rst, clk)
begin
   if rst = '1' then ps <= a;
   elsif rising_edge (clk) then ps <= ns;
   end if;
end process;

comb_proc: process (ps, up)
begin
   case ps is
      when a => q <= "00"; if up = '1' then ns <= b;
                           else ns <= d;
                           end if;
      when b => q <= "01"; if up = '1' then ns <= c;
                           else ns <= a;
                           end if;
      when c => q <= "10"; if up = '1' then ns <= d;
                           else ns <= b;
                           end if;
      when d => q <= "11"; if up = '1' then ns <= a;
                           else ns <= c;
                           end if;
   end case;
end process;
end behavioral;
```

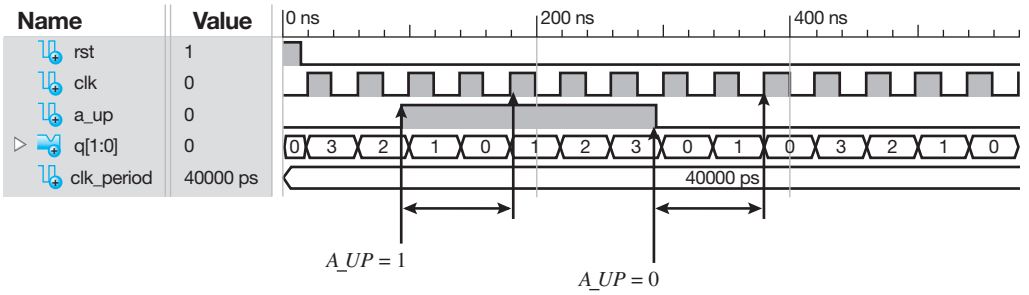
Things you should notice about the VHDL design in Listing 9.9:

- In the entity declaration in Listing 9.1, the signal *UP* was changed to *A_UP* (asynchronous *UP*).
- Two internal signals, *UP* and *UP*1, of data type std_logic were added to the design between **architecture** and the first **begin**. *UP*1 represents the output signal of the first basic syn-

chronizer, and *UP* represents the output signal of the second basic synchronizer, which is the synchronized version of *A_UP* delayed by two clock cycles. The synchronized input *UP* will be applied to the state machine it is driving *at the third clock tick*.

- A new process called **synchronize process** (synchronize_proc) was added to generate the synchronizer circuit shown in Figure 9.15, which is two basic synchronizers connected in cascade.
- Only the input *CLK* is required in the sensitivity list of the synchronize process (synchronize_proc), because the process only generates D flip-flops, which are completely controlled by just the clock input.
- No other changes were required—that is, the rest of the code is identical to the VHDL code in Listing 9.1.

Waveform 9.6 shows the correct functionality of design entity BUD_counter_A_UP (binary up/down counter) for Listing 9.9.



For *A_UP* = 1 and *A_UP* = 0, the asynchronous input *A_UP* is delayed be three clock ticks (rising edge of clock) before being applied to the state machine.

WAVEFORM 9.6
Simulation for the correct functionality of design entity BUD_counter_A_UP (binary up/down counter, 2 bits) with a synchronizer circuit to synchronize signal *A_UP* for Listing 9.9

The signal *UP* is derived from the synchronizer circuit shown in Figure 9.15 that is used to synchronize the asynchronous signal *A_UP*. The signal *UP* is not shown in Waveform 9.6. The signal *UP* follows the signal *A_UP* after the third clock tick.

Observe that the counter begins counting up when *A_UP* changes to 1 *after the third clock tick*, as shown by the arrows; when *A_UP* changes back to 0, the counter begins to count down *after the third clock tick*, as shown by the arrows.

Pay attention to the direction of the counting sequence provided by output q in Waveform 9.6. Initially, the counter begins counting down until *A_UP* is asserted, and then the counter begins counting up. It continues to count up until *A_UP* is not asserted (*A_UP* = 0), and then the counter begins to count down. Observe that the change doesn't take place until after the third clock tick when *A_UP* changes to 1 and also after the third clock tick when *A_UP* changes to 0.

The signals *UP*1 and *UP* are not shown in Waveform 9.6 because they are internal signals in Listing 9.9. If you would like to display signals *UP*1 and *UP* in a simulation waveform, simply include the signals *UP*1 and *UP* in the entity in Listing 9.9 with a mode of **inout** and remove them as internal signals; then run a new simulation.

9.11 ADDITIONAL STATE MACHINE DESIGN METHODS

This section provided two additional coding methods that may be used to design state machines. We refer to the methods as the **two-assignment PS/NS method** and the **hybrid PS/NS method**. Many other coding methods are possible for state machine design with VHDL. Just remember to verify that the coding style that you elect to use provides correct functionality by simulating your design and checking for correct functionality.

9.11.1 Two-Assignment PS/NS Method

Figure 9.16 shows a circuit model for a complex state machine with just Moore outputs. No flip-flop outputs or Mealy outputs are provided in this circuit model.

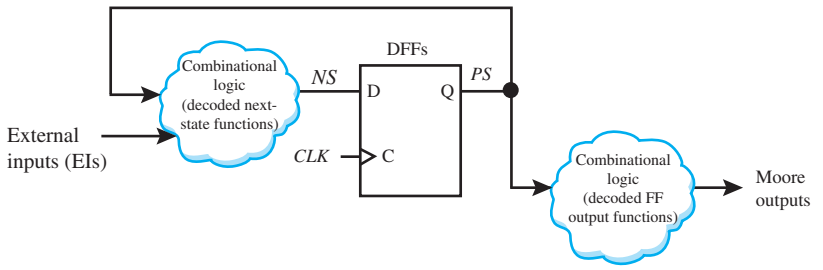


FIGURE 9.16 Circuit model for a complex state machine with just Moore outputs

If a single pulse called *ONE_PULSE* is supplied as the external input to complex state machine 1 shown in Figure 9.17a, the function of the state machine is to generate a signal called *TWO_PULSES* on its output, as shown in the counting sequence diagram in Figure 9.17b. The circuit model in Figure 9.16 represents this type of complex state machine.

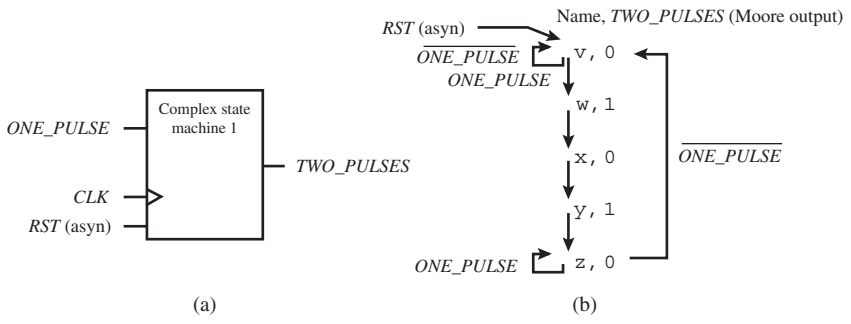


FIGURE 9.17 Complex state machine 1: (a) logic symbol; (b) state sequence diagram

Things you should notice about complex state machine 1 in Figure 9.17:

- The logic symbol for complex state machine 1 has an asynchronous reset input signal *RST*, a clock input signal *CLK*, a synchronous input signal *ONE_PULSE*, and a Moore output signal *TWO_PULSES*.
- The state sequence diagram shows that complex state machine 1 is reset to state v when *RST* is 1. When *RST* is 0, complex state machine 1 follows the counting sequence at the frequency of the signal *CLK*. When *ONE_PULSE* is 0 the state machine stays in state v. When *ONE_PULSE* is 1, the state machine transitions to state w to state x to state y and then to state z, where it remains until *ONE_PULSE* returns to 0; and then it transitions back to state v. For the state sequence to occur, it is assumed that the duration of the signal *ONE_PULSE* >> the period of *CLK*.
- While the state machine transitions from state v through state z, the output signal *TWO_PULSES* goes from 0 to 1 to 0 to 1 to 0—that is, the output signal *TWO_PULSES* generates two positive pulses. The signal *TWO_PULSES* is a Moore output because it is state dependent and not transition dependent.
- In effect, the state machine produces an output with two pulses via a signal called *TWO_PULSES* each time the signal *ONE_PULSE* occurs at its input.

Listing 9.10 shows a complete VHDL design for complex state machine 1 in Figure 9.17 using a procedure similar to the two-process PS/NS method which we call the **two-assignment**

PS/NS method. The two-assignment PS/NS method shown in Listing 9.10 uses a dataflow design style rather than a behavioral design style.

LISTING 9.10
Complete VHDL design for complex state machine 1 (CSM1) using the two-assignment PS/NS method (project: CSM1)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CSM1 is port (
   rst,clk, one_pulse : in std_logic;
   two_pulses : out std_logic
   );
end CSM1;

architecture dataflow of CSM1 is
   type state_type is (v,w,x,y,z);
   signal ps, ns: state_type;
begin

Sync_assign:
   ps <= v when rst = '1' else
         ns when rising_edge (clk);

Comb_assign:
   ns <= v when ps = v and one_pulse = '0' else
         w when ps = v and one_pulse = '1' else
         x when ps = w else
         y when ps = x else
         z when ps = y or (ps = z and one_pulse = '1') else
         v;

   two_pulses <= '0' when ps = v else
                 '1' when ps = w else
                 '0' when ps = x else
                 '1' when ps = y else
                 '0';
end dataflow;
```

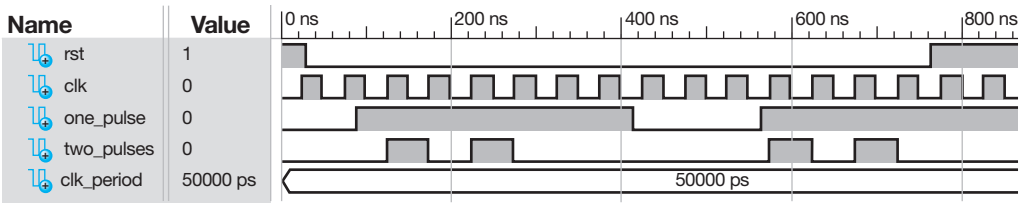
Things you should notice about the VHDL design in Listing 9.10:

- The design for the two-assignment PS/NS method is analogous to the two-process PS/NS method covered earlier in the book.
- The synchronous process is replaced by the synchronous assignment, and the combinational process is replaced by the combinational assignment—which are both conditional signal assignments (CSAs).
- We used one conditional signal assignment to write the Sync_assign (synchronous assignment) for the signal *PS* and a different conditional signal assignment to write the Comb_assign (combinational assignment) for the signal *NS* signal. Observe that the conditional signal assignment for the signal *NS* requires a final **else** or terminating **else** (**else** v;), so a latch circuit will not be generated for the signal *NS*. Without the final **else**, a latch circuit with 5 bits will be generated for the signal *NS*, which is not desirable. Remember: A latch circuit will be generated for a combinational logic circuit that uses a conditional signal assignment when the final **else** in the assignment is left out.
- The output signal *TWO_PULSES* was written using a conditional signal assignment rather than a selected signal assignment that we normally use with the two-process PS/NS

method. Observe that the conditional signal assignment for *TWO_PULSES* also requires the final **else** (**else** '0';), so a latch circuit will not be generated for the signal *TWO_PULSES*.

Waveform 9.7 shows the correct functionality of design entity CSM1 for Listing 9.10.

WAVEFORM 9.7 Simulation for the correct functionality of design entity CSM1 for Listing 9.10

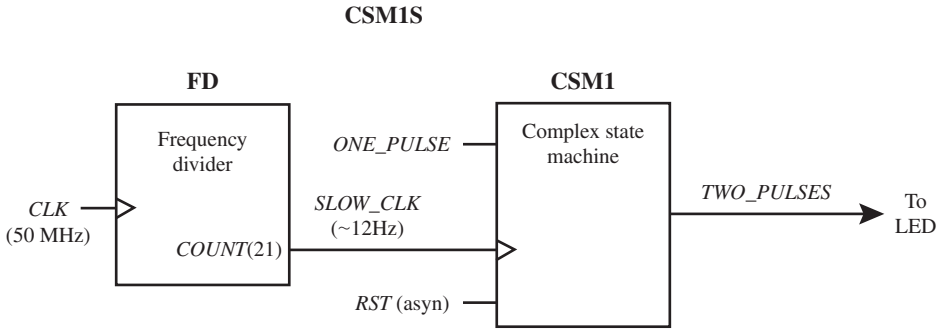


Things you should notice about the waveforms in Waveform 9.7:

- *RST* is first set to 1 and then set to 0 to reset complex state machine 1.
- *ONE_PULSE* is set to 1 which may represent a push button that has been pressed to generate an active high signal. Observe that two positive pulses are generated at the output *TWO_PULSES* during the time that *ONE_PULSE* is 1.
- *ONE_PULSE* is set to 0, which may represent a push button that has been released. During the time that *ONE_PULSE* is 0 the output *TWO_PULSES* stays at 0.
- Each time *ONE_PULSE* is set to 1, two positive pulses are generated at the output *TWO_PULSES*.

A complex state machine 1 system (CSM1S) diagram is shown in Figure 9.18. This system diagram allows you to test a hardware implementation of the CSM1 design. A frequency divider is added to the design of CSM1, so that its operation can be verified via push button switches (one push-button switch for the input *RST* and another for input *ONE_PULSE*) and a single LED (to display the output signal *TWO_PULSES*).

FIGURE 9.18 Complex state machine 1 system (CSM1S) diagram (project: CSM1S)



If you elect to use a hardware board, you will observe that after reset is asserted via its push-button switch, each time the push-button switch for *ONE_PULSE* is pressed, the LED for *TWO_PULSES* will blink twice, verifying that CSM1 works as indicated in Waveform 9.7.

The 50 MHz shown in Figure 9.18 for the signal *CLK* is the frequency provided on an FPGA board such as a BASYS2 or NEXYS2 board. The frequency for *SLOW_CLK* is $(50 \times 10^6)/2^{22} = 11.9209$ Hz, or about 12 cycles per second. If this circuit is operated at a slower frequency it may not function properly because the duration of the signal *ONE_PULSE* may not be >> the period of *SLOW_CLK*. If the circuit is operated too fast, you will not be able to observe blinking of the LED, which is provided by the signal *TWO_PULSES*.

9.11.2 Hybrid PS/NS Method

Figure 9.19 shows a circuit model for a complex state machine with flip-flop outputs and Moore outputs. No Mealy outputs are provided in this circuit model.

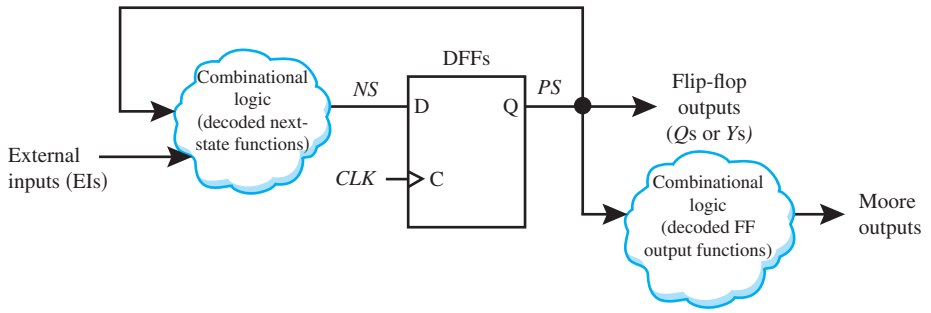


FIGURE 9.19 Circuit model for a complex state machine with flip-flop outputs and Moore outputs

Figure 9.20 shows a slightly modified version of complex state machine 1 in Figure 9.17 that includes flip-flop outputs. This state machine is called complex state machine 2. The circuit model in Figure 9.19 represents this type of complex state machine.

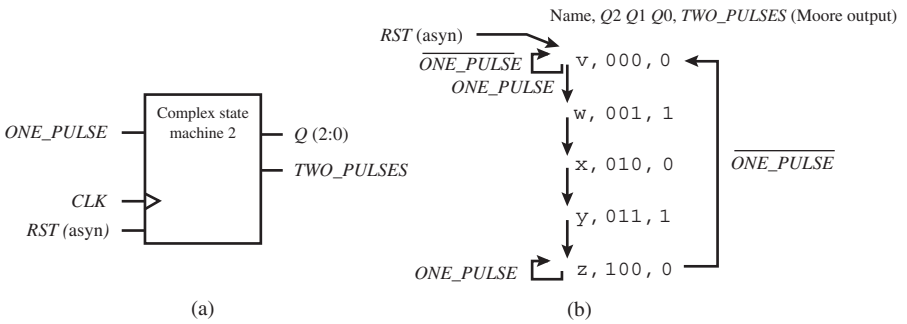


FIGURE 9.20 Complex state machine 2: (a) logic symbol; (b) state sequence diagram

Things you should notice about complex state machine 2 in Figure 9.20:

- The logic symbol for complex state machine 2 has an asynchronous reset input signal *RST*, a clock input signal *CLK*, a synchronous input signal *ONE_PULSE*, flip-flop outputs *Q*(2:0), and a Moore output signal *TWO_PULSES*.
- The state sequence diagram shows that complex state machine 2 is reset to state v (with flip-flop outputs *Q*2 *Q*1 *Q*0 = 000) when *RST* is 1. When *RST* is 0, complex state machine 2 follows the counting sequence at the frequency of the signal *CLK*. When *ONE_PULSE* is 0, the state machine stays in state v. When *ONE_PULSE* is 1, the state machine transitions to state w (*Q*2 *Q*1 *Q*0 = 001), to state x (*Q*2 *Q*1 *Q*0 = 010), to state y (*Q*2 *Q*1 *Q*0 = 011), and then to state z (*Q*2 *Q*1 *Q*0 = 100), where it remains until *ONE_PULSE* returns to 0; then it transitions back to state v. For the state sequence to occur, it is assumed that the duration of the signal *ONE_PULSE* >> the period of *CLK*.

Listing 9.11 shows a complete VHDL design for complex state machine 2 (CSM2) in Figure 9.20 using a hybrid PS/NS method. In this method, a CSA (conditional signal assignment) is used for the synchronous assignment (Sync_assign), and a process with a case statement is used for the combinational process (Comb_proc). The order could be reversed—that is, use a process with an if statement for the synchronous process (Sync_proc), as shown commented out, and a CSA for the combinational assignment (Comb_assign), as shown commented out.

LISTING 9.11
Complete VHDL design for complex state machine 2 (CSM2) using a hybrid PS/NS method (project: CSM2)

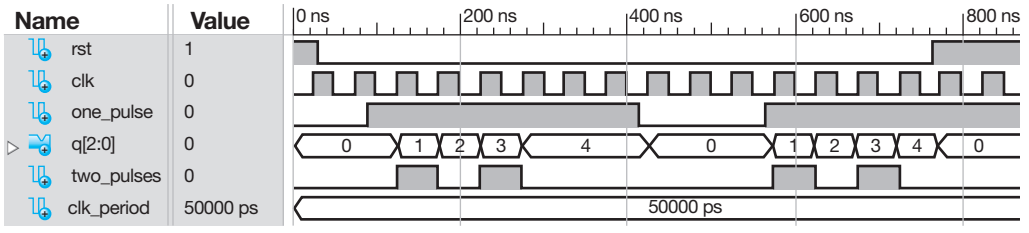
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CSM2 is port (
   rst, clk, one_pulse : in std_logic;
   q : out std_logic_vector (2 downto 0);
   two_pulses : out std_logic
   );
end CSM2;

architecture mixed of CSM2 is
   type state_type is (v,w,x,y,z);
   signal ps, ns: state_type;
begin

Sync_assign:
   ps <= v when rst = '1' else
         ns when rising_edge (clk);
--Sync_proc:
--process (rst, clk)
--begin
--    if rst = '1' then ps <= v;
--    elsif rising_edge (clk) then ps <= ns;
--    end if;
--end process;

Comb_proc:
process (ps, one_pulse)
begin
   case ps is
      when v => if one_pulse = '0' then ns <= v;
                else ns <= w;
                end if;
      when w => ns <= x;
      when x => ns <= y;
      when y => ns <= z;
      when z => if one_pulse = '1' then ns <= z;
                else ns <= v;
                end if;
   end case;
end process;
--Comb_assign:
--    ns <= v when ps = v and one_pulse = '0' else
--          w when ps = v and one_pulse = '1' else
--          x when ps = w else
--          y when ps = x else
--          z when ps = y or (ps = z and one_pulse = '1') else
--          v;
Outputs: --these could be included within the Comb_proc, but not
          the Comb_assign
```

```
    q <= "000" when ps = v else
         "001" when ps = w else
         "010" when ps = x else
         "011" when ps = y else
         "100";

    two_pulses <= '0' when ps = v else
                  '1' when ps = w else
                  '0' when ps = x else
                  '1' when ps = y else
                  '0';
end mixed;
```

Things you should notice about the VHDL design in Listing 9.11:

- The design for the hybrid PS/NS method uses part of the two-process PS/NS method and part of the two-assignment PS/NS method.
- We used one conditional signal assignment to write the Sync_assign (synchronous assignment) for the signal *PS*, and a process with a case statement to write the Comb_process (combinational process).
- The output signal *Q* was written using a conditional signal assignment rather than a selected signal assignment that we normally use with the two-process PS/NS method. Observe that the conditional signal assignment for *Q* requires the final **else** (i.e., **else** "100";), so a latch circuit will not be generated for the signal *Q*.
- The output signal *TWO_PULSES* was written using a conditional signal assignment rather than a selected signal assignment that we normally use with the two-process PS/NS method. Observe that the conditional signal assignment for *TWO_PULSES* requires the final **else** (i.e., **else** '0';), so a latch circuit will not be generated for the signal *TWO_PULSES*.

Waveform 9.8 shows the correct functionality of design entity CSM2 for Listing 9.11.



WAVEFORM 9.8
Simulation for the correct functionality of design entity CSM2 for Listing 9.11

Things you should notice about the waveforms in Waveform 9.8:

- *RST* is first set to 1 and then set to 0 to reset complex state machine 1.
- *ONE_PULSE* is set to 1, which may represent a push button that has been pressed to generate an active high signal. Observe that two positive pulses are generated at the output *TWO_PULSES* during the time that *ONE_PULSE* is 1.
- *ONE_PULSE* is set to 0, which may represent a push button that has been released. During the time that *ONE_PULSE* is 0, the output *TWO_PULSES* is at 0.
- Each time *ONE_PULSE* is set to 1, two positive pulses are generated at the output *TWO_PULSES*.
- Observe that the flip-flop outputs *Q*(2:0)are included in Waveform 9.8, which allows one to easily confirm that the waveforms represent the correct functionality for design entity CSM2.

To allow you to observe the operation of the CSM2 design working on a hardware board, replace CSM1 in Figure 9.18 with CSM2 and remove the signal *Q*(2:0), or comment it out, because it is not required to observe the output *TWO_PULSES* (project: CSM2S). If you keep *Q*(2:0) in the design of CSM2S, you will also see *Q*(2:0) swiftly move through the state sequence each time you press the push button to generate the signal *TWO_PULSES*.

PROBLEMS

Section 9.1 Introduction

9.1 How do we define a complex state machine?

9.2 When are complex state machines and simple state machines called controllers?

Section 9.2 Designing with the Two-Process PS/NS Method

9.3 What type of flip-flop is generated by the synchronous process of the two-process PS/NS method?

9.4 What signals are generated by the synchronous process of the two-process PS/NS method?

9.5 What signals are generated by the combinational process of the two-process PS/NS method?

9.6 Where does the declaration for the type state_type and the internal signals *PS* (present state) and *NS* (next state) have to be placed in the architecture declaration?

9.7 Is a when others clause required as the last choice value in the case statement for the select signal *PS*, which is an enumerated data type? Discuss your answer.

9.8 Write a behavioral architecture declaration for the partially stoppable binary up counter circuit in Figure P9.8 using the two-process PS/NS method. Assign values to the flip-flop outputs in the combinational process. Is this state machine simple or complex? Provide an explanation for your answer.

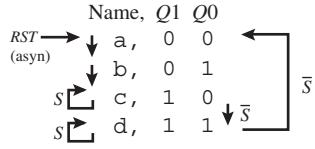


FIGURE P9.8

9.9 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter circuit in Figure P9.8.

9.10 Combine your code for problems 9.8 and 9.9 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

Section 9.3 Explanation of CPLDs and FPGAs and State Machine Encoding Styles

9.11 Why are CPLDs classified as nonvolatile?

9.12 Which state machine encoding style requires a minimum number of flip-flops?

9.13 Which state machine encoding style requires a maximum number for flip-flops? Explain your answer.

9.14 How many states can a compact encoded state machine have if there are five state bits?

9.15 What is the minimum number of flip-flops required for a compact encoded state machine with seven states?

9.16 How many states can a one-hot encoded state machine have if there are nine state bits?

9.17 What is the number of flip-flops required for a one-hot encoded state machine with 14 states?

9.18 Is the state machine in Figure P9.18 compact encoded or one-hot encoded? Provide an explanation for your answer.

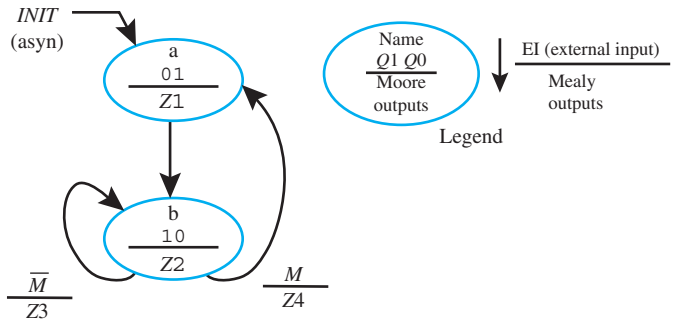


FIGURE P9.18

9.19 Is the state machine in Figure P9.19 compact encoded or one-hot encoded? Provide an explanation for your answer.

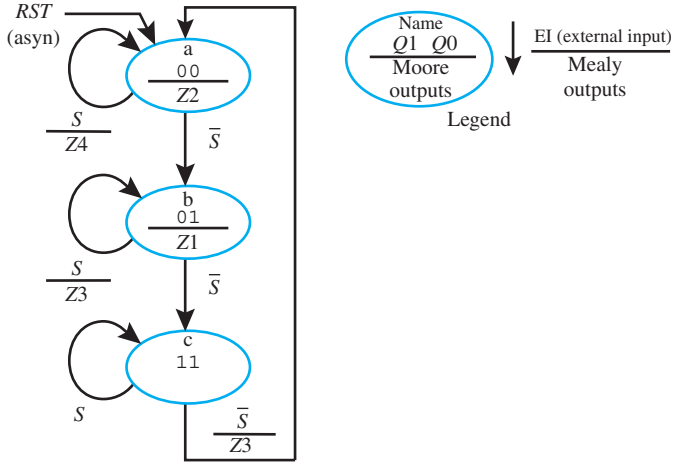


FIGURE P9.19

9.20 Is the state machine in Figure P9.20 compact encoded or one-hot encoded? Provide an explanation for your answer.

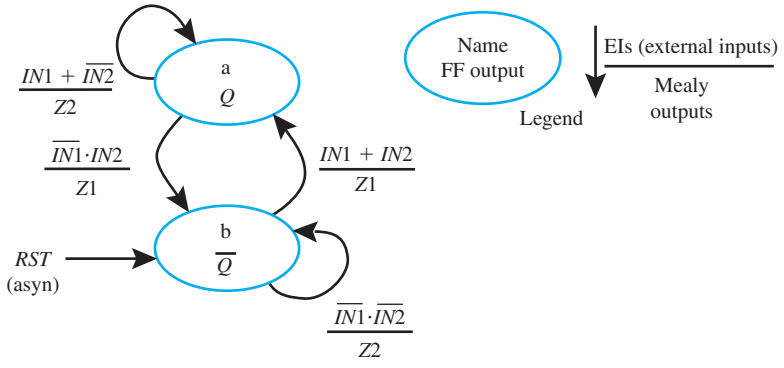


FIGURE P9.20

9.21 Is the state machine in Figure P9.21 compact encoded or one-hot encoded? Provide an explanation for your answer.

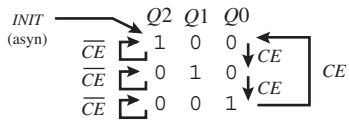


FIGURE P9.21

9.22 Is the state machine in Figure P9.22 compact encoded or one-hot encoded? Provide an explanation for your answer.

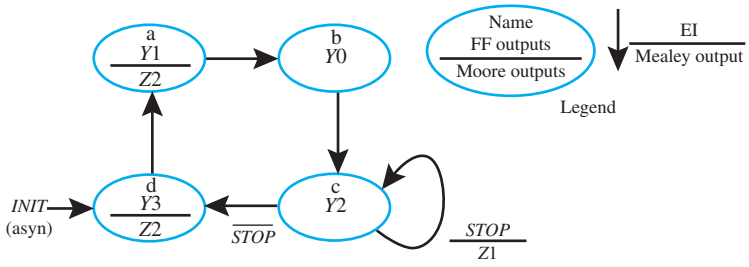


FIGURE P9.22

Section 9.4 Summary of Finite State Machine Models

9.23 Is the state machine model in Figure P9.23 simple or complex? What type of outputs does this state machine model have?

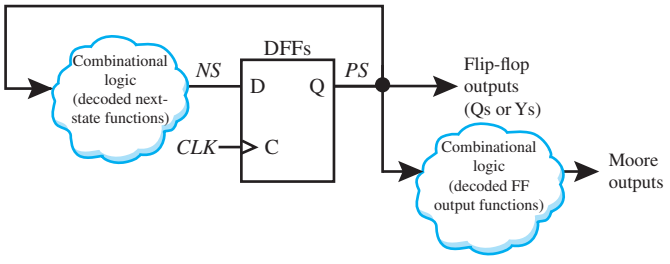


FIGURE P9.23

9.24 What is the difference between flip-flop outputs and Moore outputs in Figure P9.23?

9.25 Is the state machine model in Figure P9.25 simple or complex? What type of outputs does this state machine model have?

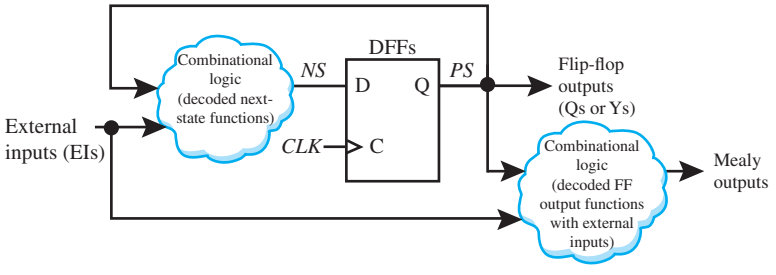


FIGURE P9.25

9.26 What is the difference between Moore outputs and Mealy outputs? Provide an example of each output type.

Section 9.5 Designing Compact Encoded State Machines with Moore Outputs

9.27 Where are the Moore outputs always shown in a state diagram?

9.28 Write the state sequence diagram for a stoppable Johnson counter (3 bits) with marching 1s from the right with an asynchronous reset input and also for a Gray code up counter (3 bits) with an asynchronous reset input.

9.29 Figure P9.29 shows the circuit diagram for a 3-bit counter that has been referred to in the literature by the following names: twisted ring counter, switch tail ring counter, Johnson counter, or mobius counter. Write the state sequence diagram for the counter. (Hint: Obtain the next-state equations for the circuit by obtaining the excitation equations.) Remember that the next-state value Y^+ for each D flip-flop is the D excitation input—that is, $Yi^+ = Di$ for $i = 0, 1$, etc. After you obtain each next-state value Y^+ for each D flip-flop, you can determine the next-state values of the counter. Use the format S<decimal value for the state> for each state name in the state sequence diagram. Is the counter a simple counter or a complex counter? Provide an explanation for your answer.

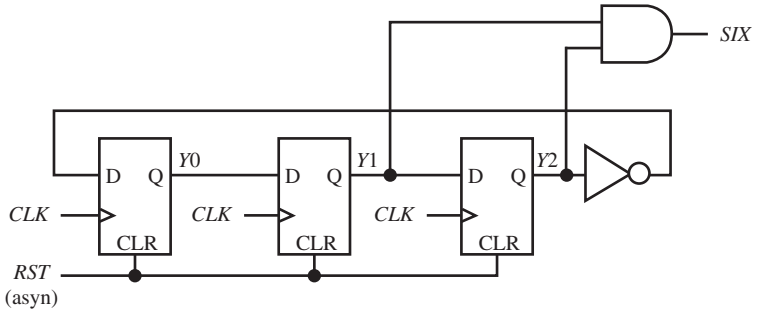


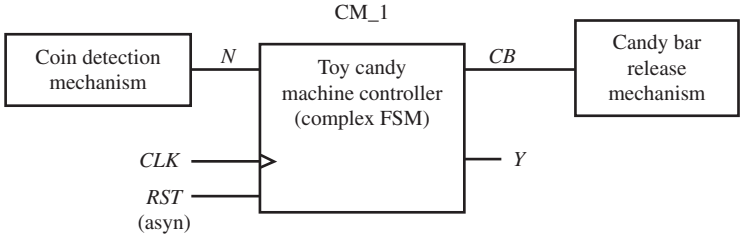
FIGURE P9.29

9.30 Draw a state diagram for the Johnson Counter shown in Figure P9.29. (Hint: Obtain the next-state equations for the circuit by obtaining the excitation equations.) Remember that the next-state value Y^+ for each D flip-flop is the D excitation input—that is, $Yi^+ = Di$ for $i = 0, 1$, etc. After you obtain each next-state value Y^+ for each D flip-flop, you can determine the next-state values of the counter. Use the format S<decimal value for the state> for each state name in the state diagram.

9.31 Figure P9.31A shows a block diagram for a toy candy machine that requires that a nickel be dropped into the coin detection mechanism to cause the release of a candy bar by the candy bar release mechanism—that is, a candy bar costs 5 cents. A state diagram for the toy candy machine controller, called candy machine 1 (CM_1), with compact encoding is shown in Figure P9.31B (top of next page). The time it takes for the nickel to drop through the coin detection mechanism is actually much greater than the period of the signal *CLK*. Write a behavioral architecture declaration in VHDL for CM_1 using the **two-process PS/NS method**.

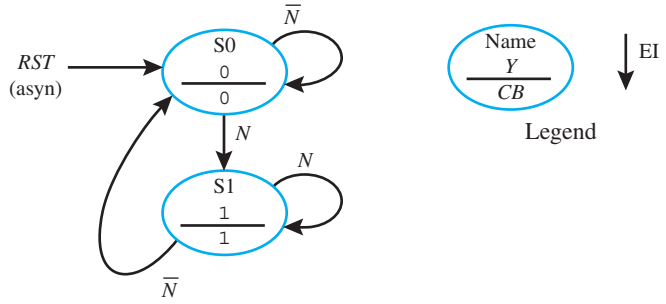
9.32 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for toy candy machine controller CM_1 in Figure P9.31b.

9.33 Combine your code for problems 9.31 and 9.32 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.



(a)

FIGURE P9.31A



(b)

FIGURE P9.31B

Section 9.6 Designing One-Hot Encoded State Machines with Moore Outputs

9.34 Write a mixed architecture declaration for the partially stoppable counter circuit in Figure P9.34 using the two-process PS/NS method. Assign values to the present state outputs in a selected signal assignment (SSA). Is this state machine compact encoded or one-hot encoded? Provide an explanation for your answer.

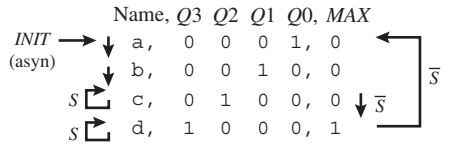


FIGURE P9.34

9.35 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the counter circuit in Figure P9.34.

9.36 Combine your code for problems 9.34 and 9.35 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

9.37 Figure P9.37 shows the circuit diagram for a 3-bit counter called a ring counter. Write the state sequence diagram for the counter. (Hint: Obtain the next-state equations for the circuit by obtaining the excitation equations.) Remember that the next-state value Q^+ for each D flip-flop is the D excitation input—that is, $Qi^+ = Di$ for $i = 0, 1$, etc. After you obtain each next-state value Q^+ for each D flip-flop, you can determine the next-state values of the counter. Use the format S<decimal value for the state> for each state name in the state sequence diagram. Is the counter a simple counter or a complex counter? Provide an explanation for your answer. What would be a good name for this counter based on its counting sequence?

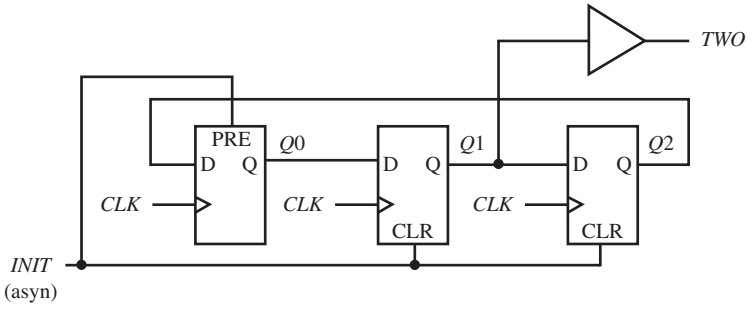


FIGURE P9.37

9.38 Draw a state diagram for the ring counter shown in Figure P9.37. (Hint: Obtain the next-state equations for the circuit by obtaining the excitation equations.) Remember that the next-state value Q^+ for each D flip-flop is the D excitation input—that is, $Q^+ = DI$ for $i = 0, 1$, etc. After you obtain each next-state value Q^+ for each D flip-flop, you can determine the next-state values of the counter. Use the format S<decimal value for the state> for each state name in the state diagram.

9.39 Do the flip-flop outputs always have to be included in the combinational process? Discuss your answer.

9.40 Figure P9.40*a* shows a block diagram for a toy candy machine that requires that a nickel be dropped into the coin detection mechanism to cause the release of a candy bar by the candy bar release mechanism—that is, a candy bar costs 5 cents. A state diagram for the toy candy machine controller, called candy machine 2 (CM_2), with one-hot encoding is shown in Figure P9.40*b*. The time it takes for the nickel to drop through the coin detection mechanism is actually much greater than the period of the signal *CLK*. Write a behavioral architecture declaration in VHDL for CM_2 using the two-process PS/NS method.

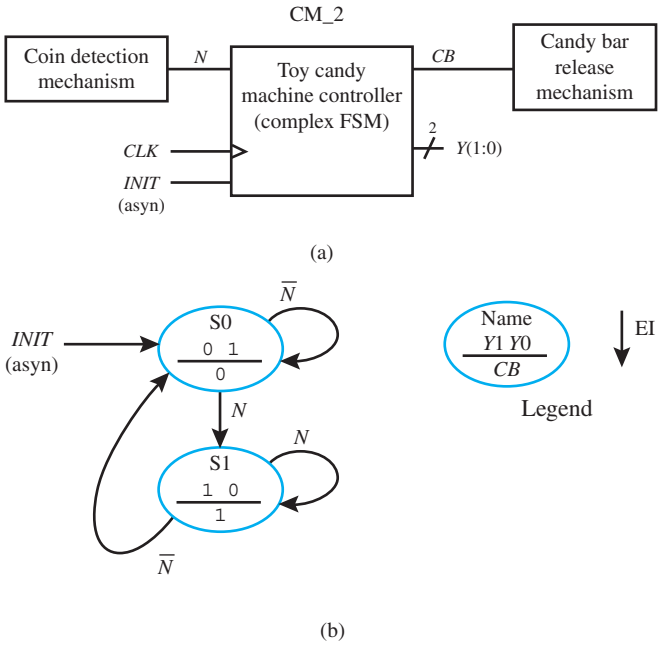


FIGURE P9.40

9.41 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for toy candy machine controller CM_2 in Figure P9.40b.

9.42 Combine your code for problems 9.40 and 9.41 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

Section 9.7 Designing Compact Encoded State Machines with Moore and Mealy Outputs

9.43 Which signal convention—the Show Where True or Show All Values—helps simplify a state diagram? Explain your answer.

9.44 Convert the Moore and Mealy outputs in the state diagram in Figure P9.44 from the Show All Values signal convention to an equivalent Show Where True signal convention. Is this state machine simple or complex? Provide an explanation for your answer.

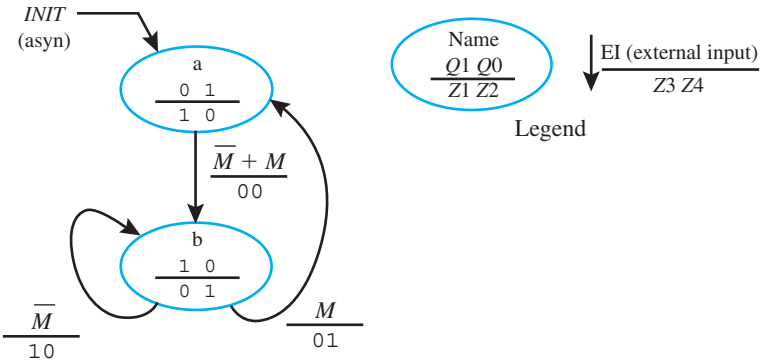


FIGURE P9.44

9.45 Convert the Moore and Mealy outputs in the state diagram in Figure P9.45 from the Show Where True signal convention to an equivalent Show All Values signal convention. Is this state machine simple or complex? Provide an explanation for your answer.

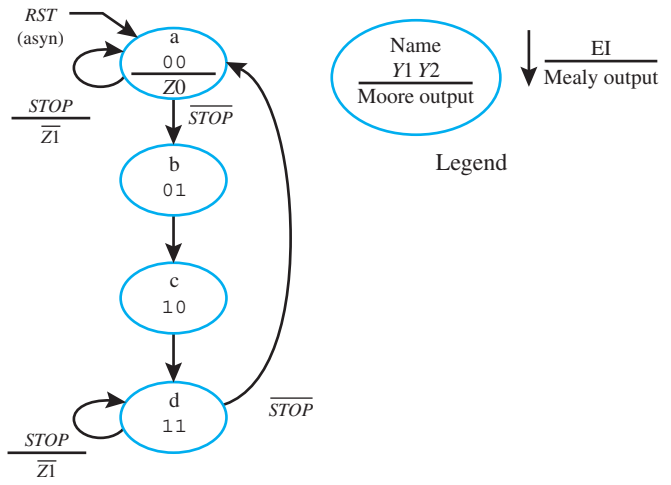


FIGURE P9.45

9.46 Write a mixed architecture declaration for the state machine in Figure P9.46 using the two-process PS/NS method. Assign values to the flip-flop outputs in a selected signal assignment (SSA).

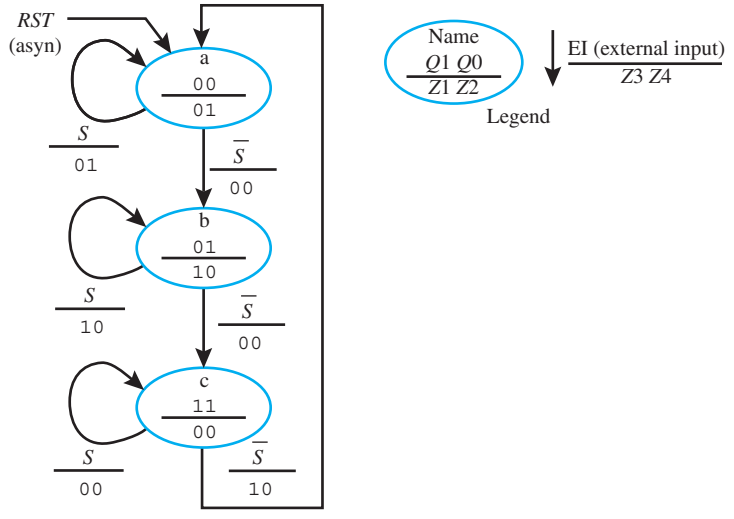


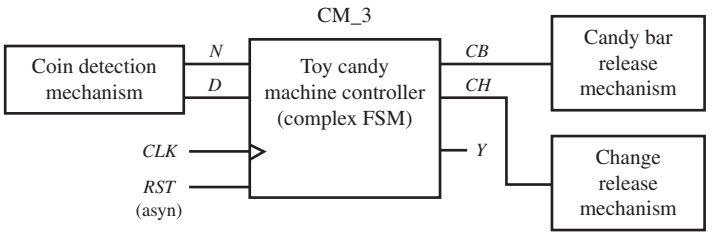
FIGURE P9.46

9.47 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the state machine in Figure P9.46.

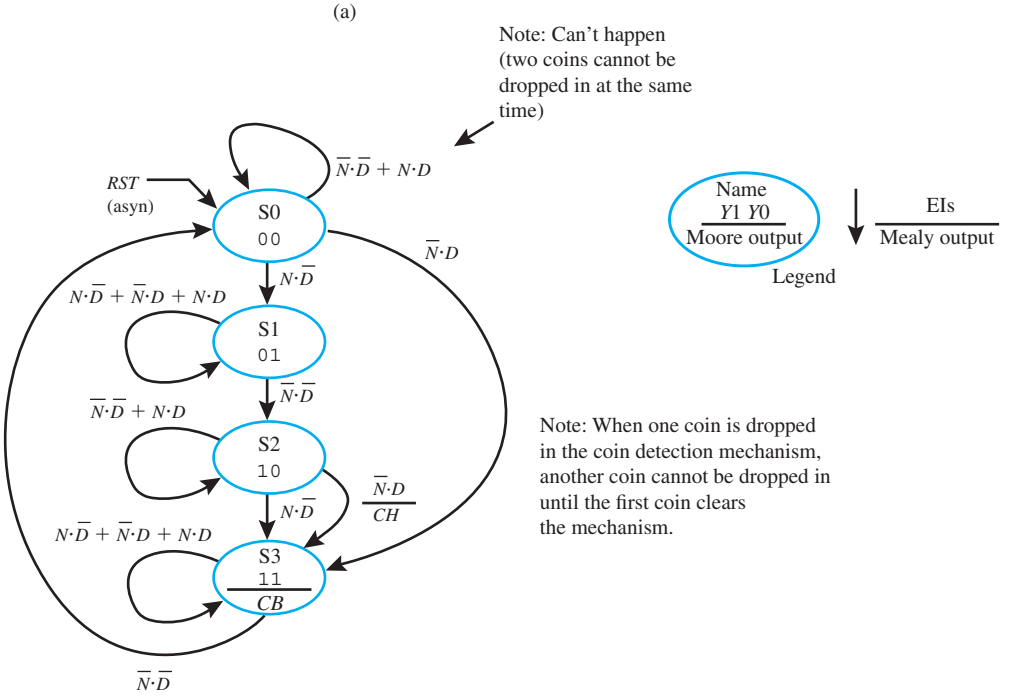
9.48 Combine your code for problems 9.46 and 9.47 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

9.49 Figure P9.49a shows a block diagram for a toy candy machine that requires that a dime, two nickels, or a nickel and a dime be dropped into the coin detection mechanism to cause the release of a candy bar by the candy bar release mechanism—that is, a candy bar costs 10 cents. If you first drop a nickel then a dime, the change release mechanism will return a nickel. A state diagram for the toy candy machine controller, called candy machine 3 (CM_3), with compact encoding is shown in Figure P9.49b. The time it takes for each coin to drop through the coin detection mechanism is

actually much greater than the period of the signal *CLK*. Write a behavioral architecture declaration in VHDL for CM_3 using the two-process PS/NS method.



(a)



(b)

FIGURE P9.49

9.50 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for toy candy machine controller CM_3 in Figure P9.49b.

9.51 Combine your code for problems 9.49 and 9.50 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

Section 9.8 Designing One-Hot Encoded State Machines with Moore and Mealy Outputs

9.52 There is one place where you can simplify a state diagram that uses a one-hot encoding scheme. By writing a single state variable in each state bubble, you can reduce the number of variables in each state. Only the active or asserted state variable needs to appear in each state bubble. The state diagram in Figure P9.52 is a simple one-hot encoded state machine. Modify the state diagram in Figure P9.52 so that a single state variable (flip-flop output $Q2$, $Q1$, or $Q0$)—that is, the one that is asserted—is used in each state bubble.

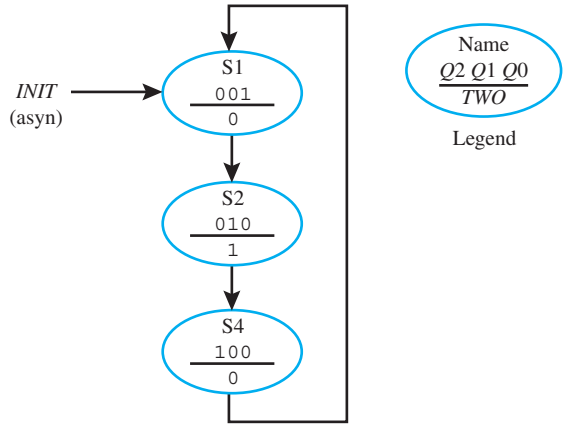


FIGURE P9.52

9.53 Write a behavioral architecture declaration for the state machine in Figure P9.53 using the two-process PS/NS method.

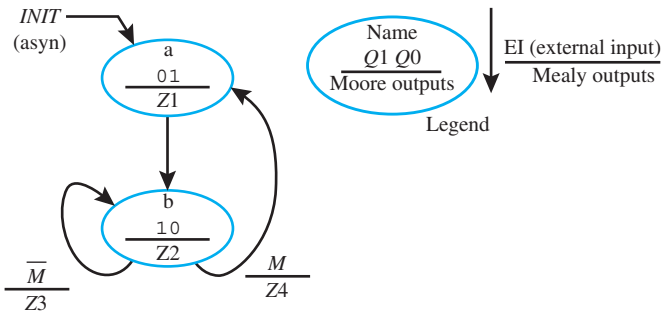


FIGURE P9.53

9.54 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the state machine in Figure P9.53.

9.55 Combine your code for problems 9.53 and 9.54 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

9.56 Figure P9.56a shows a block diagram for a toy candy machine that requires that a nickel and a dime, two nickels and a dime, or three nickels be dropped into the coin detection mechanism to cause the release of a candy bar by the candy bar release mechanism—that is, a candy bar costs 15 cents. If you first drop two nickels and a dime, the change release mechanism will return a nickel. A state diagram for the toy candy machine controller, called candy machine 4 (CM_4), with one-hot encoding is shown in Figure P9.56b. The time it takes for each coin to drop through the coin detection mechanism is actually much greater than the period of the signal *CLK*. Write a behavioral architecture declaration in VHDL for CM_3 using the two-process PS/NS method.

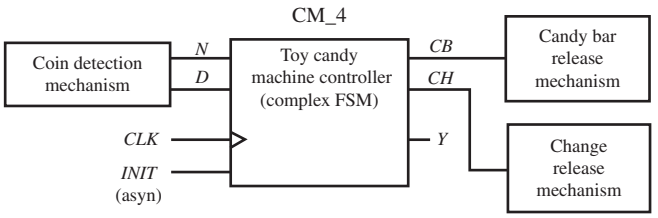
9.57 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for toy candy machine controller CM_4 in Figure P9.56b.

9.58 Combine your code for problems 9.56 and 9.57 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

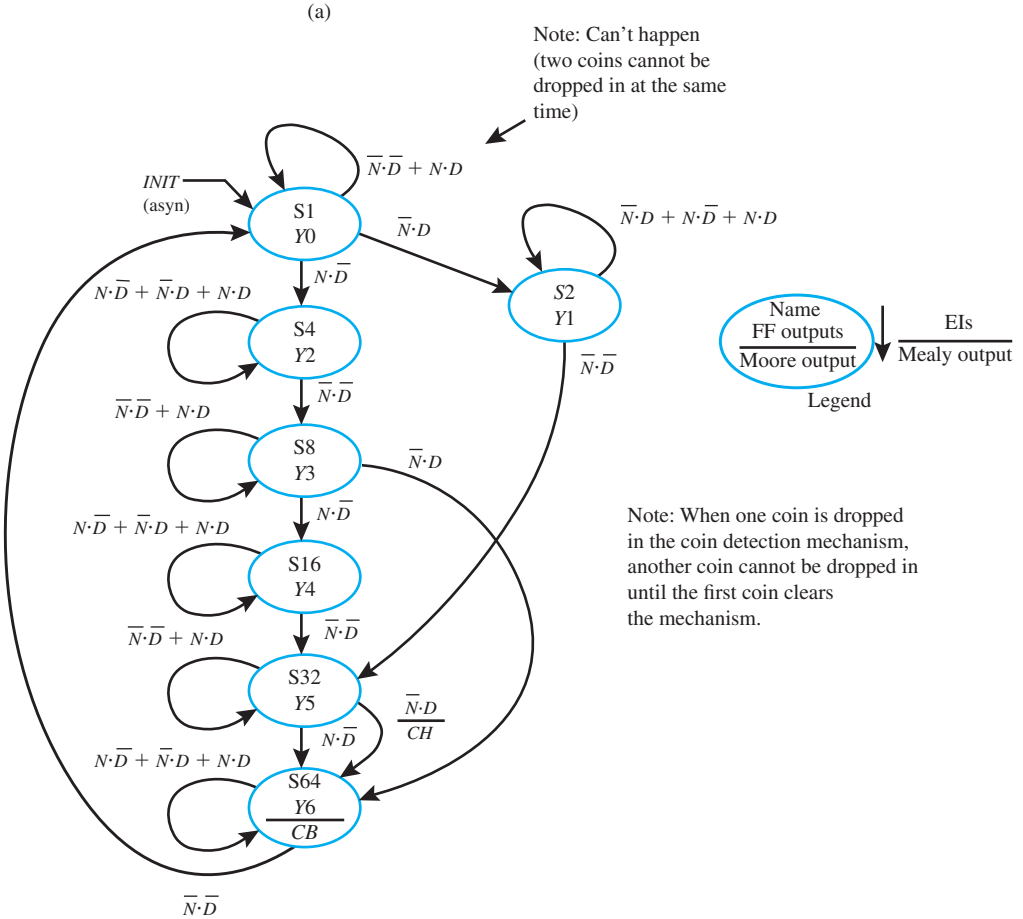
Section 9.9 Using the Algorithmic Equation Method to Design Complex State Machines

9.59 What is the disadvantage of using the algorithmic equation method compared to using the two-process PS/NS method?

9.60 Write the Set OR Hold 1 equation for a complex state machine. What is the Set OR Hold 1 equation used for? What does the *D* represent in the Set OR Hold 1 equation?



(a)



(b)

FIGURE P9.56

9.61 What is the procedure for obtaining the VHDL code for a state machine when using the algorithmic equation method?

9.62 Figure P9.62a shows a bus arbiter that is used to controls the enable inputs for a 3-state circuit. Either data on bus A or data on bus B is supplied to a common bus, depending on the outputs of the bus arbiter or controller. Figure P9.62b shows a state diagram that is one-hot encoded for the bus arbiter, which is a complex state machine. Use the Set OR Hold 1 equation to write the excitation equations for the bus arbiter. Write the Moore output equations. Draw a clouds-of-logic circuit model for the bus arbiter.

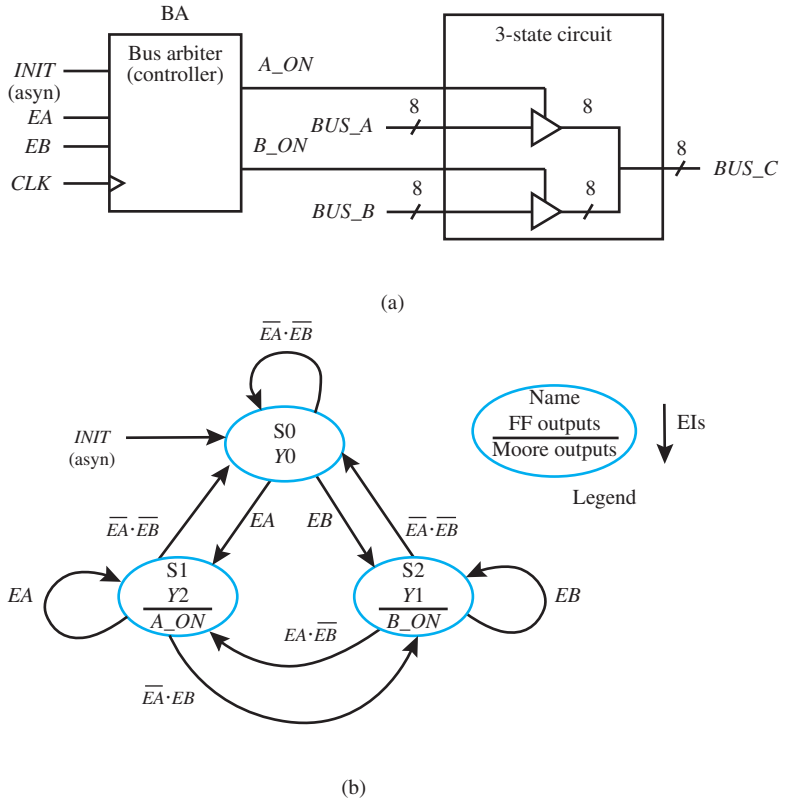


FIGURE P9.62

9.63 Write a dataflow architecture in VHDL for the bus arbiter (controller) shown in the state diagram in Figure P9.62b using the equations obtained in problem 9.62.

9.64 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the bus arbiter (controller) in Figure P9.62b.

9.65 Combine your code for problems 9.63 and 9.64 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

9.66 Figure P9.66a shows a bus arbiter that is used to controls the enable inputs for a 3-state circuit. Either data on bus A or data on bus B is supplied to a common bus, depending on the outputs of the bus arbiter or controller. Figure P9.66b shows a state diagram that is compact encoded for the bus arbiter, which is a complex state machine. Use the Set OR Hold 1 equation to write the excitation equations for the bus arbiter. Write the Moore output equations. Draw a clouds-of-logic circuit model for the bus arbiter.

9.67 Write a dataflow architecture in VHDL for the bus arbiter (controller) shown in the state diagram in Figure P9.66b using the equations obtained in problem 9.66.

9.68 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the bus arbiter (controller) in Figure P9.66b.

9.69 Combine your code for problems 9.67 and 9.68 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

Section 9.10 Improving the Reliability of Complex State Machine Designs

9.70 What causes a circuit to fail due to metastability?

9.71 What is the relationship of the flip-flop's clock frequency and the flip-flop's data frequency in relationship to MTBF?

9.72 What can be added to a complex state machine design with an external asynchronous input to improve its reliability? Briefly discuss your answer.

9.73 Can a synchronizer circuit be used to improve the reliability of a simple state machine? Briefly discuss your answer.

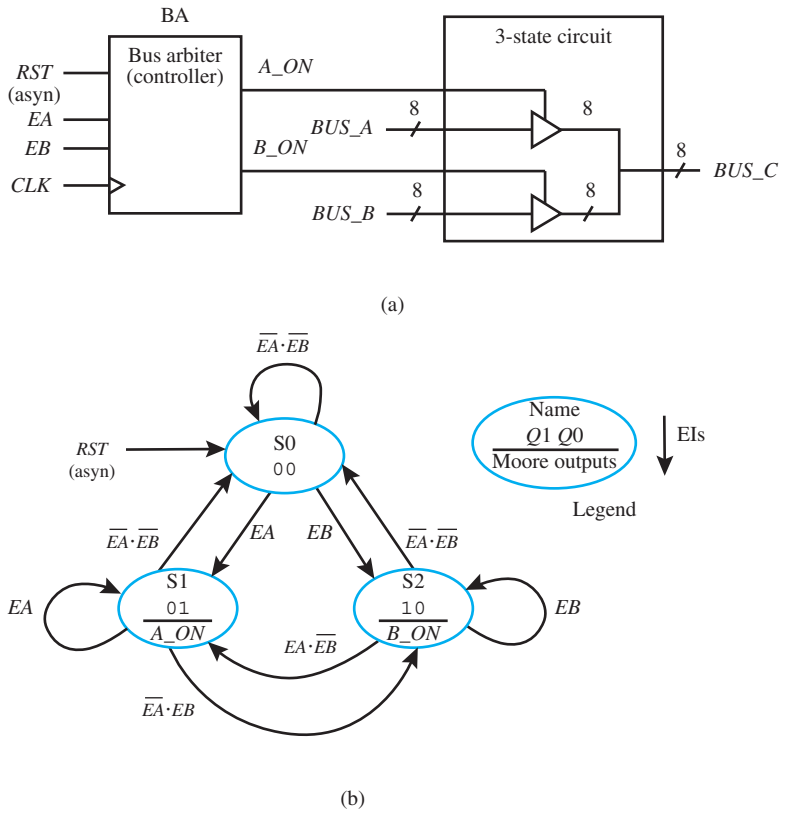


FIGURE P9.66

9.74 On which clock tick is the output of a synchronizer circuit applied to the input of a complex state machine? Briefly discuss your answer.

9.75 Write a mixed architecture declaration for the state machine in Figure P9.75 using the two-process PS/NS method. The synchronous *CE* (clock enable) input for the design must be obtained from the asynchronous input *A_CE* via a synchronizer circuit provided in the VHDL code. Assign values to the flip-flop outputs in a selected signal assignment (with-select-when statement). Let the name of the design entity be OHD_counter_A_CE.

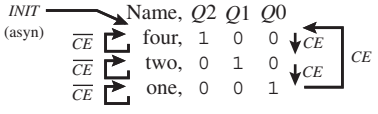


FIGURE P9.75

9.76 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the state machine in Figure P9.75.

9.77 Combine your code for problems 9.75 and 9.76 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

9.78 Write a mixed architecture declaration for the one-hot encoded state machine controller circuit and the controlled circuit in Figure P9.78 using the two-process PS/NS method. The controller state diagram in Figure P9.78a represents a state machine that is controlling the controlled circuit, which is represented by the truth table in Figure P9.78b. This design can be thought of as a decomposition of a more complicated state machine with additional Moore outputs (*M*4, *M*3, *M*2, *M*1, and *M*0) represented by the outputs in the truth table. In this decomposed design, the Moore outputs are generated by the controlled circuit. Observe that there is one flip-flop dedicated to each state, and the Show Where True signal convention is used. Assign values to the flip-flop outputs and the Moore output *Z*2 in selected signal assignments (SSAs). Use a selected signal assignment (SSA) for the controlled circuit. Assume that the input signal *STOP* must be obtained from the asynchronous

input signal *A_STOP* via a synchronizer circuit provided in the VHDL code. Let the name of the design entity be OHD_counter_A_STOP.

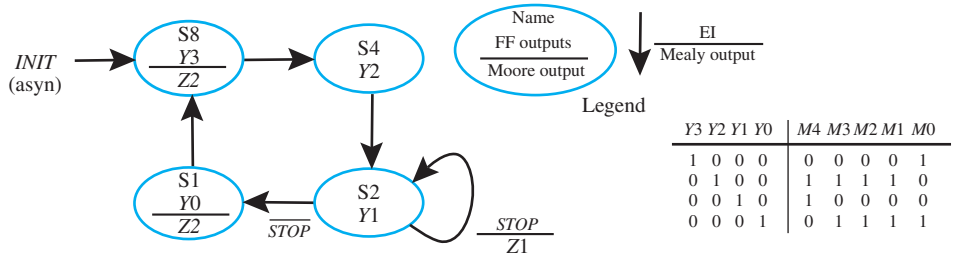


FIGURE P9.78

9.79 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the state machine in Figure P9.78.

9.80 Combine your code for problems 9.78 and 9.79 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

Section 9.11 Additional State Machine Design Methods

9.81 Show complete VHDL code for the marching 1s counter represented by the state diagram in Figure P9.81. Notice that this counter counts in the following sequence in decimal 0, 4, 6, 7, 3, 1, 0, ... In binary the sequence is 000, 100, 110, 111, 011, 001, 000, ... The counter gets its name from the binary sequence, where it appears that 1s are marching from left to right when the signal *EN* (Enable) is asserted—that is, *EN* is 1. When *EN* is 0, the counter holds the present count. Show a simulation for your design to verify that your VHDL code works. Name the design entity M1sC.

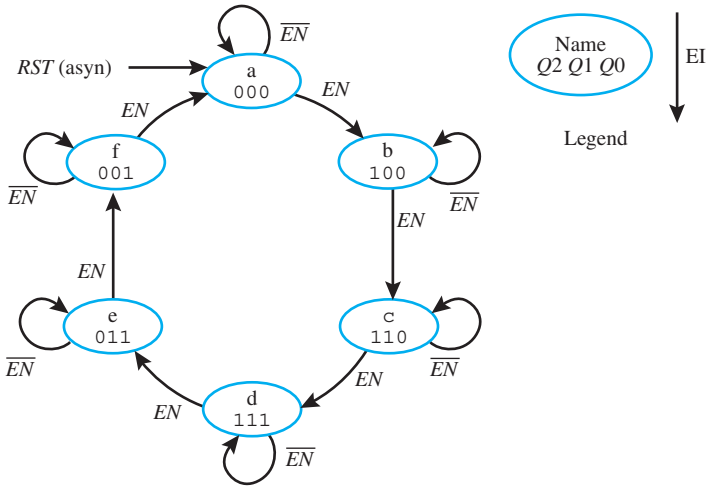


FIGURE P9.81

a. Show your design using the two-process PS/NS method.
b. Show your design using the two-assignment PS/NS method.

9.82 Show complete VHDL code for the compact encoded one-pulse circuit represented by the state diagram in Figure P9.82. When input *X* is 1, a single pulse is generated. Show a simulation for your design to verify that your VHDL code works. Name the design entity OPC_CE.
a. Show your design using the two-process PS/NS method.
b. Show your design using the two-assignment PS/NS method.

9.83 Show complete VHDL code for the one-hot encoded one-pulse circuit represented by the state diagram in Figure P9.83. When input *X* is 1, a single pulse is generated. Show a simulation for your design to verify that your VHDL code works. Name the design entity OPC_OHE.

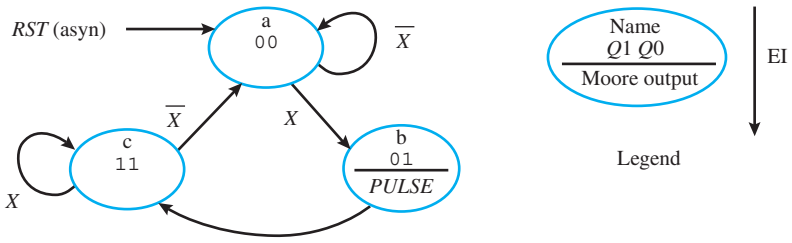


FIGURE P9.82

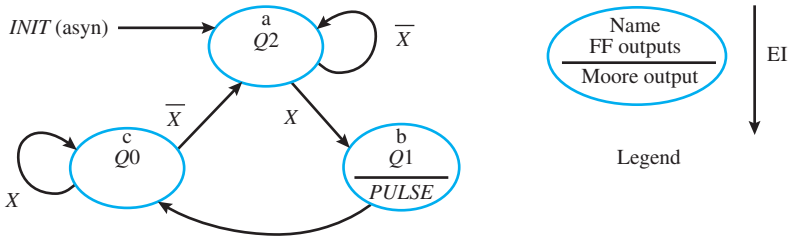


FIGURE P9.83

a. Show your design using the two-process PS/NS method.
b. Show your design using the two-assignment PS/NS method.
c. Show your design using the hybrid PS/NS method.

9.84 Show complete VHDL code for the simple fixed combinational lock circuit represented by the state diagram in Figure P9.84. The combination for the lock is *CB*5 *CB*4 *CB*3 *CB*2 *CB*1 *CB*0 = 101101. When the input for the combination is applied, the signal *OPEN_L* is asserted—that is, *OPEN_L* = 1. If the correct inputs are not applied, then output signal *ERROR* is asserted—that is, *ERROR* = 1. Show a simulation for your design to verify that your VHDL code works. Name the design entity FCLC.

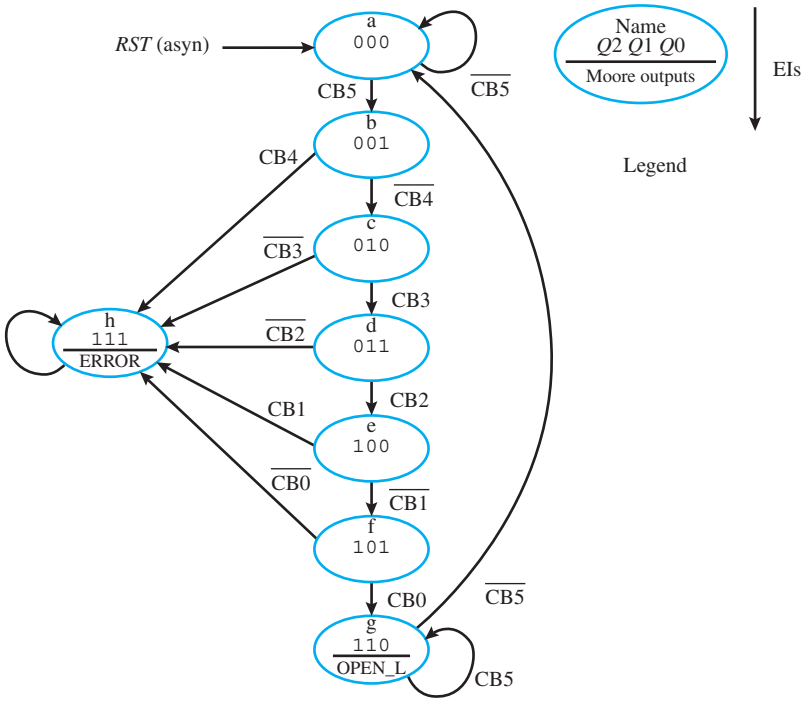


FIGURE P9.84

a. Show your design using the two-process PS/NS method.
b. Show your design using the two-assignment PS/NS method.
c. Show your design using the hybrid PS/NS method.

9.85 Show complete VHDL code for the compact encoded two-pulse circuit represented by the state diagram in Figure P9.85. When input *X* is 1, two pulses are generated. Show a simulation for your design to verify that your VHDL code works. Name the design entity TPC_CE.

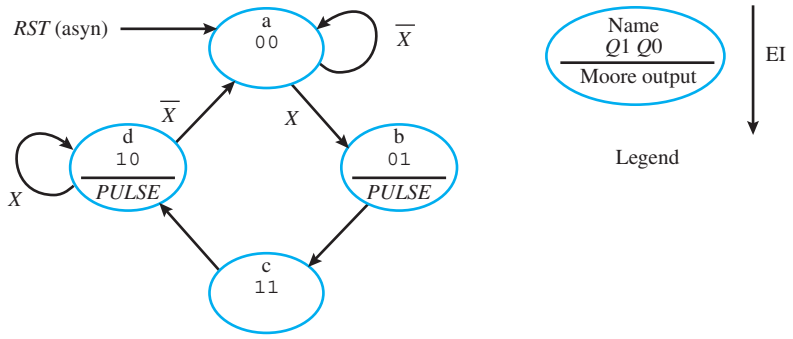


FIGURE P9.85

a. Show your design using the two-process PS/NS method.
b. Show your design using the two-assignment PS/NS method.
c. Show your design using the hybrid PS/NS method.

9.86 Show complete VHDL code for the one-hot encoded two-pulse circuit represented by the state diagram in Figure P9.86. When input *X* is 1, two pulses are generated. Show a simulation for your design to verify that your VHDL code works. Name the design entity TPC_OHE.

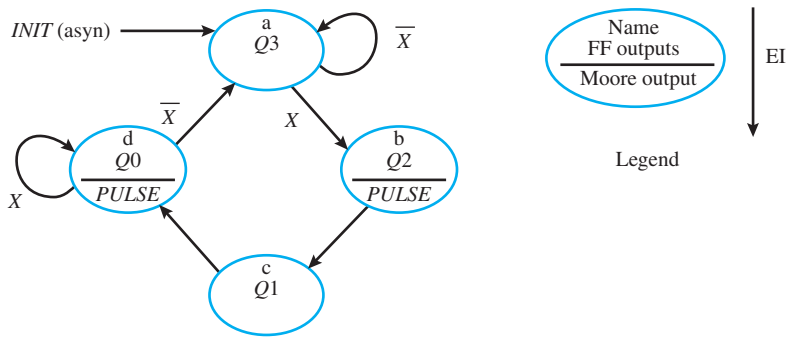
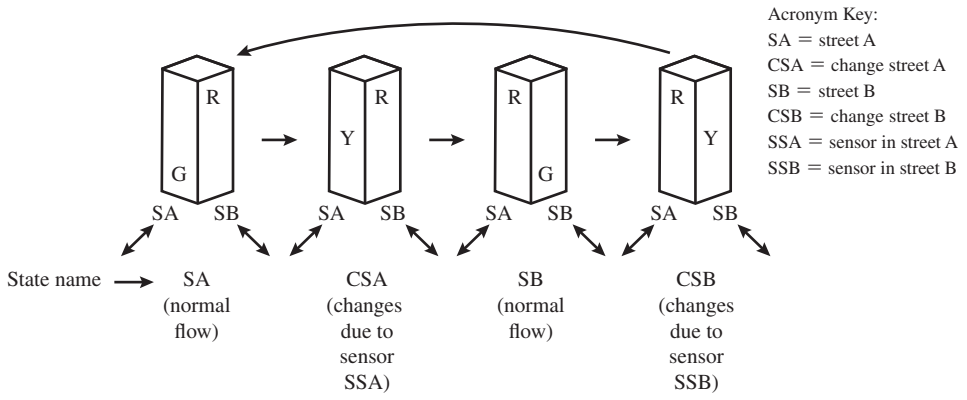


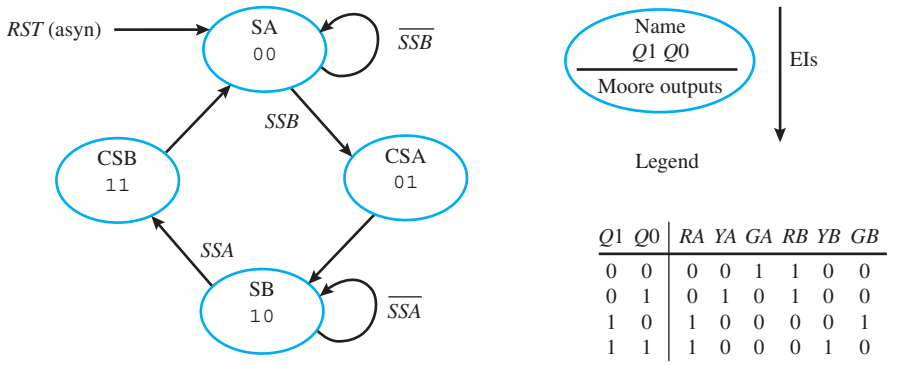
FIGURE P9.86

a. Show your design using the two-process PS/NS method.
b. Show your design using the two-assignment PS/NS method.
c. Show your design using the hybrid PS/NS method.

9.87 Show complete VHDL code for a very basic traffic light controller and controlled circuit. Figure P9.87a shows a drawing for the traffic light sequence and an acronym key. Sensors are assumed to be embedded on each side of the intersection in street A and also on each side of the intersection in street B. The signal SSA is the sensor for street A, and the signal SSB is the sensor for street B. A sensor is asserted—that is, the sensor output is 1—when a car passes over the sensor. Figure P9.87b shows the state diagram for the traffic light controller and the truth table for the controlled circuit. The controlled circuit shows the truth table for light signals for each of the streets—that is, RA, YA, and GA are the light signals for street A while, RB, YB, and GB are the light signals for street B via Figure P9.87a. Show a simulation for your design to verify that your VHDL code works. Name the design entity TLC_CC.



(a)



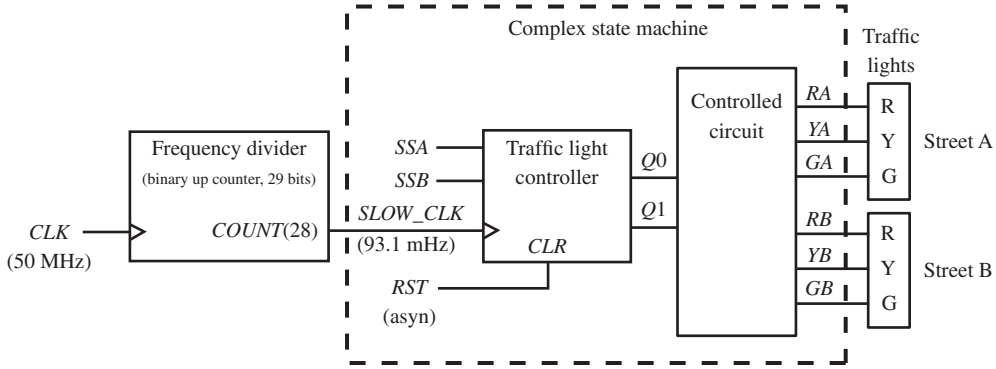
(b)

FIGURE P9.87

a. Show your design using the two-process PS/NS method.
b. Show your design using the two-assignment PS/NS method.
c. Show your design using the Hybrid PS/NS Method.

9.88 Show complete VHDL code for the complex state machine system shown in Figure P9.88a. The frequency divider is added to the complex state machine to slow down the clock so that you can observe the circuit working in hardware on a BASYS 2 board or a NEXYS 2 board. In this problem, the complex state machine in problem 9.87 is used. If you did not work problem 9.87, then you need to work that problem first to verify that it works correctly and then include the frequency divider. If you did work problem 9.87, simply expand your code to include the frequency divider. Use the arithmetic method for the design of the frequency divider. Name the design entity TLC_CC_with_f_divider.

To observe your design working on a BASYS 2 board or a NEXYS 2 board make a label to set above the single green LEDs as shown in Figure P9.88b to indicate their intended color and/or outputs. Use colored highlighters to highlight *R* (red), *Y* (yellow), and *G* (green) if you have these colors.



Recommended I/O on BASYS 2 board or NEXYS 2 board for this design: *SSA*(**SW1**), *SSB*(**SW0**), *RST*(**BTN3**), *RA*(**LD7**), *YA*(**LD6**), *GA*(**LD5**), *RB*(**LD4**), *YB*(**LD3**), *GB*(**LD2**), *Q*1(**LD1**), and *Q*0(**LD0**).

(a)

Street A			Street B				
R	Y	G	R	Y	G	*Q*1	*Q*0
LD7	LD6	LD5	LD4	LD3	LD2	LD1	LD0

(b)

FIGURE P9.88

a. Show the complex state machine using the two-process PS/NS method.
b. Show the complex state machine using the two-assignment PS/NS method.
c. Show the complex state machine using the hybrid PS/NS method.

Chapter 10

Basic Computer Architectures

Chapter Outline

10.1 INTRODUCTION

In this chapter you will learn how to identify each of the parts of a **Harvard-type computer** and **RISC architecture**, and how it compares to a **Princeton-type computer** and **CISC architecture**. You will be introduced to the **programmer's register model** (PRM) and the **instruction set architecture** (ISA) of a very basic Harvard-type computer called VBC1 (Very Basic Computer 1). A block diagram and partial schematic diagram are provided for VBC1. The format for writing assembly language programs for VBC1 is also presented. You will be introduced to an editor, assembler, and simulator—which we call EASY1—that will allow you to test your assembly language programs.

10.2 GENERIC DATA-PROCESSING SYSTEM OR COMPUTER

We refer to a **digital computer** as a device in which a **program** (a series of instructions or commands) can be stored and then executed serially—that is, one instruction followed by the next. Each instruction consists of a set of bits referred to as **machine code**. Instructions are performed by executing a series of transfer functions or micro-operations, where each transfer function represents a digital circuit. This will be discussed later when we talk about computer design. The **instruction set**—that is, the list of instructions (or commands) that the computer can execute—is selected by the computer designer(s). After you learn how to design VBC1, you can select your own instruction set and design your own computer. The instruction set for VBC1 only contains eight instructions. This makes learning the instruction set rather easy, as you will see.

Figure 10.1 shows the functional units of a generic data-processing system or computer.

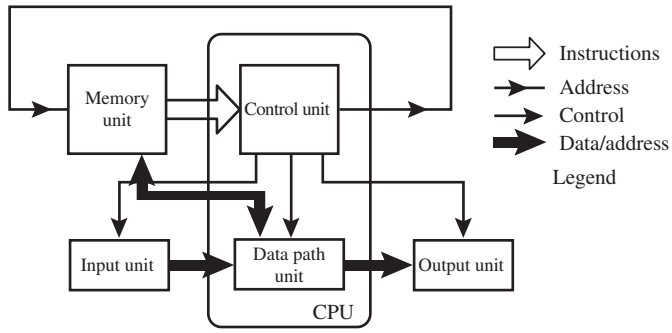


FIGURE 10.1 Functional units of a generic data-processing system or computer

The program for the generic computer is stored in the memory unit, and instructions are supplied to the control unit. The control unit directs the operation of the computer via the control lines. The data path unit receives external inputs from the input unit. The data path unit supplies external outputs to the output unit. The data path unit contains registers for temporary data storage and performs arithmetic operations, logic operations, shift operations, and so on. The control unit and the data path unit make up what is generally called the CPU (central processing unit).

If you are using a computer that does not have a particular instruction that you need, you may be able to create a software algorithm to emulate the instruction by executing a series of instructions in the instruction set that is provided. This technique is used at the end of this chapter to create a subtraction instruction and also an unconditional jump instruction from the set of existing instructions for VBC1.

Rather than adding a series of instructions via a software algorithm, you can add a computer peripheral to do a task. Suppose we wanted to use a keyboard to input data. A software algorithm could be created to read and detect which button is pressed. An alternate solution would be to add a peripheral (a keyboard encoder) to handle the conversion.

Suppose we wanted to drive a 7-segment display to output data. A software algorithm could also be created to handle this conversion. An alternate solution would be to add a peripheral (a BCD to 7-segment decoder) to handle the conversion. The trend today is to add computer peripherals on the same die or chip to save designers the trouble of adding them externally. Computer execution time can be improved by adding peripherals to the same die. This is the approach we use in this book for the BCD to 7-segment display decoder.

10.3 HARVARD-TYPE COMPUTER AND RISC ARCHITECTURE

Figure 10.2 shows the 6 basic units that make up a Harvard-type computer.

The symbols that make up each unit in Figure 10.2 are simple memory joggers. The instruction memory unit and the data memory unit are represented as separate rectangular blocks that contain arrays of flip-flops for storage. The control unit is shown as a state machine via a state diagram. The data path unit is shown as an ALU (arithmetic logic unit) schematic symbol. The input unit is shown as **inputs** from slide switches and push-button switches. The output unit is shown as **outputs** to light emitting diodes (LEDs) and a 7-segment display. The control unit, the data memory unit, and the data path unit make up the CPU.

Things you should know about the basic units of a Harvard-type computer and a RISC (reduced instruction set computer) architecture:

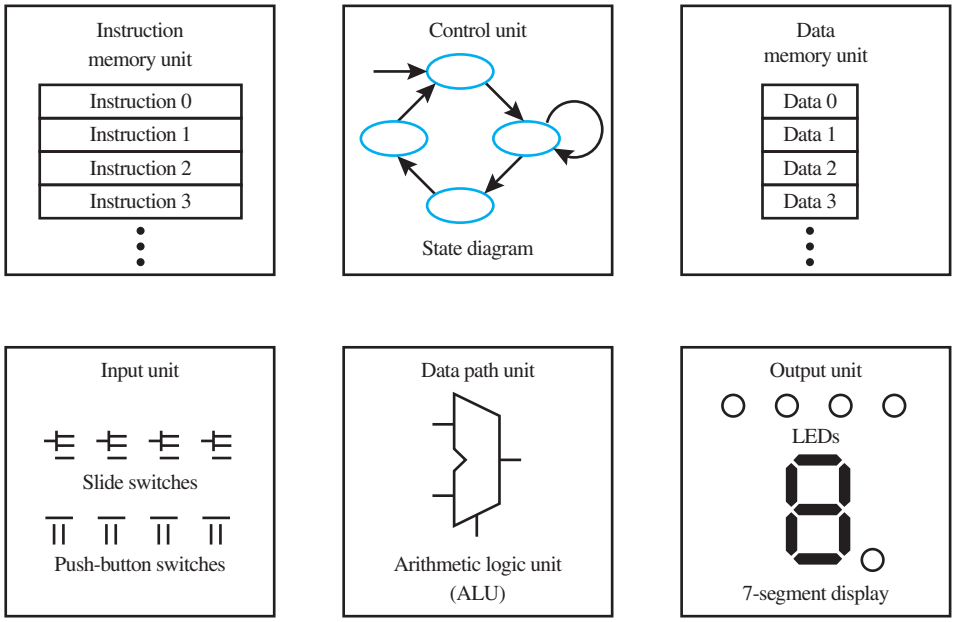


FIGURE 10.2 Six basic units that make up a Harvard-type computer

- First and foremost, a Harvard-type computer has a separate **instruction path** and **data path**. One bus is used for the instruction path, and a second bus is used for the data path. VBC1 is a Harvard-type computer that contains an instruction path of 8 bits and a data path of 4 bits. Howard Aiken is credited for inventing the Harvard-type computer in the 1940s at Harvard University. The computer was named the Mark-III, and an improved version was named Mark-IV.
- The instruction memory unit is where a program is saved for execution.
- The registers in the data path unit (or in the data memory unit) are used for temporary data storage during program execution.
- Harvard-type computers usually have a large number of registers in the CPU dedicated for fast temporary data storage. (VBC1 is a Harvard-type computer that has only two 4-bit registers for temporary data storage.)
- The number of bits for the instruction memory unit is usually larger than the number of bits for the data memory unit as emphasized by the widths of the rectangular blocks in the instruction memory unit compared to the widths of the rectangular blocks in the data memory unit.
- Operands usually have to be placed in registers in the CPU. This is sometimes referred to as a **register-register architecture**.
- The number of bits for the instruction memory unit is chosen so that each instruction resides at only one address (or location) in the instruction memory unit—that is, each instruction has a fixed length or fixed number of bits. Commercial Harvard-type computers, called **microcontrollers**, are RISC architectures. A microcontroller is a computer on a chip. These types of computers have a small number of simple instructions, and each instruction is generally executed in just one **machine cycle** or **instruction cycle**. A machine cycle represents a specified number of clock cycles.
- RISC architectures normally have few **addressing modes** for their instructions. Addressing modes are different ways in which operands are specified—that is, immediate, direct, indirect, and so on.

10.4 PRINCETON (VON NEUMANN)-TYPE COMPUTER AND CISC ARCHITECTURE

How does a Harvard-type computer and a RISC architecture compare to a Princeton-type computer? Figure 10.3 shows the five basic units that make up a Princeton-type computer.

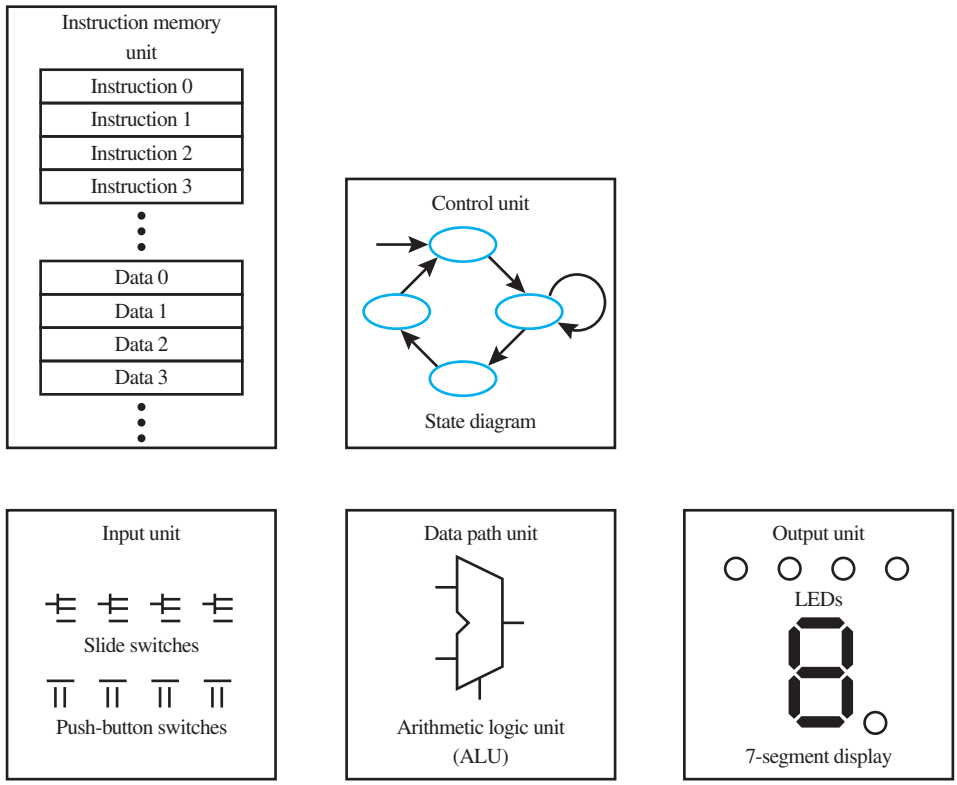


FIGURE 10.3 Five basic units of a Princeton (or von Neumann)-type computer

Things you should know about the basic units of a Princeton (or von Neumann)-type computer:

- First and foremost, a Princeton (or von Neumann)-type computer has a single memory unit for storing programs and data; thus, it has only one memory bus to communicate with the CPU. This is sometimes referred to as the von Neumann bottleneck. Because instructions and data must be accessed in sequential order, a Princeton-type computer requires a higher clock frequency to execute an equivalent number of instructions compared to a Harvard-type computer. John von Neumann is credited for inventing the first stored program computer in 1947 at the Institute for Advanced Study in Princeton, New Jersey, which is just a short distance away from Princeton University. The computer was called the IAS machine.
- The number of bits for the instruction memory is the same as the number of bits for the data memory because the same memory unit is used for both instructions and data.
- Princeton-type computers normally have just a few registers in the CPU dedicated for fast temporary data storage.
- Operands can either be placed in a dedicated register called the accumulator or be placed in instruction memory. This is sometimes referred to as **register-memory architecture**.
- Each instruction in a Princeton-type computer often requires more than one address (or location) in the instruction memory unit. Commercial Princeton-type computers called **microcontrollers** are CISC (complex instruction set computer) architectures. These types

of computers have a large number of simple as well as complex instructions. The instruction length is not fixed, and complex instructions generally require more clock cycles to complete their execution compared to simple instructions.

- CISC architectures usually have many different addressing modes for the instructions.

Many different types of hybrid architecture computers exist today. These styles borrow from the Harvard-type and the Princeton-type computers in addition to adding other features that are not found in either the Harvard or Princeton computer. A RISC architecture can be designed in a Princeton-type computer, while a CISC architecture can be designed in a Harvard-type computer. We place our emphasis on the Harvard-type computer with a RISC architecture.

10.5 OVERVIEW OF VBC1 (VERY BASIC COMPUTER 1)

Figure 10.4 shows a rather simple but general block diagram for VBC1. VBC1 is a Harvard-type computer.

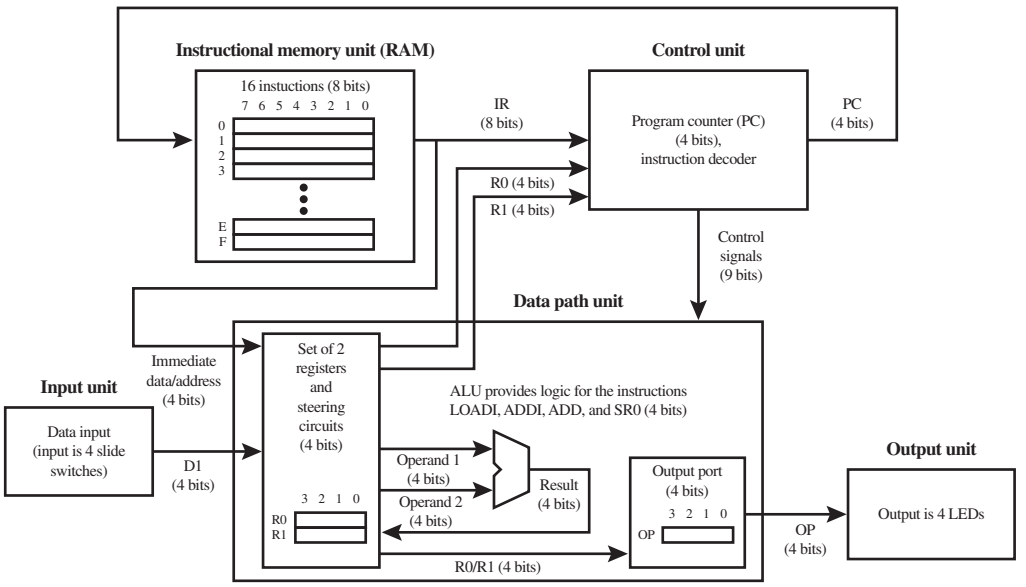


FIGURE 10.4 Block diagram for VBC1

Observe that the data path unit is 4 bits wide, while the instruction memory unit is 8 bits wide. The instruction memory unit can contain only 16 instructions because the Program Counter (PC) only has 4 bits ($2^4 = 16$ locations or addresses). The instructions stored in the instruction memory unit tell the computer what operations to do. The external input for VBC1 is 4 slide switches and the external output is 4 LEDs. The ALU provides the logic for the instructions LOADI, ADDI, ADD, and SR0. Four additional instructions are also available: IN, OUT, MOV, and JNZ. VBC1 is a basic programmable 4-bit computer with 8 different instructions with 22 variations. You will learn how to program VBC1 with assembly language in Experiment 10 in Appendix A using the editor/assembler/simulator 1, which we call EASY1. For a tutorial on EASY1, see Appendix D.

10.6 DESIGN PHILOSOPHY OF VBC1

VBC1 is a very simple computer with just a few instructions that can be easily learned. VBC1 is designed using the hardware description language VHDL to allow the hardware implementation of VBC1 to be downloaded into a FPGA for testing.

A simple loading program counter (LPC) is used to load instructions into instruction memory. The instruction memory for VBC1 is 16 by 8: only 16 instructions can be placed in the instruction memory and each instruction is only 8 bits wide. A multiplexed display system is used to display the program counter (PC) and the instruction register (IR) when loading the instructions via machine code.

We refer to the normal program counter (PC) for VBC1 as the running program counter (RPC) to distinguish it from the simple loading program counter. The PC provides the correct address when instructions are executed. An assembly language program can be executed one instruction at a time (single stepped) or run at the specified frequency of the signal *SPEED*.

Because the data path consists of 4 bits, and the instruction path consists of 8 bits, VBC1 has the form of a Harvard-type computer. The input is just 4 bits supplied by 4 dedicated slide switches. The output is just 4 bits connected to four dedicated single LEDs. This keeps the design very simple but is restricted to one input port and one output port. VBC1 does not have a data memory unit.

An annotated schematic is provided for each module of VBC1. VHDL code is written for each logic module to form a final hardware design. The hardware design can be compiled and downloaded into an FPGA. The hardware design for VBC1 can be tested by writing an assembly language program, loading the machine code for the program into instruction memory, and executing the program by either single stepping through the program or running the program at the specified frequency of the signal *SPEED*.

We designed VBC1 with VHDL using a flat design approach, which consists of part dataflow design style and part behavioral design style. A structural design style could be used for various modules. Schematic capture could be also used to draw each module by using library parts supplied by the schematic capture program. VBC1 could also be designed with Verilog HDL. As you can see, there are many ways the hardware design for VBC1 could be generated that differ from the way we elected to design VBC1.

Figures 10.5a and 10.5b show a partial schematic for VBC1.

The partial schematic for VBC1 provides a little more detail of some of the actual circuits we will cover in the following chapters. Figure 10.5a shows the data path circuit (4 bits) and Figure 10.5b shows the control (9 bits) and instruction path circuit (8 bits). The entire schematic or circuit diagram will be implemented via VHDL code. The I/O key shows the slide switches, push-button switches, and LEDs that are used on either a BASYS or NEXYS board (see Appendix C).

We will briefly discuss the circuits in Figure 10.5a and b so that you may gain a better understanding of the hardware and the instructions that VBC1 can execute. The circuit in Figure 10.5a can be described as follows:

- MUXs (multiplexers) act as steering circuits.
- Register R0 and register R1 are data registers with 4 bits.
- MUXs 1, 2, and 3 provide the data path for the data registers labeled register R0 and register R1.
- The ALU is where the instructions LOADI, ADDI, ADD, and SR0 are implemented.
- MUXs 2, 4, and 5 provide the data path for the two 4-bit operands supplied to the ALU.
- The block labeled Input switches 4 bits provides the data input.
- The block labeled Output port drives the 4 LED outputs.

The circuit in Figure 10.5b can be described as follows:

- MUX 6 is enabled to allow the JNZ instruction to pass a new address to the Adder in the program counter (PC).
- If MUX6 is disabled, then the current value of the PC is incremented to point to the next instruction.
- The PC generates the address for the instruction memory, which is where the instructions are placed.

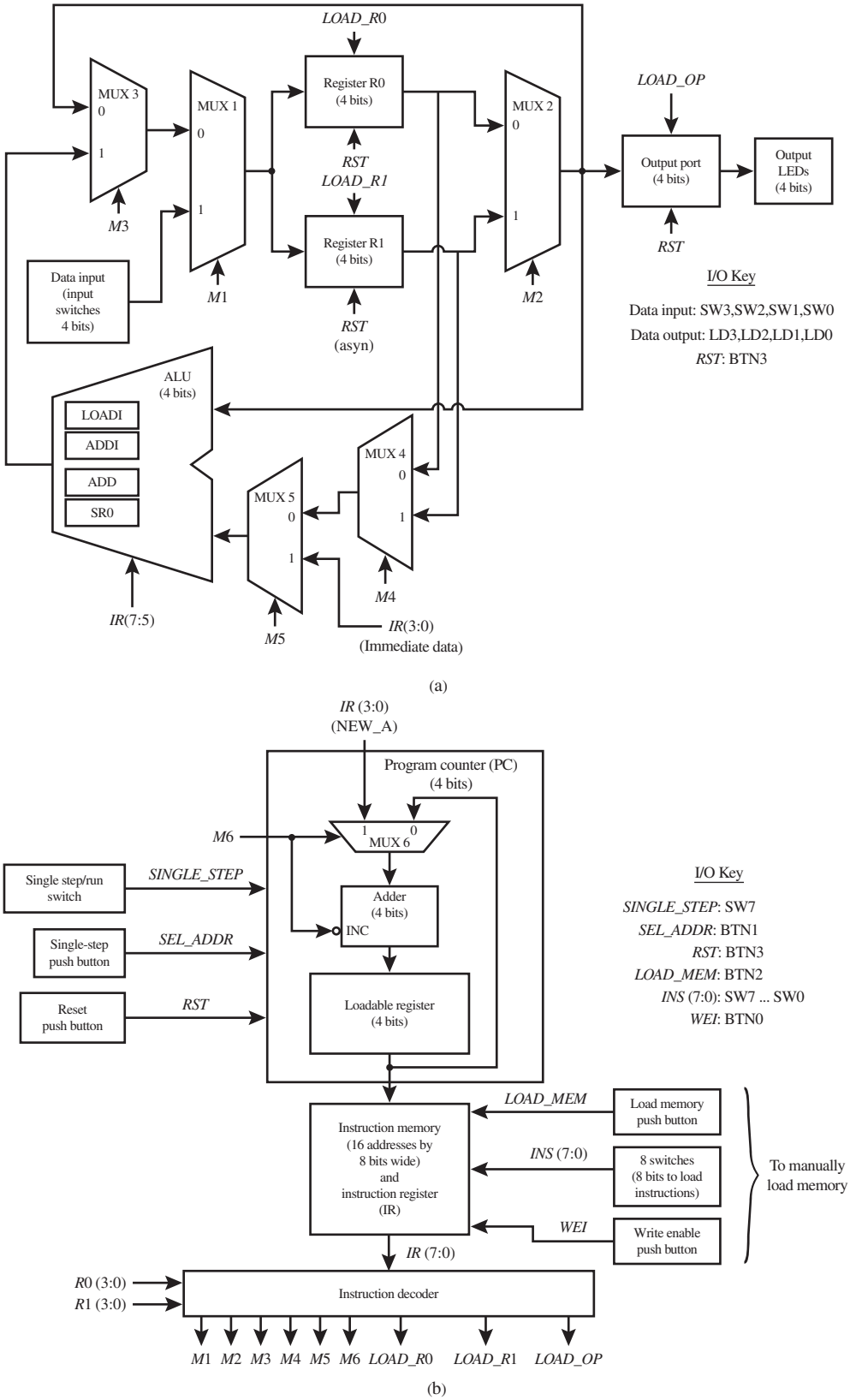


FIGURE 10.5 Partial schematic for VBC1: (a) data path circuit; (b) control and instruction path circuit

- The instruction memory provides the instructions for VBC1 via the instruction register (IR).
  For this design, the instruction register is simply the current contents of the instruction memory.
- The instruction decoder decodes the IR to provide the control for driving the MUXs (1 through 6) and the data registers R0, R1, and the output port.
- The instructions IN, OUT, MOV, and JNZ are implemented by the way the circuit is formed—that is, the architecture—controlled by the instruction decoder.

The rest of the book is about the design and programming of VBC1. We will begin with the programming of VBC1.

10.7 PROGRAMMER'S REGISTER MODEL FOR VBC1

A programmer's register model (PRM) shows the input, the output, the important registers required for a computer, the program counter, and the bit range of each register (i.e., 3 down to 0 for R0). The PRM does not provide the details of how the computer is designed or implemented in logic. The design of VBC1 in VHDL will be presented later.

Figure 10.6 shows a programmer's register model for VBC1.

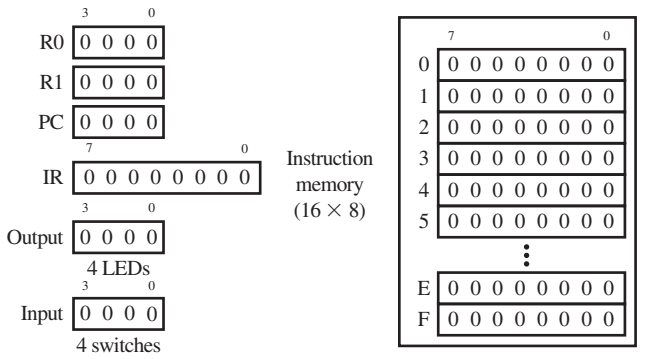


FIGURE 10.6
Programmer's register model for VBC1

The PRM, or simply the register model, shows the registers R0 (register 0) and R1 (register 1), the Output, and the Input. Because OP (output port) is connected directly to the Output (4 LEDs) in VBC1, OP is not included in the PRM. The register model also shows the PC (program counter), the IR (instruction register), and the instruction memory. To run a program, the machine code or object code for the program is first placed into instruction memory. The PC is cleared to 0000, and the computer can either be single stepped or run to execute each instruction.

For VBC1, the PC provides the 4-bit address of the next instruction to be executed. The value in the PC always provides or points to the address of the next instruction to be executed in the instruction memory. The value in the IR always provides or shows the machine code for the instruction that will be executed next.

Register 0 (R0) and register 1 (R1) provide temporary storage for a program (4 bits for VBC1). The Output is where the computer provides the external output (again, 4 bits for VBC1), and the Input is where the computer gets its external data (4 bits for VBC1). The number of bits for the width of the data is the same as the data path for the computer (4 bits for VBC1). The output port is simply the gateway to the four output LEDs. The 4-bit value placed in the OP is directly outputted to the four LEDs for VBC1.

10.8 INSTRUCTION SET ARCHITECTURE FOR VBC1

An instruction set architecture (ISA) is a programmer's view of a computer. The ISA provides detailed information of all the commands (or instructions) that a computer can execute. VBC1 has eight different instructions with 22 variations. Each instruction is presented in three different forms: (1) **assembly language form (ALF)**, (2) **transfer function form (TFF)**, and (3) **machine code form (MCF)**. A programmer writes assembly language programs for VBC1 using the assembly language form. The transfer function form (sometime called register transfer language form) is used to explain exactly what each instruction does. The machine code form shows the instructions in terms of 1s and 0s. Remember that all digital logic circuits work with only binary data. The machine code form is loaded into the instruction memory for execution.

Table 10.1 shows the assembly language form for all the instructions for VBC1 with a brief description in English of each instruction. The instructions in Table 10.1 are shown in the order in which they will be presented. **DR** represents the **destination register**, which may be either R0 or R1. **SR** represents the **source register**, which may be either R0 or R1.

TABLE 10.1 The instructions in assembly language form and a brief description of each instruction for VBC1

Instructions in assembly language form (ALF)	Brief description of the instructions
IN DR	Input the 4 slide switch values (data input) into DR
OUT DR	Output the contents of DR to output port (OP) to four LEDs
MOV DR,SR	Move the contents of SR into DR
LOADI DR,Data	Load immediate Data into DR
ADDI DR,Data	Add immediate Data to the contents of DR, and place the result in DR
ADD DR,SR	Add the contents of SR to the contents of DR, and place the result in DR
SR0 DR,SR	Shift the contents of SR to the right 1 bit with 0 fill of the MSB, then place the result into DR
JNZ DR,Address	Jump if the contents of DR is not 0 to Address, else execute the next instruction

Table 10.2 shows the instructions in transfer function form. The transfer function form provides a concise description in a mathematical or symbolic form of each instruction.

TABLE 10.2 The instructions in assembly language form and in transfer function form for VBC1

Instructions in assembly language form (ALF)	Instructions in transfer function form (TFF)
IN DR	DR ← DI(3:0)
OUT DR	OP ← DR
MOV DR,SR	DR ← SR
LOADI DR,Data	DR ← IR(3:0) where Data = IR(3:0)
ADDI DR,Data	DR ← DR + IR(3:0) where Data = IR(3:0)
ADD DR,SR	DR ← DR + SR
SR0 DR,SR	DR ← 0 SR(3:1)
JNZ DR,Address	PC ← IR(3:0), if DR ≠ 0 where Address = IR(3:0) else PC ← PC + 1

The transfer functions for each of the instructions work as follows: the 4-bit content on the right side of the **gets**, **takes on**, or **goes into symbol**, "←", is simply transferred to the register on the left side of the symbol. The previous value of the register on the left side of the symbol is overwritten.

VBC1 is a very basic small 4-bit digital computer. Table 10.3 shows the complete detailed instruction set for VBC1 in alphabetical order for easy reference. The machine code form has been added to the table to provide the binary or object code for each instruction.

TABLE 10.3 Complete detailed instruction set for VBC1 in alphabetical order in assembly language form (ALF), transfer function form (TFF), and machine code form (MCF)

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
ADD DR,SR	DR ← DR + SR	7:5 OPCODE; Dest.Reg; Source Reg 7 6 5 4 3 2 1 0 0 1 0 0/1 0/1 0 0 0 0 = R0 0 = R0 1 = R1 1 = R1
ADDI DR,Data	DR ← DR + IR(3:0)	7 6 5 4 3 2 1 0 0 1 1 0/1 D D D D 0 = R0 1 = R1 3:0 Immediate Data (Source)
IN DR	DR ← DI(3:0)	7 6 5 4 3 2 1 0 1 0 1 0/1 0 0 0 0 0 = R0 1 = R1
JNZ DR,Address	PC ← IR(3:0), if DR ≠ 0 else PC ← PC + 1	7 6 5 4 3 2 1 0 1 1 1 0/1 A A A A 0 = R0 1 = R1 3:0 Address (Destination)
LOADI DR,Data	DR ← IR(3:0)	7 6 5 4 3 2 1 0 0 0 1 0/1 D D D D 0 = R0 1 = R1 3:0 Immediate Data (Source)
MOV DR,SR	DR ← SR	Source Reg 7 6 5 4 3 2 1 0 0 0 0 0/1 0/1 0 0 0 0 = R0 0 = R0 1 = R1 1 = R1
OUT DR	OP ← DR	7 6 5 4 3 2 1 0 1 1 0 0/1 0 0 0 0 0 = R0 1 = R1 Source Reg
SR0 DR,SR	DR ← 0 SR(3:1)	7 6 5 4 3 2 1 0 1 0 0 0/1 0/1 0 0 0 0 = R0 0 = R0 1 = R1 1 = R1

In Table 10.3, the abbreviations in the column "Transfer function form (TFF)" are as follows: DR is destination register, SR is source register, IR is instruction register, DI is data input, PC is program counter, and OP is output port.

10.9 FORMAT FOR WRITING ASSEMBLY LANGUAGE PROGRAMS

Assembly language for VBC1 is written in four fields as shown in Table 10.4.

TABLE 10.4 Assembly language for VBC1 is written in four fields

Field 1	Field 2	Field 3	Field 4
[label:]	mnemonic	operands	[;comment]

The brackets are not part of assembly language and are used to represent fields that are optional. Do not use the brackets when writing programs. The first field is the **label field**, and it always begins with a letter and ends with a colon (:). The second field is the **mnemonic field**, which is the instruction. The third field is the **operands field** that contains operands that are needed by the instruction. In an instruction with two operands, the operands must be separated by a comma (,). The last field is the **comment field**, and it always begins with a semicolon (;).

A very simple program is a single instruction which contains only the mnemonic field (field 2) and the operands field (field 3) as shown in Program 10.1.

PROGRAM 10.1
Very simple program

Fields 2 & 3
IN R1

To add a label called "start" and a comment "input switch values to R1" to Program 10.1, change the program as shown in Program 10.2.

PROGRAM 10.2 Add a label called "start" and a comment "input switch values to R1" to Program 10.1

Field 1	Fields 2 & 3	Field 4
start:	IN R1	;input switch values to R1

The **assembler** (the program that is used to convert the assembly language program into machine code) will generate the same machine code for the single instruction IN R1 with two fields as the instruction with four fields. The assembler generates the address for the label "start" based on its location in the program and allows the user to refer to that address as "start." The assembler translates the instruction IN R1 into its machine code, which is 10110000. The assembler simply ignores the comment field. The comment field is used to explain what the instruction is doing so that the program is understandable to the program writer at a later time or to another person trying to understand the program for the first time.

Each program—whether it is written as one line of assembly language or multiple lines of assembly language—is treated in the same manner by the assembler. The end result of the assembler is simply to translate each instruction into machine code, which is the 1s and 0s that the computer needs to operate its logic gates.

To obtain machine code for an assembly language program, the program is written with a **text editor**. The program is assembled using an assembler. The result produced by the assembler is a program translated or assembled into machine code for a specific computer. The machine code can then be placed into the RAM to allow the instructions to be run by the computer. VBC1 can either be run by single stepping—that is, running each instruction one at a time in the

sequence that the program is written—or run at a predetermined frequency (*SPEED*) to allow its output to be observed by the output LEDs.

To write assembly language programs and check your programs, you can use EASY1 (editor/assembler/simulator 1), discussed in Appendix D (EASY1 Tutorial).

PROBLEMS

Section 10.2 Generic Data-Processing System or Computer

10.1 What is a digital computer?

10.2 Name the five units for a generic data-processing system or computer.

10.3 Which two units make up the CPU (central processing unit) in a computer?

Section 10.3 Harvard-Type Computer and RISC Architecture

10.4 How does the Harvard-type computer differ from the generic computer?

10.5 Which type of computer does the following statement describe? The instruction memory and the data memory have a different number of bits.

10.6 What does reduced instruction set computer (or RISC) architecture imply?

Section 10.4 Princeton (von Neumann)-Type Computer and CISC Architecture

10.7 Does a Princeton-type computer have a separate data memory similar to the Harvard-type computer? What problem is associated with a Princeton-type computer?

10.8 Which type of computer does the following statement describe? The instructions in memory and the data in memory have the same number of bits.

10.9 What does complex instruction set computer (or CISC) architecture imply?

Section 10.5 Overview of VBC1 (Very Basic Computer 1)

10.10 Is VBC1 classified as a Harvard-type computer or a Princeton-type computer? Give a reason for your answer.

10.11 How many bits is the data path for VBC1?

10.12 How many bits is the instruction path for VBC1?

10.13 How many different instructions does VBC1 have, and how many variations are there?

10.14 Name the units for VBC1. What is the unit that is missing in VBC1 that is contained in a Harvard-type computer?

10.15 What is the maximum number of instructions that can be used in a program for VBC1? What limits the number of instructions?

10.16 List the two temporary registers for VBC1 that can be used for operands for the ALU.

10.17 Which block in the block diagram for VBC1 provides the control signals for VBC1? In which unit does this block reside?

10.18 Which instructions are performed by the ALU for VBC1?

10.19 What instructions are available that are not performed by the ALU for VBC1?

Section 10.6 Design Philosophy of VBC1

10.20 What does MUX stand for, and how many MUXs are shown in the partial schematic for VBC1?

10.21 Which devices provide the Input and Output for VBC1?

10.22 Which block in the partial schematic for VBC1 tells the computer what operation to do?

10.23 Does VBC1 have a data memory unit?

Section 10.7 Programmer's Register Model for VBC1

10.24 What does a programmer's register model (or PRM) show about a computer?

10.25 Draw a programmer's register model for VBC1, and label the registers (including memory). Specify the bit range for each register.

10.26 What does the value in the PC (program counter) always represent?

10.27 What does the value in the IR (instruction register) always represent?

Section 10.8 Instruction Set Architecture for VBC1

10.28 Name the three different forms that are used for each instruction for VBC1.

10.29 What instruction form does a programmer use to write code for VBC1?

10.30 What is the purpose of the transfer function form for an instruction?

10.31 How are instructions presented in machine code form?

10.32 Which form for an instruction is loaded into instruction memory?

10.33 Explain what happens to the value of the register on the left side of the transfer function form after a transfer takes place.

10.34 In the complete detailed instruction set for VBC1, list what the following abbreviations represent: DR, SR, IR, DI, PC, and OP.

Section 10.9 Format for Writing Assembly Language Programs

10.35 What is the maximum number of fields that can be used when writing an assembly language instruction for VBC1? What is the minimum number of fields for VBC1?

10.36 Write the assembly language instruction for VBC1 that shifts the value in register R1 to the right 1 bit with a 0 fill bit then places the result in R0. Use a label field with the label named "start." Use a comment field with the comment "shift R1 right one bit and place result in R0."

10.37 Write the assembly language instruction for VBC1 that adds an immediate value of 5 to the value in register R1. Use a label field with the label named "next." Use a comment field with the comment "add 5 to R1."

10.38 Write the assembly language instruction for VBC1 that jumps to the address at the label "start" when the value in R0 is not 0. Use a comment field with the comment "jump to start if R0 != 0."

10.39 Write the assembly language instruction for VBC1 that moves a value placed in register R1 to register R0. Use a comment field with the comment "move R1 to R0."

10.40 Write the assembly language instruction for VBC1 that inputs the 4 slide switch values (data input) into register R1. Use a comment field with the comment "input to R1."

10.41 Write the assembly language instruction for VBC1 that outputs the value in R0 to the output port (OP) to four LEDs. Use a comment field with the comment "output R0."

10.42 Write the assembly language instruction for VBC1 that loads the immediate value 5 into register R1. Use a comment field with the comment "load 5 in R1."

10.43 Write the assembly language instruction for VBC1 that adds the value in register R1 to the value in register R0 and places the result in R0. Use a comment field with the comment "add R1 to R0."

10.44 Write the assembly language instruction for VBC1 that adds the value in register R0 to the value in register R1 and places the result in R1. Use a comment field with the comment "add R0 to R1."

10.45 Write the assembly language instruction for VBC1 that shifts the value in register R0 to the right 1 bit with a 0 fill bit then places the result in R0. Use a label field with the label named "shift." Use a comment field with the comment "shift R0 right one bit."

11 Chapter

Assembly Language Programming for VBC1

Chapter Outline

11.1 INTRODUCTION

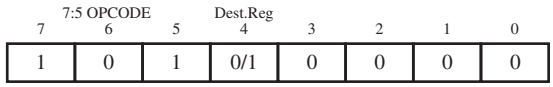
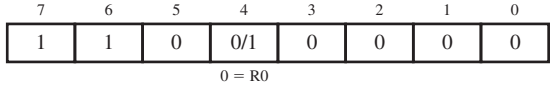
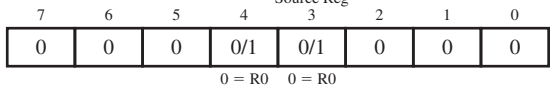
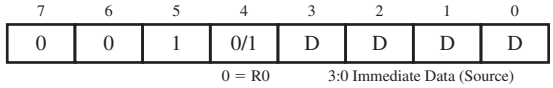
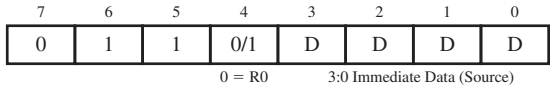
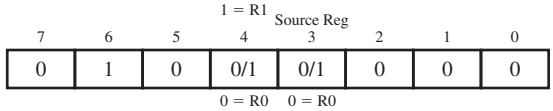
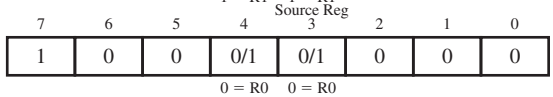
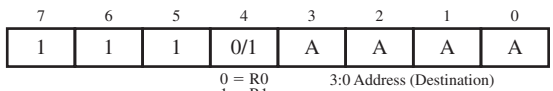
In this chapter, you will learn each of the instructions for VBC1 by reviewing each instruction in detail. You will be introduced to each instruction in **assembly language form** and in **machine code form**. The **transfer function form** is then used to show how each instruction works via the **programmer's register model (PRM)**. Simple programs will be used to illustrate how instructions work together to do a task. Programming examples and techniques will show you how to write more interesting programs for VBC1.

11.2 INSTRUCTION SET FOR VBC1

As you learned in the previous chapter, VBC1 is a very basic small 4-bit digital computer. Table 11.1 shows the complete detailed instruction set for VBC1 arranged in the order that the instructions are presented in this chapter.

In Table 11.1, the abbreviations in the column "Transfer function form (TFF)" are as follows: DR is destination register, DI is data input, OP is output port, SR is source register, IR is instruction register, and PC is program counter.

TABLE 11.1 Complete detailed instruction set for VBC1 in assembly language form (ALF), transfer function form (TFF), and machine code form (MCF)

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
IN DR	DR ← DI(3:0)	
OUT DR	OP ← DR	
MOV DR,SR	DR ← SR	
LOADI DR,Data	DR ← IR(3:0)	
ADDI DR,Data	DR ← DR + IR(3:0)	
ADD DR,SR	DR ← DR + SR	
SR0 DR,SR	DR ← 0 SR(3:1)	
JNZ DR,Address	PC ← IR(3:0), if DR ≠ 0 else PC ← PC + 1	

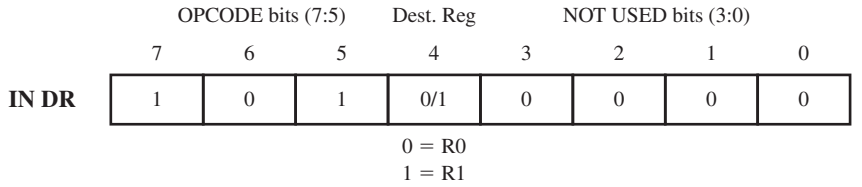
11.3 THE IN INSTRUCTION

Once you write a program for VBC1 in assembly language, the program must be assembled. To assemble a program means to convert the assembly language program into its machine code form. If a program is assembled by hand, this is referred to as **hand assembly**. A program called an **assembler** can automatically assemble an assembly language program into its machine code form. After the program is assembled, the machine code can then be placed into the instruction memory for execution either by **manual loading** or by using a program called a **loader**. In this chapter, you will learn how to do hand assembly and load the machine code into the instruction memory of the PRM (programmer's register model). The transfer function form will then be used to show how each instruction works.

To check that your hand assembly is correct, you can use EASY1 in Appendix D (EASY1 Tutorial). You can also execute a single instruction, or a program, via the EASY1 simulator to check for proper program execution.

As we discuss each instruction, we will present its assembly language form and its machine code form, beginning with the IN (**input**) instruction, as shown in Form 11.1.

FORM 11.1 The IN instruction



The IN instruction provides a means to input an external value into the computer. The external input for VBC1 is provided by four switches. The input value is either placed in R0 or R1—that is, the destination register DR. The transfer function form for IN DR is DR ← DI(3:0), where DI is data input.

The actual assembly language program for the IN R1 instruction is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 11.1.

Mnemonic field	Operands field
IN	R1

PROGRAM 11.1 Assembly language program for the IN R1 instruction

If we were using two or more input ports, there would be a second operand listed to specify the port number. With only one input, the second operand is not required.

In the machine code form, bits 7:5 or bit 7 downto bit 5 are designated the **OPCODE (operation code)** bits because they provide the binary code that specifies the operation that the instruction performs—that is, the IN instruction in this case. The OPCODE bits for the IN instruction are 101. The OPCODE bits for VBC1 were arbitrarily assigned as shown in Table 11.1. The choice of the bit values of the OPCODE bits is up to the designer. One set of assignments may be better than another, from a decoding standpoint, but *any set of assignments for the OPCODE bits will work*. A summary of the instructions and the OPCODE bit assignments for VBC1 is shown in Table 11.2.

TABLE 11.2 Summary of instructions and OPCODE bit assignments for VBC1

Instructions	OPCODE bit assignments
MOV	000
LOADI	001
ADD	010
ADDI	011
SR0	100
IN	101
OUT	110
JNZ	111

In the machine code for the IN instruction, the destination register is bit 4. When bit 4 is 0 the destination register is R0, and when bit 4 is 1 the destination register is R1. Bits (3:0) are not used and are set to 0. The unused bits can be used to provide up to 16 different input ports. Because we are only using one input port (four switches) for VBC1, we elected not to specify a port address in the assembly language form.

If we had elected to include two or more input ports, the IN instruction would be of the form IN DR,Port_address (it would require two operands) where IN DR,0 would allow an input from

port 0 while IN DR,1 would allow an input from port 1. To keep VBC1 as simple as possible, we elected to use only one port; hence, we do not have to specify the port address because there is only one port.

Assembly 11.1 shows all the versions of the IN instruction listed in assembly language and in machine code for VBC1.

Assembly language	Machine code
IN R0	10100000
IN R1	10110000

ASSEMBLY 11.1 All versions of the IN instruction

The machine code shows the bit patterns for the IN instructions as they must be placed in instruction memory to be executed. The machine code has the same OPCODE of 101 for bits 7:5 for all versions of the IN instruction. Note: When bit 4 is 0, the destination register is R0; when bit 4 is 1, the destination register is R1.

In Figure 11.1, the machine code for the instruction IN R1 is manually loaded in the instruction memory beginning at address 0 in the programmer's register model. All instruction memory locations are cleared that are not loaded with a machine code instruction. The PC is cleared to address 0 (0000) and the IR shows the machine code value at address 0 (10110000). Registers R0, R1, and the Output in the programmer's register model are also cleared. The input switches are set to the value of 0110 or 6.

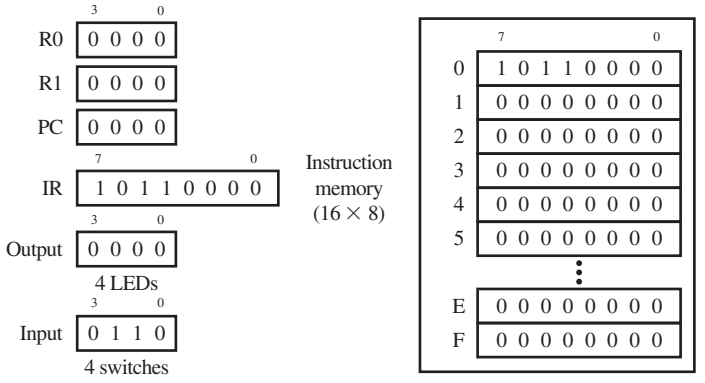


FIGURE 11.1 Result in the PRM after manually loading the instruction IN R1 into VBC1 at instruction memory address 0 with the input switches set to 6

When the instruction IN R1 is executed, the transfer function form indicates what happens to the various registers of VBC1. The transfer function form for the IN R1 instruction is R1 ← DI(3:0) or R1 ← DI(3) DI(2) DI(1) DI(0). In Figure 11.2, the value at the Input (4 switches) is transferred to register R1 after executing the instruction IN R1 as shown in the programmer's register model.

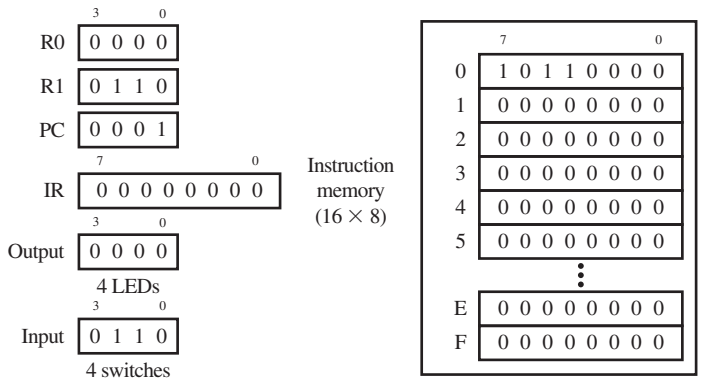


FIGURE 11.2 Result in the PRM after executing the instruction IN R1 for VBC1

In Figure 11.2, we observe that the PC is automatically incremented to the next address (address 1) after the first instruction at address 0 is executed. The IR now shows the machine code value at address 1 (00000000) in the instruction memory. The PC always points to the address for the next instruction, and the IR always lists the machine code for the next instruction. The IN R1 instruction moved the value 0110 (6) at the Input (4 switches) into register 1 (R1), which is what should have occurred. The instruction (or command) IN R1 caused no other actions to occur.

What we have illustrated in Figures 11.1 and 11.2 is how to manually load the machine code for a simple assembly language instruction IN R1 into the instruction memory of VBC1 and how the computer executes that instruction. All instructions for VBC1 can be executed in this manner to observe the resulting action taken by VBC1—that is, one at a time via single stepping. If we had executed the instruction IN R0 rather than IN R1, the value 0110 (6) would have been transferred into register 0 (R0) from the Input (4 switches). Simply changing the value of the Input prior to executing the instruction allows a different value to be transferred to the specified register.

11.4 THE OUT INSTRUCTION

Form 11.2 shows the assembly language form and the machine code form for the OUT (output) instruction.

FORM 11.2 The OUT instruction

	OPCODE bits (7:5)			Dest. Reg	NOT USED bits (3:0)			
	7	6	5	4	3	2	1	0
OUT DR	1	1	0	0/1	0	0	0	0

0 = R0
1 = R1

The OUT instruction provides a means to output an internal value from the computer. The external output for VBC1 is provided by four LEDs. The output value is either supplied by R0 or R1—that is, the destination register DR. The transfer function form for OUT DR is OP ← DR, where OP is output port.

The actual assembly language instruction OUT R1 is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 11.2.

Mnemonic fields	**Operands field**
OUT	R1

PROGRAM 11.2 Assembly language program for the OUT R1 instruction

The OPCODE bits for the OUT instruction are 110. In the machine code, the destination register is bit 4. When bit 4 is 0, the destination register is R0, and when bit 4 is 1, the destination register is R1. Bits (3:0) are not used and are set to 0. The unused bits can be used to provide up to 16 different output ports. Because we are only using one output port (four LEDs) for VBC1, we elected not to specify a port address in the assembly language form.

Assembly 11.2 shows all versions of the OUT instruction listed in assembly language and in machine code for VBC1.

Assembly language	**Machine code**
OUT R0	1 1 0 0 0 0 0 0
OUT R1	1 1 0 1 0 0 0 0

ASSEMBLY 11.2 All versions of the OUT instruction

The machine code shows the bit patterns for the OUT instructions as they must be placed in instruction memory to be executed. The machine code has the same OPCODE of 110 for bits(7:5) for all versions of the OUT instruction. Note: When bit 4 is 0, the destination register is R0; when bit 4 is 1, the destination register is R1.

Program 11.3 shows a simple assembly language program with two instructions using the instructions IN and OUT.

```
IN R1
OUT R1
```

PROGRAM 11.3 A simple assembly language program using the instructions IN and OUT

In Figure 11.3, the machine codes for the instructions IN R1 and OUT R1 in Program 11.3 are manually loaded in the instruction memory beginning at address 0 in the programmer's register model. All instruction memory locations are cleared that are not loaded with a machine code instruction. The PC is cleared to address 0 (0000), and the IR shows the machine code value at address 0 (10110000). Registers R0, R1, and the Output in the programmer's register model are also cleared. The input switches are set to the value of 0111 or 7.

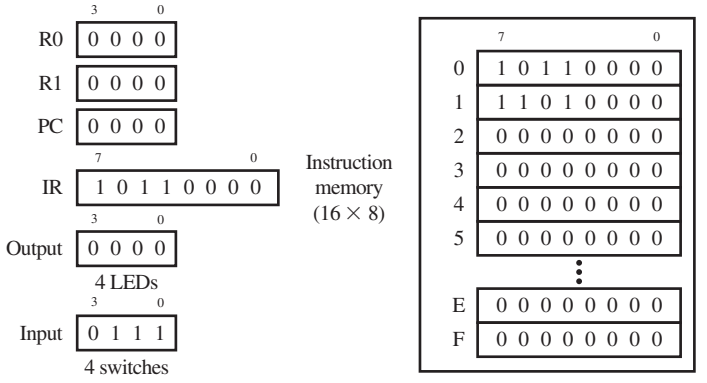


FIGURE 11.3 Result in the PRM after loading the machine code for Program 11.3 into VBC1 at instruction memory address 0 with the input switches set to 7

When the instruction IN R1 is executed, its transfer function performs the operation R1 ← DI(3:0), placing the value 7 into R1. When the instruction OUT R1 is executed, its transfer function performs the operation OP ← R1. In Figure 11.4, the value at the Input (4 switches) or 0111(7) is transferred to register R1 and then transferred to the Output (4 LEDs) after executing the instruction OUT R1, as shown in the programmer's register model.

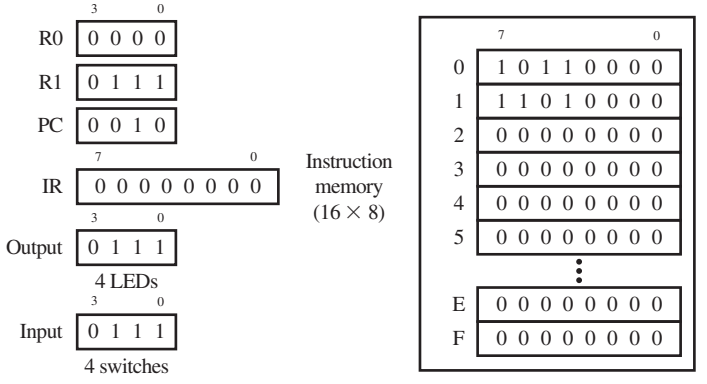


FIGURE 11.4 Result in the PRM after executing Program 11.3 for VBC1

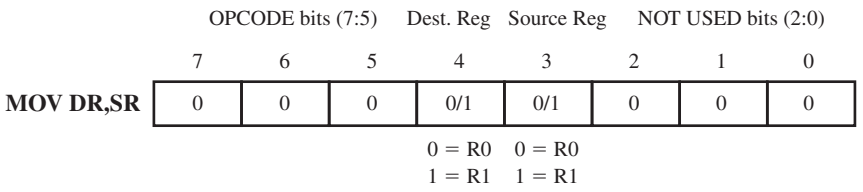
What we have illustrated in Figures 11.3 and 11.4 is how to manually load the machine code for a simple assembly language program into the instruction memory of VBC1 and how the

computer executes that program. All programs for VBC1 can be executed in this manner—that is, one at a time via single stepping, to observe the resulting actions taken by VBC1. If we had executed the assembly language program IN R0, OUT R0 rather than IN R1, OUT R1, the value 0111 (7) would have been transferred into register 0 (R0) from the Input (4 switches) and then transferred to the Output (4 LEDs). Simply changing the value of the Input (4 slide switches) prior to executing the program allows a different value to be transferred to the specified register—that is, either R0 or R1.

11.5 THE MOV INSTRUCTION

Form 11.3 shows the assembly language form and the machine code form for the MOV (**move**) instruction.

FORM 11.3 The MOV instruction



The MOV instruction moves the value in the source register to the destination register. The source register and the destination register can be either R0 or R1. The transfer function form for MOV DR,SR is DR ← SR.

The actual assembly language instruction MOV R1,R0 is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 11.4.

Mnemonic field	Operands field
MOV	R1,R0

PROGRAM 11.4 Assembly language program for the MOV R1,R0 instruction

The OPCODE bits for the MOV instruction are 000. In the machine code, the destination register is bit 4. When bit 4 is 0, the destination register is R0, and when bit 4 is 1, the destination register is R1. In the machine code, the source register is bit 3. When bit 3 is 0, the source register is R0, and when bit 3 is 1, the source register is R1. Bits (2:0) are not used and are set to 0.

Bits that are not used can be used to extend the OPCODE to provide more instructions. This concept will be utilized later when we discuss adding instructions to VBC1 to make VBC1-E. The OPCODE for VBC1 will only utilize the three bits 7:5 to keep the coding as simple as possible.

Assembly 11.3 shows all versions of the MOV instruction listed in assembly language and in machine code for VBC1.

Assembly Language	Machine code
MOV R0,R0	0 0 0 0 0 0 0 0
MOV R0,R1	0 0 0 0 1 0 0 0
MOV R1,R0	0 0 0 1 0 0 0 0
MOV R1,R1	0 0 0 1 1 0 0 0

ASSEMBLY 11.3 All versions of the MOV instruction

The machine code shows the bit patterns for the MOV instructions as they must be placed in the instruction memory to be executed. The machine code has the same OPCODE of 000 for bits 7:5 for all versions of the MOV instruction.

Program 11.5 shows a simple assembly language program using the instructions IN, MOV, and OUT.

```
IN R1
MOV R0,R1
OUT R0
```

PROGRAM 11.5 A simple assembly language program using the instructions IN, MOV, and OUT

In Figure 11.5, the machine codes for the instructions in Program 11.5 are manually loaded in the instruction memory beginning at address 0 in the programmer's register model. All instruction memory locations are cleared that are not loaded with a machine code instruction. The PC is cleared to address 0 (0000), and the IR shows the machine code value at address 0 (10110000). Registers R0, R1, and the Output in the programmer's register model are also cleared. The input switches are set to the value of 1110 or 14.

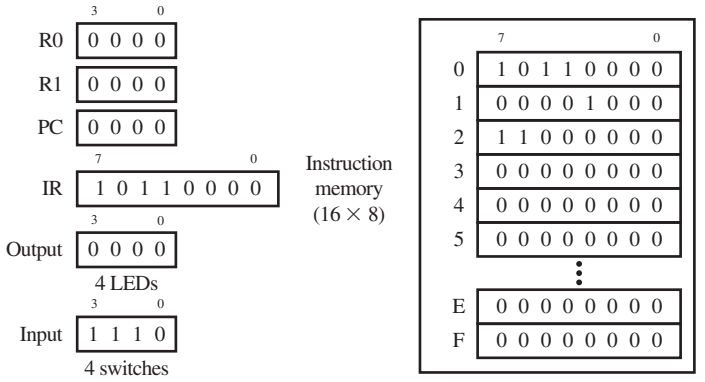


FIGURE 11.5 Result in the PRM after loading the machine code in Program 11.5 into VBC1 at instruction memory address 0 with the input switches set to 14

When the instruction IN R1 is executed, its transfer function performs the operation R1 ← DI(3:0), placing the value 14 into R1. When the instruction MOV R0,R1 is executed, its transfer function performs the operation R0 ← R1, placing the value of 14 into R0. When the instruction OUT R0 is executed, its transfer function performs the operation OP ← R0.

In Figure 11.6 the value at the Input (4 switches) or 1110(14) is transferred to register R1 and then transferred to register R0 and finally transferred to the Output (4 LEDs), as shown in the programmer's register model after executing the simple program.

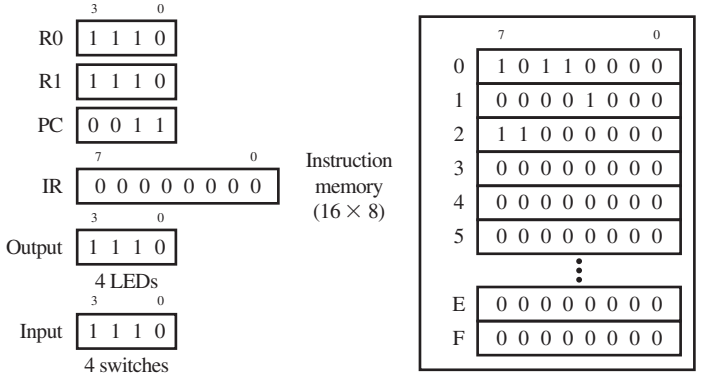


FIGURE 11.6 Result in the PRM after executing Program 11.5 for VBC1

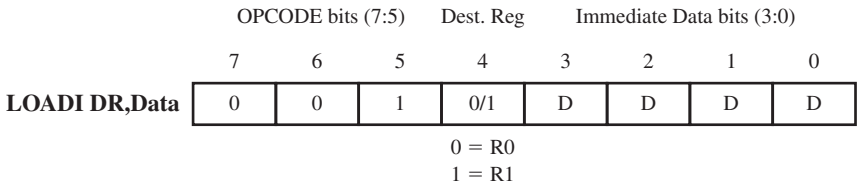
Observe that the MOV instruction can also be used to move the contents of R0 into R0, R0 into R1, and R1 into R1. MOV R0,R0 and MOV R1,R1 are instructions that appear to cause no change in the source register and the destination register. These instructions can be used as **NOP (no operation)** instructions. A NOP instruction is an instruction that causes no change to occur

and therefore can be used to simply waste time in VBC1. If you are writing a program and want the program to wait a period of time before doing a command, you can insert MOV R0,R0 or MOV R1,R1 into the program to create a delay. EASY1 supports NOP as an alias for MOV R0,R0.

11.6 THE LOADI INSTRUCTION

Form 11.4 shows the assembly language form and the machine code form for the LOADI (**load immediate**) instruction.

FORM 11.4 The LOADI instruction



The LOADI instruction provides a means to transfer immediate data, or data that is supplied as part of the instruction in bits(3:0), into the destination register. The data can be any value 0 through 15. The destination register can be either R0 or R1. The transfer function form for LOADI DR,Data is DR ← IR(3:0).

The actual assembly language instruction LOADI R0,5 is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 11.6.

Mnemonic field	Operands field
LOADI	R0,5

PROGRAM 11.6 Assembly language program for the LOADI R0,5 instruction

The OPCODE bits for the LOADI instruction are 001. In the machine code, the destination register is bit 4. When bit 4 is 0, the destination register is R0, and when bit 4 is 1, the destination register is R1. For the instruction LOADI R0,5, the data are provided in decimal.

Assembly 11.4 shows all versions of the LOADI R0,Data instruction listed in assembly language and machine code for VBC1, where Data is the decimal value of the data.

Assembly language	Machine code
LOADI R0,0	0 0 1 0 0 0 0 0
LOADI R0,1	0 0 1 0 0 0 0 1
LOADI R0,2	0 0 1 0 0 0 1 0
LOADI R0,3	0 0 1 0 0 0 1 1
LOADI R0,4	0 0 1 0 0 1 0 0
LOADI R0,5	0 0 1 0 0 1 0 1
LOADI R0,6	0 0 1 0 0 1 1 0
LOADI R0,7	0 0 1 0 0 1 1 1
LOADI R0,8	0 0 1 0 1 0 0 0
LOADI R0,9	0 0 1 0 1 0 0 1
LOADI R0,10	0 0 1 0 1 0 1 0
LOADI R0,11	0 0 1 0 1 0 1 1
LOADI R0,12	0 0 1 0 1 1 0 0
LOADI R0,13	0 0 1 0 1 1 0 1
LOADI R0,14	0 0 1 0 1 1 1 0
LOADI R0,15	0 0 1 0 1 1 1 1

ASSEMBLY 11.4 All versions of the LOADI R0,Data instruction

The machine code shows the bit patterns for the LOADI instructions as they must be placed in instruction memory to be executed. The machine code has the same OPCODE of 001 for bits 7:5 for all versions of the LOADI instruction.

Program 11.7 shows a simple assembly language program using the instructions LOADI, NOP, and OUT.

```
LOADI R1,9
NOP           ;MOV R0,R0 is used as the NOP instruction
OUT R1
```

PROGRAM 11.7 A simple assembly language program using the instructions LOADI, NOP, and OUT

In Figure 11.7, the machine codes for the instructions in Program 11.7 are manually loaded in the instruction memory beginning at address 0 in the programmer's register model. All instruction memory locations are cleared that are not loaded with a machine code instruction. The PC is cleared to address 0 (0000), and the IR shows the machine code value at address 0 (00111001). Registers R0, R1, and the Output in the programmer's register model are also cleared. The input switches are set to the value of 0000 or 0. The program is then single stepped three times to execute each of the three instructions.

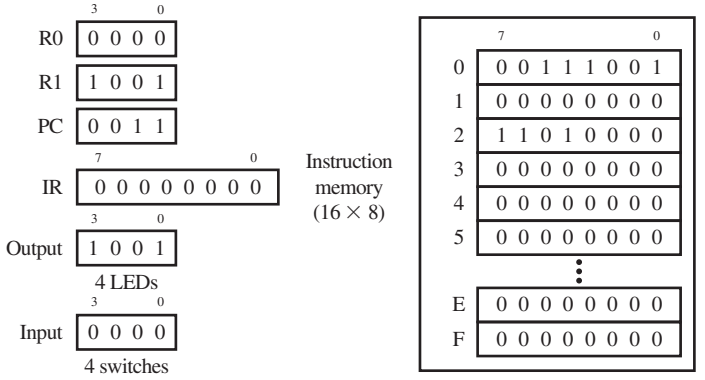
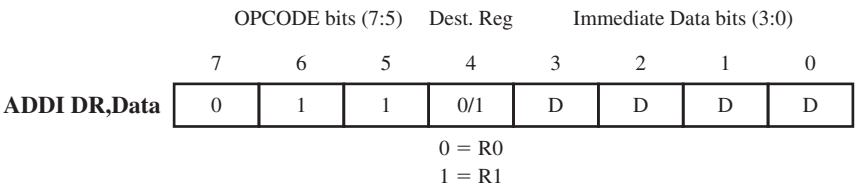


FIGURE 11.7 Result in the PRM after manually loading and then executing Program 11.7 for VBC1 with the input switches set to 0

Observe that 9 or binary 1001 is placed in R1 and then transferred to the output. The NO OPERATION instruction (NOP) is used to simply delay the time it takes until the binary value 1001 occurs on the Output while executing the program. After single stepping through each of the three instructions, the PC points to the address of the next instruction (0011), and the IR contains the instruction to be executed next (00000000).

11.7 THE ADDI INSTRUCTION

Form 11.5 shows the assembly language form and the machine code form for the ADDI (**add immediate**) instruction.



FORM 11.5 The ADDI instruction

The ADDI instruction adds the value in the destination register to the data in bits (3:0) in the instruction and places the result in the destination register. The destination register can be either R0 or R1, and data can be any value 0 through 15. The transfer function form for ADDI

DR,Data is DR ← DR + IR(3:0). To keep the hardware design for VBC1 as simple as possible, no carry-out bit is included for the ADDI instruction.

The actual assembly language instruction ADDI R1,7 is shown in two fields: the first field is the mnemonic field, and the second field is operands field as shown in Program 11.8.

Mnemonic field	Operands field
ADDI	R1,7

PROGRAM 11.8 Assembly language program for the ADDI R1,7 instruction

The OPCODE bits for the ADDI instruction are 011. In the machine code, the destination register is bit (4). When bit 4 is 0, the destination register is R0, and when bit 4 is 1, the destination register is R1.

Assembly 11.5 shows all versions of the ADDI R1,Data instruction listed in assembly language and machine code for VBC1, where Data is the decimal value of the data.

Assembly language	Machine code
ADDI R1,0	0 1 1 1 0 0 0 0
ADDI R1,1	0 1 1 1 0 0 0 1
ADDI R1,2	0 1 1 1 0 0 1 0
ADDI R1,3	0 1 1 1 0 0 1 1
ADDI R1,4	0 1 1 1 0 1 0 0
ADDI R1,5	0 1 1 1 0 1 0 1
ADDI R1,6	0 1 1 1 0 1 1 0
ADDI R1,7	0 1 1 1 0 1 1 1
ADDI R1,8	0 1 1 1 1 0 0 0
ADDI R1,9	0 1 1 1 1 0 0 1
ADDI R1,10	0 1 1 1 1 0 1 0
ADDI R1,11	0 1 1 1 1 0 1 1
ADDI R1,12	0 1 1 1 1 1 0 0
ADDI R1,13	0 1 1 1 1 1 0 1
ADDI R1,14	0 1 1 1 1 1 1 0
ADDI R1,15	0 1 1 1 1 1 1 1

ASSEMBLY 11.5 All version of the ADDI R1,Data instruction

The machine code shows the bit patterns for the ADDI instructions as they must be placed in instruction memory to be executed. For clarity, EASY1 supports INC R0 as an alias for ADDI R0,1 and INC R1 as an alias for ADDI R1,1. The machine code has the same OPCODE of 011 for bits 7:5 for all versions of the ADDI instruction.

Program 11.9 shows a simple assembly language program using the instructions IN, ADDI, MOV, and OUT.

```
IN R0
ADDI R0,4
MOV R1,R0
OUT R1
```

PROGRAM 11.9 A simple assembly language program using the instructions IN, ADDI, MOV, and OUT

In Figure 11.8, the machine codes for the instructions in Program 11.9 are manually loaded in the instruction memory beginning at address 0 in the programmer's register model. All instruction memory locations are cleared that are not loaded with a machine code instruction. The PC is cleared to address 0 (0000), and the IR shows the machine code value at address

0 (10100000). Registers R0, R1, and the Output in the programmer's register model are also cleared. The input switches are set to the value of 0011 or 3. The program is then single stepped four times to execute each of the four instructions.

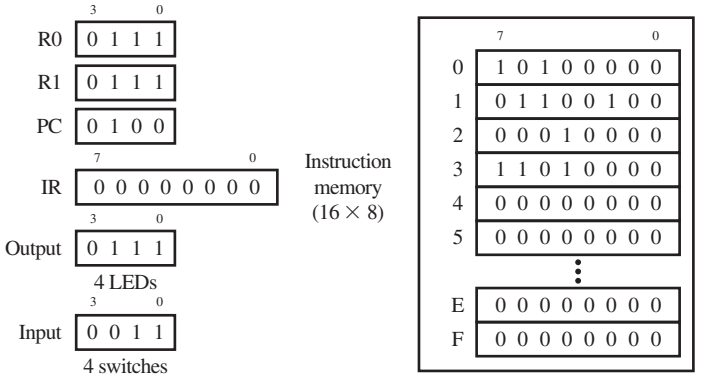
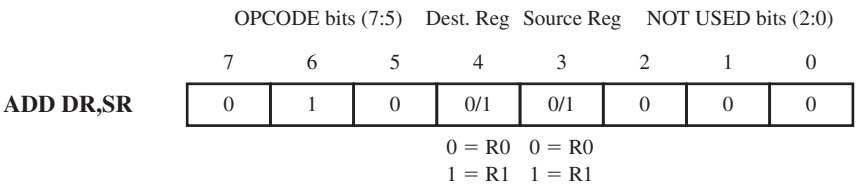


FIGURE 11.8 Result in the PRM after manually loading and then executing Program 11.9 for VBC1 with the input switches set to 3

After the instruction IN R0 is executed with 3 or 0011 at the INPUT, R0 first receives the value 3. After the instruction ADDI R0, 4 is executed, R0 receives the value 3 + 4 or 7. After the instruction MOV R1,R0 is executed, R1 receives the value 7. After the instruction OUT R1 is executed, the Output receives the value 7. After single stepping through each of the four instructions, the PC points to the address of the next instruction (0100), and the IR contains the instruction to be executed next (00000000).

11.8 THE ADD INSTRUCTION

Form 11.6 shows the assembly language form and the machine code form for the ADD instruction.



FORM 11.6 The ADD instruction

The ADD instruction adds the value in the source register to the value in the destination register and places the result in the destination register. The source register can be either R0 or R1, and the destination register can also be either R0 or R1. The transfer function form for ADD DR,SR is DR ← SR + DR. To keep the hardware design for VBC1 as simple as possible, no carry-out bit is included for the ADD instruction.

The actual assembly language instruction ADD R1,R0 is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 11.10.

Mnemonic field	**Operands field**
ADD	R1,R0

PROGRAM 11.10 Assembly language program for the ADD R1,R0 instruction

The OPCODE bits for the ADD instruction are 010. In the machine code, the destination register is bit (4). When bit 4 is 0, the destination register is R0, and when bit 4 is 1, the destination register is R1. When bit 3 is 0, the source register is R0, and when bit 3 is 1, the source register is R1.

Assembly 11.6 shows all versions of the ADD instruction listed in assembly language and machine code for VBC1.

Assembly language	Machine code
ADD R0,R0	0 1 0 0 0 0 0 0
ADD R0,R1	0 1 0 0 1 0 0 0
ADD R1,R0	0 1 0 1 0 0 0 0
ADD R1,R1	0 1 0 1 1 0 0 0

ASSEMBLY 11.6 All versions of the ADD instruction

The machine code shows the bit patterns for the ADD instructions as they must be placed in instruction memory to be executed. The machine code has the same OPCODE of 010 for bits 7:5 for all versions of the ADD instruction.

Program 11.11 shows a simple assembly language program using the instructions IN and ADD.

```
IN R1
ADD R1,R1
```

PROGRAM 11.11 A simple assembly language program using the instructions IN, and ADD

In Figure 11.9, the machine codes for the instructions in Program 11.11 are manually loaded in the instruction memory beginning at address 0 in the programmer's register model. All instruction memory locations are cleared that are not loaded with a machine code instruction. The PC is cleared to address 0 (0000), and the IR shows the machine code value at address 0 (10110000). Registers R0, R1, and the Output in the programmer's register model are also cleared. The input switches are set to the value of 0010 or 2. The program is then single stepped two times to execute each of the two instructions.

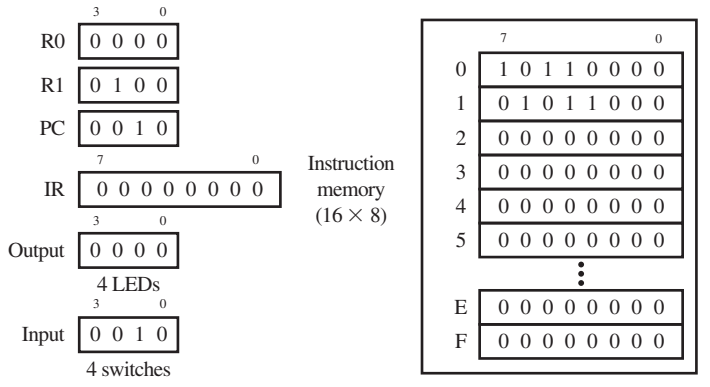
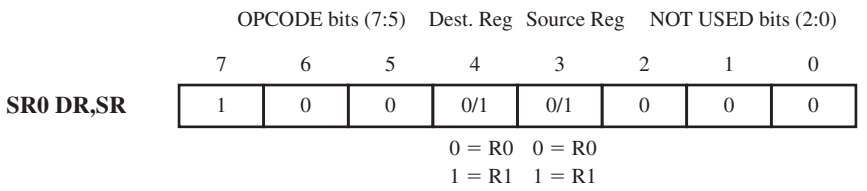


FIGURE 11.9 Result in the PRM after manually loading and then executing Program 11.11 for VBC1 with the input switches set to 2

After the instruction IN R1 is executed with 2 or 0010 at the Input, R1 receives the value 2. After the instruction ADD R1,R1 is executed, R1 receives the value 2 + 2 or 4. After single stepping through each of the two instructions, the PC points to the address of the next instruction (0010). The IR contains the instruction to be executed next (00000000).

11.9 THE SR0 INSTRUCTION

Form 11.7 shows the assembly language form and the machine code form for the SR0 (**shift right with 0 fill**) instruction. The instruction SR0 may also be referred to as a logical right shift with a 0 fill.



FORM 11.7 The SR0 instruction

The SR0 instruction takes the value in the source register and shifts it to the right by one bit position, then fills the most significant bit position with 0, and finally places the result in the destination register. The source register and the destination register can be either R0 or R1. The transfer function form for SR0 DR,SR is DR ← 0 SR(3:1).

The actual assembly language instruction SR0 R0,R1 is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 11.12.

Mnemonic field	Operands field
SR0	R0,R1

PROGRAM 11.12 Assembly language program for the SR0 R0,R1 instruction

The OPCODE bits for the SR0 instruction are 100. In the machine code, the destination register is bit 4. When bit 4 is 0, the destination register is R0, and when bit 4 is 1, the destination register is R1. When bit 3 is 0, the source register is R0, and when bit 3 is 1, the source register is R1.

Assembly 11.7 shows all versions of the SR0 instruction listed in assembly language and machine code for VBC1.

Assembly language		Machine code
SR0	R0,R0	1 0 0 0 0 0 0 0
SR0	R0,R1	1 0 0 0 1 0 0 0
SR0	R1,R0	1 0 0 1 0 0 0 0
SR0	R1,R1	1 0 0 1 1 0 0 0

ASSEMBLY 11.7 All versions of the SR0 instruction

The machine code shows the bit patterns for the SR0 instructions as they must be placed in instruction memory to be executed. The machine code has the same OPCODE of 100 for bits 7:5 for all versions of the SR0 instruction.

Program 11.13 shows a simple assembly language program using the instructions LOADI, SR0, and OUT.

```
LOADI R0,8
SR0 R1,R0
OUT R1
```

PROGRAM 11.13 A simple assembly language program using the instructions LOADI, SR0, and OUT

In Figure 11.10, the machine codes for the instructions in Program 11.13 are manually loaded in the instruction memory beginning at address 0 in the programmer's register model. All instruction memory locations are cleared that are not loaded with a machine code instruction. The PC is cleared to address 0 (0000), and the IR shows the machine code value at address 0 (00101000). Registers R0, R1, and the Output in the programmer's register model are also cleared. The input switches are set to the value of 0000 or 0. The program is then single stepped two times just to execute the first two instructions.

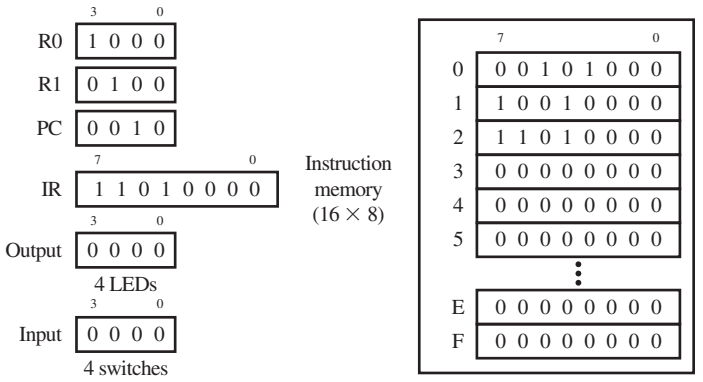


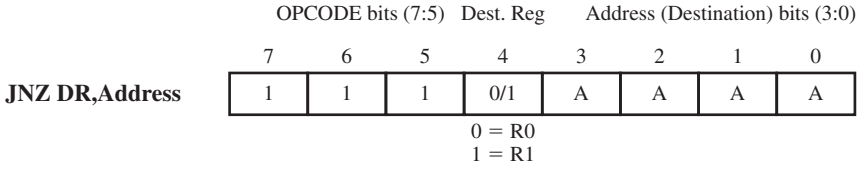
FIGURE 11.10 Result in the PRM after manually loading and then executing just the first two instructions in Program 11.13 for VBC1 with the input switches set to 0

After the instruction LOADI R0,8 is executed, R0 receives the value 8. After the instruction SR0 R1,R0 is executed, R1 receives the value 4. At this time, the PC points to the address of the next instruction (0010), and the IR contains the instruction to be executed next (11010000).

11.10 THE JNZ INSTRUCTION

Form 11.8 shows the assembly language form and the machine code form for the JNZ (**jump if not zero**) instruction.

FORM 11.8 The JNZ instruction



The JNZ instruction analyzes the value in the destination register bit 4 to determine if its value is not zero. If the value is not zero, the JNZ instruction loads the program counter with the value of the address in bits (3:0). This will cause the program counter to execute the instruction at the address specified in the bits (3:0)—that is, to jump to the address specified in the bits (3:0). If the value in the destination register is zero, then the JNZ instruction is simply skipped and the instruction following it is executed. When the JNZ instruction is skipped, this can be referred to as simply falling through the JNZ instruction. The transfer function form for JNZ DR,Address is PC ← IR(3:0), if DR ≠ 0 else PC ← PC + 1.

The actual assembly language instruction JNZ R1,9 is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 11.14.

Mnemonic field	**Operands field**
JNZ	R1,9

PROGRAM 11.14 Assembly language program for the JNZ R1,9 instruction

The OPCODE bits for the JNZ instruction are 111. In the machine code, the destination register is bit 4. When bit 4 is 0, the destination register is R0, and when bit 4 is 1, the destination register is R1. In the machine code, the address is in bits (3:0).

Assembly 11.8 shows all versions of the JNZ R1,Address instruction listed in assembly language and machine code for VBC1, where Address is the decimal value of the address.

Assembly language	Machine code
JNZ R1,0	1 1 1 1 0 0 0 0
JNZ R1,1	1 1 1 1 0 0 0 1
JNZ R1,2	1 1 1 1 0 0 1 0
JNZ R1,3	1 1 1 1 0 0 1 1
JNZ R1,4	1 1 1 1 0 1 0 0
JNZ R1,5	1 1 1 1 0 1 0 1
JNZ R1,6	1 1 1 1 0 1 1 0
JNZ R1,7	1 1 1 1 0 1 1 1
JNZ R1,8	1 1 1 1 1 0 0 0
JNZ R1,9	1 1 1 1 1 0 0 1
JNZ R1,10	1 1 1 1 1 0 1 0
JNZ R1,11	1 1 1 1 1 0 1 1
JNZ R1,12	1 1 1 1 1 1 0 0
JNZ R1,13	1 1 1 1 1 1 0 1
JNZ R1,14	1 1 1 1 1 1 1 0
JNZ R1,15	1 1 1 1 1 1 1 1

ASSEMBLY 11.8 All versions of the JNZ R1,Address instruction

The machine code shows the bit patterns for the JNZ instructions as they must be placed in instruction memory to be executed. The machine code has the same OPCODE of 111 for bits 7:5 for all versions of the JNZ instruction.

Program 11.15 shows a simple assembly language program using the instructions IN, JNZ, LOADI, and OUT.

```
start: IN R0
       JNZ R0,start
       LOADI R1,10
       OUT R1
```

PROGRAM 11.15 A simple assembly language program using the instructions JNZ, LOADI, and OUT

In Figure 11.11, the machine codes for the instructions in Program 11.15 are manually loaded in the instruction memory beginning at address 0 in the programmer's register model. All instruction memory locations are cleared that are not loaded with a machine code instruction. The PC is cleared to address 0 (0000), and the IR shows the machine code value at address 0 (10100000). Registers R0, R1, and the Output in the programmer's register model are also cleared. The input switches are set to the value of 0011 or 3. The program is then single stepped four times to execute the instructions.

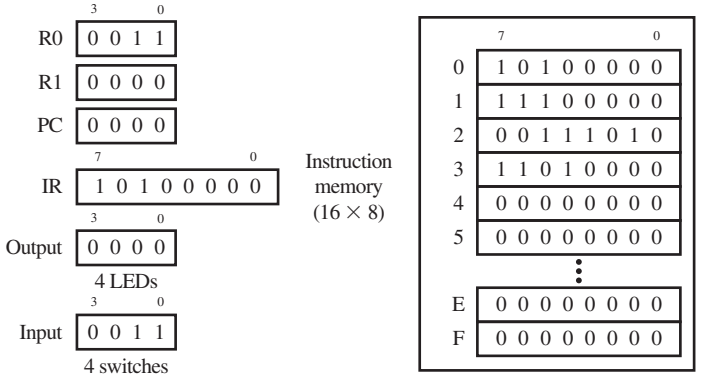


FIGURE 11.11 Result in the PRM after manually loading and then single stepping Program 11.15 four times with the input switches set to 3

After the instruction IN R0 is executed with 3 or 0011 at the Input, R0 receives the value 3. After the instruction JNZ R0,start is executed, the PC receives the value 0, which is the address of the first instruction at the label "start." Single stepping two more times causes the program to loop through the first two instructions, never getting to the third instruction. After single stepping four times, the R0 receives the value 3, the PC points to the address of the next instruction (0000), and the IR contains the instruction to be executed next (10100000).

Figure 11.12 shows the machine codes for the instructions in Program 11.15 loaded in the programmer's register model. The input switches are set to the value of 0, and the program is then single stepped four times to execute the instructions.

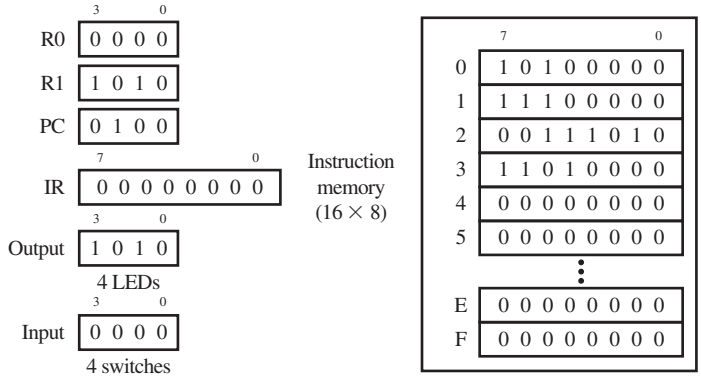


FIGURE 11.12 Result in the PRM after manually loading and then single stepping Program 11.15 four times with the input switches set to 0

After the instruction IN R0 is executed with 0 or 0000 at the Input, R0 receives the value 0. After the instruction JNZ R0,start is executed, the PC receives the value 2 or 0010, which is the address of the second instruction. After the instruction LOADI R1,10 is executed, R1 receives the value 10. After the instruction OUT R1 is executed, Output receives the value 10 or 1010 in binary, the PC points to the address of the next instruction (0100), and the IR contains the instruction to be executed next (00000000).

11.11 PROGRAMMING EXAMPLES AND TECHNIQUES FOR VBC1

We have provided a few program examples and techniques in this section to encourage you to think up programs and write them yourself. Your programs can be written, assembled, and simulated using EASY1 for VBC1 (see Appendix D).

11.11.1 Unconditional Jump

VBC1 does not have an unconditional jump instruction of the form JMP Address. A JMP Address instruction simply jumps unconditionally to the address specified in the instruction. Most computers have an unconditional jump instruction. The instructions LOADI R1,1 and JNZ R1,Address are the equivalent VBC1 instructions to use for the nonexisting unconditional jump instruction JMP Address. Placing the instruction LOADI R1,1 before the instruction JNZ R1,Address forces an unconditional jump. When writing an assembly language program for VBC1, an unconditional jump can be used to confine your program so that it only runs the instructions that you write and does not roam through instruction memory.

11.11.2 Labels

Names in the first field of an assembly language program followed by a colon ":" are referred to as labels. A label is simply a name that can be used to represent an address. The first line of code

in an assembly language program is at address 0, the second line of code is at address 1, and so on. If the address is 0, then the machine code is placed at address 0 in the instruction memory; if the address is 1, then the machine code is placed at address 1 in the instruction memory. If you use the JNZ instruction to jump to a particular line of code in a program, it is considered a good programming practice to use a label rather than to manually count the lines of code to determine an address. The instruction JNZ R0,0 will jump to address 0 if R0 is not zero. If the instruction at address 0 has the label "start," then the instruction JNZ R0,start will do the same operation. If "loop" is at address 7, then the instruction JNZ R1,loop will jump to address 7 if R1 is not zero. Using labels in programs takes less thought in writing programs because the programmer does not have to determine and list the numerical addresses for the JNZ instructions. Programs can be written to jump forward or to jump backward. In either case, a label should be used when writing programs for VBC1 to jump forward or backward.

11.11.3 Loop Counter

Very small programs can be written to perform the same instructions over and over, but this **inline programming technique** is considered inefficient for large programs. Because VBC1 only has 16 instruction memory locations, you can run out of memory in some cases if you do not use a loop counter.

Program 11.16 shows an inline program for a 3-bit MOD 6 (six states, 0 to 5) binary up counter. This inline program is quite inefficient because it uses more instruction memory than necessary.

```
; 3-bit MOD 6 binary up counter with inline program
start: LOADI R0,0     ; initialize output value
       OUT R0         ; output R0 to LEDs or 0
       ADDI R0,1      ; increment R0
       OUT R0         ; output R0 to LEDs or 1
       ADDI R0,1      ; increment R0
       OUT R0         ; output R0 to LEDs or 2
       ADDI R0,1      ; increment R0
       OUT R0         ; output R0 to LEDs or 3
       ADDI R0,1      ; increment R0
       OUT R0         ; output R0 to LEDs or 4
       ADDI R0,1      ; increment R0
       OUT R0         ; output R0 to LEDs or 5
       LOADI R0,1     ; force unconditional jump
       JNZ R0,start   ; jump back to start
```

PROGRAM 11.16 Inline program for a 3-bit MOD 6 (six states, 0 to 5) binary up counter

Writing a program with an internal loop(s) is considered a good programming practice rather than writing an inline program that performs the same instructions over and over. One technique is simply to dedicate a register as a loop counter. A simple routine is written to increment or decrement the register's value and test for loop completion with the JNZ instruction. If the loop is not completed, the loop repeats until the loop counter arrives at 0, at which time the JNZ instruction leaves the loop (because the loop is completed), and the instruction immediately following the JNZ instruction is executed.

In some cases, a loop counter is not necessary, but the JNZ instruction for testing for internal loop completion is still required.

Program 11.17 shows an equivalent program for the 3-bit MOD 6 (six states, 0 to 5) binary up counter using an internal loop with a loop counter. This program is much more efficient because it uses less instruction memory (i.e., fewer instructions).

```
; 3-bit MOD 6 binary up counter
; with a loop counter
start: LOADI R0,0     ; initialize output value
       LOADI R1,10    ; initialize loop counter
                      ; 10 = 16 − (# of loops) = 16 − 6
loop:  OUT R0         ; output R0 to LEDs
       INC R0         ; increment output value
       INC R1         ; increment loop counter
       JNZ R1,loop    ; test for loop completion
       LOADI R0,1     ; force unconditional jump
       JNZ R0,start   ; jump back to start
```

PROGRAM 11.17 Loop counter program for a 3-bit MOD 6 (six states, 0 to 5) binary up counter

11.11.4 Program Runs Amuck

When a program runs amuck, we mean a program runs in an area of memory where instructions are not specified. If a program is not designed well, it can **roam through memory**, where instructions are not present. This is bad programming practice because a program that roams through memory may fail its intended purpose. Observe in Programs 11.16 and 11.17 that the programs do not run amuck or roam through memory. In each case, the LOADI R0,1 instruction is used to force an unconditional jump when the JNZ R0,start instruction is executed. In other words, the JNZ R0,start instruction can never fall through to the address immediately following it and roam through memory.

If VBC1 executes an instruction past the last instruction in memory, then the program runs amuck. Consider this example. If all the memory locations contain 0s past the last instruction in the program, and VBC1 executes these instructions, each memory location contains the instruction MOV R0, R0 (which represents all 0s). Each MOV R0,R0 would be executed repeatedly until the program rolls over to 0000 after address 1111. This doesn't cause a problem except for timing, because each instruction MOV R0, R0 acts as a NOP. The point we are making is that you should always strive to write well-written programs that do not run amuck or roam through memory past the last instruction.

11.11.5 Subtraction Instruction

VBC1 does not have a subtraction instruction or any of the logic instructions such as AND, OR, XOR, etc. A subtraction can be accomplished indirectly by using an ADDI instruction. This technique is called **indirect subtraction by addition**. Logic instructions will be added later when we introduce an extended version of VBC1 called VBC1-E in Chapter 18.

For VBC1, we can only work with 4-bit numbers because the data path only has 4 bits. Consider the following relationship for decrementing a 4-bit number in register R0, which is the same as subtracting 1 from a 4-bit number in register R0:

$$R0 - 1 = R0 + (-1) = R0 + (\text{2's complement of } 0001) = R0 + ((\text{1's complement of } 0001) + 1)$$

Notice that -1 = (1's complement of 0001) + 1 = 1110 + 1 = 1111 or (all ones). This technique works for any word size—for example, for a 4-bit number, $-1 = 1111$, or for a 5-bit number, $-1 = 11111$, and so on.

Because $R0 - 1 = R0 + 1111$, the instruction ADDI R0,15 can be used to effectively perform the task of a nonexisting instruction SUBI R0,1.

To subtract 2 from a 4-bit number in register R0 do the following:

$R0 - 2 = R0 + (-2) = R0 +$ (1's complement of 0010) + 1

Because $-2 =$ (1's complement of 0010) $+ 1 = 1101 + 1 = 1110$ and $R0 - 2 = R0 + 1110$, the instruction ADDI R0,14 can be used to effectively perform the task of a nonexisting instruction SUBI R0,2.

Table 11.3 shows a nonexisting instruction SUBI R0,P, where P represents the number to be subtracted from register R0, and its equivalent instruction ADDI R0,Q, where Q represents the number to be added to register R0.

TABLE 11.3 Nonexisting instruction and equivalent VBC1 instruction for subtraction

Nonexisting instruction	Equivalent VBC1 instruction
SUBI R0,0	ADDI R0,0
SUBI R0,1	ADDI R0,15
SUBI R0,2	ADDI R0,14
SUBI R0,3	ADDI R0,13
SUBI R0,4	ADDI R0,12
SUBI R0,5	ADDI R0,11
SUBI R0,6	ADDI R0,10
SUBI R0,7	ADDI R0,9
SUBI R0,8	ADDI R0,8
SUBI R0,9	ADDI R0,7
SUBI R0,10	ADDI R0,6
SUBI R0,11	ADDI R0,5
SUBI R0,12	ADDI R0,4
SUBI R0,13	ADDI R0,3
SUBI R0,14	ADDI R0,2
SUBI R0,15	ADDI R0,1

To obtain the correct difference, observe that the value subtracted from R1 and the value added to R1 must sum to 16 in decimal, or 1(subtracted) + 15(added) = 16.

EASY1 supports all the nonexisting SUBI instructions in Table 11.3 as aliases in addition to the corresponding R1 versions. For clarity, EASY1 supports DEC R0 as an alias for ADDI R0,15 and DEC R1 as an alias for ADDI R1,15.

Program 11.18 shows the 3-bit MOD 6 binary up counter with a loop counter converted to a 3-bit MOD 6 (six states, 5 down to 0) binary down counter. The loop counter is also decremented rather than incremented.

```
; 3-bit MOD 6 binary down counter
; with a loop counter
start: LOADI R0,5    ; initialize output value
       LOADI R1,6    ; initialize loop counter
loop:  OUT R0        ; output value to LEDs
       DEC R0        ; decrement output value
       DEC R1        ; decrement loop counter
       JNZ R1,loop   ; test for loop completion
       LOADI R0,1    ; force unconditional jump
       JNZ R0,start  ; repeat sequence
```

PROGRAM 11.18 Loop counter program for a 3-bit MOD 6 (six states, 5 down to 0) binary down counter

11.11.6 Multiply Instruction

VBC1 does not have a multiply instruction. Multiplication by 2 can be accomplished by using the ADD R0,R0 instruction to add a number to itself, because $0 + 0 = 0 \times 2 = 0$, $1 + 1 = 1 \times 2 = 2$, $2 + 2 = 2 \times 2 = 4$, $3 + 3 = 3 \times 2 = 6$, $4 + 4 = 4 \times 2 = 8$, and so on. The nonexisting instruction MUL R0,2, where R0 contains the contents of the number to be multiplied by 2, is the same as ADD R0,R0. For the instruction ADD R0,R0 to work properly, the value of R0 must be in the range of 0 through 7. Example: MUL R0,2 when R0 is 6 is the same as ADD R0, R0, and the result is 12. If R0 is 8, then the result for ADD R0,R0 (or 8 multiplied by 2) is 16 and cannot be represented in just 4 bits, so it results in an overflow condition. Larger numbers for R0 also provide an overflow condition.

11.11.7 Divide Instruction

VBC1 does not have a divide instruction. Division by 2 can be accomplished by using the SR0 R0,R0 instruction to shift an even number to the right with 0 fill. Because, 1110 (14) shifted right with 0 fill = 0111 (7), 1100 (12) shifted right with 0 fill = 0110 (6), 1010 (10) shifted right with 0 fill = 0101(5), and so on. The nonexisting instruction DIV R0,2, where R0 contains the contents of an even number to be divided by 2, is the same as SR0 R0,R0. For the instruction SR0 R0,R0 to work properly, the value of R0 must be an even number in the range of 14 down to 2. Example: DIV R0,2 when R0 is 6 is the same as SR0 R0,R0, and the result is 3.

As you can see from the examples presented in this section, there are many different ways to write assembly language programs. One programmer may prefer one style over another. Documenting your program so that others may understand your style is considered a necessity for writing good programs.

PROBLEMS

Section 11.3 The IN Instruction

11.1 After the assembly of the assembly language, where is the machine code placed for execution?

11.2 To do hand assembly of an instruction, you must know the organization of the machine code form, have a good understanding of the instruction, and know the OPCODE for the instruction. Make a list of the OPCODEs for the eight different instructions for VBC1.

11.3 As a designer, can you choose a set of assignments for the OPCODE bits that is different than the set used in this book?

11.4 Which bits are dedicated for the OPCODE bits in the machine code form for VBC1?

11.5 Which bit is dedicated for the destination register in the machine code form for VBC1?

11.6 What are unused bits assigned in the machine code form for VBC1?

11.7 Is the DR (destination register) or the SR (source register) used in the IN instruction?

11.8 How many port addresses are possible for the IN instruction?

11.9 Write the transfer function form and the machine code form for the IN R0 instruction. Use the signal DI(3:0) as the input data from four switches. Assume DI(3:0) = 0111, and list the value of register R0 after the instruction is executed.

Section 11.4 The OUT Instruction

11.10 Is the DR (destination register) or the SR (source register) used in the OUT instruction?

11.11 Write the transfer function form and the machine code form for the OUT R0 instruction. Use the signal OP(3:0) as the output to the four LEDs. Assume R0 = 1100, and list the value of register OP after the instruction is executed.

Section 11.5 The MOV Instruction

11.12 Write the transfer function form and the machine code form for the MOV R0,R1 instruction. Assume R1 = 0101, and list the value of register R0 after the instruction is executed.

Section 11.6 The LOADI Instruction

11.13 Which bits are dedicated for the immediate data in the machine code form for the instruction LOADI for VBC1?

11.14 Write the transfer function form and the machine code form for the LOADI R1,Data instruction. Assume Data = 0011, and list the value of register R1 after the instruction is executed.

Section 11.7 The ADDI Instruction

11.15 Which bits are dedicated for the immediate data in the machine code form for the instructions ADDI for VBC1?

11.16 Write the transfer function form and the machine code form for the ADDI R0,Data instruction. Assume R0 = 0011 and Data = 1001, and list the value of register R0 after the instruction is executed.

Section 11.8 The ADD Instruction

11.17 Write the transfer function form and the machine code form for the ADD R0,R1 instruction. Assume R0 = 0011 and R1 = 0001, and list the value of register R0 after the instruction is executed.

Section 11.9 The SR0 Instruction

11.18 Write the transfer function form and the machine code form for the SR0 R1,R0 instruction. Assume R0 = 1011, and list the value of register R1 after the instruction is executed.

Section 11.10 The JNZ Instruction

11.19 Which bits are dedicated for the address in the machine code form for the instruction JNZ for VBC1?

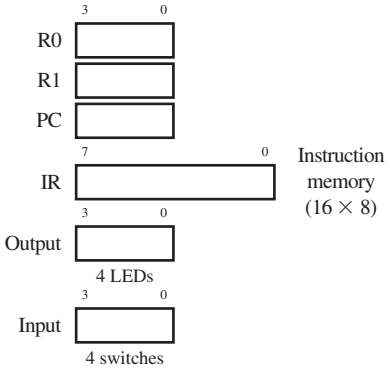
11.20 Write the transfer function form and the machine code form for the JNZ R0,Address instruction. Assume Address = 0011, R0 = 0010, and the current value of the PC = 0111. List the value of register PC after the instruction is executed.

Section 11.11 Programming Examples and Techniques for VBC1

11.21 Hand assemble the simple program in Program P11.21. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P11.21 after the last instruction is executed. Set Input (4 switches) = 0000 before executing the program.

```
LOADI R0,6
OUT R0
```

PROGRAM P11.21

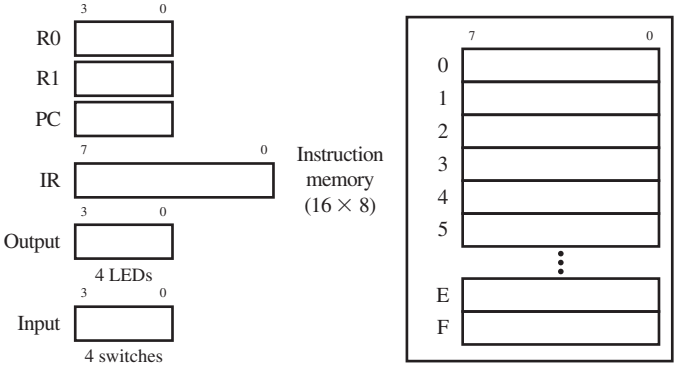


REGISTER_MODEL P11.21

11.22 Hand assemble the simple program in Program P11.22. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P11.22 after the last instruction is executed. Set Input (4 switches) = 0101 before executing the program.

```
IN R1
ADDI R1,5
OUT R1
```

PROGRAM P11.22

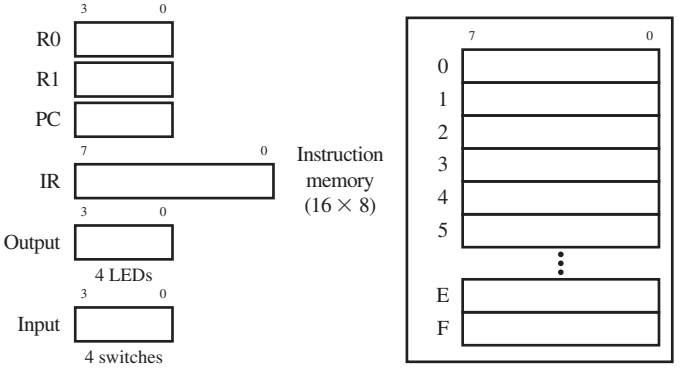


REGISTER_MODEL P11.22

11.23 Hand assemble the program in Program P11.23. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P11.23 after the last instruction is executed. Set Input (4 switches) = 0010 before executing the program.

```
      IN R0
back: SR0 R0,R0
      JNZ R0,back
      OUT R0
```

PROGRAM P11.23

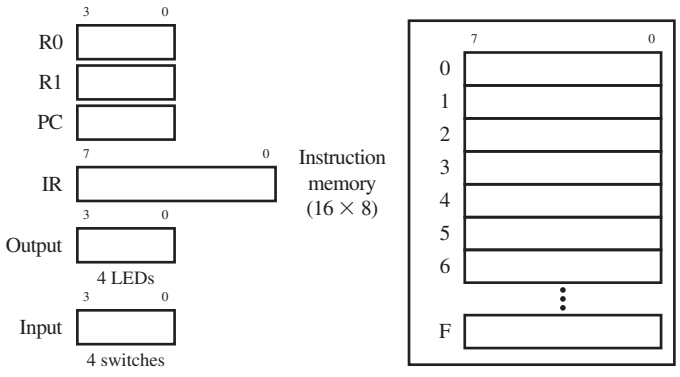


REGISTER_MODEL P11.23

11.24 Hand assemble the program in Program P11.24. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P11.24 after the last instruction is executed. Set Input (4 switches) = 0000 before executing the program.

```
      LOADI R0,8
loop: OUT R0
      SR0 R0,R0
      JNZ R0,loop
      LOADI R1,15
      OUT R1
```

PROGRAM P11.24



REGISTER_MODEL P11.24

11.25 Show the Output sequence as the program shown in Program P11.25 is single stepped.

```
      LOADI R0,8
loop: OUT R0
      SR0 R0,R0
      JNZ R0,loop
      LOADI R1,15
      OUT R1
```

PROGRAM P11.25

11.26 Show the Output sequence as the program shown in Program P11.26 is single stepped.

```
start: LOADI R1,4    ;initialize loop counter
       LOADI R0,3    ;initialize output value
loop:  OUT R0        ;output R0 to LEDs
       DEC R0        ;decrement R0
       DEC R1        ;decrement loop counter
       JNZ R1,loop   ;test for loop completion
       LOADI R1,1    ;force unconditional jump
       JNZ R1,start  ;jump back to start
```

PROGRAM P11.26

11.27 Write the instructions for VBC1 that provide the same result as the nonexisting unconditional jump instruction JMP Address. Discuss where the program jumps to, in terms of the machine code bits, when JMP Address is executed.

11.28 Which is considered a good programming practice: to write programs with labels or to write programs without labels for JNZ instructions?

11.29 In a program using a loop counter (a dedicated register) with the JNZ instruction, when does the JNZ instruction leave the loop to execute the instruction immediately following it?

11.30 What does it mean for a program to run amuck? Is it considered a good program practice for a program to run amuck?

11.31 VBC1 does not have a subtraction instruction. Write an assembly language instruction for VBC1 that will subtract 12 from the value in R1. Write the EASY1 alias for this instruction.

11.32 Show how to calculate the value needed to perform indirect subtraction by addition for decrementing the contents of register R1 by 1. What simple observation can be made concerning the value to be subtracted and the value to be added to R1 to obtain the correct difference? Write the assembly language instruction for VBC1 that will perform this decrement operation. Write the EASY1 alias for decrementing the contents of register R1 by 1.

11.33 Show how to calculate the value needed to perform indirect subtraction by addition for subtracting 5 from the contents of register R0. What simple observation can be made concerning the value to be subtracted and the value to be added to R0 to obtain the correct difference? Write the assembly language instruction for VBC1 that will perform this subtraction operation. Write the EASY1 alias for subtracting 5 from the content of register R0.

11.34 Write the instructions for VBC1 that provides the same result as the nonexisting multiply instruction MUL R1,2.

11.35 Write the instructions for VBC1 that provides the same result as the nonexisting divide instruction DIV R1,2.

12 Chapter

Designing Input/Output Circuits

Chapter Outline

12.1 INTRODUCTION

In this chapter, you will learn how to write VHDL to handle steering circuits, loadable register circuits, and input/output (I/O) circuits for VBC1. A documentation style is introduced that will enable you to expand your VHDL source code to include additional circuitry so that you, as well as others, can easily read and understand your code. Input and output circuits are combined to form a simple I/O system. Alternate VHDL design styles are presented to show different ways to write equivalent VHDL code.

12.2 DESIGNING STEERING CIRCUITS

A MUX (multiplexer) is a steering circuit. Another name for a MUX is a data selector because it selects or steers which input signal gets to the output of the MUX. Figure 12.1 shows a 2-to-1 MUX steering circuit.

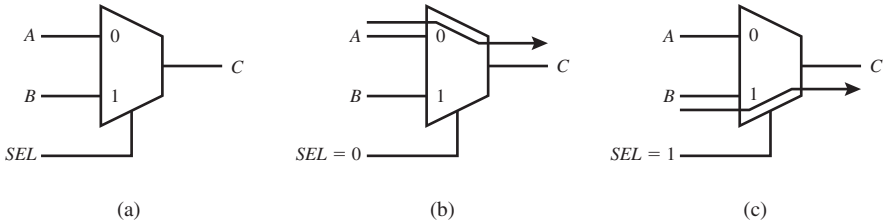


FIGURE 12.1 (a) Schematic for a 2-to-1 MUX steering circuit; (b) steering input *A* to output *C*; (c) steering input *B* to output *C*

The 2-to-1 MUX steering circuit in Figure 12.1 can be described by the wordy or verbose statement: input *A* is transferred to output *C* if *SEL* = 0 else input *B* is transferred to output *C*. A transfer function represents a concise mathematical description of a wordy or verbose statement for a circuit.

The 2-to-1 MUX steering circuit can be described by the following concise **transfer function:**

$C \leftarrow A$ *if* $SEL = 0$ *else*
$C \leftarrow B$

Things you should notice about the transfer function we used for the 2-to-1 MUX steering circuit:

- A transfer occurs based on the *SEL* (SELECT) input.
- The transfer operation $C \leftarrow A$ is only performed when *SEL* is 0.
- The transfer operation $C \leftarrow B$ is only performed when *SEL* is 1.
- This is a combinational logic circuit because there is no clock input.

The VHDL code for the 2-to-1 MUX steering circuit is shown in Listing 12.1. The design uses a dataflow architecture declaration with a **conditional signal assignment** (**CSA**).

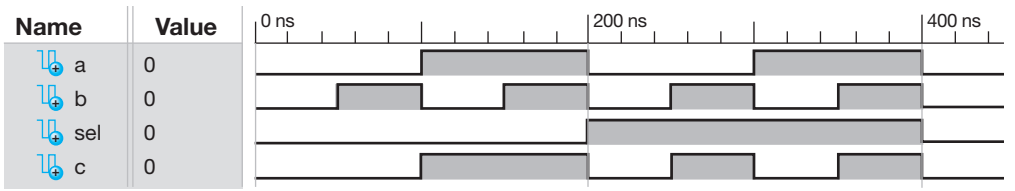
LISTING 12.1
Complete VHDL design entity for the 2-to-1 MUX steering circuit (project: scalar_steering)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity scalar_steering is port (
     a,b : in std_logic;
     sel: in std_logic;
     c : out std_logic
     );
end scalar_steering;

architecture dataflow of scalar_steering is
begin
     c <= a when sel = '0' else
          b;
end dataflow;
```

Things you should notice about the VHDL design in Listing 12.1:

- In the entity, each of the signals *A*, *B*, and *C* are declared as a scalar—that is, a signal for a single line. The mode for *A* and *B* is **in** and the mode for *C* is **out**.
- The resulting circuit consists of a 2-to-1 MUX controlled by the select input *SEL*.
- This is a combinational logic circuit because there is no clock input.

Waveform 12.1 shows the simulation with the correct functionality of design entity scalar_steering.



WAVEFORM 12.1
Simulation with the correct functionality of design entity scalar_steering

Observe that *C* follows *A* when *SEL* is 0, but it follows *B* when *SEL* is 1. Waveform 12.1 shows that the VHDL design in Listing 12.1 does in fact provide the correct design for the 2-to-1 MUX steering circuit.

12.3 DESIGNING BUS STEERING CIRCUITS

A **bus** is a **set or collection of wires** that carries multiple bits of information. A **4-bit bus steering circuit** is shown in Figure 12.2. This circuit is an array of 4 2-to-1 MUXs (multiplexers or data selectors). *BA* stands for BUS A, *BB* stands for BUS B, and *BC* stands for BUS C.

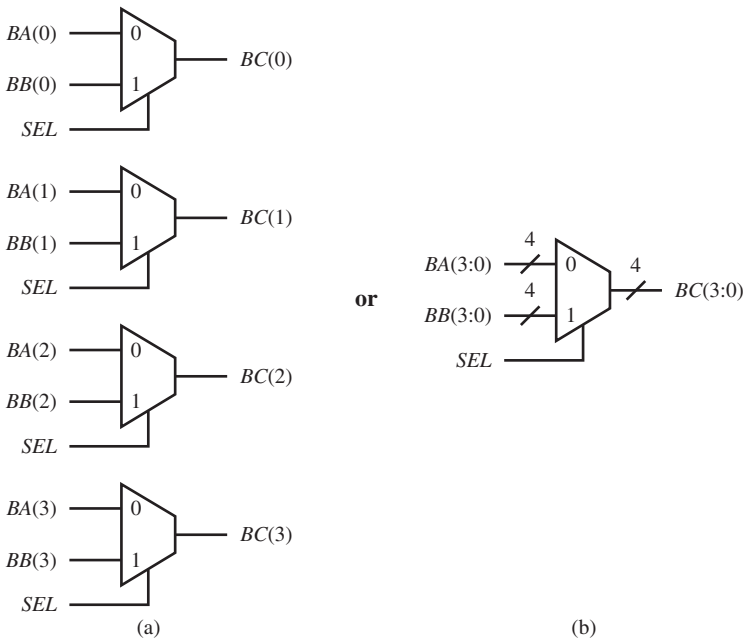


FIGURE 12.2 (a) Schematic for a 4-bit bus steering circuit; (b) simplified schematic for a 4-bit bus steering circuit

This circuit can be described by the following transfer function:

$BC \leftarrow BA$ *if* $SEL = 0$ *else*
$BC \leftarrow BB$

Things you should notice about the transfer function we used for the bus steering circuit:

- A transfer occurs based on the *SEL* (SELECT) input.
- The transfer operation $BC \leftarrow BA$ is only performed when *SEL* is 0.
- The transfer operation $BC \leftarrow BB$ is only performed when *SEL* is 1.
- This is a combinational logic circuit because there is no clock input.
- *BA*, *BB*, and *BC* all have the same bus size, and the bus size is 3:0 or 3 down to 0.

The VHDL code for the bus steering circuit in Figure 12.2 is shown in Listing 12.2. The design uses a dataflow architecture declaration with a conditional signal assignment (CSA).

LISTING 12.2 Complete VHDL design entity for a 4-bit bus steering circuit (project: bus_steering)

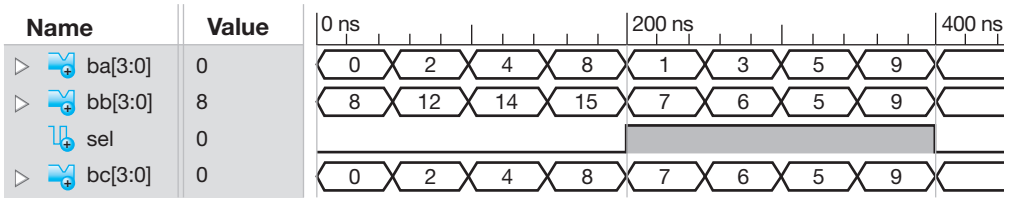
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bus_steering is port (
     ba,bb : in std_logic_vector (3 downto 0);
     sel: in std_logic;
     bc : out std_logic_vector (3 downto 0)
     );
end bus_ steering;
```

```
architecture dataflow of bus_ steering is
begin
     bc <= ba when sel = '0' else
           bb;
end dataflow;
```

Things you should notice about the VHDL design in Listing 12.2:

- In the entity, each of the signals *BA*, *BB*, and *BC* are declared as 4-bit buses—that is, vectors. The mode for *BA* and *BB* is **in** and the mode for *BC* is **out**.
- The resulting circuit consists of four 2-to-1 MUXs all controlled by the select input *SEL*.

Waveform 12.2 shows the simulation with the correct functionality of design entity bus_steering.



WAVEFORM 12.2 Simulation with the correct functionality of design entity bus_steering

Observe that *BC* follows *BA* when *SEL* is 0, but it follows *BB* when *SEL* is 1. Waveform 12.2 shows that the VHDL design in Listing 12.2 does in fact provide the correct design for the bus steering circuit.

12.4 DESIGNING LOADABLE REGISTER CIRCUITS

In the following discussion, a **register** is a **set of flip-flops** that stores information. A 4-bit **loadable register circuit** is shown in Figure 12.3.

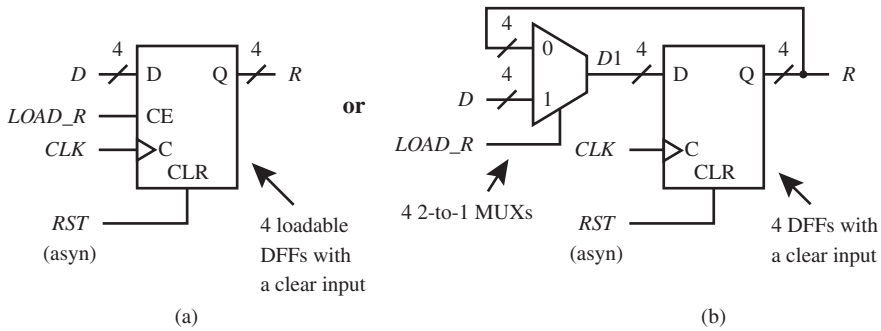


FIGURE 12.3 (a) Schematic for the 4-bit loadable register circuit; (b) equivalent circuit for the 4-bit loadable register circuit

Things you should notice about the 4-bit loadable register circuit shown in Figure 12.3b:

- Remember that the inputs D, CE (clock enable), C, and CLR, and the output Q inside the rectangle for the loadable register circuit are **labels**, not signals. When writing VHDL code, only **signals** are used, not labels. The signals in Figure 12.3b are *D*, *LOAD_R*, *CLK*, *RST*, and *R*.
- The MUX is a bus steering circuit.
- Output *R* of the D flip-flop is connected (fed back) to the 0 input of the MUX.
- Input *LOAD_R* of the MUX selects either the 0 input (when *LOAD_R* = 0) or the 1 input (when *LOAD_R* = 1).

- If the 0 input is selected, the D flip-flop stores the value *R*, because *D*1 = *R*, each time the clock ticks—that is, rising_edge (*CLK*).
- If the 1 input is selected, the D flip-flop stores the value *D*, because *D*1 = *D*, each time the clock ticks.
- *RST* overrides the control input C of the D flip-flop.
- This is a synchronous sequential logic circuit because there is a control input driven by a clock.

The loadable register circuit in Figure 12.3a or Figure 12.3b can be described by the following transfer function:

$R \leftarrow 0$ *if* $RST = 1$ *else*
$R \leftarrow D$ *if* $(LOAD\_R = 1$ *and* $\uparrow CLK)$ *else*
$R \leftarrow R$

The transfer function description represents the following wordy or verbose statement: zero is transferred to *R*(Register) if *RST* (RESET) = 1 else *D*(Data) is transferred to *R* if *LOAD_R* = 1 at the next rising edge of the clock else *R* retains its current value.

Things you should know about the transfer function we used for the 4-bit loadable register circuit:

- The directed arrow shows the direction of the transfer.
- Because *RST* precedes the clock, it is an asynchronous input and overrides the clock—that is, it is independent of the clock.
- The number of bits for *D* and *R* are currently specified as 4 bits. If desired, we can either specify the number of bits in parentheses as *D*(3:0) and *R*(3:0)—where 3:0 represents 4 bits with the range 3 downto 0—or simply say that *D* and *R* are both 4-bit buses.
- *D* is transferred to *R* only if *LOAD_R* = 1 at the next rising edge of the clock.
- *R* retains its current value if *RST* = 0, or *LOAD_R* = 0, or between clock ticks.

Listing 12.3 shows the VHDL code for the 4-bit loadable register circuit in Figure 12.3a. the design uses a dataflow architecture declaration with a conditional signal assignment (CSA).

LISTING 12.3
Complete VHDL design entity for the 4-bit loadable register circuit (project: load_reg)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity load_reg is port (
    rst, clk, load_r : in std_logic;
    d : in std_logic_vector (3 downto 0);
    r : inout std_logic_vector (3 downto 0)
    );
end load_reg;

architecture dataflow of load_reg is
begin
    r <= "0000" when rst = '1' else
         d   when load_r = '1' and rising_edge (clk);
end dataflow;
```

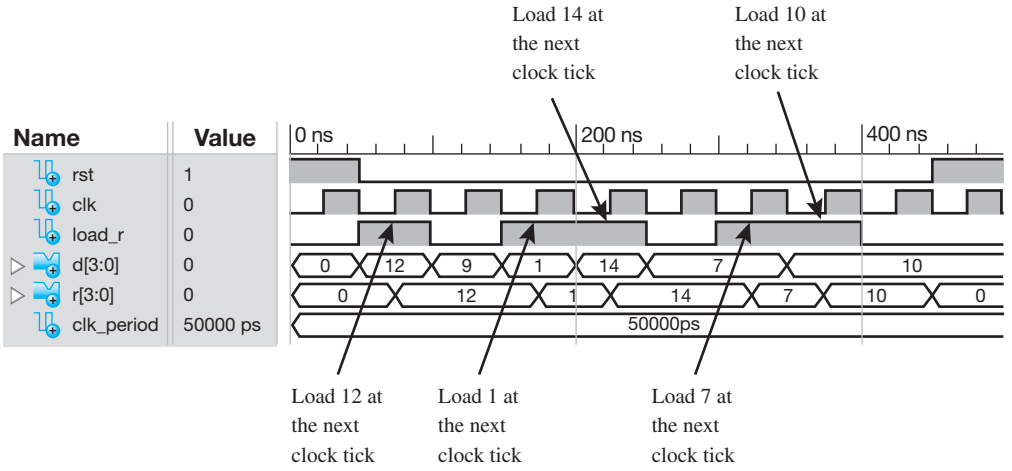
Things you should notice about the VHDL design in Listing 12.3:

- Both *D* and *R* are data type std_logic_vector (3 **downto** 0), which explicitly shows that these are 4-bit buses.
- Signal mode **inout** is required for signal *R* in the entity. This is because of the inferred conditional signal assignment *R* <= *R* in the architecture declaration.

- To transfer *D* to *R*, both *LOAD_R* and rising_edge (*CLK*) must be true.
- **Else** *R* is not included in the conditional signal assignment because it is inferred. When **else** *R* is left off of the conditional signal assignment, the present-state output value of *R* is simply retained or *R* <= *R*.

If an 8-bit bus were needed, this would only require us to change *D* and *R* in Listing 12.3 to data type std_logic_vector (7 downto 0). The vector range for an *n*-bit bus would be ((*n* -1) downto 0). All the buses and registers in the data path circuit for VBC1 are 4 bits. Any number of bits can be used to make larger bus sizes and a larger number of flip-flops. The number of flip-flops required for a register with *n* bits is *n* flip-flops. A 64-bit loadable register circuit would require 64 flip-flops.

Waveform 12.3 shows the correct functionality of design entity load_reg.



WAVEFORM 12.3
Simulation for the correct functionality of design entity load_reg

You should closely analyze the output waveform *R*(3:0) in Waveform 12.3 to verify that waveform *R*(3:0) accurately shows the correct functionality of design entity load_reg for all inputs. A clock tick is the next rising edge of signal *CLK*.

12.5 DESIGNING INPUT CIRCUITS

We will now draw a schematic to perform the following instruction using individual logic components: IN DR, where IN represents the instruction IN and DR represents the destination register (or where the input data will be placed). The input data are DI(3), DI(2), DI(1), and DI(0). Table 12.1 shows the IN instruction in both assembly language form and transfer function form for VBC1.

TABLE 12.1 The IN instruction in assembly language form and in transfer function form

Instruction in assembly language form	Instruction in transfer function form
IN DR	DR ← DI(3:0) or DR ← DI(3) DI(2) DI(1) DI(0)

Figure 12.4 is a schematic for an input circuit that performs the operation indicated by the IN DR instruction. The input data are driven or supplied by four slide switches.

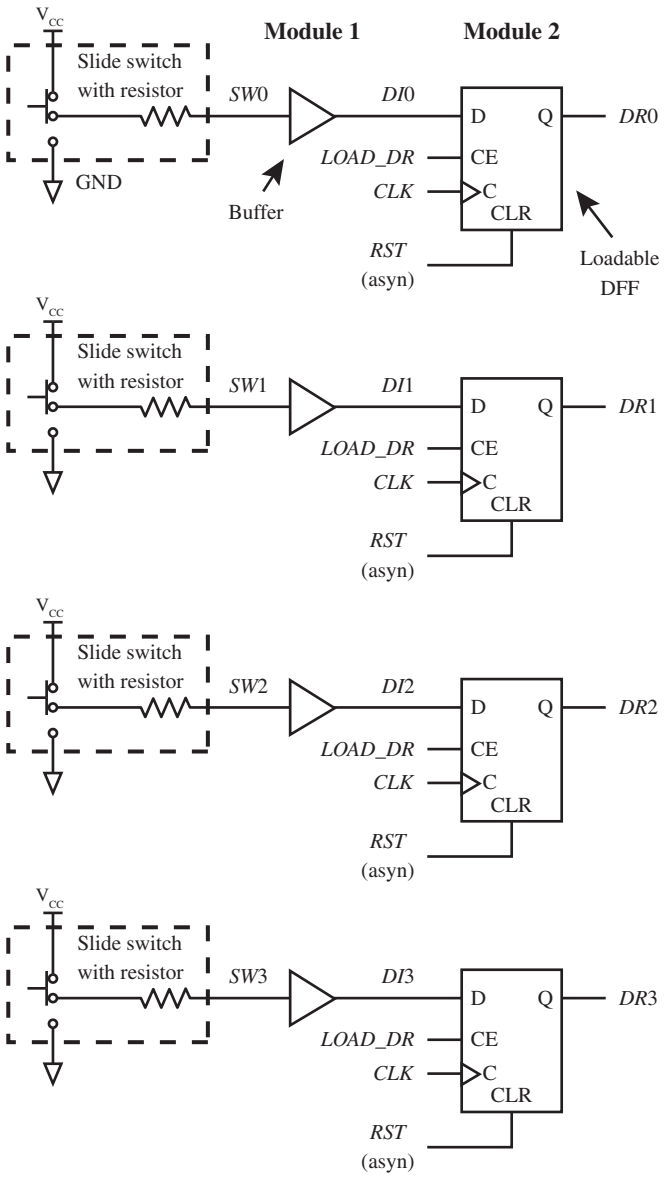


FIGURE 12.4 Schematic for an input circuit driven by four slide switches

Things you should notice about the schematic in Figure 12.4:

- There are two logic modules in the diagram—buffers (module 1) and loadable D flip-flops (module 2).
- The slide switches provide a high input voltage when the wiper is in the up position as shown in the diagram and a low input voltage when the wiper is in the down position.
- The input signals to the buffers are from the slide switches *SW*3, *SW*2, *SW*1, and *SW*0, and the output signals from the buffers are the data inputs *DI*3, *DI*2, *DI*1, and *DI*0. The buffers are used so that we can change the names of the signals at the inputs and the outputs for easier documentation. Buffers also provide additional current drive, or what I like to call "oomph," at their outputs.
- The loadable D flip-flop (FF) allows the output (Q) to be changed to the data input value (D) only when the FF's clock enable input (CE) is enabled by a high voltage. Output Q will change to the data input value D at the next rising edge of the clock, provided the FF's clear input (CLR) is not enabled by a high voltage. The clock enable input (CE) is clock depen-

dent and therefore a synchronous input. The inputs D, CE, C, CLR, and Q are **terminal labels**, not signals.

- When the FF's clear input (CLR) is enabled by a high voltage, the output (Q) is cleared to 0; that is, the FF's clear input (CLR) is clock independent, and therefore, an asynchronous input, because it overrides the control or clock input (C).
- The data input signals *DI*3, *DI*2, *DI*1, and *DI*0 are supplied to the D inputs of the loadable D flip-flops, which have the outputs *DR*3, *DR*2, *DR*1, and *DR*0.

12.5.1 Designing an Input Circuit Driven by Four Slide Switches

Figure 12.5 shows a simplified schematic for the input circuit driven by the four slide switches shown earlier in Figure 12.4.

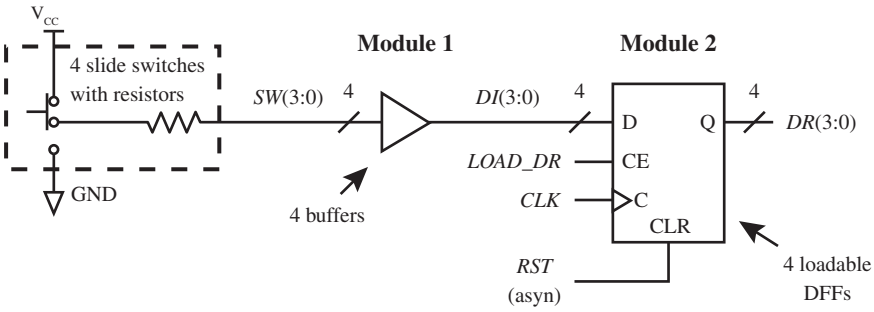


FIGURE 12.5 Simplified schematic for an input circuit driven by four slide switches

Listing 12.4 shows a design for the input circuit shown in Figure 12.5. The design uses a dataflow architecture declaration with a Boolean equation, and a conditional signal assignment (CSA).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity IN_Inst_4b is port (
     rst, clk, load_dr : in std_logic;
     sw : in std_logic_vector (3 downto 0);
     dr : inout std_logic_vector (3 downto 0)
     );
end IN_Inst_4b;

architecture dataflow of IN_Inst_4b is

--Modules 1 and 2 internal signal, di
     signal di: std_logic_vector (3 downto 0);
begin

--Module 1 code, buffer
     di <= sw;

--Module 2 code, loadable DFFs with a clear input
     dr <= "0000" when rst = '1' else
               di   when load_dr = '1' and rising_edge (clk);
end dataflow;
```

LISTING 12.4 Complete VHDL design entity for the schematic of the input circuit shown in Figure 12.5 (project: IN_Inst_4b)

Pay particular attention to the **documentation style** in Listing 12.4, because this will show you how to mark where each internal signal is placed and how to mark where each section of code is placed, for each circuit element (or module), in a large VHDL design. We will call this **documentation style M** (**DSM**), where the M reminds us to *mark* internal signals and code

locations for each module. For easy reference, notice in Listing 12.4 that the code modules are listed in numerical order so that they can be more easily found in a large design. If documentation style M is accurately used in a VHDL design, this allows anyone to easily draw a schematic for the design.

Things you should notice about the VHDL design in Listing 12.4:

- The signals *SW*, *DI*, and *DR* are data type std_logic_vector (3 **downto** 0), which explicitly shows that these are 4-bit buses.
- Signal mode **inout** is required for signal *DR* in the entity. This is because of the inferred conditional signal assignment *DR* <= *DR* in the architecture declaration.
- To transfer *DI* to *DR*, both *LOAD_DR* and rising_edge (*CLK*) must be true.
- **Else** *DR* is not included in the conditional signal assignment because it is inferred. When **else** *DR* is left off of the conditional signal assignment, the present-state output value *DR* is simply retained or *DR* <= *DR*.
- Note that DSM is used to show where the internal signal (signal *DI*) is placed and which modules it is between (modules 1 and 2 in this case). Documentation style M also shows where the code for each circuit element (modules 1 and 2 in this case) is placed and its name, which indicates what the code does. This documentation style thus allows you to easily locate each internal signal and each section of code, for each circuit element, in your VHDL design. For easier readability, provide a blank line between each internal signal section and each code section, as shown in Listing 12.4.
- Observe that this design uses a flat design approach because each of the modules in the system are included within a single architecture declaration.

12.6 DESIGNING OUTPUT CIRCUITS

We will now draw a schematic to perform the following instruction using individual logic components: OUT DR, where OUT represents the instruction OUT, which places the content of DR into an output port register OP. The content of DR is DR(3), DR(2), DR(1), and DR(0) and is provided by a destination register DR. The output port register signals are OP(3), OP(2), OP(1), and OP(0). Table 12.2 shows the OUT instruction in both assembly language form and transfer function form for VBC1.

TABLE 12.2 The OUT instruction in assembly language form and in transfer function form

Instruction in assembly language form	Instruction in transfer function form
OUT DR	OP ← DR or OP(3) OP(2) OP(1) OP(0) ← DR(3) DR(2) DR(1) DR(0)

Figure 12.6 is a schematic for an output circuit that performs the operation indicated by the OUT DR instruction. The output port signals are driving four LEDs.

Things you should notice about the schematic in Figure 12.6:

- There are two logic modules in the diagram—loadable D flip-flops (module 1) and buffers (module 2).
- The loadable D flip-flop's output signals *OP*(3:0) are supplied to the buffers, and the output signal from the buffers are the signals *LD*(3:0), which drive the LEDs. The buffers are used so that we can change the names of the signals at the inputs and the outputs for easier documentation. Remember that buffers also provide additional current drive, or oomph, at their outputs.

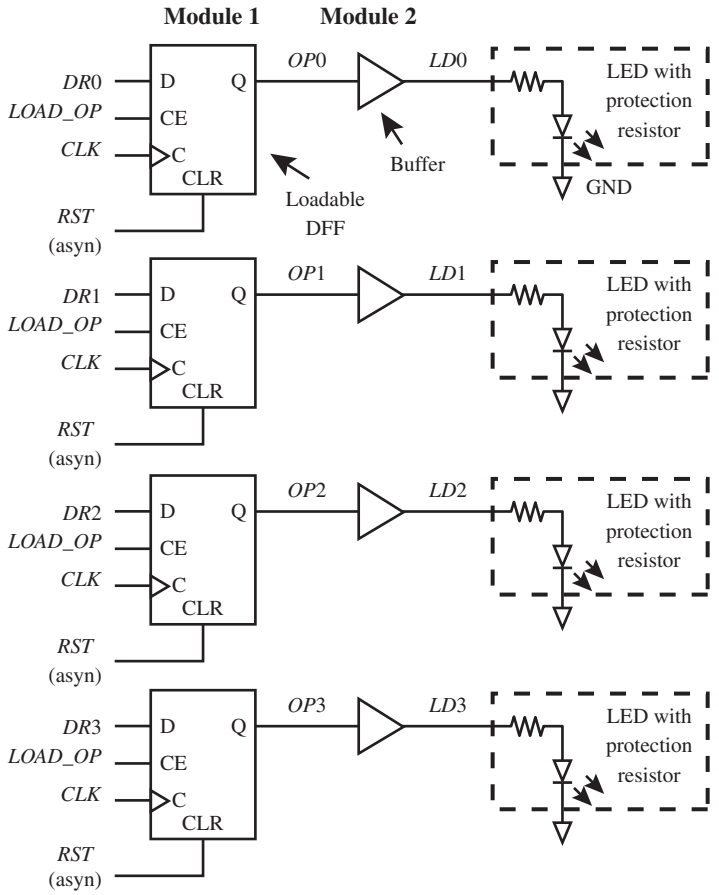


FIGURE 12.6 Schematic for an output circuit driving four LEDs

- The LEDs light, or turn on, when a high-input voltage signal is applied to them, and they turn off when a low-input voltage is applied to them.

12.6.1 Designing an Output Circuit to Drive Four LEDs

Figure 12.7 shows a simplified schematic for the OUT instruction shown earlier in Figure 12.6.

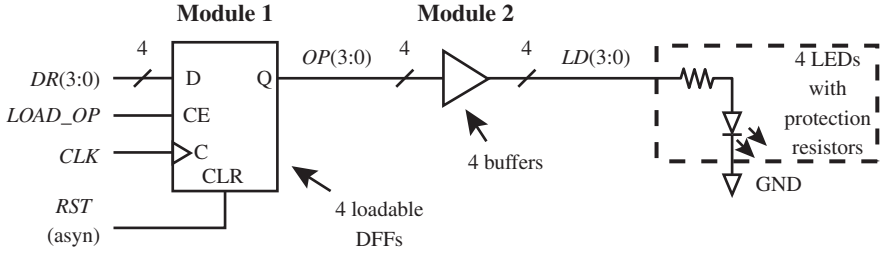


FIGURE 12.7 Simplified schematic for an output circuit driving four LEDs

Listing 12.5 shows a design for the output circuit shown in Figure 12.7. The design uses a dataflow architecture declaration with a conditional signal assignment (CSA) and a Boolean equation.

LISTING 12.5
Complete VHDL design entity for the schematic of the output circuit shown in Figure 12.7 (project: OUT_Inst_4b)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OUT_Inst_4b is port (
     rst, clk, load_op : in std_logic;
     dr : in std_logic_vector (3 downto 0);
     ld : out std_logic_vector (3 downto 0)
     );
end OUT_Inst_4b;

architecture dataflow of OUT_Inst_4b is

--Modules 1 and 2 internal signal, op
     signal op: std_logic_vector (3 downto 0);
begin

--Module 1 code, loadable DFFs with a clear input
     op <= "0000" when rst = '1' else
               dr   when load_op = '1' and rising_edge (clk);
--Module 2 code, buffer
     ld <= op;
end dataflow;
```

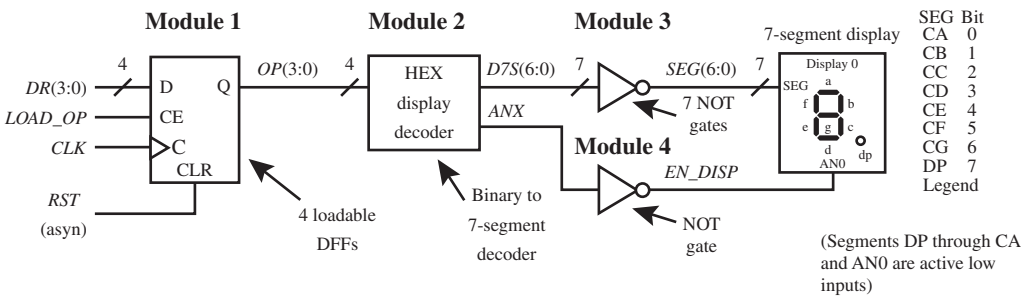
Things you should notice about the VHDL design in Listing 12.5:

- The signals *DR*, *OP*, and *LD* are data type std_logic_vector (3 **downto** 0), which explicitly shows that these are 4-bit buses.
- To transfer *DR* to *OP*, both *LOAD_OP* and rising_edge (*CLK*) must be true.
- **Else** *OP* is not included in the conditional signal assignment because it is inferred. When **else** *OP* is left off of the conditional signal assignment, the present-state output value *OP* is simply retained, or *OP* <= *OP*.
- DSM shows where the internal signal (signal *OP*) is placed and which modules it is between (modules 1 and 2 in this case). Documentation style M also shows where the code for each circuit element (modules 1 and 2 in this case) is placed and its name, which indicates what the code does.

12.6.2 Designing an Output Circuit to Drive a 7-Segment Display

Figure 12.8 shows a simplified schematic for an output circuit driving a 7-segment display via a HEX (hexadecimal) display decoder.

FIGURE 12.8
Simplified schematic for an output circuit driving a 7-segment display via a HEX (hexadecimal) display decoder



Things you should notice about the schematic in Figure 12.8:

- There are four logic modules in the diagram—four loadable D flip-flops (module 1); the HEX display decoder, or binary to 7-segment decoder (module 2); the seven NOT gates (module 3); and the single NOT gate (module 4).

- The loadable D flip-flop's output signals *OP*(3:0) are supplied to the HEX display decoder. The HEX display decoder output signals *D7S*(6:0) are supplied to the seven NOT gates. The HEX display decoder output signal *ANX* is supplied to the single NOT gate. The NOT gate outputs *SEG*(6:0) drive the segments of the 7-segment display. The single NOT gate output *EN_DISP* drives the enable input of the display (input AN0). The individual segments A through G light, or turn on, when a low voltage is applied to them only if the enable input of the display (AN0) has a low voltage applied to it.
- The NOT gates are used to allow the active high outputs from the HEX display decoder to drive the active low inputs of the individual segments A through G and the enable input AN0 of the display. The NOT gates allow us to work with active high signals when designing the output signals for the HEX display decoder.

Listing 12.6 shows a design for the output circuit shown in Figure 12.8. This design uses a dataflow architecture declaration with conditional signal assignments (CSAs) and Boolean equations.

LISTING 12.6
Complete VHDL design entity for the schematic of the output circuit shown in Figure 12.8 (project: OUT_Inst_4b_7s)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OUT_Inst_4b_7s is port (
     rst, clk, load_op : in std_logic;
     dr : in std_logic_vector (3 downto 0);
     seg : out std_logic_vector (6 downto 0);
     en_disp : out std_logic
     );
end OUT_Inst_4b_7s;

architecture dataflow of OUT_Inst_4b_7s is

--Modules 1 and 2 internal signal, op
     signal op: std_logic_vector (3 downto 0);

--Modules 2 and 3 internal signal, d7s
     signal d7s: std_logic_vector (6 downto 0);

--Modules 2 and 4 internal signal, anx
     signal anx: std_logic;
begin

--Module 1 code, loadable DFFs with a clear input
     op <= "0000" when rst = '1' else
               dr   when load_op = '1' and rising_edge (clk);

--Module 2 code, HEX display decoder
     d7s <= "0111111" when op = "0000" else
            "0000110" when op = "0001" else
            "1011011" when op = "0010" else
            "1001111" when op = "0011" else
            "1100110" when op = "0100" else
            "1101101" when op = "0101" else
            "1111101" when op = "0110" else
            "0000111" when op = "0111" else
            "1111111" when op = "1000" else
            "1101111" when op = "1001" else
            "1110111" when op = "1010" else
            "1111100" when op = "1011" else
                                                    (Continued)
```

```
            "0111001" when op = "1100" else
            "1011110" when op = "1101" else
            "1111001" when op = "1110" else
            "1110001";
    anx <= '1';

--Module 3 code, NOT gates
    seg <= not d7s;

--Module 4 code, NOT gate
    en_disp <= not anx;

end dataflow;
```

Things you should notice about the VHDL design in Listing 12.6:

- The signals *DR* and *OP* are data type std_logic_vector (3 **downto** 0), which explicitly shows that these are 4-bit buses.
- The signals *D7S* and *SEG* are data type std_logic_vector (6 **downto** 0), which explicitly shows that these are 7-bit buses.
- To transfer *DR* to *OP*, both *LOAD_OP* and rising_edge (*CLK*) must be true.
- **Else** *OP* is not included in the conditional signal assignment because it is inferred. When **else** *OP* is left off of the conditional signal assignment, the present state output value *OP* is simply retained.
- In module 2 (HEX display decoder), a 1 in each *D7S* assignment represents a segment that will be lighted or turned on while a 0 represents a segment that will be turned off. In the case of 0111111, segment G will be turned off, while segments A through F will be turned on. The value 0 will be displayed on the 7-segment display when the output *ANX* is 1. The Boolean equations *SEG* <= **not** *D7S* and *EN_DISP* <= **not** *ANX* convert the active high signals (*D7S* and *ANX*) to the active low signals (*SEG* and *EN_DISP*) required by the inputs to the 7-segment display.
- Documentation style M shows where the internal signals (signals *OP*, *D7S*, and *ANX* in this case) are placed and which modules they are between. Documentation style M also shows where the code for each circuit element (modules 1, 2, 3, and 4 in this case) is placed along with its name, which indicates what the code does. This documentation style allows you to easily locate all of the internal signals and code associated with each circuit element in your VHDL design.

12.6.3 A Closer Look at the Circuitry for Display 0

Figure 12.9 shows the 7-segment display in Figure 12.8 with the required circuitry to function properly. Display 0 is wired up as a **common anode** display because all the anodes are connected to a common point. The LED for the decimal point is also shown in Figure 12.9. The LED segments are protected by protection resistors to limit current flow through the segments. The anodes of the LEDs are driven by the N-doped semiconductor material between two layers of P-doped material (PNP) transistor, called a BJT (bipolar junction transistor). The anodes of the LEDs could also be driven by a P-doped semiconductor material channel (P-channel) transistor, called a FET (field effect transistor) as illustrated in the figure.

CA means cathode A, CB means cathode B, and so on. Applying a low voltage or logic 0 to input CA through CG will turn on the respective LED segments a through g provided that a low voltage or logic 0 is also applied to the base of the BJT via input AN0. The display will not turn on (none of the segments will turn on) when a high voltage or logic 1 is applied at input AN0. Inputs CA through CG and input AN0 are active low inputs.

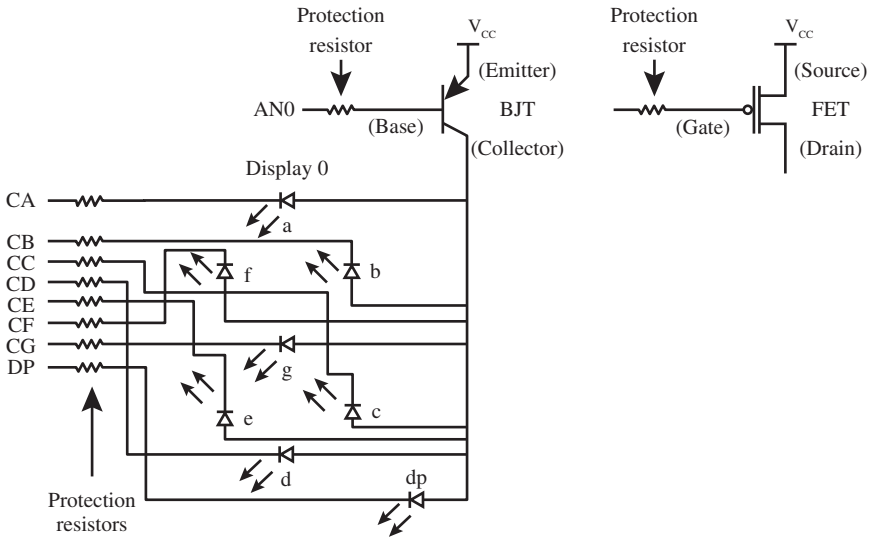


FIGURE 12.9 Common anode circuitry for display 0 in Figure 12.8.

12.7 COMBINING INPUT AND OUTPUT CIRCUITS TO FORM A SIMPLE I/O SYSTEM

Figure 12.10 shows a simple I/O system consisting of the input circuit in Figure 12.5 and the output circuits in Figures 12.7 and 12.8.

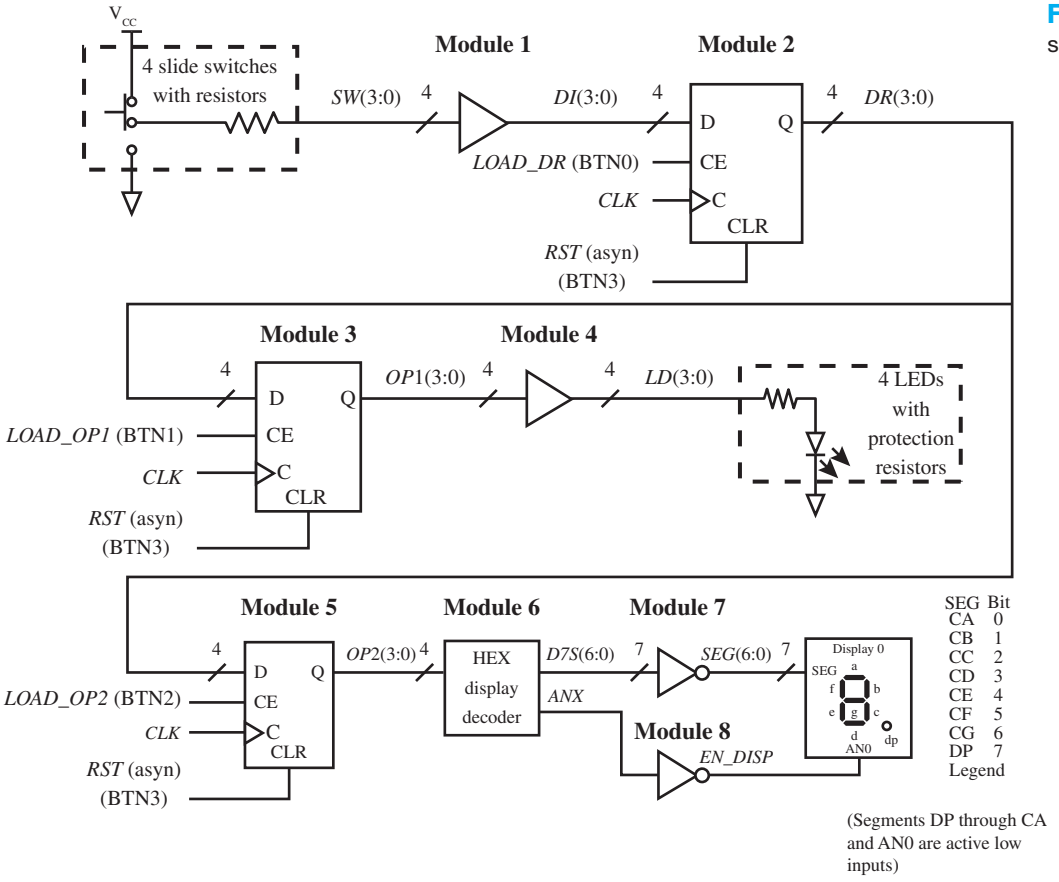


FIGURE 12.10 A simple I/O system

Things you should notice about the schematic in Figure 12.10:

- The logic modules are numbered 1 through 8.
- The external input and output signals—that is, the port signals for the entity in VHDL—are *SW*(3:0), *LOAD_DR*, *CLK*, *RST*, *LOAD_OP*1, *LD*(3:0), *LOAD_OP*2, *SEG*(6:0), and *EN_DISP*.
- The internal signals, for the architecture in VHDL, are *DI*(3:0), *DR*(3:0), *OP*1(3:0), *OP*2(3:0), *D7S*(6:0), and *ANX*.
- Modules 1 and 2 provide a data input port.
- Modules 3 and 4 provide data output port 1, which drives four LEDs.
- Modules 5, 6, 7, and 8 provide data output port 2, which drives a 7-segment display.
- BTN0 through BTN3 are provided to indicate recommended push buttons to use if you decide to download the VHDL design in Listing 12.7 on a BASYS 2 board or a NEXYS 2 board. If you download the design in Listing 12.7, you can verify that the VHDL code works in hardware.

Listing 12.7 shows a design for the simple I/O system shown in Figure 12.10. The VHDL code consists of the code for the input circuit in Listing 12.4, the code for the output circuit in Listing 12.5, and the code for the output circuit in Listing 12.6 documented using modules 1 through 8 in Figure 12.10.

LISTING 12.7
Complete VHDL design entity for the schematic of the simple I/O system shown in Figure 12.10 (project: Simple_IO_System)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Simple_IO_System is port (
     sw : in std_logic_vector (3 downto 0);
     load_dr, clk, rst, load_op1, load_op2 : in std_logic;
     ld : out std_logic_vector (3 downto 0);
     seg : out std_logic_vector (6 downto 0);
     en_disp : out std_logic
     );
end Simple_IO_System;

architecture dataflow of Simple_IO_System is

--Modules 1 and 2 internal signal, di
     signal di: std_logic_vector (3 downto 0);

--Modules 2 and 3, Modules 2 and 5 internal signal, dr
     signal dr: std_logic_vector (3 downto 0);

--Modules 3 and 4 internal signal, op1
     signal op1: std_logic_vector (3 downto 0);

--Modules 5 and 6 internal signal, op2
     signal op2: std_logic_vector (3 downto 0);

--Modules 6 and 7 internal signal, d7s
     signal d7s: std_logic_vector (6 downto 0);

--Modules 6 and 8 internal signal, anx
     signal anx: std_logic;
begin

--Module 1 code, buffer
     di <= sw;
```

```
--Module 2 code, loadable DFFs with a clear input
    dr <= "0000" when rst = '1' else
            di   when load_dr = '1' and rising_edge (clk);

--Module 3 code, loadable DFFs with a clear input
    op1 <= "0000" when rst = '1' else
             dr   when load_op1 ='1' and rising_edge (clk);

--Module 4 code, buffer
    ld <= op1;

--Module 5 code, loadable DFFs with a clear input
    op2 <= "0000" when rst = '1' else
             dr   when load_op2 = '1' and rising_edge (clk);

--Module 6 code, HEX display decoder
    d7s <= "0111111" when op2 = "0000" else
           "0000110" when op2 = "0001" else
           "1011011" when op2 = "0010" else
           "1001111" when op2 = "0011" else
           "1100110" when op2 = "0100" else
           "1101101" when op2 = "0101" else
           "1111101" when op2 = "0110" else
           "0000111" when op2 = "0111" else
           "1111111" when op2 = "1000" else
           "1101111" when op2 = "1001" else
           "1110111" when op2 = "1010" else
           "1111100" when op2 = "1011" else
           "0111001" when op2 = "1100" else
           "1011110" when op2 = "1101" else
           "1111001" when op2 = "1110" else
           "1110001";
    anx <= '1';

--Module 7 code, NOT gates
    seg <= not d7s;

--Module 8 code, NOT gate
    en_disp <= not anx;
end dataflow;
```

Things you should notice about the VHDL design in Listing 12.7:

- The entity specifies all the external input signals (*SW*, *LOAD_DR*, *CLK*, *RST*, *LOAD_OP*1, *LOAD_OP*2) and the external output signals (*LD*, *SEG*, *EN_DISP*). The signal *SW* is data type std_logic_vector (3 **downto** 0), while the rest of the inputs are data type std_logic. The signal *LD* is data type std_logic_vector (3 **downto** 0). The signal *SEG* is data type std_logic_vector (6 **downto** 0). The signal EN_DISP is data type std_logic.
- The internal signal declarations must be placed between **architecture** and the first **begin**. Observe that DSM is used *for each internal signal*. This shows where each internal signal is placed and the modules each internal signal is between. This documentation style allows us to easily keep track of where each internal signal is located in the VHDL design.

- Observe that DSM is used *for each section of code*. This shows where each section of code is placed for each circuit element. This documentation style allows us to easily keep track of where the code for each circuit element is located in the VHDL design.
- As your VHDL code for a design grows, due to the design of larger circuits or systems, it is very important to follow a consistent documentation style that marks the location of each internal signal and the section of code for each circuit element (or module) in your VHDL design.

12.8 ALTERNATE VHDL DESIGN STYLES

It is usually easier to design a D flip-flop circuit, such as a loadable register, using a conditional signal assignment. If statements and case statements can also be used to design a D flip-flop circuit, but these statements also require a few more lines of code. When you write more lines of VHDL code, this sometimes contributes to the possibility of more errors in your code that must be found and corrected. Listing 12.8 shows an alternate design of module 2 (in Figure 12.10). The design uses a behavioral design style with two if statements.

LISTING 12.8
Module 2 code using two if statements for Figure 12.10

```
--Module 2 code, loadable DFFs with a clear input
process (rst, clk)
begin
     if rst = '1' then dr <= "0000";
     elsif rising_edge (clk) then
          if load_dr = '1' then dr <= di;
          end if;
     end if;
end process;
```

Listing 12.9 shows an alternate design of module 3 (in Figure 12.10). This design uses a behavioral design style with one if statement and one case statement.

LISTING 12.9
Module 3 code using one if statement and one case statement for Figure 12.10

```
--Module 3 code, loadable DFFs with a clear input
process (rst, clk)
begin
     if rst = '1' then op1 <= "0000";
     elsif rising_edge (clk) then
          case load_op1 is
               when '1' => op1 <= dr;
               when '0' => op1 <= op1;
               when others => null;
          end case;
     end if;
end process;
```

Listing 12.10 shows an alternate design of module 5 (in Figure 12.10). This design uses a behavioral design style with one if statement and two case statements.

In each of these examples, you can see that using a conditional signal assignment to generate a D flip-flop circuit requires less code and generally results in fewer errors. As a student, you should strive to learn the many different ways to write VHDL code.

```
--Module 5 code, loadable DFFs with a clear input
process (rst, clk)
begin
     case rst is
          when '1' => op2 <= "0000";
          when '0' =>  if rising_edge(clk) then
                              case load_op2 is
                                   when '1' => op2 <= dr;
                                   when '0' => op2 <= op2;
                                   when others => null;
                              end case;
                         end if;
          when others => null;
     end case;
end process;
```

LISTING 12.10 Module 5 code using one if statement and two case statements for Figure 12.10

PROBLEMS

Section 12.2 Designing Steering Circuits

12.1 Write the transfer function for a 1-bit steering circuit, and draw an annotated circuit diagram for the steering circuit.

12.2 Write a behavioral architecture declaration with an if statement for the transfer function in problem 12.1.

12.3 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the behavioral architecture declaration in problem 12.2.

Section 12.3 Designing Bus Steering Circuits

12.4 Write the transfer function for a 4-to-1 bus steering circuit with the following signals: inputs are *BA*, *BB*, *BC*, *BD*, and *BSEL*, and the output is *BE*. Draw a MUX array circuit that represents the transfer function if each data input *BA* through *BD* is 8 bits.

12.5 Write a dataflow architecture declaration with a conditional signal assignment for the transfer function in problem 12.4.

12.6 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the dataflow architecture declaration in problem 12.5.

Section 12.4 Designing Loadable Register Circuits

12.7 What is a register?

12.8 What is the transfer function for a loadable register circuit with inputs *RST* (asyn), *D*, *LOAD_R*, *CLK*, and output *R*?

12.9 Write a behavioral architecture declaration with a single if statement for the transfer function in problem 8 for a 2-bit loadable register circuit that uses two D flip-flops.

12.10 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the behavioral architecture declaration in problem 12.9.

Section 12.5 Designing Input Circuits

12.11 What is the IN instruction in both assembly language form and transfer function form for VBC1?

12.12 How are the input data supplied for the IN instruction for VBC1?

12.13 Why are the buffers used in the input circuit in Section 12.5, Figure 12.4? Provide two reasons.

12.14 Are the flip-flop inputs that are driven by the signal *RST* in the input circuit in Section 12.5, Figure 12.4, synchronous or asynchronous inputs? Are the clear inputs (CLR) active high or active low input?

12.15 What input in the loadable DFF allows the output (Q) to be changed to the data input (D) in the input circuit in Section 12.5, Figure 12.4? Under what conditions can the output (Q) be changed to the data input (D)?

12.16 Draw a simplified schematic for a 64-bit input circuit (or an input circuit driven by 64 slide switches). To make the circuit as simple as possible, do not include the buffers. Be sure to completely label the circuit—that is, *DI* with its range, *DR* with its range, *LOAD_DR*, *CLK*, and *RST* (asyn). How many loadable D flip-flops are required for the circuit?

12.17 What is the range for the std_logic_vector for the signal *DR* (destination register) for the 64-bit input circuit in problem 12.16?

12.18 Describe how the location of an internal signal in the circuit in Section 12.5.1, Figure 12.5, is marked in VHDL using documentation style M. Provide an example for signal *DI* using documentation style M. Also include the declaration for the signal.

12.19 Where do internal signals have to be placed in the VHDL architecture?

12.20 Describe how the location of a section of code represented by module 1 in the circuit in Section 12.5.1, Figure 12.5, is marked in VHDL using documentation style M. Provide an example for the code for module 1 using documentation style M. Also include the VHDL code.

Section 12.6 Designing Output Circuits

12.21 What is the OUT instruction in both assembly language form and transfer function form for VBC1?

12.22 Where does the output data go for the OUT instruction for VBC1?

12.23 Do the LEDs in Section 12.6, Figure 12.6, have active high inputs or active low inputs? Provide the reason for your answer, and also discuss the purpose of the resistors associated with the LEDs.

12.24 What type of VHDL design style is used in the architecture in Section 12.6.1, Listing 12.5? What type of VHDL statement is used in the section of code for module 1?

12.25 How would you change the signals *DR*, *OP*, and *LD* for the output circuit in Section 12.6.1, Listing 12.5, from 4 bits to 12 bits? Are any other changes in the VHDL design required?

12.26 What is the purpose of the inverters in Section 12.6.2, Figure 12.8? Is it possible to design the output circuit without the inverters? Discuss what must be done if the inverters are not used.

12.27 Can a process be used with a case statement for the section of code for module 2 in Section 12.6.2, Listing 12.6? If so, what signal must be included in the sensitivity list of the process that contains the case statement for the section of code for module 2.

12.28 What are the active levels of inputs CA through CG and AN0 in Section 12.6.3, Figure 12.9? Discuss what causes segments a through g to light or turn on.

12.29 What is the function of the protection resistors in Section 12.6.3, Figure 12.9?

Section 12.7 Combining Input and Output Circuits to Form a Simple I/O System

12.30 Draw a simplified schematic for a simple I/O system like the one shown in Section 12.7, Figure 12.10, that reads 8 bits via 8 switches, displays the upper 4 bits on the single LEDs, and displays the lower 4 bits on the 7-segment display. To make the circuit as simple as possible, do not include the buffers and the inverters. With modules 1, 4, 7, and 8 removed—that is, the buffers and inverters—what changes must be made to the remaining modules? What changes must be made to the port signals in the entity and the internal signals in the architecture? Include complete VHDL code for your design.

Section 12.8 Alternate VHDL Design Styles

12.31 What is the transfer function for a loadable register that receives its input from eight switches (*BA*7 through *BA*0) and stores the switch values at its output (*RA*7 through *RA*0) at the next rising edge of the clock when *LOAD_A* is asserted? If *LOAD_A* is not asserted, *RA* retains its current value.

12.32 Write the simplest dataflow architecture declaration for the transfer function

$$RA \leftarrow BA \; if \; (LOAD\_A = 1 \; and \uparrow CLK) \; else$$
$$RA \leftarrow RA$$

12.33 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the dataflow architecture declaration in problem 12.32 for 8 bits.

12.34 Combine your VHDL code for problems 12.32 and 12.33, and show a simulation for your design.

12.35 Write a behavioral architecture declaration with two if statements for the transfer function

$$RA \leftarrow BA \; if \; (LOAD\_A = 1 \; and \uparrow CLK) \; else$$
$$RA \leftarrow RA$$

12.36 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the behavioral architecture declaration in problem 12.35.

12.37 Combine your VHDL code for problems 12.35 and 12.36, and show a simulation for your design.

12.38 Repeat problem 12.35 using a behavioral architecture declaration with one if statement and one case statement.

12.39 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for the behavioral architecture declaration in problem 12.38.

12.40 Combine your VHDL code for problems 12.38 and 12.39, and show a simulation for your design.

Chapter 13

Designing Instruction Memory, Loading Program Counter, and Debounced Circuit

Chapter Outline

13.1 INTRODUCTION

In this chapter, you will learn how to write VHDL code for an instruction memory for storing instructions. Next, you will learn how to design a loading program counter for addressing memory to store instructions. Then, you will learn how to design a debounced one-pulse circuit for loading the instruction memory one instruction at a time. Finally, you will design a press-to-increment circuit, to allow verification of the debounce one-pulse circuit.

13.2 DESIGNING AN INSTRUCTION MEMORY

Instruction memory is important because that is where the instructions for a program are placed to be executed. Figure 13.1 shows a logic symbol with the necessary inputs and outputs for the instruction memory for VBC1.

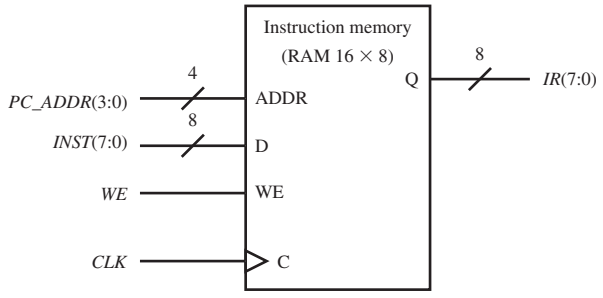


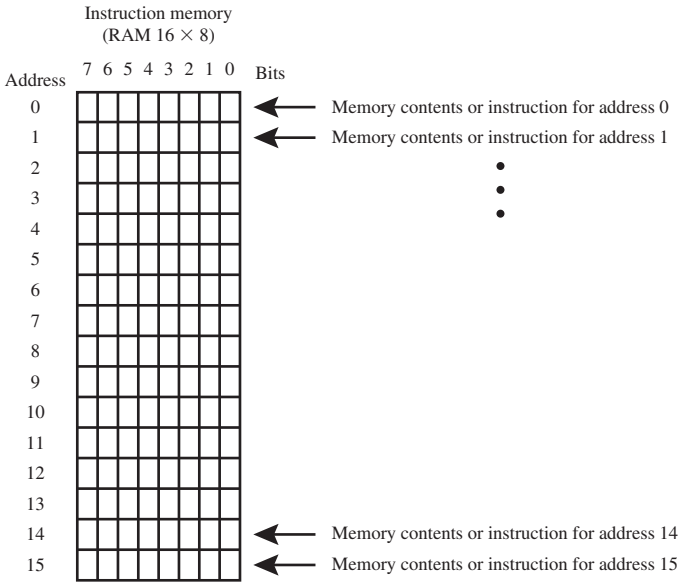
FIGURE 13.1
Logic symbol diagram for the instruction memory for VBC1

Things you should notice about the instruction memory in Figure 13.1:

- Input ADDR (address) selects the location of where an instruction is placed in the memory.
- Input D (data) provides the instruction that is placed in the memory.
- Input WE (write enable) enables data (an instruction) to be written into the memory at the next rising edge of the clock.
- Input C (control) provides the clock for storing (writing) instructions in the memory.
- Output Q provides the flip-flop outputs for the memory.
- ADDR, D, WE, C, and Q are labels.
- The signals for the memory are inputs *PC_ADDR* (program counter address), *INST* (instruction), *WE* (write enable), *CLK* (clock), and output *IR* (instruction register).
- The instruction memory map contains 16 memory locations, and each location has a storage capacity of 8 bits (or 1 byte) as shown in Figure 13.2.
- The instruction memory is referred to as a RAM (random-access memory) 16 by 8 (or 16 × 8). Because $16 = 2^4$, a minimum of four address lines are required to determine the address for each location in memory.
- The instruction memory is synchronous because storing (writing) data into each address location in the memory requires a clock tick (a rising edge of the signal *CLK*).

Figure 13.2 shows the instruction memory map for VBC1.

FIGURE 13.2 Instruction memory map for VBC1



Listing 13.1 shows a complete VHDL design for the instruction memory in Figure 13.1.

LISTING 13.1 Complete VHDL design for the instruction memory for VBC1 (project: Instruction_Memory)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity instruction_memory is port (
   pc_addr : in std_logic_vector (3 downto 0);
   inst : in std_logic_vector (7 downto 0);
   we, clk : in std_logic;
```

```
    ir : out std_logic_vector (7 downto 0)
    );
end instruction_memory;

architecture Mixed of instruction_memory is
    type mem_type is array (0 to 15) of std_logic_vector (7 downto 0);
    signal mem : mem_type;
begin
process (clk)
begin
    if rising_edge (clk) then
       if we = '1' then mem (conv_integer (pc_addr)) <= inst;
       end if;
    end if;
end process;
    ir <= mem (conv_integer (pc_addr));
end Mixed;
```

Things you should notice about the VHDL design in Listing 13.1:

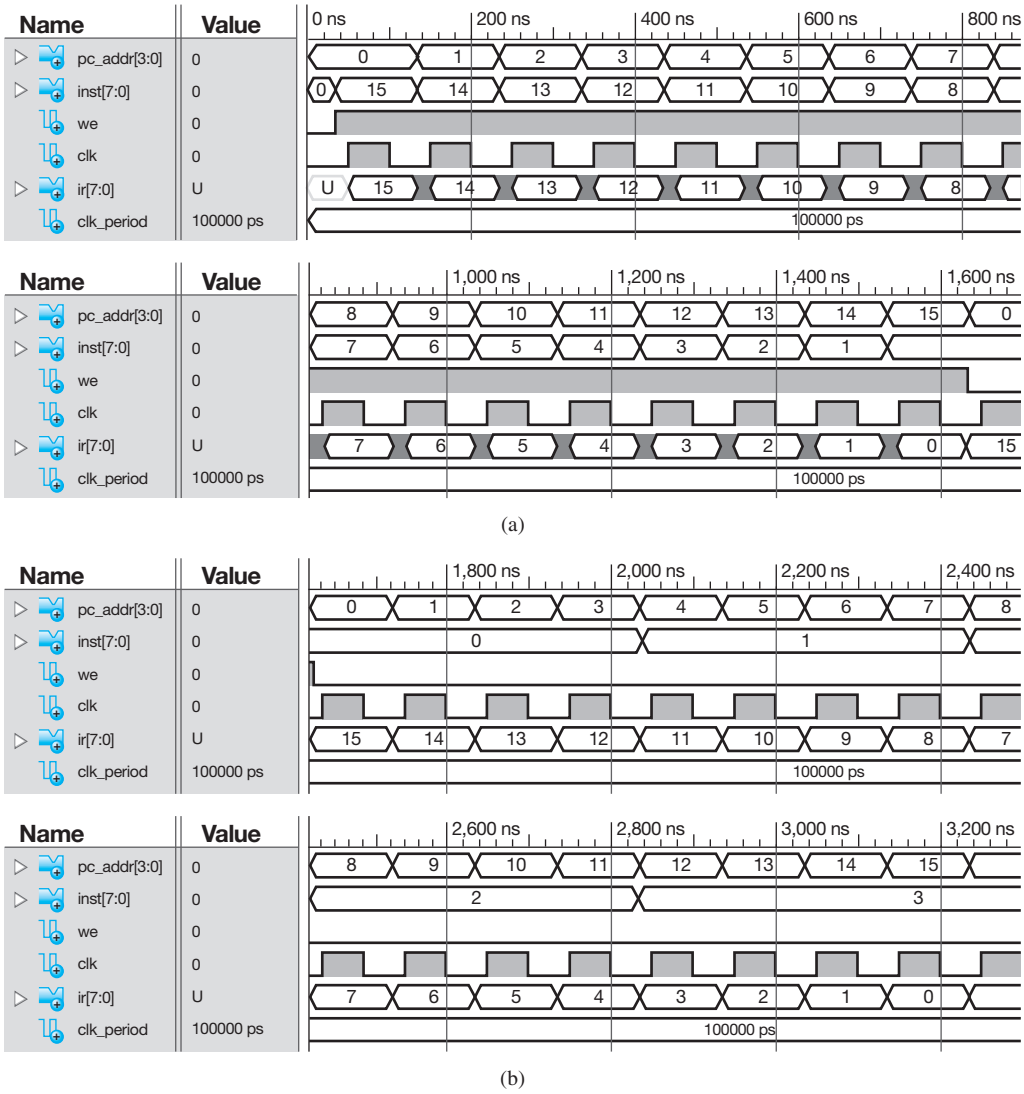
- The use clause **use** IEEE.STD_LOGIC_UNSIGNED.**ALL** is required because it contains the definition for the conversion function conv_integer, which is used in the design.
- A new data type called **mem_type** is used to declare an array of vectors for the instruction memory. Mem_type is called an **enumerated data type** with the ordered values array (0 to 15) of the data type std_logic_vector (7 downto 0), and it is a unique new data type. An internal **signal** mem is declared as the data type called mem_type. Signal mem is also a unique new signal of type mem_type, not of the type std_logic or std_logic_vector.
- Only the signal *CLK* is necessary in the sensitivity list of the process because there is no reset or initialization signal. The synchronous signals *WE* and *PC_ADDR* cannot be read until the clock ticks, and so they are not required in the sensitivity list.
- The conversion function conv_integer converts the signal *PC_ADDR* from a 4-bit Boolean signal of type std_logic_vector to a decimal integer for the array of type mem_type.
- The design style is a mixed design style because a process and the Boolean equation *IR* <= *MEM* (conv_integer (pc_addr)) are used in the architecture declaration of the design.

Waveform 13.1 shows the correct functionality of design entity instruction_memory. Things you should notice about the waveforms in Waveform 13.1:

- In the waveform diagrams in panel a, the memory address is changed from 0 through 15 via the input signal *PC_ADDR*.
- Random values of data that represent instructions (machine code instructions) are written into the instruction memory each time the clock ticks because *WE* is set to 1 as shown by the output signal *IR*.
- In the waveform diagrams in panel b, *WE* is set to 0, which allows the instruction memory to be read. By closely observing the waveform diagrams, it can be seen that the data written into each memory address in panel a is read correctly in panel b confirming that the instruction memory is working properly.

13.2.1 Coding Alterations for Instruction Memory

Listing 13.2 shows the complete VHDL design for the instruction memory in Figure 13.1 with some coding alterations to show a slightly different design style.



WAVEFORM 13.1 Simulation for the correct functionality of design entity instruction_memory: (a) writing instructions into memory; (b) reading previously written instructions from the memory

LISTING 13.2
Altered complete VHDL design for the instruction memory for VBC1 (project: Instruction_Memory_MOD)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity instruction_memory_mod is port (
   pc_addr : in std_logic_vector (3 downto 0);
   inst : in std_logic_vector (7 downto 0);
   we, clk : in std_logic;
   ir : out std_logic_vector (7 downto 0)
   );
end instruction_memory_mod;
```

```
architecture Behavioral of instruction_memory_mod is
   type mem_type is array (0 to 15) of std_logic_vector (7 downto 0);
   signal mem : mem_type;
begin
process (clk, mem)
begin
   if rising_edge (clk) then
      if we = '1' then mem (conv_integer (pc_addr)) <= inst;
      end if;
   end if;
   ir <= mem (conv_integer (pc_addr));
end process;
end Behavioral;
```

Can you spot the changes? Things you should notice about the VHDL design in Listing 13.2:

- The design style was changed from mixed to behavioral. Now the architecture declaration only contains a process—that is, the Boolean equation *IR* <= *MEM* (conv_integer (*PC_ADDR*)) is no longer outside the process.
- The assignment *IR* <= *MEM* (conv_integer (*PC_ADDR*)) has been moved inside the process. Because the assignment for *IR* is not located inside the process, the signal *MEM* must be added to the sensitivity list of the process so that the value of signal *MEM* is read during the rising edge of *CLK* and assigned to *IR*.
- The name of the architecture was changed from mixed to behavioral to indicate that the design is now behavioral—that is, the design consists of only a process.

The simulation for the altered complete VHDL design in Listing 13.2 is the same as Waveform 13.1. So, the VHDL code for Listing 13.1 or 13.2 can be use as the instruction memory for VBC1.

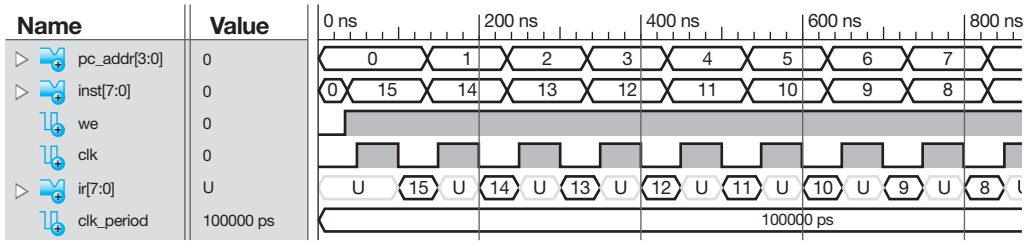
Waveform 13.2 shows what happens when the signal *MEM* is left out of the sensitivity list in the VHDL code for Listing 13.2 (project: IM_MOD_WARNING).

Things you should notice about the Waveforms in Waveform 13.2:

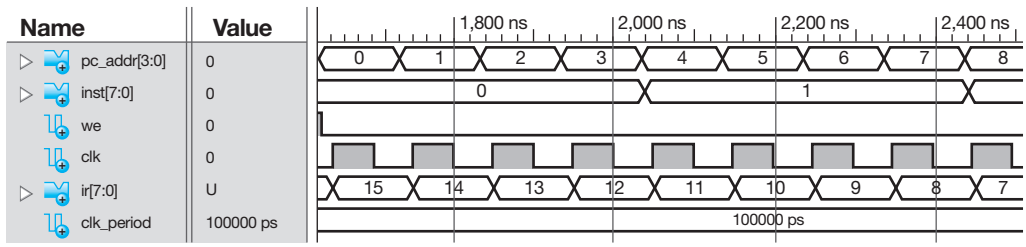
- Observe that the instruction memory stores the correct values on the falling edge of the clock pulse, not the rising edge of the clock pulse as it should. This caused a warning to be reported after running Synthesize—XST. To correct this problem, *MEM* must be added to the sensitivity list of the process, or the assignment *IR* <= *MEM* (conv_integer (*PC_ADDR*)) must be placed outside the process—that is, not between **process** and **end process**.
- The values that are stored in the instruction memory are read correctly. Leaving necessary signals out of the sensitivity list may result in hardware designs that do not follow your design specification.
- Code verification for a design is very important. A simulation shows you if your code generates the expected result. To verify the correct functionality of a design, get in the habit of running a simulation for your design.

13.2.2 Initializing Instruction Memory for VBC1 at Startup

Instruction Memory can be initialized at startup—that is, when you download a program into an FPGA.



(a)



(b)

WAVEFORM 13.2 Simulation with the signal *MEM* left out of the sensitivity list in Listing 13.2: (a) writing instructions into memory; (b) reading previously written instruction from the memory

The following VHDL code shows how to initialize the instruction memory for VBC1 at startup. Listing 13.3 shows how to initialize instruction memory for a stoppable 4-bit binary-up counter program.

LISTING 13.3
Initializing instruction memory for a stoppable 4-bit binary-up counter program

```
type mem_type is array (0 to 15) of std_logic_vector (7 downto 0);
signal mem: mem_type := (
    --stoppable 4-bit binary-up counter program
        X"20", X"c0", X"b0", X"f2",X"61", X"e1", X"21", X"e0",
        X"00", X"00", X"00", X"00", X"00", X"00", X"00", X"00");
```

Using this technique, a value must be specified for each of the 16 locations in instruction memory. The operator ":=" is called an **assignment operator** and is used to assign initial val-

ues. Each value represents a VBC1 instruction in machine code. Uppercase X specifies that the machine code is provided in hexadecimal. To observe the machine code in hexadecimal, choose Hex in the Display section of EASY1.

The state sequence diagram for the stoppable 4-bit binary-up counter is presented in Appendix D in Figure D.14, and the assembly language program is presented in Appendix D in Program D.4. In the Programmer's Register Model part of EASY1, we chose Hex in the Display section to display the machine code in hexadecimal. The machine code in hexadecimal was then inserted as shown in Listing 13.3 to initialize the instruction memory at startup.

An easier way to obtain the machine code is to use the EASY1 Save button, which saves a copy of the machine code shown in the instruction memory into a file (choose a text file). After you save the file, simply open the file then copy and paste the machine code into the VHDL code for the instruction memory on the line following the signal for the memory. The assembly language program is also provided as a handy reference. If you elect to copy and paste the machine code, remember to change the semicolon ";" after mem_type to ":= (". This procedure is used in the following example.

Figure 13.3 shows the output 4-bit LED sequence for a robot eye program for VBC1.

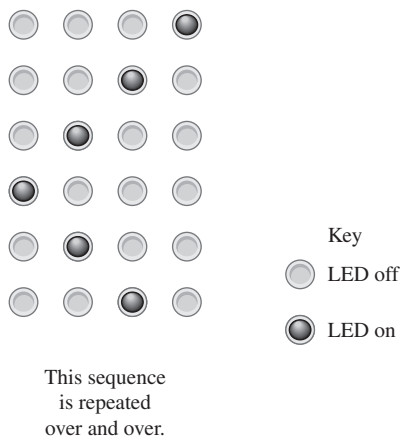


FIGURE 13.3 Output 4-bit LED sequence for a robot eye program

Program 13.1 shows an assembly language program for the robot eye output sequence in Figure 13.3 using inline programming; that is, no internal loops or no loop counters are used in the program.

```
; robot eye program
; inline programming, i.e., no internal loops

start:   loadi r1,1
         out r1
         loadi r1,2
         out r1
         loadi r1,4
         out r1
         loadi r1,8
         out r1
         loadi r1,4
         out r1
         loadi r1,2
         out r1
         jnz r1, start
```

PROGRAM 13.1 Assembly language program for a robot eye program

Be sure to study the assembly language program so that you know how it works. To obtain the machine code for the assembly language program in Program 13.1, enter the program in the

Assembly part of EASY1, then assemble and load the program by clicking on the Assemble and Load button. To observe the machine code in hexadecimal in the instruction memory of EASY1, choose Hex in the Display section of EASY1. You may elect to copy each machine code value into your VHDL code in the format shown in Listing 13.3, or you may elect to copy and paste the machine code into your VHDL code. If you elect to copy and paste the machine code, click on the EASY1 Save button and save the file. Open the saved file and copy and paste the machine code after the statement **signal** mem: mem_type;. Be sure to change the semicolon ";" after mem_type to ":= (" as shown in Listing 13.4.

LISTING 13.4
Initializing instruction memory for the robot eye program via copy and paste

```
type mem_type is array (0 to 15) of std_logic_vector (7 downto 0);
signal mem: mem_type := (

      -- ; robot eye program
      -- ; inline programming, i.e., no internal loops
      --
X"31",-- start:   loadi r1,1
X"d0",--      out r1
X"32",--      loadi r1,2
X"d0",--      out r1
X"34",--      loadi r1,4
X"d0",--      out r1
X"38",--      loadi r1,8
X"d0",--      out r1
X"34",--      loadi r1,4
X"d0",--      out r1
X"32",--      loadi r1,2
X"d0",--      out r1
X"f0",--      jnz r1, start
      --

X"00",X"00",X"00");
```

Each of the instructions that are used in the initialization of the instruction memory must be implemented in hardware in order for the program to run successfully when the VHDL code is downloaded into an FPGA.

Initializing instruction memory is useful when you want to load a program automatically at startup or when you want to load a program and there are not enough hardware switches to design the circuitry to manually load a program. We used 8 bits for the instructions for VBC1 because there were eight hardware switches available on the hardware boards we elected to use. Using the eight hardware switches along with some additional pushbutton switches, we will design the circuitry to manually load our programs after startup for VBC1.

Additional comment about counter design: The last two counter designs could be designed from scratch using VHDL code. After we complete the design for VBC1, it is often easier to write a program to generate a digital design using VBC1 rather than start from scratch and generate the design via VHDL code.

13.3 DESIGNING A LOADING PROGRAM COUNTER

VBC1 uses two program counters (PCs). The first type of program counter we will discuss is a loading program counter (LPC). The LPC is used to load instructions manually into instruction

memory via a set of slide switches. The second type of program counter, which we will discuss in Chapter 17, is a running program counter (RPC). The RPC is used to manually single step through a program and also to run a program at a specified frequency.

Figure 13.4 shows the logic symbol with the necessary inputs and output for a loading program counter for VBC1. The LPC is used to load instructions manually into the instruction memory of VBC1.

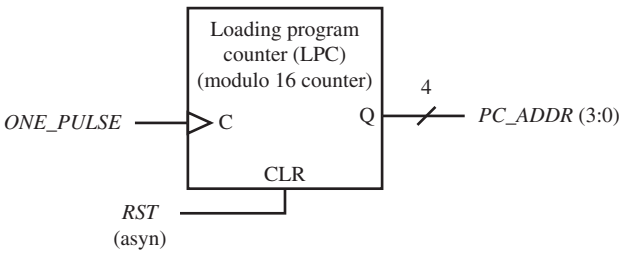


FIGURE 13.4 Logic symbol for a loading program counter for VBC1

Things you should notice about the LPC in Figure 13.4:

- An asynchronous reset signal *RST* (asyn) is used to reset the LPC.
- A signal called *ONE_PULSE* is supplied to the control (or clock) input of the loading program counter. The clock signal is generated manually for VBC1 by a push-button switch, which will be discussed later.
- The signal *PC_ADDR* is a 4-bit output that supplies an address to the instruction memory for loading the instruction memory.
- Because the LPC has 16 states, it is referred to as a modulo 16 counter.
- The LPC is a simple state machine because its counting sequence is fixed—that is, it has no external inputs to affect its counting sequence.

Figure 13.5 shows the counting sequence diagram or state sequence diagram and the equivalent state diagram for the LPC. Observe that the LPC is a binary-up counter. Because the loading program counter has 16 states, it is referred to as a modulo 16 counter; that is, a modulo n counter is a counter with n states where $n > 1$.

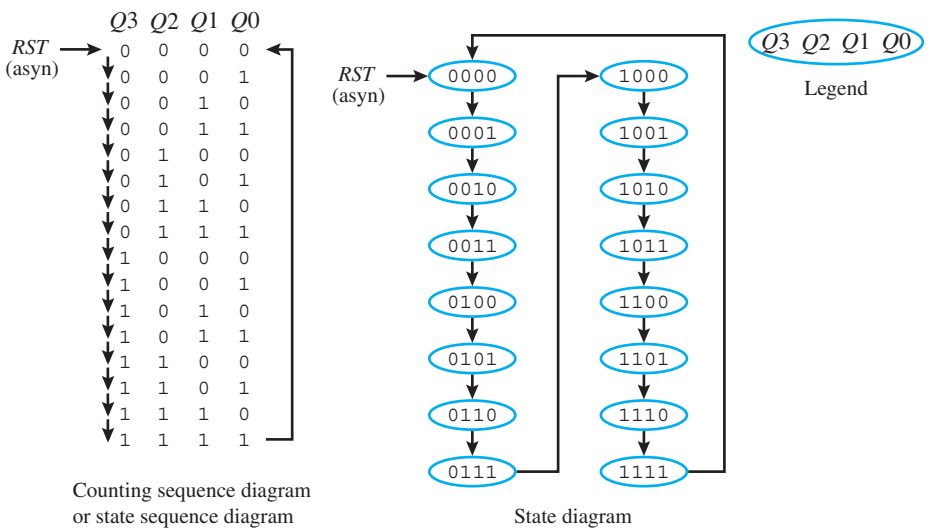


FIGURE 13.5 Counting sequence diagram or state sequence diagram and equivalent state diagram for the loading program counter for VBC1

Listing 13.5 shows a complete VHDL design for the loading program counter in Figure 13.5.

LISTING 13.5
Complete VHDL design for the LPC for VBC1 (project: LPC)

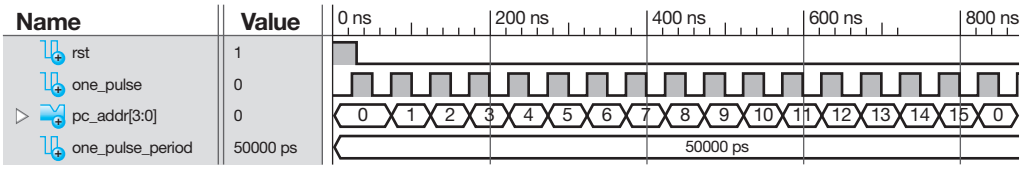
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LPC is port (
     rst, one_pulse : in std_logic;
     pc_addr : out std_logic_vector (3 downto 0)
     );
end LPC;

architecture Mixed of LPC is
     signal q : std_logic_vector (3 downto 0);
begin
process (rst, one_pulse)
begin
     if rst = '1' then q <= "0000";
     elsif rising_edge (one_pulse) then q <= q + 1;
     end if;
end process;
     pc_addr <= q;
end Mixed;
```

Things you should notice about the VHDL design in Listing 13.5:

- The use clause **use** IEEE.STD_LOGIC_UNSIGNED.**ALL** is required because it contains the definition for "+" operator, which is used in the design.
- The internal signal *Q* is used to agree with variable *Q* specified in the counting sequence or state diagram for the loading program counter. It is not necessary to use *Q* if the variable *PC_ADDR* is used as the variable in the counting sequence or state diagram.
- Only *RST* and *ONE_PULSE* are required in the sensitivity list of the process, because they are the only signals required to wake up the process. The signal *Q* is not required because it is a synchronous signal that is controlled by the clock signal *ONE_PULSE*.
- The arithmetic method is used in an if statement.
- The output signal *PC_ADDR* is assigned the value of the internal signal *Q* outside the process—that is, not between **process** and **end process**.
- The design style is a mixed design style because a process and the Boolean equation *PC_ADDR* <= *Q* are used in the architecture declaration of the design.
- It would not be a good design practice to include the Boolean equation inside the process without including *Q* in the sensitivity list, because *PC_ADDR* would not get assigned *Q* until the clock goes low, thus delaying the output *PC_ADDR* assignment by half a clock cycle. Failing to include *Q* in the sensitivity list for this design would be a design error, because the intention of the design is to provide a state change on the rising edge of the clock cycle not the falling edge of the clock cycle.

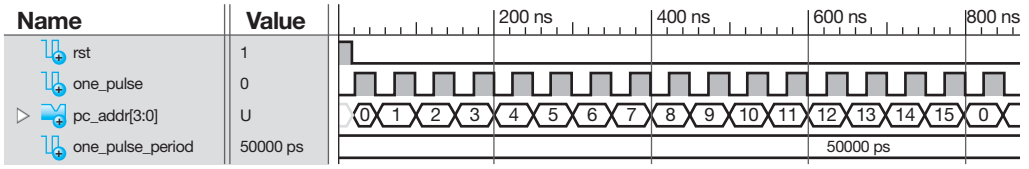
Waveform 13.3 shows the correct functionality of design entity LPC (loading program counter).

WAVEFORM 13.3
Simulation for the correct functionality of design entity LPC



Notice that Waveform 13.3 shows that each state change occurs when *RST* = 0 after the rising edge of the clock signal *ONE_PULSE*—that is, after each clock tick—as expected.

Waveform 13.4 shows an incorrect functionality of design entity LPC. In the VHDL code for this simulation, the assignment *PC_ADDR* <= *Q* was moved inside the process in Listing 13.5 just before **end process** without including *Q* in the sensitivity list (project: RPC_WARNING).



WAVEFORM 13.4
Simulation with the signal *Q* left out of the sensitivity list in Listing 13.5

Things you should notice about the waveforms in Waveform 13.4:

- Notice that Waveform 13.4 shows that each state change occurs when *RST* = 0 after the falling edge of the clock signal *ONE_PULSE*, not the rising edge of the clock signal as it should be. This caused a warning to be reported after running Synthesize—XST.
- The LPC also does not reset asynchronously like it should as shown in Waveform 13.3. To correct this problem, *Q* must be added to the sensitivity list of the process, or the assignment *PC_ADDR* <= *Q* must be placed outside the process—that is, not between **process** and **end process**.
- This is another reminder that code verification for a design is very important. To verify the correct functionality of a design, you need to run a simulation for the design.

13.4 DESIGNING A DEBOUNCED ONE-PULSE CIRCUIT

In this section, we will design a debounced one-pulse circuit (DOPC) for single stepping through the memory for VBC1. The output of the DOPC should generate just one pulse each time a push-button switch is pressed. Before the switch is pressed, the output should be zero. When the push-button switch is pressed and continuously held down, one pulse should be generated—that is, the output should go from zero to one and back to zero. After the push-button switch is released, the output should remain at zero until the push button is pressed again to generate another pulse.

Figure 13.6 shows a push-button switch and a logic symbol for a debounced one-pulse circuit. A push-button switch or practically any switch bounces on its contacts as it is closed or opened. Figure 13.6 also shows the switch output waveform *SEL_ADDR* when the push button is not pressed, then pressed and held, and finally released, and the DOPC outputs *ONE_PULSE*. Notice that the waveform *SEL_ADDR* shows multiple pulses when the push button is pressed and released due to the bouncing of the switch contacts—that is, contact bounce—while the waveform *ONE_PULSE* shows one clean pulse.

Figure 13.7 shows a register transfer level (RTL) circuit for a debounced one-pulse circuit. Things you should notice about the RTL circuit for the DOPC:

- If the signal *SEL_ADDR* is 1 (or high) longer than two clock ticks, then the output *ONE_PULSE* goes to 1 because *Q*1 and *Q*2 are 1 and *Q*3 is 0. Output *ONE_PULSE* remains a 1 only for the duration of one clock cycle, or $T_{CLK} = 1/f_{CLK}$.
- The following things can cause the signal *SEL_ADDR* to be 1 (or high) longer than two clock ticks: (a) when contact bounce of the push-button switch is 1 (or high) longer than two clock ticks and (b) when the push-button switch is held down longer than two clock ticks.

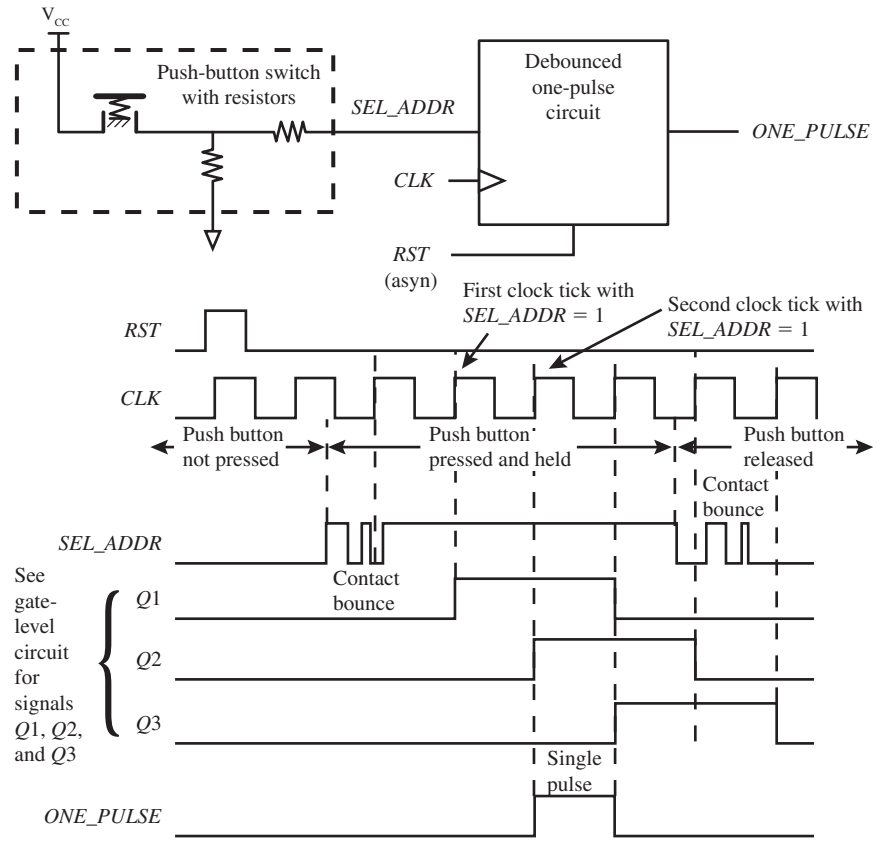


FIGURE 13.6 Push-button switch, logic symbol for a debounced one-pulse circuit and input and output waveforms for the DOPC

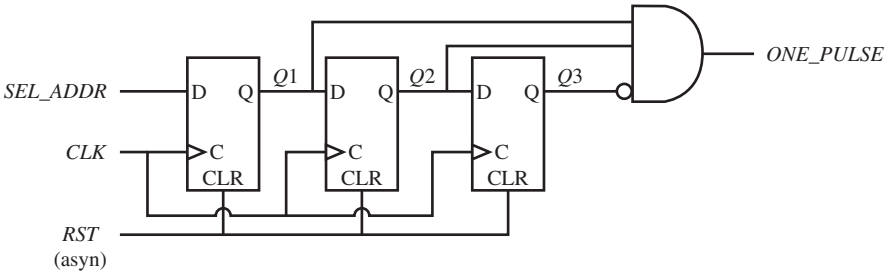


FIGURE 13.7 Register transfer level circuit for a DOPC

- To help prevent contact bounce from causing the select signal *SEL_ADDR* to be a 1 (or high) longer than two clock ticks simply requires reducing the frequency of *CLK* (by experimentation, i.e., trial and error), so that the longest duration of contact bounce is less than T_{CLK}(or the duration of T_{CLK} is longer than the longest duration of contact bounce). If the frequency of *CLK* is fixed, then a frequency divider can be used to supply a reduced frequency of *SLOW_CLK* to the register transfer level circuit so that the duration of T_{SLOW_CLK} is longer than the longest duration of contact bounce.
- An RTL circuit is generally referred to as any synchronous digital circuit that transfers data via flip-flops, which are registers, to a combinational logic circuit.

Listing 13.6 shows a complete VHDL design for the debounced one-pulse circuit in Figure 13.7.

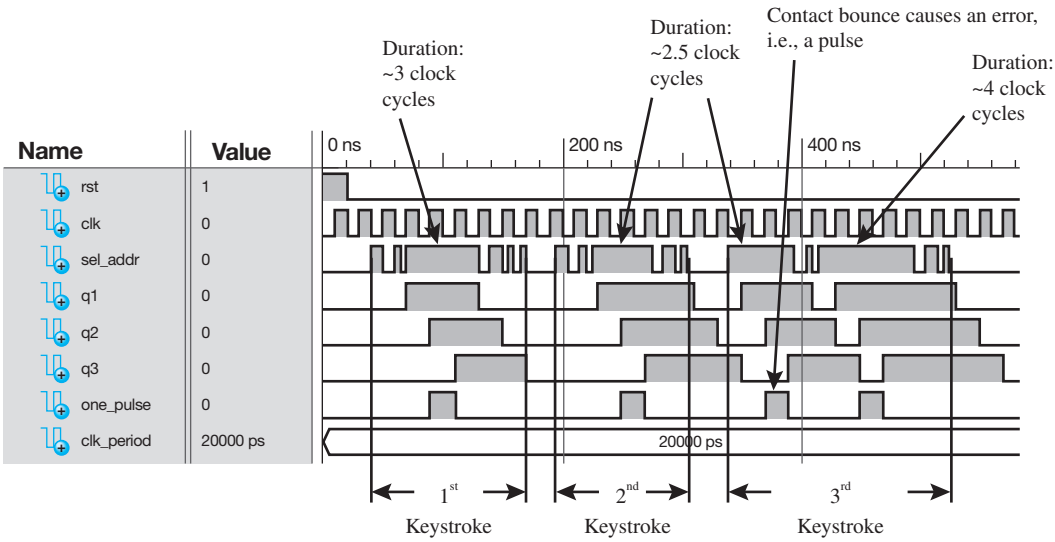
LISTING 13.6
Complete VHDL design for the DOPC (project: debounced_OPC)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity debounced_OPC is port (
     rst, clk, sel_addr : in std_logic;
     Q1, Q2, Q3 : inout std_logic;
     one_pulse : out std_logic
     );
end debounced_OPC;

architecture Mixed of debounced_OPC is
begin
process (rst, clk)
begin
     if rst = '1' then Q1 <= '0'; Q2 <= '0'; Q3 <= '0';
     elsif (clk'event and clk = '1') then Q1 <= sel_addr;
                                          Q2 <= Q1; Q3 <= Q2;
     end if;
end process;
     one_pulse <= Q1 and Q2 and not Q3;
end Mixed;
```

Things you should notice about the VHDL design in Listing 13.6:

- The internal signals *Q*1, *Q*2, and *Q*3 are placed in the entity, so they can be displayed in the simulation.
- In the process, the expression (clk'event **and** clk = '1') is used instead of rising_edge (clk). When VHDL was first introduced, this was the only way to represent rising_edge (clk). An alternate way of writing falling_edge (clk) may be written as (clk'event **and** clk = '0'). You may see these alternate expressions in references and books, and you need to know their meaning. The expression "clk'event" should be read as "clk tick event."
- From analyzing the VHDL code, you may observe that *ONE_PULSE* provides an output of 1 only when *Q*1 and *Q*2 are 1 and *Q*3 is 0.

Waveform 13.5 shows the correct functionality of design entity debounced_OPC (debounced one-pulse circuit).



WAVEFORM 13.5
Simulation for the correct functionality of design entity debounced_OPC

Notice that the signal *SEL_ADDR* is the keystroke output for the push-button switch circuit, which is also the input to the debounced one-pulse circuit. For the first keystroke, there are two narrow pulses (contact bounce) followed by a longer pulse (the push button is held down) followed by three narrow pulses (contact bounce). The output signal *ONE_PULSE* provides a single pulse, as it should. For the second keystroke, there are two narrow pulses (contact bounce) followed by a longer pulse (the push button is held down) followed by two narrow pulses (contact bounce). The output signal *ONE_PULSE* provides a single pulse, as it should. For the third keystroke, there is one wide pulse (contact bounce) followed by one narrow pulse (contact bounce) followed by a longer pulse (the push button held down) followed by two narrow pulses (contact bounce). The output signal *ONE_PULSE* provides a single pulse for the wide pulse (contact bounce) and a second single pulse for the longer pulse (push button is held down). The simulation is correct because the wide pulse (contact bounce) and the longer pulse (push button is held down) both last longer than two clock ticks.

13.5 DESIGN VERIFICATION FOR A DEBOUNCED ONE-PULSE CIRCUIT

Figure 13.8 shows a press-to-increment circuit (PTIC) that will allow verification of the DOPC by testing an actual hardware implementation.

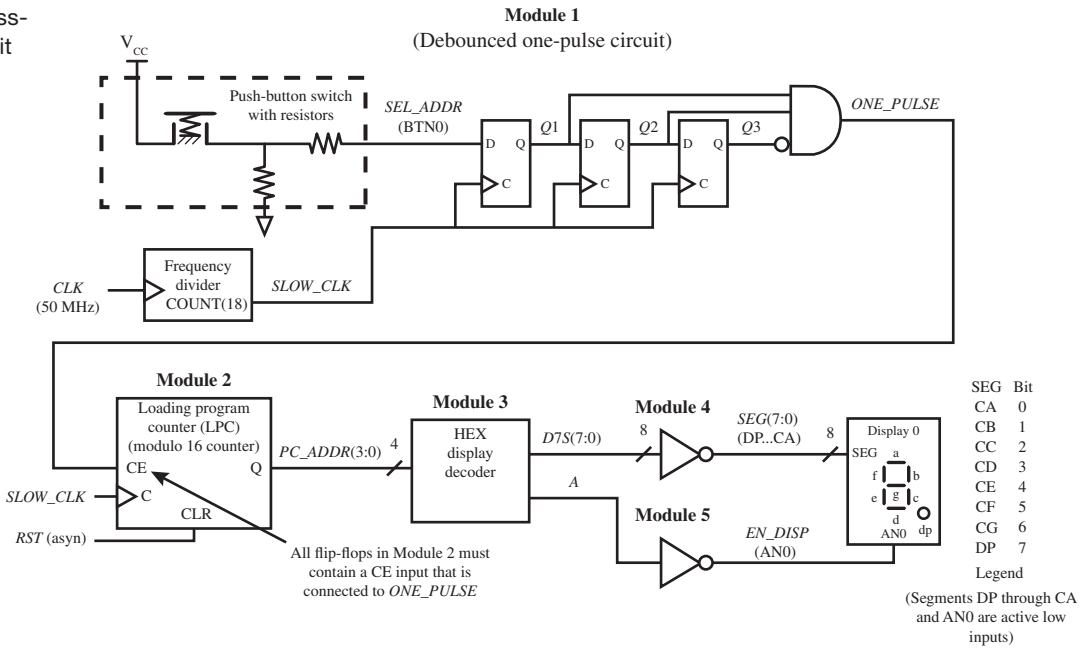


FIGURE 13.8 Press-to-increment circuit

Things you should notice about the press-to-increment circuit in Figure 13.8:

- Module 1 is the debounced one-pulse circuit in Figure 13.7 with a frequency divider added. The frequency divider must be added to the register transfer level circuit to reduce the frequency of *CLK*—that is, 50 MHz—to a slower frequency—that is, *SLOW_CLK*—to help prevent contact bounce from causing unintentional output pulses. We chose to reduce the frequency of *CLK* by 2^{19}, which results in a period for *SLOW_CLK* of $T_{SLOW\_CLK} = 1/f_{SLOW\_CLK} = 1/[(f_{clk})/2^{19}] = 1/[(50 \times 10^6 \text{ Hz})/2^{19}] = 10.5$ ms. With this period, contact bounce, via the push-button switch, must not last longer than $T_{SLOW\_CLK} = 10.5$ ms, which is considered a long duration compared to the contact bounce duration of most small push-button switches.

- Module 2 is a modified version of the loading program counter in Figure 13.4 with a CE (clock enable) input. The CE input is used to enable the clock input of the LPC. If signal *ONE_PULSE* were to be supplied directly to the dynamic input C (or clock input) as shown in Figure 13.4, this would create a **gated clock circuit** or simply a **gated clock**. A gated clock is considered a bad design and should not be used. In general, if a clock net is sourced by a combinational circuit output, the circuit is called a gated clock. Applying signal *ONE_PULSE* to the CE input of module 2 and applying signal *SLOW_CLK* to the clock input as shown in Figure 13.8 removes the gated clock. A gated clock circuit will be discussed later. If you get a gated clock warning in your designs, you need to remove the warning by removing the gated clock.
- Modules 4 and 5 are used to drive the 7-segment display. An additional bit is added to the HEX display decoder to turn off the decimal point DP.
- For the circuit to work correctly, each push-button keystroke should provide a single pulse and increment the loading program counter by 1. This should be observed via display 0.

Listing 13.7 shows a complete VHDL design for the press-to-increment circuit in Figure 13.8.

LISTING 13.7
Complete VHDL design for the PTIC (project: PTIC)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity PTIC is port (
     rst,clk,sel_addr: in std_logic;
     seg : out std_logic_vector (7 downto 0);
     en_disp : out std_logic;
     en_disp1 : out std_logic;--added to turn off Display1
     en_disp2 : out std_logic;--added to turn off Display2
     en_disp3 : out std_logic --added to turn off Display3
     );
end PTIC;

architecture Mixed of PTIC is
--Module 1 internal signal, count, q1, q2, q3
     signal count : std_logic_vector (18 downto 0);
     signal q1, q2, q3 : std_logic;
--Modules 1 and 2 internal signal, one_pulse, slow_clk
     signal one_pulse, slow_clk : std_logic;
--Module 2 internal signal, q
     signal q : std_logic_vector (3 downto 0);
--Module 2 and 3 internal signal, pc_addr
     signal pc_addr : std_logic_vector (3 downto 0);
--Module 3 and 4 internal signal, d7s
     signal d7s : std_logic_vector (7 downto 0);
--Module 3 and 5 internal signal, a
     signal a : std_logic;
begin

--Module 1 code, Debounced One_Pulse Circuit
     count <= count + 1 when rising_edge (clk) else
              count;
     slow_clk <= count (18);
process (slow_clk)
```

(Continued)

```
begin
   if rising_edge (slow_clk) then q1 <= sel_addr; q2 <= q1;
                                  q3 <= q2;
   end if;
end process;
   one_pulse <= q1 and q2 and not q3;

--Module 2 code, Loading Program Counter
process (rst, slow_clk)
begin
   if rst = '1' then q <= "0000";
   elsif rising_edge(slow_clk) and one_pulse = '1' then q <= q + 1;
   end if;
end process;
   pc_addr <= q;

--Module 3 code, HEX Display Decoder
   d7s <= "00111111" when pc_addr = "0000" else
          "00000110" when pc_addr = "0001" else
          "01011011" when pc_addr = "0010" else
          "01001111" when pc_addr = "0011" else
          "01100110" when pc_addr = "0100" else
          "01101101" when pc_addr = "0101" else
          "01111101" when pc_addr = "0110" else
          "00000111" when pc_addr = "0111" else
          "01111111" when pc_addr = "1000" else
          "01101111" when pc_addr = "1001" else
          "01110111" when pc_addr = "1010" else
          "01111100" when pc_addr = "1011" else
          "00111001" when pc_addr = "1100" else
          "01011110" when pc_addr = "1101" else
          "01111001" when pc_addr = "1110" else
          "01110001";
   a <= '1';

--Module 4 code, inverter
   seg <= not d7s;

--Module 5 code, inverter
   en_disp <= not a;

--Added to turn off Displays 1, 2, and 3
   en_disp1 <= '1';--added to turn off Display1
   en_disp2 <= '1';--added to turn off Display2
   en_disp3 <= '1';--added to turn off Display3
end Mixed;
```

Things you should notice about the VHDL design in Listing 13.7:

- The signals *EN_DISP*1, *EN_DISP*2, and *EN_DISP*3 are declared in the entity so that displays 1 through 3 can be turned off. These displays turn on when they are not provided with signals that disable them.
- A mixed design style is used in the architecture declaration.

- The internal signals in the design are declared between the **architecture** and the first **begin** using documentation style M presented earlier in Chapter 12, Section 12.5.1.
- The VHDL code for modules 1 through 5 is listed in numerical order and is marked using documentation style M presented earlier in Chapter 12, Section 12.5.1.
- The code for modules 1 and 2 uses processes with if statements. The code for module 3 uses a conditional signal assignment. The code for modules 4 and 5 uses Boolean equations.

If you elect to download the design on a hardware board, you will observe that the display increments each time you press and release the push-button switch. After resetting the program counter, each pushbutton keystroke will provide the sequence 1, 2, 3, . . . , D, E, F, 0, 1, 2, 3, . . . on the 7-segment display. Each push-button keystroke should provide only one pulse and will not skip through the counting sequence.

Figure 13.9 shows a bad circuit; that is, this is one form of a gated clock circuit. In this circuit, the flip-flop can be triggered by a glitch; that is, a glitch is an undesirable momentary pulse that occurs at the output of the AND gate, as illustrated in the waveform diagram in Figure 13.9.

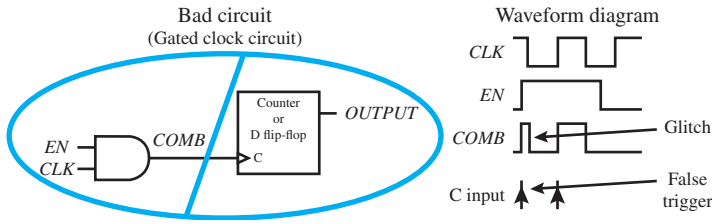


FIGURE 13.9 Bad gated clock circuit and corresponding waveform diagram

In the waveform diagram in Figure 13.9, observe that the C input of the counter gets two rising edges for a single rising edge of the *CLK* signal when $EN = 1$ due to the glitch at the output of the AND gate. Any combinational circuit output regulating a clock signal that is supplied to the C input of a single flip-flop or a set of flip-flops such as a counter is called a gated clock, and this type of circuit can produce or create a glitch, which may result in a false trigger. When a gated clock is used in your VHDL design, this usually causes a warning to be issued by the ISE software that this is not good design practice. As we mentioned earlier, if you get a gated clock warning in your designs, you need to remove the warning by removing the gated clock circuit.

Figure 13.10 shows a good circuit that can be used to remove the gated clock circuit shown in Figure 13.9. This circuit does not produce a false trigger at its C input.

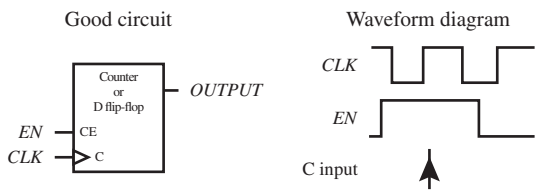


FIGURE 13.10 Good circuit to control the C input via an *EN* signal and corresponding waveform diagram

In the waveform diagram in Figure 13.10, observe that the C input of the counter gets only one rising edge, or the C input causes the counter to only count for each rising edge of the *CLK* signal when $EN = 1$. Because the circuit in Figure 13.10 does not have a combinational circuit output regulating a clock signal that is supplied to the C input, no warning will be issued by the ISE software when this circuit is used in a VHDL design.

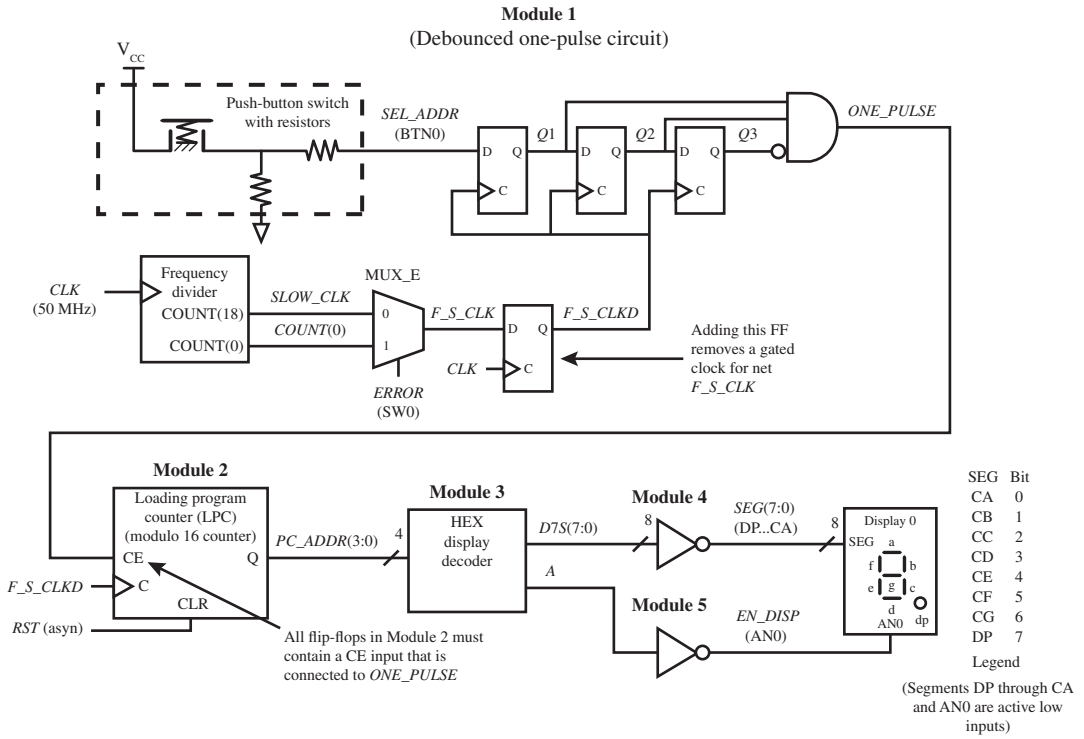
The good circuit in Figure 13.10 is used in the press-to-increment circuit in Figure 13.8 to prevent a gated clock error message from occurring.

Figure 13.11 shows a modification to the PTIC in Figure 13.8.

Things you should notice about the modified press-to-increment circuit in Figure 13.11:

- The circuit in Figure 13.11 allows you to observe what happens when the signal *CLK* is not slowed down enough before it is applied to the first flip-flop in the debounced one-pulse circuit.

FIGURE 13.11
Modified PTIC



- MUX_E is added to the design to allow you to switch between the frequencies *SLOW_CLK* and *COUNT*(0) via a slide switch, which provides the signal *ERROR*. When the signal *ERROR* is 1, a push-button keystroke may cause the circuit to skip through the binary-up counting sequence when it is pressed. When the signal *ERROR* is 0, each push-button keystroke provides only one pulse as it does for the design in Listing 13.7 and, therefore, does not skip through the sequence.
- Because the output of a flip-flop that is synchronized with a clock does not create glitches, the flip-flop output for the signal *F_S_CLKD* (which represents *F_S_CLK* delayed) is used to drive the clock inputs of the flip-flops for *Q*1, *Q*2, and *Q*3 and the clock input of module 2 so that a gated clock warning is not issued. In other words, if *F_S_CLK* were connected to the clock inputs of *Q*1, *Q*2, and *Q*3 and the clock input of module 2, then a gated clock warning would be issued for net *F_S_CLK*.
- The signal COUNT(0) (which represents half the frequency of *CLK* or *CLK* divided by 2) is required in the circuit, because *CLK* cannot be used to delay itself via a flip-flop, due to the setup time required by the flip-flop—that is, for *F_S_CLKD*.

Listing 13.8 shows a complete VHDL design for the modified press-to-increment circuit in Figure 13.11.

LISTING 13.8
Complete VHDL design for the PTIC modified (project: PTIC_Modified)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity PTIC_MOD is port (
     rst,clk,sel_addr : in std_logic;
     seg : out std_logic_vector (7 downto 0);
     en_disp : out std_logic;
```

```
    en_disp1 : out std_logic;  --added to turn off Display1
    en_disp2 : out std_logic;  --added to turn off Display2
    en_disp3 : out std_logic;  --added to turn off Display3
    error : in std_logic       --added to select either count(0)
                               --or slow_clk
    );
end PTIC_MOD;
architecture Mixed of PTIC_MOD is
--Module 1 internal signals, count, slow_clk, q1, q2, q3,f_s_clk
    signal count : std_logic_vector (18 downto 0);
    signal slow_clk, q1, q2, q3, f_s_clk : std_logic;
--Modules 1 and 2 internal signals, one_pulse, f_s_clkd
    signal one_pulse, f_s_clkd : std_logic;
--Module 2 internal signal, q
    signal q : std_logic_vector (3 downto 0);
--Module 2 and 3 internal signal, pc_addr
    signal pc_addr : std_logic_vector (3 downto 0);
--Module 3 and 4 internal signal, d7s
    signal d7s : std_logic_vector (7 downto 0);
--Module 3 and 5 internal signal, a
    signal a : std_logic;
begin
--Module 1 code, Debounced One_Pulse Circuit
    count <= count + 1 when rising_edge (clk) else
             count;
    slow_clk <= count (18);
    f_s_clk <= slow_clk when error = '0' else
               count(0);
    f_s_clkd <= f_s_clk when rising_edge (clk) else
                f_s_clkd;
process (f_s_clkd)
begin
    if rising_edge (f_s_clkd) then q1 <= sel_addr; q2 <= q1;
                                   q3 <= q2;
    end if;
end process;
    one_pulse <= q1 and q2 and not q3;
--Module 2 code, Loading Program Counter
process (rst, f_s_clkd)
begin
    if rst = '1' then q <= "0000";
    elsif rising_edge(f_s_clkd) and one_pulse = '1' then q <= q + 1;
    end if;
end process;
    pc_addr <= q;
--Module 3 code, HEX Display Decoder
    d7s <= "00111111" when pc_addr = "0000" else
```

(Continued)

```
            "00000110" when pc_addr = "0001" else
            "01011011" when pc_addr = "0010" else
            "01001111" when pc_addr = "0011" else
            "01100110" when pc_addr = "0100" else
            "01101101" when pc_addr = "0101" else
            "01111101" when pc_addr = "0110" else
            "00000111" when pc_addr = "0111" else
            "01111111" when pc_addr = "1000" else
            "01101111" when pc_addr = "1001" else
            "01110111" when pc_addr = "1010" else
            "01111100" when pc_addr = "1011" else
            "00111001" when pc_addr = "1100" else
            "01011110" when pc_addr = "1101" else
            "01111001" when pc_addr = "1110" else
            "01110001";
    a <= '1';
--Module 4 code, inverter
    seg <= not d7s;
--Module 5 code, inverter
    en_disp <= not a;
--Added to turn off Displays 1, 2, and 3
    en_disp1 <= '1';--added to turn off Display1
    en_disp2 <= '1';--added to turn off Display2
    en_disp3 <= '1';--added to turn off Display3
end Mixed;
```

Things you should notice about the VHDL design in Listing 13.8:

- The signal *ERROR* is added to the entity of the design.
- The signal *F_S_CLKD* is added as an internal signal between modules 1 and 2.
- In the code for module 1, a steering circuit (a 2-to-1 MUX) is added to steer the signal *SLOW_CLK* to *F_S_CLK* when *ERROR* is 0 or steer the signal *COUNT*(0) to *F_S_CLK* when *ERROR* is 1.
- In the code for module 1, the signal *SLOW_CLK* is changed to the signal *F_S_CLKD* in the process.
- In the code for module 1, flip-flop *F_S_CLKD* is added to prevent a gated clock.
- In the code for module 2, the signal *SLOW_CLK* is changed to the signal *F_S_CLKD* in the process.

See if you can discover an alternate way to remove the gated clock warning for the VHDL code in Listing 13.8 without using flip-flop *F_S_CLKD*. [Hint: Use D flip-flops with clock enable inputs (CE inputs)].

Clock skew is the delay of the clock signal to a flip-flop from the clock source such that all flip-flops are not triggered at the same time, as they should be. This can cause problems for high-speed digital circuits. For digital circuits that operate at a slower speed, clock skew is generally not a problem. VHDL compilers generate an error message for a clock net that may have excessive clock skew. If loads on the net are OK for your design and the timing constrain requirement on the net is met, then the clock skew warning can be safely ignored. In the design for Listing 13.7, an excessive clock skew warning is issued for *CLK* net: *COUNT*(18), indicating that certain *CLK* pins failed to route using a CLK template. This is simply a warning and can be safely ignored.

PROBLEMS

Section 13.2 Designing an Instruction Memory

13.1 What is the name of the unit that contains the instructions that VBC1 can execute?

13.2 How many instructions can be loaded into the instruction memory of VBC1, and how many bits are used in each instruction?

13.3 How many address bits are contained in the instruction memory of VBC1? How many locations does this number of bits have access to in the instruction memory?

13.4 Explain how a single instruction is loaded into instruction memory.

13.5 What is instruction memory called or referred to as, and what is the size of the instruction memory for VBC1?

13.6 Write the VHDL code for an enumerated data type for an instruction memory of 32 by 16.

13.7 Write a VHDL signal declaration for the signal called *NEW_MEM* of type mem_type.

13.8 What would happen if you included the Boolean equation *IR* <= *MEM*(conv_integer(*PC_ADDR*)); inside the process in Section 13.2, Listing 13.1 and did not include *MEM* in the sensitivity list for the process?

13.9 Write the VHDL code for an instruction memory of 16 by 8 to replace the code in Section 13.2, Listing 13.1, using a conditional signal assignment for *MEM* and a Boolean equation for *IR*.

13.10 Show a template for initializing instruction memory for VBC1 at startup.

13.11 Is it necessary to initialize the instruction memory? If it is not necessary, show how to write the VHDL code for instruction memory for VBC1 that is not initialized.

13.12 Can you name some advantages of initializing the instruction memory?

Section 13.3 Designing a Loading Program Counter

13.13 How many program counters does VBC1 have, and what are their names? Discuss what each program counter does.

13.14 How many states does a modulo 5 counter have? How many states does a modulo j counter have?

13.15 Is a program counter a binary-up counter or a binary-down counter? Provide a brief explanation to support your answer in terms of the VBC1 architecture.

13.16 Is the loading program counter for VBC1 a simple state machine or is it a complex state machine? Provide an explanation for your answer.

13.17 What is the name of the signal that drives the control or clock input of the loading program counter for VBC1?

13.18 The arithmetic method is one of the easiest ways to design a simple program counter. Write a conditional signal assignment for output *Q* of the LPC in Section 13.3, Listing 13.5, using the arithmetic method.

13.19 In the design in Section 13.3, Listing 13.5, would the design be correct if *PC_ADDR* <= *Q* were included inside the process—that is, before **end process;**,— without any other changes to the VHDL code? Provide a solution if one is needed.

Section 13.4 Designing a Debounced One-Pulse Circuit

13.20 What problem can occur when a push-button switch is pressed to generate a single pulse in a circuit?

13.21 In a circuit that uses a single push-button switch to generate a single output pulse, what circuit is used to help prevent multiple output pulses from being generated?

13.22 For the debounced one-pulse circuit in Section 13.4, Figure 13.6, and the register transfer level circuit in Section 13.4, Figure 13.7, what is the restriction on the duration of contact bounce to help prevent more than a single output pulse from occurring when a push-button switch is pressed that drives the signal *SEL_ADDR*?

13.23 In Section 13.4, Listing 13.6, use three conditional signal assignments to write the VHDL code for flip-flop outputs *Q*1 through *Q*3.

13.24 In Section 13.4, Listing 13.6, use a single conditional signal assignment to write the VHDL code for flip-flop outputs *Q*1 through *Q*3. Assume the declaration **signal** *Q*1, *Q*2, *Q*3 : std_logic; is replaced by the declaration **signal** *Q* : std_logic_vector (1 to 3);. Also make the required changes to the Boolean equation *ONE_PULSE*.

13.25 Show another way to represent rising_edge (*CLK*) using the clock tick event.

13.26 Show another way to represent falling_edge (*CLK*) using the clock tick event.

Section 13.5 Design Verification for a Debounced One-Pulse Circuit

13.27 What would be the duration for T_{CLK} in Section 13.5, Figure 13.8, if a frequency divider is not used? Do you think that contact bounce would occur?

13.28 In Section 13.5, Figure 13.8 when a frequency divider is used to divide the frequency of *CLK* by 2^{19}, what is the duration for T_{SLOW_CLK} to help prevent contact bounce from causing unintentional output pulses?

13.29 What is a gated clock? What should you do if you get a gated clock warning in your design?

13.30 What is the purpose of modules 4 and 5 in the VHDL code in Section 13.5, Listing 13.7?

13.31 What is the purpose of the signals *EN_DISP*1, *EN_DISP*2, and *EN_DISP*3 in the VHDL code in Section 13.5, Listing 13.7? How can the same thing be accomplished differently?

13.32 Does the ISE software provide any indication that your VHDL code may contain a gated clock? If so, what is the warning that is issued?

13.33 Draw the bad gated clock circuit that is shown in the book. Why is the use of a gated clock considered a bad design practice?

13.34 What is a glitch, as defined in the book?

13.35 Draw the good circuit that is shown in the book for controlling a clock signal to a counter or D flip-flop via an *EN* signal that does not produce a glitch. If the recommended circuit is used in a VHDL design, is a warning issued by the ISE software indicating that the circuit contains a gated clock?

13.36 What is the purpose of the modification to the press-to-increment circuit shown in Figure 13.11?

13.37 What is the value for the signal *ERROR* in Figure 13.11 that causes the circuit to skip through the binary-up counting sequence?

13.38 What is the value for the signal *ERROR* in Figure 13.11 that allows each push-button keystroke to provide only one pulse?

13.39 Does the output of a flip-flop that is synchronized with a clock create glitches? What flip-flop is used in Figure 13.11 so that a gated clock warning is not issued?

13.40 Explain why the signal *COUNT*(0) must be supplied to MUX_E in Figure 13.11 and not the signal *CLK*. Illustrate this principle by drawing a waveform diagram showing *CLK* as input 1 of MUX_E. When signal *ERROR* is 1, show the waveform diagram for *F_S_CLK* and the output waveform diagram for *F_S_CLKD*. Now draw another waveform diagram showing *COUNT*(0) as input 1 of MUX_E. When signal *ERROR* is 1, also show the waveform diagrams for *CLK*, *F_S_CLK*, and *F_S_CLKD*.

13.41 Provide an alternate way to remove the gated clock warning for the VHDL code in Listing 13.8 without using flip-flop *F_S_CLKD*. Draw a circuit that has the same functionality as the circuit in Figure 13.11. Use the circuit symbol shown in Figure P13.41 for the debounced one-pulse circuit. (Hint: Use two DOPCs and two LPCs. Steer the signals out of the LPCs to the HEX display decoder.

13.42 Obtain the VHDL code for your circuit diagram in problem 13.41 and implement the design using a BASYS 2 board or a NEXYS 2 board to ensure that it works the same as the circuit in Figure 13.11.

13.43 What is clock skew? Can clock skew be ignored for digital circuits that operate at a high speed? Can clock skew be ignored for digital circuits that operate at a slower speed?

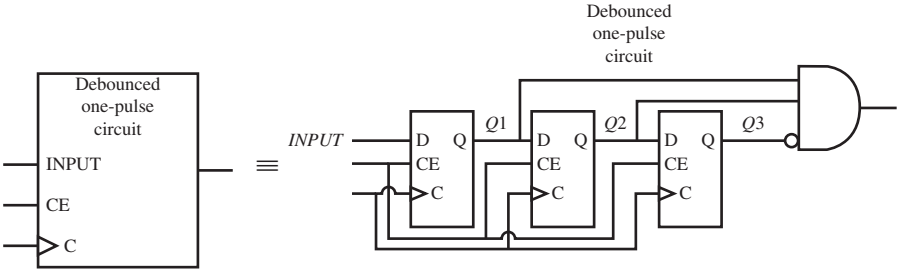


FIGURE P13.41

Chapter 14

Designing Multiplexed Display Systems

Chapter Outline

14.1 INTRODUCTION

In this chapter you will learn how to write VHDL for a multiplexed display system. Each module in the circuit will be designed separately. The modules will first be combined to form a **flat design approach** and then combined to form a **hierarchal design approach**. Documentation style M (introduced in Chapter 12) will continue to be stressed as we combine our VHDL source code to include additional modules. The chapter ends with the design of a word display system using a flat design approach for displaying words on four 7-segment LED displays.

14.2 MULTIPLEXED DISPLAY SYSTEM FOR FOUR 7-SEGMENT LED DISPLAYS

We will design a multiplexed display system using four 7-segment LED displays as shown in Figure 14.1. The displays are called Disp 3 through Disp 0 (short for display 3 through display 0) and are used to simultaneously display four different bus signals almost at the same time. This is the purpose of a multiplexed display system. The bus signals are the input signals *DISPLAY_3*, *DISPLAY_2*, *DISPLAY_1*, and *DISPLAY_0*. The purpose of the 4-to-1 MUX array is to steer or route each one of the input bus signals to its output, one at a time (called time-division multiplexing). Because the input bus signals in Figure 14.1 are fixed at the values F, A, 9, and 0, the values that appear on the four 7-segment LED displays will be F, A, 9, and 0, respectively, reading left to right. The values are not actually displayed at the same time, but they appear to be displayed at the same time when the frequency of *SLOW_CLK* is correctly chosen due to

the persistence of vision of human eyes. The purpose of the frequency divider is to reduce the frequency of the system clock *CLK* to a proper frequency—that is, *SLOW_CLK*. The frequency of *SLOW_CLK* must be chosen to allow the individual displays to turn on and off fast enough to remove **blinking** but slow enough to remove **bleeding**, which causes the displayed symbols to be overlaid and thus appear indistinguishable.

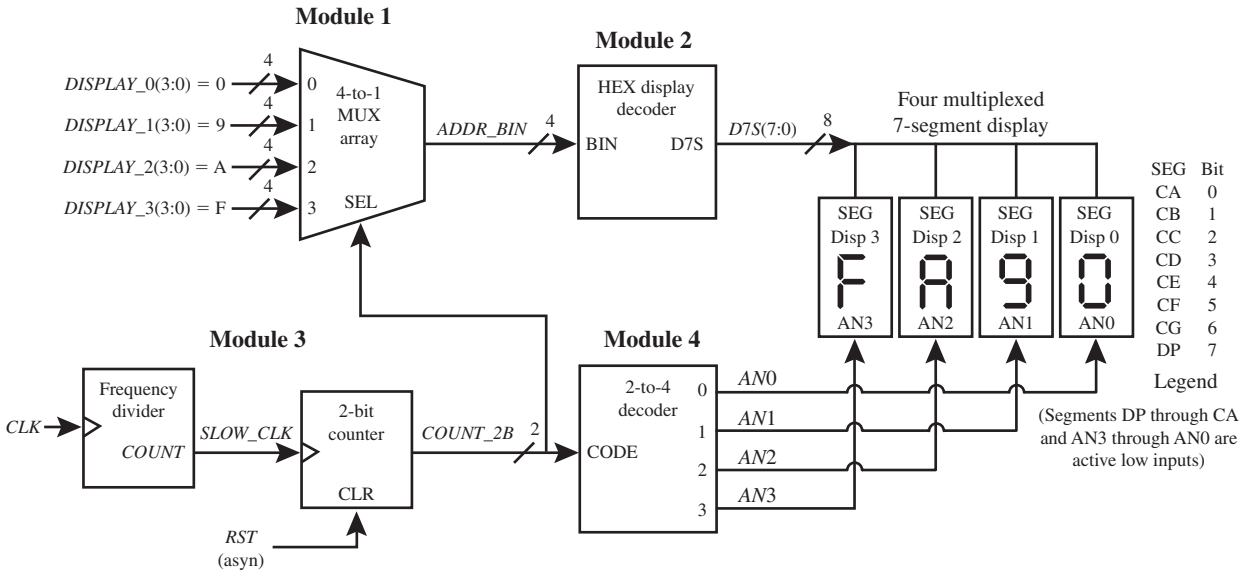


FIGURE 14.1 Multiplexed display system for displaying F, A, 9, 0 on four 7-segment LED displays display 3 through display 0

The operation of the circuit is fairly straightforward. A signal *SLOW_CLK* drives a 2-bit counter that continuously cycles through its four states 0 through 3. When the 2-bit counter is in state 0, the 4-to-1 MUX array guides the signal *DISPLAY_0* to its output. The HEX display decoder decodes the value of *DISPLAY_0*, a **fixed signal** with the **constant value** of 0. The signal D7S (display 7-segment) supplies the 7-segment decoded value 0 to all four of the 7-segment displays—display 3 through 0.

The display that lights is selected by the 2-to-4 decoder. The 2-to-4 decoder provides the signal *AN*0 to light display 0 when the 2-bit counter is in state 0 and also provides the signals *AN*1 through *AN*3 to turn off displays 1 through 3. When the 2-bit counter changes to state 1, then the signal *AN*1 is provided to light display 1 and 9 is lighted (all the other displays are turned off). When the 2-bit counter changes to state 2, then the signal *AN*2 is provided to light display 2 and A is lighted (all the other displays are turned off), and so on. The purpose of the 2-bit counter is to supply a binary-up counting signal, *COUNT_2B*, to the select input (SEL) of the 4-to-1 MUX array and to the code input (CODE) of the 2-to-4 decoder.

To make a blinking display appear as though it is not blinking requires the display to be turned on and off at a rate of approximately 24 cycles per second or faster. If the rate is slower than this, you will see the display blinking. For n displays sequentially blinking on and off in sequence (as a result of the 2-to-4 Decoder), each display must be turned on and off at a rate of approximately 24 cycles per second or faster. In the circuit in Figure 14.1, the 2-to-4 decoder must be driven by a counter with a minimum frequency of n times 24 cycles per second, or $n \times 24$ Hz $= 4 \times 24$ Hz $= 96$ Hz, or 96 cycles per second or faster to observe no blinking.

In Figure 14.1, the frequency of *SLOW_CLK* must be approximately 96 cycles per second or faster for displays 3 through 0 to appear to be on, showing the characters FA90. A frequency for *SLOW_CLK* below 96 cycles per second will result in each display alternating blinking in

the sequence 0, 9, A, followed by F and then back to 0, and so on. If you elect to implement the circuit on a hardware board, it is interesting to change the frequency of *SLOW_CLK* to observe the multiplexing operation working at a frequency of approximately 96 Hz or faster and also at a frequency much slower than 96 Hz. At the minimum frequency of 96 Hz, flicker will probably occur when you move your head or move the display slightly while observing the multiplexed display. To prevent flicker, simply increase the frequency until no flicker is observed.

A four multiplexed 7-segment display is a standard item on the Digilent BASYS 2 and NEXYS 2 boards as illustrated in Figure 14.2a. Figure 14.2b shows four separate 7-segment displays like those found on the Altera DE1 and DE2 boards. Figure 14.2c shows a design for a four multiplexed 7-segment display using 7 × 4, or 28 three-state buffers and four separate 7-segment displays.

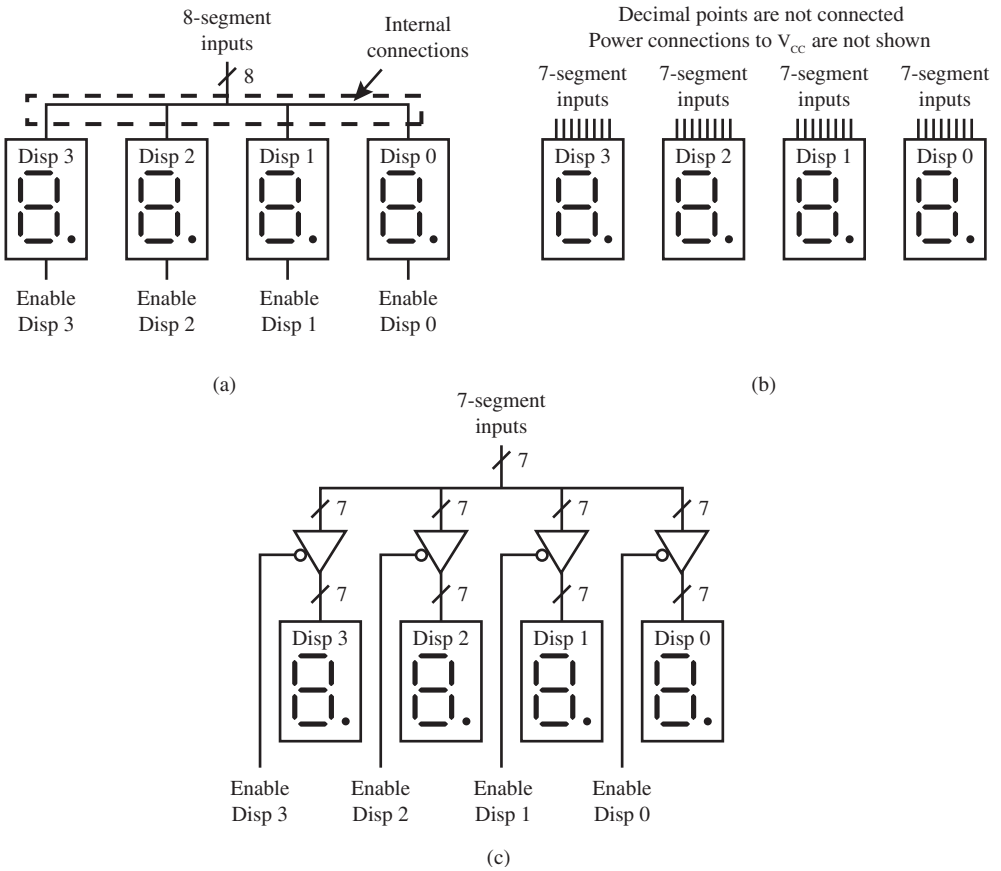


FIGURE 14.2 (a) A four multiplexed 7-segment display; (b) four separate 7-segment displays; (c) a design for a four multiplexed 7-segment display using 28 three-state buffers and four separate 7-segment displays

Multiplexed displays are often used to minimize the number of wires or conductors on a PC board. Observe in Figure 14.2a that only 8 + 4 or 12 wires are required to access the four multiplexed 7-segment display. The four separate 7-segment displays in Figure 14.2b require 7 × 4 or 28 wires to access the displays. When the four separate 7-segment displays are connected as shown in Figure 14.2c, only 7 + 4 or 11 wires are required to access the four multiplexed 7-segment display. Reminder: The segment inputs for both the Digilent boards and the Altera boards are active low inputs. The enable inputs for the Digilent multiplexed 7-segment display in Figure 14.2a are also active low inputs. The enable inputs for the multiplexed 7-segment display in Figure 14.2c are also active low inputs. The design in Figure 14.2c can be substituted for the four multiplexed 7-segment display in Figure 14.2a minus the ability to light the decimal points because the decimal points are not connected.

14.3 DESIGNING A MULTIPLEXED DISPLAY SYSTEM USING VHDL

In this section, we will design each of modules 1 through 4 for the multiplexed display system in Figure 14.1 using either a dataflow or a behavioral design style. In the next section, we combine the four modules together in one file using a flat design approach.

14.3.1 Designing Module 1: A 4-to-1 MUX Array

Listing 14.1 shows a complete VHDL design for module 1 using a behavioral design style with a case statement.

LISTING 14.1
Complete VHDL design for module 1 using a behavioral design style with a case statement (project: MUX_Array)

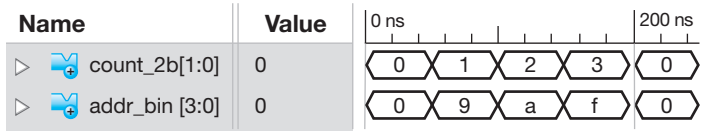
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX_Array is port (
     count_2b : in std_logic_vector (1 downto 0);
     addr_bin : out std_logic_vector (3 downto 0)
     );
end MUX_Array;

architecture Behavioral of MUX_Array is
begin
process (count_2b)
begin
     case count_2b is
          when "00" => addr_bin <= "0000";
          when "01" => addr_bin <= "1001";
          when "10" => addr_bin <= "1010";
          when "11" => addr_bin <= "1111";
          when others => null;
     end case;
end process;
end Behavioral;
```

Things you should notice about the VHDL design in Listing 14.1:

- Signals *DISPLAY*_0 through *DISPLAY*_3 all have a constant value and must not be placed in the entity declaration or in the sensitivity list for the process.
- **When others** is required in the case statement and **null** may be used to indicate a do nothing action.

Waveform 14.1 shows a simulation with the correct functionality of design entity MUX_ARRAY.

WAVEFORM 14.1 Simulation for the correct functionality of design entity MUX_ARRAY



Things you should notice about the waveforms in Waveform 14.1:

- The values for the signal *COUNT_2B* are displayed in Waveform 14.1 as unsigned decimal values.
- The values for the signal *ADDR_BIN* are displayed in Waveform 14.1 as hexadecimal values.

14.3.2 Designing Module 2: A HEX Display Decoder

Listing 14.2 shows a complete VHDL design for module 2 using a dataflow design style with a conditional signal assignment.

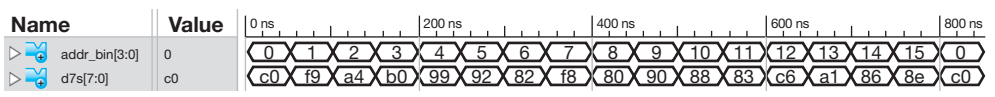
LISTING 14.2
Complete VHDL design for module 2 using a dataflow design style with a conditional signal assignment (project: HEX_Display_Decoder)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity HEX_Display_Decoder is port (
     addr_bin : in std_logic_vector (3 downto 0);
     d7s : out std_logic_vector (7 downto 0)
     );
end HEX_Display_Decoder;

architecture Dataflow of HEX_Display_Decoder is
begin
d7s <= "11000000" when addr_bin = "0000" else
        "11111001" when addr_bin = "0001" else
        "10100100" when addr_bin = "0010" else
        "10110000" when addr_bin = "0011" else
        "10011001" when addr_bin = "0100" else
        "10010010" when addr_bin = "0101" else
        "10000010" when addr_bin = "0110" else
        "11111000" when addr_bin = "0111" else
        "10000000" when addr_bin = "1000" else
        "10010000" when addr_bin = "1001" else
        "10001000" when addr_bin = "1010" else
        "10000011" when addr_bin = "1011" else
        "11000110" when addr_bin = "1100" else
        "10100001" when addr_bin = "1101" else
        "10000110" when addr_bin = "1110" else
        "10001110";
end Dataflow;
```

Things you should notice about the VHDL design in Listing 14.2:

- The signal *D7S* is a std_logic_vector (7 downto 0). Bit 7 is the decimal point DP.
- All the bits for the signal *D7S* are entered as active low—that is, logic 0 turns a segment on and logic 1 turns a segment off.
- The decimal point DP is disabled by setting *D7S*(7) to logic 1.
- The signal *ADDR_BIN* is represented as a 4-bit bus, and the relational operator "=" is required rather than the assignment operator "<=". An assignment operator "<=" is often used by accident.
- **When others** is not allowed in a conditional signal assignment statement. Using **when others** is a common mistake made when writing conditional signal assignment statements.

Waveform 14.2 shows a simulation with the correct functionality of design entity HEX_Display_Decoder.



WAVEFORM 14.2 Simulation for the correct functionality of design entity HEX_Display_Decoder

Things you should notice about the waveforms in Waveform 14.2:

- The values for the signal *ADDR_BIN* are displayed in Waveform 14.2 as unsigned decimal values.
- The values for the signal *D7S* are displayed in Waveform 14.2 as hexadecimal values.

14.3.3 Designing Module 3: A 2-bit Counter and a Frequency Divider

Listing 14.3 shows a partial VHDL design for module 3 for a 2-bit counter with a behavioral design style with an if statement—that is, using the arithmetic method with a "+" arithmetic operator. This design does not include the frequency divider and thus allows us to verify the functionality of the 2-bit counter by running a simulation using *CLK*—that is, the system clock frequency. A simulation obtained with a frequency divider takes more clock cycles because of the frequency division thus making the simulation harder to obtain and analyze. We will add the frequency divider circuit after we verify that the 2-bit counter works as expected.

LISTING 14.3 Partial VHDL design for module 3 just for the 2-bit counter with a behavioral design style with an arithmetic "+" operator (project: Counter)

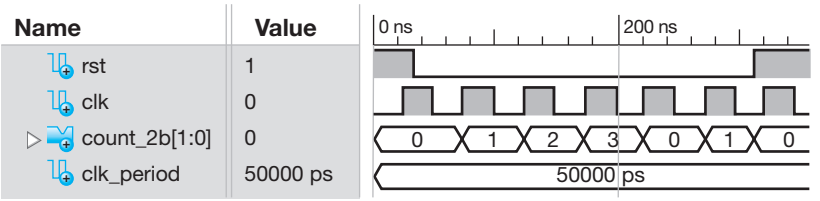
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter is port (
    rst, clk : in std_logic; --clk is the system clock signal for
                             --the 2-bit counter
    count_2b : inout std_logic_vector (1 downto 0)
    );
end Counter;

architecture behavioral of Counter is
begin
process (rst, clk)
begin
    if rst = '1' then count_2b <= "00";
    elsif rising_edge (clk) then count_2b <= count_2b + 1;
    end if;
end process;
end behavioral;
```

Things you should notice about the VHDL design in Listing 14.3:

- Because the "+" arithmetic operator is used in this design, the package "IEEE.STD_LOGIC_UNSIGNED" is required to define the "+" arithmetic operator. Leaving out the use clause statement "**use** IEEE.STD_LOGIC_UNSIGNED.**ALL**;" is a common error.
- The signal *RST* (*RESET*) must be included in the sensitivity list of the design for the counter process. It is a common error to leave out signals in the sensitivity list that must be read to perform the process.

Waveform 14.3 shows a simulation with the correct functionality of design entity Counter. This simulation does not include the frequency divider.

WAVEFORM 14.3 Simulation for the correct functionality of design entity Counter



Things you should notice about the waveforms in Waveform 14.3:

- The signal *RST* is an asynchronous signal that overrides the signal *CLK*.
- The values for the signal *COUNT_2*b are displayed in Waveform 14.3 as unsigned decimal values.

Listing 14.4 shows a complete VHDL design for module 3. This design includes a frequency divider with an internal signal *SLOW_CLK* of ~1Hz for *CLK* = 50 MHz.

LISTING 14.4
Complete VHDL design for module 3 using a behavioral design style with an arithmetic "+" operator (project: Module3)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Module3 is port (
     rst,clk : in std_logic;
     count_2b : inout std_logic_vector (1 downto 0)
     );
end Module3;
architecture behavioral of Module3 is

--Module 3 internal signals, count, slow_clk
     signal count: std_logic_vector (24 downto 0) ;
     signal slow_clk: std_logic;
begin
--Module 3 code, Frequency Divider
process (clk, count)
begin
     if rising_edge (clk) then count <= count + 1;
     end if;
--the frequency of slow_clk is 1.49 Hz for clk = 50 MHz (50 M/2^25
--Hz = 1.49 Hz)
     slow_clk <= count (24);
end process;
--Module 3 code, 2-bit Counter
process (rst, slow_clk)
begin
     if rst = '1' then count_2b <= "00";
     elsif rising_edge (slow_clk) then count_2b <= count_2b + 1;
     end if;
end process;
end behavioral;
```

Things you should notice about the VHDL design in Listing 14.4:

- The signals *COUNT* and *SLOW_CLK* are internal signals and must be placed between **architecture** and the first **begin**. One common error is to list internal signals in the entity. Another common error is to include a mode—that is, **in**, **out**, **inout**, or **buffer**—for internal signals. Do not provide a mode for an internal signal.
- Documentation style M marks the location for each internal signal (*COUNT* and *SLOW_CLK*) and the location for each section of code for the circuit elements (the frequency divider and the 2-bit counter) in the VHDL design for each module so that you can easily find them.

14.3.4 Designing Module 4: A 2-to-4 Decoder

Listing 14.5 shows a complete VHDL design for module 4 using a behavioral design style with a case statement.

LISTING 14.5
Complete VHDL design for module 4 using a behavioral design style with a case statement (project: Decoder)

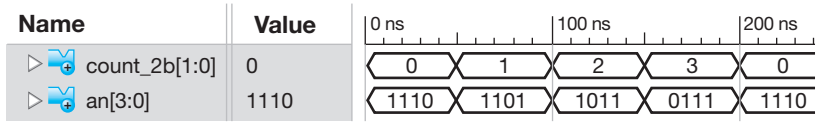
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Decoder is port (
     count_2b : in std_logic_vector (1 downto 0);
     AN : out std_logic_vector (3 downto 0)
     );
end Decoder;

architecture Behavioral of Decoder is
begin
process (count_2b)
begin
     case count_2b is
          when "00" => AN <= "1110"; --Light only Display 0
          when "01" => AN <= "1101"; --Light only Display 1
          when "10" => AN <= "1011"; --Light only Display 2
          when "11" => AN <= "0111"; --Light only Display 3
          when others => null;
     end case;
end process;
end Behavioral;
```

Things you should notice about the VHDL design in Listing 14.5:

- The signal values for *AN* are shown as active low signals, where 0 turns a display on and 1 turns a display off.
- Notice that the keyword **is** follows the entity name in the architecture and also follows the signal name after case. It is a common error to leave off the keyword **is** in either one of these places.

Waveform 14.4 shows a simulation with the correct functionality of design entity Decoder.

WAVEFORM 14.4 Simulation for the correct functionality of design entity Decoder



Things you should notice about the waveforms in Waveform 14.4:

- The values for the signal *COUNT_2B* are displayed in Waveform 14.4 as unsigned decimal values.
- The values for the signal *AN* are displayed in Waveform 14.4 as 4-bit binary values. The signal values for *AN*(3:0) are shown as active low signals. Active low signals for *AN* are required to drive the four 7-segment LED displays on a BASYS 2 board and a NEXYS 2 board.

14.4 COMPLETE DESIGN OF A MULTIPLEXED DISPLAY SYSTEM USING A FLAT DESIGN APPROACH

Listing 14.6 shows a complete VHDL design for the multiplexed display system in Figure 14.1 using a flat design approach. In this design, each of the separate modules is placed within a

single architecture. Documentation style M is used to mark the location for each internal signal and the location for each section of code for the circuit elements in the VHDL design for each module. Marking these locations makes it easier to better understand large VHDL designs.

LISTING 14.6
Complete VHDL design for a multiplexed display system using a flat design approach (project: Multi_Display_System)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use clause required for arithmetic operator "+"
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Multi_Display_System is port (
     rst, clk : in std_logic;
     d7s : out std_logic_vector(7 downto 0);
     AN : out std_logic_vector (3 downto 0)
     );
end Multi_Display_System;

architecture Mixed of Multi_Display_System is

--Modules 1 and 2 internal signal, addr_bin
     signal addr_bin : std_logic_vector (3 downto 0);

--Modules 1, 3, and 4 internal signal, count_2b
     signal count_2b : std_logic_vector (1 downto 0);

--Module 3 internal signals, count, slow_clk
     signal count: std_logic_vector (24 downto 0) ;
     signal slow_clk: std_logic;
begin

--Module 1 code, MUX_Array
process (count_2b)
begin
     case count_2b is
          when "00" => addr_bin <= "0000";
          when "01" => addr_bin <= "1001";
          when "10" => addr_bin <= "1010";
          when "11" => addr_bin <= "1111";
          when others => null;
     end case;
end process;

--Module 2 code, HEX Display Decoder
d7s <=  "11000000" when addr_bin = "0000" else
        "11111001" when addr_bin = "0001" else
        "10100100" when addr_bin = "0010" else
        "10110000" when addr_bin = "0011" else
        "10011001" when addr_bin = "0100" else
        "10010010" when addr_bin = "0101" else
        "10000010" when addr_bin = "0110" else
        "11111000" when addr_bin = "0111" else
        "10000000" when addr_bin = "1000" else
        "10010000" when addr_bin = "1001" else
        "10001000" when addr_bin = "1010" else
        "10000011" when addr_bin = "1011" else
        "11000110" when addr_bin = "1100" else
```

(Continued)

```
        "10100001" when addr_bin = "1101" else
        "10000110" when addr_bin = "1110" else
        "10001110";

--Module 3 code, Frequency Divider
process (clk, count)
begin
    if rising_edge (clk) then count <= count + 1;
    end if;
--the frequency of slow_clk is 1.49 Hz for clk = 50 MHz (50 M/2^25
--Hz = 1.49 Hz)
    slow_clk <= count (24); --display blinks
--the frequency of slow_clk is 762.94 Hz for clk = 50 MHz (50
--M/2^16 Hz = 762.94 Hz )
--   slow_clk <= count (15); --display without flicker
end process;

--Module 3 code, 2-bit Counter
process (rst, slow_clk)
begin
    if rst = '1' then count_2b <= "00";
    elsif rising_edge (slow_clk) then count_2b <= count_2b + 1;
    end if;
end process;

--Module 4 code, 2-to-4 Decoder
process (count_2b)
begin
    case count_2b is
        when "00" => AN <= "1110"; --Lights only Display 0
        when "01" => AN <= "1101"; --Lights only Display 1
        when "10" => AN <= "1011"; --Lights only Display 2
        when "11" => AN <= "0111"; --Lights only Display 3
        when others => null;
    end case;
end process;
end Mixed;
```

Things you should notice about the VHDL design in Listing 14.6:

- The individual modules or circuit elements that were designed earlier in this chapter for the multiplexed display system are combined in this design under a single entity and a single architecture.
- DSM is used to mark the location for each internal signal in the VHDL design for each module.
- DSM is also used to mark the location for each section of code for the circuit elements in the VHDL design for each module.

If you elect to use a hardware board, you will observe that each display blinks in the sequence 0, 9, A, followed by F and then back to 0, and so on, with *SLOW_CLK* = *COUNT*(24). To see all the characters F, A, 9, and 0 displayed at the same time without flicker, make *SLOW_CLK* = *COUNT*(15) so that *SLOW_CLK* has a frequency of 762.94 Hz (50 M/2^{16} Hz). At this

frequency, the human eye cannot tell that each display actually turns off and on, and we perceive that the total display has all the characters turned on at the same time, which is the desired result for a multiplexed display.

14.5 COMPLETE DESIGN OF A MULTIPLEXED DISPLAY SYSTEM USING A HIERARCHAL DESIGN APPROACH

Figure 14.3 shows the same schematic for the multiplexed display system shown earlier in Figure 14.1—with some small modifications. Since we use a hierarchal design approach or structural design style in this section, each module is represented in the structural design as a component—that is, module 1 is component 1, module 2 is component 2, and so on.

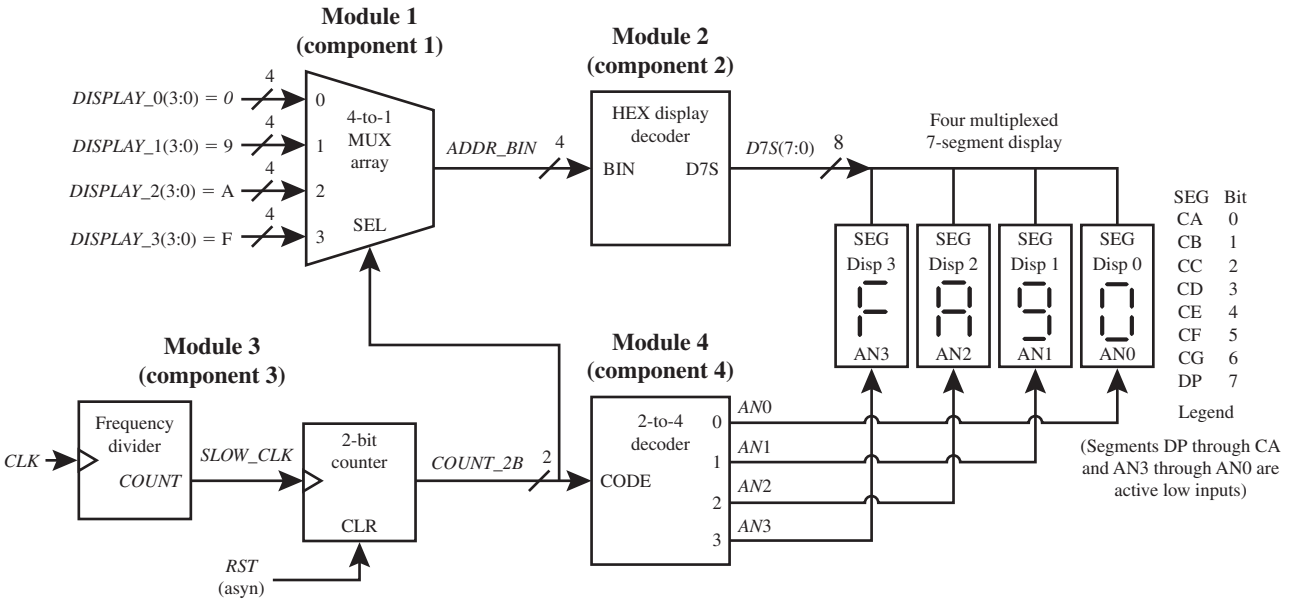


FIGURE 14.3 Multiplexed display system for displaying F, A, 9, 0 on four 7-segment LED displays 3 through 0

The formal signals for the components are shown in Figure 14.4 (on the next page). The formal signal names can be chosen to be any name desired (except a keyword). For simplicity, we elected to make the formal signal names the same as the actual signal names.

Listing 14.7 (on the next page) shows a design for the multiplexed display system in Figure 14.2 using a structural design style with components. The components are installed or **instantiated** and then interconnected using **port map statements** in a manner similar to wiring components via a schematic. The **formal signals** (the input and output signals for each component) are mapped to the **actual signals** (the port signals and the internal signals).

Things you should notice about the VHDL design in Listing 14.7:

- The individual components or circuit elements that were designed earlier in this chapter for the multiplexed display system are combined in this design using a structural design style. The separate designs are the component definitions in a structural design.
- Each component declaration is obtained from its corresponding entity declaration in the component definition section by doing the following: (1) simply copy the entity declaration and place it between **architecture** and the first **begin** in the top level of the structural design, and (2) change **entity** to **component** and change **end** <**name of entity**> to **end component**.

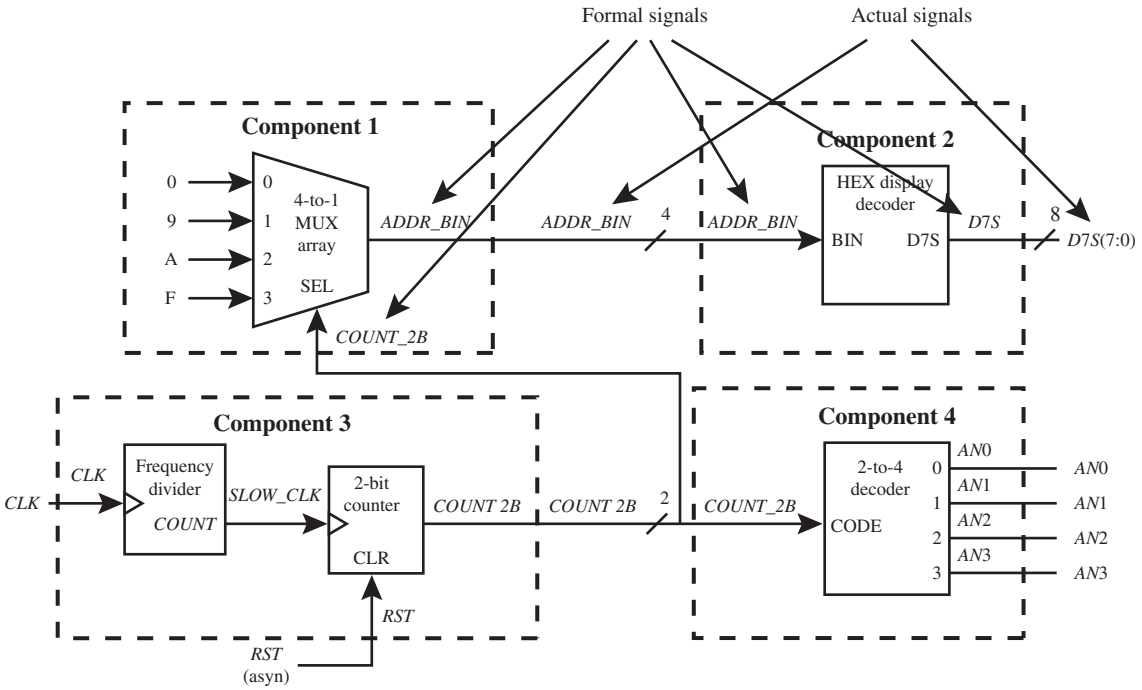


FIGURE 14.4 Formal signals for components versus actual signals

LISTING 14.7
Complete VHDL design for a multiplexed display system using a hierarchal design approach (project: Multi_Display_System_Stru)

```
--Component 1 definition, MUX_Array
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX_Array is port (
     count_2b : in std_logic_vector (1 downto 0);
     addr_bin : out std_logic_vector (3 downto 0)
     );
end MUX_Array;

architecture behavioral of MUX_Array is
begin
process (count_2b)
begin
     case count_2b is
          when "00" => addr_bin <= "0000";
          when "01" => addr_bin <= "1001";
          when "10" => addr_bin <= "1010";
          when "11" => addr_bin <= "1111";
          when others => null;
     end case;
end process;
end behavioral;

--Component 2 definition, HEX Display Decoder
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity HEX_Display_Decoder is port (
     addr_bin : in std_logic_vector(3 downto 0);
     d7s : out std_logic_vector(7 downto 0)
     );
end HEX_Display_Decoder;

architecture dataflow of HEX_Display_Decoder is
begin
d7s <=  "11000000" when addr_bin = "0000" else
        "11111001" when addr_bin = "0001" else
        "10100100" when addr_bin = "0010" else
        "10110000" when addr_bin = "0011" else
        "10011001" when addr_bin = "0100" else
        "10010010" when addr_bin = "0101" else
        "10000010" when addr_bin = "0110" else
        "11111000" when addr_bin = "0111" else
        "10000000" when addr_bin = "1000" else
        "10010000" when addr_bin = "1001" else
        "10001000" when addr_bin = "1010" else
        "10000011" when addr_bin = "1011" else
        "11000110" when addr_bin = "1100" else
        "10100001" when addr_bin = "1101" else
        "10000110" when addr_bin = "1110" else
        "10001110";
end dataflow;
--Component 3 definition, Counter
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter is port (
     clk : in std_logic;
     rst : in std_logic;
     count_2b : inout std_logic_vector (1 downto 0)
     );
end Counter;

architecture behavioral of Counter is
     signal count: std_logic_vector (24 downto 0) ;
     signal slow_clk: std_logic;
begin

--Component 3 code, Frequency Divider
process (clk, count)
begin
     if rising_edge (clk) then count <= count + 1;
     end if;
--the frequency of slow_clk is 1.49 Hz for clk = 50 MHz (50 M/2^25
--Hz = 1.49 Hz)
                                                        (Continued)
```

```
    slow_clk <= count (24); --display blinks
--the frequency of slow_clk is 762.94 Hz for clk = 50 MHz (50
--M/2^16 Hz = 762.94 Hz )
--   slow_clk <= count (15); --display without flicker
end process;

--Component 3 code, 2-bit Counter
process (rst, slow_clk)
begin
    if rst = '1' then count_2b <= "00";
    elsif rising_edge (slow_clk) then count_2b <= count_2b + 1;
    end if;
end process;
end behavioral;

--Component 4 definition, 2-to-4 Decoder
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Decoder is port (
    count_2b : in std_logic_vector (1 downto 0);
    AN : out std_logic_vector (3 downto 0)
    );
end Decoder;

architecture behavioral of Decoder is
begin
process (count_2b)
begin
    case count_2b is
        when "00" => AN <= "1110"; --Lights only Display 0
        when "01" => AN <= "1101"; --Lights only Display 1
        when "10" => AN <= "1011"; --Lights only Display 2
        when "11" => AN <= "0111"; --Lights only Display 3
        when others => null;
    end case;
end process;
end behavioral;

--Structural Design (top level)
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use clause required for arithmetic operator "+"
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Multi_Display_System_Stru is port (
    rst,clk : in std_logic;
    d7s : out std_logic_vector (7 downto 0);
    AN : out std_logic_vector (3 downto 0)
    );
```

```
end Multi_Display_System_Stru;

architecture Structural of Multi_Display_System_Stru is

--Components 1 and 2 internal signal, addr_bin
     signal addr_bin : std_logic_vector (3 downto 0);

--Components 1, 3, and 4 internal signal, count_2b
     signal count_2b : std_logic_vector (1 downto 0);

--Component 1 declaration, MUX_Array
component MUX_Array is port (
     count_2b : in std_logic_vector (1 downto 0);
     addr_bin : out std_logic_vector (3 downto 0)
     );
end component;

--Component 2 declaration, HEX Display Decoder
component HEX_Display_Decoder is port (
     addr_bin : in std_logic_vector(3 downto 0);
     d7s : out std_logic_vector(7 downto 0)
     );
end component;

--Component 3 declaration, Counter
component Counter is port (
     clk : in std_logic;
     rst : in std_logic;
     count_2b : inout std_logic_vector (1 downto 0)
     );
end component;

--Component 4 declaration, 2-to-4 Decoder
component Decoder is port (
     count_2b : in std_logic_vector (1 downto 0);
     AN : out std_logic_vector (3 downto 0)
     );
end component;
begin

--Component installation and connections (formally called component
--instantiations)
     C1: MUX_Array port map (count_2b => count_2b,
                             addr_bin => addr_bin);
     C2: HEX_Display_Decoder port map (addr_bin => addr_bin,
                                       d7s =>d7s);
     C3: Counter port map (clk => clk, rst => rst,
                           count_2b => count_2b);
     C4: Decoder port map (count_2b => count_2b, an => an);
--NOTE: it's OK to use the same names for the formal and actual
--signals as we have done here
end Structural;
```

- Documentation style M is used to mark the location for (1) each internal signal and (2) each component declaration that must be placed between **architecture** and the first **begin** in the VHDL design. Only internal signals that connect to other components need to be included in the top level of the structure design file, Multi_Display_System_Stru. The internal signals *SLOW_CLK* and *COUNT* that are only associated with component 3 do not have to be included in the top level of the structural design file.
- The actual signals are the external input signals and the internal signals. The same names are used for the formal signals and the actual signals as shown in the design. Even when the same names are used, the mapping for the formal signals and the actual signals must be shown via the port map.

A hierarchal design approach or structural design style is a form of **top-down** or **bottom-up design**. In a top-down or bottom-up design, there is a top level and sublevels. For large systems, a hierarchal design approach can be worked on by different engineers assigned to the same project. The design is partitioned so that each engineer is assigned a part or parts of the design. The sublevels of a hierarchal design approach can be designed and simulated independently to check their functionalities. A flat design approach does not have sublevels because the complete design is contained inside one architecture. Individual modules in a flat design should also be designed and simulated separately to check their functionalities prior to including them in the flat design.

As we mentioned earlier, a hierarchal design approach is basically a netlist description of a schematic. We believe that a flat design approach is easier to read and easier to understand, and it generally requires fewer lines of code than a netlist description of a schematic in VHDL. For these three reasons, we chose to use a flat design approach for our VHDL design of VBC1. Both the hierarchal design approach and the flat design approach work well and can be used to design any system in VHDL.

14.6 DESIGNING A WORD DISPLAY SYSTEM USING A FLAT DESIGN APPROACH

Figure 14.5 shows the schematic for a word display system that flashes words on four 7-segment LED displays at a specified frequency.

Things you should notice about the word display system in Figure 14.5:

- Modules 1 through 4 provide the words.
- Module 5 is the master MUX that drives the individual segments of the four 7-segment LED displays.
- Module 6 is a frequency divider that divides the input frequency *CLK* (50 MHz) by 2^{13} for *SLOW_CLK* and by 2^{26} for *VERY_SLOW_CLK*.
- Module 7 is a 2-bit counter that drives the select inputs of modules 1 through 4 and the code input of module 9 in order to scan a word across the four 7-segment LED displays.
- Module 8 is a second 2-bit Counter that drives the master MUX to change the word that is being displayed.
- The words that can be displayed on the four 7-segment LED displays in the circuit in Figure 14.5 are HI GUYS And GALS (note lowercase n and d). Each of the letters in a word are provided as a constant that consists of 8 bits, the bits necessary to form each required letter—that is, H, I, G, U, Y, S, A, n, d, and L. Bit 7 turns off the decimal point DP.
- The inputs to the display are all active low inputs.

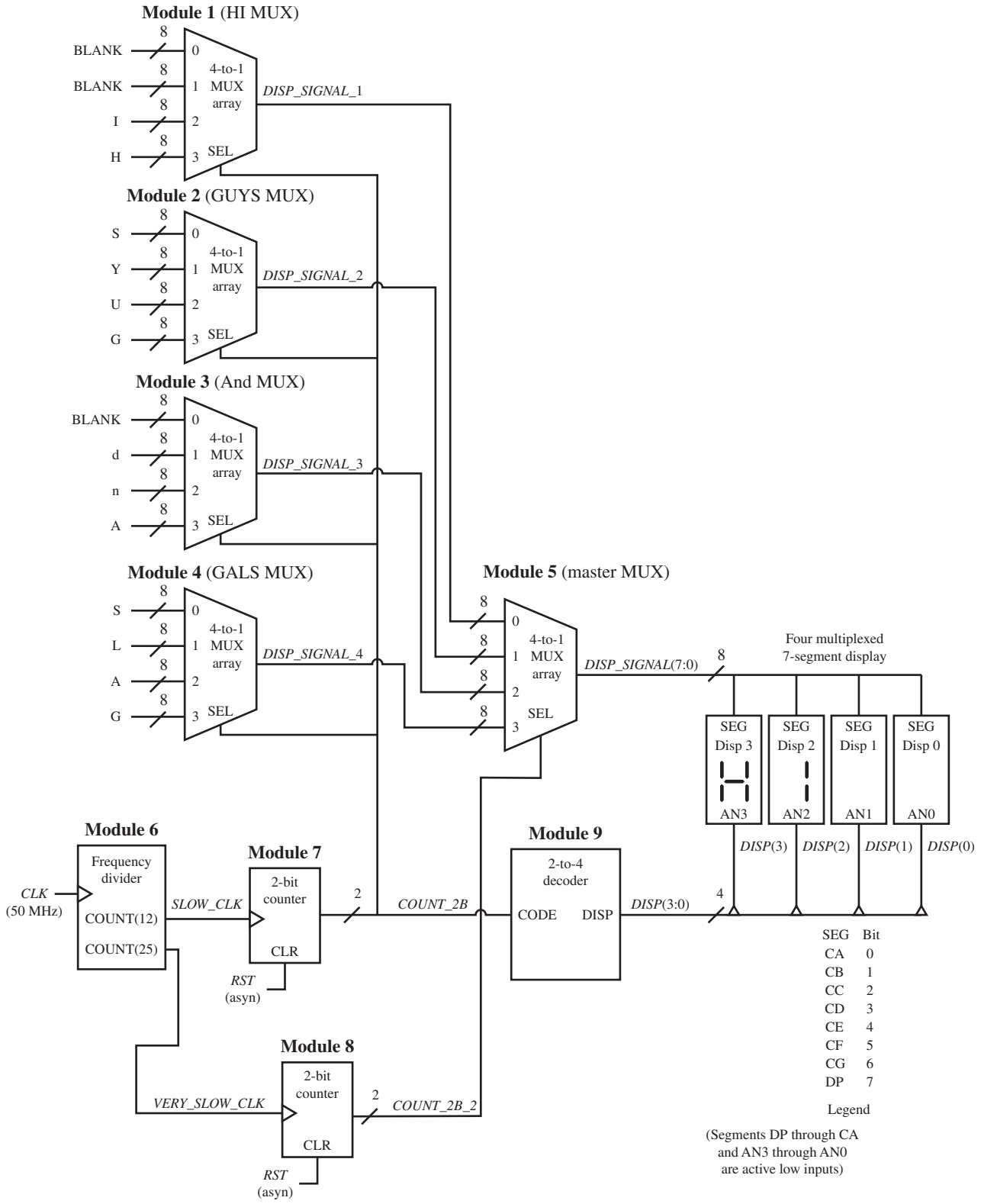


FIGURE 14.5 Schematic for a word display system

Listing 14.8 shows a complete VHDL design for the word display system using a flat design approach.

LISTING 14.8
Complete VHDL design for the word display system (project: HI_GUYS_And_GALS)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity HI_GUYS_And_GALS is port (
   rst,clk : in std_logic;
   Disp_signal: out std_logic_vector (7 downto 0);
   disp: out std_logic_vector (3 downto 0)
   );
end HI_GUYS_And_GALS;

architecture Mixed of HI_GUYS_And_GALS is

--Modules 1 and 5 internal signal, Disp_signal_1
   signal Disp_signal_1: std_logic_vector (7 downto 0);

--Modules 2 and 5 internal signal, Disp_signal_2
   signal Disp_signal_2: std_logic_vector (7 downto 0);

--Modules 3 and 5 internal signal, Disp_signal_3
   signal Disp_signal_3: std_logic_vector (7 downto 0);

--Modules 4 and 5 internal signal, Disp_signal_4
   signal Disp_signal_4: std_logic_vector (7 downto 0);

--Modules 6 and 7 internal signal, slow_clk
   signal slow_clk: std_logic;

--Modules 6 and 8 internal signal, very_slow_clk
   signal very_slow_clk: std_logic;

--Modules 7,1,2,3,4, and 9 internal signal, count_2b
   signal count_2b: std_logic_vector (1 downto 0);

--Modules 8 and 5 internal signal, count_2b_2
   signal count_2b_2: std_logic_vector (1 downto 0);

--Module 6 internal signal, count
   signal count: std_logic_vector (25 downto 0);
begin

--Module 1 code, HI MUX
process (count_2b)
begin
   case count_2b is                    --Drives active low inputs
      when "00" => Disp_signal_1 <= not "00000000"; --Blank
      when "01" => Disp_signal_1 <= not "00000000"; --Blank
      when "10" => Disp_signal_1 <= not "00000110"; --I
      when "11" => Disp_signal_1 <= not "01110110"; --H
      when others => null;
   end case;
end process;
```

```
--Module 2 code, GUYS MUX
process (count_2b)
begin
   case count_2b is                    --Drives active low inputs
      when "00" => Disp_signal_2 <= not "01101101"; --S
      when "01" => Disp_signal_2 <= not "01100110"; --Y
      when "10" => Disp_signal_2 <= not "00111110"; --U
      when "11" => Disp_signal_2 <= not "00111101"; --G
      when others => null;
   end case;
end process;

--Module 3 code, And MUX
process (count_2b)
begin
   case count_2b is                    --Drives active low inputs
      when "00" => Disp_signal_3 <= not "00000000"; --Blank
      when "01" => Disp_signal_3 <= not "01011110"; --d
      when "10" => Disp_signal_3 <= not "01010100"; --n
      when "11" => Disp_signal_3 <= not "01110111"; --A
      when others => null;
   end case;
end process;

--Module 4 code, GALS MUX
process (count_2b)
begin
   case count_2b is                    --Drives active low inputs
      when "00" => Disp_signal_4 <= not "01101101"; --S
      when "01" => Disp_signal_4 <= not "00111000"; --L
      when "10" => Disp_signal_4 <= not "01110111"; --A
      when "11" => Disp_signal_4 <= not "00111101"; --G
      when others => null;
   end case;
end process;

--Module 5 code, Master MUX
process (count_2b_2, Disp_signal_1, Disp_signal_2, Disp_signal_3,
         Disp_signal_4)
begin
   case count_2b_2 is
      when "00" => Disp_signal <= Disp_signal_1;
      when "01" => Disp_signal <= Disp_signal_2;
      when "10" => Disp_signal <= Disp_signal_3;
      when "11" => Disp_signal <= Disp_signal_4;
      when others => null;
   end case;
end process;

--Module 6 code, Frequency Divider
process (clk, count)
begin
   if rising_edge (clk) then count <= count + 1;
   end if;
   slow_clk <= count(12);
   very_slow_clk <= count(25);
end process;
```

(Continued)

```
--Module 7 code, 2-bit counter
process (rst, slow_clk)
begin
   if rst = '1' then count_2b <= "00";
   elsif rising_edge (slow_clk) then count_2b <= count_2b + 1;
   end if;
end process;

--Module 8 code, 2-bit counter
   count_2b_2 <= "00"           when rst = '1' else
                 count_2b_2 + 1 when rising_edge (very_slow_clk);

--Module 9 code, 2-to-4 Decoder
process (count_2b)
begin
   case count_2b is          --Drives active low inputs
      when "00" => disp <= not "0001"; --Turn on Display 0
      when "01" => disp <= not "0010"; --Turn on Display 1
      when "10" => disp <= not "0100"; --Turn on Display 2
      when "11" => disp <= not "1000"; --Turn on Display 3
      when others => null;
   end case;
end process;
end Mixed;
```

Things you should notice about the VHDL design in Listing 14.8:

- The VHDL design uses both dataflow and behavioral design styles and therefore is specified as a mixed design style.
- DSM is consistently followed to mark each internal signal and each section of code for the circuit elements.
- The inputs to the display are all active low inputs; however, the output signals that drive the display are written as active high signals and then complemented. Logic 1 turns a segment off while logic 0 turns a segment on. To turn all of the segments off—that is, blank the display—we wrote *DISP_SIGNAL*_1 <= **not** "00000000", which evaluates to *DISP_SIGNAL*_1 <= "11111111". The input "11111111" is the required signal for active low inputs to turn off all the segments.
- For the letter H, active high output signals are used and then complemented—that is, **not** "01110110". For the letter I, active high output signals are used and then complemented—that is, **not** "00000110". All the other letters are formed in the VHDL design in the same way. Bit 7 turns off the decimal point DP.
- A frequency divider is used to obtain the signals *SLOW_CLK* and *VERY_SLOW_CLK*.
- The internal signal *SLOW_CLK* is assigned the internal signal count(12), which provides a frequency of 50 MHz/2^{13} = 6,104 Hz for a system clock with a frequency of 50 MHz. The frequency of the internal signal *SLOW_CLK* determines the character scan rate of the display—that is, how fast the characters are turned on and off across the four 7-segment LED displays.
- The internal signal *VERY_SLOW_CLK* is assigned the internal signal count(25), which provides a frequency of 50 MHz/2^{26} = 0.7451 Hz for a system clock with a frequency of 50 MHz. The frequency of the internal signal *VERY_SLOW_CLK* determines the word display rate of the display—that is, how fast the words are turned on and off on the four 7-segment LED displays.

- Logic 1 turns a display off while logic 0 turns a display on. To turn only display 0 on and all other displays off, we write *DISP* <= **not** "0001", which evaluates to "1110". To turn only display 1 on and all other displays off, we write *DISP* <= **not** "0010", which evaluates to "1101". Displays 2 and 3 are treated in a similar manner.

If you elect to use a hardware board, you will observe the words HI GUYS And GALS sequentially displayed on the four 7-segment LED displays. Other words or characters can be displayed by providing the bit patterns for each of the letters of the word. There are a few letters—such as K, M, N, Q, T, V, W, X, and Z—that cannot be displayed on a 7-segment LED display because of insufficient segments or segments that are not in the required position or location. Lowercase letters or numbers can be used to display certain letters, such as r for the letter R, n for the letter N, 4 for the letter Y, and 5 for the letter S.

PROBLEMS

Section 14.2 Multiplexed Display System for Four 7-Segment LED Displays

14.1 What is the purpose of the multiplexed display system in Figure 14.1?

14.2 What is the purpose of the frequency divider in Figure 14.1?

14.3 What *SLOW_CLK* frequency is necessary to light all the displays simultaneously for the multiplexed display system in Figure 14.1 to observe no blinking?

14.4 What *SLOW_CLK* frequency would be necessary to light all the displays simultaneously for the multiplexed display system in Figure 14.1 if it were set up to light nine displays rather than four to observe no blinking?

14.5 What is the purpose of the 4-to-1 MUX array in Figure 14.1?

14.6 What is the purpose of the HEX display decoder in Figure 14.1?

14.7 What is the purpose of the 2-to-4 decoder in Figure 14.1? What signal controls the output of the 2-to-4 decoder?

14.8 What signal controls the output of the 4-to-1 MUX array in Figure 14.1?

14.9 What is the purpose of the 2-bit counter in Figure 14.1?

Section 14.3 Designing a Multiplexed Display System Using VHDL

14.10 How are the signals *DISPLAY_0* through *DISPLAY_3* in Figure 14.1 entered into the VHDL code for the design of the 4-to-1 MUX array?

14.11 In Waveform 14.1 for the MUX_ARRAY, how are the values displayed for the signal *COUNT_2B*(1:0)? How are the values displayed for the signal *ADDR_BIN*(3:0)?

14.12 Analyze the partial section of code in Listing P14.12 for the HEX display decoder in Figure 14.1. What are the active levels (active high or active low) of the outputs of the HEX display decoder that drives the inputs of the 7-segment display as determined from the code?

```
d7s <= "11000000" when addr_bin =
          "0000" else
       "11111001" when addr_bin =
          "0001" else
       "10100100" when addr_bin =
          "0010" else
```

LISTING P14.12

14.13 What is the error in changing *ADDR_BIN* = "0000" to *ADDR_BIN* <= "0000" in Listing P14.12?

14.14 Can when others be used in a conditional signal assignment statement? Name two other statements where when others can be used.

14.15 What use clause is required for the section of code in Listing P14.15 to define the "+" arithmetic operator, and why is it required?

```
process (rst, clk)
begin
   if rst = '1' then count_2b <= "00";
   elsif rising_edge (clk) then
         count_2b <= count_2b + 1;
   end if;
end process;
```

LISTING P14.15

14.16 Why are the two signals *RST* and *CLK* necessary in the sensitivity list for the process in Listing P14.15?

14.17 Listing P14.17 shows a section of code for a frequency divider that provides a frequency for *SLOW_CLK* that is ~1 Hz. Write a simple signal assignment (Boolean equation) for *SLOW_CLK* that will be closer to 1 Hz than the statement for *SLOW_CLK* that is shown in Listing P14.17.

```
process (clk, count)
begin
   if rising_edge (clk) then count <=
        count + 1;
   end if;
--the frequency of slow_clk is 1.49 Hz
--for clk = 50 MHz (50 M/2^25 Hz = 1.49
--Hz)
   Slow_clk <= count (24);
end process;
```

LISTING P14.17

14.18 In what location must internal signals be placed in VHDL? What is a common error that is sometimes made when listing internal signals?

14.19 Find the errors in the architecture declaration shown in Listing P14.19 for a 2-to-4 decoder.

```
architecture Behavioral of Decoder
begin
process (count_2b)
begin
     case count_2b
          when "00" => AN <= "1110";
               --Lights only Display 0
          when "01" => AN <= "1101";
               --Lights only Display 1
          when "10" => AN <= "1011";
               --Lights only Display 2
          when "11" => AN <= "0111";
               --Lights only Display 3
          when others => null;
     end case;
end process;
```

LISTING P14.19

14.20 How can the code in Listing P14.19 easily be modified to drive four 7-segment LED displays that have active high inputs?

Section 14.4 Complete Design of a Multiplexed Display System Using a Flat Design Approach

14.21 Provide the two main reasons for using documentation style M in a flat design approach.

14.22 What frequency must *SLOW_CLK* be in order to activate four multiplexed displays at least 24 times per second? What is the slowest frequency if the system clock is 50 MHz? What assignment is required to make this frequency available for *SLOW_CLK*?

Section 14.5 Complete Design of a Multiplexed Display System Using a Hierarchal Design Approach

14.23 In our example of a hierarchal design approach for the multiplexed display system, how are modules represented?

14.24 Is it necessary to label the formal signals (the input and output signals for each component) in a structural design?

14.25 If the component definitions are first defined in the component definition section of a structural design, how can the component declarations be easily obtained?

14.26 What does documentation style M mark in a hierarchal design approach?

14.27 Listing P14.27 provides the VHDL code for the installed or instantiated component C1. Describe what part of the circuit this component represents for the multiplexed display system and how it is interconnected in the circuit.

```
C1: MUX_Array port map (count_2b =>
count_2b, addr_bin => addr_bin);
```

LISTING P14.27

Section 14.6 Designing a Word Display System Using a Flat Design Approach

14.28 In the schematic for a word display system in Figure 14.5, how many bits are used for each letter or character that is displayed and each word that is displayed?

14.29 How is the frequency selected for the signal *VERY_SLOW_CLK* in Figure 14.5? Calculate the period of the frequency for the signal *VERY_SLOW_CLK*.

14.30 Calculate the period of the frequency for the signal *SLOW_CLK* in Figure 14.5.

14.31 What is the bit pattern required to display the letter n in the word display system in Figure 14.5?

14.32 How many flip-flops are required in the design of the Frequency Divider in the word display system in Figure 14.5?

14.33 Do you think it is possible to redesign the word display system in Figure 14.5 so that words can be scrolled across Disp 3, Disp 2, Disp 1, and Disp 0 from right to left?

Chapter 15

Designing Instruction Decoders

Chapter Outline

15.1 INTRODUCTION

In this chapter, you will learn about the instruction decoder, how to obtain instruction decoder truth tables, and how to write VHDL code for an instruction decoder for VBC1.

15.2 PURPOSE OF THE INSTRUCTION DECODER

The instruction decoder is a combinational logic circuit that provides the control signals (or control bits) to operate the data path unit and the control unit of a computer. Figure 15.1 shows a logic symbol for the instruction decoder for VBC1.

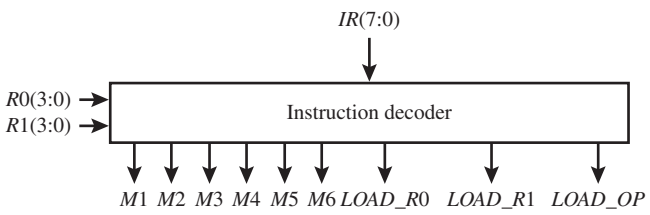


FIGURE 15.1
Logic Symbol for the instruction decoder for VBC1

The inputs to the instruction decoder are provided by the instruction register (*IR*) bits 7:0, *R*0 bits 3:0, and *R*1 bits 3:0. IR bits 7:0 are the current content of the instruction memory, and *R*0 bits 3:0 and *R*1 bits 3:0 are the outputs of the data registers R0 and R1, respectively. Recall that the bits in the IR represent the machine code for a single instruction in the instruction set of VBC1 and that each instruction in a program will be interpreted—that is, decoded—one instruction at a time. The outputs of the instruction decoder are the control bits *M*1, *M*2, *M*3, *M*4, *M*5, *M*6, *LOAD_R*0, *LOAD_R*1, and *LOAD_OP*, and these control bits supply the inputs to the data path unit and the control unit to cause each instruction to be executed.

We will determine the instruction decoder truth table for each instruction and then write the Boolean equations for the control bits. Once the Boolean equations are generated, we can write the VHDL code for the instruction decoder.

15.3 INSTRUCTION DECODER TRUTH TABLES FOR THE IN, OUT, AND MOV INSTRUCTIONS

Figure 15.2 shows the schematic for a partial data path circuit for VBC1. We can write the truth table for the IN instruction using this schematic.

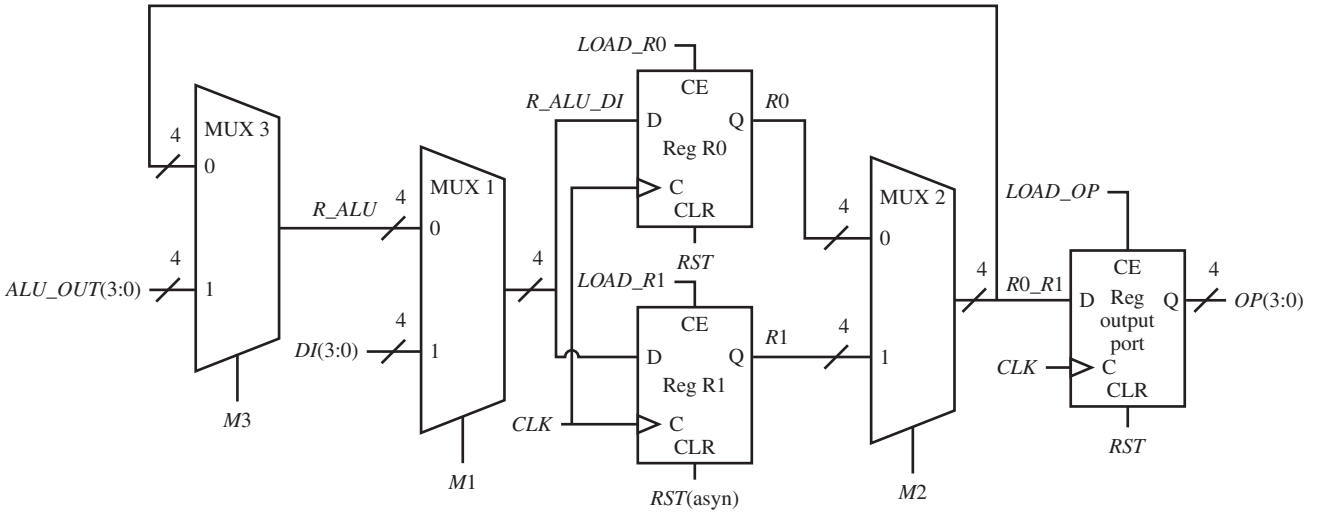


FIGURE 15.2 Schematic for a partial data path circuit for VBC1

Table 15.1 shows the instruction decoder truth table for the IN instruction for VBC1.

TABLE 15.1 Instruction decoder truth table for the IN instruction for VBC1

	IR								Control bits					
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	*M*1	*M*2	*M*3	*LOAD_R*0	*LOAD_R*1	*LOAD_OP*
IN R0	1	0	1	0	0	0	0	0	1	0*	0*	1	0	0
IN R1	1	0	1	1	0	0	0	0	1	0*	0*	0	1	0

*Actual value does not matter

Things you should notice about the instruction decoder truth table for the IN instruction in Table 15.1:

- The values for the *IR* bits for the instruction decoder are the machine code bits for the IN instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the IN DR instruction, which is DR ← DI(3:0).
- The instruction IN R0, which loads data input DI into register R0, requires control bit *M*1 to be set to 1 and control bit *LOAD_R*0 to be set to 1. During this instruction, *LOAD_R*1 must be set to 0 so the contents of register R1 do not change.
- The instruction IN R1, which loads data input DI into register R1, requires control bit *M*1 to be set to 1 and control bit *LOAD_R*1 to be set to 1. During this instruction, *LOAD_R*0 must be set to 0 so the contents of register R0 do not change.

The Boolean equations for the control bits for the IN instruction are

$$M1 = 1, M2 = 0, M3 = 0$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

We can also write the truth table for the OUT instruction using the schematic for the partial data path circuit for VBC1 shown in Figure 15.2. Table 15.2 shows the instruction decoder truth table for the OUT instruction for VBC1.

TABLE 15.2 Instruction decoder truth table for the OUT instruction for VBC1

	IR								Control bits					
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***M*2**	***M*3**	***LOAD_R*0**	***LOAD_R*1**	***LOAD_OP***
OUT R0	1	1	0	0	0	0	0	0	0*	0	0*	0	0	1
OUT R1	1	1	0	1	0	0	0	0	0*	1	0*	0	0	1

*Actual value does not matter

Things you should notice about the instruction decoder truth table for the OUT instruction in Table 15.2:

- The values for the *IR* bits for the instruction decoder are the machine code bits for the OUT instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the OUT DR instruction, which is OP ← DR.
- The instruction OUT R0, which loads register R0 into output port OP, requires control bit *M*2 to be set to 0 and control bit *LOAD_OP* to be set to 1. During this instruction, *LOAD_R*0 and *LOAD_R*1 must be set to 0 so that the values of registers R0 and R1 are not changed.
- The instruction OUT R1, which loads register R1 into the output port OP, requires control bit *M*2 to be set to 1 and control bit *LOAD_OP* to be set to 1. During this instruction, *LOAD_R*0 and *LOAD_R*1 must be set to 0 so that the values of registers R0 and R1 are not changed.

The Boolean equations for the control bits for the OUT instruction are

$$M1 = 0, M2 = IR(4), M3 = 0$$
$$LOAD\_R0 = 0, LOAD\_R1 = 0, LOAD\_OP = 1$$

We can write the truth table for the MOV instruction using the schematic for the partial data path circuit for VBC1 shown in Figure 15.2. Table 15.3 shows the instruction decoder truth table for the MOV instruction for VBC1.

TABLE 15.3 Instruction decoder truth table for the MOV instruction for VBC1

	IR								Control bits					
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***M*2**	***M*3**	***LOAD_R*0**	***LOAD_R*1**	***LOAD_OP***
MOV R0,R0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
MOV R0,R1	0	0	0	0	1	0	0	0	0	1	0	1	0	0
MOV R1,R0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
MOV R1,R1	0	0	0	1	1	0	0	0	0	1	0	0	1	0

Things you should notice about the instruction decoder truth table for the MOV instruction in Table 15.3:

- The values for the IR bits for the instruction decoder are the machine code bits for the MOV instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the MOV DR,SR instruction which is DR ← SR.
- The instruction MOV R0,R0, which moves the contents of source register R0 to destination register R0, requires control bit *M*2 to be set to 0, control bit *M*3 to be set to 0, control bit *M*1 to be set to 0, and *LOAD_R*0 to be set to 1. During this instruction, *LOAD_R*1 must be set to 0 so the contents of register R1 do not change.
- The instruction MOV R0,R1, which moves the contents of source register R1 to destination register R0, requires control bit *M*2 to be set to 1, control bit *M*3 to be set to 0, control bit *M*1 to be set to 0, and *LOAD_R*0 to be set to 1. During this instruction, *LOAD_R*1 must be set to 0 so the contents of register R1 do not change.
- The instruction MOV R1,R0, which moves the contents of source register R0 to destination register R1, requires control bit *M*2 to be set to 0, control bit *M*3 to be set to 0, control bit *M*1 to be set to 0, and *LOAD_R*1 to be set to 1. During this instruction, *LOAD_R*0 must be set to 0 so the contents of register R0 do not change.
- The instruction MOV R1,R1, which moves the contents of source register R1 to destination register R1, requires control bit *M*2 to be set to 1, control bit *M*3 to be set to 0, control bit *M*1 to be set to 0, and *LOAD_R*1 to be set to 1. During this instruction, *LOAD_R*0 must be set to 0 so the contents of register R0 do not change.

The Boolean equations for the control bits for the MOV instruction are

$$M1 = 0, M2 = IR(3), M3 = 0$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

15.4 DESIGNING AN INSTRUCTION DECODER FOR THE IN INSTRUCTION

To design an instruction decoder, we recommend using **Procedure ID**, which is listed as follows: (1) use a process with a case statement to select each instruction by its OPCODE, (2) specify the default instruction decoder output values before the case statement, and (3) use Boolean equations for the control bits within the case statement. (Note: Only the control bits that are different from the default instruction decoder output values need to be added to the case statement.)

Procedure ID will now be used for the design of the instruction decoder for the IN instruction. The OPCODE is 101 for the IN instruction.

As obtained earlier, the Boolean equations for the control bits for the IN instruction are

$$M1 = 1, M2 = 0, M3 = 0$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

Listing 15.1 shows a partial VHDL design for an instruction decoder for the IN instruction using a behavioral design style—that is, a process with a case statement.

LISTING 15.1 Partial VHDL design for an instruction decoder for the IN instruction using a behavioral design style

```
process (ir)
begin
     --default instruction decoder output values
     m1 <= '0'; m2 <= '0'; m3 <= '0';
     load_r0 <= '0'; load_r1 <= '0'; load_op <= '0';
     case ir (7 downto 5) is
          --for the IN instruction
          when "101" => m1 <= '1'; load_r0 <= not ir(4);
                        load_r1 <= ir(4);
          when others => null;
     end case;
end process;
```

Things you should notice about the partial VHDL design in Listing 15.1:

- Default instruction decoder output values are assigned to all control bits before the case statement to ensure proper circuit operation—that is, so that inferred latches will not be generated, when different instructions are selected in the case statement.
- The case statement evaluates the signal *IR*(7 downto 5) or the bits 7:5 in the IR, which is the OPCODE for the IN instruction.
- When the OPCODE 101 for the IN instruction is true, assignments are made to the control bits following the symbol =>. The assignments to the control bits establish the data paths necessary for the IN instruction to be performed by the VBC1 architecture.
- One can write a CSA rather that a process with a case statement for each of the control bits for the output of the instruction decoder. This is not a good idea because any additional instructions that are added will require changing the CSA for each of the current control bits. We use a process with a case statement for writing the control bits so that the current control bits do not have to be changed as additional instructions are added.

15.5 DESIGNING AN INSTRUCTION DECODER FOR THE OUT AND MOV INSTRUCTIONS

Procedure ID will now be used for the design of the instruction decoder for the OUT and MOV instructions. The OPCODE is 110 for the OUT instruction and 000 for the MOV instruction.

As obtained earlier, the Boolean equations for the control bits for the OUT instruction are

$$M1 = 0, M2 = IR(4), M3 = 0$$
$$LOAD\_R0 = 0, LOAD\_R1 = 0, LOAD\_OP = 1$$

As obtained earlier, the Boolean equations for the control bits for the MOV instruction are

$$M1 = 0, M2 = IR(3), M3 = 0$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

Listing 15.2 shows a partial VHDL design for an instruction decoder for the OUT and MOV instructions using a behavioral design style—that is, a process with a case statement.

LISTING 15.2 Partial VHDL design for an instruction decoder for the OUT and MOV instructions using a behavioral design style

```
process (ir)
begin
     --default instruction decoder output values
     m1 <= '0'; m2 <= '0'; m3 <= '0';
     load_r0 <= '0'; load_r1 <= '0'; load_op <= '0';
     case ir (7 downto 5) is
          --for the OUT instruction
          when "110" => m2 <= ir(4); load_op <= '1';
          --for the MOV instruction
          when "000" => m2 <= ir(3); load_r0 <= not ir(4);
                        load_r1 <= ir(4);
          when others => null;
     end case;
end process;
```

Things you should notice about the partial VHDL design in Listing 15.2:

- The case statement evaluates the signal *IR*(7 downto 5) or the bits 7:5 in the IR, which is the OPCODE for the OUT and MOV instructions.
- When the OPCODE 110 for the OUT instruction is true, assignments are made to the control bits following the symbol =>. The assignments to the control bits establish the data paths necessary for the OUT instruction to be performed by the VBC1 architecture.
- When the OPCODE 000 for the MOV instruction is true, assignments are made to the control bits following the symbol =>. The assignments to the control bits establish the data paths necessary for the MOV instruction to be performed by the VBC1 architecture.

15.6 INSTRUCTION DECODER TRUTH TABLE FOR THE LOADI INSTRUCTION

Figure 15.3 shows the schematic for the complete data path circuit for VBC1. We can write the truth table for the LOADI instruction using this schematic.

Table 15.4 shows the instruction decoder truth table for the LOADI instruction for VBC1.

TABLE 15.4 Instruction decoder truth table for the LOADI instruction for VBC1

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***M*2**	***M*3**	***M*4**	***M*5**	***LOAD_R*0**	***LOAD_R*1**	***LOAD_OP***
LOADI R0,Data	0	0	1	0	D	D	D	D	0	0*	1	0*	1	1	0	0
LOADI R1,Data	0	0	1	1	D	D	D	D	0	0*	1	0*	1	0	1	0

*Actual value does not matter

Things you should notice about the instruction decoder truth table for the LOADI instruction in Table 15.4:

- The values for the IR bits for the instruction decoder are the machine code bits for the LOADI instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the LOADI DR,Data instruction, which is DR ← IR(3:0).
- The instruction LOADI R0,Data, which moves the contents of instruction register bits 3:0 to destination register R0, requires control bit *M*5 to be set to 1, control bit *M*3 to be set to 1, control bit *M*1 to be set to 0, and *LOAD_R*0 to be set to 1. During this instruction, *LOAD_R*1 must be set to 0 so the contents of register R1 do not change.
- The instruction LOADI R1,Data, which moves the contents of instruction register bits 3:0 to destination register R1, requires control bit *M*5 to be set to 1, control bit *M*3 to be set

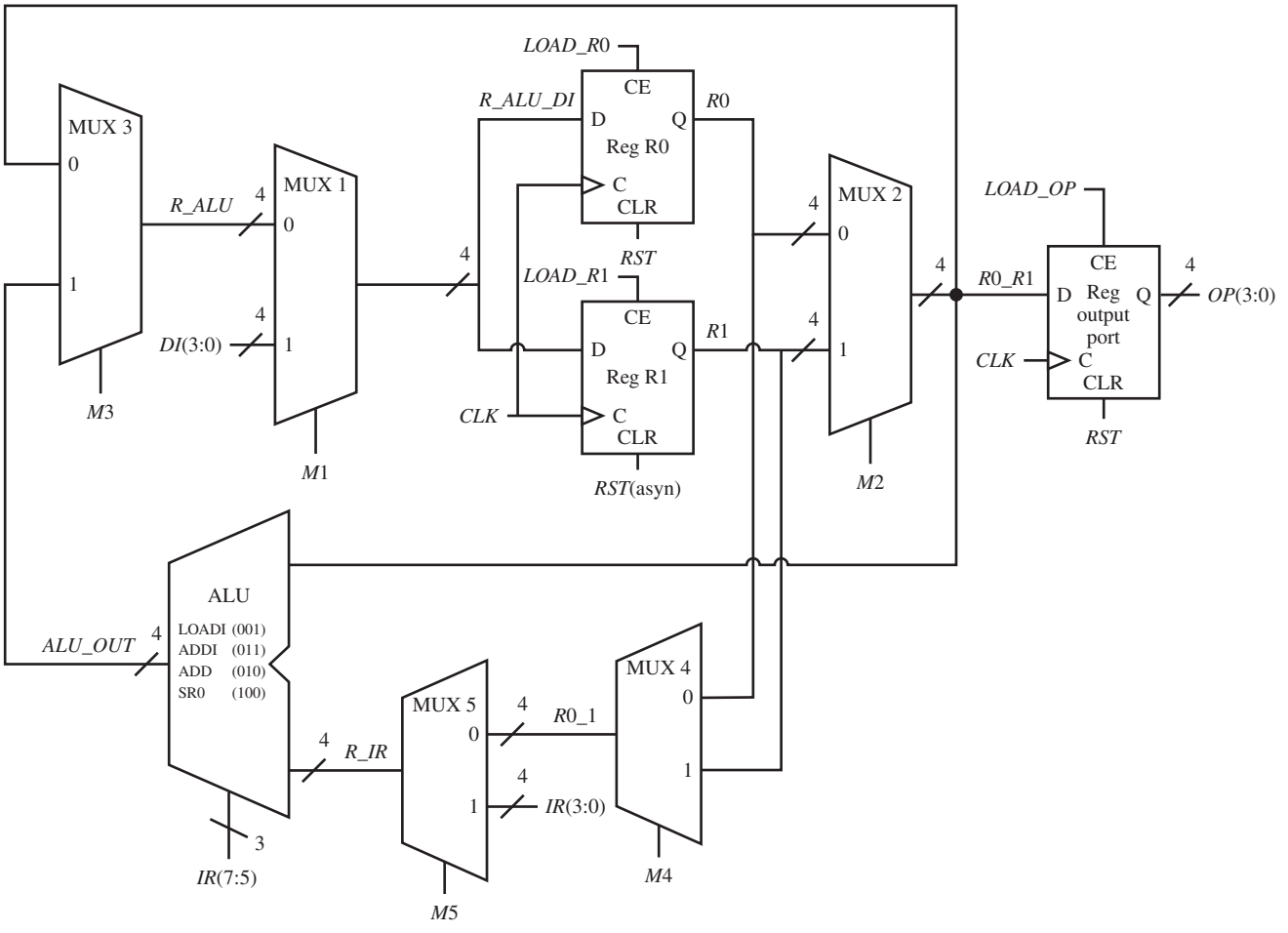


FIGURE 15.3 Schematic for the complete data path circuit for VBC1

to 1, control bit $M1$ to be set to 0, and $LOAD\_R1$ to be set to 1. During this instruction, $LOAD\_R0$ must be set to 0 so the contents of register R0 do not change.

The Boolean equations for the control bits for the LOADI instruction are

$$M1 = 0, M2 = 0, M3 = 1, M4 = 0, M5 = 1$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

15.7 INSTRUCTION DECODER TRUTH TABLE FOR THE ADDI INSTRUCTION

We can write the truth table for the ADDI instruction using the schematic shown in Figure 15.3 for the complete data path circuit for VBC1. Table 15.5 shows the instruction decoder truth table for the ADDI instruction for VBC1.

TABLE 15.5 Instruction decoder truth table for the ADDI instruction for VBC1

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	*M1*	*M2*	*M3*	*M4*	*M5*	*LOAD_R0*	*LOAD_R1*	*LOAD_OP*
ADDI R0,Data	0	1	1	0	D	D	D	D	0	0	1	0*	1	1	0	0
ADDI R1,Data	0	1	1	1	D	D	D	D	0	1	1	0*	1	0	1	0

*Actual value does not matter

Things you should notice about the instruction decoder truth table for the ADDI instruction in Table 15.5:

- The values for the IR bits for the instruction decoder are the machine code bits for the ADDI instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the ADDI DR,Data instruction, which is DR ← DR + IR(3:0).
- The instruction ADDI R0,Data, which adds the contents of destination register R0 with instruction register bits 3:0 and places the result in destination register R0, requires control bit *M*2 to be set to 0, control bit *M*5 to be set to 1, control bit *M*3 to be set to 1, control bit *M*1 to be set to 0, and *LOAD_R*0 to be set to 1. During this instruction, *LOAD_R*1 must be set to 0 so the contents of register R1 do not change.
- The instruction ADDI R1,Data, which adds the contents of destination register R1 with instruction register bits 3:0 and places the result in destination register R1, requires control bit *M*2 to be set to 1, control bit *M*5 to be set to 1, control bit *M*3 to be set to 1, control bit *M*1 to be set to 0, and *LOAD_R*1 to be set to 1. During this instruction, *LOAD_R*0 must be set to 0 so the contents of register R0 do not change.

The Boolean equations for the control bits for the ADDI instruction are

$$M1 = 0, M2 = IR(4), M3 = 1, M4 = 0, M5 = 1$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

15.8 INSTRUCTION DECODER TRUTH TABLE FOR THE ADD INSTRUCTION

We can write the truth table for the ADD instruction using the schematic shown in Figure 15.3 for the complete data path circuit for VBC1. Table 15.6 shows the instruction decoder truth table for the ADD instruction for VBC1.

TABLE 15.6 Instruction decoder truth table for the ADD instruction for VBC1

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	*M1*	*M2*	*M3*	*M4*	*M5*	*LOAD_R0*	*LOAD_R1*	*LOAD_OP*
ADD R0,R0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0
ADD R0,R1,	0	1	0	0	1	0	0	0	0	0	1	1	0	1	0	0
ADD R1,R0	0	1	0	1	0	0	0	0	0	1	1	0	0	0	1	0
ADD R1,R1	0	1	0	1	1	0	0	0	0	1	1	1	0	0	1	0

Things you should notice about the instruction decoder truth table for the ADD instruction in Table 15.6:

- The values for the IR bits for the instruction decoder are the machine code bits for the ADD instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the ADD DR,SR instruction, which is DR ← DR + SR.
- The instruction ADD R0,R0, which adds the contents of destination register R0 with source register R0 and places the result in destination register R0, requires control bit *M*2 to be set to 0, control bit *M*4 to be set to 0, control bit *M*5 to be set to 0, control bit *M*3 to be set to 1, control bit *M*1 to be set to 0, and *LOAD_R*0 to be set to 1. During this instruction, *LOAD_R*1 must be set to 0 so the contents of register R1 do not change.
- The instruction ADD R0,R1, which adds the contents of destination register R0 with source register R1 and places the result in destination register R0, requires control bit *M*2 to be

set to 0, control bit $M4$ to be set to 1, control bit $M5$ to be set to 0, control bit $M3$ to be set to 1, control bit $M1$ to be set to 0, and $LOAD\_R0$ to be set to 1. During this instruction, $LOAD\_R1$ must be set to 0 so the contents of register R1 do not change.

- The instruction ADD R1,R0, which adds the contents of destination register R1 with source register R0 and places the result in destination register R1, requires control bit $M2$ to be set to 1, control bit $M4$ to be set to 0, control bit $M5$ to be set to 0, control bit $M3$ to be set to 1, control bit $M1$ to be set to 0, and $LOAD\_R1$ to be set to 1. During this instruction, $LOAD\_R0$ must be set to 0 so the contents of register R0 do not change.
- The instruction ADD R1,R1, which adds the contents of destination register R1 with source register R1 and places the result in destination register R1, requires control bit $M2$ to be set to 1, control bit $M4$ to be set to 1, control bit $M5$ to be set to 0, control bit $M3$ to be set to 1, control bit $M1$ to be set to 0, and $LOAD\_R1$ to be set to 1. During this instruction, $LOAD\_R0$ must be set to 0 so the contents of register R0 do not change.

The Boolean equations for the control bits for the ADD instruction are

$$M1 = 0, M2 = IR(4), M3 = 1, M4 = IR(3), M5 = 0$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

15.9 INSTRUCTION DECODER TRUTH TABLE FOR THE SR0 INSTRUCTION

We can write the truth table for the SR0 instruction using the schematic shown in Figure 15.3 for the complete data path circuit for VBC1. Table 15.7 shows the instruction decoder truth table for the SR0 instruction for VBC1.

TABLE 15.7 Instruction decoder truth table for the SR0 instruction for VBC1

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M1***	***M2***	***M3***	***M4***	***M5***	***LOAD_R0***	***LOAD_R1***	***LOAD_OP***
SR0 R0,R0	1	0	0	0	0	0	0	0	0	0*	1	0	0	1	0	0
SR0 R0,R1	1	0	0	0	1	0	0	0	0	0*	1	1	0	1	0	0
SR0 R1,R0	1	0	0	1	0	0	0	0	0	0*	1	0	0	0	1	0
SR0 R1, R1	1	0	0	1	1	0	0	0	0	0*	1	1	0	0	1	0

*Actual value does not matter

Things you should notice about the instruction decoder truth table for the SR0 instruction in Table 15.7:

- The values for the IR bits for the instruction decoder are the machine code bits for the SR0 instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the SR0 DR,SR instruction, which is DR ← 0 SR(3:1).
- The instruction SR0 R0,R0, which shifts the contents of source register R0 to the right by one bit then fills the MSB with a 0 and places the result in destination register R0, requires control bit $M4$ to be set to 0, control bit $M5$ to be set to 0, control bit $M3$ to be set to 1, control bit $M1$ to be set to 0, and $LOAD\_R0$ to be set to 1. During this instruction, $LOAD\_R1$ must be set to 0 so the contents of register R1 do not change.
- The instruction SR0 R0,R1, which shifts the contents of source register R1 to the right by one bit then fills the MSB with a 0 and places the result in destination register R0, requires

control bit $M4$ to be set to 1, control bit $M5$ to be set to 0, control bit $M3$ to be set to 1, control bit $M1$ to be set to 0, and $LOAD\_R0$ to be set to 1. During this instruction, $LOAD\_R1$ must be set to 0 so the contents of register R1 do not change.

- The instruction SR0 R1,R0, which shifts the contents of source register R0 to the right by one bit then fills the MSB with a 0 and places the result in destination register R1, requires control bit $M4$ to be set to 0, control bit $M5$ to be set to 0, control bit $M3$ to be set to 1, control bit $M1$ to be set to 0, and $LOAD\_R1$ to be set to 1. During this instruction, $LOAD\_R0$ must be set to 0 so the contents of register R0 do not change.
- The instruction SR0 R1,R1, which shifts the contents of source register R1 to the right by one bit then fills the MSB with a 0 and places the result in destination register R1, requires control bit $M4$ to be set to 1, control bit $M5$ to be set to 0, control bit $M3$ to be set to 1, control bit $M1$ to be set to 0, and $LOAD\_R1$ to be set to 1. During this instruction, $LOAD\_R0$ must be set to 0 so the contents of register R0 do not change.

The Boolean equations for the control bits for the SR0 instruction are

$$M1 = 0, M2 = 0, M3 = 1, M4 = IR(3), M5 = 0$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

15.10 DESIGNING AN INSTRUCTION DECODER FOR THE SR0 INSTRUCTION

Procedure ID will now be used for the design of the instruction decoder for the SR0 instruction. The OPCODE is 100 for the SR0 instruction.

As obtained earlier, the Boolean equations for the control bits for the SR0 instruction are

$$M1 = 0, M2 = 0, M3 = 1, M4 = IR(3), M5 = 0$$
$$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0$$

Listing 15.3 shows a partial VHDL design for an instruction decoder for the SR0 instruction using a behavioral design style—that is, a process with a case statement.

LISTING 15.3 Partial VHDL design for an instruction decoder for the SR0 instruction using a behavioral design style

```
process (ir)
begin
    --default instruction decoder output values
    m1 <= '0'; m2 <= '0'; m3 <= '0'; m4 <= '0'; m5 <= '0';
    load_r0 <= '0'; load_r1 <= '0'; load_op <= '0';
    case ir (7 downto 5) is
        --for the SR0 instruction
        when "100" => m3 <= '1'; m4 <= ir(3);
                      load_r0 <= not ir(4);
                      load_r1 <= ir(4);
        when others => null;
    end case;
end process;
```

Things you should notice about the partial VHDL design in Listing 15.3:

- When the OPCODE 100 for the SR0 instruction is true, assignments are made to the control bits following the symbol =>. The assignments to the control bits establish the data paths necessary for the SR0 instruction to be performed by the VBC1 architecture.

15.11 INSTRUCTION DECODER TRUTH TABLE FOR THE JNZ INSTRUCTION

Figure 15.4 shows a partial schematic for the control circuit for VBC1.

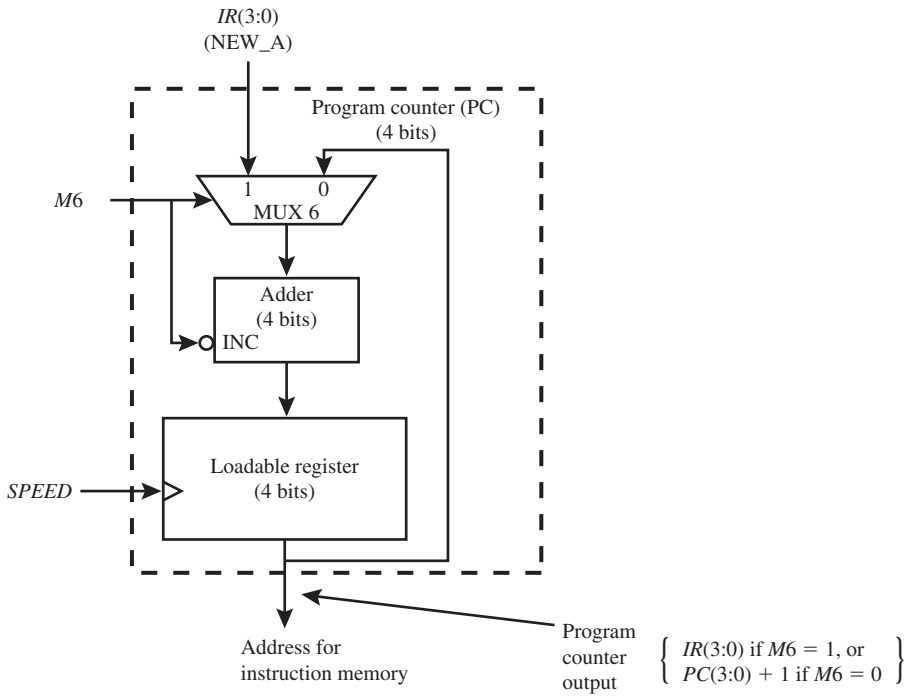


FIGURE 15.4 Partial schematic for the control circuit for VBC1

The value of the control bit *M*6 determines how the JNZ instruction JNZ DR,Address is executed. If *M*6 is 1, then the JNZ instruction jumps to the address *IR*(3:0) specified in the instruction; however, if *M*6 is 0, then the JNZ instruction falls through or simply executes the next instruction at the address PC(3:0) + 1.

The value of *M*6 is dependent of the value of register R0 or register R1—that is, the destination register (DR).

The transfer function form for the JNZ instruction JNZ DR,Address is PC ← IR(3:0) if DR ≠ 0 else PC ← PC + 1. In order to execute the JNZ instruction, we must design circuits for VBC1 that sample both registers R0 and R1 to determine when their values are 0000 or not 0000.

Figure 15.5 (on the next page) shows the complete circuit design of comparator circuits that sample the outputs of registers R0 and R1. The comparator circuit in Figure 15.5a provides an output $Z0 = 1$ when register R0 = 0000. The comparator circuit in Figure 15.5b provides an output $Z1 = 1$ when register R1 = 0000.

Things you should notice about the design of the comparator circuits in Figure 15.5:

- When comparing the value of the 4-bit registers R0 and R1 with the value 0000, the Comparators circuits can be reduced via Boolean algebra to simple Decoders circuits—for example,

$$Z0 = \overline{R0(0)\oplus 0}\cdot\overline{R0(1)\oplus 0}\cdot\overline{R0(2)\oplus 0}\cdot\overline{R0(3)\oplus 0} = \overline{R0(0)}\cdot\overline{R0(1)}\cdot\overline{R0(2)}\cdot\overline{R0(3)}$$
$$Z1 = \overline{R1(0)\oplus 0}\cdot\overline{R1(1)\oplus 0}\cdot\overline{R1(2)\oplus 0}\cdot\overline{R1(3)\oplus 0} = \overline{R1(0)}\cdot\overline{R1(1)}\cdot\overline{R1(2)}\cdot\overline{R1(3)}$$

- Output $Z0 = 1$ when $R0 = 0000$ and output Z0 = 0 when $R0 \neq 0000$.
- Output $Z1 = 1$ when $R1 = 0000$ and output Z1 = 0 when $R1 \neq 0000$.

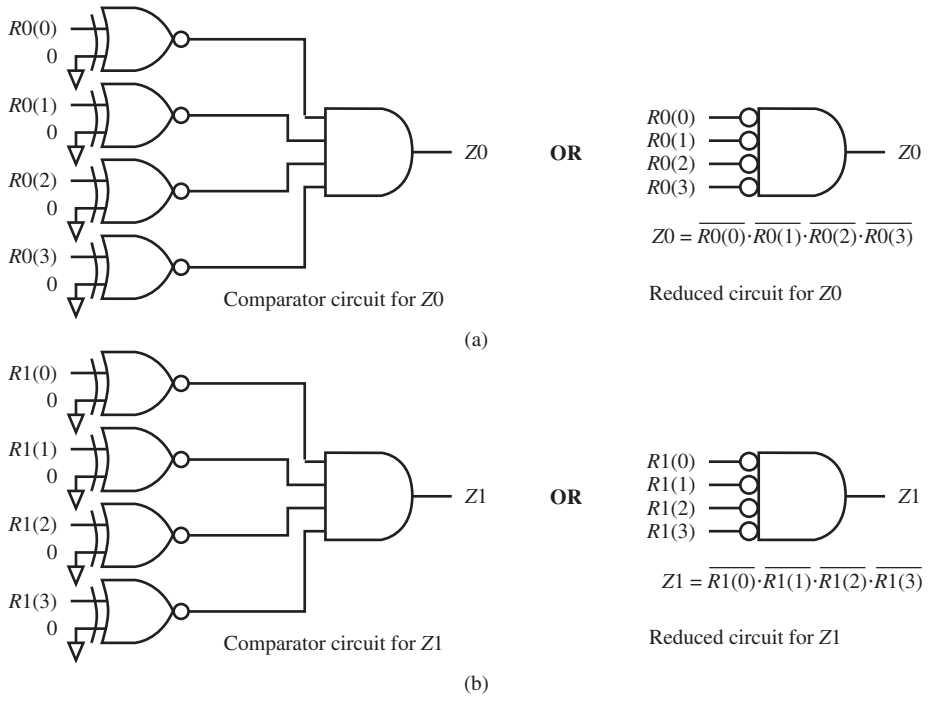


FIGURE 15.5 Design of comparator circuits that sample both registers R0 and R1: (a) comparator circuit for register R0; (b) comparator circuit for register R1

- Z0 = 1 acts as flags to indicate when R0 = 0000, and Z1 = 1 acts as flags to indicate when R1 = 0000.
- Observe that the comparator circuits for Z0 and Z1 reduces to simple decoder circuits.

Table 15.8 shows the instruction decoder truth table for the JNZ instruction for VBC1 using the inputs Z0 and Z1—that is, the outputs of the comparator (or decoder) circuits.

TABLE 15.8 Instruction decoder truth table for the JNZ instruction for VBC1

	IR										Control bits								
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***Z*1**	***Z*0**	***M*1**	***M*2**	***M*3**	***M*4**	***M*5**	***M*6**	*LOAD_R***0**	*LOAD_R***1**	*LOAD_OP*
JNZ R0,Address	1	1	1	0	A	A	A	A	0	0	0*	0*	0*	0*	0*	1	0	0	0
JNZ R0,Address	1	1	1	0	A	A	A	A	0	1	0*	0*	0*	0*	0*	0	0	0	0
JNZ R0,Address	1	1	1	0	A	A	A	A	1	0	0*	0*	0*	0*	0*	1	0	0	0
JNZ R0,Address	1	1	1	0	A	A	A	A	1	1	0*	0*	0*	0*	0*	0	0	0	0
JNZ R1,Address	1	1	1	1	A	A	A	A	0	0	0*	0*	0*	0*	0*	1	0	0	0
JNZ R1,Address	1	1	1	1	A	A	A	A	0	1	0*	0*	0*	0*	0*	1	0	0	0
JNZ R1,Address	1	1	1	1	A	A	A	A	1	0	0*	0*	0*	0*	0*	0	0	0	0
JNZ R1,Address	1	1	1	1	A	A	A	A	1	1	0*	0*	0*	0*	0*	0	0	0	0

*Actual value does not matter

Things you should notice about the instruction decoder truth table for the JNZ instruction in Table 15.8:

- The values for the *IR* bits for the instruction decoder are the machine code bits for the JNZ instruction. The values for *Z*1 and *Z*0 represent all possible values for these bits for each JNZ instruction. The values for the control bits for the instruction decoder truth table are

determined by ensuring that the control bits perform the transfer function form for the JNZ DR,Address instruction, which is PC ← IR(3:0) if DR ≠ 0 else PC ← PC + 1.

- The instruction JNZ R0,Address will load a new address into the PC if R0 ≠ 0. This requires that control bit *M*6 must be 1 when *IR*(4) is 0 and *Z*0 is 0.
- The instruction JNZ R0,Address will increment the PC if R0 = 0. This requires that control bit *M*6 must be 0 when *IR*(4) is 0 and *Z*0 is 1.
- The instruction JNZ R1,Address will load a new address into the PC if R1 ≠ 0. This requires that control bit *M*6 must be 1 when *IR*(4) is 1 and *Z*1 is 0.
- The instruction JNZ R1,Address will increment the PC if R1 = 0. This requires that control bit *M*6 must be 0 when *IR*(4) is 1 and *Z*1 is 1.

The Boolean equation for the control bit for M6 for the JNZ instruction is

$$M6 = \overline{IR(4)} \cdot \overline{Z0} + IR(4) \cdot \overline{Z1}$$

where $Z0 = \overline{R0(0)} \cdot \overline{R0(1)} \cdot \overline{R0(2)} \cdot \overline{R0(3)}$ and $Z1 = \overline{R1(0)} \cdot \overline{R1(1)} \cdot \overline{R1(2)} \cdot \overline{R1(3)}$

so

$$M6 = \overline{IR(4)} \cdot \overline{\overline{R0(0)} \cdot \overline{R0(1)} \cdot \overline{R0(2)} \cdot \overline{R0(3)}} + IR(4) \cdot \overline{\overline{R1(0)} \cdot \overline{R1(1)} \cdot \overline{R1(2)} \cdot \overline{R1(3)}}$$

or

$$M6 = \overline{IR(4)} \cdot (R0(0) + R0(1) + R0(2) + R0(3)) + IR(4) \cdot (R1(0) + R1(1) + R1(2) + R1(3))$$

The Boolean equations for the control bits for the JNZ instruction are

$$M1 = 0, M2 = 0, M3 = 0, M4 = 0, M5 = 0,$$
$$M6 = \overline{IR(4)} \cdot (R0(0) + R0(1) + R0(2) + R0(3)) + IR(4) \cdot (R1(0) + R1(1) + R1(2) + R1(3))$$
$$LOAD\_R0 = 0, LOAD\_R1 = 0, LOAD\_OP = 0$$

15.12 DESIGNING AN INSTRUCTION DECODER FOR THE JNZ INSTRUCTION

Procedure ID will now be used for the design of the instruction decoder for the JNZ instruction. The OPCODE is 111 for the JNZ instruction.

As obtained earlier, the Boolean equations for the control bits for the JNZ instruction are

$$M1 = 0, M2 = 0, M3 = 0, M4 = 0, M5 = 0,$$
$$M6 = \overline{IR(4)} \cdot (R0(0) + R0(1) + R0(2) + R0(3)) + IR(4) \cdot (R1(0) + R1(1) + R1(2) + R1(3))$$
$$LOAD\_R0 = 0, LOAD\_R1 = 0, LOAD\_OP = 0$$

Listing 15.4 shows a partial VHDL design for an instruction decoder for the JNZ instruction using a behavioral design style—that is, a process with a case statement.

```
process (ir, r0, r1)
begin
    --default instruction decoder output values
    m1 <= '0'; m2 <= '0'; m3 <= '0'; m4 <= '0'; m5 <= '0';
    m6 <= '0'; load_r0 <= '0'; load_r1 <= '0'; load_op <= '0';
    case ir (7 downto 5) is
                                                    (Continued)
```

LISTING 15.4 Partial VHDL design for an instruction decoder for the JNZ instruction using a behavioral design style

```
        --for the JNZ instruction
        when "111" => m6 <= (not ir(4) and (r0(0) or r0(1) or
                             r0(2) or r0(3))) or (ir(4) and (r1(0)
                             or r1(1) or r1(2) or r1(3)));
        when others => null;
    end case;
end process;
```

Things you should notice about the partial VHDL design in Listing 15.4:

- When the OPCODE 111 for the JNZ instruction is true, an assignment is made to the control bit *M*6 following the symbol =>. The assignment to the control bit establishes the address path necessary for the JNZ instruction to be performed by the VBC1 architecture.

Listing 15.5 shows a partial VHDL design for an instruction decoder for the JNZ instruction using a behavioral design style—that is, a process with a case statement with if statements to specify the control bit *M*6 without using the Boolean equation obtained earlier.

```
process (ir, r0, r1)
begin
  --default instruction decoder output values
  m1 <= '0'; m2 <= '0'; m3 <= '0'; m4 <= '0'; m5 <= '0'; m6 <= '0';
  load_r0 <= '0'; load_r1 <= '0'; load_op <= '0';
  case ir (7 downto 5) is
    --for the JNZ instruction
    when "111" => if ir(4) = '0' then if r0 /= "0000" then m6 <= '1';  --jump
                                      else m6 <= '0';                  --increment
                                      end if;
                  elsif ir(4) = '1' then if r1 /= "0000" then m6 <= '1'; --jump
                                         else m6 <= '0';                 --increment
                                         end if;
                  end if;
    when others => null;
  end case;
end process;
```

LISTING 15.5 Partial VHDL design for an instruction decoder for the JNZ instruction using a behavioral design style with if statements

Things you should notice about the partial VHDL design in Listing 15.5:

- When the OPCODE 111 for the JNZ instruction is true, if statements are used to specify the control bit *M*6 following the symbol =>. The if statements for the control bit *M*6 establish the address path necessary for the JNZ instruction to be performed by the VBC1 architecture.
- The first if statement checks the value of *IR*(4) to see if the destination register is *R*0. The second if statement checks to see if the value of *R*0 is not 0. The operator /= means not equal in VHDL. If the destination register is *R*0 and its value is not 0, *M*6 takes on the value of 1; otherwise, *M*6 takes on the value of 0.
- The elsif statement checks the value of *IR*(4) to see if the destination register is a *R*1. The third if statement checks to see if the value of *R*1 is not 0. If the destination register is *R*1 and its value is not 0, *M*6 takes on the value of 1; otherwise, *M*6 takes on the value of 0.

15.13 DESIGNING AN INSTRUCTION DECODER FOR VBC1

An instruction decoder for VBC1 can be written by combining the VHDL code for each separate instruction decoder for the instructions of VBC1. The VHDL code for each individual case statement can be placed in one case statement under one process. This decreases the number of lines of code for the instruction decoder for VBC1. An example of this technique was provided earlier for the instruction decoder for the OUT and MOV instructions.

Listing 15.6 shows a general format or template for the instruction decoder for VBC1.

```
process (ir, r0, r1)
begin
     --put default instruction decoder output values here, e.g., m1 <= '0' m2 <=
     --'0', etc.
     case IR (7 downto 5) is--the OPCODE for each instruction is in bits (7:5) in the IR

          --the IN instruction has the OPCODE 101
          when "101" => --put output equations for the IN instruction here

          --the OUT instruction has the OPCODE 110
          when "110" => --put output equations for the OUT instruction here

          --the MOV instruction has the OPCODE 000
          when "000" => --put output equations for the MOV instruction here

          --the LOADI instruction has the OPCODE 001
          when "001" => --put output equations for the LOADI instruction here

          --the ADDI instruction has the OPCODE 011
          when "011" => --put output equations for the ADDI instruction here

          --the ADD instruction has the OPCODE 010
          when "010" => --put output equations for the ADD instruction here

          --the SR0 instruction has the OPCODE 100
          when "100" => --put output equations for the SR0 instruction here

          --the JNZ instruction has the OPCODE 111
          when "111" => --put output equations for the JNZ instruction here
          when others => null;
     end case;
end process;
```

LISTING 15.6 General format or template for the instruction decoder for VBC1

PROBLEMS

Section 15.2 Purpose of the Instruction Decoder

15.1 What is the purpose of the instruction decoder for VBC1?

15.2 Is the instruction decoder a combinational or a sequential logic circuit?

15.3 What are the input signals to the instruction decoder for VBC1?

15.4 How many control bits are generated by the instruction decoder for VBC1? List the control bits generated by the instruction decoder for VBC1.

Section 15.3 Instruction Decoder Truth Tables for the IN, OUT, and MOV Instructions

15.5 The instruction decoder truth table for the IN instruction for VBC1 is shown in Table P15.5 (on the next page). Explain why the actual values of the control bits *M*2 and *M*3 do not matter for the IN instruction. (Hint: Refer to the schematic in Section 15.3, Figure 15.2.)

	IR								Control bits					
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	*M***1**	*M***2**	*M***3**	*LOAD_R***0**	*LOAD_R***1**	*LOAD_OP*
IN R0	1	0	1	0	0	0	0	0	1	0*	0*	1	0	0
IN R1	1	0	1	1	0	0	0	0	1	0*	0*	0	1	0

*Actual value does not matter

TABLE P15.5

15.6 List the instructions where the control bit *M*2 must be used based on the schematic in Section 15.3, Figure 15.2.

15.7 Write the transfer function form for the IN DR instruction for VBC1.

15.8 One way to write the Boolean equation for the control bit for *M*1 in Table P15.5 is *M*1 = 1. List three different ways of writing the Boolean equation for the control bit for *M*1 with a minimum number of bits in terms of the inputs *IR*(7:5).

15.9 The instruction decoder truth table for the OUT instruction for VBC1 is shown in Table P15.9. Explain why the control bit *LOAD_OP* must be set to 1 for the OUT instruction. (Hint: Refer to the schematic in Section 15.3, Figure 15.2.)

	IR								Control bits					
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	*M***1**	*M***2**	*M***3**	*LOAD_R***0**	*LOAD_R***1**	*LOAD_OP*
OUT R0	1	1	0	0	0	0	0	0	0*	0	0*	0	0	1
OUT R1	1	1	0	1	0	0	0	0	0*	1	0*	0	0	1

*Actual value does not matter

TABLE P15.9

15.10 In the instruction decoder truth table for the OUT instruction for VBC1 shown in Table P15.9, explain why *LOAD_R*0 and *LOAD_R*1 must be set to 0.

15.11 Write the transfer function form for the OUT DR instruction for VBC1.

15.12 In the instruction decoder truth table for the OUT instruction for VBC1 shown in Table P15.9, explain why *M*2 has the value 0 and also 1. (Hint: Refer to the schematic in Section 15.3, Figure 15.2.)

15.13 In the instruction decoder truth table for the OUT instruction for VBC1 shown in Table P15.9, explain why the actual value of the control signal *M*3 does not matter. (Hint: Refer to the schematic in Section 15.3, Figure 15.2.)

15.14 In the instruction decoder truth table for the MOV instruction for VBC1 shown in Table P15.14, explain why *M*1 has the value 0. (Hint: Refer to the schematic in Section 15.3, Figure 15.2.)

	IR								Control bits					
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	*M***1**	*M***2**	*M***3**	*LOAD_R***0**	*LOAD_R***1**	*LOAD_OP*
MOV R0,R0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
MOV R0,R1	0	0	0	0	1	0	0	0	0	1	0	1	0	0
MOV R1,R0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
MOV R1,R1	0	0	0	1	1	0	0	0	0	1	0	0	1	0

TABLE P15.14

15.15 In the instruction decoder truth table for the MOV instruction for VBC1 shown in Table P15.14, explain why *LOAD_R*1 has the value 0 and also 1. (Hint: Refer to the schematic in Section 15.3, Figure 15.2.)

15.16 Write the transfer function form for the MOV DR,SR instruction for VBC1.

15.17 In the instruction decoder truth table for the MOV instruction for VBC1 shown in Table P15.14, explain why *M*2 has the value of 0 and also 1. (Hint: Refer to the schematic in Section 15.3, Figure 15.2.)

Section 15.4 Designing an Instruction Decoder for the IN Instruction

15.18 List the steps for Procedure ID, which we use to design an instruction decoder.

Section 15.5 Designing an Instruction Decoder for the OUT and MOV Instructions

15.19 Show a partial VHDL design for an instruction decoder for the OUT instruction in Table P15.9 for VBC1 using Procedure ID.

15.20 Show a partial VHDL design for an instruction decoder for the MOV instruction in Table P15.14 for VBC1 using Procedure ID.

15.21 Why are default instruction decoder output values placed before the case statement when writing the VHDL code for the instruction decoder?

Section 15.6 Instruction Decoder Truth Table for the LOADI Instruction

15.22 In the instruction decoder truth table for the LOADI instruction for VBC1 shown in Table P15.22, explain why *M*3 has the value 1. (Hint: Refer to the schematic in Section 15.6, Figure 15.3.)

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***M*2**	***M*3**	***M*4**	***M*5**	***LOAD_R*0**	***LOAD_R*1**	***LOAD_OP***
LOADI R0,Data	0	0	1	0	D	D	D	D	0	0*	1	0*	1	1	0	0
LOADI R1,Data	0	0	1	1	D	D	D	D	0	0*	1	0*	1	0	1	0

*Actual value does not matter

TABLE P15.22

15.23 Write the transfer function form for the LOADI DR,Data instruction for VBC1.

15.24 What is the value of control bit *M*5 for the LOADI instruction for VBC1 shown in Table P15.22? Explain your answer. (Hint: Refer to the schematic in Section 15.6, Figure 15.3.)

15.25 One way to write the Boolean equations for the control bits for *M*3 and *M*5 in Table P15.22 is *M*3 = 1 and *M*5 = 1, respectively. List a different way to write the Boolean equations for the control bits for *M*3 and *M*5 with a minimum number of bits in terms of the inputs *IR*(7:5).

15.26 Show a partial VHDL design for an instruction decoder for the LOADI instruction in Table P15.22 for VBC1 using a Procedure ID.

Section 15.7 Instruction Decoder Truth Table for the ADDI Instruction

15.27 In the instruction decoder truth table for the ADDI instruction for VBC1 shown in Table P15.27, write the Boolean equations for all the control bits with a minimum number of bits in terms of the inputs *IR*(7:4).

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***M*2**	***M*3**	***M*4**	***M*5**	***LOAD_R*0**	***LOAD_R*1**	***LOAD_OP***
ADDI R0,Data	0	1	1	0	D	D	D	D	0	0	1	0*	1	1	0	0
ADDI R1,Data	0	1	1	1	D	D	D	D	0	1	1	0*	1	0	1	0

*Actual value does not matter

TABLE P15.27

15.28 Write the transfer function form for the ADDI DR,Data instruction for VBC1.

15.29 In Tables P15.22 and P15.27 for the LOADI and ADDI instructions, the immediate data are shown as DDDD. What signal does DDDD represent for both the LOADI instruction and the ADDI instruction, and where does this immediate data come from?

15.30 In Tables P15.22 and P15.27 for the LOADI and ADDI instructions, the value of the control bit *M*4 does not matter. Explain why this is true. (Hint: Refer to the schematic in Section 15.6, Figure 15.3.)

Section 15.8 Instruction Decoder Truth Table for the ADD Instruction

15.31 In the instruction decoder truth table for the ADD instruction for VBC1 shown in Table P15.31, write the Boolean equation six different ways for the control bit *M*3 with a minimum number of bits in terms of the inputs *IR*(7:0).

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***M*2**	***M*3**	***M*4**	***M*5**	*LOAD_R***0**	*LOAD_R***1**	*LOAD_OP*
ADD R0,R0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0
ADD R0,R1	0	1	0	0	1	0	0	0	0	0	1	1	0	1	0	0
ADD R1,R0	0	1	0	1	0	0	0	0	0	1	1	0	0	0	1	0
ADD R1,R1	0	1	0	1	1	0	0	0	0	1	1	1	0	0	1	0

TABLE P15.31

15.32 Write the transfer function form for the ADD DR,SR instruction for VBC1.

15.33 Write a partial VHDL design for an instruction decoder for the ADD instruction in Table P15.31 for VBC1 using Procedure ID.

15.34 Show a partial VHDL design for an instruction decoder with one process with a case statement for the LOADI, ADDI, and ADD instructions in Tables P15.22, P15.27, and P15.31 for VBC1 using Procedure ID.

Section 15.9 Instruction Decoder Truth Table for the SR0 Instruction

15.35 In the instruction decoder truth table for the SR0 instruction for VBC1 shown in Table P15.35, write the Boolean equation six different ways for the control bit *M*5 with a minimum number of bits in terms of the inputs *IR*(7:0).

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***M*2**	***M*3**	***M*4**	***M*5**	*LOAD_R***0**	*LOAD_R***1**	*LOAD_OP*
SR0 R0,R0	1	0	0	0	0	0	0	0	0	0*	1	0	0	1	0	0
SR0 R0,R1	1	0	0	0	1	0	0	0	0	0*	1	1	0	1	0	0
SR0 R1,R0	1	0	0	1	0	0	0	0	0	0*	1	0	0	0	1	0
SR0 R1,R1	1	0	0	1	1	0	0	0	0	0*	1	1	0	0	1	0

*Actual value does not matter

TABLE P15.35

15.36 Write the transfer function form for the SR0 DR,SR instruction for VBC1.

15.37 A different version of an instruction decoder truth table is shown in Table P15.37. Write simple Boolean equations for the control bits for *M*1, *M*2, and *M*3.

	IR								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***M*2**	***M*3**	***M*4**	***M*5**	*LOAD_R***0**	*LOAD_R***1**	*LOAD_OP*
SRO R0,R0	1	0	0	0	0	0	0	0	0	0	1	0*	0*	1	0	0
SR0 R0,R1	1	0	0	0	1	0	0	0	0	1	1	0*	0*	1	0	0
SR0 R1,R0	1	0	0	1	0	0	0	0	0	0	1	0*	0*	0	1	0
SR0 R1,R1	1	0	0	1	1	0	0	0	0	1	1	0*	0*	0	1	0

*Actual value does not matter

TABLE P15.37

Section 15.10 Designing an Instruction Decoder for the SR0 Instruction

15.38 Write a partial VHDL design for an instruction decoder for the SR0 instruction in Table P15.37 for VBC1 using Procedure ID.

Section 15.11 Instruction Decoder Truth Table for the JNZ Instruction

15.39 For the partial schematic of the control circuit for VBC1 shown in Figure 15.4, what is the value of the control bit *M*6 that causes the JNZ instruction JNZ DR,Address to fall through or simply execute the next instruction? Write an expression for the address for the JNZ instruction when DR = 0. Explain how the address for instruction memory for the JNZ instruction is generated via Figure 15.4 when DR = 0.

15.40 For the partial schematic for VBC1 shown in Figure 15.4, what is the value of the control bit *M*6 that causes the JNZ instruction JNZ DR,Address to jump to the address specified in the instruction? Write an expression for the address for the JNZ instruction when DR ≠ 0. Explain how the address for instruction memory for the JNZ instruction is generated via Figure 15.4 when DR ≠ 0.

15.41 Write the transfer function form for the JNZ DR,Address instruction for VBC1.

15.42 In order to execute the JNZ instruction, we must design circuits for VBC1 that sample both registers R0 and R1 to determine when their values are not 0000 or are 0000. What is the name of a circuit that performs this type of operation?

15.43 Show a comparator circuit design that will compare the output of an 8-bit register R(7:0) to see if the register's output is 0. Let *Z* be the output of the comparator; *Z* = 1 indicates that the register's output is 0, and *Z* = 0 indicates that the register's output is ≠ 0. Show a reduced circuit for the final design and its Boolean equation.

Section 15.12 Designing an Instruction Decoder for the JNZ Instruction

15.44 Listing P15.44 shows an incomplete partial VHDL design for an instruction decoder for the JNZ instruction. Here, *Z*0 is a flag that indicates that *R*0 is 0000—that is, *Z*0 = 1 when *R*0 = 0000 else *Z*0 = 0; *Z*1 is a flag that indicates that *R*1 is 0000—that is, *Z*1 = 1 when *R*1 = 0000 else *Z*1 = 0. Write the simple signal assignment statements for *Z*0 and *Z*1 that must be added to the code for it to work correctly for VBC1.

```
process (ir, r0, r1)
begin
   m1 <= '0'; m2 <= '0'; m3 <= '0'; m4 <= '0'; m5 <= '0'; m6 <= '0';
   load_r0 <= '0'; load_r1 <= '0'; load_op <= '0';
   case ir (7 downto 5) is
      when "111" => z0 <=
         z1 <=
         if ir(4) = '0' then if z0 = '0' then m6 <= '1';     --jump
                             else m6 <= '0';                 --fall through
                             end if;
         elsif ir(4) = '1' then if z1 = '0' then m6 <= '1'; --jump
                                else m6 <= '0';              --fall through
                                end if;
         end if;
      when others => null;
   end case;
end process;
```

LISTING P15.44

15.45 What do the if statements that contain the control bit *M*6 establish in the code in Listing P15.44?

16 Chapter

Designing Arithmetic Logic Units

Chapter Outline

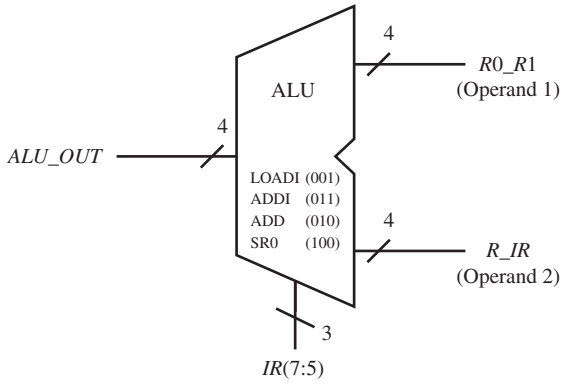
16.1 INTRODUCTION

In this chapter, you will learn about the arithmetic logic unit (ALU), how to obtain both arithmetic and logic functions, and how to write VHDL code for an arithmetic logic unit for VBC1. You will also learn how to design an ALU with a carry-out output, shifter circuits, barrel shifter circuits, and shift register circuits in VHDL.

16.2 UTILIZATION OF THE ARITHMETIC LOGIC UNIT

The arithmetic logic unit is a combinational logic circuit that provides both arithmetic and logic functions for a computer. Figure 16.1 shows a logic symbol for the arithmetic logic unit for VBC1.

FIGURE 16.1
Logic Symbol for the ALU for VBC1



Observe that the operands and the output of the ALU in Figure 16.1 have a data path of only 4 bits. The inputs to the arithmetic logic unit are provided by *R*0_*R*1 (operand 1), *R_IR* (operand 2), and *IR*(7:5). The instruction that the ALU performs is determined by the OPCODE for the instruction—that is, bits 7:5 in the IR. The output of the arithmetic logic unit is *ALU_OUT*. The ALU for VBC1 is rather simple because it only has two arithmetic functions (ADDI and ADD) and two logic functions (LOADI and SR0).

Figure 16.2 shows an annotated schematic for the complete data path circuit for VBC1.

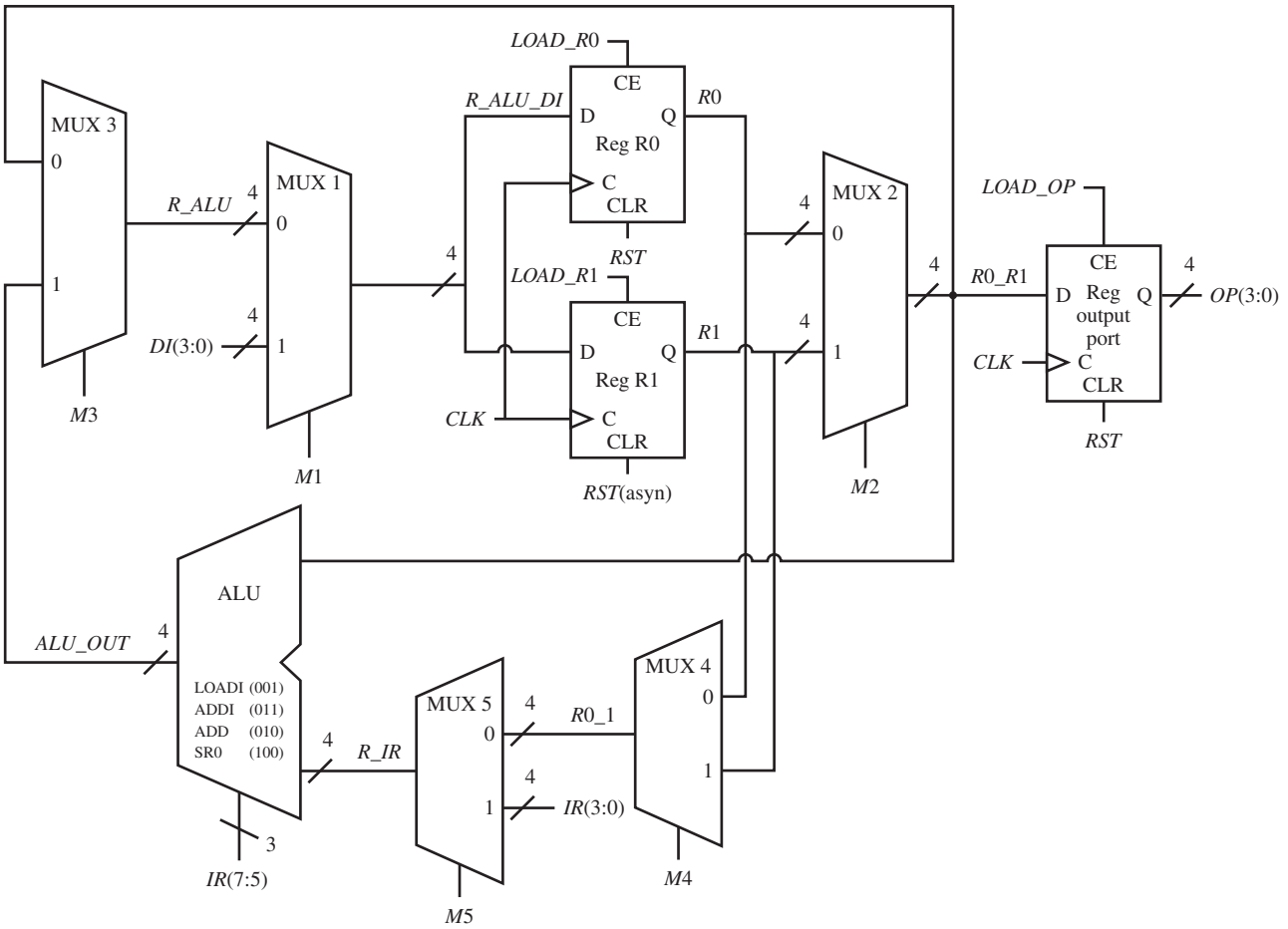


FIGURE 16.2 Annotated schematic for the complete data path circuit for VBC1

The arithmetic function ADDI uses data supplied by operand 1 (the contents of a register) and data supplied by operand 2 (the contents of immediate data in the ADDI instruction) to obtain the ALU output. The arithmetic function ADD uses data supplied by operand 1 (the contents of a register) and data supplied by operand 2 (also the contents of a register) to obtain the ALU output. The logic function LOADI uses data supplied by operand 2 (the contents of immediate data in the LOADI instruction) to obtain the ALU output. The logic function SR0 uses data supplied by operand 2 (the contents of a register) to obtain the ALU output.

16.3 DESIGNING THE LOADI INSTRUCTION PART OF THE ALU

The transfer function form for the LOADI DR,Data instruction is DR ← IR(3:0). The function LOADI uses data supplied by operand 2 (the contents of immediate data in the LOADI

instruction) to obtain the ALU output. Observe that the right side of the transfer function form for the LOADI instruction indicates which signal the ALU output will receive: *ALU_OUT* ← *R_IR*. *R_IR* receives the signal *IR*(3:0), and destination register *R*0 or *R*1 stores the signal *ALU_OUT* during the LOADI instruction via the control bits supplied by the instruction decoder.

Listing 16.1 shows a partial VHDL design for an arithmetic logic unit for the LOADI instruction using a behavioral design style—that is, a process with a case statement.

LISTING 16.1 Partial VHDL design for an ALU for the LOADI instruction using a behavioral design style

```
alu_process:
process (ir(7 downto 5), r_ir)
begin
     alu_out <= "0000";   --default value to prevent creating
                          --inferred latches
     case ir(7 downto 5) is
          when "001" => alu_out <= r_ir;   --LOADI
          when others => null;
     end case;
end process alu_process;
```

Things you should notice about the partial VHDL design in Listing 16.1:

- A default arithmetic logic unit output value is listed for the **process** after **begin**. The default value is assigned before the case statement to ensure proper circuit operation—that is, so that inferred latches will not be created.
- The case statement evaluates the signal *IR*(7 **downto** 5) or the bits 7:5 in the IR that contain the OPCODE for the LOADI instruction.
- When the OPCODE 001 for the LOADI instruction is true, the assignment is made to the ALU output signal *ALU_OUT* following the symbol =>. For the LOADI instruction to be performed by the ALU, the ALU output *ALU_OUT* must be assigned the signal *R_IR*—that is, *ALU_OUT* <= *R_IR*.

16.4 DESIGNING THE ADDI INSTRUCTION PART OF THE ALU

The transfer function form for the ADDI DR,Data instruction is DR ← DR + IR(3:0). The function ADDI uses data supplied by operand 1 (the contents of a register) and data supplied by operand 2 (the contents of immediate data in the ADDI instruction) to obtain the ALU output. Observe that the right side of the transfer function form for the ADDI instruction indicates which signals the ALU output will receive: *ALU_OUT* ← *R*0_*R*1 + *R_IR*. *R*0_*R*1 receives the signal *R*0 or *R*1, *R_IR* receives the signal *IR*(3:0), and destination register *R*0 or *R*1 stores the signal *ALU_OUT* during the ADDI instruction via the control bits supplied by the instruction decoder.

Listing 16.2 shows a partial VHDL design for an arithmetic logic unit for the ADDI instruction using a behavioral design style—that is, a process with a case statement.

LISTING 16.2 Partial VHDL design for an ALU for the ADDI instruction using a behavioral design style

```
alu_process:
process (ir(7 downto 5), r0_r1, r_ir)
begin
     alu_out <= "0000";  --default value to prevent creating
                         --inferred latches
     case ir(7 downto 5) is
          when "011" => alu_out <= r0_r1 + r_ir;  --ADDI
          when others => null;
     end case;
end process;
```

Things you should notice about the partial VHDL design in Listing 16.2:

- The case statement evaluates the signal *IR*(7 **downto** 5) or the bits 7:5 in the IR that contain the OPCODE for the ADDI instruction.
- When the OPCODE 011 for the ADDI instruction is true, the assignment is made to the ALU output signal *ALU_OUT* following the symbol =>. For the ADDI instruction to be performed by the ALU, the ALU output *ALU_OUT* must be assigned the expression *R*0_*R*1 + *R_IR*—that is, *ALU_OUT* ← *R*0_*R*1 + *R_IR*.

16.5 DESIGNING THE ADD INSTRUCTION PART OF THE ALU

The transfer function form for the ADD DR,SR instruction is DR ← DR + SR. The function ADD uses data supplied by operand 1 (the contents of a register) and data supplied by operand 2 (also the contents of a register) to obtain the ALU output. Observe that the right side of the transfer function form for the ADD instruction indicates which signals the ALU output will receive: *ALU_OUT* ← *R*0_*R*1 + *R_IR*. *R*0_*R*1 receives the signal *R*0 or *R*1, *R_IR* receives the signal *R*0 or *R*1, and destination register *R*0 or *R*1 stores the signal *ALU_OUT* during the ADD instruction via the control bits supplied by the instruction decoder.

Listing 16.3 shows a partial VHDL design for an arithmetic logic unit for the ADD instruction using a behavioral design style—that is, a process with a case statement.

LISTING 16.3 Partial VHDL design for an ALU for the ADD instruction using a behavioral design style

```
alu_process:
process (ir(7 downto 5), r0_r1, r_ir)
begin
     alu_out <= "0000";  --default value to prevent creating
                         --inferred latches
     case ir(7 downto 5) is
          when "010" => alu_out <= r0_r1 + r_ir;  --ADD
          when others => null;
     end case;
end process;
```

Things you should notice about the partial VHDL design in Listing 16.3:

- The case statement evaluates the signal *IR*(7 **downto** 5) or the bits 7:5 in the IR that contain the OPCODE for the ADD instruction.
- When the OPCODE 010 for the ADD instruction is true, the assignment is made to the ALU output signal *ALU_OUT* following the symbol =>. For the ADD instruction to be performed by the ALU, the ALU output *ALU_OUT* must be assigned the expression *R*0_*R*1 + *R_IR*—that is, *ALU_OUT* ← *R*0_*R*1 + *R_IR*.

16.6 DESIGNING THE SR0 INSTRUCTION PART OF THE ALU

The transfer function form for the SR0 DR,SR instruction is DR ← 0 SR(3:1). The logic function SR0 uses data supplied by operand 2 (the contents of the source register) to obtain the ALU output. Observe that the right side of the transfer function form for the SR0 instruction indicates which signal the ALU output will receive: *ALU_OUT* (3) ← 0, *ALU_OUT*(2:0) ← *R_IR*(3:1). *R_IR* receives the signal *R*0 or *R*1 and destination register *R*0 or *R*1 stores the signal *ALU_OUT* during the SR0 instruction via the control bits supplied by the instruction decoder for the SR0 instruction.

Listing 16.4 shows a partial VHDL design for an arithmetic logic unit for the SR0 instruction using a behavioral design style—that is, a process with a case statement.

```
alu_process:
process (ir(7 downto 5), r_ir)
begin
   alu_out <= "0000";  --default value to prevent creating inferred latches
   case ir(7 downto 5) is
      when "100" => alu_out(3) <= '0'; alu_out(2 downto 0) <= r_ir(3 downto 1); --SR0
    --when "100" => alu_out <= '0' & r_ir (3 downto 1);  --alternate statement
      when others => null;
   end case;
end process;
```

LISTING 16.4 Partial VHDL design for an ALU for the SR0 instruction using a behavioral design style

Things you should notice about the partial VHDL design in Listing 16.4:

- The case statement evaluates the signal *IR*(7 **downto** 5) or the bits 7:5 in the IR that contain the OPCODE for the SR0 instruction.
- When the OPCODE 100 for the SR0 instruction is true, the assignment is made to the ALU output signal *ALU_OUT* following the symbol =>. For the SR0 instruction to be performed by the ALU, the ALU output *ALU_OUT* must be assigned the expression 0 *R_IR*(3:1)—that is, *ALU_OUT* (3) ← 0, *ALU_OUT*(2:0) ← *R_IR*(3:1).
- The alternate statement simplifies the assignment for *ALU_OUT* by using the concatenation operator &.

16.7 DESIGNING AN ALU FOR VBC1

Listing 16.5 shows a general format or template for the ALU for VBC1.

```
process (ir(7 downto 5), r0_r1, r_ir)
begin
   alu_out <= "0000";  --default value to prevent creating inferred latches
   case ir(7 downto 5) is  --the OPCODE for each ALU instruction is in bits (7:5) in
                           --the IR

      --the LOADI instruction has the OPCODE 001
      when "001" =>  --enter the ALU output equation for the LOADI instruction here

      --the ADDI instruction has the OPCODE 011
      when "011" =>  --enter the ALU output equation for the ADDI instruction here

      --the ADD instruction has the OPCODE 010
      when "010" =>  --enter the ALU output equation for the ADD instruction here

      --the SR0 instruction has the OPCODE 100
      when "100" =>  --enter the ALU output equation for the SR0 instruction here
      when others => null;
   end case;
end process;
```

LISTING 16.5 General format or template for the ALU for VBC1

16.8 ADDITIONAL CIRCUIT DESIGNS WITH VHDL

The following circuits are presented to give you a better understanding of some additional computer circuits that are a little more complex. The circuits are designed with VHDL and simulated to show that they are functionally correct. The circuits in this section are not used in the design of VBC1.

16.8.1 Designing Additional ALU Circuits

An ALU performs as follows: (1) it has **word-wise** arithmetic operations such as ADD (ADDition), INC (INCrement), SUB (SUBtraction), and DEC (DECrement), and (2) it has **bit-wise** logic operations such as AND, OR, and NAND. For word-wise operations, each bit position has a carry-out bit (or borrow bit) that must be used to accomplish the arithmetic operation.

For bit-wise operations, there are no carry-out bits (or borrow bits) that must be used to accomplish the operations, so the final carry-out bit for the overall result (or borrow bit) should be ignored.

Table 16.1 shows a summary of the operator symbols that we use throughout this chapter in transfer functions.

TABLE 16.1 Operator symbols

Operation	Operator symbol	Comment
Addition	+	Word-wise operation
Subtraction	−	Word-wise operation
AND	$\wedge$	Bit-wise operation
OR	$\vee$	Bit-wise operation
XOR	$\oplus$	Bit-wise operation
XNOR	$\overline{\oplus}$	Bit-wise operation

An **addition circuit** is represented by the following transfer function:

$$BC \leftarrow BA + BB$$

Things you should know about the transfer function we use for the addition circuit:

- Bus B (*BB*) is added to bus A (*BA*) and the result is transferred to bus C (*BC*).
- Addition is a word-wise operation.
- A carry-out bit may be generated for the MSB (most significant bit) of the result—that is, signal *BC*.
- This is a combinational logic circuit because there is no clock input.

An **increment circuit** is represented by the following transfer function:

$$BC \leftarrow BA + 1$$

Things you should know about the transfer function we use for the increment circuit:

- A 1 is added to bus A (*BA*) and the result is transferred to bus C (*BC*).
- Increment is a word-wise operation.
- A carry-out bit may be generated for the MSB of the result—that is, signal *BC*.
- This is a combinational logic circuit because there is no clock input.

An **AND circuit** is represented by the following transfer function:

$$BC \leftarrow BA \wedge BB$$

Things you should know about the transfer function we use for the AND circuit:

- Bus B (*BB*) is ANDed with bus A (*BA*) and the result is transferred to bus C (*BC*).
- AND is a bit-wise operation.
- No carry-out bit can be generated for the MSB of the result—that is, signal *BC*.
- This is a combinational logic circuit because there is no clock input.

An **XOR circuit** is represented by the following transfer function:

$$BC \leftarrow BA \oplus BB$$

Things you should know about the transfer function we use for the XOR circuit:

- Bus B (*BB*) is XORed with bus A (*BA*) and the result is transferred to bus C (*BC*).
- XOR is a bit-wise operation.
- No carry-out bit can be generated for the MSB of the result—that is, signal *BC*.
- This is a combinational logic circuit because there is no clock input.

A function table for an ALU that uses the four transfer functions just discussed is shown in Table 16.2.

TABLE 16.2 Function table for an ALU

S(1:0)	Output: *BC*(1:0)	Comment
00	$BC \leftarrow BA + BB$	Arithmetic operation (ADD), *BC*(1) is the carry-out bit
01	$BC \leftarrow BA + 1$	Arithmetic operation (INC), *BC*(1) is the carry-out bit
10	$BC \leftarrow BA \wedge BB$	Logic operation (AND), ignore carry-out bit *BC*(1)
11	$BC \leftarrow BA \oplus BB$	Logic operation (XOR), ignore carry-out bit *BC*(1)

When *S*(1:0) = 00, ADD is performed; when *S*(1:0) = 01, INC is performed; when *S*(1:0) = 10, AND is performed; and when *S*(1:0) = 11, XOR is performed.

Figure 16.3 shows a logic symbol for the ALU in Table 16.2 with a word size of 1 bit.

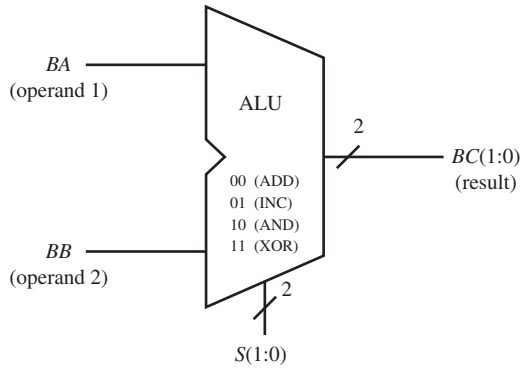


FIGURE 16.3 Logic symbol for the ALU in Table 16.2 with a word size of 1 bit

Listing 16.6 shows the VHDL code for a 1-bit version of the ALU in Figure 16.3 using a dataflow architecture declaration with a conditional signal assignment (CSA).

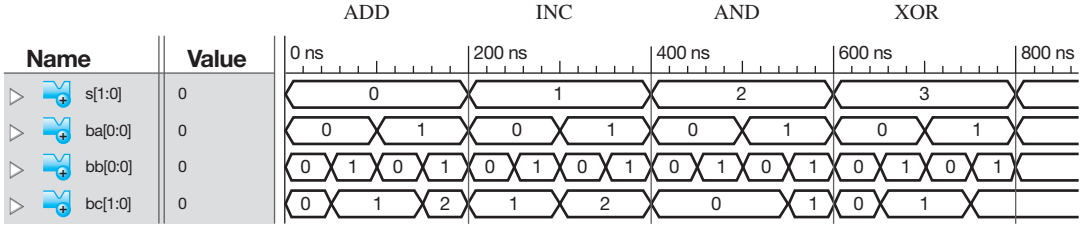
LISTING 16.6
Complete VHDL design entity for a 1-bit version of the ALU in Table 16.2 (project: ALU_wcarry_bit)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ALU_wcarry_bit is port (
     s : in std_logic_vector (1 downto 0);
     ba, bb : in std_logic_vector (0 downto 0);
     bc : out std_logic_vector (1 downto 0)
     );
end ALU_wcarry_bit;

architecture dataflow of ALU_wcarry_bit is
begin
     bc <= ('0'& ba) + bb     when s = "00" else
           ('0'& ba) + 1      when s = "01" else
           '0'& (ba and bb)   when s = "10" else
           '0'& (ba xor bb);
end dataflow;
```

Things you should notice about the VHDL design in Listing 16.6:

- Observe that the buses *BA* and *BB* are 1-bit buses represented by the data type std_logic_vector (0 **downto** 0). These 1-bit buses could also be represented by the data type std_logic. Using vector data types allows the code to be easily changed to provide wider buses.
- The ampersand symbol "&" is a concatenation operator. The concatenation operator is used to connect or link signals together. With this operator, we have padded bus *BA* to extend its width. The padding or extension is on the left side to allow for the carry-out bit during the addition operation. There is no carry-out bit for the AND and the XOR operations, because a carry-out bit cannot be generated for bit-wise operations.
- Bus *BC*(1:0) contains the sum in bit position 0 and carry-out bit in bit position 1.
- This is a combinational logic circuit because there is no clock input.

Waveform 16.1 shows a simulation with the correct functionality of design entity ALU_wcarry_bit.



WAVEFORM 16.1 Simulation for the correct functionality of design entity ALU_wcarry_bit

Things you should notice about the waveforms in Waveform 16.1:

- For the section of Waveform 16.1 marked ADD where $S(1:0) = 0$ (in decimal), observe that the output or result for $BC(1:0)$ (in decimal) is $BA + BB$.
- For the section of Waveform 16.1 marked INC where $S(1:0) = 1$ (in decimal), observe that the output or result for $BC(1:0)$ (in decimal) is $BA + 1$.
- For the section of Waveform 16.1 marked AND where $S(1:0) = 2$ (in decimal), observe that the output or result for $BC(1:0)$ (in decimal) is $BA \wedge BB$.
- For the section of Waveform 16.1 marked XOR where $S(1:0) = 3$ (in decimal), observe that the output or result for $BC(1:0)$ (in decimal) is $BA \oplus BB$.

An ALU with a larger word size can be easily designed by simply changing the vector ranges of the signals BA, BB, and BC in the entity declaration. To increase the number of bits in the word size of the design of the ALU in Listing 16.6 from 1 bit to 4 bits, simply change the range of the vector for BA and BB from (0 **downto** 0) to (3 **downto** 0) and change the range of the vector for BC from (1 **downto** 0) to (4 **downto** 0).

16.8.2 Designing Shifter Circuits

A function table for a **shifter circuit** that uses two transfer functions is shown in Table 16.3.

TABLE 16.3 Function table for a shifter circuit with a 0 fill bit

MODE	**Output:** *BC*	**Comment**
0	*BC* ← shift right *BA* w0 *fill*	Shift right operation with a fill bit of 0 for MSB
1	*BC* ← shift left *BA* w0 *fill*	Shift left operation with a fill bit of 0 for LSB

MODE = 0 is a shift-right operation with a 0 fill bit. If *BA* has a bus width of 4 bits and contains 1111, then after the shift-right operation *BC* contains 0111—that is, a 0 fill bit is added to the MSB (most significant bit). *MODE* = 1 is a shift-left operation with a 0 fill bit. Assuming *BA* has a bus width of 4 bits and contains 1111, then after the shift-left operation, *BC* contains 1110, i.e., a 0 fill bit is added to the LSB (Least Significant Bit). Notice that a shift-right operation discards the LSB, and a shift-left operation discards the MSB in this shifter circuit.

Figure 16.4 shows a logic symbol for a 4-bit version of the shifter circuit with a 0 fill bit in Table 16.3.

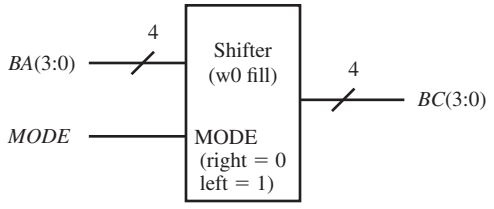


FIGURE 16.4 Logic symbol for a 4-bit version of the shifter circuit with a 0 fill bit in Table 16.3

For this design, we will use the VHDL shift operators **srl** and **sll** (see Chapter 4, Figure 4.10 for a list of supported operators.) To use the VHDL shift operators, here is the information you need to know: *b* **srl** *i* means to shift right logical the bit_vector *b*, *i* bit positions where *i* is an integer. The result is a bit_vector. The declaration *b*: bit_vector (*n* −1 downto 0) must be placed in the entity, where *n* is the width of the vector *b*. Only 0 fill bits are used for the srl operator. The sll operator has the same requirements only it shifts left logical with 0 fill bits.

Listing 16.7 shows the VHDL code for a Shifter Circuit with a 0 fill bit using a dataflow architecture declaration with a selected signal assignment (SSA).

LISTING 16.7 Complete VHDL design entity for the shifter circuit with a 0 fill bit (project: Shifter_W0F)

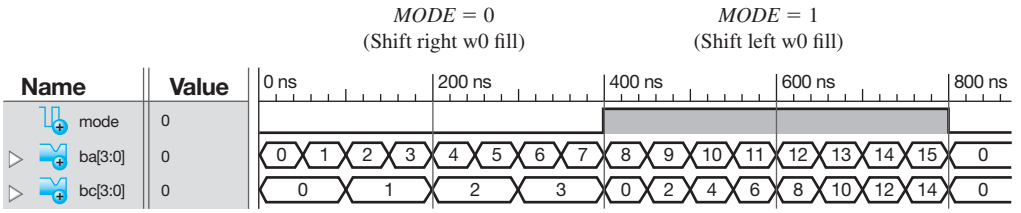
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Shifter_W0F is port (
    mode : in std_logic;
    ba : in bit_vector (3 downto 0);
    bc : out bit_vector (3 downto 0)
    );
end Shifter_W0F;
```

```
architecture dataflow of Shifter_W0F is
begin
with mode select
     bc <= ba srl 1 when '0',
           ba sll 1 when '1',
           ba       when others;
end dataflow;
```

Things you should notice about the VHDL design in Listing 16.7:

- The signal *MODE* in the entity has a data type of std_logic. It could have a data type of bit.
- The signals *BA* and *BC* must be data type bit_vector with the range 3 **downto** 0 for a 4-bit bus because the VHDL operators srl and sll require these data types.
- A 1 is used for the integer *i* for a single-bit shift.
- The shifter circuit is bidirectional because it shifts one bit position to the right with a 0 fill bit and also shifts one bit position to the left with a 0 fill bit.
- This is a combinational logic circuit because there is no clock input.

Waveform 16.2 shows a simulation with the correct functionality of design entity Shifter_W0F.



WAVEFORM 16.2 Simulation for the correct functionality of design entity Shifter_W0F

Things you should notice about the waveforms in Waveform 16.2:

- For the section of Waveform 16.2 marked *MODE* = 0 (shift right w0 fill), observe that the output or result for *BC*(3:0) (in decimal) is *shift right BA*(3:0) *w0 fill* in each case for *BA*(3:0) 0 through 7 (in decimal).
- For the section of Waveform 16.2 marked *MODE* = 1 (shift left w0 fill), observe that the output or result for *BC*(3:0) (in decimal) is *shift left BA*(3:0) *w0 fill* in each case for *BA*(3:0) 8 through 15 (in decimal).
- To generate Waveform 16.2, be sure to change the VHDL test bench code to data type bit_vector for signals ba(3:0) and bc(3:0), if the test bench code is set up to use data type std_logic_vector for all signals.

The VHDL operators srl and sll will work just fine for some designs. Remember that you must use a data type of bit_vector for these operators. Both operators srl and sll only provide a fill bit of 0. The following approach allows you to use the data type of std_logic_vector to create designs that provide a fill bit of 0 or a fill bit of 1 as illustrated in Table 16.4.

TABLE 16.4 Function table for a shifter circuit with a 1-fill bit

MODE	**Output:** *BC*	**Comment**
0	*BC* ← shift right *BA* *w*1 *fill*	Shift-right operation with a fill bit of 1 for MSB
1	*BC* ← shift left *BA* *w*1 *fill*	Shift-left operation with a fill bit of 1 for LSB

Figure 16.5 shows a logic symbol for a 4-bit version of the shifter circuit with a 1-fill bit in Table 16.4.

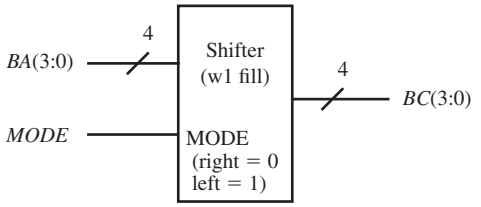


FIGURE 16.5 Logic symbol for a 4-bit version of the shifter circuit with a 1-fill bit in Table 16.4

Listing 16.8 shows the VHDL code for a shifter circuit with a 1-fill bit using a behavioral architecture declaration with a case statement.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Shifter_W1F is port (
   mode : in std_logic;
   ba: in std_logic_vector (3 downto 0);
   bc: out std_logic_vector (3 downto 0)
   );
end Shifter_W1F;
architecture behavioral of Shifter_W1F is

begin
process (mode, ba)
begin
   case mode is
       --this is a right shift with 1 fill
         when '0' => bc(3)<='1'; bc(2)<=ba(3); bc(1)<=ba(2); bc(0)<=ba(1);
       --this is a left shift with 1 fill
         when '1' => bc(3)<=ba(2); bc(2)<=ba(1); bc(1)<=ba(0); bc(0)<='1';
      when others => null;
   end case;
end process;
end behavioral;
```

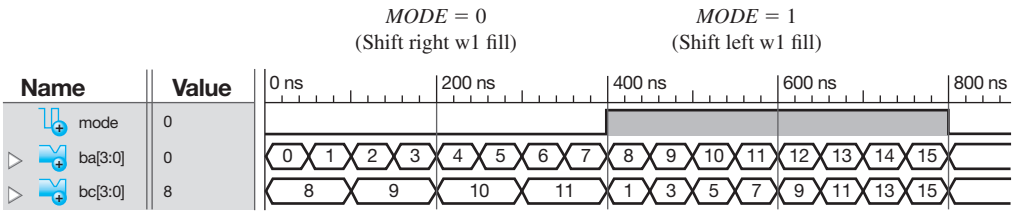
LISTING 16.8 Complete VHDL design entity for the shifter circuit with a 1-fill bit (project: Shifter_W1F)

Things you should notice about the VHDL design in Listing 16.7:

- All the signals in the design are standard logic data types.
- Alternate assignments for a right shift with 1 fill are bc(3) <= '1'; bc(2 **downto** 0) <= ba(3 **downto** 1); and bc <= '1' & ba(3 **downto** 1).
- Alternate assignments for a left shift with 1 fill are bc(3 **downto** 1) <= ba(2 **downto** 0); bc(0)<='1'; and bc <= ba(2 **downto** 0) & '1'.

Waveform 16.3 shows a simulation with the correct functionality of design entity Shifter_W1F.

WAVEFORM 16.3 Simulation for the correct functionality of design entity Shifter_W1F



Things you should notice about the waveforms in Waveform 16.3:

- For the section of Waveform 16.3 marked *MODE* = 0 (shift right w1 fill) where *MODE* = 0, observe that the output or result for *BC*(3:0) (in decimal) is *shift right BA*(3:0) *w*1 *fill* in each case for *BA*(3:0) 0 through 7 (in decimal).
- For the section Waveform 16.3 marked *MODE* = 1 (shift left w1 fill) where *MODE* = 1, observe that the output or result for *BC*(3:0) (in decimal) is *shift left BA*(3:0) *w*1 *fill* in each case for *BA*(3:0) 8 through 15 (in decimal).

The designs we show for shifter circuits only shift one bit position either right or left. By adding select inputs to the design, we could design a shifter circuit that would shift the data input *BA*(3:0) multiple bit positions based on the select inputs. For example, when *S*(1:0) = 00, the 4-bit data input gets transferred to the output with no shift; when *S*(1:0) = 01, the 4-bit data input gets transferred to the output shifted by 1 bit position; when *S*(1:0) = 10, the data input gets transferred to the output shifted by 2 bit positions; and when *S*(1:0) = 11, the data input gets transferred to the output shifted by 3 bit positions. The direction for the shift can be fixed or be dependent on the *MODE*—that is, right or left. The fill bits can be a 0, a 1, or supplied by an additional input. As you can see, shifter circuits can be designed in various ways depending on the designer's needs.

16.8.3 Designing Barrel Shifter Circuits

Table 16.5 shows a function table for a 4-bit barrel shifter circuit with eight transfer functions.

TABLE 16.5 Function table for a 4-bit barrel shifter circuit

MODE	*S*(1:0)	Output: *BC*	Comment
0	00	*BC* ← rotate right *BA* 0 bits	No rotation
0	01	*BC* ← rotate right *BA* 1 bit	Rotate right 1 bit
0	10	*BC* ← rotate right *BA* 2 bits	Rotate right 2 bits
0	11	*BC* ← rotate right *BA* 3 bits	Rotate right 3 bits
1	00	*BC* ← rotate left *BA* 0 bits	No rotation
1	01	*BC* ← rotate left *BA* 1 bit	Rotate left 1 bit
1	10	*BC* ← rotate left *BA* 2 bits	Rotate left 2 bits
1	11	*BC* ← rotate left *BA* 3 bits	Rotate left 3 bits

A barrel shifter circuit is similar to the shifter circuit discussed earlier, with one major difference. The output of a barrel shifter circuit rotates the data bits right (LSB back to the MSB in a circle or around a barrel) or rotates the data bits left (MSB back to the LSB in a circle). The output of a shifter circuit simply shifts the data bits right or left and thus requires fill bits to fill vacant spaces after each shift. A barrel shifter circuit is commonly used as part of the circuitry for high-speed graphics hardware in the computer industry.

Figure 16.6 shows a logic symbol for the barrel shifter circuit in Table 16.5.

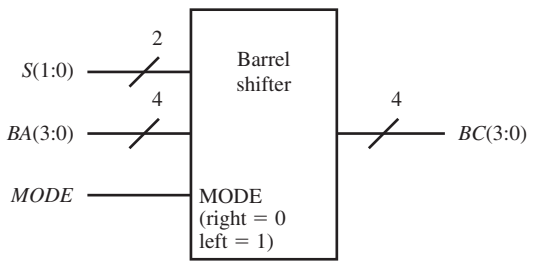


FIGURE 16.6 Logic symbol for the 4-bit barrel shifter circuit in Table 16.5

Listing 16.9 shows the VHDL code for a 4-bit barrel shifter circuit with standard logic vectors (SLVs) using a behavioral architecture declaration with a case statement.

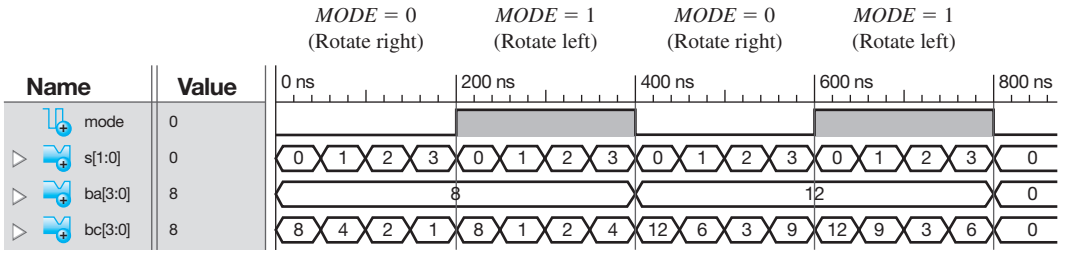
LISTING 16.9
Complete VHDL design entity for a 4-bit barrel shifter (project: Barrel_Shifter_SLVs)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Barrel_Shifter_SLVs is port (
    mode : in std_logic;
    s : in std_logic_vector (1 downto 0);
    ba: in std_logic_vector(3 downto 0);
    bc: out std_logic_vector(3 downto 0)
    );
end Barrel_Shifter_SLVs;

architecture behavioral of Barrel_Shifter_SLVs is
begin
process (mode, s, ba)
begin
    if mode = '0' then
        case s is
            when "00" => bc(3)<= ba(3); bc(2)<=ba(2);
                         bc(1)<=ba(1); bc(0)<=ba(0);
            when "01" => bc(3)<= ba(0); bc(2)<=ba(3);
                         bc(1)<=ba(2); bc(0)<=ba(1);
            when "10" => bc(3)<= ba(1); bc(2)<=ba(0);
                         bc(1)<=ba(3); bc(0)<=ba(2);
            when "11" => bc(3)<= ba(2); bc(2)<=ba(1);
                         bc(1)<=ba(0); bc(0)<=ba(3);
            when others => null;
        end case;
    else
        case s is
            when "00" => bc(3)<= ba(3); bc(2)<=ba(2);
                         bc(1)<=ba(1); bc(0)<=ba(0);
            when "01" => bc(3)<= ba(2); bc(2)<=ba(1);
                         bc(1)<=ba(0); bc(0)<=ba(3);
            when "10" => bc(3)<= ba(1); bc(2)<=ba(0);
                         bc(1)<=ba(3); bc(0)<=ba(2);
            when "11" => bc(3)<= ba(0); bc(2)<=ba(3);
                         bc(1)<=ba(2); bc(0)<=ba(1);
            when others => null;
        end case;
    end if;
end process;
end behavioral;
```

Waveform 16.4 shows a simulation with the correct functionality of design entity Barrel_Shifter_SLVs.

Things you should notice about the waveforms in Waveform 16.4:

- When *S*(1:0) is 0, no rotation occurs.
- Observe that output *BC*(3:0) provides the correct value in all cases for the specified inputs.



WAVEFORM 16.4 Simulation for the correct functionality of design entity Barrel_Shifter_SLVs

To simplify the VHDL code in Listing 16.9, you may elect to use the **ror** (rotate right) and **rol** (rotate left) operators. Listing 16.10 shows the VHDL code for a 4-bit barrel shifter circuit with bit vectors (BVs) using a behavioral architecture description with the ror and rol operators. To use these operators, you must define the data input signal *BA*(3:0) and the output signal *BC*(3:0) in the entity declaration as bit_vectors.

LISTING 16.10
Complete VHDL design entity for a 4-bit barrel shifter with the ror and rol operators (project: Barrel_Shifter_BVs)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Barrel_Shifter_BVs is port (
     mode : in std_logic;
     s : in std_logic_vector (1 downto 0);
     ba: in bit_vector (3 downto 0);
     bc: out bit_vector (3 downto 0)
     );
end Barrel_Shifter_BVs;

architecture behavioral of Barrel_Shifter_BVs is
begin
process (mode, s, ba)
begin
     if mode = '0' then
          case s is
               when "00" => bc <=ba ror 0;
               when "01" => bc <=ba ror 1;
               when "10" => bc <=ba ror 2;
               when "11" => bc <=ba ror 3;
               when others => null;
          end case;
     else
          case s is
               when "00" => bc <=ba rol 0;
               when "01" => bc <=ba rol 1;
               when "10" => bc <=ba rol 2;
               when "11" => bc <=ba rol 3;
               when others => null;
          end case;
     end if;
end process;
end behavioral;
```

The simulation for Listing 16.10 is the same as Waveform 16.4 obtained earlier. This proves that using standard logic vectors or bit vectors provide the same result. To generate the waveforms for Listing 16.10 using bit_vectors, be sure to change the VHDL test bench code to data type bit_vector for signals *BA*(3:0) and *BC*(3:0), if the test bench code is set up to use data type std_logic_vector for all signals.

In some applications, it may be desirable to provide an output enable (*OE*) input that allows the barrel shifter's output *BC* to be **tri-stated**. Barrel shifter circuits can also be designed with a larger number of data bits for the input and output. For a design like the one in Listing 16.10, 3 select inputs would be required for a barrel shifter circuit with 8 data bits, 4 select inputs for 16 data bits, and so on.

16.8.4 Designing Shift Register Circuits

A function table for a **shift register circuit** that uses four transfer functions is shown in Table 16.6.

TABLE 16.6 Function table for a shift register circuit

RST (sync)	*S*(1:0)	Output: *RC*	Comment
1	××	*RC* ← 0	Synchronous reset
0	00	*RC* ← *RC*	Hold operation
0	01	*RC* ← shift right *RC wSRSI fill*	Shift-right operation
0	10	*RC* ← shift left *RC wSLSI fill*	Shift-left operation
0	11	*RC* ← *BA*	Load operation

Register C (*RC*) is the output of a group of D flip-flops. *RST* is a synchronous reset—that is, the reset occurs (*RC* = 0) when *RST* = 1 after the next rising edge of the signal *CLK*. When *RST* = 0, the select signal *S* determines which operation occurs at the next rising edge of the signal *CLK*. *S*(1:0) = 00 is a hold operation that holds the current value in register C (*RC*). *S*(1:0) = 01 is a shift-right operation with an input signal ***SRSI*** **(shift-right serial input)** as a fill bit to the MSB of *RC*. *S*(1:0) = 10 is a shift-left operation with an input signal ***SLSI*** **(shift-left serial input)** as a fill bit to the LSB of *RC*. *S*(1:0) = 11 is a load operation that loads register C (*RC*) with new data from bus A (*BA*).

The shifter register circuit represented in Table 16.6 is sometimes called a universal shift register circuit because it is an all-purpose circuit that can be cleared (reset) and can also hold current data, shift current data to the right, shift current data to the left, or load new data. A shifter register circuit is a complex state machine because it has external inputs to change its state sequence.

Figure 16.7 shows a logic symbol for a 4-bit version of the shift register circuit in Table 16.6.

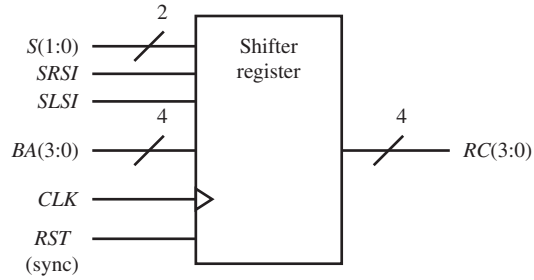


FIGURE 16.7 Logic symbol for a 4-bit version of the shifter register circuit in Table 16.6

Listing 16.11 shows the VHDL code for the 4-bit version of the shift register circuit in Table 16.6 using a behavioral architecture declaration with a single process, two if statements, and a case statement.

LISTING 16.11
Complete VHDL design entity for a shift register circuit with 4 bits (project: Shift_Register)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Shift_Register is port (
     rst, clk : in std_logic;
     s : in std_logic_vector (1 downto 0);
     ba: in std_logic_vector(3 downto 0);
     srsi, slsi : in std_logic;
     rc: inout std_logic_vector(3 downto 0)
     );
end Shift_Register;

architecture behavioral of Shift_Register is
begin
process (clk)
begin
     if rising_edge (clk) then
          if rst = '1' then rc <= "0000";
          else
               case s is
                    when "00" => rc <= rc;
                    when "01" => rc(3)<= srsi; rc(2)<= rc(3);
                                 rc(1)<= rc(2); rc(0)<= rc(1);
                    when "10" => rc(3)<= rc(2); rc(2)<= rc(1);
                                 rc(1)<= rc(0); rc(0)<= slsi;
                    when "11" => rc <= ba;
                    when others => null;
               end case;
          end if;
     end if;
end process;
end behavioral;
```

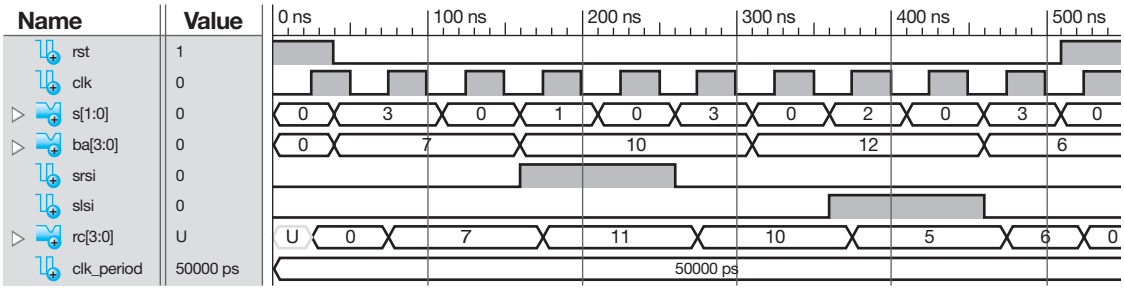
Things you should notice about the VHDL design in Listing 16.11:

- The signal *RST* is synchronous and must not be included in the sensitivity list.
- The reset signal *RST* is synchronous and therefore must be placed after rising_edge (*CLK*) in the process.
- The signals *BA* and *RC* are data type std_logic_vector with the range 3 **downto** 0 for a 4-bit bus.
- This is a sequential logic circuit because there is a clock input.

Waveform 16.5 shows a simulation with the correct functionality of design entity Shift_Register.

Things you should notice about the waveforms in Waveform 16.5:

- The shift register circuit is a completely synchronous circuit that only responds to the input signals at the tick of the clock, which is the rising edge of the clock signal *CLK* for this design.
- *RST* is a synchronous reset that can only occur after a clock tick when *RST* = 1 or high.



WAVEFORM 16.5 Simulation for the correct functionality of design entity Shift_Register

- When *RST* goes low, the select signal *S* determines the next operation performed by the shift register circuit after each clock tick, where $S = 0$ (hold), $S = 1$ (shift right), and $S = 2$ (shift left), and $S = 3$ (load).
- Observe that the each operation of the shift register circuit follows Table 16.6, which indicates the proper functionality of design entity Shift_Register.

PROBLEMS

Section 16.2 Utilization of the Arithmetic Logic Unit

16.1 What is an arithmetic logic unit (ALU)?

16.2 List all the logic functions (instructions) that are associated with the ALU for VBC1.

16.3 How are the different functions selected by the ALU for VBC1?

16.4 What OPCODE selects the LOADI instruction for the ALU for VBC1?

16.5 What OPCODE selects the ADDI instruction for the ALU for VBC1?

16.6 What OPCODE selects the ADD instruction for the ALU for VBC1?

16.7 What OPCODE selects the SR0 instruction for the ALU for VBC1?

Section 16.3 Designing the LOADI Instruction Part of the ALU

16.8 For Figure 16.2, write the VHDL code for the output signal assignment for the ALU in terms of the required input signal for the logic function LOADI.

Section 16.4 Designing the ADDI Instruction Part of the ALU

16.9 For Figure 16.2, write the VHDL code for the output signal assignment for the ALU in terms of the required input signals for the arithmetic function ADDI.

Section 16.5 Designing the ADD Instruction Part of the ALU

16.10 For Figure 16.2, write the VHDL code for the output signal assignment for the ALU in terms of the required input signals for the arithmetic function ADD.

Section 16.6 Designing the SR0 Instruction Part of the ALU

16.11 For Figure 16.2, write the VHDL code for the output signal assignments for the ALU in terms of the required input signals for the logic function SR0. Also provide an alternate way using the concatenation operator.

16.12 Why is it important to include a default value before the case statement for the output of an ALU when writing the VHDL code with a process and a case statement?

Section 16.7 Designing an ALU for VBC1

16.13 Show a partial VHDL design for an ALU for VBC1 for the LOADI and ADDI instructions using a single process and a case statement.

16.14 Show a partial VHDL design for an ALU for VBC1 for the ADD and SR0 instructions using a single process and a case statement.

Section 16.8 Additional Circuit Designs with VHDL

16.15 Is the INC (increment) operation a bit-wise or word-wise operation? Explain your answer by showing examples for a single-bit position for the INC operation.

16.16 Is the XNOR operation a bit-wise or word-wise operation? Explain your answer by showing examples for a single-bit position for the XNOR operation.

16.17 Show the transfer function for a subtraction circuit with input buses *BA* and *BB* and output bus *BC*. Is the subtraction operation a bit-wise or a word-wise operation?

16.18 Show the transfer function for an ADD circuit and an AND circuit with input buses *BX* and *BY* and the output bus *BZ*. Which circuit is a bit-wise operation and which circuit is a word-wise operation?

16.19 What is the purpose of the ampersand symbol "&" in VHDL?

16.20 Show a partial VHDL design for a 4-bit version of the ALU with the function table in Table P16.20 called ALU_DF. Show only the architecture declaration, and use a dataflow design style with a conditional signal assignment.

TABLE P16.20

S(1:0)	Output: BC(4:0)	Comment
00	$BC \leftarrow BB + 1$	Arithmetic operation (INC), BC(4) is the carry-out bit
01	$BC \leftarrow BB - BA$	Arithmetic operation (SUB), BC(4) is the carry-out bit
10	$BC \leftarrow BA \vee BB$	Logic operation (OR), ignore carry-out bit BC(4)
11	$BC \leftarrow BA \overline{\oplus} BB$	Logic operation (XNOR), ignore carry-out bit BC(4)

16.21 Write the required library clause, use clauses, and entity declaration for ALU_DF in problem 16.20.

16.22 Show a partial VHDL design for an 8-bit version of the ALU with the function table in Table P16.22 called ALU_BEH. Show only the architecture declaration, and use a behavioral design style with a process and a case statement.

TABLE P16.22

S(1:0)	Output: BC(8:0)	Comment
00	$BC \leftarrow BA - BB$	Arithmetic operation (SUB), BC(8) is the carry-out bit
01	$BC \leftarrow BA + BB$	Arithmetic operation (ADD), BC(8) is the carry-out bit
10	$BC \leftarrow BA \wedge BB$	Logic operation (AND), ignore carry-out bit BC(8)
11	$BC \leftarrow BA \vee BB$	Logic operation (OR), ignore carry-out bit BC(8)

16.23 Write the required library clause, use clauses, and entity declaration for ALU_BEH in problem 16.22.

16.24 Is a shifter circuit a combinati0onal or sequential logic circuit? Provide a reason for your answer.

16.25 Explain how the srl operator works in the following statement BC <= BA **srl** 2, where BC represents the signal on bus C and BA represents the signal on bus A. Determine BC for BA = 10111.

16.26 Explain how the sll operator works in the following statement BC <= BA **sll** 3, where BC represents the signal on bus C and BA represents the signal on bus A. Determine BC for BA = 10111.

16.27 What data type must be used for signals that use the srl and sll operators? Show the data type required for the statement BC <= BA **srl** 1 if BA is 10111110.

16.28 Show complete VHDL code for a simple 8-bit shifter circuit called SR_w1f_2b for the transfer function BC ← shift right BA *w*1 *fill* that shifts two bit positions. Use a dataflow design style with simple signal assignment statements.

16.29 How does a barrel shifter circuit differ from a shifter circuit?

16.30 Is a barrel shifter circuit a combinational or sequential logic circuit? Provide a reason for your answer.

16.31 Explain how the ror operator works in the following statement BC <= BA **ror** 2, where BC represents the signal on bus C and BA represents the signal on bus A. Determine BC for BA = 10111.

16.32 Explain how the rol operator works in the following statement BC <= BA **rol** 3, where BC represents the signal on bus C and BA represents the signal on bus A. Determine BC for BA = 10111.

16.33 What data type must be used for signals that use the ror and rol operators? Show the data type required for the statement BC <= BA **rol** 1 if BA is 101111.

16.34 Show complete VHDL code for a simple 8-bit barrel shifter circuit called RR_4b for the transfer function BC ← rotate right BA for 4 bits. Use a dataflow design style with simple signal assignment statements.

16.35 What is the function of the input MODE for a shifter circuit or a barrel shifter circuit?

16.36 Is a shift register circuit a combinational or sequential logic circuit? Provide a reason for your answer.

16.37 Does the shift register circuit discussed in Section 16.8.4, Figure 16.7, have a synchronous or an asynchronous reset? Where must *RST* appear in the VHDL code to handle this reset—before rising_edge (clk) or after rising_edge (clk)?

16.38 Discuss what the transfer function $RC \leftarrow BA$ does in a shift register circuit.

16.39 Discuss what the transfer function $RC \leftarrow RC$ does in a shift register circuit.

16.40 Discuss what the transfer function RC ← shift right RC *wSRSI fill* does in a shift register circuit.

16.41 Discuss what the transfer function RC ← shift left RC *wSLSI fill* does in a shift register circuit.

16.42 Show complete VHDL code for a simple shift register circuit called SRC_SR for the transfer function RC ← shift right RC *wSRSI fill* with 8 bits. Provide a synchronous reset signal *RST* for the design. Use a behavioral design style with a process, an if statement, and simple signal assignment statements.

16.43 Show complete VHDL code for a simple shift register circuit called SRC_SL for the transfer function RC ← shift left RC *wSLSI fill* with 6 bits. Provide an asynchronous reset signal *RST* for the design. Use a behavioral design style with a process, an if statement, and simple signal assignment statements.

16.44 Show complete VHDL code for a simple shift register circuit called SRC_SRL with the transfer functions RC ← shift right RC *wSRSI fill* and $RC \leftarrow BA$ with 8 bits. Provide a synchronous reset signal *RST* for the design. Use a behavioral design style with a process, an if statement, and a case statement.

17 Chapter

Completing the Design for VBC1

Chapter Outline

17.1 INTRODUCTION

In this chapter, we present circuits for a final hardware design of VBC1. A Running program counter is added to allow loading an address for the JNZ instruction. Circuits are added to combine the loading program counter and the running program counter. A run frequency circuit is added to slow down the frequency for VBC1 to allow the output to be observed via four LEDs and also to allow single stepping through a program.

17.2 DESIGNING A RUNNING PROGRAM COUNTER

Figure 17.1 shows the circuit with the necessary inputs and output for a running program counter (RPC). The purpose of the RPC is to manually single step programs and also to run programs for VBC1 at a specified frequency.

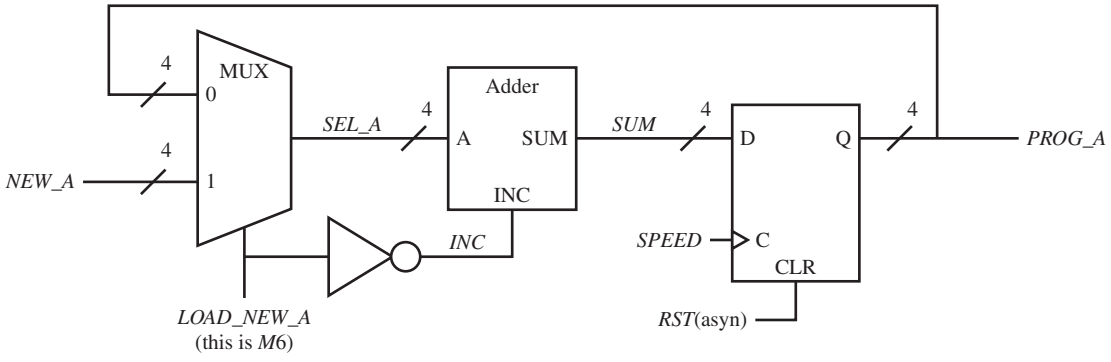


FIGURE 17.1 Circuit for an RPC for VBC1

Things you should notice about the Running program counter in Figure 17.1:

- An asynchronous reset signal *RST* (asyn) is used to reset the RPC.
- The loadable D flip-flops store the value of the signal *SUM*, which is the next address of the program counter via the output signal *PROG_A* at the next rising edge of the clock signal *SPEED* when *RST* is 0.
- A signal called *SPEED* is supplied to the control (or clock) input of the running program counter. The signal *SPEED* is provided by a push-button switch or by a frequency divider circuit. The push-button switch allows programs to be single stepped, while the frequency divider circuit allows programs to be run at a fixed frequency.
- The signal *PROG_A* (program address) is a 4-bit output that supplies an address to the instruction memory for single stepping or running programs.
- The running program counter has 16 states, which provide the addresses 0000 through 1111 for VBC1. It is a complex state machine because its state sequence can be changed by the external inputs *NEW_A* (new address) and *LOAD_NEW_A*. When a new address is loaded, this changes the counting sequence of the RPC.
- When the input signal *LOAD_NEW_A* (this is *M*6) is set to 0, the RPC is incremented for each clock tick at the control input C. Incrementing the RPC is required to execute all the instructions for VBC1. The JNZ instruction is a special instruction that also requires the RPC to load a new address for the condition $R0 \neq 0$ or $R1 \neq 0$. When the input signal *LOAD_NEW_A* is set to 1, the RPC is loaded with the signal *NEW_A*. Loading the signal *NEW_A* into the RPC will cause VBC1 to jump to that address at the next clock tick at the control input C. These are the two important operations that the RPC performs.
- The Adder may be thought of as an array of half adders with inputs A and B connected as a 4-bit ripple-carry adder labeled as input A and INC (increment or carry-in). The signal *SEL_A* is connected to input A, and signal *INC* is connected to input B(0). The rest of the B inputs are connected to the carry-outs from the adjacent half adders. *SUM*(3:0) = *A*(3:0) + *INC* = *SEL_A*(3:0) + *INC*.

Listing 17.1 shows a complete VHDL design for the running program counter in Figure 17.1.

LISTING 17.1
Complete VHDL design for the RPC for VBC1 (project: RPC)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RPC is port (
     rst, speed, load_new_a : in STD_LOGIC;
     new_a : in STD_LOGIC_VECTOR (3 downto 0);
     prog_a : inout STD_LOGIC_VECTOR (3 downto 0)
     );
end RPC;

architecture Mixed of RPC is
     signal sel_a : std_logic_vector (3 downto 0);
     signal inc : std_logic;
     signal sum : std_logic_vector (3 downto 0);
begin
     sel_a <= prog_a when load_new_a = '0' else
              new_a;
     inc <= not load_new_a;
     sum <= sel_a + inc when inc = '1' else
            sel_a;

process (rst, speed)
```

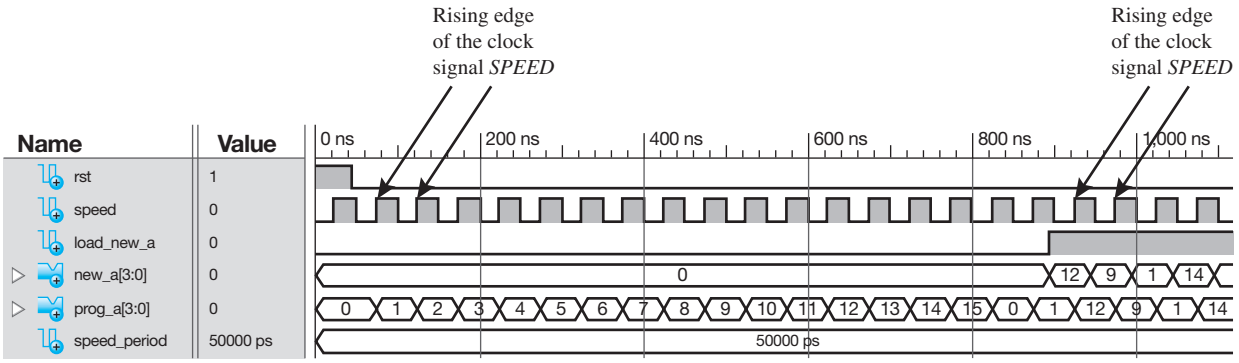
(Continued)

```
begin
     if rst = '1' then prog_a <= "0000";
     elsif rising_edge (speed) then prog_a <= sum;
     end if;
end process;
end Mixed;
```

Things you should notice about the VHDL design in Listing 17.1:

- The use clause **use** IEEE.STD_LOGIC_UNSIGNED.**ALL** is required because it contains the definition for "+" operator, which is used in the design.
- Signal mode inout is required for signal *PROG_A* in the entity. This is because the signal *PROG_A* is fed back to the MUX and also because the conditional signal assignment *PROG_A* <= *PROG_A* is inferred in the process.
- The internal signals *SEL_A*, *INC*, and *SUM* are declared between **architecture** and the first **begin**.
- The code for each element in the RPC—that is, the MUX, the inverter, the Adder, and loadable D flip-flop—is declared in the **architecture** after the first **begin**. The design style for the architecture is declared as mixed, because the elements MUX, inverter, and Adder are declared using a dataflow design style and the element loadable D flip-flop is declared using a process.

Waveform 17.1 shows the correct functionality of design entity RPC.



WAVEFORM 17.1 Simulation for the correct functionality of design entity RPC

Things you should notice about the waveforms in Waveform 17.1:

- Notice that RPC is incremented when *RST* = 0 and *LOAD_NEW_A* = 0 at each rising edge of the clock signal *SPEED*, as expected.
- When *RST* = 0 and *LOAD_NEW_A* = 1, then *NEW_A* is transferred to *PROG_A* at each rising edge of the clock signal *SPEED*, as expected.
- A **machine cycle** or **instruction cycle** is the period of time it takes to fetch and execute each assembly language instruction. **Fetch** means to get the instruction from instruction memory while **execute** means to decode and execute the instruction. Each assembly language instruction for VBC1 is fetched and executed at the frequency *SPEED*, so the period of time for a machine cycle or instruction cycle is $T_{mc} = T_{SPEED} = 1/f_{SPEED}$.
- The simulation was run at the frequency 20 MHz ($f_{SPEED} = 1/T_{\mathrm{SPEED}} = 1/50000ps = 20$ MHz). In practice, we will run VBC1 at the frequency of approximately 12 Hz, so we can observer the LED outputs.

17.3 COMBINING A LOADING AND A RUNNING PROGRAM COUNTER

To load an instruction into the instruction memory, a loading program counter (LPC) must supply an address to the instruction memory. The LPC must increment the address to allow the machine code for each instruction to be entered into consecutive locations in instruction memory. To execute an instruction that is already loaded into instruction memory, a running program counter (RPC) must supply an address to the instruction memory. The RPC must increment the address or load a new address based on the instruction that is being executed.

Figure 17.2 shows the combined circuits for a loading program counter, a running program counter, and a steering circuit that steers (or routes) the output of each program counter to the instruction memory of VBC1.

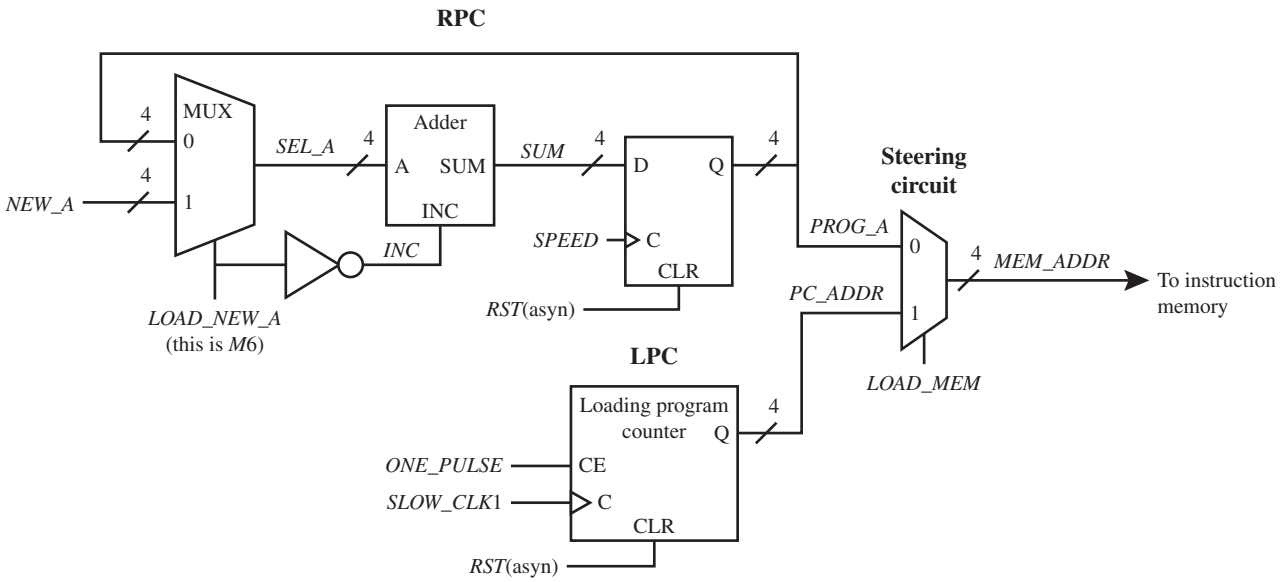


FIGURE 17.2 Combined Circuits for an LPC, an RPC, and a steering circuit that steers (or routes) the output of each program counter to the instruction memory

Things you should notice about the combined circuits in Figure 17.2:

- The loading program counter can be reset so that it starts loading machine code at the first address in instruction memory, which is 0. When *LOAD_MEM* is asserted or pulled to a 1, the output of the LPC supplies its output *PC_ADDR* to *MEM_ADDR*, which is the signal for the address of the instruction memory. The signal *SLOW_CLK*1 increments the address of the LPC each time a single pulse (*ONE_PULSE*) is supplied at the CE input.
- The running program counter can be reset so that it starts executing machine code at the beginning of the first address, which is 0 in instruction memory. When *LOAD_MEM* is not asserted or pulled to a 0, the output of the RPC supplies its output *PROG_A* to *MEM_ADDR*, which is the signal for the address of the instruction memory.
- The signal *SPEED* controls the RPC. All the instructions for VBC1 (IN, OUT, MOV, LOADI, ADDI, ADD, SR0, and JNZ) only require one clock cycle of the signal *SPEED* to be fetched from instruction memory and executed.

Listing 17.2 shows a complete VHDL design for the combined circuits in Figure 17.2.

LISTING 17.2
Complete VHDL design for the combined circuits for VBC1 (project: Com_Ckts)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Com_Ckts is port (
     rst, speed : in std_logic;
     load_mem : in std_logic;
     new_a : in std_logic_vector (3 downto 0);
     load_new_a : in std_logic;
     slow_clk1, one_pulse : in std_logic;
     mem_addr : out std_logic_vector (3 downto 0)
     );
end Com_Ckts;

architecture Mixed of Com_Ckts is
     signal q : std_logic_vector (3 downto 0);
     signal pc_addr : std_logic_vector (3 downto 0);
     signal sel_a : std_logic_vector (3 downto 0);
     signal inc : std_logic;
     signal sum : std_logic_vector (3 downto 0);
     signal prog_a : std_logic_vector (3 downto 0);
begin

Steering_Circuit:
     mem_addr <= pc_addr when load_mem = '1' else
                 prog_a;

LPC:
process (rst, slow_clk1)
begin
     if rst = '1' then q <= "0000";
     elsif (rising_edge (slow_clk1) and one_pulse = '1') then q <=
                                                          q + 1;
     end if;
end process LPC;
     pc_addr <= q;

RPC:
     sel_a <= prog_a when load_new_a = '0' else
              new_a;
     inc <= not load_new_a;
     sum <= sel_a + inc when inc = '1' else
            sel_a;
process (rst, speed)
begin
     if rst = '1' then prog_a <= "0000";
     elsif rising_edge (speed) then prog_a <= sum;
     end if;
end process;
end Mixed;
```

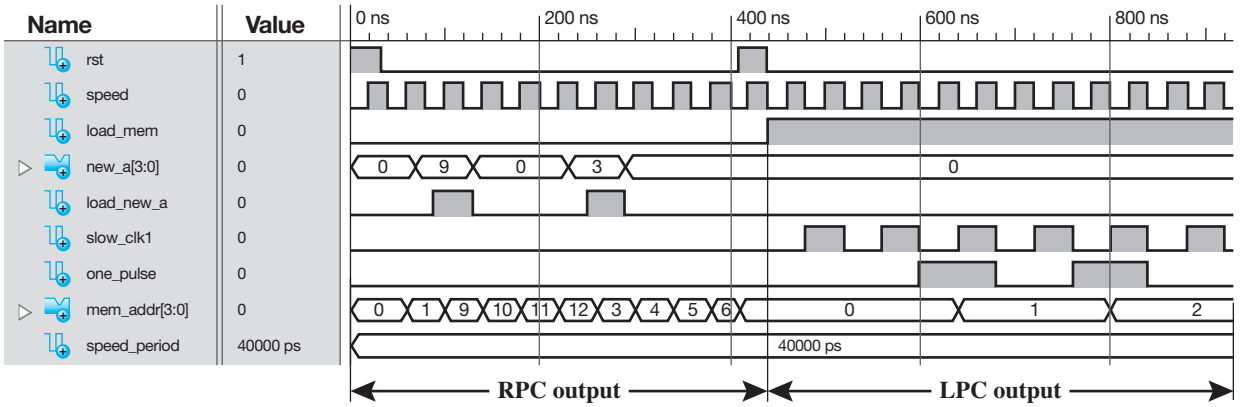
Things you should notice about the VHDL design in Listing 17.2:

- Each individual part of the VHDL code is marked by the name of the circuit followed by a colon (:)—that is, Steering_Circuit:, LPC:, and RPC:. This is a valid form of documentation

using VHDL. Notice that the LPC process is ended with **end process** LPC; but the RPC process is simply ended with **end process**;, which shows that either method of ending a process is valid.

- We used both conditional signal assignments and processes to provide a variety of VHDL coding styles.
- We used the arithmetic method in a process to keep the design of the LPC simple.

Waveform 17.2 shows the correct functionality of design entity Com_Ckts.



WAVEFORM 17.2 Simulation for the correct functionality of design entity Com_Ckts

Things you should notice about the waveforms in Waveform 17.2:

- *RST* is first set to 1 (asserted) and then to 0 (de-asserted) to clear the RPC and the LPC.
- *LOAD_MEM* is initially set to 0 so that *MEM_ADDR* follows the output of the RPC.
- When *NEW_A* = 9 and *LOAD_NEW_A* = 1, the new address is loaded into the RPC and the RPC counts up after the new address is loaded as expected—that is, from 9 to 10 to 11, and so on.
- When *NEW_A* = 3 and *LOAD_NEW_A* = 1, the new address is loaded into the RPC and the RPC counts up after the new address is loaded as expected—that is, from 3 to 4, to 5, and so on.
- *RST* is asserted then de-asserted, which clears RPC and LPC.
- *LOAD_MEM* is set to 1 so that *MEM_ADDR* follows the output of the LPC.
- Each time *ONE_PULSE* is asserted and the rising edge of *SLOW_CLK*1 occurs, the LPC counts up as expected—that is, from 0 to 1 and then from 1 to 2.

17.4 DESIGNING A RUN FREQUENCY CIRCUIT AND A SPEED CIRCUIT

Figure 17.3 shows the run frequency circuit and the speed circuit for VBC1.

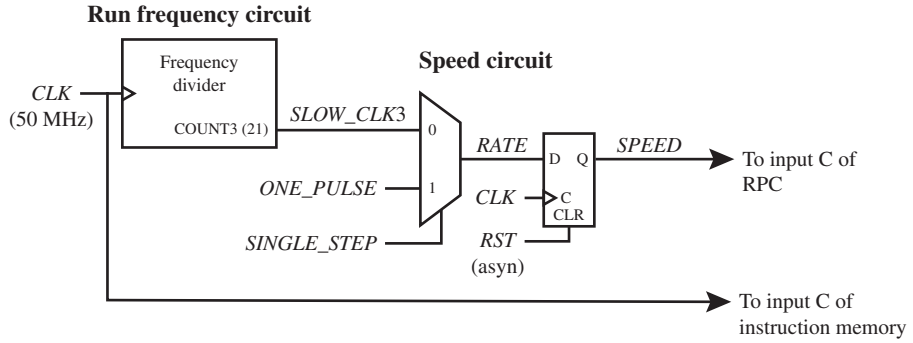


FIGURE 17.3 Run frequency circuit and speed circuit for VBC1

Things you should notice about the run frequency circuit and the speed circuit in Figure 17.3:

- Signal *CLK* supplies the frequency 50 MHz to the instruction memory.
- The frequency of *SLOW_CLK*3 is obtained from a frequency divider circuit.
- The output of the frequency divider is selected so that the output signals to the LEDs (the four output LEDs) can be visually observed when a program is running—that is, not blinking too fast for observation. For a clock frequency of 50 MHz, we chose a frequency of *SLOW_CLK*3 of 11.9209 Hz (50 M/2^{22} Hz). At this frequency, each instruction is fetched and executed at 11.9209 Hz.
- When the signal *SINGLE_STEP* is 0, the signal *SLOW_CLK*3 is steered to the output of the MUX. The flip-flop SPEED is used to prevent a gated clock from occurring in the circuit. Only one clock cycle of the frequency *SPEED* is required to fetch and execute each instruction in instruction memory. The RPC determines the address of the instruction that will be fetched and executed.
- When the signal *SINGLE_STEP* is 1, the signal *ONE_PULSE* is steered to the output of the MUX. Only one pulse is required for VBC1 to fetch and execute each instruction in Instruction Memory when single stepping.
- As a reminder, the LPC determines the address of the instruction that will be manually loaded into instruction memory, and the RPC determines the address of the instruction that will be fetched and executed from instruction memory.

Listing 17.3 shows a complete VHDL design for the run frequency circuit and the speed circuit in Figure 17.3 with the signal *COUNT*3(21) changed to *COUNT*3(2) to show "proof of concept" of the circuits. Remember, *SLOW_CLK*3 = *COUNT*3.

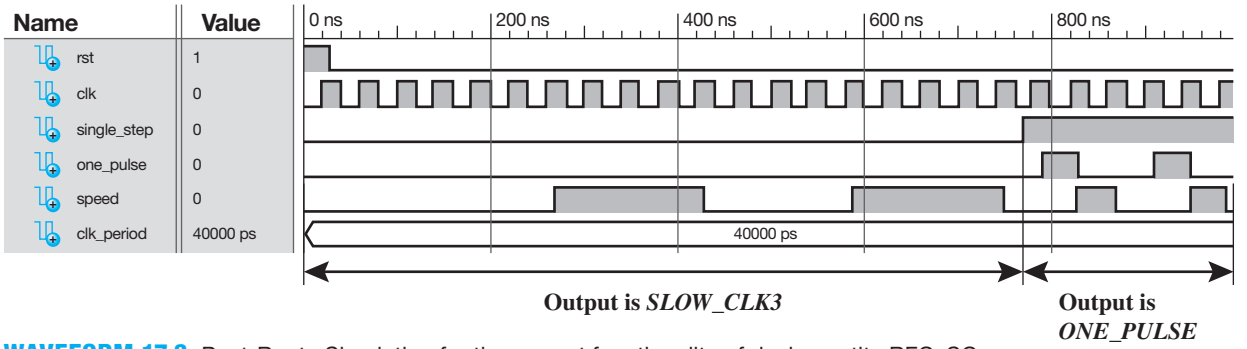
LISTING 17.3 Complete VHDL design for the run frequency circuit and speed circuit for VBC1 (project: RFC_SC)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity RFC_SC is port (
     rst, clk, one_pulse, single_step : in std_logic;
     speed : out std_logic
     );
end RFC_SC;

architecture dataflow of RFC_SC is
     signal count3: std_logic_vector (2 downto 0);
     signal slow_clk3, rate: std_logic;
begin

Run_Frequency_Circuit:
     count3 <= count3 + 1 when rising_edge(clk) else
               count3;
     slow_clk3 <= count3(2);

Speed_Circuit:
     rate <= one_pulse when single_step = '1' else
             slow_clk3;
     speed <= '0'  when rst = '1' else
              rate when rising_edge (clk);
end dataflow;
```

Things you should notice about the complete VHDL design in Listing 17.3:

- Each individual part of the VHDL code is marked by the name of the circuit followed by a colon (:)—that is, Run Frequency Circuit: and Speed Circuit:.
- We used a dataflow design style for each circuit in the design—that is, only conditional signal assignments and Boolean equations are used.
- The frequency of the signal *COUNT*3(21) is too slow compared to the frequency of the signal *CLK* to provide an observable simulation output for *SPEED*. To show a simulation output that provides a "proof of concept" of the circuits in Figure 17.3, we changed the signal *COUNT*3(21) to *COUNT*3(2). *COUNT*3(2) divides the frequency *CLK* by 2^3(or 8), while *COUNT*3(21) divides the frequency of *CLK* by 2^{22} (or 4,194,304), which would be stretched out too far to observe via a simulation waveform diagram.

Waveform 17.3 shows the correct functionality of design entity RFC_SC.



WAVEFORM 17.3 Post-Route Simulation for the correct functionality of design entity RFC_SC

Things you should notice about the waveforms in Waveform 17.3:

- *SINGLE_STEP* is first set to 0 to allow the signal *SLOW_CLK*3 to be steered to the output of the speed circuit. The frequency of *SLOW_CLK*3 is the frequency of *CLK* divided by 8 as observed in the waveform for the signal *SPEED*.
- *SINGLE_STEP* is then set to 1 to allow the signal *ONE_PULSE* to be steered to the output of the speed circuit. The signal *SPEED* is the same as the signal *ONE_PULSE* as observed in the waveform for *SPEED*.
- Waveform 17.3 was obtained using a post-route simulation. A behavioral simulation failed to provide an output for the signal *SPEED*, so a post-route simulation was used.

17.5 DESIGNING CIRCUITS TO PROVIDE A LOADER FOR INSTRUCTION MEMORY FOR VBC1

Experiment 17L is an experiment designed to provide the capability to load a program into the instruction memory of VBC1 via a file created by the Save button in EASY1. Additional circuitry must be added to VBC1 to create VBC1-L. After VBC1-L is downloaded into the FPGA on a BASYS 2 board or on a NEXYS 2 board, the machine code for a program can be loaded into the instruction memory via the USB connector on the board. Experiment 17L is self-contained and shows the additional circuitry necessary to design VBC1-L.

In order to load instruction memory contents via the USB connector, a computer software program is required. From this textbook's website, download the installer for the VBC1-L (VBC1-EL) Memory Loader to your computer, and install the software.

The machine code file that is generated by EASY1 can be entered into the VBC1-L (VBC1-EL) Memory Loader. Then the machine code can be automatically loaded into the instruction memory of VBC1-L by clicking on the Load Memory button. For more information, see Appendix E, Section E.3, Loading Memory via the Memory Loader Program.

PROBLEMS

Section 17.2 Designing a Running Program Counter

17.1 What is the purpose of the running program counter circuit for VBC1 shown in Figure 17.1?

17.2 What are the two important operations that are performed by the RPC circuit in Figure 17.1?

17.3 What is the purpose of the Adder in the RPC circuit in Figure 17.1?

17.4 What is the purpose of the MUX in the RPC circuit in Figure 17.1?

17.5 What are the instructions that cause the signal *NEW_A* to be supplied to the signal *SEL_A* in the RPC circuit in Figure 17.1? Be sure to show complete instructions and provide the required register and the contents of the register.

17.6 What is the purpose of the loadable D flip-flops in the RPC circuit in Figure 17.1?

17.7 Write a conditional signal assignment for the loadable D flip-flops described by the process in Figure 17.1.

17.8 How can the RPC circuit in Figure 17.1 be used as a loading program counter?

Section 17.3 Combining a Loading and a Running Program Counter

17.9 What is the purpose of the loading program counter (LPC) for VBC1?

17.10 How many clock cycles of the signal *SPEED* are required to fetch and execute each instruction for VBC1?

17.11 For the combined circuits in Figure 17.2, what is the name of the logic device used for the steering circuit? What is the name of the input signal to the steering circuit that provides the steering?

17.12 In the steering circuit in Figure 17.2, what signal is steered to the signal *MEM_ADDR*—that is, the output of the steering circuit—when the signal *LOAD_MEM* is 1?

17.13 In the steering circuit in Figure 17.2, what signal is steered to signal *MEM_ADDR*—that is, the output of the steering circuit—when the signal *LOAD_MEM* is 0?

17.14 In Figure 17.2, why is there a reset signal required for the LPC?

17.15 In Figure 17.2, why is there a reset signal required for the RPC?

17.16 In Figure 17.2, what signals increment the LPC?

17.17 In Figure 17.2, what signal clocks the RPC?

17.18 In the design of the RPC for VBC1, how long does it take to fetch and execute each instruction in Waveform 17.2?

17.19 Show a partial VHDL design called LPC_PSNS for the LPC for VBC1 in Figure 17.2. Show only the architecture declaration, and use the PS/NS tabular method even though this is not the most compact method for this circuit design.

17.20 Write the required library clause, use clauses, and entity declaration for LPC_PSNS in problem 17.19. Combine the VHDL code in problems 17.19 and 17.20 and obtain a simulation for the design entity LPC_PSNS to show correct functionality of your VHDL design. To make the simulation realistic, only allow *ONE_PULSE* to exist for one clock cycle of *SLOW_CLK*1.

17.21 Show a partial VHDL design called RPC_MIXED for the RPC for VBC1 in Figure 17.2. Show only the architecture declaration, and use a process for the MUX, a process for the NOT gate, a process for the Adder, and a conditional signal assignment for the D flip-flops.

17.22 Write the required library clause, use clauses, and entity declaration for RPC_MIXED in problem 17.21. Combine the VHDL code in problems 17.21 and 17.22 and obtain a simulation for the Design Entity RPC_MIXED to show correct functionality of your VHDL design.

Section 17.4 Designing a Run Frequency Circuit and a Speed Circuit

17.23 Write a complete VHDL design for the run frequency circuit for VBC1 shown in Figure 17.3. In the architecture declaration, use the arithmetic method within a process with an if statement. Also use a Boolean equation.

17.24 Obtain a simulation for your design in problem 17.23 with the signal *COUNT*3(21) changed to *COUNT*3(1) to show "proof of concept." What is the frequency of the signal *COUNT*3(1) compared to the frequency of *CLK*?

17.25 Write a complete VHDL design just for the speed circuit in Figure 17.3. In the architecture declaration, use a process with a case statement for the MUX and an if statement for the D flip-flop.

17.26 Obtain a behavioral simulation and a post-route simulation for your design in problem 17.25. How is a post-route simulation different from a behavioral simulation? Hint: Refer to Chapter 5, Section 5.4.

17.27 Write a complete VHDL design for the run frequency circuit and the speed circuit in Figure 17.3. Use a dataflow design style for the complete design. Show a simulation for the design for proof of concept of the circuit with *COUNT*3(21) changed to *COUNT*3(0). What is the frequency of the signal *COUNT*3(0) compared to the frequency of *CLK*?

Chapter 18

Assembly Language Programming for VBC1-E

Chapter Outline

18.1 INTRODUCTION

In this chapter, you will learn each of the instructions for VBC1-E by reviewing each instruction in detail. VBC1-E is an enhanced version of VBC1 that has modified input and output instructions (IN and OUT). The enhanced version also has data memory. New instructions (STORE and FETCH) access data memory. The enhanced version has a new subtraction instruction (SUB) and several new logic instructions (NOT, AND, OR, and XNOR). Several new shift instructions (SR1, SL0, and SL1) have also been added with two new rotate instructions (RR and RL). The enhanced version has three new unconditional jump instructions (JMP, JMPR, and HALT). The enhanced version also has a software and hardware interrupt. New instructions (INT and IRET) provide software interrupt capability. In addition, the assembler for VBC1-E has two basic assembler directives to assist in programming VBC1-E. In the last section, we present the complete instruction set for VBC1-E.

18.2 INSTRUCTION SUMMARY

Like VBC1, VBC1-E is still a very basic small 4-bit digital computer. Table 18.1 shows a brief summary of instructions and OPCODE bit assignments for VBC1-E. In the instructions for VBC1, the three instructions LOADI, ADDI, and JNZ use all of the machine bits. These instructions remain the same for VBC1-E—that is, they are unchanged instructions. To enhance VBC1 to make VBC1-E for just 4 bits is a little tricky. The remaining five instructions for VBC1

each have unused machine code bits that were arbitrarily set to 0. Some of these unused machine code bits are now used for VBC1-E as OPCODE extension bits.

TABLE 18.1 Brief summary of instructions and OPCODE bit assignments for VBC1-E

Instructions	Unchanged, modified, or new instruction	OPCODE bit assignments 7:5 plus extension bits	
MOV	Modified	000 00	Extension bits 1:0
STORE	New	000 01	Extension bits 1:0
FETCH	New	000 10	Extension bits 1:0
LOADI	Unchanged	001	
ADD	Modified	010 000	Extension bits 2:0
SUB	New	010 001	Extension bits 2:0
NOT	New	010 100	Extension bits 2:0
AND	New	010 101	Extension bits 2:0
OR	New	010 110	Extension bits 2:0
XNOR	New	010 111	Extension bits 2:0
ADDI	Unchanged	011	
SR0	Modified	100 000	Extension bits 2:0
SR1	New	100 001	Extension bits 2:0
SL0	New	100 010	Extension bits 2:0
SL1	New	100 011	Extension bits 2:0
RR	New	100 100	Extension bits 2:0
RL	New	100 101	Extension bits 2:0
IN	Modified	101 10	Extension bits 3:2
OUT	Modified	101 00	Extension bits 3:2
INT	New	101 01	Extension bits 3:2
IRET	New	101 11	Extension bits 3:2
JMP	New	110 0	Extension bit 4
JMPR	New	110 1	Extension bit 4
HALT	New	110 10000	Extension bits 4:0
JNZ	Unchanged	111	

In Table 18.1, there are only 3 instructions for VBC1-E that are exactly the same as for VBC1. All the other instructions for VBC1-E have been modified, or they are new instructions. Observe that 5 of the VBC1 instructions are modified, and there are 17 new instructions. VBC1-E has a total of 25 instructions.

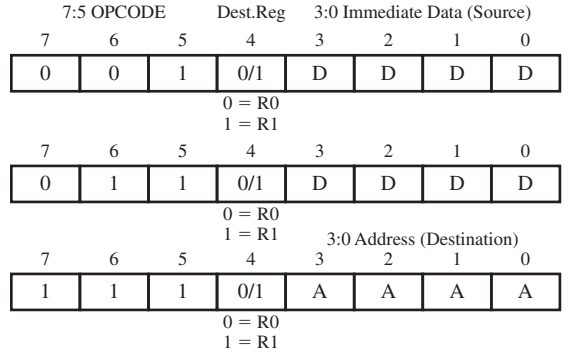
Observe in Table 18.1 that the OUT instruction no longer has the OPCODE 110 as it did for VBC1. The OUT instruction for VBC1-E has the OPCODE 101 00 with the extension bits 3:2. This change allows us to use the OPCODEs 110 0 and 110 1 to provide the new instructions JMP

(**unconditional jump**), JMPR (**unconditional jump relative**), and HALT (**unconditional jump to itself**).

Table 18.2 provides a summary of the three instructions that are unchanged for VBC1-E.

TABLE 18.2 Unchanged instructions for VBC1-E in ALF, TFF, and MCF

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
LOADI DR,Data	DR ← IR(3:0)	7:5 OPCODE; Dest.Reg (0 = R0, 1 = R1); 3:0 Immediate Data (Source)
ADDI DR,Data	DR ← DR + IR(3:0)	7:5 OPCODE; Dest.Reg (0 = R0, 1 = R1); 3:0 Immediate Data (Source)
JNZ DR,Address	PC(3:0) ← IR(3:0), if DR ≠ 0 else PC(3:0) ← PC(3:0) + 1	7:5 OPCODE; Dest.Reg (0 = R0, 1 = R1); 3:0 Address (Destination)



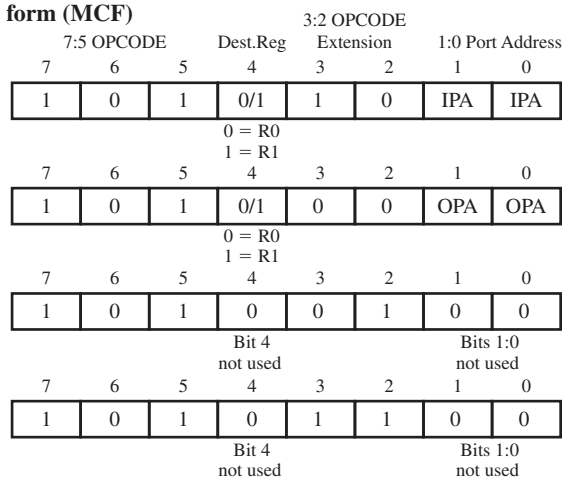
In Table 18.2, the abbreviations in the transfer function form column are as follows: DR is destination register, IR is instruction register, and PC is program counter.

18.3 INPUT, OUTPUT, AND INTERRUPT INSTRUCTIONS

Table 18.3 shows the modified input and output instructions and the new interrupt instructions for VBC1-E.

TABLE 18.3 Modified input and output instructions and the new interrupt instructions for VBC1-E in ALF, TFF, and MCF

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
IN DR,Port_A	DR ← IP[IR(1:0)]	7:5 OPCODE; Dest.Reg (0 = R0, 1 = R1); 3:2 OPCODE Extension; 1:0 Port Address
OUT DR,Port_A	OP[IR(1:0)] ← DR	7:5 OPCODE; Dest.Reg (0 = R0, 1 = R1); 3:2 OPCODE Extension; 1:0 Port Address
INT	RETA(3:0) ← PC(3:0) + 1 PC(3:0) ← 0000, IPROC ← 1	7:5 OPCODE; Bit 4 not used; 3:2 OPCODE Extension; Bits 1:0 not used
IRET	PC(3:0) ← RETA(3:0) IPROC ← 0	7:5 OPCODE; Bit 4 not used; 3:2 OPCODE Extension; Bits 1:0 not used



Hardware interrupt: Signal *TRIG_INT* causes RETA(3:0) ← PC(3:0), PC(3:0) ← 0000, IPROC ← 1

In Table 18.3, the abbreviations in the transfer function form column are as follows: DR is destination register, IP is input port, IR is instruction register, OP is output port, RETA is return address, PC is program counter, and IPROC is interrupt process.

The machine code bits 3:0 are not used for the IN instruction for VBC1. Taking advantage of this fact allows us to simply change the machine code such that bits 3:2 are converted to OPCODE extension bits. In addition, machine code bits 1:0 are converted to input port address (IPA) bits for the IN instruction. The assembly language form, transfer function form, and machine code form for the IN instruction are all different for VBC1-E compared to VBC1.

The new OPCODE for the modified IN instruction is 101 10. The first 3 bits for the new OPCODE are 101, which represent bits 7:5 (the OPCODE bits), and the last 2 bits are 10, which represent bits 3:2 (the OPCODE extension bits). Using OPCODE extension bits, we can now add three new instructions with the following new OPCODEs: 101 00, 101 01, and 101 11. We elected to add the modified instruction (OUT) and two new instructions (INT and IRET) as shown in Table 18.3.

In this section, we will cover both the modified IN and OUT instructions. We will present the new instructions INT (interrupt) and IRET (interrupt return) and also discuss the hardware interrupt later. INT represents a software interrupt instruction and IRET represents an interrupt return instruction.

Form 18.1 shows the assembly language form and the machine code form for the IN DR,1 instruction for VBC1-E.

FORM 18.1 The IN DR,1 instruction for VBC1-E

	OPCODE bits (7:5)			Dest.Reg	OPCODE Extension bits 3:2		Input Port Address bits 1:0	
	7	6	5	4	3	2	1	0
IN DR,1	1	0	1	0/1	1	0	0	1
				0 = R0 1 = R1				

The IN (input) instruction provides a means to input an external value into the computer. The external input for VBC1-E can be supplied from four different locations or input ports, each with a different port address. Each location provides just 4 bits such as the inputs from four slide switches. The input value is either placed in R0 or R1—that is, the destination register DR. The transfer function form for IN DR,Port_A is DR ← IP[IR(1:0)], where IP is the input port. For the instruction IN DR,1 shown in Form 18.1, the port address is 1. For the IN instruction, VBC1-E has four different input port addresses, 00, 01, 10, and 11, or 0 through 3 in decimal.

The actual assembly language for the IN instruction IN R1,1 is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 18.1.

Mnemonic field	**Operands field**
IN	R1,1

PROGRAM 18.1 Assembly language program for the IN R1,1 instruction for VBC1-E

Notice that two operands are required; the first operand specifies a destination register, and the second operand specifies the port address. When writing assembly language for VBC1-E, you must specify the port address for the IN instruction.

Assembly 18.1 shows all versions of the IN R0,Port_A instruction listed in assembly language and in machine code for VBC1-E.

Assembly language	**Machine code**
IN R0,0	1 0 1 0 1 0 0 0
IN R0,1	1 0 1 0 1 0 0 1
IN R0,2	1 0 1 0 1 0 1 0
IN R0,3	1 0 1 0 1 0 1 1

ASSEMBLY 18.1 All versions of the IN R0,Port_A instruction for VBC1-E

Assembly 18.2 shows all versions of the IN R1,Port_A instruction listed in assembly language and in machine code for VBC1-E.

Assembly language	Machine code
IN R1,0	1 0 1 1 1 0 0 0
IN R1,1	1 0 1 1 1 0 0 1
IN R1,2	1 0 1 1 1 0 1 0
IN R1,3	1 0 1 1 1 0 1 1

ASSEMBLY 18.2 All versions of the IN R1,Port_A instruction for VBC1-E

The machine code shows the bit patterns for the IN instructions as they must be placed in instruction memory to be executed.

In the **programmer's register model** for VBC1-E shown in Figure 18.1, the machine code for the instruction IN R1,1 is manually loaded in the instruction memory at address 0. All other instruction memory locations are cleared. The PC (program counter) is cleared to address 0, and the IR (instruction register) shows the machine code value at address 0 (10111001). Registers R0, R1, and all the outputs in the programmer's register model are cleared. The input at port address 1 (IP1) is set to the value of 0110 or 6, and all other inputs in the programmer's register model are set to 0. The programmer's register model in Figure 18.1 does not include the data memory, which will be added later.

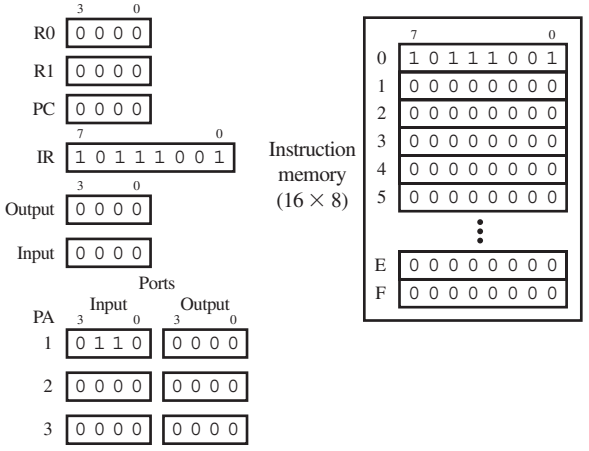


FIGURE 18.1 Result in the programmer's register model after manually loading the instruction IN R1,1 into VBC1-E instruction memory at address 0 with IP1 set to 6

After executing an instruction, the transfer function form indicates what happens to the various registers of VBC1-E. When the instruction IN R1,1 is executed, its transfer function form performs the operation R1 ← IP1, which places the value 0110(6) into R1.

So, in Figure 18.2, after executing the instruction IN R1,1, the value 0110(6) is transferred to register R1, as shown in the programmer's register model. The PC is now pointing to the address of the next instruction (0001), and the IR now contains the machine code for the next instruction (00000000).

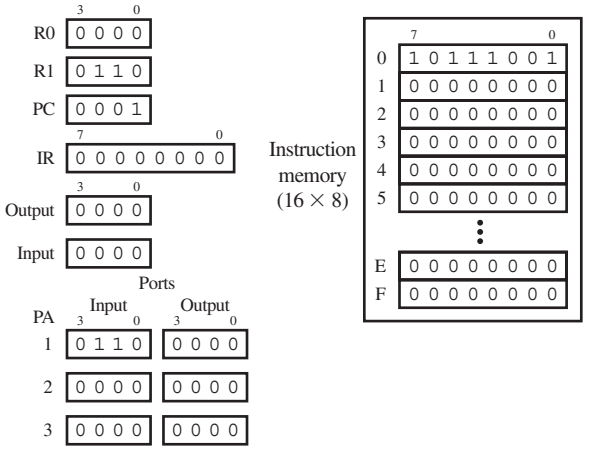


FIGURE 18.2 Result in the Programmer's Register Model after executing the instruction IN R1,1 for VBC1-E

Form 18.2 shows the assembly language form and the machine code form for the OUT DR,2 instruction for VBC1-E.

FORM 18.2 The OUT DR,2 instruction for VBC1-E

	OPCODE bits (7:5)			Dest.Reg	OPCODE Extension bits 3:2		Output Port Address bits 1:0	
	7	6	5	4	3	2	1	0
OUT DR,2	1	0	1	0/1	0	0	1	0
				0 = R0 1 = R1				

The OUT (output) instruction provides a means to output an internal value from the computer. The external output for VBC1-E can be supplied to four different locations or output ports, each with a different port address. Each location receives just 4 bits such as the outputs to four LEDs. The output value is either supplied by R0 or R1—that is, the destination register DR. The transfer function form for OUT DR,Port_A is OP[IR(1:0)] ← DR, where OP is the output port. For the OUT DR,2 instruction shown in Form 18.2, the port address is 2. For the OUT instruction, VBC1-E has four different output port addresses, 00, 01, 10, and 11, or 0 through 3 in decimal.

The actual assembly language for the OUT R0,2 instruction is shown in two fields: the first field is the mnemonic field, and the second field is the operands field as shown in Program 18.2.

Mnemonic field	**Operands field**
OUT	R0,2

PROGRAM 18.2 Assembly language program for the OUT R0,2 instruction for VBC1-E

Notice that two operands are required; the first operand specifies a destination register, and the second operand specifies the port address. When writing assembly language for VBC1-E, you must specify the port address for the OUT instruction.

Assembly 18.3 shows all versions of the OUT R0,Port_A instruction listed in assembly language and in machine code for VBC1-E.

Assembly language	**Machine code**
OUT R0,0	1 0 1 0 0 0 0 0
OUT R0,1	1 0 1 0 0 0 0 1
OUT R0,2	1 0 1 0 0 0 1 0
OUT R0,3	1 0 1 0 0 0 1 1

ASSEMBLY 18.3 All versions of the OUT R0,Port_A instruction for VBC1-E

Assembly 18.4 shows all versions of the OUT R1,Port_A instruction listed in assembly language and in machine code for VBC1-E.

Assembly language	**Machine code**
OUT R1,0	1 0 1 1 0 0 0 0
OUT R1,1	1 0 1 1 0 0 0 1
OUT R1,2	1 0 1 1 0 0 1 0
OUT R1,3	1 0 1 1 0 0 1 1

ASSEMBLY 18.4 All versions of the OUT R1,Port_A instruction for VBC1-E

The machine code shows the bit patterns for the OUT instructions as they must be placed in instruction memory to be executed.

Program 18.3 shows a simple assembly language program using the instructions IN, LOADI, and OUT.

```
IN R1,1
LOADI R0,9
OUT R0,2
```

PROGRAM 18.3 A simple assembly language program using the instructions IN, LOADI, and OUT for VBC1-E

In the programmer's register model for VBC1-E shown in Figure 18.3, the machine codes for the instructions in Program 18.3 are manually loaded in the instruction memory beginning at address 0. All other instruction memory locations are cleared. The PC (program counter) is cleared to address 0, and the IR (instruction register) shows the machine code value at address 0 (10111001). Registers R0 and R1 are cleared. The input at port address 1 (IP1) is set to the value of 1111 or 15, and all other inputs and outputs in the programmer's register model are set to 0.

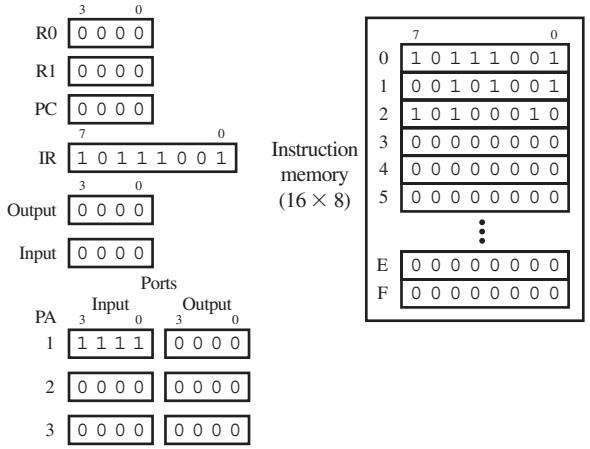


FIGURE 18.3 Result in the programmer's register model after manually loading the instructions in Program 18.3 into VBC1-E instruction memory beginning at address 0 with IP1 set to 15

After executing each instruction in Program 18.3, the transfer function form indicates what happens to the various registers of VBC1-E.

The following is a detailed description for the execution of each instruction in Program 18.3. When the instruction IN R1,1 is executed, its transfer function performs the operation R1 ← IP1, which places the value 1111(15) into R1. When the instruction LOADI R0,9 is executed, its transfer function performs the operation R0 ← 9, which places the value 9 into R0. When the instruction OUT R0,2 is executed, its transfer function performs the operation OP2 ← R0, which places the value 9 in output port 2.

So, in Figure 18.4, after executing the instruction IN R1,1, the value at input port 1, the value 1111(15) is transferred to R1. After executing the instruction LOADI R0,9, the value 9 is loaded into R0. After executing the instruction OUT R0,2, the value 1001(9) is transferred to output port 2, as shown in the programmer's register model. The PC is now pointing to the address of the next instruction to be executed which is 3, and the IR now contains the machine code for the next instruction at address 3, which is 00000000.

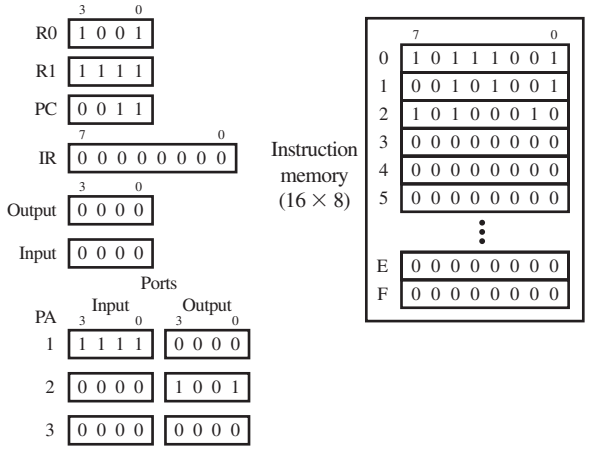


FIGURE 18.4 Result in the programmer's register model after executing the instructions in Program 18.3 for VBC1-E

18.4 DATA MEMORY INSTRUCTIONS

Table 18.4 shows the modified MOV instruction and the new STORE and FETCH instructions for VBC1-E.

TABLE 18.4 MOV, STORE, and FETCH instructions for VBC1-E in ALF, TFF, and MCF

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
MOV DR,SR	DR ← SR	7:0 OPCODE: 0 0 0 0 1; Dest.Reg: 0/1 (0 = R0, 1 = R1); Source Reg: 0/1 (0 = R0, 1 = R1); 1:0 OPCODE Extension: 0 0
STORE DR,Addr	DM[IR(3:2)] ← DR	7:0 OPCODE: 0 0 0 0 1; Dest.Reg: 0/1 (0 = R0, 1 = R1); A A: 3:2 Addr of Data Memory (DM); 0 1
FETCH DR, Addr	DR ← DM[IR(3:2)]	7:0 OPCODE: 0 0 0 0 1; Dest.Reg: 0/1 (0 = R0, 1 = R1); A A: 3:2 Addr of Data Memory (DM); 1 0

In Table 18.4, the abbreviations in the transfer function form column are as follows: DR is destination register, SR is source register, DM is data memory, and IR is instruction register.

The MOV instruction for VBC1 does not use the machine code bits 2:0. Taking advantage of this fact allows us to simply modify the machine code such that bits 1:0 are converted to OPCODE extension bits. The assembly language form and the transfer function form for the modified MOV instruction for VBC1-E are the same as the MOV instruction for VBC1, but the machine code form is different because of the OPCODE extension bits.

The new OPCODE for the modified MOV instruction is 000 00. The first 3 bits for the new OPCODE are 000, which represent bits 7:5 (the OPCODE bits), and the last 2 bits are 00, which represent bits 1:0 (the OPCODE extension bits). Using OPCODE extension bits, we can now add three new instructions with the following new OPCODEs: 000 01, 000 10, and 000 11. We elected to add two new instructions (STORE and FETCH) to access a data memory as shown in Table 18.4.

Observe in Table 18.4 that the address for the data memory in the machine code is contained in IR bits 3:2.

Notice in Table 18.4 that the OPCODE for the STORE instruction is 000 01 and the OPCODE for the FETCH instruction is 000 10. The OPCODEs for these instructions have the form (OPCODE bits) plus the (OPCODE extension bits).

Figure 18.5 shows the data memory map for VBC1-E. The STORE instruction is used to place data (4 bits) in one of four locations in the data memory, and the FETCH instruction is used to retrieve data (4 bits) from one of the four locations in data memory. Data memory provides a small amount of temporary memory for storing data values during the execution of a program.

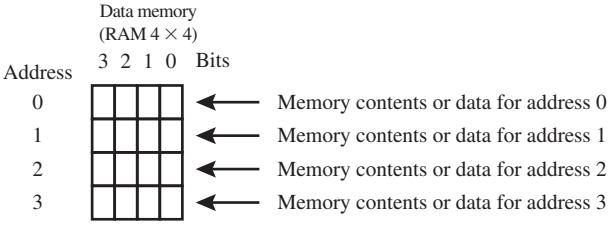


FIGURE 18.5 Data memory map for VBC1-E

Program 18.4 shows a simple assembly language program using the instructions IN, STORE, FETCH, and OUT.

```
IN R1,0
STORE R1,0
FETCH R0,0
OUT R0,1
```

PROGRAM 18.4 A simple assembly language program using the instructions IN, STORE, FETCH, and OUT for VBC1-E

In the programmer's register model for VBC1-E shown in Figure 18.6, the machine codes for the instructions in Program 18.4 are manually loaded into the instruction memory beginning at address 0. All other instruction memory locations are cleared. The PC (program counter) is cleared to address 0, and the IR (instruction register) shows the machine code value at address 0 (10111000). Registers R0, R1, and all the outputs in the programmer's register model are cleared. The input at port address 0 (IP0) is set to the value of 0110(6), and all other inputs in the programmer's register model are set to 0. This programmer's register model includes the data memory. All the data memory locations are cleared.

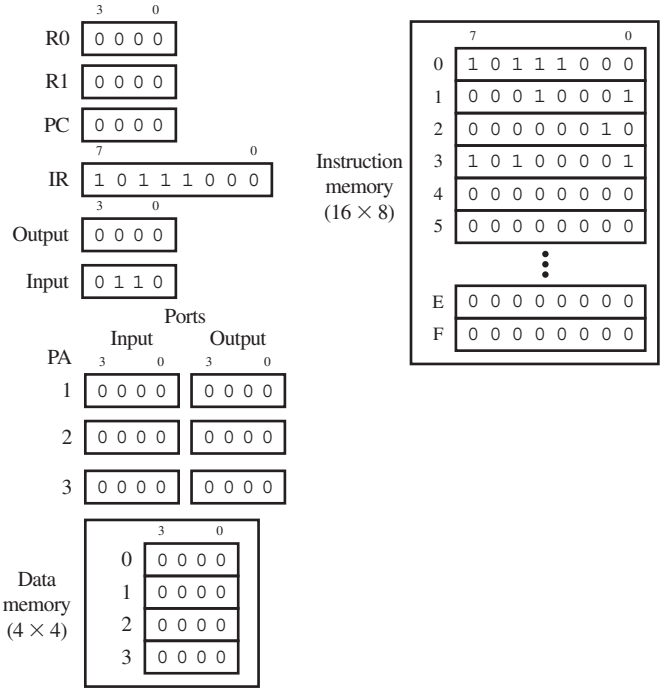


FIGURE 18.6 Result in the programmer's register model after manually loading the instructions in Program 18.4 into VBC1-E instruction memory at address 0 with IP0 set to 6

After executing each instruction in Program 18.4, the transfer function form indicates what happens to the various registers of VBC1-E.

The following is a detailed description for the execution of each instruction in Program 18.4. When the instruction IN R1,0 is executed, its transfer function performs the operation R1 ← IP0, which places the value 0110(6) into R1. When the instruction STORE R1,0 is executed, its transfer function performs the operation DM0 ← R1, which places the value of 0110(6) into data memory 0. When the instruction FETCH R0,0 is executed, its transfer function performs the operation R0 ← DM0, which places the value 0110(6) into R0. When the instruction OUT R0,1 is executed, its transfer function performs the operation OP1 ← R0, which places the value 0110(6) in output port 1.

So, in Figure 18.7, after executing the instruction IN R1,0, the value at input port 0 or 0110(6) is transferred to register R1. After executing the instruction STORE R1,0, the value 0110(6) is transferred to data memory 0. After executing the instruction FETCH R0,0, the value 0110(6) is transferred to R0. After executing the instruction OUT R0,1, then the value 0110(6) is transferred to output port 1, as shown in the programmer's register model. The PC is now pointing to the address of the next instruction to be executed which is 4, and the IR now contains the machine code for the next instruction at address 4, which is 00000000.

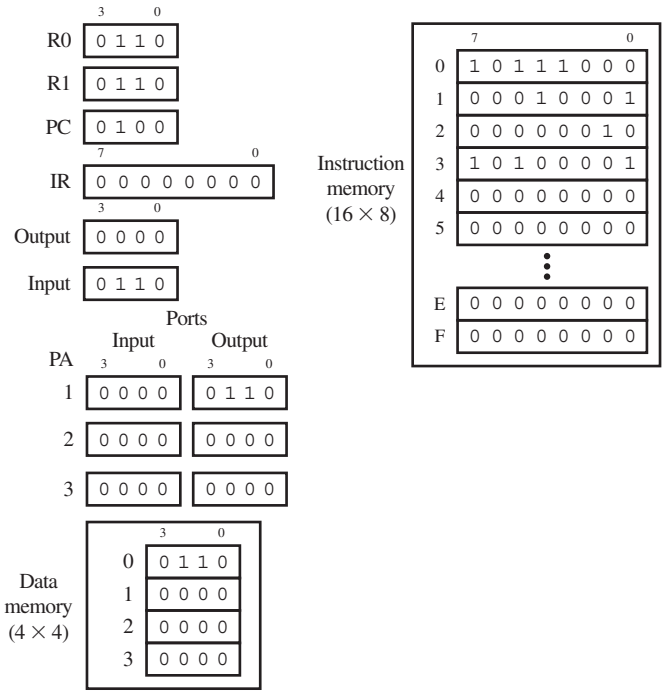


FIGURE 18.7 Result in the programmer's register model after executing the instructions in Program 18.4 for VBC1-E

18.5 ARITHMETIC AND LOGIC INSTRUCTIONS

Table 18.5 shows the modified ADD instruction and the new SUB, NOT, AND, OR, and XNOR instructions for VBC1-E.

In Table 18.5, the abbreviations in the transfer function form column are as follows: DR is destination register, and SR is source register.

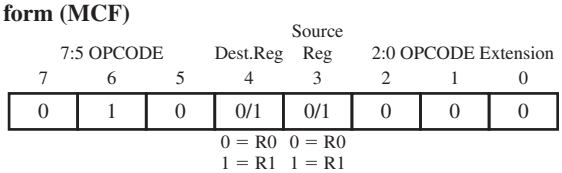
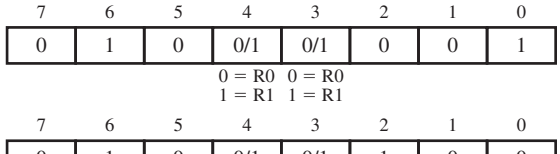
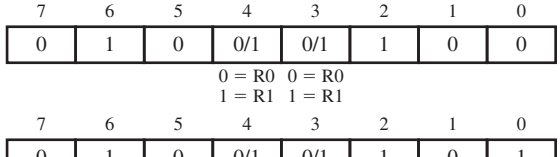
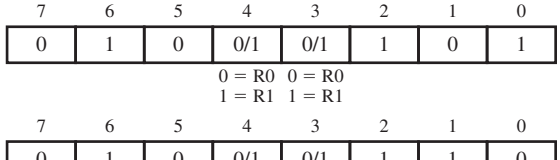
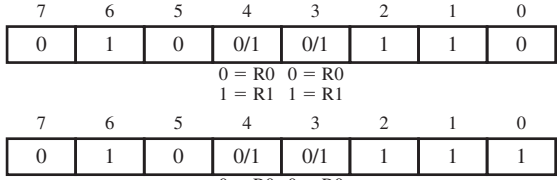
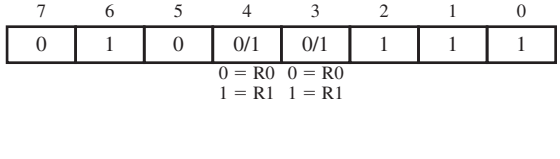
The ADD instruction for VBC1 does not use the machine code bits 2:0. Taking advantage of this fact allows us to simply change the machine code such that bits 2:0 are converted to OPCODE extension bits. The assembly language form and the transfer function form for the modified ADD instruction for VBC1-E are the same as the ADD instruction for VBC1, but the machine code form is different because of the OPCODE extension bits.

The new OPCODE for the modified ADD instruction is 010 000. The first 3 bits for the new OPCODE are 010, which represent bits 7:5 (the OPCODE bits), and the last 3 bits are 000, which represent bits 2:0 (the OPCODE extension bits). Using OPCODE extension bits, we can now add seven new instructions with the following new OPCODEs: 010 001, 010 010, 010 011, 010 100, 010 101, 010, 110, and 010 111. We elected to add five new instructions (SUB, NOT, AND, OR, and XNOR) as shown in Table 18.5.

Program 18.5 shows an assembly language program using the instructions IN, STORE, FETCH, NOT, OUT, and OR.

In the programmer's register model for VBC1-E shown in Figure 18.8, the machine codes for the instructions in Program 18.5 are manually loaded into the instruction memory beginning at address 0. All other instruction memory locations are cleared. The PC (program counter) is

TABLE 18.5 ADD, SUB, NOT, AND, OR, and XNOR instructions for VBC1-E in ALF, TFF, MCF

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
ADD DR,SR	DR ← DR + SR	
SUB DR,SR	DR ← DR − SR	
NOT DR,SR	DR ← !SR	
AND DR,SR	DR ← DR ∧ SR	
OR DR,SR	DR ← DR ∨ SR	
XNOR DR,SR	DR ← !(DR ⊕ SR)	


```
IN R0,0
STORE R0,0
FETCH R1,0
NOT R0,R0
OUT R0,0
OR R1,R0
OUT R1,1
```

PROGRAM 18.5 An assembly language program using the instructions IN, STORE, FETCH, NOT, OUT, and OR for VBC1-E

cleared to address 0 and the IR (instruction register) shows the machine code value at address 0 (10101000). Registers R0, R1, and all the outputs in the programmer's register model are cleared. The input at port address 0 (IP0) is set to the value of 0101 or 5, and all other inputs in the programmer's register model are set to 0. All the data memory locations are cleared.

After executing each instruction in Program 18.3, the transfer function form indicates what happens to the various registers of VBC1-E.

The following is a detailed description for the execution of each instruction in Program 18.5. When the instruction IN R0,0 is executed, its transfer function performs the operation R0 ← IP0, which places the value 0101(5) into R0. When the instruction STORE R0,0 is executed, its transfer function performs the operation DM0 ← R0, which places the value 0101(5) into data memory 0. When the instruction FETCH R1,0 is executed, its transfer function performs the operation R1 ← DM0, which places the value 0101(5) in R1. When the instruction NOT R0,R0 is executed, its transfer function performs the operation R0 ← !R0, which places the value 1010(10) in R0. When the instruction OUT R0,0 is executed, its transfer function performs

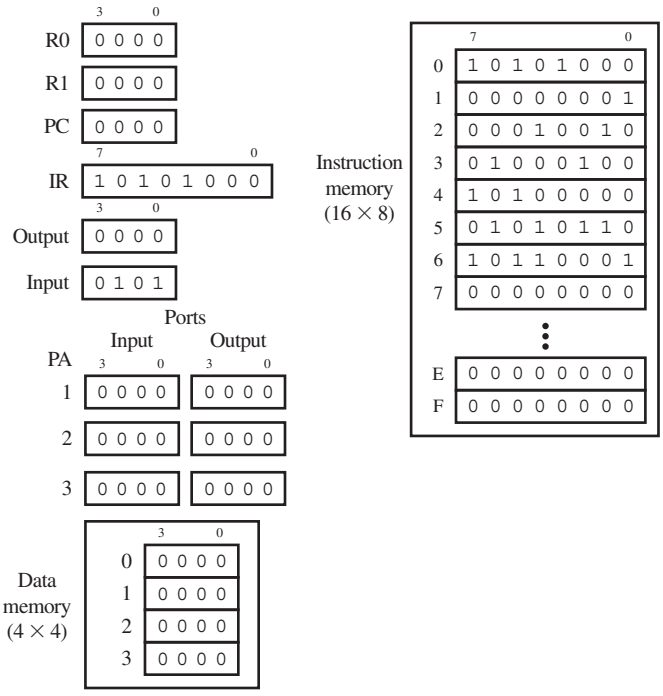


FIGURE 18.8 Result in the programmer's register model after manually loading the instructions in Program 18.5 into VBC1-E instruction memory at address 0 with IP0 set to 5

the operation OUT0 ← R0, which places the value 1010(10) in output port 0. When the instruction OR R1,R0 is executed, its transfer function performs the operation R1 ← R1 ∨ R0, which places the value 1111(15) into R1. When the instruction OUT R1,1 is executed, its transfer function performs the operation OP1 ← R1, which places the value 1111(15) in output port 1.

So, in Figure 18.9, after executing the instruction IN R0,0, the value at input port 0 or 0101(5) is transferred to register R0. After executing the instruction STORE R0,0, the value 0101(5) is transferred to data memory 0. After executing the instruction FETCH R1,0, the value

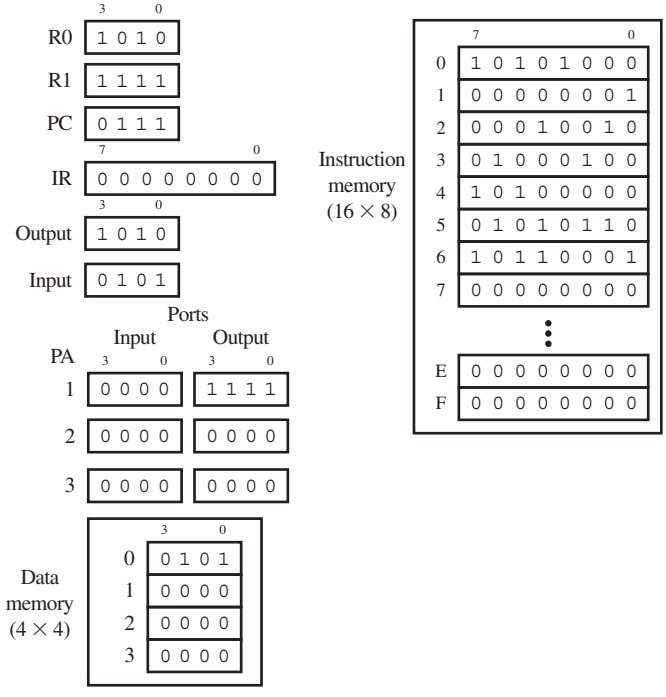


FIGURE 18.9 Result in the programmer's register model after executing the instructions in Program 18.5 for VBC1-E

0101(5) is transferred to R1. After executing the instruction NOT R0,R0, the value 1010(10) is transferred to R0. After executing the instruction OUT R0,0, the value 1010(10) is transferred to output port 0. After executing the instruction OR R1,R0, the value 1111(15) is transferred to R1. After executing the instruction OUT R1,1, the value 1111(15) is transferred to output port 1, as shown in the programmer's register model. The PC is now pointing to the address of the next instruction to be executed which is 7, and the IR now contains the machine code for the next instruction at address 7, which is 00000000.

18.6 SHIFT AND ROTATE INSTRUCTIONS

Table 18.6 shows the modified SR0 instruction and the new SR1, SL0, SL1, RR, and RL instructions for VBC1-E.

TABLE 18.6 SR0, SR1, SL0, SL1, RR, and RL instructions for VBC1-E in ALF, TFF, and MCF

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
SR0 DR,SR	DR ← 0 SR(3:1)	7:5 OPCODE = 100; Dest.Reg = 0/1; Source Reg = 0/1; 2:0 OPCODE Extension = 000
SR1 DR,SR	DR ← 1 SR(3:1)	7:5 OPCODE = 100; Dest.Reg = 0/1; Source Reg = 0/1; 2:0 OPCODE Extension = 001
SL0 DR,SR	DR ← SR(2:0) 0	7:5 OPCODE = 100; Dest.Reg = 0/1; Source Reg = 0/1; 2:0 OPCODE Extension = 010
SL1 DR,SR	DR ← SR(2:0) 1	7:5 OPCODE = 100; Dest.Reg = 0/1; Source Reg = 0/1; 2:0 OPCODE Extension = 011
RR DR,SR	DR ← SR(0) SR(3:1)	7:5 OPCODE = 100; Dest.Reg = 0/1; Source Reg = 0/1; 2:0 OPCODE Extension = 100
RL DR,SR	DR ← SR(2:0) SR(3)	7:5 OPCODE = 100; Dest.Reg = 0/1; Source Reg = 0/1; 2:0 OPCODE Extension = 101

0 = R0 0 = R0
1 = R1 1 = R1

In Table 18.6, the abbreviations in the transfer function form column are as follows: DR is destination register, and SR is source register.

The SR0 instruction for VBC1 does not use the machine code bits 2:0. Taking advantage of this fact allows us to simply change the machine code such that bits 2:0 are converted to OPCODE extension bits. The assembly language form and the transfer function form for the modified SR0 instruction for VBC1-E are the same as the SR0 instruction for VBC1, but the machine code form is different because of the OPCODE extension bits.

The new OPCODE for the modified SR0 instruction is 100 000. The first 3 bits for the new OPCODE are 100, which represent bits 7:5 (the OPCODE bits), and the last 3 bits are 000, which represent bits 2:0 (the OPCODE extension bits). Using the OPCODE extension bits, we can now

add seven new instructions with the following new OPCODEs: 100 001, 100 010, 100 011, 100 100, 100 101, 100 110, and 100 111. We elected to add five new instructions (SR1, SL0, SL1, RR, and RL) as shown in Table 18.6.

Program 18.6 shows an assembly language program using the instructions LOADI, STORE, SL1, OUT, RR, and FETCH.

```
LOADI R0, 0011b
LOADI R1, 1010b
STORE R0,0
SL1 R0,R0
OUT R0,0
RR R1,R1
OUT R1,1
FETCH R0,0
OUT R0,2
```

PROGRAM 18.6 An assembly language program using the instructions LOADI, STORE, SL1, OUT, RR, and FETCH for VBC1-E

Observe that numeric operands can be expressed in binary as 0011b or 11b, in decimal as 3d or 3, and also in hexadecimal as 3h.

In the programmer's register model for VBC1-E shown in Figure 18.10, the machine codes for the instructions in Program 18.6 are manually loaded into the instruction memory beginning at address 0. All other instruction memory locations are cleared. The PC (program counter) is cleared to address 0 and the IR (instruction register) shows the machine code value at address 0 (00100011). Registers R0, R1, and all the outputs in the programmer's register model are cleared. All the inputs in the programmer's register model are set to 0. All the data memory locations are cleared.

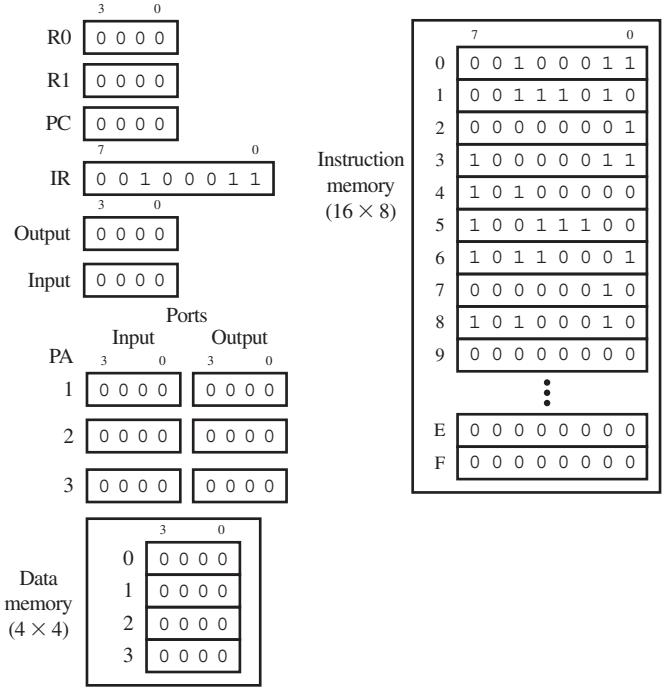


FIGURE 18.10 Result in the programmer's register model after manually loading the instructions in Program 18.6 into VBC1-E instruction memory at address 0 with all the inputs set to 0

After executing each instruction in Program 18.6, the transfer function form indicates what happens to the various registers of VBC1-E.

The following is a detailed description for the execution of each instruction in Program 18.6. When the instruction LOADI R0, 0011b is executed, its transfer function performs the operation

R0 ← 0011b, which places the value 0011(3) into R0. When the instruction LOADI R1, 1010b is executed, its transfer function performs the operation R1 ← 1010b, which places the value 1010(10) into R1. When the instruction STORE R0,0 is executed, its transfer function performs the operation DM0 ← R0, which places the value 0011(3) into data memory 0. When SL1 R0,R0 is executed, its transfer function performs the operation R0 ← R0(2:0) 1, which places 0111(7) into R0. When OUT R0,0 is executed, its transfer function performs the operation OP0 ← R0, which places the value 0111(7) in Output Port 0. When the instruction RR R1,R1 is executed, its transfer function performs the operation R1 ← R1(0) R1(3:1), which places the value 0101(5) into R1. When the instruction OUT R1,1 is executed, its transfer function performs the operation OP1 ← R1, which places the value 0101(5) into output port 1. When the instruction FETCH R0,0 is executed, its transfer function performs the operation R0 ← DM0, which transfers the value 0011(3) into R0, When the instruction OUT R0,2 is executed, its transfer function performs the operation OP2 ← R0, which places the value 0011(3) into output port 2.

So, in Figure 18.11, after the instruction LOADI R0,0011b is executed, the value 0011b is transferred to register R0. After the instruction LOADI R1,1010b is executed, the value 1010b is transferred to register R1. After the instruction STORE R0,0 is executed, the value 0011(3) is transferred to data memory 0. After the instruction SL1 R0,R0 is executed, the value 0111(7) is transferred to R0. After the instruction OUT R0,0 is executed, the value 0111(7) is transferred to output port 0. After instruction RR R1,R1 is executed, the value 0101(5) is transferred to R1. After the instruction OUT R1,1 is executed, the value 0101(5) is transferred to output port 1. After the instruction FETCH R0,0 is executed, the value 0011(3) is transferred to R0. After the instruction OUT R0,2 is executed, the value 0011(3) is transferred to output port 2, as shown in the programmer's register model. The PC is now pointing to the address of the next instruction to be executed which is 9, and the IR now contains the machine code for the next instruction at address 9, which is 00000000.

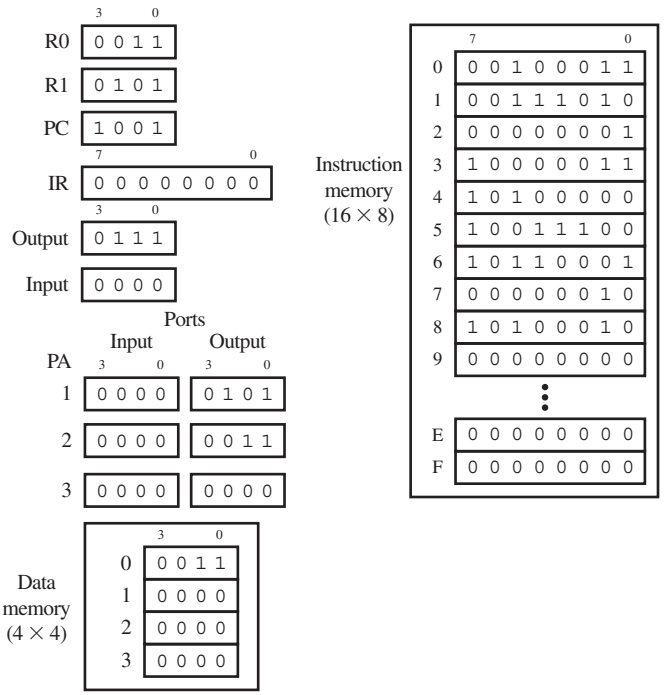


FIGURE 18.11 Result in the programmer's register model after executing the instructions in Program 18.6 for VBC1-E

18.7 JUMP, JUMP RELATIVE, AND HALT INSTRUCTIONS

The OUT instruction was modified to use OPCODE 101 with extension bits 3:2 in Section 18.3. This change allows us to use OPCODE 110 for new instructions.

Three new instructions JMP (**unconditional jump**), JMPR (**unconditional jump relative**) and HALT (**unconditional jump to itself**) for VBC1-E are shown in Table 18.7.

TABLE 18.7 Three new instructions JMP, JMPR, and HALT for VBC1-E

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
JMP Addr	PC(3:0) ← IR(3:0)	7:5 OPCODE; OPCODE Extension; 3:0 Address (Destination) — 7 6 5 4 3 2 1 0: 1 1 0 0 A A A A
JMPR OffSet	PC(3:0) ← PC(3:0) + IR(3:0)	3:0 Off Set to Destination — 7 6 5 4 3 2 1 0: 1 1 0 1 OS OS OS OS
HALT	PC(3:0) ← PC(3:0)	4:0 OPCODE Extension — 7 6 5 4 3 2 1 0: 1 1 0 1 0 0 0 0

In Table 18.7, the abbreviations in the transfer function form column are as follows: PC is program counter, and IR is instruction register.

The new instruction JMP (unconditional jump) provides an unconditional jump to the address specified in the address bits of the instruction. When the instruction JMP 0 is executed, this causes jump to address 0 in instruction memory. When the instruction JMP 6 is executed, this causes an unconditional jump to address 6 in instruction memory.

The new instruction JMPR (unconditional jump relative) provides an unconditional jump relative to the current address pointed to by the program counter (PC) to an address calculated by adding four offset bits to the current value of the program counter. As you will see in the next section, the actual program counter for VBC1-E will be increased from 4 bits to 5 bits to allow for additional program storage in extended instruction memory. The new address in the program counter for the instruction JMPR is determined as follows: New Address = Current address + Offset. When the actual program counter overflows to more than 4 bits, only the bottom 4 bits of the resulting addition are used for the destination address in the program counter. Jumping backward is accomplished by a forward jump. For example, JMPR 3 represents an unconditional jump forward of three instructions or a jump backward of 13 instructions from the current PC address in the instruction memory or extended instruction memory. Observe that 3 + 13 = 16, which is the maximum number of addresses in the instruction memory or extended instruction memory after dropping PC(4), the 5th bit.

Table 18.8 shows all the possibilities for the instruction JMPR *i*, where *i* ranges from 0 through 15 for a jump forward or a jump backwards from the current PC address.

When the value of the offset bits is 0, the instruction JMPR 0 is executed over and over. The new instruction HALT (unconditional jump to itself) provides the same operation as JMPR 0. When the instruction HALT is executed, the instruction HALT is executed over and over, which effectively halts the program.

Program 18.7 shows an assembly language program using the instructions LOADI, STORE, JMPR, SR0, OUT, JMP, FETCH, and HALT. Notice that this program is well written, because it does not run amuck or roam through memory.

In the programmer's register model for VBC1-E shown in Figure 18.12, the machine codes for the instructions in Program 18.7 are manually loaded into the instruction memory beginning at address 0. All other instruction memory locations are cleared. The PC (program counter) is cleared to address 0, and the IR (instruction register) shows the machine code value at address 0 (00100111). Registers R0, R1, and all the outputs in the programmer's register model are cleared.

TABLE 18.8 **All possibilities for the instruction JMPR *i*, where *i* ranges from 0 through 15 and the number of instructions the JMPR *i* will jump forward or jump backward from the current PC address**

Instruction	Jump forward from current PC address	Jump backward from current PC address
JMPR 0	0	16
JMPR 1	1	15
JMPR 2	2	14
JMPR 3	3	13
JMPR 4	4	12
JMPR 5	5	11
JMPR 6	6	10
JMPR 7	7	9
JMPR 8	8	8
JMPR 9	9	7
JMPR 10	10	6
JMPR 11	11	5
JMPR 12	12	4
JMPR 13	13	3
JMPR 14	14	2
JMPR 15	15	1

```
LOADI R0,7
STORE R0,0
JMPR 4
SR0 R0,R0
OUT R0,0
JMP 7
JMPR 13
FETCH R0,0
OUT R0,1
HALT
```

PROGRAM 18.7 An assembly language program using the instructions LOADI, STORE, JMPR, SR0, OUT, JMP, FETCH, and HALT for VBC1-E

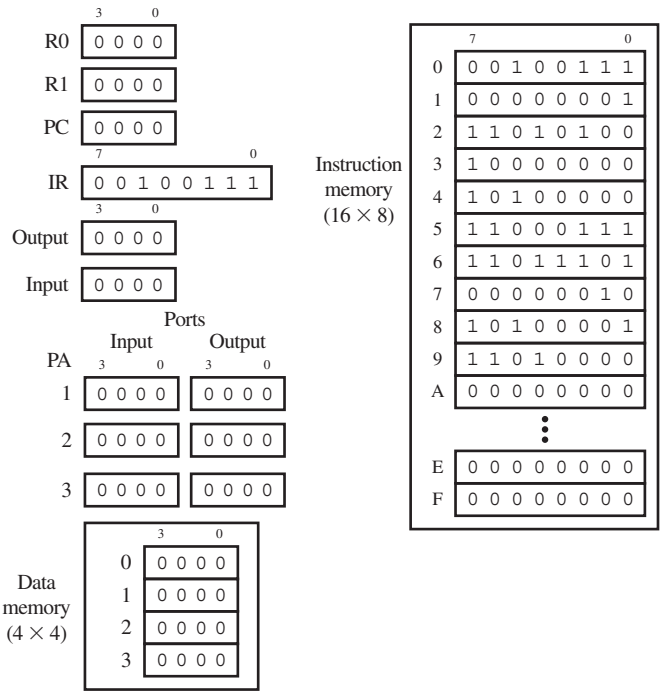


FIGURE 18.12 Result in the programmer's register model after manually loading the instructions in Program 18.7 into VBC1-E instruction memory at address 0 with all the inputs set to 0

All the inputs in the programmer's register model are set to 0. All the data memory locations are cleared.

After executing each instruction in Program 18.7, the transfer function form indicates what happens to the various registers of VBC1-E.

The following is a detailed description for the execution of each instruction in Program 18.7. When the instruction LOADI R0, 7 is executed, its transfer function performs the operation R0 ← 7, which places the value 0111(7) into R0. When the instruction STORE R0,0 is executed, its transfer function performs the operation DM0 ← R0, which places the value 0111(7) into data memory 0. When the instruction JMPR 4 is executed, its transfer function performs the operation PC ← PC + 4, which jumps forward four instructions to instruction JMPR 13. When the instruction JMPR 13 is executed, its transfer function performs the operation PC ← PC + 13, which jumps forward 13 instructions or backward 3 instructions to instruction SR0 R0,R0. When the instruction SR0 R0,R0 is executed, its transfer function performs the operation R0 ← 0 R0(2:0), which places 0011(3) into R0. When the instruction OUT R0,0 is executed, its transfer function performs the operation OP0 ← 0011(3), which places 0011(3) into output port 0. When instruction JMP 7 is executed, its transfer function performs the operation PC ← 7, which jumps to address 7 to instruction FETCH R0,0. When the instruction FETCH R0,0 is executed, its transfer function performs the operation R0 ← DM0, which places the value 0111(7) in R0. When OUT R0,1 is executed, its transfer function performs the operation OP1 ← R0, which places the value 0111(7) in output port 1. When the instruction HALT is executed, its transfer function performs the operation PC ← PC, which repeats the instruction HALT over and over.

So, in Figure 18.13, after the instruction LOADI R0,7 is executed, the value 0111(7) is transferred to register R0. After the instruction STORE R0,0 is executed, the value 0111(7) is transferred to data memory 0. After the instruction JMPR 4 is executed, the program counter is changed to jump forward four instructions to instruction JMPR 13. After the instruction JMPR 13 is executed, the program counter is changed to jump forward 13 instructions or back-

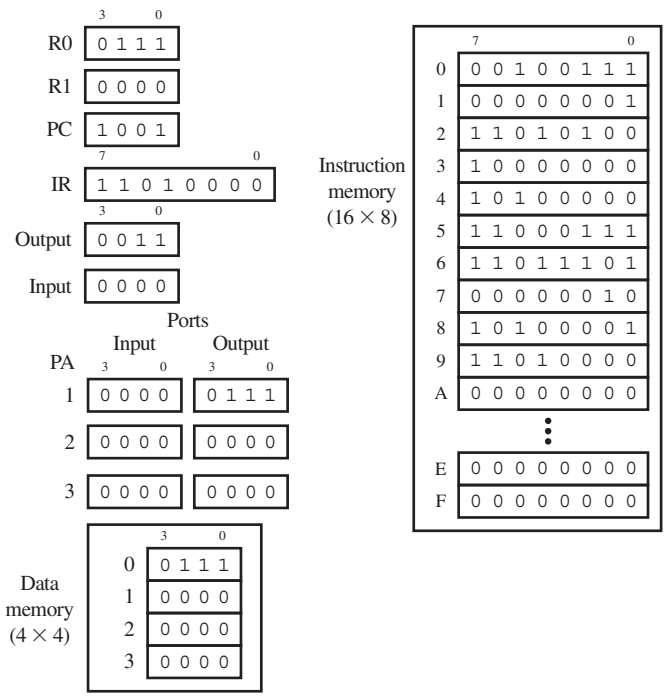


FIGURE 18.13 Result in the programmer's register model after executing the instructions in Program 18.7 for VBC1-E

ward 3 instructions to instruction SR0 R0,R0. After the instruction SR0 R0,R0 is executed, the value 0011(3) is loaded into R0. After executing the instruction OUT R0,0 is executed, the value 0011(3) is transferred to output port 0. After the instruction JMP 7 is executed, the program counter is changed to jump forward to address 7 to instruction FETCH R0,0. After the instruction FETCH R0,0 is executed, the value 0111(7) is transferred to R0. After the instruction OUT R0,1 is executed, the value 0111(7) is transferred to output port 1. After the instruction HALT is executed, the program counter is changed to jump backward to instruction HALT, as shown in the programmer's register model. The PC is now pointing to the address of the next instruction to be executed, which is 9, and the IR now contains the machine code for the next instruction at address 9, which is 11010000.

18.8 MORE ABOUT INTERRUPTS AND ASSEMBLER DIRECTIVES

The interrupt instructions are repeated in Table 18.9.

TABLE 18.9 Interrupt instructions

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
INT	RETA(3:0) ← PC(3:0) + 1 PC(3:0) ← 0000, IPROC ← 1	7:5 OPCODE; 3:2 OPCODE Extension 7 6 5 4 3 2 1 0 1 0 1 0 0 1 0 0 Bit 4 not used; Bits 1:0 not used
IRET	PC(3:0) ← RETA(3:0) IPROC ← 0	7 6 5 4 3 2 1 0 1 0 1 0 1 1 0 0 Bit 4 not used; Bits 1:0 not used

Hardware interrupt: Signal *TRIG_INT* causes RETA(3:0) ← PC(3:0), PC(3:0) ← 0000, IPROC ← 1

In Table 18.9, the abbreviations in the transfer function form column are as follows: RETA is return address, PC is program counter, and IPROC is interrupt process.

Instruction INT provides a means to execute instructions in an extended instruction memory, thus providing more programming capability for VBC1-E. When the instruction INT is executed, its transfer function performs the following three actions: (1) first, the value of the return address in instruction memory (the address of the instruction immediately following the instruction INT) is moved to the register named RETA (**return address**); (2) next, the PC (bits 3:0) is loaded with 0000 or 0 in decimal; and (3) then, the 5th bit in the PC named IPROC (**interrupt process**) or PC(4) is set to 1, or the actual program counter, PC(4:0), is set to 1 0000, which is the address for the first instruction in extended instruction memory.

In assembly language, an assembler directive named BIPROC (**begin interrupt process**) must be used to notify the assembler that all the subsequent instructions belong in the interrupt service routine. Assembly instructions occurring before the assembler directive BIPROC are placed in the first half of instruction memory (0 0000 through 0 1111). Assembly instructions occurring after the BIPROC directive are placed in the last half of instruction memory (1 0000 through 1 1111), which we call extended instruction memory.

The instruction INT should only be used in the instruction memory. The instruction IRET should only be used in the extended instruction memory, to allow the program to return to instruction memory.

When instruction IRET is executed, its transfer function must perform the following two actions: (1) place the return address in register RETA in the PC (bits 3:0) and (2) the 5th bit in the PC named IPROC is set to 0 as bits 3 through 0 in the PC are set to the return address.

Up to 16 instructions can be included in the interrupt routine. All instructions in VBC1-E are 8-bit instructions just as they were for VBC1. All the instructions for VBC1-E operate within the memory section in which they are placed—that is, at address 0 0000 through 0 1111 (section 0) for instruction memory or between address 1 0000 through 1 1111 (section 1) for extended instruction memory.

When a normal program is running, the 5th bit in the PC for VBC1-E, named IPROC, is 0 and the instructions are executed in instruction memory (section 0), but when the interrupt routine is running, IPROC is 1 and instructions are executed in extended instruction memory (section 1).

Figure 18.14 shows a sample layout for a program using the instructions INT, IRET, and the assembler command or directive BIPROC. Remember: The assembler command or directive BIPROC tells the assembler to place the instructions following the directive BIPROC in extended instruction memory.

FIGURE 18.14
Sample layout for a program using the instructions INT, IRET, and the assembler command or directive BIPROC

```
;Normal program in Instruction Memory (16x8)
;for these instructions IPROC is 0 which represents Section 0
Instruction at address 0
Instruction at address 1
Instruction at address 2
.
.
.
INT ;at address x, executes next instruction at 0000 in Extended
    ;Instruction Memory (Section 1)
Instruction at address x + 1; return address
Instruction at address x + 2
Instruction at address x + 3
.
.
.
Instruction at address F ;last address in Instruction Memory

;Program in Extended Instruction Memory (16x8)
;for these instructions IPROC is 1 which represents Section 1
BIPROC ;assembler command, interrupt service routine to follow
Instruction at address 0
Instruction at address 1
.
.
.
IRET ;at address y, returns program to Instruction Memory (Section 0)
Instruction at address y + 1
Instruction at address y + 2
.
.
.
Instruction at address F ;last address in Extended Instruction Memory
```

Observe in Figure 18.14 that the instruction IRET does not have to be the last instruction in the extended instruction memory in section 1, but it must be the last instruction executed in order to allow the program to return to instruction memory in section 0.

For VBC1-E, it is illegal to execute the instruction INT inside extended instruction memory—that is, after the assembler directive BIPROC. It is also illegal to use the assembler directive BIPROC more than once and to execute the instruction IRET outside extended instruction memory. The Assembler for VBC1-E, which we call EASY1-E, is designed to flag some of these basic errors.

An interrupt can also be caused by an external input interrupt signal. This type of interrupt is called a **hardware interrupt**. For VBC1-E, a hardware interrupt causes the normal program to jump to an interrupt service routine that is placed in the extended instruction memory. When the instruction IRET in the interrupt service routine is executed, the normal program begins executing again at the return address until another hardware interrupt occurs. The return address is loaded into register RETA. The return address is the current address of the program counter in instruction memory (section 0) when the external input interrupt signal *TRIG_INT* is applied.

A hardware interrupt monitors an external signal called *TRIG_INT* (trigger interrupt). When *TRIG_INT* is true (a single positive pulse), a hardware interrupt is initiated, which saves the current value of PC(3:0) in RETA(3:0) then loads PC(4:0) with the value 1 0000, which is the address for the first instruction in extended instruction memory. This action is expressed by the transfer functions RETA(3:0) ← PC(3:0), PC(3:0) ← 0000, IPROC ← 1.

A hardware interrupt allows another program (the interrupt service routine) to be executed at the request of an external input interrupt signal at any time during the execution of the normal program. This type of interrupt is important because it provides a means for an external input interrupt signal to initiate a process for servicing an I/O device. The process for servicing an I/O device is placed in extended instruction memory.

In addition to the assembler directive BIPROC, there is one more assembler directive available for the assembler for VBC1-E. The assembler directive EQU can be used to define a new constant variable. EQU can be used as a simple convenience for writing assembly language code for VBC1-E. The format for using EQU is <variable name> EQU <value>, which can be used for assigning a variable name a given value. The variable name can then be used in the assembly language program. When the program is assembled, the assembler substitutes the value for the variable name in the assembly language code. Assembler directives do not appear in the machine code after it is assembled.

Example:

```
X EQU 5     ; creates the variable 'X' and gives it the value 5
ADDI R0,X   ; adds X(5) to R0;
```

Program 18.8 shows an assembly language program using the assembler directives EQU and BIPROC. Notice that this program is well written, because it does not run amuck or roam through memory.

In the programmer's register model for VBC1-E shown in Figure 18.15, the machine codes for the instructions in Program 18.8 are manually loaded into the instruction memory beginning at address 0 and into the extended instruction memory beginning at address 0. All other instruction memory locations and extended instruction memory locations are cleared. The PC (program counter) is cleared to address 0 in instruction memory, and the IR (instruction register) shows the machine code value at address 0 (00100011). Registers R0 and R1 are cleared, all the outputs are cleared in the programmer's register model. The input at port address 0 (IP0) is set to the value 1111 or 15, and all the other inputs in the programmer's register model are set to 0. Data memory is cleared.

After executing each instruction in Program 18.8, the transfer function form indicates what happens to the various registers of VBC1-E.

```
; Assembler directives
X EQU 3
Y EQU 10

; Instruction Memory
LOADI R0,X
LOADI R1,Y
STORE R1,0
XNOR R1,R1

FETCH R1,0
INT
OUT R0,2
HALT

; Extended Instruction Memory
BIPROC
OUT R0,0
OUT R1,1
IN R0,0    ;set IP0 to 1111
IRET
```

PROGRAM 18.8 An assembly language program using the assembler directives BIPROC and EQU

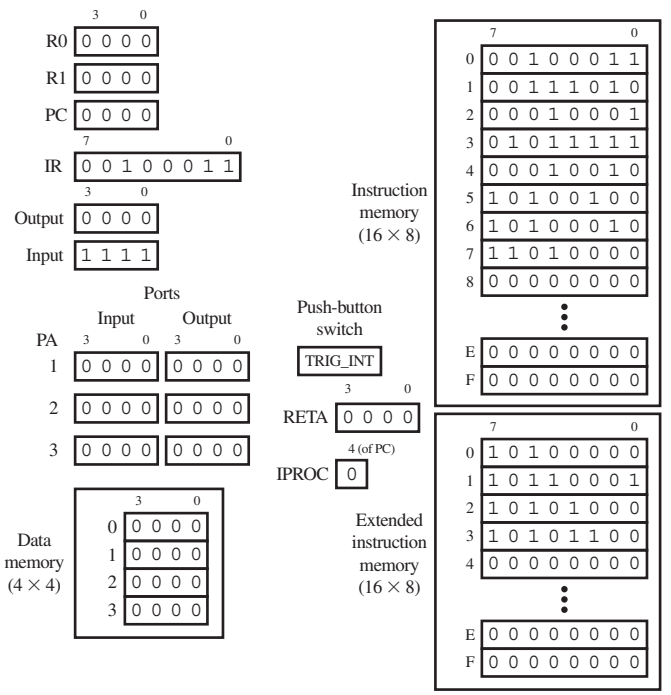


FIGURE 18.15 Result in the programmer's register model after manually loading the instructions in Program 18.8 into VBC1-E instruction memory at address 0 with IP0 set to 15

The following is a detailed description for the execution of each instruction in Program 18.8. When the assembler directive X EQU 3 is performed, X is assigned the value 3. When the assembler directive Y EQU 10 is performed, Y is assigned the value 10. When the instruction LOADI R0,X is executed, its transfer function performs the operation R0 ← 3, which places the value 0011(3) into R0. When the instruction LOADI R1,Y is executed, its transfer function performs the operation R1 ← 10, which places the value 1010(10) into R1. When the instruction STORE R1,0 is executed, its transfer function performs the operation DM0 ← R1, which places the value 1010(10) into data memory 0. When the instruction XNOR R1,R1 is

executed, its transfer function performs the operation R1 ← 1111(15), which places 1111(15) into R1. When the instruction FETCH R1,0 is executed, its transfer function performs the operation R1 ← DM0, which places 1010(10) into R1. When the instruction INT is executed, its transfer function performs the following operations: (1) RETA ← PC + 1, which places the return address 0110(6) into RETA; (2) PC ← 0000, which transfers the address 0000(0) in extended instruction memory into the PC; and (3) IPROC ← 1, which sets IPROC to 1. When the instruction OUT R0,0 is executed, its transfer function performs the operation OP0 ← R0, which places the value 0011(3) in output port 0. When the instruction OUT R1,1 is executed, its transfer function performs the operation OP1 ← R1, which places the value 1010(10) in output port 1. When the instruction IN R0,0 is executed, its transfer function performs the operation R0 ← IP0, which places the value 1111(15) into R0. When the instruction IRET is executed, its transfer function performs the following operations: (1) PC ← RETA, which places the return address 0110(6) into PC, and (2) IPROC ← 0, which sets IPROC to 0. When the instruction OUT R0, 2 is executed, its transfer function performs the operation OP2 ← R0, which places the value 1111(15) in output port 2. When the instruction HALT is executed, its transfer function performs the operation PC ← PC, which loads the PC with the address of the instruction HALT.

So, in Figure 18.16, because X is equal to 3, after executing the instruction LOADI R0, X, the value 0011(3) is transferred to register R0. Because Y is equal to 10, after executing the instruction LOADI R1,Y, the value 1010(10) is transferred to R1. After executing the instruction STORE R1,0, the value 1010(10) is transferred to data memory 0. After executing the instruction XNOR R1,R1, the value 1111(15) is transferred to R1. After executing the instruction FETCH R1,0, 1010(10) is transferred to R1. After executing the instruction INT, the return address 0110(6) is transferred to RETA, the address 0000(0) is transferred to PC, and 1 is transferred to IPROC. After executing the instruction OUT R0,0, the value 0011(3) is transferred to output port 0. After executing the instruction OUT R1,1, the value 1010(10) is transferred to output port 1. After executing the instruction IN R0,0, the value 1111(15) is transferred to R0. After executing the instruction IRET, the return address 0110(6) is transferred to PC, and 0 is transferred to IPROC. After the instruction OUT R0,2, the value 1111(15) is transferred to output port 2. After executing the instruction HALT, the program loads the PC with the address of the instruction HALT. The

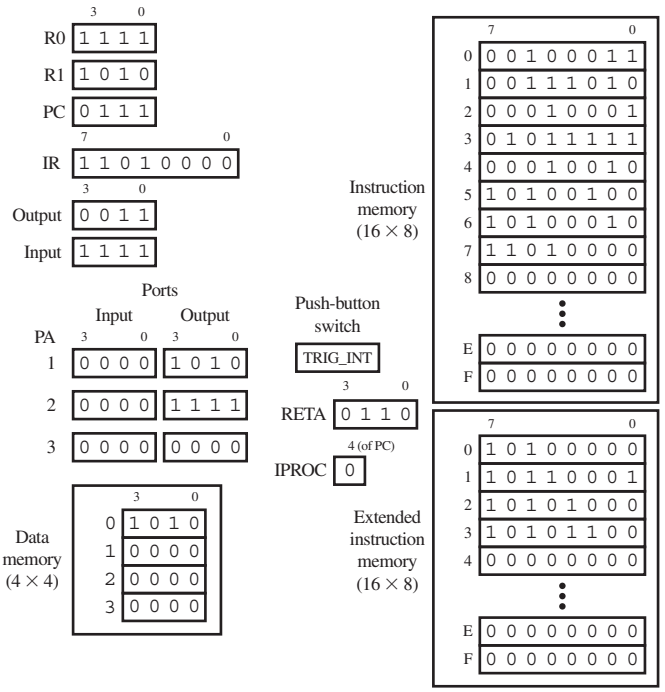


FIGURE 18.16 Result in the programmer's register model after executing the instructions in Program 18.8 for VBC1-E

PC is now pointing to the address of the next instruction to be executed, which is 7, and the IR now contains the machine code for the next instruction at address 7, which is 11010000.

18.9 COMPLETE INSTRUCTION SET SUMMARY FOR VBC1-E

Table 18.10 shows the complete instruction set summary for VBC1-E as a handy reference. The instructions are provided in alphabetical order.

TABLE 18.10
Complete instruction set summary for VBC1-E in alphabetical order

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF) 7	6	5	4	3	2	1	0
ADD DR,SR	DR ← DR + SR	0	1	0	0/1	0/1	0	0	0
ADDI DR,Data	DR ← DR + IR(3:0)	0	1	1	0/1	D	D	D	D
AND DR,SR	DR ← DR ∧ SR	0	1	0	0/1	0/1	1	0	1
FETCH DR,Addr	DR ← DM[IR(3:2)]	0	0	0	0/1	A	A	1	0
HALT	PC(3:0) ← PC(3:0)	1	1	0	1	0	0	0	0
IN DR,Port_A	DR ← IP[IR(1:0)]	1	0	1	0/1	1	0	IPA	IPA
INT	RETA(3:0) ← PC(3:0) + 1 PC(3:0) ← 0000, IPROC ← 1	1	0	1	0	0	1	0	0
IRET	PC(3:0) ← RETA(3:0) IPROC ← 0	1	0	1	0	1	1	0	0
JMP Addr	PC(3:0) ← IR(3:0)	1	1	0	0	A	A	A	A
JMPR OffSet	PC(3:0) ← PC(3:0) + IR(3:0)	1	1	0	1	OS	OS	OS	OS
JNZ DR,Address	PC(3:0) ← IR(3:0), if DR ≠ 0 else PC(3:0) ← PC(3:0) + 1	1	1	1	0/1	A	A	A	A
LOADI DR,Data	DR ← IR(3:0)	0	0	1	0/1	D	D	D	D
MOV DR,SR	DR ← SR	0	0	0	0/1	0/1	0	0	0
NOT DR,SR	DR ← !SR	0	1	0	0/1	0/1	1	0	0
OR DR,SR	DR ← DR ∨ SR	0	1	0	0/1	0/1	1	1	0
OUT DR,Port_A	OP[IR(1:0)] ← DR	1	0	1	0/1	0	0	OPA	OPA
RL DR,SR	DR ← SR(2:0) SR(3)	1	0	0	0/1	0/1	1	0	1
RR DR,SR	DR ← SR(0) SR(3:1)	1	0	0	0/1	0/1	1	0	0
SL0 DR,SR	DR ← SR(2:0) 0	1	0	0	0/1	0/1	0	1	0
SL1 DR,SR	DR ← SR(2:0) 1	1	0	0	0/1	0/1	0	1	1
SR0 DR,SR	DR ← 0 SR(3:1)	1	0	0	0/1	0/1	0	0	0
SR1 DR,SR	DR ← 1 SR(3:1)	1	0	0	0/1	0/1	0	0	1
STORE DR,Addr	DM[IR(3:2)] ← DR	0	0	0	0/1	A	A	0	1
SUB DR,SR	DR ← DR − SR	0	1	0	0/1	0/1	0	0	1
XNOR DR,SR	DR ← !(DR ⊕ SR)	0	1	0	0/1	0/1	1	1	1

Hardware interrupt: Signal *TRIG_INT* causes RETA(3:0) ← PC(3:0), PC(3:0) ← 0000, IPROC ← 1

PROBLEMS

Section 18.1 Introduction

18.1 Does VBC1-E have a data memory? If so, name the instructions that access the data memory.
18.2 Does VBC1-E have logic instructions? If so, name the logic instructions.
18.3 Does VBC1-E have unconditional jump instructions? If so, name the unconditional jump instructions.
18.4 Does VBC1-E have interrupt capability? If so, name the instructions that provide software interrupt capability.

Section 18.2 Instruction Summary

18.5 How many of the instructions in VBC1 are the same in VBC1-E?
18.6 List the instructions that are the same in VBC1 and VBC1-E.
18.7 What is the trick to changing some of the instructions for VBC1 to obtain new instructions for VBC1-E?
18.8 How many instructions in VBC1 have unused machine code bits?
18.9 How many new instructions are there, and what is the total number of instructions for VBC1-E?
18.10 What value is assigned to the unused bits in the machine code form for VBC1?
18.11 List the OPCODE for the STORE instructions for VBC1-E. List the extension bits for the STORE instruction.
18.12 List the OPCODE for the OR instruction for VBC1-E. List the extension bits for the OR instruction.
18.13 List the OPCODE for the RR instruction for VBC1-E. List the extension bits for the RR instruction.
18.14 List the OPCODE for the OUT instruction for VBC1-E. List the extension bits for the OUT instruction.
18.15 List the OPCODE for the JMP instruction for VBC1-E. List the extension bit for the JMP instruction.

Section 18.3 Input, Output, and Interrupt Instructions

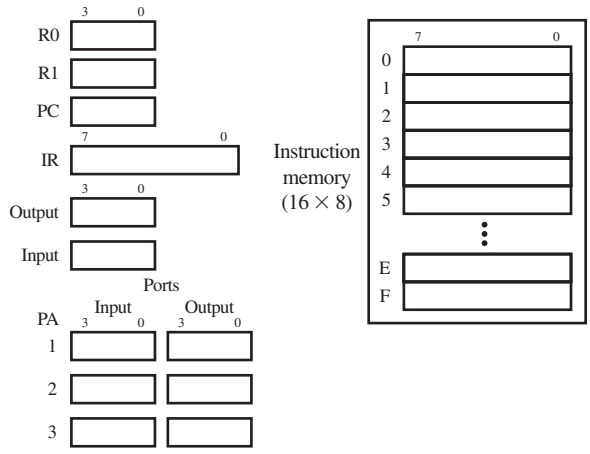
18.16 The IN instruction for VBC1 has been modified so that it contains OPCODE bits 7:5 and extension OPCODE bits 3:2. What is the OPCODE for the IN instruction for VBC1-E?
18.17 Which bits in the IN instruction are used for the input port address?
18.18 How many input ports can be used with the IN instruction for VBC1-E?
18.19 The OUT instruction for VBC1 has been modified so that it contains OPCODE bits 7:5 and extension OPCODE bits 3:2. What is the OPCODE for the OUT instruction for VBC1-E?
18.20 Which bits in the OUT instruction are used for the output port address?
18.21 How many output ports can be used with the OUT instruction for VBC1-E?
18.22 Two new instructions use 101 as the OPCODE bits 7:5 with 01 and 11, respectively, as the extension OPCODE bits 3:2. What are the names of these new instructions for VBC1-E?
18.23 What type of operations do the new instructions INT and IRET represent?
18.24 What are the input port addresses for the IN instruction?
18.25 Write the transfer function form for the instruction IN DR,Port_A.
18.26 Specify what occurs when the instruction IN R0,3 is executed.
18.27 Where are the instructions that must be executed by VBC1-E placed, and how many bits make up each instruction?
18.28 What are the output port addresses for the OUT instruction?
18.29 Write the transfer function form for the instruction OUT DR,Port_A.
18.30 Specify what occurs when the instruction OUT R1,1 is executed.
18.31 Table P18.31 shows a partial instruction set for VBC1-E. Hand assemble the program in Program P18.31. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P18.31 after the last instruction JNZ is executed the first time. Set input port 1 to 0001 or 1 before executing the program.

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
IN DR,Port_A	DR ← IP[IR(1:0)]	7:5 OPCODE; Dest.Reg; 3:2 OPCODE Extension; 1:0 Port Address 7 6 5 4 3 2 1 0 1 0 1 0/1 1 0 IPA IPA 0 = R0, 1 = R1
OUT DR,Port_A	OP[IR(1:0)] ← DR	7 6 5 4 3 2 1 0 1 0 1 0/1 0 0 OPA OPA 0 = R0, 1 = R1
LOADI DR,Data	DR ← IR(3:0)	7:5 OPCODE; Dest.Reg; 3:0 Immediate Data (Source) 7 6 5 4 3 2 1 0 0 0 1 0/1 D D D D 0 = R0, 1 = R1
ADDI DR,Data	DR ← DR + IR(3:0)	7 6 5 4 3 2 1 0 0 1 1 0/1 D D D D 0 = R0, 1 = R1
JNZ DR,Address	PC(3:0) ← IR(3:0), if DR ≠ 0 else PC(3:0) ← PC(3:0) + 1	3:0 Address (Destination) 7 6 5 4 3 2 1 0 1 1 1 0/1 A A A A 0 = R0, 1 = R1

TABLE P18.31

```
back:
in r0,1
addi r0,14
out r0,0
loadi r0,1
jnz r0,back
```

PROGRAM P18.31



REGISTER_MODEL P18.31

Section 18.4 Data Memory Instructions

18.32 The MOV instruction for VBC1 has been modified so that it contains OPCODE bits 7:5 and extension OPCODE bits 1:0. What is the OPCODE for the MOV instruction for VBC1-E?

18.33 With the modified MOV instruction and its extension bits, how many new instructions can be added?

18.34 What are the new instructions that use OPCODE bits 7:5 as 000 and extension OPCODE bits 1:0 of 01 and 10, respectively?

18.35 Write the transfer function form for the instruction STORE DR,Addr.

18.36 Specify what occurs when the instruction STORE R0, 1 is executed.

18.37 Write the transfer function form for the instruction FETCH DR,Addr.

18.38 Specify what occurs when the instruction FETCH R1, 0 is executed.

18.39 How many addresses does the data memory contain for VBC1-E? List the addresses for data memory.

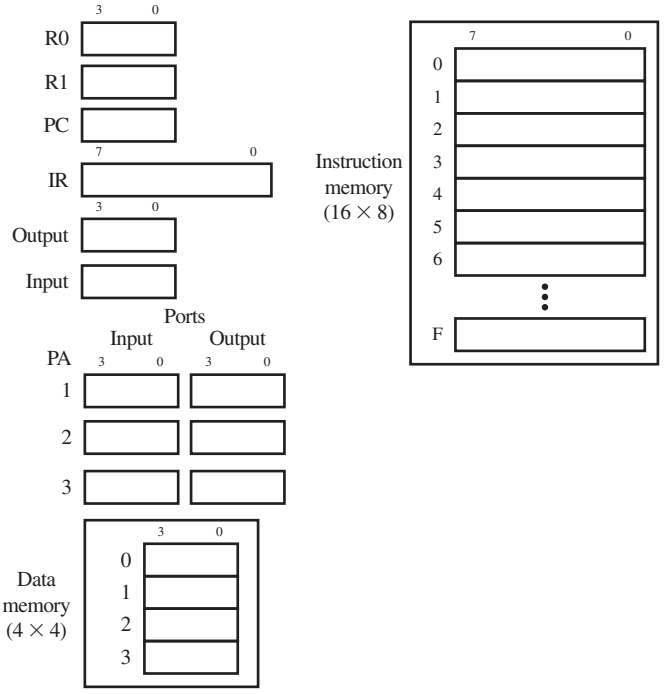
18.40 Table P18.40 shows a partial instruction set for VBC1-E. Hand assemble the program in Program P18.40. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P18.40 after the last instruction OUT is executed.

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
LOADI DR,Data	DR ← IR(3:0)	7:5 OPCODE; Dest.Reg (4); 3:0 Immediate Data (Source). 7 6 5 4 3 2 1 0: 0 0 1 0/1 D D D D. 0 = R0, 1 = R1
OUT DR,Port_A	OP[IR(1:0)] ← DR	7:5 OPCODE; Dest.Reg (4); 3:2 OPCODE Extension; 1:0 Port Address. 7 6 5 4 3 2 1 0: 1 0 1 0/1 0 0 OPA OPA. 0 = R0, 1 = R1
MOV DR,SR	DR ← SR	7:5 OPCODE; Dest.Reg (4); Source Reg (3); 1:0 OPCODE Extension. 7 6 5 4 3 2 1 0: 0 0 0 0/1 0/1 0 0 0. 0 = R0, 1 = R1
STORE DR,Addr	DM[IR(3:2)] ← DR	7 6 5 4 3 2 1 0: 0 0 0 0/1 A A 0 1. 0 = R0, 1 = R1; 3:2 Addr of Data Memory (DM)
FETCH DR,Addr	DR ← DM[IR(3:2)]	7 6 5 4 3 2 1 0: 0 0 0 0/1 A A 1 0. 0 = R0, 1 = R1; 3:2 Addr of Data Memory (DM)

TABLE P18.40

```
loadi r0,6
store r0,0
fetch r1,0
loadi r0,0
mov r0,r1
out r0,1
```

PROGRAM P18.40



REGISTER_MODEL P18.40

Section 18.5 Arithmetic and Logic Instructions

18.41 The ADD instruction for VBC1 has been modified so that it contains OPCODE bits 7:5 and extension OPCODE bits 2:0. What is the OPCODE for the ADD instruction for VBC1-E?

18.42 With the modified ADD instruction and its extension bits, how many new instructions can be added?

18.43 What are the new instructions that use OPCODE bits 7:5 and extension OPCODE bits 2:0 of 001, 100, 101, 110, and 111, respectively?

18.44 Write the transfer function form for the instruction SUB DR,SR.

18.45 Specify what occurs when the instruction SUB R0,R1 is executed.

18.46 Write the transfer function form for the instruction NOT DR,SR.

18.47 Specify what occurs when the instruction NOT R1,R0 is executed.

18.48 Write the transfer function form for the instruction AND DR,SR.

18.49 Specify what occurs when the instruction AND R0,R1 is executed.

18.50 Write the transfer function form for the instruction OR DR,SR.

18.51 Specify what occurs when the instruction OR R1,R0 is executed.

18.52 Write the transfer function form for the instruction XNOR DR,SR.

18.53 Specify what occurs when the instruction XNOR R0,R1 is executed.

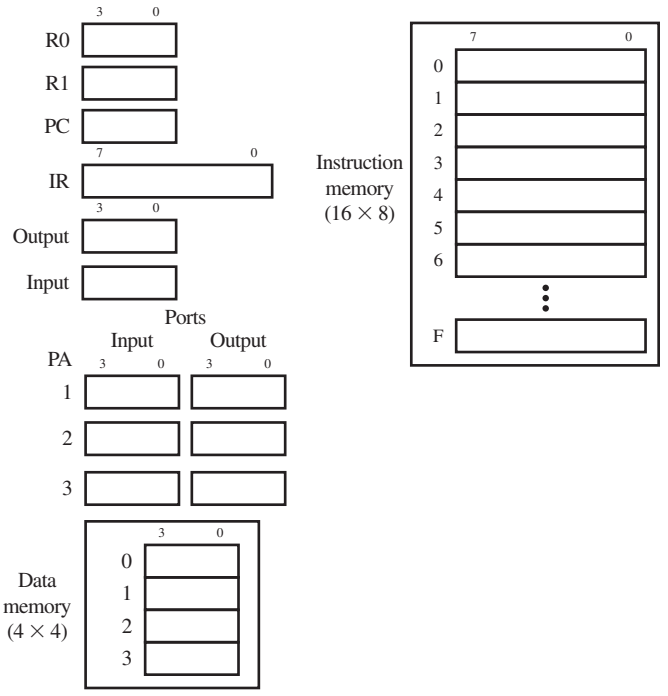
18.54 Table P18.54 shows a partial instruction set for VBC1-E. Hand assemble the program in Program P18.54. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P18.54 after the last instruction OUT is executed.

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
LOADI DR,Data	DR ← IR(3:0)	7:5 OPCODE; Dest.Reg (0 = R0, 1 = R1); 3:0 Immediate Data (Source) 7 6 5 4 3 2 1 0 0 0 1 0/1 D D D D
OUT DR,Port_A	OP[IR(1:0)] ← DR	7:5 OPCODE; Dest.Reg (0 = R0, 1 = R1); 3:2 OPCODE Extension; 1:0 Port Address 7 6 5 4 3 2 1 0 1 0 1 0/1 0 0 OPA OPA
SUB DR,SR	DR ← DR − SR	7:5 OPCODE; Dest.Reg (0 = R0, 1 = R1); Source Reg (0 = R0, 1 = R1); 2:0 OPCODE Extension 7 6 5 4 3 2 1 0 0 1 0 0/1 0/1 0 0 1
AND DR,SR	DR ← DR ∧ SR	7 6 5 4 3 2 1 0 0 1 0 0/1 0/1 1 0 1 (0 = R0, 1 = R1; 0 = R0, 1 = R1)

TABLE P18.54

```
loadi r0,4
loadi r1,10
sub r1,r0
out r1,0
and r1,r0
out r1,1
```

PROGRAM P18.54



REGISTER_MODEL P18.54

Section 18.6 Shift and Rotate Instructions

18.55 The SR0 instruction for VBC1 has been modified so that it contains OPCODE bits 7:5 and extension OPCODE bits 2:0. What is the OPCODE for the SR0 instruction for VBC1-E?

18.56 With the modified SR0 instruction and its extension bits, how many new instructions can be added?

18.57 What are the new instructions that use OPCODE bits 7:5 and extension OPCODE bits 2:0 of 001, 010, 011, 100, and 101, respectively?

18.58 Write the transfer function form for the instruction SR1 DR,SR.

18.59 Specify what occurs when the instruction SR1 R1,R0 is executed.

18.60 Write the transfer function form for the instruction SL0 DR,SR.

18.61 Specify what occurs when the instruction SL0 R0,R1 is executed.

18.62 Write the transfer function form for the instruction SL1 DR,SR.

18.63 Specify what occurs when the instruction SL1 R1,R0 is executed.

18.64 Write the transfer function form for the instruction RR DR,SR.

18.65 Specify what occurs when the instruction RR R0,R1 is executed.

18.66 Write the transfer function form for the instruction RL DR,SR.

18.67 Specify what occurs when the instruction RL R1,R0 is executed.

18.68 Table P18.68 shows a partial instruction set for VBC1-E. Hand assemble the program in Program P18.68. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P18.68 after the last instruction OUT is executed.

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
LOADI DR,Data	DR ← IR(3:0)	7:5 OPCODE; Dest.Reg; 3:0 Immediate Data (Source)
OUT DR,Port_A	OP[IR(1:0)] ← DR	7:5 OPCODE; Dest.Reg; 3:2 OPCODE Extension; 1:0 Port Address
SR1 DR,SR	DR ← 1 SR(3:1)	7:5 OPCODE; Dest.Reg; Source Reg; 2:0 OPCODE Extension
RL DR,SR	DR ← SR(2:0) SR(3)	7:5 OPCODE; Dest.Reg; Source Reg; 2:0 OPCODE Extension

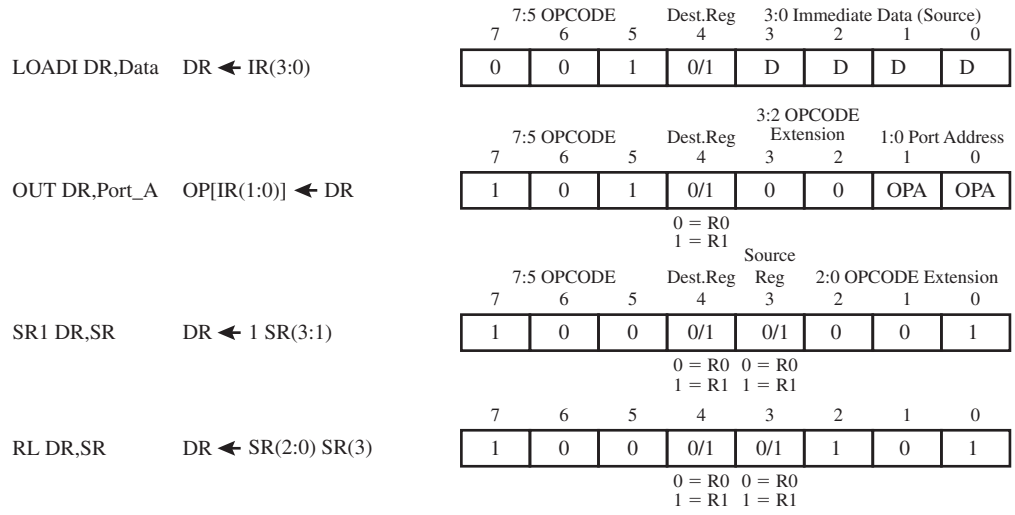
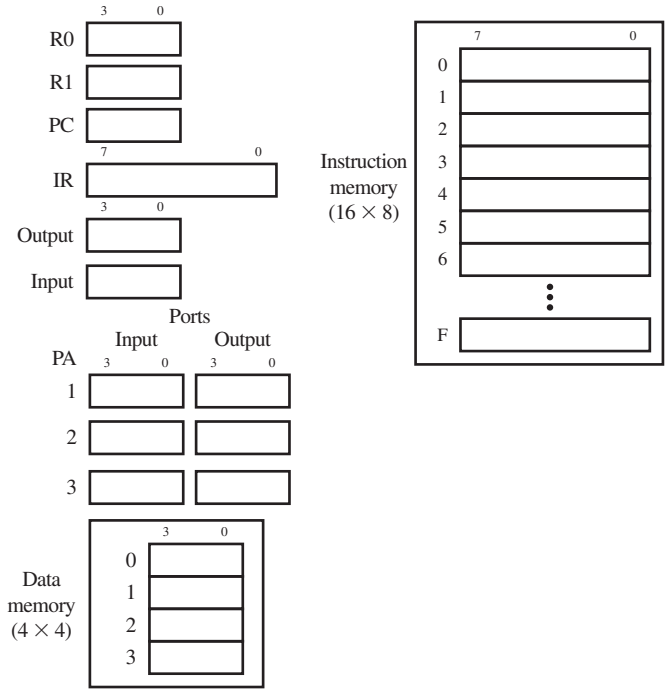


TABLE P18.68

```
loadi r0,6h
sr1 r0,r0
out r0,0
loadi r1,8h
rl r1,r1
out r1,1
```

PROGRAM P18.68



REGISTER_MODEL P18.68

Section 18.7 Jump, Jump Relative, and Halt Instructions

18.69 The OUT instruction in VBC1 has the OPCODE 110. In VBC1-E the OUT instruction was modified to use OPCODE 101 with extension bits 3:2. This freed up OPCODE 110 to be used for new instructions for VBC1-E. List the new instructions.

18.70 Is the new JMP instruction conditional or unconditional?

18.71 Write the transfer function form for the instruction JMP Addr.

18.72 Specify what occurs when the instruction JMP 9 is executed.

18.73 Write the transfer function form for the instruction JMPR OffSet.

18.74 Specify what occurs when the instruction JMPR 5 is executed.

18.75 Write the transfer function form for the instruction HALT.

18.76 Specify what occurs when the instruction HALT is executed.

18.77 Table P18.77 shows a partial instruction set for VBC1-E. Hand assemble the program in Program P18.77. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P18.77 after the last instruction JMPR is executed the second time.

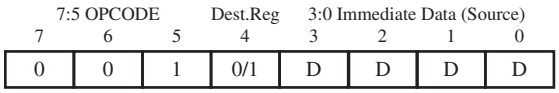
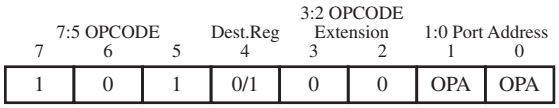
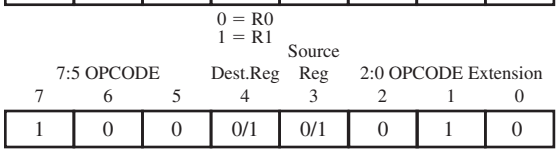
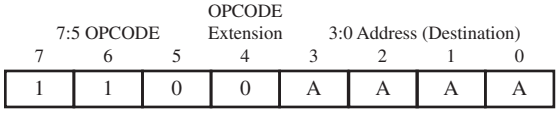
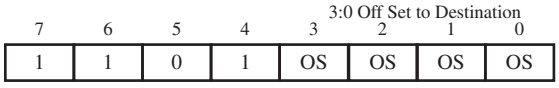
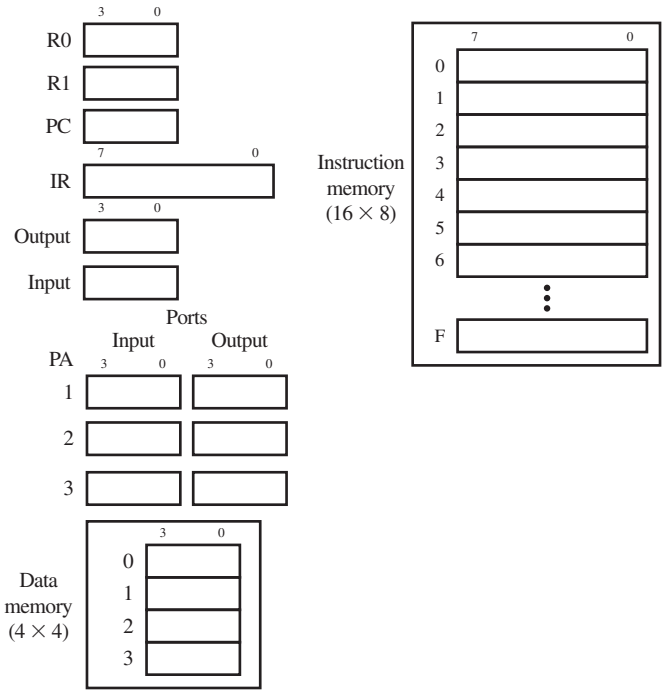
Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
LOADI DR,Data	DR ← IR(3:0)	
OUT DR,Port_A	OP[IR(1:0)] ← DR	
SL0 DR,SR	DR ← SR(2:0) 0	
JMP Addr	PC(3:0) ← IR(3:0)	
JMPR OffSet	PC(3:0) ← PC(3:0) + IR(3:0)	

TABLE P18.77

```
loadi r0,3
jmp 3
sl0 r1,r0
out r0,0
out r1,1
jmpr 13
```

PROGRAM P18.77



REGISTER_MODEL P18.77

18.78 Is program P18.77 in Problem 18.77 well written—that is, does it run amuck or roam through memory when it is executed?

Section 18.8 More about Interrupts and Assembler Directives

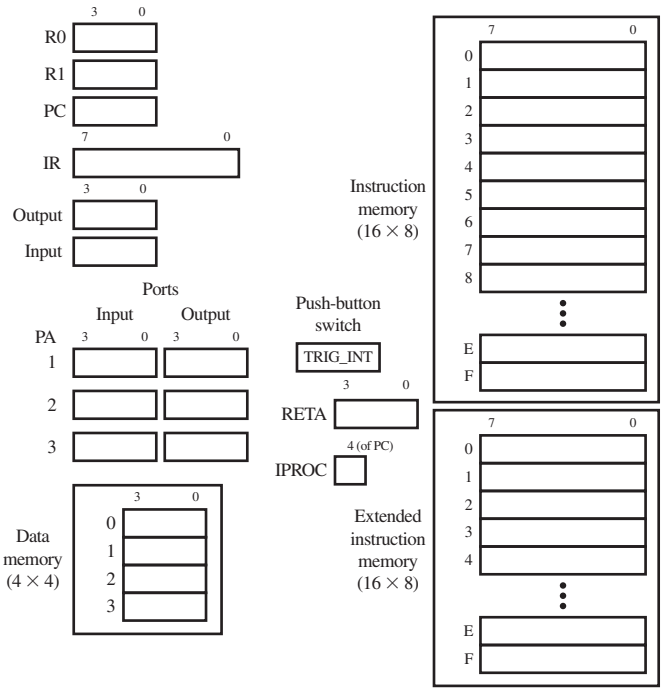
18.79 Write the transfer function form for the instruction INT.
18.80 Specify what occurs when the instruction INT is executed.
18.81 Write the transfer function form for the instruction IRET.
18.82 Specify what occurs when the instruction IRET is executed.
18.83 What is the purpose of the assembler directive BIPROC?
18.84 List the address range for section 0 for instruction memory.
18.85 List the address range for section 1 for extended instruction memory.
18.86 How many instructions can be placed in extended instruction memory?
18.87 Can the instruction IRET be placed outside of extended instruction memory?
18.88 Can the instruction INT be placed inside extended instruction memory?
18.89 Can the assembler directive BIPROC be used more than once in a program?
18.90 When an external input interrupt signal *TRIG_INT* is applied, what does this cause?
18.91 Write the transfer function form for a hardware interrupt.
18.92 Specify what occurs when an external input interrupt signal *TRIG_INT* is applied.
18.93 Show the format for the assembler directive EQU. Provide an example of how to use the EQU assembler directive with the instruction LOADI for a constant variable Y = 7.
18.94 Table P18.94 shows a partial instruction set for VBC1-E. Hand assemble the program in Program P18.94. Load the program beginning at address 0, and show all the values in the programmer's register model in Register_Model P18.94 after the last instruction HALT is executed.
18.95 Is Program P18.94 in problem 18.94 well written—that is, does it run amuck or roam through memory when it is executed?

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
LOADI DR,Data	DR ← IR(3:0)	7:5 OPCODE; Dest.Reg; 3:0 Immediate Data (Source) 7 6 5 4 3 2 1 0 0 0 1 0/1 D D D D
OUT DR,Port_A	OP[IR(1:0)] ← DR	7:5 OPCODE; Dest.Reg; 3:0 OPCODE Extension; 1:0 Port Address 7 6 5 4 3 2 1 0 1 0 1 0/1 0 0 OPA OPA 0 = R0 1 = R1
INT	RETA(3:0) ← PC(3:0) + 1 PC(3:0) ← 0000 IPROC ← 1	7 6 5 4 3 2 1 0 1 0 1 0 0 1 0 0 Bit 4 not used; Bit 1:0 not used
HALT	PC(3:0) ← PC(3:0)	4:0 OPCODE Extension 7 6 5 4 3 2 1 0 1 1 0 1 0 0 0 0
OR DR,SR	DR ← DR ∨ SR	7 6 5 4 3 2 1 0 0 1 0 0/1 0/1 1 1 0 0 = R0 0 = R0 1 = R1 1 = R1
IRET	PC(3:0) ← RETA(3:0) IPROC ← 0	7 6 5 4 3 2 1 0 1 0 1 0 1 1 0 0 Bit 4 not used; Bit 1:0 not used

TABLE P18.94

```
;assembler directive
z equ 7

;main program in Instruction Memory
loadi r0,z
out r0,0
int
halt

;interrupt process in Extended
;Instruction Memory
BIPROC
loadi r1,8
or r0,r1
out r0,1
iret
```

PROGRAM P18.94



REGISTER_MODEL P18.94

19 Chapter

Designing Input/Output Circuits for VBC1-E

Chapter Outline

19.1 INTRODUCTION

This is the first in a series of chapters that teaches you how to expand the design of VBC1 to make VBC1-E, which is an enhanced version of VBC1. In this chapter, you will learn how to design the input circuit for VBC1-E. You will also learn how to design the output circuit for VBC1-E. The input instruction for VBC1-E requires a port number to specify one of four separate input ports, and the output instruction for VBC1-E also requires a port number to specify one of four separate output ports. This allows VBC1-E to input data from four separate locations and output data to four separate locations. Port numbers for inputs and outputs increase the functionality of VBC1-E compared to VBC1. Enhancing the design for VBC1 to handle a larger number of modified and/or additional instructions is not a giant step, but it does take some planning before writing the code, and it also takes perseverance to verify your design changes work properly by simulating the designs to verify their correct functionality.

19.2 DESIGNING THE INPUT CIRCUIT FOR VBC1-E

The input circuitry for VBC1-E can be performed by a MUX bus steering circuit. Figure 19.1 shows a 4-to-1 MUX bus steering circuit, where *IP* stands for INPUT PORT and *DI* stands for DATA INPUT.

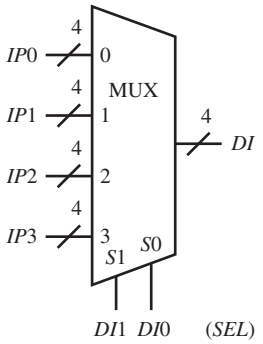


FIGURE 19.1 Schematic for a 4-to-1 MUX bus steering circuit.

The 4-to-1 MUX bus steering circuit in Figure 19.1 can be described by the following concise transfer function:

$DI \leftarrow IP0$ *if* $SEL = 00$ *else*
$DI \leftarrow IP1$ *if* $SEL = 01$ *else*
$DI \leftarrow IP2$ *if* $SEL = 10$ *else*
$DI \leftarrow IP3$

Things you should notice about the transfer function for the 4-to-1 MUX bus steering circuit:

- A transfer occurs based on the *SEL* (SELECT) input.
- The transfer operation $DI \leftarrow IP0$ is only performed when *SEL* is 00.
- The transfer operation $DI \leftarrow IP1$ is only performed when *SEL* is 01.
- The transfer operation $DI \leftarrow IP2$ is only performed when *SEL* is 10.
- The transfer operation $DI \leftarrow IP3$ is only performed when *SEL* is 11.
- A MUX is a combinational logic circuit because there is no clock input.

Listing 19.1 shows a complete VHDL design, for the 4-to-1 MUX bus steering circuit using a behavioral design with a case statement.

LISTING 19.1 Complete VHDL design for a 4-to-1 MUX bus steering circuit using a behavioral design with a case statement

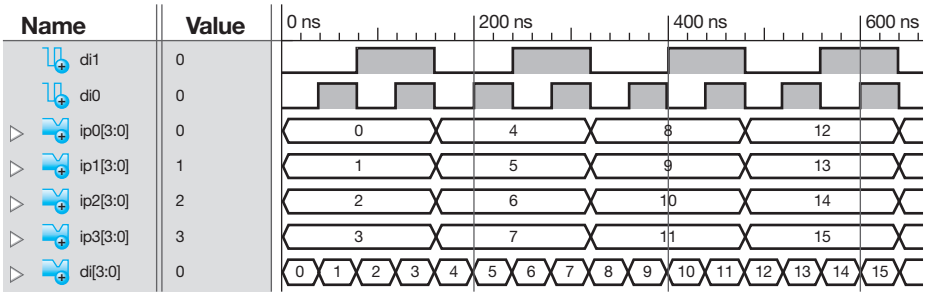
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity BUS_STEERING_CIRCUIT is port (
     di1,di0 : in std_logic;
     ip0, ip1, ip2, ip3 : in std_logic_vector(3 downto 0);
     di : out std_logic_vector(3 downto 0)
     );
end BUS_STEERING_CIRCUIT;

architecture Behavioral of BUS_STEERING_CIRCUIT is
     signal sel :std_logic_vector (1 downto 0);
begin
     sel <= di1&di0;
process (sel,ip0,ip1,ip2,ip3)
begin
     case sel is
          when "00" => di <= ip0;
          when "01" => di <= ip1;
          when "10" => di <= ip2;
          when "11" => di <= ip3;
          when others => null;
     end case;
end process;
end Behavioral;
```

Things you should notice about the VHDL design in Listing 19.1:

- *DI*1 and *DI*0 are scalar signals.
- *IP*3 down to *IP*0 are vector signals with the range 3 downto 0.
- *DI* is a vector signal with the range 3 downto 0.
- *SEL* is an internal vector signal with the range 1 downto 0.
- The concatenation operator & is used to form the internal vector signal *SEL* from the scalar signals *DI*1 and *DI*0. Because *DI*1 is on the left of the concatenation operator, it is the MSB (most significant bit), and DI0 is the LSB (least significant bit) of the vector signal *SEL*.
- A process with a case statement is used to specify the code for the 4-to-1 MUX bus steering circuit.

Waveform 19.1 shows a simulation with the correct functionality of design entity BUS_STEERING_CIRCUIT.

WAVEFORM 19.1
Simulation with the correct functionality of design entity BUS_STEERING_CIRCUIT



Things you should notice about the waveforms in Waveform 19.1:

- All the vector signals are displayed in unsigned decimal values.
- Observe in each case that the output *DI* of the BUS_STEERING_CIRCUIT has the correct value *IP*0 through *IP*3 for *SEL* = *DI*1&*DI*0 = 00 through 11, respectively.

19.3 INSTRUCTION DECODER TRUTH TABLE FOR THE MODIFIED IN INSTRUCTION FOR VBC1-E

For the modified IN instruction to work properly, we must redesign the instruction decoder truth table to provide the necessary control bits, so that the data path unit will operate correctly. Figure 19.2 shows where the modified input circuit in Figure 19.1 must be added to the design of VBC1-E.

As a review, the ALF, TFF, and MCF for the modified IN instruction for VBC1-E are shown in Table 19.1.

TABLE 19.1 The ALF, TFF, and MCF for the IN instruction for VBC1-E

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
IN DR,Port_ A	DR ← IP[IR(1:0)]	7:5 OPCODE: 1 0 1; Dest.Reg (0 = R0, 1 = R1): 0/1; 3:2 OPCODE Extension: 1 0; 1:0 Port Address: IPA IPA

Table 19.2 shows the instruction decoder truth table for the modified IN instruction for VBC1-E.

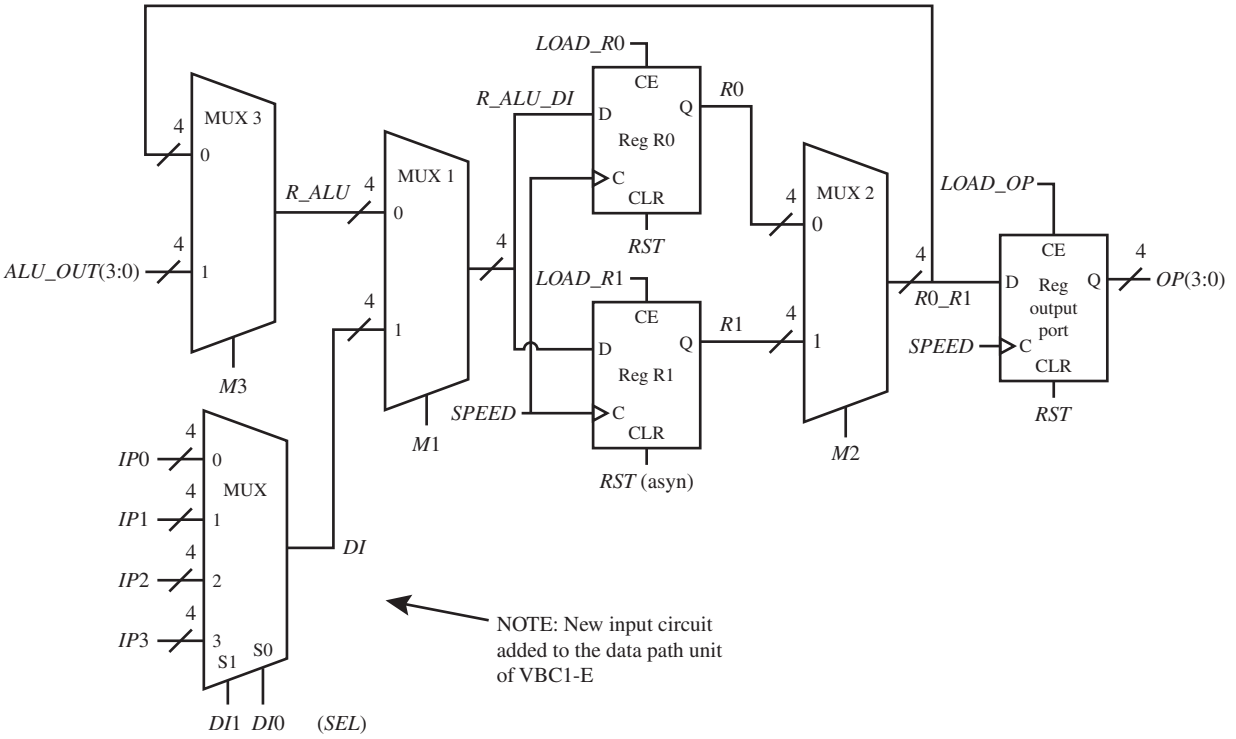


FIGURE 19.2 Modified input circuit added to the Data Path Unit of VBC1-E

TABLE 19. 2 Instruction decoder truth table for the modified IN instruction for VBC1-E

	IR								Control bits				
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M*1**	***LOAD_R*0**	***LOAD_R*1**	**DI1**	**DI0**
IN R0,0	1	0	1	0	1	0	0	0	1	1	0	0	0
IN R0,1	1	0	1	0	1	0	0	1	1	1	0	0	1
IN R0,2	1	0	1	0	1	0	1	0	1	1	0	1	0
IN R0,3	1	0	1	0	1	0	1	1	1	1	0	1	1
IN R1,0	1	0	1	1	1	0	0	0	1	0	1	0	0
IN R1,1	1	0	1	1	1	0	0	1	1	0	1	0	1
IN R1,2	1	0	1	1	1	0	1	0	1	0	1	1	0
IN R1,3	1	0	1	1	1	0	1	1	1	0	1	1	1

Things you should notice about the instruction decoder truth table for the modified IN instruction in Table 19.2:

- The inputs to the instruction decoder for the modified IN instruction are the instruction register bits 7:0 or *IR*(7), *IR*(6), *IR*(5), *IR*(4), *IR*(3), *IR*(2), *IR*(1), and *IR*(0), and the outputs are the control bits *M*1, *LOAD_R*0, *LOAD_R*1, *DI*1, and *DI*0. The values for the *IR* bits for the instruction decoder are the machine code bits for the modified IN instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring

that the control bits perform the transfer function form for the modified IN instruction IN DR,Port_A, which is DR ←IP[IR(1:0)].

- The instruction IN R0,Port_A, which loads the input ports (port addresses 0 through 3) into register R0 via *DI*1 and *DI*0, requires control bit *M*1 to be set to 1 and control bit *LOAD_R*0 to be set to 1. During this instruction, *LOAD_R*1 must be set to 0 so the contents of register R1 do not change.
- The instruction IN R1,Port_A, which loads the input ports (port addresses 0 through 3) into register R1 via *DI*1 and *DI*0, requires control bit *M*1 to be set to 1 and control bit *LOAD_R*1 to be set to 1. During this instruction, *LOAD_R*0 must be set to 0 so the contents of register R0 do not change.

The Boolean equations for the control bits for the modified IN instruction are

$$M1 = 1, LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4),$$
$$DI1 = IR(1), DI0 = IR(0)$$

19.4 DESIGNING THE OUTPUT CIRCUIT FOR VBC1-E

The output circuitry for VBC1-E can be performed by an array of loadable D flip-flops. Figure 19.3 shows an array of four loadable D flip-flops, where *OP* stands for OUTPUT PORT.

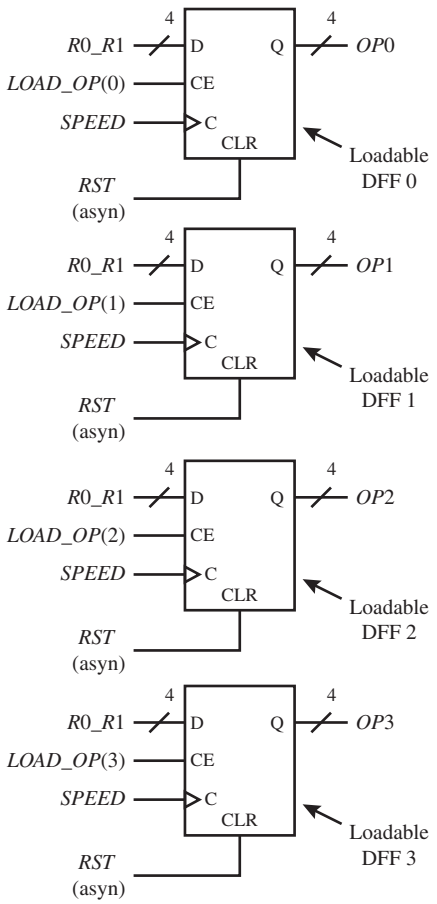


FIGURE 19.3 Array of four loadable D flip-flops, where *OP* stands for OUTPUT PORT

Things you should notice about the schematic in Figure 19.3:

- Four separate outputs are required to provide the output circuitry for VBC1-E.
- All four of the loadable D flip-flops are cleared by the same signal, *RST* (asyn).
- The data input to each loadable D flip-flop is *R*0_*R*1, which is 4 bits.
- All four of the control inputs (C) to the loadable D flip-flops are driven by the clock signal *SPEED*.
- The clock-enable input (CE) to each loadable D flip-flops are enabled by a separate signal *LOAD_OP* (0 through 3).

Each loadable D flip-flop in Figure 19.3 can be described by the following concise transfer function:

$OP \leftarrow 0000$ *if* $RST = 1$ *else*
$OP \leftarrow R0\_R1$ *if* $(LOAD\_OP = 1$ *and* $\uparrow SPEED)$ *else*
$OP \leftarrow OP$

Things you should notice about the transfer function for each loadable D flip-flop:

- The directed arrow shows the direction of the transfer.
- Because *RST* precedes the clock signal *SPEED*, it is an asynchronous input and overrides the clock signal *SPEED*—that is, it is independent of the clock signal *SPEED*.
- *R*0_*R*1 is transferred to *OP* only if $LOAD\_OP = 1$ at the next rising edge of the clock signal *SPEED*.
- *OP* retains its current value if $RST = 0$, or $LOAD\_OP = 0$, or between clock ticks.

Listing 19.2 shows the complete VHDL design, for the array of four loadable D flip-flops using a behavioral design with if statements.

LISTING 19.2
Complete VHDL design for the array of four loadable D flip-flops using a behavioral design with if statements

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity LOADABLE_DFFs is port (
     rst, speed : in std_logic;
     r0_r1, load_op : in std_logic_vector(3 downto 0);
     op0, op1, op2, op3 : out std_logic_vector(3 downto 0)
     );
end LOADABLE_DFFs;

architecture Behavioral of LOADABLE_DFFs is
begin
process (rst, speed)
begin

--loadable DFF 0
     if rst = '1' then op0 <= "0000";
     elsif (load_op(0) = '1' and rising_edge (speed)) then op0 <=
                                                          r0_r1;
     end if;

--loadable DFF 1
     if rst = '1' then op1 <= "0000";
     elsif (load_op(1) = '1' and rising_edge (speed)) then op1 <=
                                                          r0_r1;
     end if;
                                                        (Continued)
```

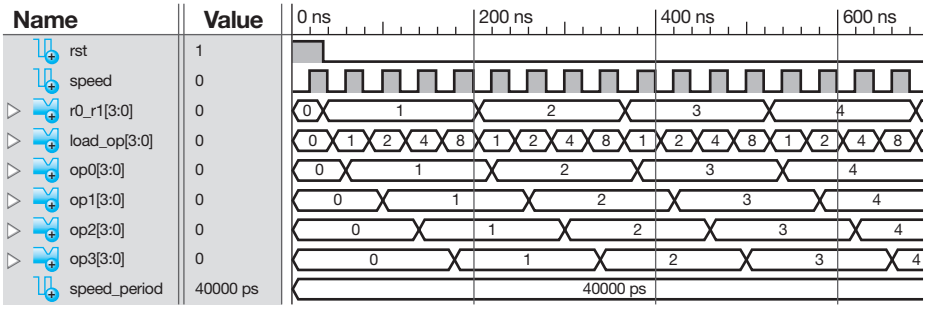
```
--loadable DFF 2
    if rst = '1' then op2 <= "0000";
    elsif (load_op(2) = '1' and rising_edge (speed)) then op2 <=
                                                        r0_r1;
    end if;

--loadable DFF 3
    if rst = '1' then op3 <= "0000";
    elsif (load_op(3) = '1' and rising_edge (speed)) then op3 <=
                                                        r0_r1;
    end if;
end process;
end Behavioral;
```

Things you should notice about the VHDL design in Listing 19.2:

- Both *R*0_*R*1 and *OP*0 through *OP*3 are data type std_logic_vector (3 **downto** 0), which explicitly shows that these are 4-bit buses.
- To transfer *R*0_*R*1 to *OP*0 through *OP*3, both *LOAD_OP* and rising_edge (SPEED) must be true.
- Four separate loadable D flip-flops are required because each one is enabled by a different *LOAD_OP* signal to provide an output to a different output port.

Waveform 19.2 shows a simulation with the correct functionality of design entity LOADABLE_DFFs.

WAVEFORM 19.2
Simulation with the correct functionality of design entity LOADABLE_DFFs



Things you should notice about the waveforms in Waveform 19.2:

- All the vector signals are displayed in unsigned decimal values.
- At the rising edge of the clock signal SPEED, observe that the outputs *OP*0(3:0), *OP*1(3:0), *OP*2(3:0), and *OP*3(3:0) load the correct value of *R*0_*R*1 = 1 for *LOAD_OP*(3:0); load the correct value of *R*0_*R*1 = 2 for *LOAD_OP*(3:0); load the correct value of *R*0_*R*1 = 3 for *LOAD_OP*(3:0), and load the correct value of *R*0_*R*1 = 4 for *LOAD_OP*(3:0), respectively.
- This partial simulation shows that design entity LOADABLE_DFFs is functionally correct.

19.5 INSTRUCTION DECODER TRUTH TABLE FOR THE MODIFIED OUT INSTRUCTION FOR VBC1-E

For the modified OUT instruction to work properly, we must redesign the instruction decoder truth table to provide the necessary control bits, so that the data path unit will operate correctly. Figure 19.4 shows where the modified output circuit in Figure 19.3 must be added to the design of VBC1-E.

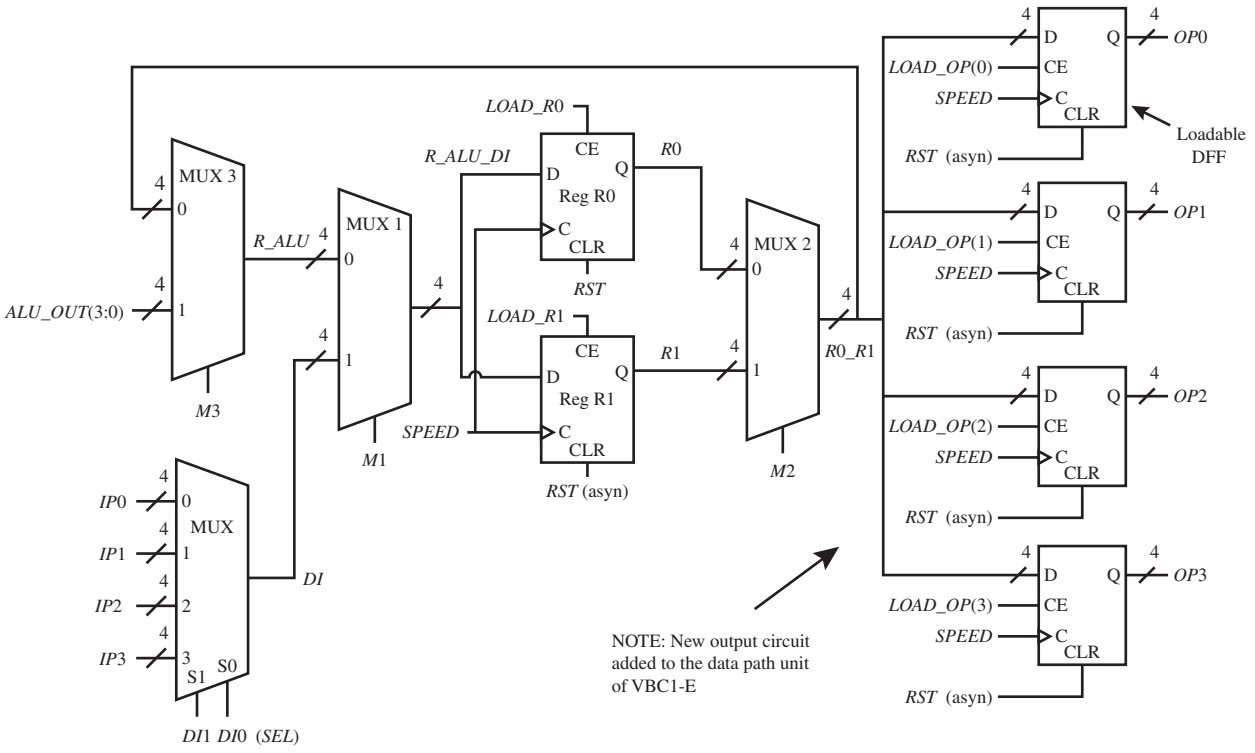


FIGURE 19.4 Modified output circuit added to the data path unit of VBC1-E

As a review, the ALF, TFF, and MCF for the modified OUT instruction for VBC1-E are shown in Table 19.3.

TABLE 19.3 The ALF, TFF, and MCF for the OUT instruction for VBC1-E

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)
OUT DR,Port_ A	OP[IR(1:0)] ← DR	7:5 OPCODE (7 6 5) = 1 0 1; Dest.Reg (4) = 0/1 (0 = R0, 1 = R1); 3:2 OPCODE Extension (3 2) = 0 0; 1:0 Port Address (1 0) = OPA OPA

Table 19.4 shows the instruction decoder truth table for the modified OUT instruction for VBC1-E.

TABLE 19.4 Instruction decoder truth table for the modified OUT instruction for VBC1-E

	IR								Control bits				
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M2***	***LOAD_OP*(0)**	***LOAD_OP*(1)**	***LOAD_OP*(2)**	***LOAD_OP*(3)**
OUT R0,0	1	0	1	0	0	0	0	0	0	1	0	0	0
OUT R0,1	1	0	1	0	0	0	0	1	0	0	1	0	0
OUT R0,2	1	0	1	0	0	0	1	0	0	0	0	1	0
OUT R0,3	1	0	1	0	0	0	1	1	0	0	0	0	1
OUT R1,0	1	0	1	1	0	0	0	0	1	1	0	0	0
OUT R1,1	1	0	1	1	0	0	0	1	1	0	1	0	0
OUT R1,2	1	0	1	1	0	0	1	0	1	0	0	1	0
OUT R1,3	1	0	1	1	0	0	1	1	1	0	0	0	1

Things you should notice about the instruction decoder truth table for the modified OUT instruction in Table 19.4:

- The values for the *IR* bits for the instruction decoder are the machine code bits for the modified OUT instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the modified OUT instruction OUT DR,Port_A, which is OP[IR(1:0)] ← DR.
- The instruction OUT R0,Port_A, which loads the register R0 into the output ports (port addresses 0 through 3), requires control bit *M*2 to be set to 0.
- The instruction OUT R1,Port_A, which loads the register R1 into the output ports (port addresses 0 through 3), requires control bit *M*2 to be set to 1.

The Boolean equations for the control bits for the modified OUT instruction are

$$M2 = IR(4), LOAD\_OP(0) = \overline{IR(1)} \cdot \overline{IR(0)},$$
$$LOAD\_OP(1) = \overline{IR(1)} \cdot IR(0), LOAD\_OP(2) = IR(1) \cdot \overline{IR(0)}, LOAD\_OP(3) = IR(1) \cdot IR(0)$$

The Boolean equations for the control bits for *LOAD_OP*(0 through 3) are written using just the bits *IR*(1) and *IR*(0)—that is, the output port address bits. This is assuming that we will use the procedure for designing the instruction decoder, as shown in Section 19.6 where the complete OPCODE is specified for the modified OUT instruction prior to listing the Boolean equations.

19.6 DESIGNING AN INSTRUCTION DECODER FOR THE MODIFIED IN AND OUT INSTRUCTIONS FOR VBC1-E

Procedure ID that we recommend using for the design of the instruction decoder is repeated as follows: (1) use a process with a case statement to select each instruction by its OPCODE, (2) specify the default instruction decoder output values before the case statement, and (3) use Boolean equations for the control bits within the case statement. (Note: Only the control bits that are different from the default instruction decoder output values need to be added to the case statement.)

For the modified IN instruction, the OPCODE is IR(7:5) = 101 and IR(3:2) = 10. An easy way to specify the location of the OPCODE bits for the modified IN instruction is to write an 8-bit sequence for the OPCODE bits with the letter V (for Void) placed in each bit position that has no OPCODE bit. The 8-bit sequence for the OPCODE bits for the modified IN instruction is 101V10VV.

For the modified OUT instruction, the OPCODE is IR(7:5) = 101 and IR(3:2) = 00. The 8-bit sequence for the OPCODE bits for the modified OUT instruction is 101V00VV, where V (for Void) is placed in each bit position that has no OPCODE bit.

As obtained earlier, the Boolean equations for the control bits for the modified IN instruction are

$$M1 = 1, LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4),$$
$$DI1 = IR(1), DI0 = IR(0)$$

As obtained earlier, the Boolean equations for the control bits for modified OUT instruction are

$$M2 = IR(4), \ LOAD\_OP(0) = \overline{IR(1)} \cdot \overline{IR(0)},$$
$$LOAD\_OP(1) = \overline{IR(1)} \cdot IR(0), LOAD\_OP(2) = IR(1) \cdot \overline{IR(0)}, LOAD\_OP(3) = IR(1) \cdot IR(0)$$

Listing 19.3 shows a partial VHDL design for an instruction decoder, for the modified IN and modified OUT instructions using a behavioral design style—that is, a process with a case statement.

```
decode_process:
process (ir)
begin
   --default Instruction Decoder output values
   m1<= '0'; m2<= '0'; load_r0<= '0'; load_r1<= '0';
   di1<= '0'; di0<= '0'; load_op<="0000";
   case ir(7 downto 5) is
      --provides the control bits for the modified IN and modified OUT instructions
      when "101" => if ir(3 downto 2) = "10" then m1 <= '1'; load_r0 <= not ir(4);
                                                  load_r1 <= ir(4);
                                                  di1 <= ir(1); di0 <= ir(0);
                    elsif ir(3 downto 2) = "00" then m2 <= ir(4);
                                                  load_op(0) <= not ir(1) and not ir(0);
                                                  load_op(1) <= not ir(1) and     ir(0);
                                                  load_op(2) <=     ir(1) and not ir(0);
                                                  load_op(3) <=     ir(1) and     ir(0);
                    end if;
      when others => null;
   end case;
end process decode_process;
```

LISTING 19.3 Partial VHDL design for an instruction decoder for the modified IN and modified OUT instructions using a behavioral design style

Things you should notice about the partial VHDL design in Listing 19.3:

- Default instruction decoder output values are assigned to all the control bits before the case statement to ensure proper circuit operation—that is, so that inferred latches will not be generated.
- The case statement evaluates the signal *IR*(7 downto 5) or the bits 7:5 in the *IR*, which is part of the OPCODE for both the modified IN and modified OUT instructions. The if statement evaluates the signal *IR*(3 downto 2) or the bits 3:2 and assigns the correct control bits values for the modified IN instruction and the modified OUT instruction.
- When the OPCODE is *IR*(7:5) = 101 and *IR*(3:2) = 10 for the modified IN instruction, the assignments are made to the control bits. The assignments to the control bits establish the data paths necessary for the modified IN instruction to be performed by the VBC1-E architecture.
- When the OPCODE is *IR*(7:5) = 101 and *IR*(3:2) = 00 for the modified OUT instruction, the assignments are made to the control bits. The assignments to the control bits establish the data paths necessary for the modified OUT instruction to be performed by the VBC1-E architecture.

19.7 DESIGNING AN INSTRUCTION DECODER FOR THE LOADI, ADDI, AND JNZ INSTRUCTIONS FOR VBC1-E

The instructions LOADI, ADDI, and JNZ are the same for VBC1 and VBC1-E.

For the LOADI instruction, the OPCODE is IR(7:5) = 001. The 8-bit sequence for the OPCODE bits for the LOADI instruction is 101VVVVV, where V (for Void) is placed in each bit position that has no OPCODE bit.

For the ADDI instruction, the OPCODE is IR(7:5) = 011. The 8-bit sequence for the OPCODE bits for the ADDI instruction is 011VVVVV, where V (for Void) is placed in each bit position that has no OPCODE bit.

For the JNZ instruction, the OPCODE is IR(7:5) = 111. The 8-bit sequence for the OPCODE bits for the JNZ instruction is 111VVVVV, where V (for Void) is placed in each bit position that has no OPCODE bit.

As obtained earlier in Chapter 15, Section 15.6, the Boolean equations for the control bits for the LOADI instruction are

$$M1 = 0, M2 = 0, M3 = 1, M4 = 0, M5 = 1$$
$$LOAD\_R0 = \overline{IR(4)},\ LOAD\_R1 = IR(4),\ LOAD\_OP = 0$$

As obtained earlier in Chapter 15, Section 15.7, the Boolean equations for the control bits for the ADDI instruction are

$$M1 = 0, M2 = IR(4), M3 = 1, M4 = 0, M5 = 1$$
$$LOAD\_R0 = \overline{IR(4)},\ LOAD\_R1 = IR(4),\ LOAD\_OP = 0$$

As obtained earlier in Chapter 15, Section 15.11, the Boolean equations for the control bits for the JNZ instruction are

$$M1 = 0, M2 = 0, M3 = 0, M4 = 0, M5 = 0$$
$$M6 = \overline{IR(4)}\cdot(R0(0) + R0(1) + R0(2) + R0(3)) + IR(4)\cdot(R1(0) + R1(1) + R1(2) + R1(3))$$
$$LOAD\_R0 = 0, LOAD\_R1 = 0, LOAD\_OP = 0$$

Listing 19.4 shows just the case statement for Procedure ID for the instructions LOADI, ADDI, and JNZ for VBC1-E—that is, the default instruction decoder output values are not shown in Listing 19.4.

LISTING 19.4 Case statement for Procedure ID for the instructions LOADI, ADDI, and JNZ for VBC1-E

```
case ir (7 downto 5) is
    --provides the control bits for the LOADI instruction
    when "001" => m3 <= '1'; m5 <= '1'; load_r0 <= not ir(4);
                  load_r1 <= ir(4);

    --provides the control bits for the ADDI instruction
    when "011" => m2 <= ir(4); m3 <= '1'; m5 <= '1'; load_r0 <=
                  not ir(4); load_r1 <= ir(4);

    --provides control bit m6 for the JNZ instruction
    when "111" => m6 <= (not ir(4) and (r0(0) or r0(1) or r0(2)
                  or r0(3))) or
                  (ir(4) and (r1(0) or r1(1) or r1(2) or r1(3)));
    when others => null;
end case;
```

PROBLEMS

Section 19.2 Designing the Input Circuit for VBC1-E

19.1 How many different INPUT PORTS does VBC1-E have?

19.2 Name the logic circuit that is used to perform the input for VBC1-E.

19.3 What do *IP* and *DI* stand for in the input circuitry for VBC1-E?

19.4 Write a concise transfer function for a 4-to-1 MUX bus steering circuit.

19.5 Write a concise transfer function for an 8-to-1 MUX bus steering circuit with inputs *IP*0 through *IP*7, *SEL* inputs (000 through 111), and output *DI*.

19.6 Show a partial VHDL design called MUX_BUS4 for the 4-to-1 MUX bus steering circuit in Figure 19.1, in Section 19.2. Use just one conditional signal assignment.

19.7 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for MUX_BUS4 in problem 19.6.

19.8 Combine your code for problems 19.6 and 19.7 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

19.9 In the concatenation operation of *A&B&C*, which signal is the MSB and which signal is the LSB in the final result?

19.10 What must be changed in the code for the complete design called MUX_BUS4 in problem 19.6 to expand the designs to make the input circuits for the design contain 8 bits, 16 bits, and 32 bits, respectively?

Section 19.3 Instruction Decoder Truth Table for the Modified IN Instruction for VBC1-E

19.11 Where is the modified input circuit added to the data path unit of VBC1-E?

19.12 What are the control bits for the modified IN instruction?

19.13 Which input signals provide the steering for the input ports (port addresses 0 through 3) in Figure 19.2, in Section 19.3?

19.14 Write the transfer function form for the modified IN instruction IN DR,Port_A for VBC1-E.

19.15 In Figure 19.2, what are the values of each of the control bits *DI*1, *DI*0, *M*1, *LOAD_R*0, and *LOAD_R*1 when the modified IN instructions IN R0,0 and IN R1,2 are executed?

19.16 In Figure 19.2, what are the values of each of the control bits *DI*1, *DI*0, *M*1, *LOAD_R*0, and *LOAD_R*1 when the modified IN instructions IN R1,1 and IN R0,3 are executed?

Section 19.4 Designing the Output Circuit for VBC1-E

19.17 Name the logic circuit that is used to perform the output for VBC1-E.

19.18 How many bits are used for each OUTPUT PORT for VBC1-E?

19.19 What is the name of the clock signal that drives each OUTPUT PORT for VBC1-E?

19.20 What is the name of the signal that drives the D inputs of each OUTPUT PORT for VBC1-E?

19.21 Write a concise transfer function for each loadable D flip-flop of each OUTPUT PORT for VBC1-E.

19.22 Show a partial VHDL design called OUTPUT_PORT4 for the array of four loadable D flip-flops in Figure 19.3, in Section 19.4. Use four separate conditional signal assignments—that is, one for each of the output ports.

19.23 Write the required library clause, use clause (for the package IEEE.STD_LOGIC_1164), and entity declaration for OUTPUT_PORT4 in problem 19.22.

19.24 Combine your code for problems 19.22 and 19.23 to form a complete VHDL design. Obtain a simulation waveform diagram that shows correct functionality for the complete VHDL design.

Section 19.5 Instruction Decoder Truth Table for the Modified OUT Instruction VBC1-E

19.25 Where is the modified output circuit added to the data path unit of VBC1-E?

19.26 What are the control bits for the modified OUT instruction?

19.27 Can the new OUTPUT PORTS for VBC1-E be cleared? Explain why or why not.

19.28 Write the transfer function form for the modified OUT instruction OUT DR,Port_A for VBC1-E.

19.29 In Figure 19.4, what are the values of each of the control bits *M*2, *LOAD_OP*(0), *LOAD_OP*(1), *LOAD_OP*(2), and *LOAD_OP*(3) when the modified OUT instructions OUT R0,0 and OUT R1,2 are executed?

19.30 In Figure 19.4, what are the values of each of the control bits *M*2, *LOAD_OP*(0), *LOAD_OP*(1), *LOAD_OP*(2), and *LOAD_OP*(3) when the modified OUT instructions OUT R1,1 and OUT R0,3 are executed?

Section 19.6 Designing an Instruction Decoder for the Modified IN and OUT Instructions for VBC1-E

19.31 Write the OPCODE for the modified IN instruction.

19.32 Write an 8-bit sequence for the OPCODE bits for the modified IN instruction with the letter V placed in each bit position that has no OPCODE bit.

19.33 Show a partial VHDL design for an instruction decoder for VBC1-E just for the modified IN instruction using Procedure ID.

19.34 Write the OPCODE for the modified OUT instruction.

19.35 Write an 8-bit sequence for the OPCODE bits for the modified OUT instruction with the letter V placed in each bit position that has no OPCODE bit.

19.36 Show a partial VHDL design for an instruction decoder for VBC1-E just for the modified OUT instruction using Procedure ID.

Section 19.7 Designing an Instruction Decoder for the LOADI, ADDI, and JNZ Instructions for VBC1-E

19.37 Write the OPCODE for the LOADI instruction for VBC1-E.

19.38 Write an 8-bit sequence for the OPCODE bits for the LOADI instruction with the letter V placed in each bit position that has no OPCODE bit.

19.39 Write the Boolean equations for the control bits *M*3, *M*1, *LOAD_R*0, *LOAD_R*1, and *M*5 for the LOADI instruction for VBC1-E after you obtain the proper truth table via Figure 16.2 in Chapter 16 (Section 16.2).

19.40 Write the OPCODE for the ADDI instruction for VBC1-E.

19.41 Write an 8-bit sequence for the OPCODE bits for the ADDI instruction with the letter V placed in each bit position that has no OPCODE bit.

19.42 Write the Boolean equations for the control bits $M3$, $M1$, $LOAD\_R0$, $LOAD\_R1$, $M2$, and $M5$ for the ADDI instruction for VBC1-E after you obtain the proper truth table via Figure 16.2 in Chapter 16 (Section 16.2).

19.43 Write the OPCODE for the JNZ instruction for VBC1-E.

19.44 Write an 8-bit sequence for the OPCODE bits for the JNZ instruction with the letter V placed in each bit position that has no OPCODE bit.

19.45 Use the following Boolean equations for the control bits, and write just the case statement for Procedure ID for the LOADI instruction for VBC1-E. Discuss how the Boolean equation for the control bit $M0$ should be handled.

$$M3 = 1,\ M1 = 0,\ LOAD\_R0 = \overline{IR(4)},$$
$$LOAD\_R1 = IR(4),\ M5 = 1$$

19.46 Use the following Boolean equations for the control bits, and write just the case statement for Procedure ID for the ADDI instruction for VBC1-E. Discuss how the Boolean equation for the control bit $M0$ should be handled.

$$M3 = 1,\ M1 = 0,\ LOAD\_R0 = \overline{IR(4)},$$
$$LOAD\_R1 = IR(4),\ M2 = IR(4),\ M5 = 1$$

19.47 Use the following Boolean equation for the control bit $M6$, and write just the case statement for Procedure ID for the JNZ instruction for VBC1-E.

$$M6 = \overline{IR(4)}\cdot(R0(0) + R0(1) + R0(2) + R0(3)) + IR(4)\cdot(R1(0) + R1(1) + R1(2) + R1(3))$$

Chapter 20

Designing the Data Memory Circuit for VBC1-E

Chapter Outline

20.1 INTRODUCTION

This is the second in the series of chapters that teaches you how to expand the design of VBC1 to make VBC1-E. In this chapter, you will learn how to design a data memory circuit for VBC1-E.

20.2 DESIGNING THE DATA MEMORY FOR VBC1-E

Data memory allows a programmer to store data and then fetch the data as needed in a program. VBC1 does not have a data memory; however, VBC1-E has a small data memory with four storage locations.

Figure 20.1 shows a logic symbol with the necessary inputs and output for the data memory for VBC1-E.

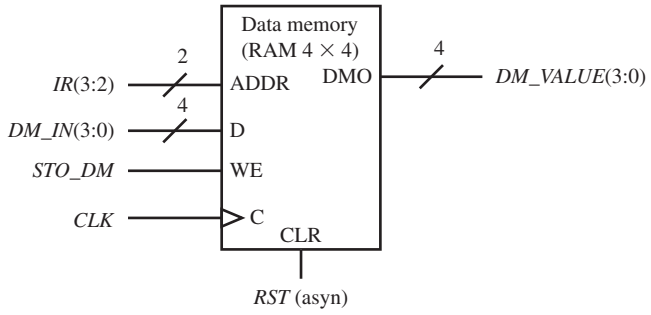


FIGURE 20.1 Logic symbol for the data memory for VBC1-E with an asynchronous reset input

Things you should notice about the data memory in Figure 20.1:

- Input ADDR (address) selects the location of where data is placed in the data memory.
- Input D (data) provides the data that is placed in the data memory.
- Input WE (write enable) enables data (the data to be stored) to be written into the memory at the next rising edge of the clock signal *CLK*.
- Input C (control) provides the clock for writing memory.
- Input CLR provides an asynchronous reset capability to clear the data memory.
- Output DMO provides the outputs for the memory.
- ADDR, D, WE, C, CLR, and DMO are labels.
- The signals for the memory are inputs *IR* (instruction register), *DM_IN* (data), *STO_DM* (store data memory), *CLK* (clock), *RST* (asyn), and output *DM_VALUE* (data memory value).
- The data memory map contains four memory locations and each location has a storage capacity of 4 bits (or 1/2 byte) as shown in Figure 20.2.
- The data memory is referred to as a RAM (random-access memory) 4 by 4 (or 4 × 4). Because $4 = 2^2$, a minimum of two address lines are required to determine the address for each location in data memory.
- The data memory is stored (written) synchronously at each rising edge of the signal *CLK* when signal *STO_DM* is 1 and retrieved (read) from the signal *DM_VALUE*.
- When *RST* (asyn) = 1, the entire data memory is cleared asynchronously to "0000" at every address—that is, 0 through 3.

Figure 20.2 shows the data memory map for VBC1-E.

FIGURE 20.2 Data memory map for VBC1-E

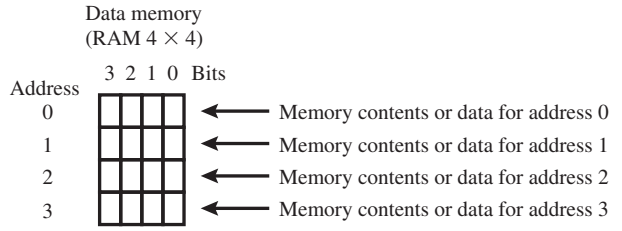


Figure 20.3 shows a circuit for the 4 × 4 data memory represented by the logic symbol in Figure 20.1.

Things you should notice about the gate level circuit for the 4 × 4 data memory:

- The decoder selects a D flip-flop in the loadable D flip-flop array that the data *DM_IN* is written into. The location that the data is written or stored in the data memory is specified by the *IR* bits 3 down to 2. The AND array only allows data to be written or stored when the control signal *STO_DM* = 1.
- The loadable D flip-flops in the loadable D flip-flop array can be asynchronously cleared by the signal *RST*, at any time independent of the clock.
- The MUX provides the output of the data memory via the signal *DM_VALUE*. The *IR* bits 3 down to 2 specify the location of the data that is read or fetched from the data memory.

Listing 20.1 shows a complete VHDL design for the data memory for VBC1-E in Figure 20.3. Things you should notice about the VHDL design in Listing 20.1:

- The list of internal signals in the design is shown between **architecture** and the first **begin**. The list includes the following signals: *DEC_OUT*, *CE*, *Q*0, *Q*1, *Q*2, and *Q*3.
- The design for the decoder is shown with four Boolean equations in terms of the inputs *IR*(3) and *IR*(2).
- The design for the And array is shown with four Boolean equations in terms of the inputs *STO_DM*, *DEC_OUT*(0), *DEC_OUT*(1), *DEC_OUT*(2), and *DEC_OUT*(3),

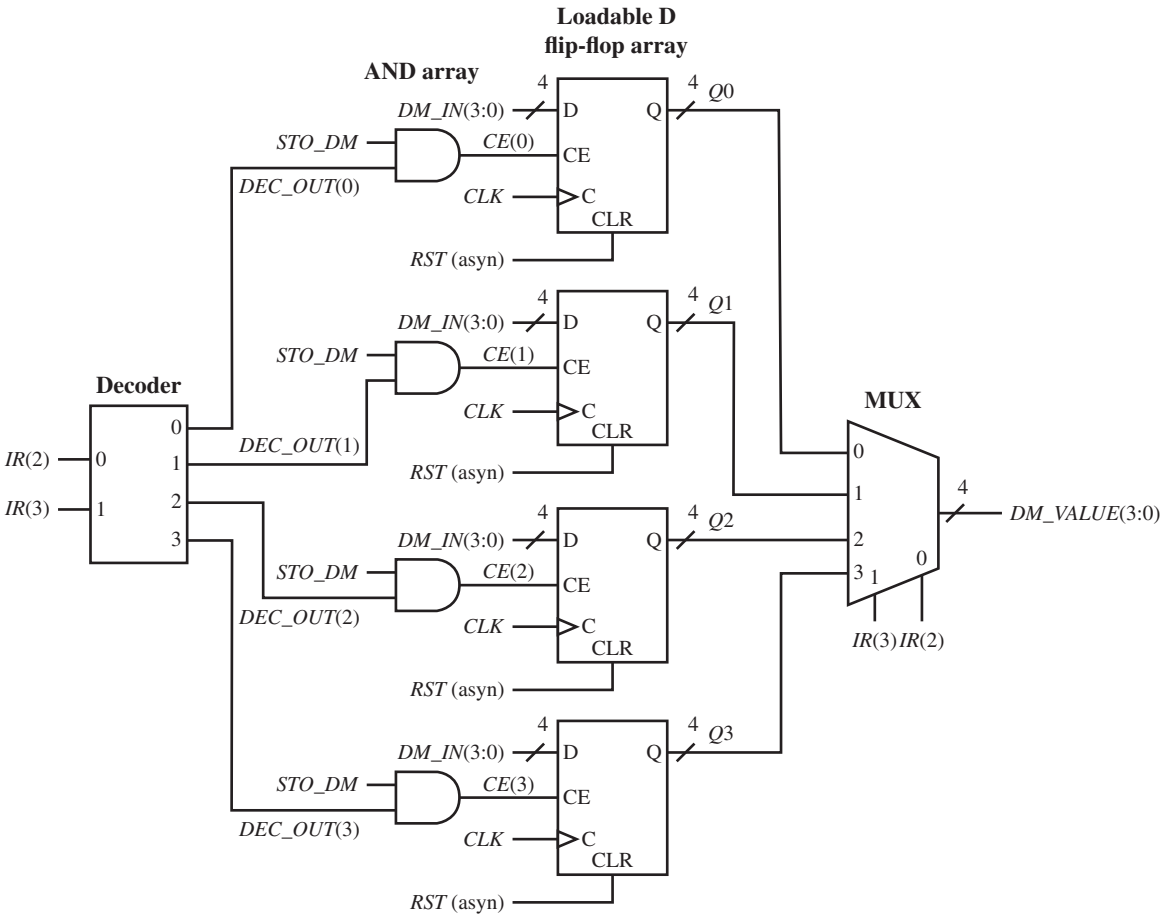


FIGURE 20.3 Circuit for the 4 × 4 data memory represented by the logic symbol in Figure 20.1 for VBC1-E

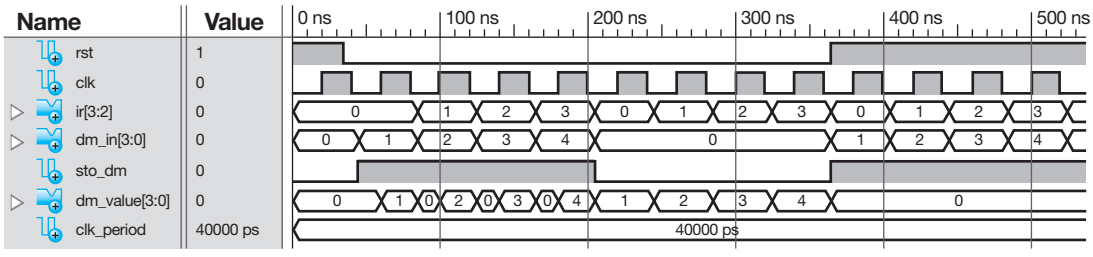
LISTING 20.1
Complete VHDL design for the data memory for VBC1-E (project: Data_Memory)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Data_Memory is port (
     rst, clk : in STD_LOGIC;
     ir : in STD_LOGIC_VECTOR (3 downto 2);
     dm_in : in STD_LOGIC_VECTOR (3 downto 0);
     sto_dm : in STD_LOGIC;
     dm_value : out STD_LOGIC_VECTOR (3 downto 0)
     );
end Data_Memory;

architecture Mixed of Data_Memory is
     signal dec_out, ce: std_logic_vector (3 downto 0);
     signal q0: std_logic_vector (3 downto 0);
     signal q1: std_logic_vector (3 downto 0);
     signal q2: std_logic_vector (3 downto 0);
     signal q3: std_logic_vector (3 downto 0);
begin

--Decoder
     dec_out(0) <= not ir(3) and not ir(2);
     dec_out(1) <= not ir(3) and     ir(2);
                                                    (Continued)
```

```
    dec_out(2) <=     ir(3) and not ir(2);
    dec_out(3) <=     ir(3) and     ir(2);

--And Array
    ce(0) <= sto_dm and dec_out(0);
    ce(1) <= sto_dm and dec_out(1);
    ce(2) <= sto_dm and dec_out(2);
    ce(3) <= sto_dm and dec_out(3);

--Loadable D Flip-Flops
process (rst, clk)
begin
    if rst = '1' then q0 <= "0000";
    elsif ce(0) = '1' and rising_edge (clk) then q0 <= dm_in;
    end if;

    if rst = '1' then q1 <= "0000";
    elsif ce(1) = '1' and rising_edge (clk) then q1 <= dm_in;
    end if;

    if rst = '1' then q2 <= "0000";
    elsif ce(2) = '1' and rising_edge (clk) then q2 <= dm_in;
    end if;

    if rst = '1' then q3 <= "0000";
    elsif ce(3) = '1' and rising_edge (clk) then q3 <= dm_in;
    end if;
end process;

--MUX
    dm_value <= q0 when ir(3 downto 2) = "00" else
                q1 when ir(3 downto 2) = "01" else
                q2 when ir(3 downto 2) = "10" else
                q3;
end Mixed;
```

- The design for the loadable D flip-flops is shown with a process using an if statement for each of the loadable D flip-flops in terms of the inputs *RST*, *CE*(0), *CE*(1), *CE*(2), *CE*(3), *CLK*, and *DM_IN*. The data memory is asynchronously cleared via the signal *RST*. Remember that this is accomplished by placing the *RST* condition prior to the rising_edge (CLK) condition in the if statements.
- The design for the MUX is shown with a conditional signal assignment in terms of the inputs *Q*0, *Q*1, *Q*2, *Q*3, *IR*(3), and *IR*(2).

Waveform 20.1 shows the correct functionality of design entity Data_Memory.



WAVEFORM 20.1 Simulation for the correct functionality of design entity Data Memory.

Things you should notice about the waveforms in Waveform 20.1:

- First, the data memory is asynchronously cleared by the signal *RST*.
- The memory address is cycled from 0 through 3 via the input signal *IR*(3:2).
- Random values of data (1, 2, 3, and 4) are written into the data memory during the first four clock ticks after *RST* goes to 0 when *STO_DM* is set to 1, as shown by the output signal *DM_VALUE*.
- In the waveform diagrams, during the next four clock ticks when *STO_DM* is set to 0, the data memory is read. By closely observing the waveform diagrams, it can be seen that the data written into each memory address (0, 1, 2, and 3) is read correctly, confirming that the data memory is working properly.
- When *RST* = 1 at the end of the simulation, the data memory is asynchronously cleared.

20.3 DESIGNING CIRCUITS TO SELECT THE REGISTERS AND DATA FOR VBC1-E

Figure 20.4 shows the circuit select register that is used to perform the STORE instruction—that is, STORE DR, Addr. Observe that the content of register *R*0 or *R*1 is supplied to the data memory via the MUX named select register, and the value of the data memory is monitored by the circuit data memory monitor via the signal *DM*(3:0). Figure 20.4 also shows the circuit select data that is used to perform the FETCH instruction—that is, FETCH DR, Addr. For the FETCH instruction, the value in the data memory is placed in the destination register when the signal *FET_DM* = 1.

Things you should notice about the circuits for selecting registers and data:

- The MUX for the circuit select register is used to direct the data from *R*0 or *R*1 into the data memory where the data is stored when the STORE instruction is executed.
- The circuit data memory monitor is used to monitor the contents of data memory via four LEDs at the address specified by the *IR* bits 3 down to 2 when a STORE or FETCH instruction is executed.
- The MUX for the circuit select data is used to direct the data supplied from *R_ALU_DI* or *DM_VALUE* into register *R*0 or *R*1.

20.4 INSTRUCTION DECODER TRUTH TABLES FOR THE STORE AND FETCH INSTRUCTIONS FOR VBC1-E

For the STORE instructions to work properly, we must specify the instruction decoder truth tables to provide the control signals so that the data path unit will operate correctly. Figure 20.4 shows the control signals required for the design of VBC1-E for the STORE instruction—that is, *STO_DR* (store destination register), and *STO_DM* (store data memory).

As a review, the ALF, TFF, and MCF for the STORE instruction for VBC1-E are shown in Table 20.1.

TABLE 20.1
The ALF, TFF, and MCF for the STORE instruction for VBC1-E

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)							
		7:5 OPCODE			Dest.Reg			1:0 OPCODE Extension	
		7	6	5	4	3	2	1	0
STORE DR,Addr	DM[IR(3:2)] ← DR	0	0	0	0/1	A	A	0	1
					0 = R0 1 = R1	3:2 Addr of Data Memory (DM)			

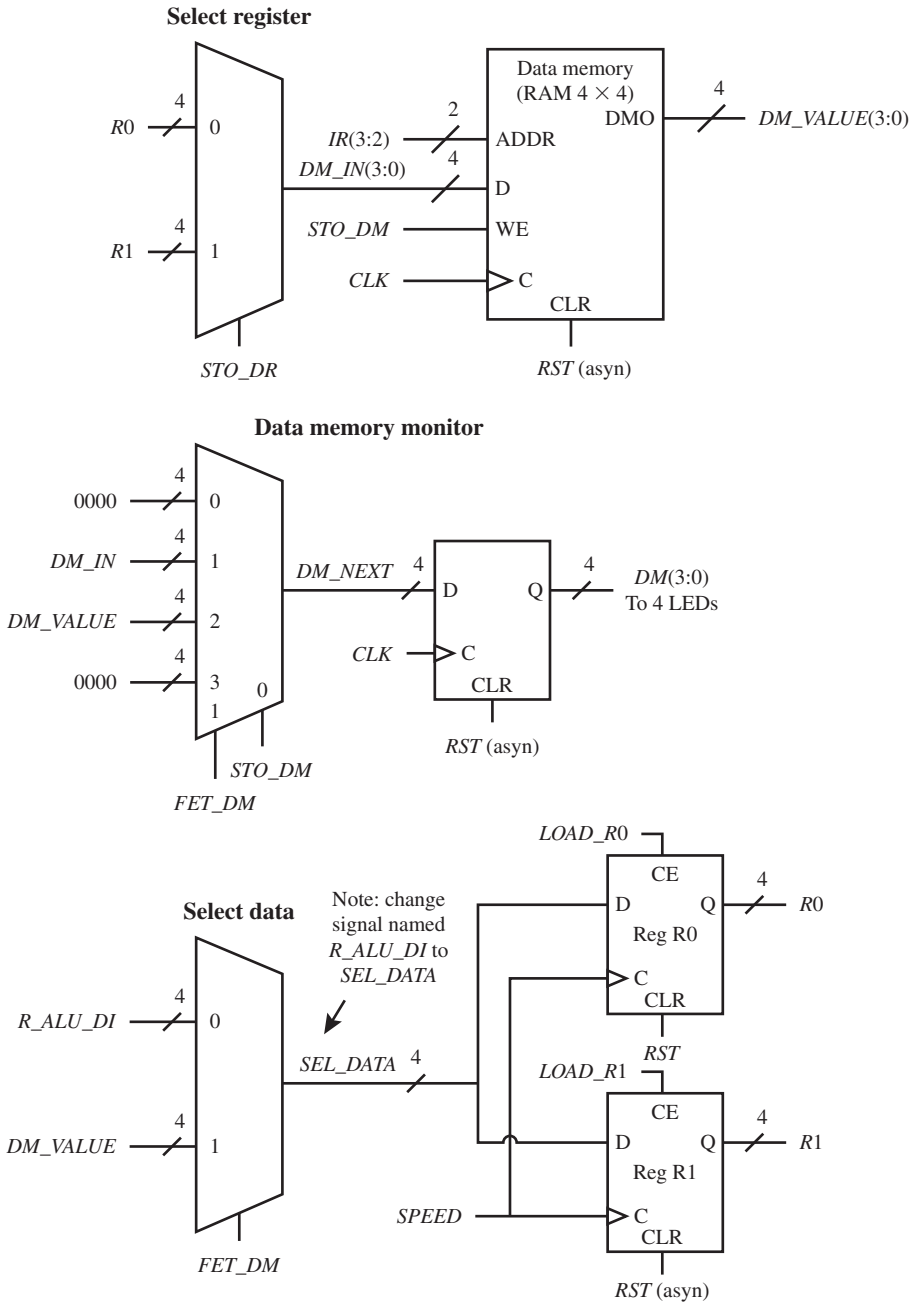


FIGURE 20.4 Circuits for selecting registers and data for VBC1-E

Table 20.2 shows the instruction decoder truth table for the STORE instruction for VBC1-E.

Things you should notice about the instruction decoder truth table for the STORE instruction in Table 20.2:

- The values for the *IR* bits for the instruction decoder are the machine code bits for the STORE instruction. The values for the control bits for the instruction decoder truth table

TABLE 20.2 Instruction decoder truth table for the STORE instruction for VBC1-E

	IR								Control bits	
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***STO_DR***	***STO_DM***
STORE R0,0	0	0	0	0	0	0	0	1	0	1
STORE R0,1	0	0	0	0	0	1	0	1	0	1
STORE R0,2	0	0	0	0	1	0	0	1	0	1
STORE R0,3	0	0	0	0	1	1	0	1	0	1
STORE R1,0	0	0	0	1	0	0	0	1	1	1
STORE R1,1	0	0	0	1	0	1	0	1	1	1
STORE R1,2	0	0	0	1	1	0	0	1	1	1
STORE R1,3	0	0	0	1	1	1	0	1	1	1

are determined by ensuring that the control bits perform the transfer function form for the STORE instruction STORE DR,Addr, which is DM[IR(3:2)] ← DR.

- The instruction STORE R0,Addr, which stores the value in the destination register at location Addr in data memory, requires control bit *STO_DR* to be set to 0 and control bit *STO_DM* to be set to 1.
- The instruction STORE R1,Addr, which stores the value in the destination register at location Addr in data memory, requires control bit *STO_DR* to be set to 1 and control bit *STO_DM* to be set to 1.
- The control signal *STO_DM* is used in the circuit data memory monitor to display the value of the data memory each time a STORE instruction is executed when ST*O_DM* = 1.

The Boolean equations for the control bits for the STORE instruction are

$STO\_DR = IR(4)$, $STO\_DM = 1$

For the FETCH instructions to work properly, we must also specify the instruction decoder truth tables to provide the control signals so that the data path unit will operate correctly. Figure 20.4 shows the control signals required for the design of VBC1-E for a FETCH instruction—that is, *FET_DM* (fetch data memory), *LOAD_R*0, and *LOAD_R*1.

As a review, the ALF, TFF, and MCF for the FETCH instruction for VBC1-E are shown in Table 20.3.

TABLE 20.3 The ALF, TFF, and MCF for the FETCH instruction for VBC1-E

Assembly language form (ALF)	**Transfer function form (TFF)**	**Machine code form (MCF)**

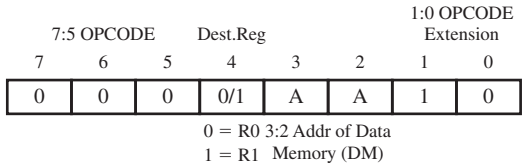


Table 20.4 shows the instruction decoder truth table for the FETCH instruction for VBC1-E.

TABLE 20.4 Instruction decoder truth table for the FETCH instruction for VBC1-E

	IR								Control bits		
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***FET_DM***	***LOAD_R0***	***LOAD_R1***
FETCH R0,0	0	0	0	0	0	0	1	0	1	1	0
FETCH R0,1	0	0	0	0	0	1	1	0	1	1	0
FETCH R0,2	0	0	0	0	1	0	1	0	1	1	0
FETCH R0,3	0	0	0	0	1	1	1	0	1	1	0
FETCH R1,0	0	0	0	1	0	0	1	0	1	0	1
FETCH R1,1	0	0	0	1	0	1	1	0	1	0	1
FETCH R1,2	0	0	0	1	1	0	1	0	1	0	1
FETCH R1,3	0	0	0	1	1	1	1	0	1	0	1

Things you should notice about the instruction decoder truth table for the FETCH instruction in Table 20.4:

- The values for the *IR* bits for the instruction decoder are the machine code bits for the FETCH instruction. The values for the control bits for the instruction decoder truth table are determined by ensuring that the control bits perform the transfer function form for the FETCH instruction FETCH DR,Addr, which is DR ← DM[IR(3:2)].
- The instruction FETCH R0,Addr, which stores the value in the data memory at location Addr in register R0, requires control bit *FET_DM* to be set to 1, control bit *LOAD_R*0 to be set to 1, and control bit *LOAD_R*1 to be set to 0.
- The instruction FETCH R1,Addr, which stores the value in the data memory at location Addr in register R1, requires control bit *FET_DM* to be set to 1, control bit *LOAD_R*0 to be set to 0, and control bit *LOAD_R*1 to be set to 1.
- The control signal *FET_DM* is used in the circuit data memory monitor to display the value of the data memory each time a FETCH instruction is executed when *FET_DM* = 1.

The Boolean equations for the control bits for the FETCH instruction are

$$FET\_DM = 1,\ LOAD\_R0 = \overline{IR(4)},$$
$$LOAD\_R1 = IR(4)$$

20.5 DESIGNING AN INSTRUCTION DECODER FOR THE STORE AND FETCH INSTRUCTIONS FOR VBC1-E

Procedure ID will now be used for the design of the instruction decoder for the STORE and FETCH instructions.

The OPCODE for the STORE instruction is *IR*(7:5) = 000 and *IR*(1:0) = 01. An easy way to specify the location of the OPCODE bits for the STORE instruction is to write an 8-bit sequence for the OPCODE bits with the letter V for Void placed in each bit position that has no OPCODE bit. The 8-bit sequence for the OPCODE bits for the STORE instruction is 000VVV01.

The OPCODE for the FETCH instruction is *IR*(7:5) = 000 and *IR*(1:0) = 10. The 8-bit sequence for the OPCODE bits for the FETCH instruction is 000VVV10, where V for Void is placed in each bit position that has no OPCODE bit.

The Boolean equations for the control bits for the STORE instruction, as obtained earlier, are

$STO\_DR = IR(4), STO\_DM = 1$

The Boolean equations for the control bits for the FETCH instruction, as obtained earlier, are

$FET\_DM = 1, LOAD\_R0 = \overline{IR(4)}, \ LOAD\_R1 = IR(4)$

Listing 20.2 shows a partial VHDL design for an instruction decoder for the STORE and FETCH instructions using a behavioral design style—that is, a process with a case statement.

```
--Instruction Decoder
process (ir)
begin
   --default Instruction Decoder output values
   load_r0 <= '0'; load_r1 <= '0';
   sto_dr <= '0'; sto_dm <= '0'; fet_dm <= '0';
   case ir(7 downto 5) is
      when "000" => if ir(1 downto 0) = "01" then sto_dr <= ir(4); sto_dm <= '1';
                    elsif ir(1 downto 0) = "10" then fet_dm <= '1';
                                                     load_r0 <= not ir(4);
                                                     load_r1 <= ir(4);
                    end if;
      when others => null;
   end case;
end process;
```

LISTING 20.2 Partial VHDL design for an instruction decoder for the STORE and FETCH instructions using a behavioral design style

Things you should notice about the partial VHDL design in Listing 20.2:

- Default instruction decoder output values are assigned to all the control bits before the case statement to ensure proper circuit operation—that is, so that inferred latches will not be generated.
- The case statement evaluates the signal *IR*(7 downto 5) or the bits 7:5 in the *IR*, which is part of the OPCODE for both the STORE and FETCH instructions. The if statement evaluates the signal *IR*(1 downto 0) or the bits 1:0 and assigns the correct control bits values for the STORE instruction and the FETCH instruction.
- When the OPCODE is *IR*(7:5) = 000 and *IR*(1:0) = 01 for the STORE instruction, the assignments are made to the control bits. The assignments to the control bits establish the data paths necessary for the STORE instruction to be performed by the VBC1-E architecture.
- When the OPCODE is *IR*(7:5) = 000 and *IR*(1:0) = 10 for the FETCH instruction, the assignments are made to the control bits. The assignments to the control bits establish the data paths necessary for the FETCH instruction to be performed by the VBC1-E architecture.

20.6 DESIGNING AN INSTRUCTION DECODER FOR THE MOV INSTRUCTION FOR VBC1-E

The instruction MOV has the same control equations for VBC1 and VBC1-E only when $FET\_DM = 0$. The OPCODE needs to be modified for the instruction MOV.

For the modified MOV instruction, the OPCODE is IR(7:5) = 000 and IR(1:0) = 00. The 8-bit sequence for the OPCODE bits for the MOV instruction is 000VVV00, where V for Void is placed in each bit position that has no OPCODE bit.

As obtained earlier, the Boolean equations for the control bits for the MOV instruction are

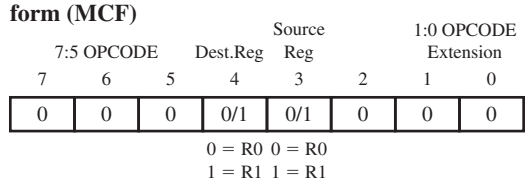
$M1 = 0, M2 = IR(3), M3 = 0$

$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), LOAD\_OP = 0, FET\_DM = 0$ (added)

As a review, the ALF, TFF, and MCF for the MOV instruction for VBC1-E are shown in Table 20.5.

TABLE 20.5 The ALF, TFF, and MCF for the MOV instruction for VBC1-E

Assembly language form (ALF)	Transfer function form (TFF)	Machine code form (MCF)



A case statement for the instruction decoder for the instruction MOV for VBC1-E uses the OPCODE bits *IR* (7 **downto** 5) followed by an if statement for the OPCODE bits *IR*(1 **downto** 0) as shown in Listing 20.3. *FET_DM* = 0 may be included in the default list.

```
case ir(7 downto 5) is
    --provides the data path control signals for the MOV instruction
    when "000" => if ir(1 downto 0) = "00" then m2 <= ir(3); load_r0 <= not ir(4);
                                                load_r1 <= ir(4); fet_dm <= '0';
    when others => null;
end case;
```

LISTING 20.3 Case statement for the instruction decoder for the instruction MOV for VBC1-E

PROBLEMS

Section 20.2 Designing the Data Memory for VBC1-E

20.1 What is the purpose of the data memory?

20.2 How many address bits are contained in the data memory for VBC1-E? How many locations do those bits have access to in the data memory?

20.3 What is data memory called or referred to as, and what is the size of the data memory for VBC1-E?

20.4 Is the data written into data memory for VBC1-E written synchronously or asynchronously?

20.5 Is the data memory for VBC1-E cleared synchronously or asynchronously, and what signal is used to clear the data memory?

20.6 What is the total number of bits that can be stored in the data memory of VBC1-E?

20.7 In Figure 20.3, what component selects a D flip-flop in the loadable D flip-flop array when *STO_DM* is asserted?

20.8 In Figure 20.3, what component provides the output for the 4 × 4 data memory for VBC1-E?

20.9 In Figure 20.3, what control signals are used to select the D flip-flops that store and fetch the data?

20.10 In Figure 20.3, what control signal is used to store the data in the D flip-flops?

20.11 In Figure 20.3, write the VHDL code for just the decoder using a conditional signal assignment.

20.12 In Figure 20.3, write the VHDL code for just the loadable D flip-flop array using four conditional signal assignments.

20.13 In Figure 20.3, write the VHDL code for just the MUX using four Boolean equations. (Hint: First draw the gate-level circuit diagram for the MUX, and write the Boolean equations from the circuit.)

Section 20.3 Designing Circuits to Select the Registers and Data for VBC1-E

20.14 In Figure 20.4, what component supplies the contents of register *R*0 or *R*1 to the data memory?

20.15 In Figure 20.4, write the VHDL code for just the MUX labeled select register using four Boolean equations. (Hint: First draw the gate level circuit diagram for the MUX, and write the Boolean equations from the circuit.)

20.16 In Figure 20.4, write the VHDL code for just the data memory monitor using conditional signal assignments.

20.17 In Figure 20.4, which control signal allows the data memory to be placed in the destination register for the FETCH instruction?

Section 20.4 Instruction Decoder Truth Tables for the STORE and FETCH Instructions for VBC1-E

20.18 In Figure 20.4, what value of the control signal *STO_DR* causes the value of register *R*0 to be stored in data memory?

20.19 In Figure 20.4, what value of the control signal *STO_DR* causes the value of register *R*1 to be stored in data memory?

20.20 In Figure 20.4, what values of the control signals *FET_DM* and *STO_DM* cause the data memory monitor circuit to display the value of the data memory each time a STORE instruction is executed?

20.21 Write the transfer function form for the STORE instruction STORE DR,Addr for VBC1-E.

20.22 In Figure 20.4, what value of the control signal *FET_DM* causes the MUX named select data to store the signal *R_ALU_DI* in register *R*0 or *R*1?

20.23 In Figure 20.4, what value of the control signal *FET_DM* causes the MUX named select data to store the signal *DM_VALUE* in register *R*0 or *R*1?

20.24 In Figure 20.4, what values of the control signals *FET_DM* and *STO_DM* cause the data memory monitor circuit to display the value of the data memory each time a FETCH instruction is executed?

20.25 Write the transfer function form for the FETCH instruction FETCH DR,Addr for VBC1-E.

Section 20.5 Designing an Instruction Decoder for the STORE and FETCH Instructions for VBC1-E

20.26 Write the OPCODE for the STORE instruction.

20.27 Write an 8-bit sequence for the OPCODE bits for the STORE instruction with the letter V placed in each bit position that has no OPCODE bit.

20.28 Show a partial design for an instruction decoder for VBC1-E just for the STORE instruction using Procedure ID.

20.29 Write the OPCODE for the FETCH instruction.

20.30 Write an 8-bit sequence for the OPCODE bits for the FETCH instruction with the letter V placed in each bit position that has no OPCODE bit.

20.31 Show a partial design for an instruction decoder for VBC1-E just for the FETCH instruction using Procedure ID.

Section 20.6 Designing an Instruction Decoder for the MOV Instruction for VBC1-E

20.32 Write the OPCODE for the MOV instruction for VBC1-E.

20.33 Write an 8-bit sequence for the OPCODE bits for the MOV instruction with the letter V placed in each bit position that has no OPCODE bit.

20.34 What is different about the control equations for VBC1 and VBC1-E for the instruction MOV?

21 Chapter

Designing the Arithmetic, Logic, Shift, Rotate, and Unconditional Jump Circuits for VBC1-E

Chapter Outline

21.1 INTRODUCTION

This is the third in the series of chapters that teaches you how to expand the design of VBC1 to make VBC1-E. In this chapter, you will learn how to design the arithmetic, logic, shift, rotate, and unconditional jump circuits for VBC1-E. This includes the circuits for the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR; the circuits for the shift and rotate instructions SR0, SR1, SL0, SL1, RR, and RL; and the circuits for the unconditional jump instructions JMP and JMPR.

21.2 DESIGNING THE ARITHMETIC AND LOGIC INSTRUCTIONS PART OF THE ALU FOR VBC1-E

Figure 21.1 shows the data path unit for the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR for VBC1-E. Observe that each instruction shown inside the ALU logic

symbol is followed by an 8-bit sequence that indicates the OPCODE bits for that instruction, with the letter V for Void placed in each bit position that has no OPCODE bit.

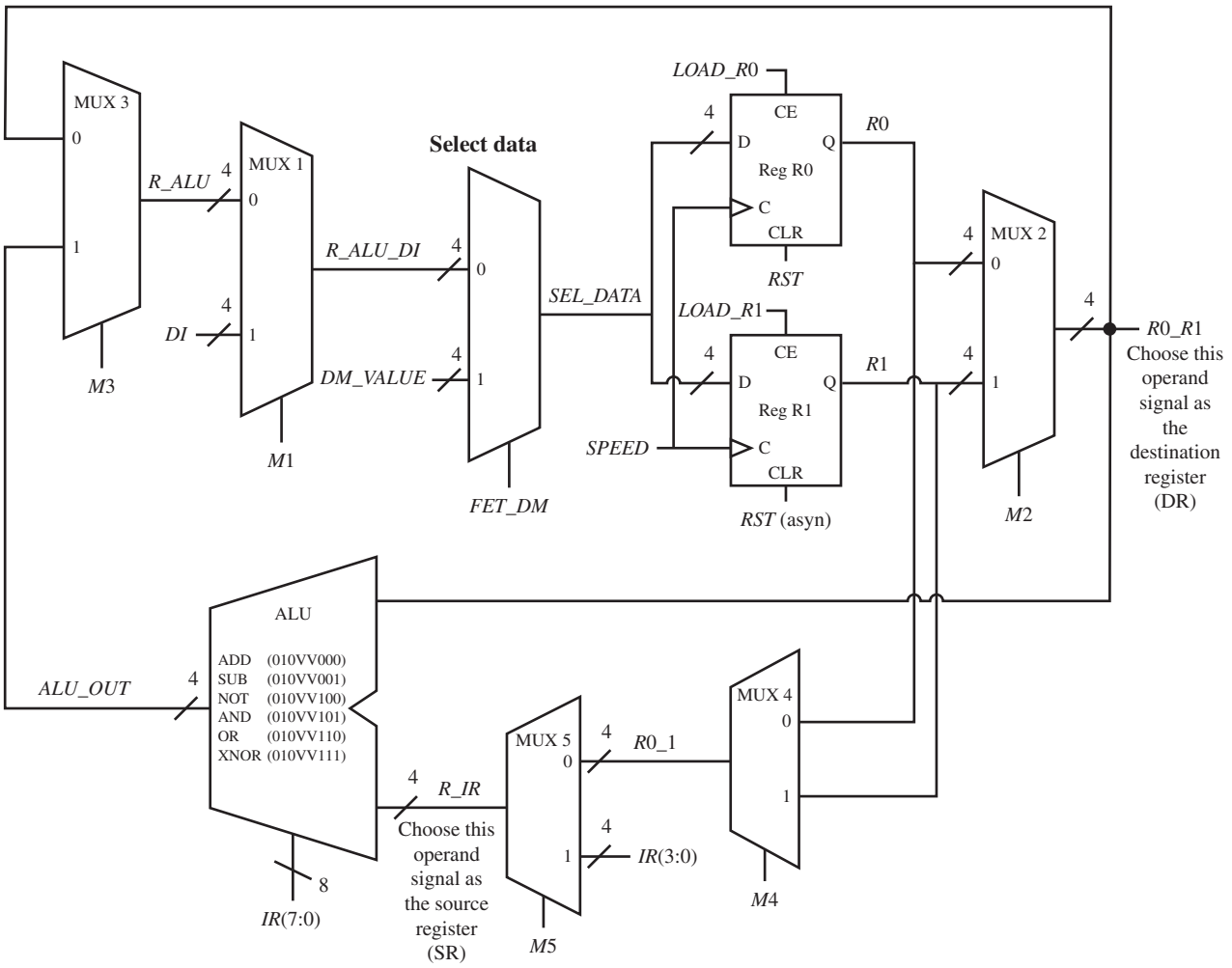


FIGURE 21.1 Data path unit for the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR for VBC1-E

Each of the arithmetic and logic instructions must be written from their transfer function forms in terms of the ALU signals *ALU_OUT*, *R*0_*R*1, and *R_IR*. For consistency, *R*0_*R*1 is used as the destination register and *R_IR* is used as the source register on the right-hand side of the transfer function form for each of the following arithmetic and logic instructions:

The transfer function form for the ADD DR,SR instruction is DR ← DR + SR, so the ALU ADD instruction must be written as *ALU_OUT* ← *R*0_*R*1 + *R_IR*.

The transfer function form for the SUB DR,SR instruction is DR ← DR − SR, so the ALU SUB instruction must be written as *ALU_OUT* ← *R*0_*R*1 − *R_IR*.

The transfer function form for the NOT DR,SR instruction is DR ← !SR, so the ALU NOT instruction must be written as *ALU_OUT* ← !*R_IR*.

The transfer function form for the AND DR,SR instruction is DR ← DR ∧ SR, so the ALU AND instruction must be written as *ALU_OUT* ← *R*0_*R*1 ∧ *R_IR*.

The transfer function form for the OR DR,SR instruction is DR ← DR ∨ SR, so the ALU OR instruction must be written as *ALU_OUT* ← *R*0_*R*1 ∨ *R_IR*.

The transfer function form for the XNOR DR,SR instruction is DR ← !(DR ⊕ SR), so the ALU XNOR instruction must be written as *ALU_OUT* ← !(*R*0_*R*1 ⊕ *R_IR*).

Listing 21.1 shows a partial VHDL design for an arithmetic logic unit for the ADD and SUB instructions for VBC1-E using a behavioral design style—that is, a **process** with a **case statement**.

```
alu_process:
process (ir, r0_r1, r_ir)
begin
   alu_out <= "0000";    --default value to prevent creating inferred latches
   case ir(7 downto 5) is
      --for the ADD and SUB instructions
      when "010" => if ir(2 downto 0) = "000"    then alu_out <= r0_r1 + r_ir;
                    elsif ir(2 downto 0) = "001" then alu_out <= r0_r 1- r_ir;
                    end if;
      when others => null;
   end case;
end process alu_process;
```

LISTING 21.1 Partial VHDL design for an arithmetic logic unit for the ADD and SUB instructions for VBC1-E using a behavioral design style

Things you should notice about the VHDL design in Listing 21.1:

- The default value of 0000 is used for the signal alu_out to ensure proper circuit operation—that is, so that inferred latches will not be generated.
- The ADD OPCODE *IR*(7:5) = 010 and *IR*(2:0) = 000 is used to select the ADD instruction in the ALU via a case statement followed by an if statement.
- The SUB OPCODE *IR*(7:5) = 010 and *IR*(2:0) = 001 is used to select the SUB instruction in the ALU via the case statement followed by an elsif statement.

21.3 DESIGNING THE INSTRUCTION DECODER FOR THE ARITHMETIC AND LOGIC INSTRUCTIONS FOR VBC1-E

In Figure 21.1, when writing the input signals for the ALU, we consistently made the operand signal *R*0_*R*1 the destination register (DR) and the operand signal *R*_*IR* the source register (SR), for each of the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR for VBC1-E. Because the operand signal *R*0_*R*1 is DR and the operand signal *R*_*IR* is SR for each of the arithmetic and logic instructions, the instruction decoder truth table for each of the arithmetic and logic instructions can be represented by the generalized truth table shown in Table 21.1. The Ext bits are the extension bits. Observe that the Boolean equations for the control bits can be calculated without using the extension bits.

TABLE 21.1 Generalized instruction decoder truth table for the arithmetic and logic instructions

Arithmetic or logic instruction (X)	*IR*								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M1***	***M2***	***M3***	***M4***	***M5***	***LOAD_R0***	***LOAD_R1***	***FET_DM***
X R0,R0	0	1	0	0	0	Ext bits			0	0	1	0	0	1	0	0
X R0,R1	0	1	0	0	1	Ext bits			0	0	1	1	0	1	0	0
X R1,R0	0	1	0	1	0	Ext bits			0	1	1	0	0	0	1	0
X R1,R1	0	1	0	1	1	Ext bits			0	1	1	1	0	0	1	0

To obtain the Boolean equations for the control bits for each of the arithmetic and logic instructions, substitute ADD, SUB, NOT, AND, OR, and XNOR for X in turn in Table 21.1. Observe that the Boolean equations for the control bits for each instruction can be written identically (in the exact same way), because each instruction has the same truth table.

The Boolean equations for the control bits for each of the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR can be written as:

$M1 = 0, M2 = IR(4), M3 = 1, M4 = IR(3), M5 = 0$

$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), FET\_DM = 0.$

Keep in mind that bit *IR*(4) specifies the destination register and bit *IR*(3) specifies the source register in the machine code form of an instruction for each of the arithmetic and logic instructions.

Procedure ID will now be used for the design of the instruction decoder for the ADD and SUB instructions.

Listing 21.2 shows a partial VHDL design for an instruction decoder for the ADD and SUB instructions for VBC1-E using a behavioral design style—that is, a process with a case statement.

```
process (ir)
begin
   --default Instruction Decoder output values
   m1 <= '0'; m2 <= '0'; m3 <= '0'; m4 <= '0'; m5 <= '0';
   load_r0 <= '0'; load_r1 <= '0'; fet_dm <= '0';
   case ir (7 downto 5) is
      --for the ADD and SUB instructions
      when "010" => m2 <= ir(4); m3 <= '1'; m4< = ir(3);
                    load_r0 <= not ir(4); load_r1 <= ir(4);
      when others => null;
   end case;
end process;
```

LISTING 21.2 Partial VHDL design for an instruction decoder for the ADD and SUB instructions for VBC1-E using a behavioral design style

21.4 DESIGNING THE SHIFT AND ROTATE INSTRUCTIONS PART OF THE ALU FOR VBC1-E

Figure 21.2 shows the data path unit for the shift and rotate instructions SR0, SR1, SL0, SL1, RR, and RL for VBC1-E. Observe that each instruction shown inside the ALU logic symbol is followed by an 8-bit sequence that indicates the OPCODE bits for that instruction, with the letter V for Void placed in each bit position that has no OPCODE bit.

Each of the shift and rotate instructions must be written from their transfer function forms in terms of the ALU signals *ALU_OUT*, *R*0_*R*1, and *R_IR*. For consistency, *R*0_*R*1 is used as the destination register and *R_IR* is used as the source register on the right-hand side of the transfer function form for each of the following shift and rotate instructions.

The transfer function form for the SR0 DR,SR instruction is DR ← 0 SR(3:1), so the ALU SR0 instruction must be written as *ALU_OUT* ← 0 *R_IR*(3:1).

The transfer function form for the SR1 DR,SR instruction is DR ← 1 SR(3:1), so the ALU SR1 instruction must be written as *ALU_OUT* ← 1 *R_IR*(3:1).

The transfer function form for the SL0 DR,SR instruction is DR ← SR(2:0) 0, so the ALU SL0 instruction must be written as *ALU_OUT* ← *R_IR*(2:0) 0.

The transfer function form for the SL1 DR,SR instruction is DR ← SR(2:0) 1, so the ALU SL1 instruction must be written as *ALU_OUT* ← *R_IR*(2:0) 1.

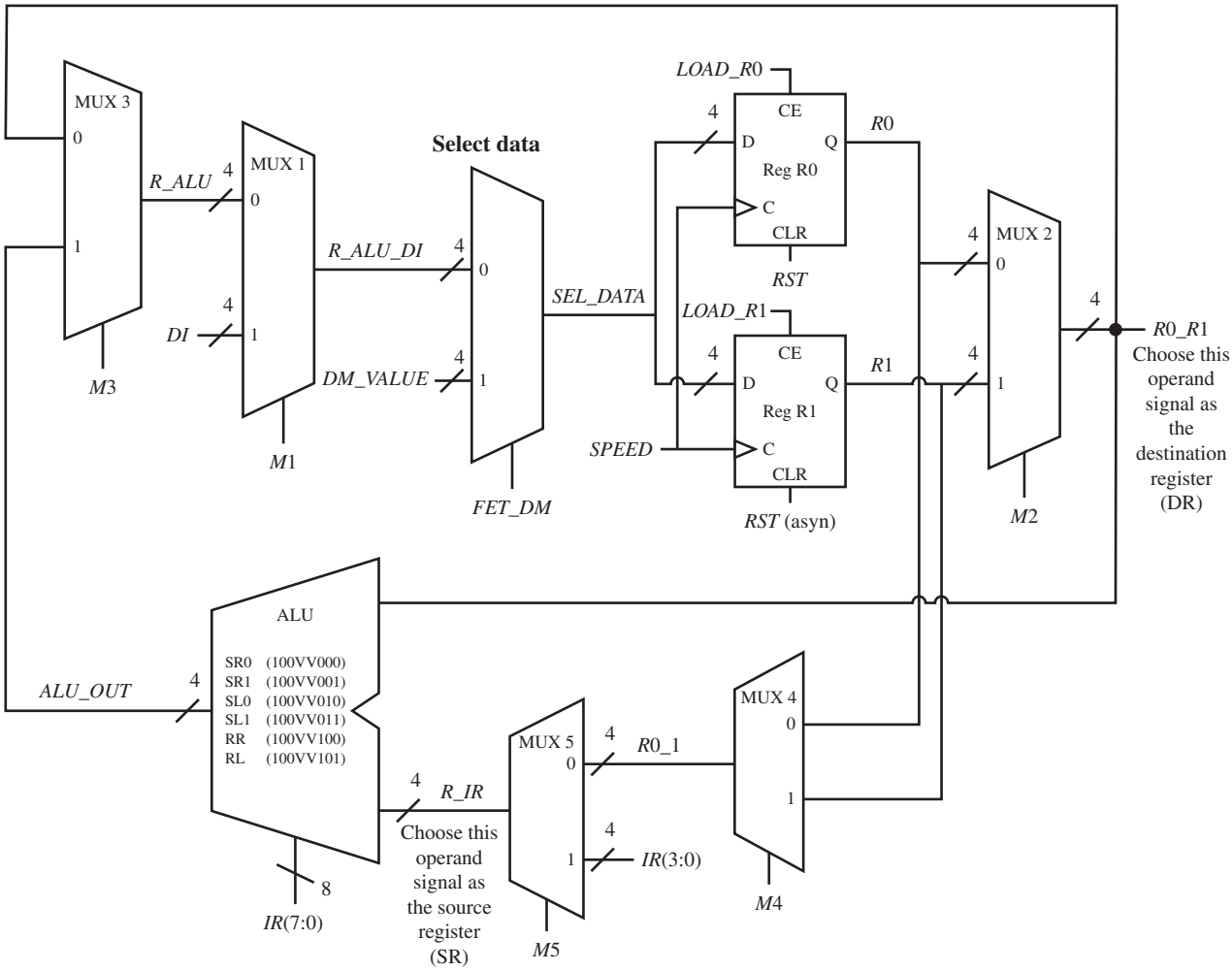


FIGURE 21.2 Data path unit for the shift and rotate instructions SR0, SR1, SL0, SL1, RR, and RL for VBC1-E

The transfer function form for the RR DR,SR instruction is DR ← SR(0) SR(3:1), so the ALU RR instruction must be written as *ALU_OUT* ← *R_IR*(0) *R_IR*(3:1).

The transfer function form for the RL DR,SR instruction is DR ← SR(2:0) SR(3), so the ALU RL instruction must be written as *ALU_OUT* ← *R_IR*(2:0) *R_IR*(3).

Listing 21.3 shows a partial VHDL design for an arithmetic logic unit for the SL0 and RR instructions for VBC1-E using a behavioral design style—that is, a process with a case statement.

21.5 DESIGNING THE INSTRUCTION DECODER FOR THE SHIFT AND ROTATE INSTRUCTIONS FOR VBC1-E

In Figure 21.2, when writing the input signals for the ALU, we consistently made the operand signal *R*0_*R*1 the destination register (DR), and the operand signal *R_IR* the source register (SR) for each of the shift and rotate instructions SR0, SR1, SL0, SL1, RR, and RL for VBC1-E. Because the operand signal for *R*0_*R*1 is DR and the operand signal for *R_IR* is SR for each of

```
alu_process:
process (ir, r_ir)
begin
   alu_out <= "0000";   --default value to prevent creating inferred latches
   case ir(7 downto 5) is
      --for the SL0 and RR instructions
      when "100" => if ir(2 downto 0) = "010" then alu_out <= r_ir(2 downto 0) & '0';
                    elsif ir(2 downto 0) = "100" then alu_out <= r_ir(0) &
                                                                 r_ir(3 downto 1);
                    end if;
      when others => null;
   end case;
end process alu_process;
```

LISTING 21.3 Partial VHDL design for an arithmetic logic unit for the SL0 and RR instructions for VBC1-E using a behavioral design style

the shift and rotate instructions, the instruction decoder truth table for each of the shift and rotate instructions can be represented by the generalized truth table shown in Table 21.2. The Ext bits are the extension bits. Observe that the Boolean equations for the control bits can be calculated without using the extension bits.

TABLE 21.2 Generalized instruction decoder truth table for the shift and rotate instructions

Shift or rotate instruction (Y)	*IR*								Control bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M1***	***M2***	***M3***	***M4***	***M5***	***LOAD_R0***	***LOAD_R1***	***FET_DM***
Y R0,R0	1	0	0	0	0	Ext bits			0	0	1	0	0	1	0	0
Y R0,R1	1	0	0	0	1	Ext bits			0	0	1	1	0	1	0	0
Y R1,R0	1	0	0	1	0	Ext bits			0	1	1	0	0	0	1	0
Y R1,R1	1	0	0	1	1	Ext bits			0	1	1	1	0	0	1	0

To obtain the Boolean equations for the control bits for each of the shift and rotate instructions, substitute SR0, SR1, SL0, SL1, RR, and RL for Y in turn in Table 21.2. Observe that the Boolean equations for the control bits for each instruction can be written identically (in the exact same way), because each instruction has the same truth table.

The Boolean equations for the control bits for each of the shift and rotate instructions SR0, SR1, SL0, SL1, RR, and RL can be written as:

$M1 = 0, M2 = IR(4), M3 = 1, M4 = IR(3), M5 = 0$

$LOAD\_R0 = \overline{IR(4)}, LOAD\_R1 = IR(4), FET\_DM = 0.$

Procedure ID will now be used for the design of the instruction decoder for the SL0 and RR instructions.

Listing 21.4 shows a partial VHDL design for an instruction decoder for the SL0 and RR instructions for VBC1-E using a behavioral design style—that is, a process with a case statement.

LISTING 21.4 Partial VHDL design for an instruction decoder for the SL0 and RR instructions for VBC1-E using a behavioral design style

```
process (ir)
begin
   --default Instruction Decoder output values
   m1 <= '0'; m2 <= '0'; m3 <= '0'; m4 <= '0'; m5 <= '0';
   load_r0 <= '0'; load_r1 <= '0'; fet_dm <= '0';
   case ir (7 downto 5) is
      --for the SL0 and RR instructions
      when "100" => m2 <= ir(4); m3 <= '1'; m4 <= ir(3);
                    load_r0 <= not ir(4); load_r1 <= ir(4);
      when others => null;
   end case;
end process;
```

21.6 DESIGNING THE JMP AND JMPR CIRCUITS FOR VBC1-E

Figure 21.3 shows the circuits for the JMP and JMPR instructions for VBC1-E. These circuits interface directly to the running program counter. The JMP instruction allows a programmer to use an unconditional jump instruction, while the JMPR instruction allows a programmer to use an unconditional jump relative instruction. The HALT instruction, which is an unconditional jump to itself, also works with the circuit in Figure 21.3.

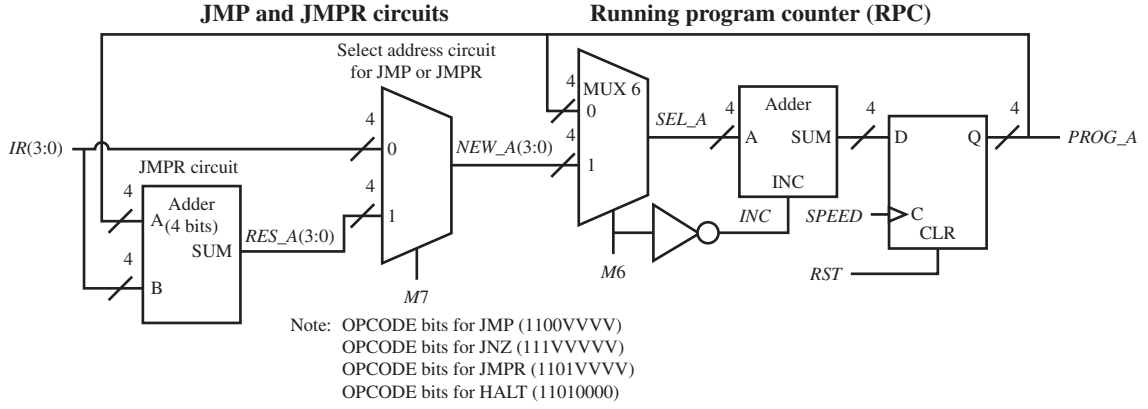


FIGURE 21.3 Circuits for the JMP and JMPR instructions for VBC1-E

Things you should notice about the circuits for the JMP and JMPR instructions in Figure 21.3:

- When $M6 = 0$, this will allow all the instructions except the jump instructions (JNZ, JMP, JMPR, and HALT) to be executed.
- Both $M6$ and a new control signal $M7$ are used to select the JMP instruction or the JMPR instruction.
- When $M6 = 1$ and $M7 = 0$, the JMP can be executed. Using the same control signals also allows the instruction JNZ to be executed.
- When $M6 = 1$ and $M7 = 1$, either the JMPR instruction or the HALT instruction can be executed.

- The JMPR circuit is an Adder with 4 bits that adds the current value of the running program counter that is contained in the bits *PROG_A* to the offset address contained in the bits *IR*(3:0) in the JMPR instruction. When the JMPR instruction has an offset address of 0000, this is the same as the instruction HALT—that is, the instruction JMPR 0 is the same as the instruction HALT.

Listing 21.5 shows a partial VHDL design for the JMP and JMPR circuits for VBC1-E using a behavioral design styles for each circuit—that is, a process with an arithmetic expression and a process with an if statement.

```
--JMPR Circuit
process (prog_a, ir)
begin
     res_a <= prog_a + ir(3 downto 0);
end process;

--Select Address Circuit
process (m7, res_a, ir)
begin
     if m7 = '1' then new_a <= res_a;
     else new_a <= ir(3 downto 0);
     end if;
end process;
```

LISTING 21.5 Partial VHDL design for the JMP and JMPR circuits for VBC1-E using a behavioral design style for each circuit

21.7 DESIGNING THE INSTRUCTION DECODER FOR THE JMP AND JMPR INSTRUCTIONS FOR VBC1-E

We can write the truth table for the JMP and JMPR instructions using the schematic shown in Figure 21.3. Table 21.3 shows a compressed truth table for the JMP instruction.

TABLE 21.3 Compressed truth table for the JMP instruction

	IR								Control bits	
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M6***	***M7***
JMP Addr	1	1	0	0	A	A	A	A	1	0

The Boolean equations for the control bits for the JMP instruction are:

$M6 = 1, M7 = 0$

Table 21.4 shows a compressed truth table for the JMPR instruction.

TABLE 21.4 Compressed truth table for the JMPR instruction

	IR								Control bits	
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***M6***	***M7***
JMP offset	1	1	0	1	OS	OS	OS	OS	1	1

The Boolean equations for the control bits for the JMPR instruction are:

$M6 = 1, M7 = 1$

Procedure ID will now be used for the design of the instruction decoder for the JMP, JMPR, and HALT instructions.

Listing 21.6 shows a partial VHDL design for an instruction decoder for the JMP, JMPR, and HALT instructions for VBC1-E using a behavioral design style—that is, a process with a case statement.

```
process (ir)
begin
   --default Instruction Decoder output values
   m6 <= '0'; m7 <= '0';
   case ir (7 downto 4) is
      when "1100" => m6 <= '1';             --for the JMP instruction
      when "1101" => m6 <= '1'; m7< = '1'; --for the JMPR and HALT instructions
      when others => null;
   end case;
end process;
```

LISTING 21.6 Partial VHDL design for an instruction decoder for the JMP, JMPR, and HALT instructions for VBC1-E using a behavioral design style

PROBLEMS

Section 21.2 Designing the Arithmetic and Logic Instructions Part of the ALU for VBC1-E

21.1 Write the transfer function form for the ALU ADD instruction for VBC1-E shown in Figure 21.1.

21.2 For the data path unit for the arithmetic and logic instructions for VBC1-E shown in Figure 21.1, what OPCODE selects the ADD instruction for the ALU?

21.3 Write the transfer function form for the ALU SUB instruction for VBC1-E shown in Figure 21.1.

21.4 For the data path unit for the arithmetic and logic instructions for VBC1-E shown in Figure 21.1, what OPCODE selects the SUB instruction for the ALU?

21.5 Write the transfer function form for the ALU NOT instruction for VBC1-E shown in Figure 21.1.

21.6 For the data path unit for the arithmetic and logic instructions for VBC1-E shown in Figure 21.1, what OPCODE selects the NOT instruction for the ALU?

21.7 Write the transfer function form for the ALU AND instruction for VBC1-E shown in Figure 21.1.

21.8 For the data path unit for the arithmetic and logic instructions for VBC1-E shown in Figure 21.1, what OPCODE selects the AND instruction for the ALU?

21.9 Write the transfer function form for the ALU OR instruction for VBC1-E shown in Figure 21.1.

21.10 For the data path unit for the arithmetic and logic instructions for VBC1-E shown in Figure 21.1, what OPCODE selects the OR instruction for the ALU?

21.11 Write the transfer function form for the ALU XNOR instruction for VBC1-E shown in Figure 21.1.

21.12 For the data path unit for the arithmetic and logic instructions for VBC1-E shown in Figure 21.1, what OPCODE selects the XNOR instruction for the ALU?

21.13 Write a partial VHDL design for an arithmetic logic unit for the NOT and AND instructions for VBC1-E using a behavioral design style—that is, a process with a case statement.

21.14 Write a partial VHDL design for an arithmetic logic unit for the OR and XNOR instructions for VBC1-E using a behavioral design style—that is, a process with a case statement.

Section 21.3 Designing the Instruction Decoder for the Arithmetic and Logic Instructions for VBC1-E

21.15 What procedure was followed in the text to write the ALU signals to satisfy the transfer function forms for the arithmetic and logic instructions in the design of VBC1-E?

21.16 Write the control bits for the instructions ADD, SUB, NOT, AND, OR, and XNOR in Figure 21.1 when the operand signal *R*0_*R*1 is the destination register for the ALU and the operand signal *R_IR* is the source register for the ALU. Hint: Use Table 21.1.

21.17 In the machine code form of an instruction, what bit in the arithmetic and logic instructions specifies the destination register?

21.18 In the machine code form of an instruction, what bit in the arithmetic and logic instructions specifies the source register?

21.19 Show a partial VHDL design for an instruction decoder for the NOT and AND instructions for VBC1-E using Procedure ID.

21.20 Show a partial VHDL design for an instruction decoder for the OR and XNOR instructions for VBC1-E using Procedure ID.

Section 21.4 Designing the Shift and Rotate Instructions Part of the ALU for VBC1-E

21.21 Write the transfer function form for the ALU SR0 instruction for VBC1-E shown in Figure 21.2.

21.22 For the data path unit for the shift and rotate instructions for VBC1-E shown in Figure 21.2, what OPCODE selects the SR0 instruction for the ALU?

21.23 Write the transfer function form for the ALU SR1 instruction for VBC1-E shown in Figure 21.2.

21.24 For the data path unit for the shift and rotate instructions for VBC1-E shown in Figure 21.2, what OPCODE selects the SR1 instruction for the ALU?

21.25 Write the transfer function form for the ALU SL0 instruction for VBC1-E shown in Figure 21.2.

21.26 For the data path unit for the shift and rotate instructions for VBC1-E shown in Figure 21.2, what OPCODE selects the SL0 instruction for the ALU?

21.27 Write the transfer function form for the ALU SL1 instruction for VBC1-E shown in Figure 21.2.

21.28 For the data path unit for the shift and rotate instructions for VBC1-E shown in Figure 21.2, what OPCODE selects the SL1 instruction for the ALU?

21.29 Write the transfer function form for the ALU RR instruction for VBC1-E shown in Figure 21.2.

21.30 For the data path unit for the shift and rotate instructions for VBC1-E shown in Figure 21.2, what OPCODE selects the RR instruction for the ALU?

21.31 Write the transfer function form for the ALU RL instruction for VBC1-E shown in Figure 21.2.

21.32 For the data path unit for the shift and rotate instructions for VBC1-E shown in Figure 21.2, what OPCODE selects the RL instruction for the ALU?

21.33 Write a partial VHDL design for an arithmetic logic unit for the SR0 and SR1 instructions for VBC1-E using a behavioral design style—that is, a process with a case statement.

21.34 Write a partial VHDL design for an arithmetic logic unit for the SL1 and RL instructions for VBC1-E using a behavioral design style—that is, a process with a case statement.

Section 21.5 Designing the Instruction Decoder for the Shift and Rotate Instructions for VBC1-E

21.35 What procedure was followed in the text to write the ALU signals to satisfy the transfer function forms for the shift and rotate instructions in the design of VBC1-E?

21.36 Write the control bits for the instructions SR0, SR1, SL0, SL1, RR, and RL in Figure 21.2 when the operand signal *R*0_*R*1 is the destination register for the ALU and the operand signal *R_IR* is the source register for the ALU. Hint: Use Table 21.2.

21.37 Show a partial VHDL design for an instruction decoder for the SR0 and SR1 instructions for VBC1-E using Procedure ID.

21.38 Show a partial VHDL design for an instruction decoder for the SL1 and RL instructions for VBC1-E using Procedure ID.

Section 21.6 Designing the JMP and JMPR Circuits for VBC1-E

21.39 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, what control bits are used to select the JMP instruction or the JMPR instruction?

21.40 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, what are the values of the control bits *M*6 and *M*7 that will allow the instruction JMP to be executed?

21.41 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, what are the values of the control bits *M*6 and *M*7 that will allow the instruction JNZ to be executed?

21.42 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, what are the values of the control bits *M*6 and *M*7 that will allow the instruction JMPR to be executed?

21.43 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, what are the values of the control bits *M*6 and *M*7 that will allow the instruction HALT to be executed?

21.44 Write a partial VHDL design for the JMP and JMPR circuits for VBC1-E using an arithmetic expression and a conditional signal assignment.

Section 21.7 Designing the Instruction Decoder for the JMP and JMPR Instructions for VBC1-E

21.45 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, write a compressed truth table for the JMP instruction.

21.46 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, write a compressed truth table for the JNZ instruction.

21.47 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, write a compressed truth table for the JMPR instruction.

21.48 For the circuits for the JMP and JMPR instructions for VBC1-E shown in Figure 21.3, write a truth table for the HALT instruction.

21.49 Write a partial VHDL design for an instruction decoder for just the JMP instruction for VBC1-E using Procedure ID.

21.50 Write a partial VHDL design for an instruction decoder for just the JMPR, and HALT instructions for VBC1-E using Procedure ID.

Chapter 22

Designing a Circuit to Prevent Program Execution During Manual Loading for VBC1-E

Chapter Outline

22.1 INTRODUCTION

This is the fourth in the series of chapters that teaches you how to expand the design of VBC1 to make VBC1-E. In this chapter, you will learn how to design a circuit that modifies manual loading to provide fewer distractions from flashing outputs as a program is manually loaded into instruction memory or manually stepped through instruction memory to view its contents. This chapter and its corresponding experiment (Experiment 22 in Appendix A) may be skipped without affecting the design of VBC1-E.

22.2 DESIGNING A CIRCUIT TO MODIFY MANUAL LOADING FOR VBC1-E

Figure 22.1 shows a circuit that prevents program execution during manual loading for VBC1-E.

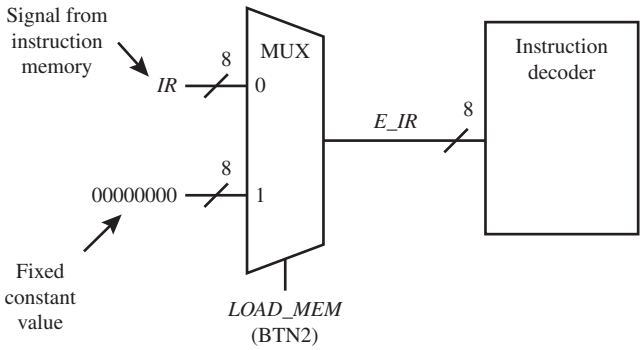


FIGURE 22.1 Circuit that prevents program execution during manual loading for VBC1-E.

When the signal *LOAD_MEM* = 0 (BTN2 is not pressed), instructions are supplied to the instruction decoder via the signal *IR*. This allows instructions to be single stepped or run by VBC1-E. When the signal *LOAD_MEM* = 1 (BTN2 is pressed), instruction can be loaded into instruction memory; however, the instruction decoder is supplied with a fixed constant value of 00000000, which prevents instructions from executing during the manual loading process.

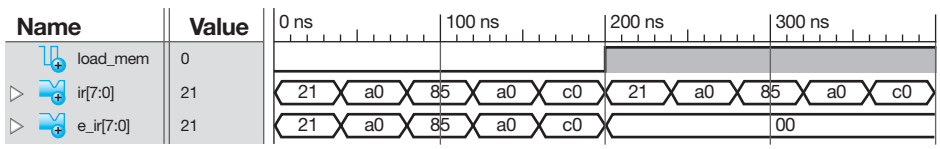
Listing 22.1 shows a complete VHDL design for the MUX that prevents program execution during manual loading for VBC1-E.

LISTING 22.1
Complete VHDL design for the MUX that prevents program execution during manual loading for VBC1-E (project: MUX)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MUX is port (
     load_mem : in STD_LOGIC;
     ir : in STD_LOGIC_VECTOR (7 downto 0);
     e_ir : out STD_LOGIC_VECTOR (7 downto 0)
     );
end MUX;

architecture Behavioral of MUX is
begin
process (ir)
begin
     if load_mem = '0' then e_ir <= ir;
     else e_ir <= "00000000";
     end if;
end process;
end Behavioral;
```

Waveform 22.1 shows a simulation waveform diagram with the correct functionality of design entity MUX.

WAVEFORM 22.1
Simulation waveform diagram with the correct functionality for the design entity MUX



Things you should notice about the waveforms in Waveform 22.1:

- First, *LOAD_MEM* is set to 0.
- Notice that a 2-bit flashing lights program LOADI R0,1(21), OUT R0,0(A0), RL R0,R0(85), OUT R0,0(A0), and JMP 0(C0) is shown by the signal *IR*(7:0) via the machine code bits (21, A0, 85, A0, C0) shown in hexadecimal following each assembly language instruction.
- The signal *E_IR* at the output of MUX follows the signal *IR*(7:0) when *LOAD_MEM* is 0, which allows the simple program to be single stepped or run.
- When *LOAD_MEM* is set to 1, the 2-bit flashing lights program is repeated via the signal *IR*(7:0). Notice that the signal *E_IR* at the output of the MUX follows the value 00000000 (00) that is supplied to input 1 of the MUX and not the 2-bit flashing lights program. This prevents the 2-bit flashing lights program from executing when manual loading occurs.
- Waveform 22.1 shows that the VHDL design in Listing 22.1 does in fact provide the correct design for the design entity MUX.

22.3 MODIFYING THE INSTRUCTION DECODER FOR MANUAL LOADING FOR VBC1-E

In addition to the MUX in Figure 22.1, every signal *IR* in the VHDL code for the instruction decoder for VBC1-E must be changed to the signal *E_IR*. A simple find-and-replace operation can be used to make these changes to the code for the instruction decoder.

These changes will provide fewer distractions due to the output ports that may be lighted, while you manually load instruction memory or you manually view the contents of instruction memory of VBC1-E.

Note: These changes can also be made to the final hardware design of VBC1—that is, Experiment 17 and Experiment 17L in Appendix A.

PROBLEMS

Section 22.2 Designing a Circuit to Modify Manual Loading for VBC1-E

22.1 In Figure 22.1, do you think the fixed value must be eight zeros, or will any constant consisting of 8 bits prevent program execution during manual loading or during manually stepping through instruction memory to observe its contents? Provide a reason for your answer.

22.2 In Figure 22.1, can the machine code for the AND instruction 01000101 be used for the fixed value? What occurs when *E_IR* = 01000101 due to *LOAD_MEM* = 1? What occurs when *E_IR* = *IR* due to *LOAD_MEM* = 0?

22.3 In Figure 22.1, can the machine code for the STORE instruction 00000101 be used for the fixed constant value? What occurs when *E_IR* = 00000101 due to *LOAD_MEM* = 1? What occurs when *E_IR* = *IR* due to *LOAD_MEM* = 0?

22.4 How can you confirm or disprove that any fixed constant value can be used at input 1 of the MUX in Figure 22.1 in a hardware design of VBC1-E?

22.5 Show complete VHDL code for the MUX in Figure 22.1. Use a conditional signal assignment statement. Show a simulation for your design to show that your VHDL code is functionally correct. Name the design entity MUX. Show the simulation waveform using unsigned decimal numbers.

22.6 Write a conditional signal assignment for the MUX in Figure 22.1 with the fixed constant value changed from 00000000 to 11111111.

22.7 Write a process with an if statement for the MUX in Figure 22.1 with the fixed constant value changed from 00000000 to 10101010.

Section 22.3 Modifying the Instruction Decoder for Manual Loading for VBC1-E

23.8 With the MUX added to the circuit in Figure 22.1, explain how the instruction decoder for VBC1-E must be modified to provide proper circuit operation.

22.9 From a distraction point of view, discuss the purpose of the MUX in the circuit in Figure 22.1 when you desire to manually load or view the contents of instruction memory.

23 Chapter

Designing Extended Instruction Memory for VBC1-E

Chapter Outline

23.1 INTRODUCTION

This is the fifth in the series of chapters that teaches you how to expand the design of VBC1 to make VBC1-E. In this chapter, you will learn how to design circuits that double the number of program memory storage locations to add an extended instruction memory. Modifications will be made to the circuits for the running program counter, the proper address, the loading program counter, and the JMPR circuit.

23.2 MODIFYING THE INSTRUCTION MEMORY TO ADD EXTENDED INSTRUCTION MEMORY FOR VBC1-E

Currently, there are 16 storage locations in the instruction memory. Our task is to add 16 more storage locations. The added block of memory is called extended instruction memory. Up to this point, all instructions were placed in the normal instruction memory (section 0). We will refer to the normal instruction memory as just instruction memory. With added memory, 16 more storage locations are placed in extended instruction memory (section 1). This means that we need to add one more memory address bit—that is, *MEM_ADDR*(4) to the design of the instruction memory to obtain 32 ($2^5 = 32$) total memory storage locations. *MEM_ADDR*(4) is actually the 5th bit of the memory address.

With the added bit or 5th bit of memory address, the instruction memory covers the address range of 0 0000 through 0 1111 (section 0), and extended instruction memory covers the address range 1 0000 through 1 1111 (section 1). The instruction memory and the extended instruction memory represent the two parts of total instruction memory (RAM 32 × 8).

Remember that extended instruction memory is only used to execute instructions initiated by a software interrupt caused by the execution of the instruction INT (interrupt), or to execute instructions initiated by a hardware interrupt caused by pressing a push-button switch. A software interrupt and a hardware interrupt will be added to VBC1-E later.

Figure 23.1 shows the modified circuits for the total instruction memory and the 4-to-1 MUX array for the multiplexed display system for VBC1-E.

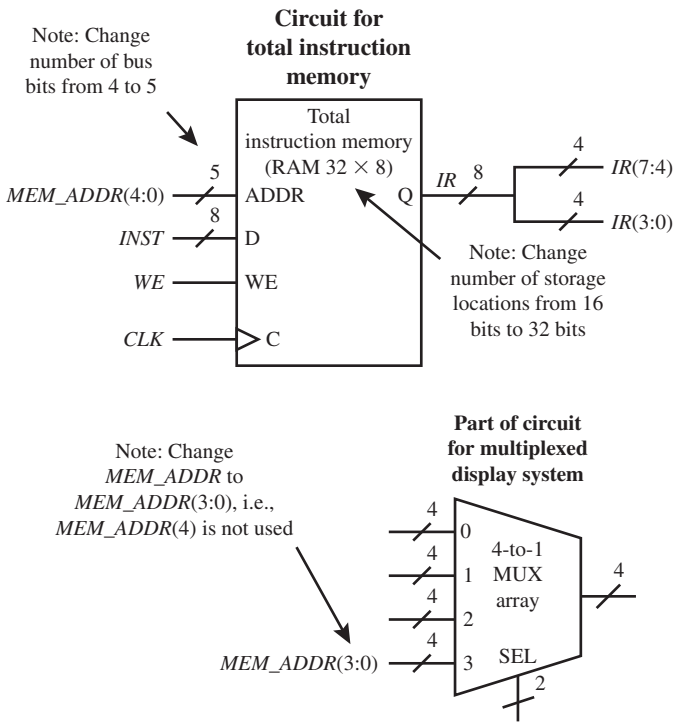


FIGURE 23.1 Modified circuits for the total instruction memory and the 4-to-1 MUX array for the multiplexed display system for VBC1-E

The changes to the circuit for instruction memory are made to double the number of program memory storage locations, which consists of instruction memory (section 0) with addresses 0 through 15 and extended instruction memory (section 1) with addresses 16 through 31. The change to the circuit for the multiplexed display system is made to ensure only 4 bits are used for the *MEM_ADDR* and not 5 bits. Later, a single LED will be added to indicate the status of *MEM_ADDR*(4).

Figure 23.2 (on page 499) shows the total instruction memory map for VBC1-E.

Listing 23.2 shows a complete VHDL design for the total instruction memory for VBC1-E in Figure 23.1.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity Total_Instruction_Memory is port (
   mem_addr : in STD_LOGIC_VECTOR (4 downto 0);
   inst : in STD_LOGIC_VECTOR (7 downto 0);
                                                    (Continued)
```

LISTING 23.2 Complete VHDL design for the total instruction memory for VBC1-E (project: Total_Instruction_Memory)

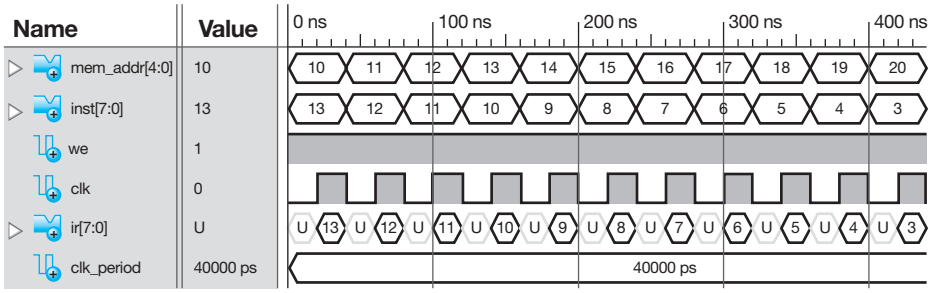
```
   we, clk : in STD_LOGIC;
   ir : out STD_LOGIC_VECTOR (7 downto 0)
   );
end Total_Instruction_Memory;
architecture Mixed of Total_Instruction_Memory is
   type mem_type is array (0 to 31) of std_logic_vector (7 downto 0);
   signal mem : mem_type;
begin
process (clk)
begin
   if rising_edge (clk) then
      if we = '1' then mem (conv_integer (mem_addr)) <= inst;
      end if;
   end if;
end process;
      ir <= mem (conv_integer (mem_addr));
end Mixed;
```

Things you should notice about the VHDL design in Listing 23.2:

- The range of mem_addr is (4 **downto** 0).
- The clock is signal clk.
- Mem_type is an enumerated data type with the ordered array (0 **to** 31) of the data type std_logic_vector (7 **downto** 0).

Waveform 23.1 shows the correct functionality of design entity Total_Instruction_Memory.

WAVEFORM 23.1
Simulation for the correct functionality of design entity Total_Instruction_Memory: (a) writing instructions into memory; (b) reading previously written instructions from memory



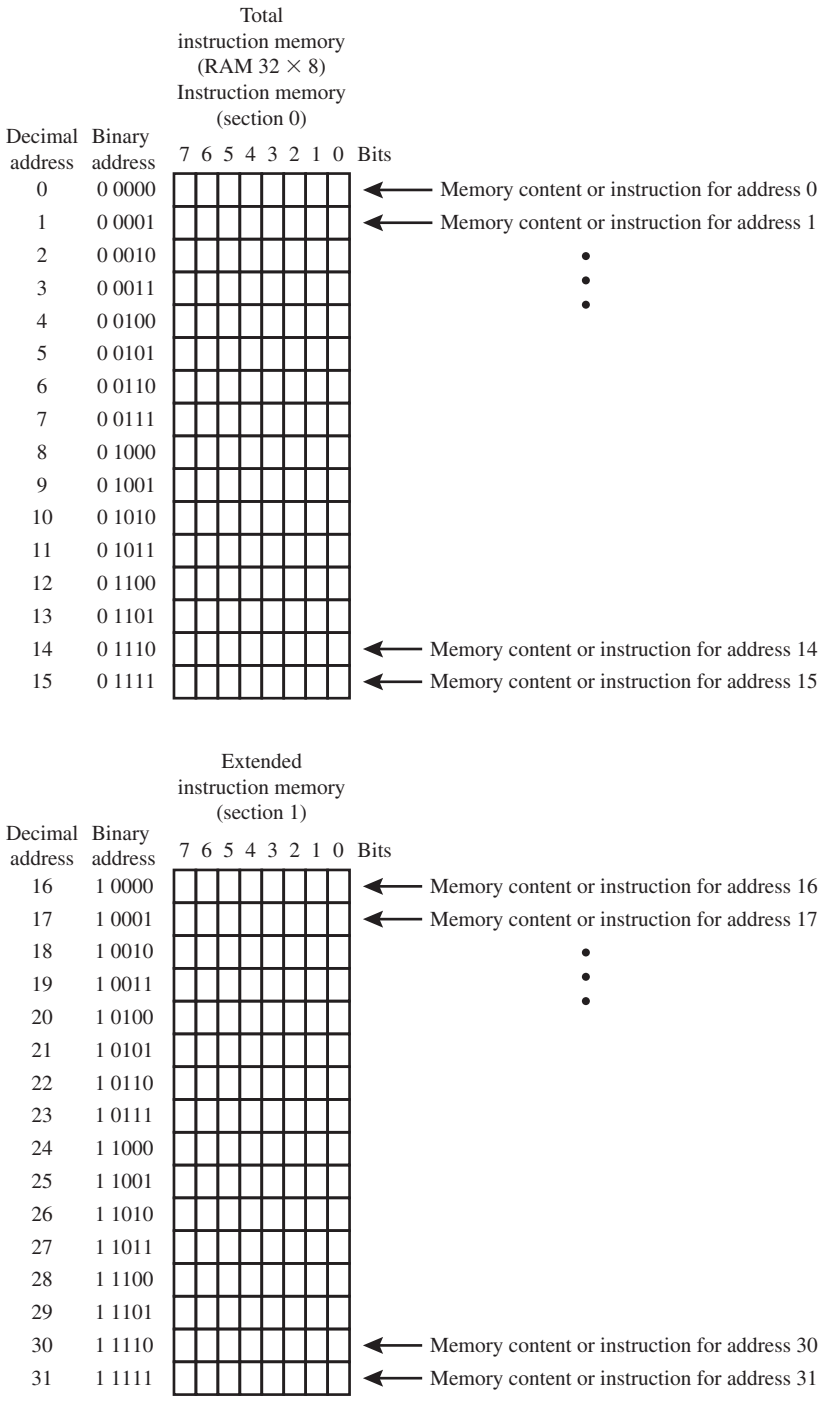


FIGURE 23.2 Total instruction memory map for VBC1-E

Things you should notice about the waveforms in Waveform 23.1:

- In the waveform diagrams in (a), the memory address is changed from 10 through 20 via the input signal *MEM_ADDR*.

- Random values of data that represent instructions (machine code instructions) are written into the total instruction memory each time the clock *CLK* ticks because *WE* is set to 1, as shown by the output signal *IR*.
- In the waveform diagrams in (b), *WE* is set to 0, which allows the total instruction memory to be read. By closely observing the waveform diagrams, it can be seen that the data written into each memory address in (a) is read correctly in (b), confirming that the design entity Total_Instruction_Memory works properly.

23.3 MODIFYING THE RUNNING PROGRAM COUNTER CIRCUIT FOR VBC1-E

Figure 23.3 shows the modified circuits for the running program counter, the proper address, the loading program counter, and the JMP and JMPR circuits for VBC1-E.

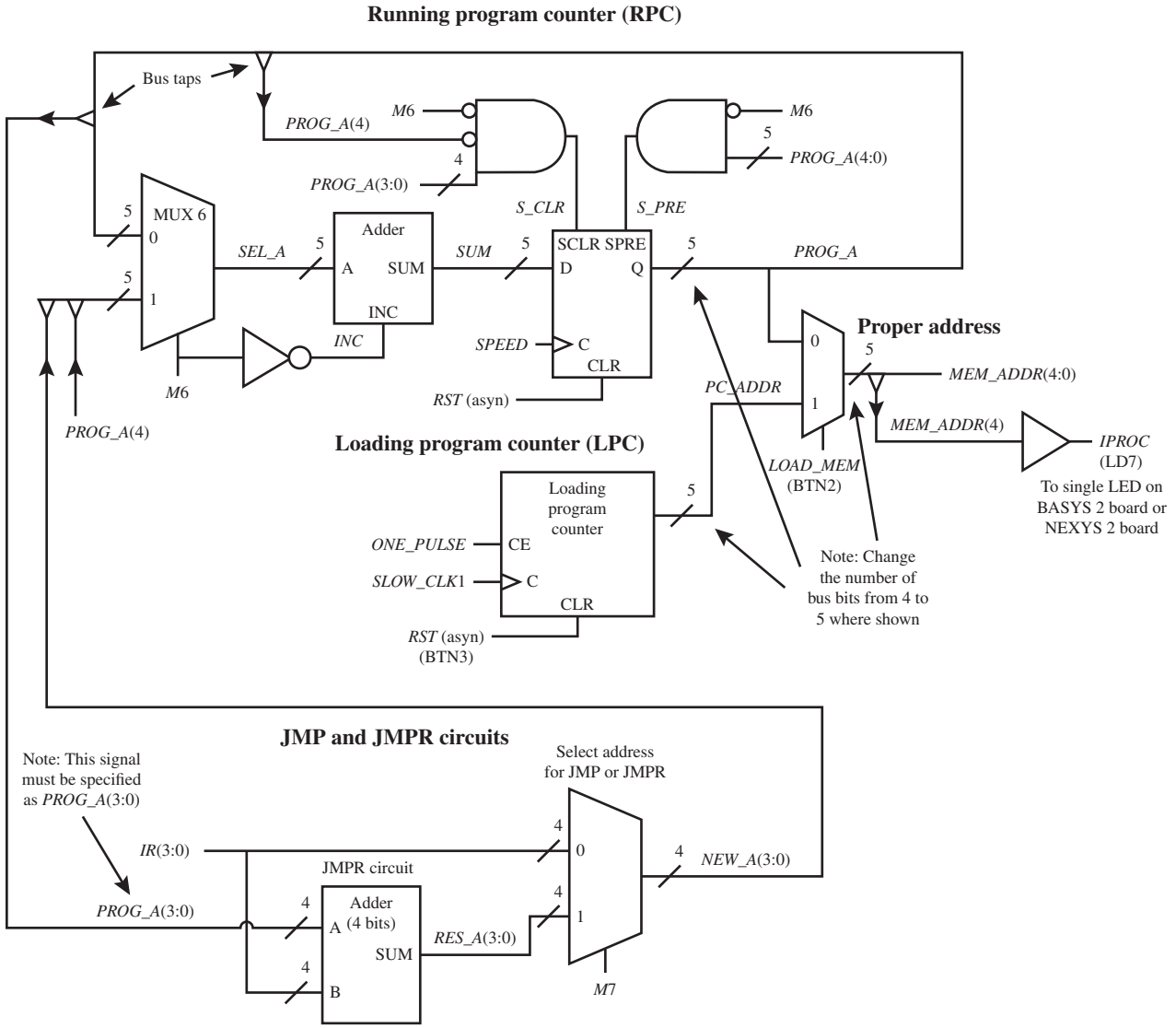


FIGURE 23.3 Modified circuits for the RPC, the proper address, the LPC, and the JMP and JMPR circuits for VBC1-E

Some of the modifications to the running program counter are subtle. First, notice that all the main buses are now 5 bits rather than 4 bits. One additional D flip-flop is added to the array of D flip-flops in the running program counter. The not-so-subtle modifications are the addition of the D flip-flop input SCLR (synchronous clear) and the addition of the D flip-flop input SPRE (synchronous preset). Remember that input SCLR and input SPRE are synchronous inputs, which means that they do not cause a change unless they are asserted or equal to 1 at the next rising edge of *SPEED*.

The running program counter in Figure 23.3 operates according to two key principles: (1) the running program counter rolls over to the lowest address in each section of memory; (2) the running program counter is designed so that all the instructions except INT and RETA only operate within the memory section in which they are placed.

The AND gate circuit feeding the D flip-flop input SCLR via signal *S_CLR* ensures that the running program counter rolls over from address 0 1111 to 0 0000 when signal *PROG_A* (3:0) = 1111, signal *PROG_A*(4) = 0, and signal *M*6 = 0 at the next rising edge of *SPEED*. If a jump instruction is located in section 0 and *M*6 = 1, which indicates that a jump instruction is in the process of being executed, then the jump instruction can only jump to an address in section 0 because the address of the jump instruction contains only 4 bits.

The AND gate circuit feeding the D flip-flop input SPRE via signal *S_PRE* ensures that the running program counter rolls over from address 1 1111 to 1 0000 when signal *PROG_A* (4:0) = 1 1111 and signal *M*6 = 0 at the next rising edge of *SPEED*. If a jump instruction is located in section 1 and *M*6 = 1, which indicates that a jump instruction is in the process of being executed, then the jump instruction can only jump to an address in section 1 because the address of the jump instruction contains only 4 bits.

23.4 MODIFYING THE PROPER ADDRESS CIRCUIT FOR VBC1-E

In Figure 23.3, observe that the proper address circuit has been modified. Notice that the inputs and the output for the proper address circuit are now 5 bits rather than 4 bits. Also observe that a bus tap is shown for the signal *MEM_ADDR*(4). This signal feed into a buffer that provides the output signal *IPROC* (interrupt process). The signal for the 5th bit of the memory address is named *IPROC*, so *IPROC* = *MEM_ADDR*(4). When the signal *IPROC* is 0, VBC1-E executes instructions in instruction memory (section 0), but when the signal *IPROC* is 1, VBC1-E executes instructions in extended instruction memory (section 1). Extended instruction memory is used to execute instructions initiated by a software or hardware interrupt. Later, a software interrupt (instruction INT) will be added in (Chapter 24), and a hardware interrupt (a manual push-button switch) will be added in Chapter 25.

Signal *IPROC* is connected to a single LED so that we can observe when VBC1-E is in instruction memory (section 0) or in extended instruction memory (section 1). When signal *IPROC* is 0, the single LED will turn off to indicate that an instruction in instruction memory (section 0) is being executed, and when signal *IPROC* is 1, the single LED will turn on to indicate that an instruction in extended instruction memory (section 1) is being executed.

23.5 MODIFYING THE LOADING PROGRAM COUNTER CIRCUIT FOR VBC1-E

In Figure 23.3, observe that the loading program counter has been modified. Notice that the number of output bits for signal *PC_ADDR* is now 5 bits rather than 4 bits. This means that the loading program counter becomes a binary-up counter with 5 bits that counts from 0 0000 to 1 1111 (0 to 31).

23.6 MODIFYING THE JMPR CIRCUIT FOR VBC1-E

In Figure 23.3, observe that the JMPR circuit has been modified. This change is very subtle. We changed the signal that feeds into the A input of the Adder of the JMPR circuit from *PROG_A* to *PROG_A*(3:0). If this change is not made, there is a mismatch in the number of bits supplied to the A input of the Adder, because *PROG_A* was changed from *PROG_A*(3:0) to *PROG_A*(4:0) when the running program counter was changed.

After all the changes are made that are discussed in this chapter, instructions can be added to instruction memory and extended instruction memory, but when push button BTN3 is pressed to clear the program counter to 0 0000, only instructions in instruction memory (section 0) will execute. In the next chapter, circuitry will be added for the instructions INT and IRET so that instructions in instruction memory (section 0) and also extended instruction memory (section 1) can be executed.

PROBLEMS

Section 23.2 Modifying the Instruction Memory to Add Extended Instruction Memory for VBC1-E

23.1 In Figure 23.1, how many address bits are used in the total instruction memory? What signal supplies the address bits for the total instruction memory (be sure to include the range for the bus)?

23.2 In Figure 23.1, how many memory storage locations does the total instruction memory contain for VBC1-E?

23.3 In Figure 23.1, what is the range of the instruction memory (section 0) and what is the range of the extended instruction memory (section 1)?

23.4 Will the VHDL code in Listing P23.4 on the facing page work correctly for storing data into the total instruction memory in Figure 23.1? Explain your answer. Note that this code has been modified from the code provide earlier in the chapter. (Hint: Compile the code and run a simulation on the code to observe the simulation result.)

23.5 In Figure 23.1, explain why is it necessary to change the signal *MEM_ADDR* to *MEM_ADDR*(3:0).

23.6 In Figure 23.1, how many bits are required for the machine code for each instruction for VBC1-E?

Section 23.3 Modifying the Running Program Counter Circuit for VBC1-E

23.7 In Figure 23.3, how many bits are required for each of the main buses—that is, *SEL_A*, *SUM*, and *PROG_A*—in the running program counter for VBC1-E?

23.8 In Figure 23.3, how many D flip-flops are contained in the array of D flip-flops in the running program counter for VBC1-E?

23.9 List the two key principles concerning the operation of the running program counter in Figure 23.3.

23.10 Provide the Boolean equation for signal *S_CLR* that forces the program counter to roll over to the lowest address in instruction memory (section 0) at the next rising edge of *SPEED*. What is the lowest address in instruction memory (section 0)?

23.11 Provide the Boolean equation for signal *S_PRE* that forces the program counter to roll over to the lowest address in extended instruction memory (section 1) at the next rising edge of *SPEED*. What is the lowest address in extended instruction memory (section 1)?

23.12 Explain why a jump instruction can only jump to an address in the section of memory in which it is placed. How many bits would a jump instruction have to contain to jump outside of the section of memory in which it is placed?

Section 23.4 Modifying the Proper Address Circuit for VBC1-E

23.13 In Figure 23.3, list the modifications to the proper address circuit for VBC1-E.

23.14 In Figure 23.3, when *IPROC* = 0, in which section of the total instruction memory are instructions being executed by VBC1-E?

23.15 In Figure 23.3, when *IPROC* = 1, in which section of the total instruction memory are instructions being executed by VBC1-E?

23.16 What will initiate the execution of instructions in extended instruction memory?

23.17 How can we observe which section of the total instruction memory for VBC1-E is being used to execute an instruction?

23.18 When the LED for the signal *IPROC* is turned off, which section of the total instruction memory is executing an instruction?

23.19 When the LED for the signal *IPROC* is turned on, which section of the total instruction memory is executing an instruction?

Section 23.5 Modifying the Loading Program Counter Circuit for VBC1-E

23.20 In Figure 23.3, how many bits does the modified loading program counter contain?

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity total_instruction_memory2 is port (
     mem_addr : in STD_LOGIC_VECTOR (4 downto 0);
     inst : in STD_LOGIC_VECTOR (7 downto 0);
     we, clk : in STD_LOGIC;
     ir : out STD_LOGIC_VECTOR (7 downto 0)
     );
end total_instruction_memory2;

architecture Behavioral of total_instruction_memory2 is
     type mem_type is array (0 to 31) of std_logic_vector (7 downto 0);
     signal mem : mem_type;
begin
process (clk)
begin
     if rising_edge (clk) then
          if we = '1' then mem (conv_integer (mem_addr)) <= inst;
          end if;
     end if;
     ir <= mem (conv_integer (mem_addr));
end process;
end Behavioral ;
```

LISTING P23.4

23.21 In Figure 23.3, what is the range of addresses for the modified loading program counter?

23.22 A modulo n counter is a counter with n states, where $n > 1$. How many states does the modified loading program counter for VBC1-E contain?

23.23 In Figure 23.3, what is the signal at the output of the modified loading program counter (be sure to include the range for the bus)?

Section 23.6 Modifying the JMPR Circuit for VBC1-E

23.24 In Figure 23.3, what is the very subtle change that is made to the JMPR circuit?

23.25 In Figure 23.3, why is it necessary to change the signal *PROG_A* to *PROG_A*(3:0) in the JMPR circuit?

24 Chapter

Designing the Software Interrupt Circuits for VBC1-E

Chapter Outline

24.1 INTRODUCTION

This is the sixth in the series of chapters that teaches you how to expand the design of VBC1 to make VBC1-E. In this chapter, you will learn how to design the circuits that provide the INT (software interrupt) and IRET (interrupt return) instructions. With this additional circuitry, you can load an interrupt service routine in extended instruction memory (section 1), access the extended instruction memory via the INT instruction, execute the interrupt service routine, and return to instruction memory (section 0) via the IRET instruction.

24.2 DESIGNING THE MODIFIED CIRCUIT FOR THE RUNNING PROGRAM COUNTER AND THE SELECT CIRCUIT FOR VBC1-E

Figure 24.1 shows the modified circuit for the running program counter and the select circuit for VBC1-E.

Things you should notice about the circuits for the modified running program counter and the select circuit in Figure 24.1:

- The two control signals *B*1 and *B*0 determine which signal is routed to the output of the 4-to-1 MUX array. *B*1 and *B*0 are generated by the instruction decoder, which is in the control unit of VBC1-E.
- When *B*1*B*0 is 00, SUM1 is routed through the 4-to-1 MUX array and is stored in the loadable register circuit at the next rising edge of the signal *SPEED*.
- SUM1 provides the address of the next instruction for the running program counter.

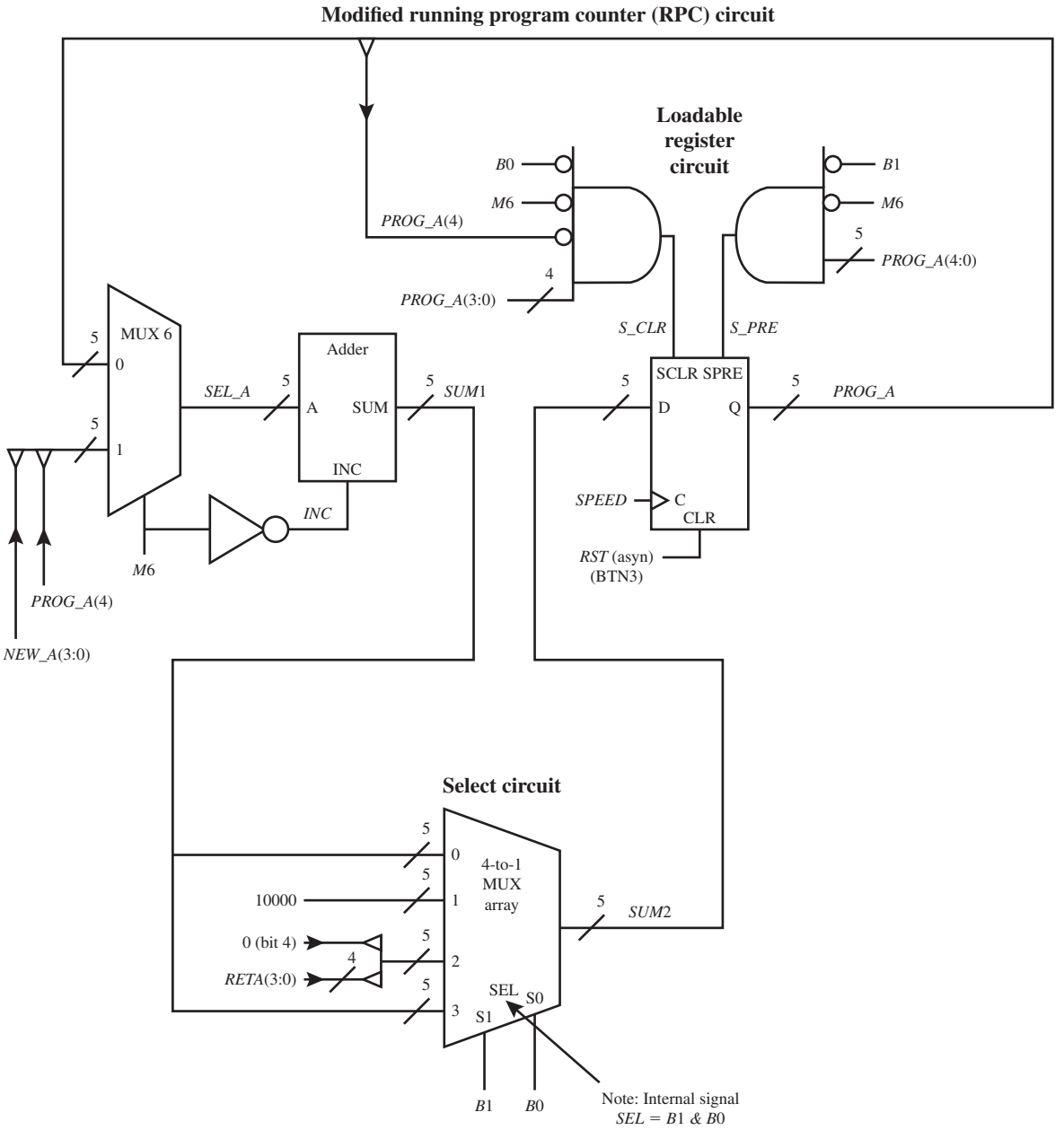


FIGURE 24.1 Modified circuit for the running program counter and select circuit for VBC1-E

- When *B*1*B*0 is 01, the value 10000 provides the address for the running program counter, and this only occurs when the INT instruction is executed.
- When *B*1*B*0 is 10, 0 RETA(3:0) provides the address for the running program counter, which is the address of the instruction following the INT instruction, and this only occurs when the IRET instruction is executed.
- The signal *B*0 is inverted and added to the AND gate that supplies the signal *S_CLR* to the loadable register circuit.

- The signal *B*1 is inverted and added to the AND gate that supplies the signal *S_PRE* to the loadable register circuit.
- When the signal *S_CLR* is asserted—that is, is equal to 1, this causes the loadable register circuit of the RPC to synchronously clear to 00000 at the next rising edge of the signal *SPEED*. The signal *S_CLR* causes the running program counter to follow the sequence 00000(0) though 01111(15) and roll over (or return) to 00000(0) when the last instruction at address 01111(15) is executed in instruction memory (section 0). Only the INT instruction and the jump instructions do not cause a rollover from 01111(15) back to 00000(0) via the signal *S_CLR*.
- When the signal *S_PRE* is asserted—that is, is equal to 1—this causes the loadable register circuit of the RPC to synchronously preset to 10000 at the next rising edge of the signal *SPEED*. The signal *S_PRE* causes the running program counter to follows the sequence 10000(16) though 11111(31) and roll over (or return) to 10000(16) when the last instruction at address 11111(31) is executed in extended instruction memory (section 1). Only the IRET instruction and the jump instructions do not cause a rollover from 11111(31) back to 100000(16) via the signal *S_PRE*.

Listing 24.1 shows a complete VHDL design for the select circuit—that is, the 4-to-1 MUX array—for VBC1-E.

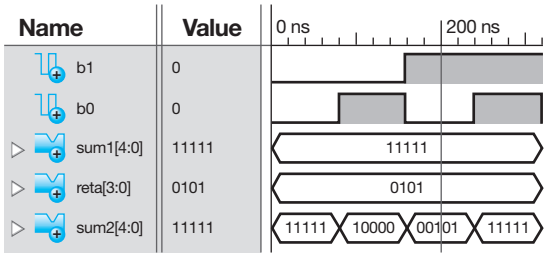
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Select_Circuit is port (
   b1,b0 : in STD_LOGIC;
   sel : inout STD_LOGIC_VECTOR (1 downto 0);
   sum1 : in STD_LOGIC_VECTOR (4 downto 0);
   reta : in STD_LOGIC_VECTOR (3 downto 0);
   sum2 : out STD_LOGIC_VECTOR (4 downto 0)
   );
end Select_Circuit;

architecture dataflow of Select_Circuit is
begin
   sel <= b1 & b0;   --concatenation operator used to form a vector or a bus
with sel select      --selected signal assignment
   sum2 <= sum1                    when "00",
           "10000"                 when "01",  --occurs only when INT is executed
           '0' & reta(3 downto 0) when "10",  --occurs only when IRET is executed
           sum1                    when "11",
           sum1                    when others;
end dataflow;
```

LISTING 24.1 Complete VHDL design for the select circuit for VBC1-E (project: Select_Circuit)

Things that you should notice about the complete VHDL design in Listing 24.1:

- The concatenation operator is used to form a bus named *SEL*, which has the range (1:0).
- The output of the select circuit is SUM1 when INT or IRET is not executed.
- The output of the select circuit is 10000 only when INT is executed.
- The output of the select circuit is 0 *RETA*(3:0) only when IRET is executed.

Waveform 24.1 shows a simulation waveform diagram with the correct functionality of design entity Select_Circuit.



WAVEFORM 24.1 Simulation waveform diagram with the correct functionality of design entity Select_Circuit

Things you should notice about the waveforms in Waveform 24.1:

- All the vector signals are displayed in binary values.
- Observe that *SUM*2 follows *SUM*1 when *B*1*B*0 = 00 and *B*1*B*0 = 11.
- Observe that *SUM*2 is the constant value 10000 when *B*1*B*0 = 01.
- Observe that *SUM*2 follows 0 *RETA*(3:0) when *B*1*B*0 = 10.
- Waveform 24.1 shows that design entity Select_Circuit is functionally correct.

The transfer function form for the INT instruction is RETA(3:0) ← PC(3:0) +1, PC(3:0) ← 0000, and IPROC ← 1. When the INT instruction is executed, (1) the address of the next instruction PC(3:0) +1 must be stored in RETA(3:0); (2) the address 0000 must be stored in PC(3:0); and (3) a 1 must be stored in IPROC. Steps 2 and 3 may be expressed as PC(4:0) ← 10000, where IPROC represents PC(4) and is the 5th bit in the program counter. When when *B*1*B*0 = 01, the value 10000 is routed through the 4-to-1 MUX array and is stored in the loadable register circuit of the running program counter at the next rising edge of the signal *SPEED*. This causes the next instruction to be executed at the beginning of the extended instruction memory—that is, at address 10000 (16).

The purpose of the INT instruction is to cause VBC1-E to execute an interrupt service routine beginning at address 10000 in extended instruction memory (section 1) and to provide part of the return address to instruction memory (section 0) via the signal *RETA*(3:0). The generation of *RETA*(3:0) is covered in the next section.

The transfer function form for the IRET instruction is PC(3:0) ← RETA(3:0) and IPROC ← 0. When the IRET instruction is executed, (1) the part of the return address of the next instruction RETA(3:0) must be stored in PC(3:0), and (2) a 0 must be stored in IPROC. Steps 1 and 2 may be expressed as PC(3:0) ← RETA(3:0) and PC(4) ← 0, because IPROC represents PC(4) and is the 5th bit in the program counter. Steps 1 and 2, therefore, represent PC(4:0) ← 0 RETA(3:0). When when *B*1*B*0 = 10, 0 *RETA*(3:0) is routed through the 4-to-1 MUX array and is stored in the loadable register circuit of the running program counter at the next rising edge of the signal *SPEED*. This causes the next instruction to be executed at the address specified by 0 *RETA*(3:0) in instruction memory (section 0) for VBC1-E.

The purpose of the IRET instruction is to cause VBC1-E to execute the instruction following the INT instruction in instruction memory (section 0) via the signal 0 *RETA*(3:0).

Listing 24.2 shows a complete VHDL design for the loadable register circuit (LRC) in the RPC for VBC1-E.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity LRC is port (
   rst, speed, b1, b0, m6 : in std_logic;
   sum2 : in std_logic_vector(4 downto 0);
```

(Continued)

LISTING 24.2 Complete VHDL design for the loadable register circuit in the RPC for VBC1-E (project: LRC)

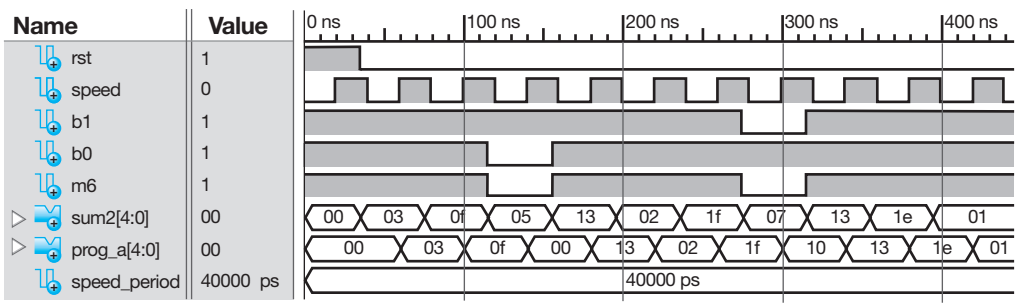
```
   prog_a : inout std_logic_vector (4 downto 0)
   );
end LRC;

architecture behavioral of LRC is
   signal s_clr,s_pre: std_logic;
begin
   s_clr <= (not prog_a(4) and prog_a(3) and prog_a(2) and
             prog_a(1) and prog_a(0)) and (not m6) and (not b0);
   s_pre <= prog_a(4) and prog_a(3) and prog_a(2) and
             prog_a(1) and prog_a(0) and (not m6) and (not b1);

process (rst, speed)
begin
   if rst = '1' then prog_a(4 downto 0) <= "00000";
   elsif rising_edge (speed) then
      if s_clr = '1' then prog_a(4 downto 0) <= "00000";
      elsif s_pre = '1' then prog_a(4 downto 0) <= "10000";
      else prog_a(4 downto 0) <= sum2(4 downto 0);
      end if;
   end if;
end process;
end behavioral;
```

Things that you should notice about the complete VHDL design in Listing 24.2:

- The AND gate code for *S_CLR* and *S_PRE* is generated via Boolean equations.
- The signals *S_CLR* and *S_PRE* are used as synchronous inputs to the loadable register circuit. This means that they can only cause the output to change after the rising edge of *SPEED*, so they must be placed after rising_edge (speed) in the VHDL code.
- The loadable register circuit output is changed to 00000 if the signal *S_CLR* is asserted—that is, is equal to 1.
- The loadable register circuit output is changed to 10000 if the signal *S_PRE* is asserted—that is, is equal to 1.
- The loadable register circuit stores the value of *SUM2* if *S_CLR* and *S_PRE* are not asserted—that is, not equal to 1—after the rising edge of *SPEED*.

Waveform 24.2 shows a simulation waveform diagram with the correct functionality of design entity LRC.

WAVEFORM 24.2
Simulation waveform diagram with the correct functionality of design entity LRC



Things you should notice about the waveforms in Waveform 24.2:

- All the vector signals are displayed in hexadecimal values.
- *RST* is set to 1 and then set back to 0 to clear the LRC.
- Notice that *PROG_A* follows *SUM2* at the next rising edge of *SPEED,* except when *PROG_A*(3:0) = 0F, *b*0 = 0, and *m*6 = 0, which causes the LRC to go to 00000 on the next rising edge of the signal *SPEED*, or when *PROG_A*(4:0) = 1F, *b*1 = 0, and *m*6 = 0, which causes the LRC to go to 10000 on the next rising edge of the signal *SPEED*.
- Waveform 24.2 shows that design entity LRC is functionally correct.

24.3 DESIGNING THE CIRCUIT TO STORE PCPLUS1 FOR VBC1-E

Figure 24.2 shows the store PCPLUS1 circuit for VBC1-E.

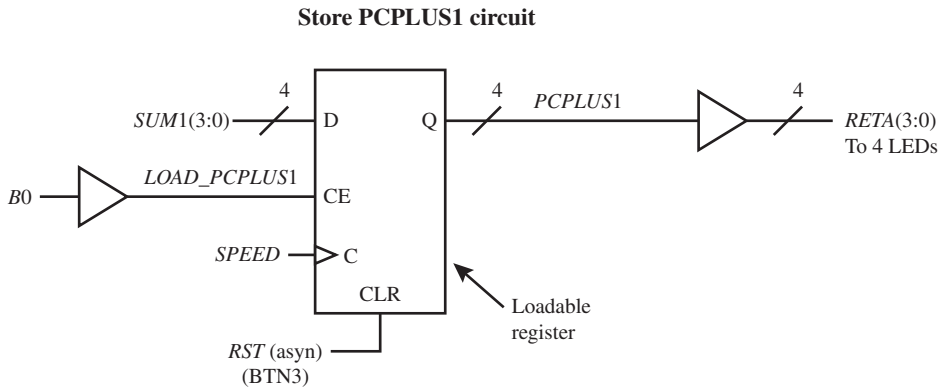


FIGURE 24.2 Store PCPLUS1 circuit for VBC1-E

Things that you should notice about the store PCPLUS1 circuit for VBC1-E in Figure 24.2:

- *SUM*1(3:0) is loaded into the loadable register at the next rising edge of *SPEED* when *B*0 is asserted—that is, is equal to 1. *SUM*1(3:0) contains the return address after the INT instruction is executed, which is the address of the instruction that follows the INT instruction. *B*0 is asserted only when the INT instruction is decoded.
- The final output of the circuit is the signal *RETA*(3:0), which is fed to four LEDs so that the value of the return address can be monitored.

Listing 24.3 shows a complete VHDL design for the store PCPLUS1 circuit for VBC1-E.

LISTING 24.3 Complete VHDL design for the store PCPLUS1 circuit for VBC1-E (project: Store_PCPLUS1_Circuit)

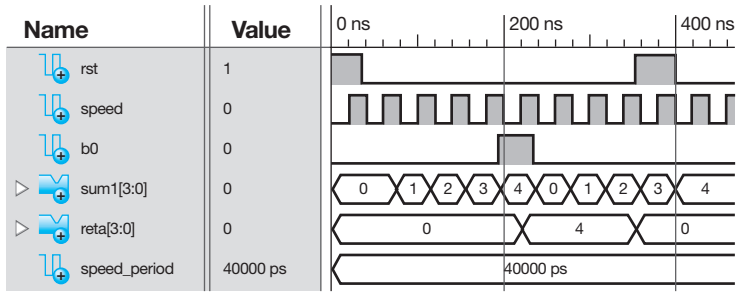
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Store_PCPLUS1_Circuit is port (
     rst, speed, b0 : in STD_LOGIC;
     sum1 : in STD_LOGIC_VECTOR (3 downto 0);
     reta : out STD_LOGIC_VECTOR (3 downto 0)
     );
end Store_PCPLUS1_Circuit;

architecture behavioral of Store_PCPLUS1_Circuit is
     signal load_pcplus1 : std_logic;
     signal pcplus1 : std_logic_vector (3 downto 0);
begin
     load_pcplus1 <= b0;
```

(Continued)

```
process (rst, speed)
begin
     if rst = '1' then pcplus1 <= "0000";
     elsif rising_edge (speed) and load_pcplus1 = '1' then
          pcplus1 <= sum1;
     end if;
end process;
     RETA <= pcplus1;
end behavioral;
```

Waveform 24.3 shows a simulation waveform diagram with the correct functionality of design entity Store_PCPLUS1_Circuit for VBC1-E.

WAVEFORM 24.3 Simulation waveform diagram with the correct functionality of design entity Store_PCPLUS1_Circuit



Things you should notice about the waveforms in Waveform 24.3:

- At the beginning of the simulation, when *RST* = 1, *RETA*(3:0) goes to 0000.
- When *RST* goes to 0, *RETA*(3:0) remains at 0000 at each rising edge of *SPEED* until *B*0 = 1, then *RETA*(3:0) = *SUM*1(3:0).
- When when *RST* = 1 close to the end of the simulation, observe that *RETA*(3:0) goes to 0000.
- Waveform 24.3 shows that design entity Store_PCPLUS1_Circuit is functionally correct.

Each time the INT instruction is executed, the single LED for IPROC turns on (see Figure 23.3 in Chapter 23). Each time the IRET instruction is executed, the single LED for IPROC turns off. Each time the INT instruction is executed, the four LEDs for RETA(3:0) change to indicate the return address in instruction memory (section 0). To manually turn off the single LED for IPROC and also turn off the four LEDs for RETA(3:0), simply press and release the reset push-button BTN3. Push-button BTN3 is used to reset VBC1-E to address 00000 in instruction memory (section 0), which forces the LEDs for IPROC and RETA to turn off.

24.4 INSTRUCTION DECODER TRUTH TABLES FOR THE INT AND IRET INSTRUCTIONS FOR VBC1-E

For the INT and IRET instructions to work properly, we must specify the instruction decoder truth tables to provide the control signals for the instructions INT and IRET so that the running program counter in the control unit will operate correctly. Figure 24.1 in Section 24.2 shows the control signals *B*1 and *B*0 that are required for the design of VBC1-E.

Table 24.1 shows the instruction decoder truth table for the INT instruction for VBC1-E

TABLE 24.1 Instruction decoder truth table for the INT instruction for VBC1-E

	IR								Control bit
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***B*0**
INT	1	0	1	0	0	1	0	0	1

Using the instruction decoder truth table, we can write the Boolean equation for the control bit for the INT instruction as

$B0 = 1$. The default value for $B0$ is 0.

Table 24.2 shows the instruction decoder truth table for the IRET instruction for VBC1-E

TABLE 24.2 Instruction decoder truth table for the IRET instruction for VBC1-E

	IR								Control bit
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	***B*1**
IRET	1	0	1	0	1	1	0	0	1

Using the instruction decoder truth table, we can write the Boolean equation for the control bit for the IRET instruction as

$B1 = 1$. The default value for $B1$ is 0.

24.5 DESIGNING THE INSTRUCTION DECODER FOR THE INT AND IRET INSTRUCTIONS FOR VBC1-E

In the VHDL design for the instruction decoder for the INT and IRET instructions, we will use Procedure ID, which is listed as follows: (1) use a process with a case statement to select each of the instructions by their OPCODE, (2) specify the default instruction decoder output values before the case statement, and (3) use Boolean equations for the control bits for each of the instructions within the case statement.

The OPCODE for the INT instruction is $IR(7:5) = 101$ and $IR(3:2) = 01$. The 8-bit sequence for the OPCODE bits for the INT instruction is 101V01VV, where V is placed in each bit position that has no OPCODE bit.

The OPCODE for the IRET instruction is $IR(7:5) = 101$ and $IR(3:2) = 11$. The 8-bit sequence for the OPCODE bits for the IRET instruction is 101V11VV, where V is placed in each bit position that has no OPCODE bit.

As obtained earlier, the Boolean equation for the INT instruction is

$B0 = 1$. The default value $B0$ is 0.

As obtained earlier, the Boolean equation for the IRET instruction is

$B1 = 1$. The default value of $B1$ is 0.

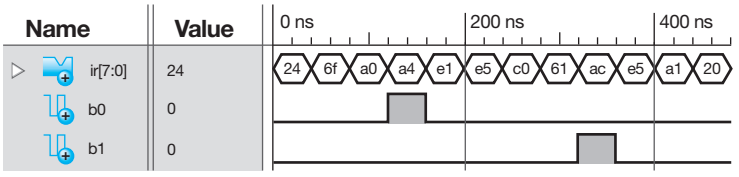
Listing 24.4 shows a complete VHDL design for the instruction decoder for the INT and IRET instructions using a process with a case statement.

LISTING 24.4 Complete VHDL design for the instruction decoder for the INT and IRET instructions (project: ID_Circuit)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ID_Circuit is port (
     ir : in STD_LOGIC_VECTOR (7 downto 0);
     b0, b1 : out STD_LOGIC
     );
end ID_Circuit;

architecture Behavioral of ID_Circuit is
begin
process (ir)
begin
     --default Instruction Decoder output values
     b0 <= '0' ; b1 <= '0';
     case ir (7 downto 5) is
          --provides the control signal b0 for the INT instruction
          when "101" => if ir(3 downto 2) = "01" then b0 <= '1';
          --provides the control signal b1 for the IRET instruction
                        elsif ir(3 downto 2) = "11" then b1 <= '1';
                        end if;
          when others => null;
     end case;
end process;
end Behavioral;
```

Waveform 24.4 shows a simulation waveform diagram with the correct functionality of design entity ID_Circuit for VBC1-E.

WAVEFORM 24.4 Simulation waveform diagram with the correct functionality of design entity ID_Circuit



Things you should notice about the waveforms in Waveform 24.4:

- The vector signal is displayed in hexadecimal values.
- The only time $B0$ goes to 1 is when when ir(7:0) = 10100100 (A4). For all others values, $B0 = 0$.
- The only time $B1$ goes to 1 is when when ir(7:0) = 10101100 (AC). For all other values, $B1 = 0$.
- Waveform 24.4 shows that design entity ID_Circuit provides the correct functionality for the INT and IRET instructions.

PROBLEMS

Section 24.2 Designing the Modified Circuit for the Running Program Counter and the Select Circuit for VBC1-E

24.1 In Figure 24.1, what type of signals are *B*1 and *B*0? What unit in the VBC1-E computer generates these signals?

24.2 In Figure 24.1, what is *SUM*2 when *B*1*B*0 is 10? Show the range of each signal.

24.3 In Figure 24.1, what is *SUM*2 when *B*1*B*0 is 00? Show the range of each signal.

24.4 In Figure 24.1, what is *SUM*2 when *B*1*B*0 is 01? Show the range of each signal.

24.5 In Figure 24.1, what is the & operator called? What is the expression *B*1 & *B*0 called in VHDL?

24.6 Show complete VHDL code for the select circuit in Figure 24.1. Use the VHDL representation for *SEL* = *B*1&*B*0 and a conditional signal assignment in the architecture. Show a simulation for your design to verify that your VHDL code is functionally correct. Name the design entity Select_Circuit1.

24.7 Show complete VHDL code for the select circuit in Figure 24.1. Use the VHDL representation for *SEL* = *B*1&*B*0 and a process with a case statement in the architecture. Show a simulation for your design to verify that your VHDL code is functionally correct. Name the design entity Select_Circuit2.

24.8 Show complete VHDL code for the select circuit in Figure 24.1. Use the VHDL representation for *SEL* = *B*1&*B*0 and a process with an if statement in the architecture. Show a simulation for your design to verify that your VHDL code is functionally correct. Name the design entity Select_Circuit3.

24.9 What is the purpose of the INT instruction?

24.10 What is the purpose of the IRET instruction?

24.11 For Figure 24.1, write the Boolean equation for the signal *S_CLR* and also the corresponding representation for the signal *S_CLR* in VHDL.

24.12 For Figure 24.1, write the Boolean equation for the signal *S_PRE* and also the corresponding representation for the signal *S_PRE* in VHDL.

24.13 What address in instruction memory is affected by the signal *S_CLR* in Figure 24.1?

24.14 Which instructions do not cause rollover from the last address in instruction memory (section 0)?

24.15 What address in instruction memory is affected by the signal *S_PRE* in Figure 24.1?

24.16 Which instructions do not cause rollover from the last address in extended instruction memory (section 1)?

24.17 Listing P24.17 shows the VHDL code for the architecture for the loadable register circuit in Figure 24.1. Explain how the signal *S_CLR* is used as a synchronous input to the loadable register circuit.

24.18 In Listing P24.17, explain how the signal *S_PRE* is used as a synchronous input to the loadable register circuit.

24.19 In Listing P24.17, explain how the signal *RST* is used as an asynchronous input to the loadable register circuit.

LISTING P24.17

```
architecture behavioral of LRC is
   signal s_clr,s_pre: std_logic;
begin
   s_clr <= (not prog_a(4) and prog_a(3) and prog_a(2)
             and prog_a(1) and prog_a(0)) and (not m6) and (not b0);

   s_pre <= prog_a(4) and prog_a(3) and prog_a(2)
             and prog_a(1) and prog_a(0) and (not m6) and (not b1);

process (rst, speed) --process for loadable register circuit
begin
   if rst = '1' then prog_a(4 downto 0) <= "00000";
   elsif rising_edge (speed) then
      if s_clr = '1' then prog_a(4 downto 0) <= "00000";
      elsif s_pre = '1' then prog_a(4 downto 0) <= "10000";
      else prog_a(4 downto 0) <= sum2(4 downto 0);
      end if;
   end if;
end process;
end behavioral;
```

24.20 Listing P24.20 shows the VHDL code for the output of the AND gate in Figure 24.1. Show a complete VHDL design for this output to investigate if it will or will not compile correctly. Explain your finding.

```
S_CLR <= (NOT PROG_A(4)) AND
PROG_A(3 DOWNTO 0) AND (NOT M6)
AND (NOT B0);
```

LISTING P24.20

Section 24.3 Designing the Circuit to Store PCPLUS1 for VBC1-E

24.21 In Figure 24.2, what instruction must be executed to store *SUM*1(3:0)?

24.22 In Figure 24.2, after the instruction INT is executed, where is the return address stored?

24.23 Explain how the return address for the INT instruction can be monitored by the circuit in Figure 24.2. What is the name of the return address signal called?

24.24 Show complete VHDL code for the store PCPLUS1 circuit in Figure 24.2. Use Boolean equations and a conditional signal assignment in the architecture. Show a simulation for your design to verify that your VHDL code is functionally correct. Name the design entity Store_PCPLUS1.

24.25 List the instruction that turns on the single LED for IPROC.

24.26 List the instruction that turns off the single LED for IPROC.

24.27 List the instruction that turns on the four LEDs for RETA.

24.28 Is there an instruction that can be executed to turn off the four LEDs for RETA?

24.29 When the reset push-button BTN3 is pressed and released, does this turn the LEDs for IPROC and RETA on or off?

Section 24.4 Instruction Decoder Truth Tables for the INT and IRET Instructions for VBC1-E

24.30 Table P24.30a shows the original instruction decoder truth table for the INT instruction for VBC1-E. Table P24.30b shows a design modification to the instruction decoder truth table for the INT instruction for VBC1-E. Show or list the required circuit changes to Figure 24.1 and Figure 24.2 for the design modification.

24.31 For the modified instruction decoder truth table for the INT instruction in Table P24.30b, is it correct to specify the Boolean equation for the control bit as $B0 = \overline{IR(2)}$? Provide a different way to write the Boolean equation for the control bit.

24.32 Table P24.32a shows the original instruction decoder truth table for the IRET instruction for VBC1-E. Table P24.32b shows a design modification to the instruction decoder truth table for the IRET instruction for VBC1-E. Show or list the required circuit changes to Figure 24.1 for the design modification.

24.33 For the modified instruction decoder truth table for the IRET instruction in Table P24.32b, is it correct to specify the Boolean equation for the control bit as $B1 = \overline{IR(7)}$? Provide a different way to write the Boolean equation for the control bit.

24.34 In Table P24.30b, what is the default value for the control signal *B*0 in the modified instruction decoder truth table for the INT instruction for the case statement in VHDL?

24.35 In Table P24.32b, what is the default value for the control signal *B*1 in the modified instruction decoder truth table for the IRET instruction for the case statement in VHDL?

Section 24.5 Designing the Instruction Decoder for the INT and IRET Instructions for VBC1-E

24.36 List the procedure used for designing the instruction decoder for the INT and IRET instructions.

	IR 7 6 5 4 3 2 1 0	Control bit *B*0	
INT	1 0 1 0 0 1 0 0	1	← Original

(a)

	IR 7 6 5 4 3 2 1 0	Control bit *B*0	
INT	1 0 1 0 0 1 0 1	0	← Modified

(b)

TABLE P24.30 (a) Original instruction decoder truth table for the INT instruction; (b) design modification to the instruction decoder truth table for the INT instruction

	IR 7 6 5 4 3 2 1 0	Control bit *B*1	
IRET	1 0 1 0 1 1 0 0	1	← Original

(a)

	IR 7 6 5 4 3 2 1 0	Control bit *B*1	
IRET	1 0 1 0 1 1 0 0	0	← Modified

(b)

TABLE P24.32 (a) Original instruction decoder truth table for the IRET instruction; (b) design modification to the instruction decoder truth table for the IRET instruction

24.37 What is the purpose of including the default instruction decoder output values when designing the instruction decoder?

24.38 Write the OPCODE for the INT instruction.

24.39 Write the OPCODE for the IRET instruction.

24.40 Show a complete VHDL design for the instruction decoder required for Figure 24.1 for just the INT instruction using a conditional signal assignment. Name the design entity ID_circuit_INT. Show a simulation for the VHDL code to verify that it works. Show the simulation waveform using hexadecimal numbers.

24.41 Show a complete VHDL design for the instruction decoder required for Figure 24.1 for just the IRET instruction using a conditional signal assignment. Name the design entity ID_circuit_IRET. Show a simulation for the VHDL code to verify that it works. Show the simulation waveform using hexadecimal numbers.

24.42 Show a complete VHDL design for the instruction decoder required for Figure 24.1 for the INT and IRET instructions using conditional signal assignments. Name the design entity ID_ckt. Show a simulation for the VHDL code to verify that it works. Show the simulation waveform using hexadecimal numbers.

25 Chapter

Completing the Design for VBC1-E

Chapter Outline

25.1 INTRODUCTION

This is the seventh and final chapter in the series of chapters that teaches you how to expand the design of VBC1 to make VBC1-E. In this chapter, you will learn how to design circuits that provide a hardware interrupt for VBC1-E. You will also learn how to modify existing circuits and add some new circuits for displaying the signal *RETA*. With this additional circuitry, you can load an interrupt service routine in extended instruction memory (section 1), access the extended instruction memory via a manual push-button switch, execute the interrupt service routine, and return to instruction memory (section 0) via the IRET instruction.

25.2 DESIGNING A DEBOUNCED ONE-PULSE TRIGGER INTERRUPT CIRCUIT AND MODIFYING THE RPC CIRCUIT FOR VBC1-E

Figure 25.1 shows a debounced one-pulse trigger interrupt circuit and the modified RPC circuit for VBC1-E.

Things you should notice about the debounced one-pulse trigger interrupt circuit and modified RPC circuit for VBC1-E in Figure 25.1:

- Push-button switch BTN3, on the button module, is used as the input to the debounced one-pulse trigger interrupt circuit via the D input of the *QA* flip-flop.
- This debounced circuit was previously used to generate a single pulse for the loading program counter. In this case, it is used to generate a single pulse to trigger a hardware interrupt.
- Each time push-button BTN3 is pressed on the button module, a hardware interrupt will occur when a program is running in instruction memory (section 0). The signal *TID* (trig-

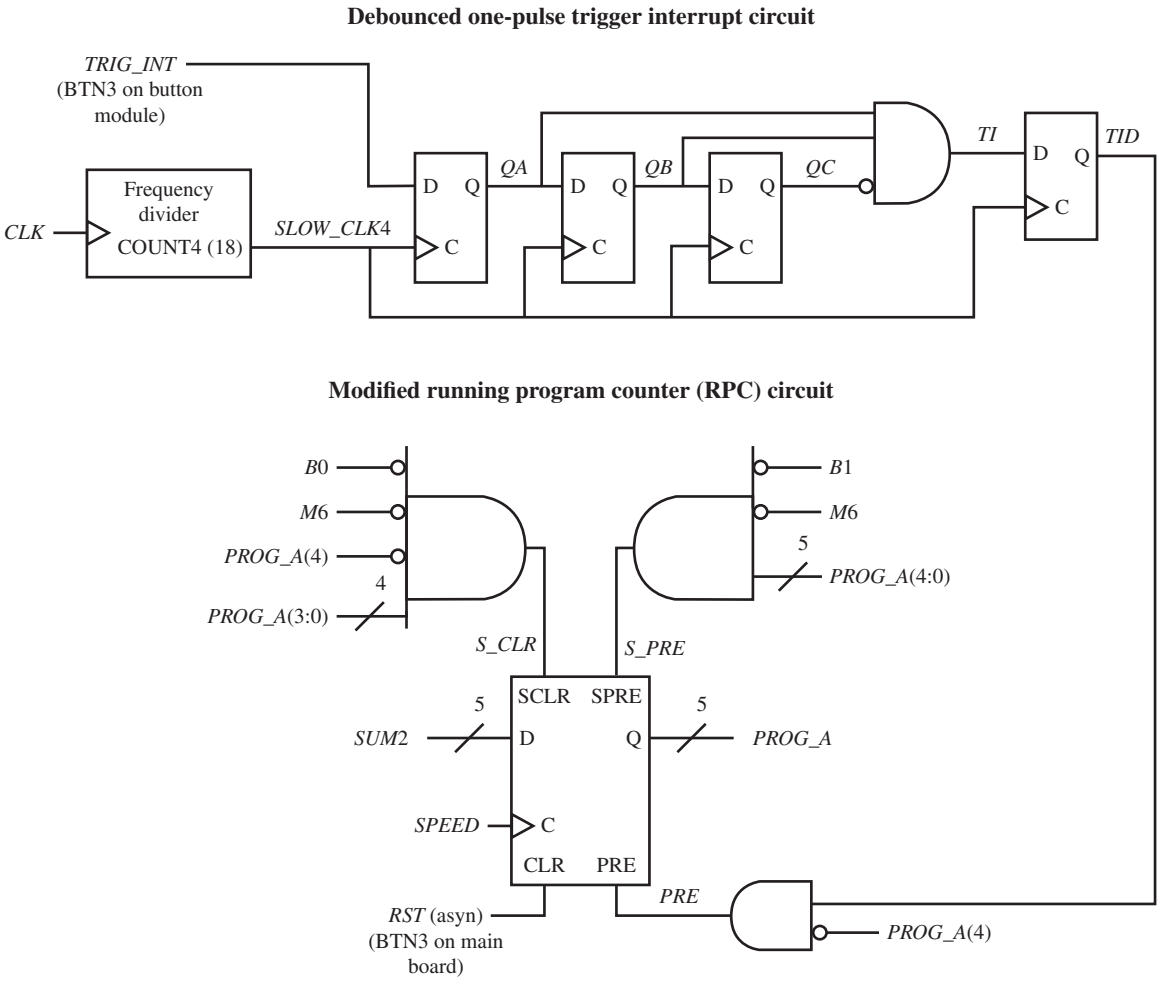


FIGURE 25.1 Debounced one-pulse trigger interrupt circuit and modified RPC circuit for VBC1-E

ger interrupt delayed) and the inverted signal *PROG_A*(4) are connected to an AND gate. When the signal *TID* is 1 and the signal *PROG_A*(4) is 0, the signal *PRE* is 1. The signal *PRE* is connected to the PRE input of the loadable register circuit of the RPC. The signal *PRE* is used as an asynchronous preset for the loadable register circuit to preset the RPC to the address 10000 (16), which is the first address of expanded instruction memory—that is, the starting address of the interrupt service routine.

- The flip-flop for storing the signal *TID* is used later so that a gated clock circuit is not created by connecting the signal *TI* directly to the control input (or clock input) of a flip-flop to store the value of *RETA* in Figure 25.2 (see store PC circuit) in the next section. Recall: If the signal *TI* is used to drive the clock input of a flip-flop, this is considered a bad design practice, and a gated clock warning will be issued by the ISE software. The output of a flip-flop driving the clock input of another flip-flop is not a gated clock, because the output of the flip-flop is synchronized with the clock and does not create glitches.

Listing 25.1 shows a complete VHDL design for the debounced one-pulse trigger interrupt circuit for VBC1-E in Figure 25.1 with the signal *COUNT*4(18) changed to *COUNT*4(1).

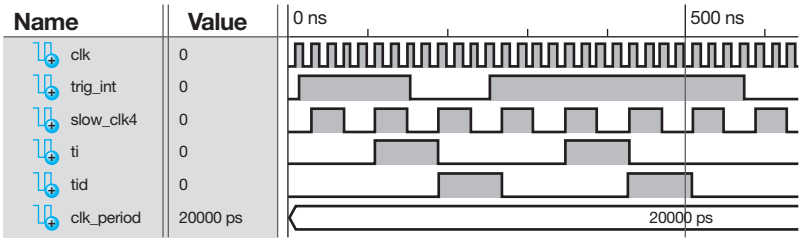
LISTING 25.1
Complete VHDL design for the debounced one-pulse trigger interrupt circuit for VBC1-E (project: Debounce_Circuit)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Debounced_Circuit is port (
     clk : in STD_LOGIC;
     trig_int : in STD_LOGIC;
     slow_clk4 : inout STD_LOGIC;
     ti : inout STD_LOGIC;
     tid : inout STD_LOGIC := '0'
     );
end Debounced_Circuit;

architecture dataflow of Debounced_Circuit is
     signal qa, qb, qc : std_logic := '0';
     signal count4 : std_logic_vector (1 downto 0) :="00";
begin
--frequency divider
     count4 <= count4 + 1 when rising_edge(clk) else
               count4;
     slow_clk4 <= count4(1);

--4 flip-flops
      qa <= trig_int when rising_edge (slow_clk4);
      qb <= qa when rising_edge (slow_clk4);
      qc <= qb when rising_edge (slow_clk4);
     tid <= ti when rising_edge (slow_clk4);
--AND gate
     ti <= qa and qb and not qc;
end dataflow ;
```

Things that you should notice about the complete VHDL design in Listing 25.1:

- The **use** IEEE.STD_LOGIC_UNSIGNED.**ALL**; entry is required in the library part because the arithmetic operator "+" is used for the frequency divider.
- The arithmetic method is use to obtain the VHDL code for the frequency divider.
- The signals *SLOW_CLK*4 and *TI* are used in the entity and not as internal signals so that these signals will be displayed in the simulation waveform diagram for design entity Debounced_Circuit.
- All flip-flops are initialized to 0 via := '0' for flip-flop outputs *TID*, *QA*, *QB*, and *QC* including :="00" for flip-flop outputs *COUNT*4(1:0). Initialization is only required when running a simulation.
- The frequency of the signal *COUNT*4(18) is too slow compared to the frequency of the signal *CLK* to provide an observable simulation output for *SLOW_CLK*4. To show a simulation output that provides a *proof of concept* of the circuit, the signal *COUNT*4(18) was changed to *COUNT*4(1). *COUNT*4(1) divides the frequency *CLK* by 2^2 (or 4) while *COUNT*4(18) divides the frequency of *CLK* by 2^{19} (or 524,288), which would be stretched out too far to observe via a simulation waveform diagram.
- Four separate conditional signal assignments are used for each of the four flip-flops *QA*, *QB*, *QC*, and *TID*.
- A Boolean equation is used for the AND gate.

Waveform 25.1 shows the correct functionality of design entity Debounced_Circuit.



WAVEFORM 25.1 Simulation for the correct functionality of design entity Debounced_Circuit

Things you should notice about the waveforms in Waveform 25.1:

- *SLOW_CLK*4 is 1/4 the frequency of *CLK*.
- When *TRIG_INT* is applied to cause a hardware interrupt, via push-button switch BTN3 on the button module, a single pulse *TI* followed by a delayed pulse *TID* is generated by the debounced one-pulse trigger interrupt circuit shown in Figure 25.1.
- Observe that *TRIG_INT* is applied twice in Waveform 25.1. The first time BTN3 is pressed briefly, and the second time it is held longer before it is released. In both cases, a single pulse *TI* is generated followed by a delayed pulse *TID*.
- The pulse *TID* is delayed by one clock cycle of the signal *SLOW_CLK*4, which is the clock signal that drives the *TID* flip-flop.
- Waveform 25.1 shows that design entity Debounced_Circuit is functionally correct.

The transfer function for the hardware interrupt is *RETA*(3:0) ← PC(3:0), PC(3:0) ← 0000, IPROC ← 1. The modified RPC circuit for VBC1-E only performs the parts PC(3:0) ← 0000 and IPROC ← 1, which is the same as PC(4:0) ← 10000. The signal *PRE* is used as an asynchronous preset input to the loadable register circuit of the RPC.

Listing 25.2 shows a complete VHDL design for the modified RPC circuit for VBC1-E.

LISTING 25.2 Complete VHDL design for the modified RPC circuit for VBC1-E (project: RPC_Circuit_MOD)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RPC_Circuit_MOD is port (
   rst, speed, b1, b0, m6, tid : in std_logic;
   sum2 : in std_logic_vector(4 downto 0);
   prog_a : inout std_logic_vector (4 downto 0)
   );
end RPC_Circuit_MOD;

architecture Mixed of RPC_Circuit_MOD is
   signal s_clr,s_pre,pre: std_logic;
begin
   s_clr <= (not prog_a(4) and prog_a(3) and prog_a(2)
             and prog_a(1) and prog_a(0)) and (not m6) and (not b0);

   s_pre <= prog_a(4) and prog_a(3) and prog_a(2)
            and prog_a(1) and prog_a(0) and (not m6) and (not b1);

   pre <= not prog_a(4) and tid;
process (rst, speed, pre)
begin
   if rst = '1' then prog_a(4 downto 0) <= "00000";
   elsif pre = '1' then prog_a(4 downto 0) <= "10000";
                                                    (Continued)
```

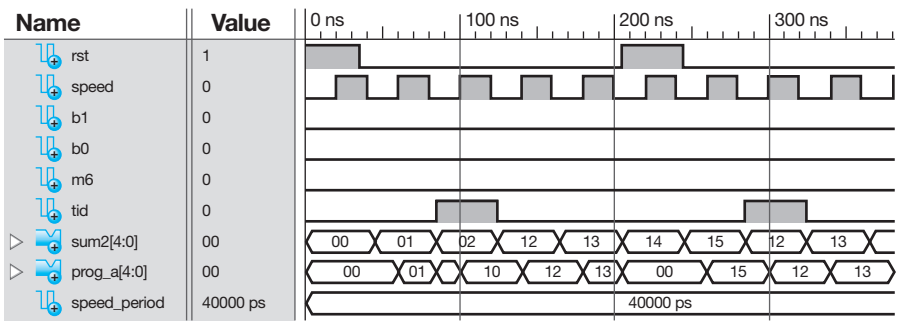
```
    elsif rising_edge (speed) then
       if s_clr = '1' then prog_a(4 downto 0) <= "00000";
       elsif s_pre = '1' then prog_a(4 downto 0) <= "10000";
       else prog_a(4 downto 0) <= sum2(4 downto 0);
       end if;
    end if;
end process;
end Mixed;
```

Things that you should notice about the complete VHDL design for the modified RPC Circuit in Listing 25.2:

- The signal *TID* is included in the entity.
- The VHDL code is written using a mixed design style—that is, Boolean equations are used and a process with an if statement is used, which represents two different design styles. The architecture is named Mixed to reflect the two different design styles.
- The signal *PRE* is an internal signal, which requires it to be included in the **architecture** before the first **begin**.
- The Boolean equation for the signal *PRE* is a concurrent assignment that is included after the first **begin** but before the **process**.
- The signal *PRE* must be included in the process before the rising_edge (speed) to make it an asynchronous signal. If the signal *PRE* is true or 1, then prog_a(4 **downto** 0) is set to 10000.

Waveform 25.2 shows a simulation waveform diagram with the correct functionality of design entity RPC_Circuit_MOD.

WAVEFORM 25.2
Simulation waveform diagram with the correct functionality of design entity RPC_Circuit_MOD



Things you should notice about the waveforms in Waveform 25.2:

- Notice that *PROG_A* always follows *SUM*2 at the next rising edge of *SPEED*, except when the signal *TID* is 1 while executing instructions in instruction memory (section 0). When the signal *TID* is 1, this represents a hardware interrupt—that is, the push-button BTN3, on the button module, has been pressed to generate the hardware interrupt. When a hardware interrupt occurs, the loadable register circuit in the RPC asynchronously presets the RPC to the valued 10000 as shown in Waveform 25.2 after the first *TID* pulse so that the interrupt service routine can be executed in extended instruction memory (section 1).
- Observe that the RPC is cleared to 00000—that is, *PROG_A*(4:0) goes to 00000—after the second reset pulse is provided by the signal *RST*.
- When VBC1-E is executing instructions in extended instruction memory (section 1), pressing push-button BTN3 will not cause a hardware interrupt because *PROG_A*(4) is 1 as shown Waveform 25.2 after the second *TID* pulse.
- Waveform 25.2 shows that design entity RPC_Circuit_MOD is functionally correct.

25.3 DESIGNING CIRCUITS FOR DISPLAYING THE SIGNAL *RETA* FOR VBC1-E

Figure 25.2 shows the circuits for displaying the signal *RETA* (return address) for VBC1-E.

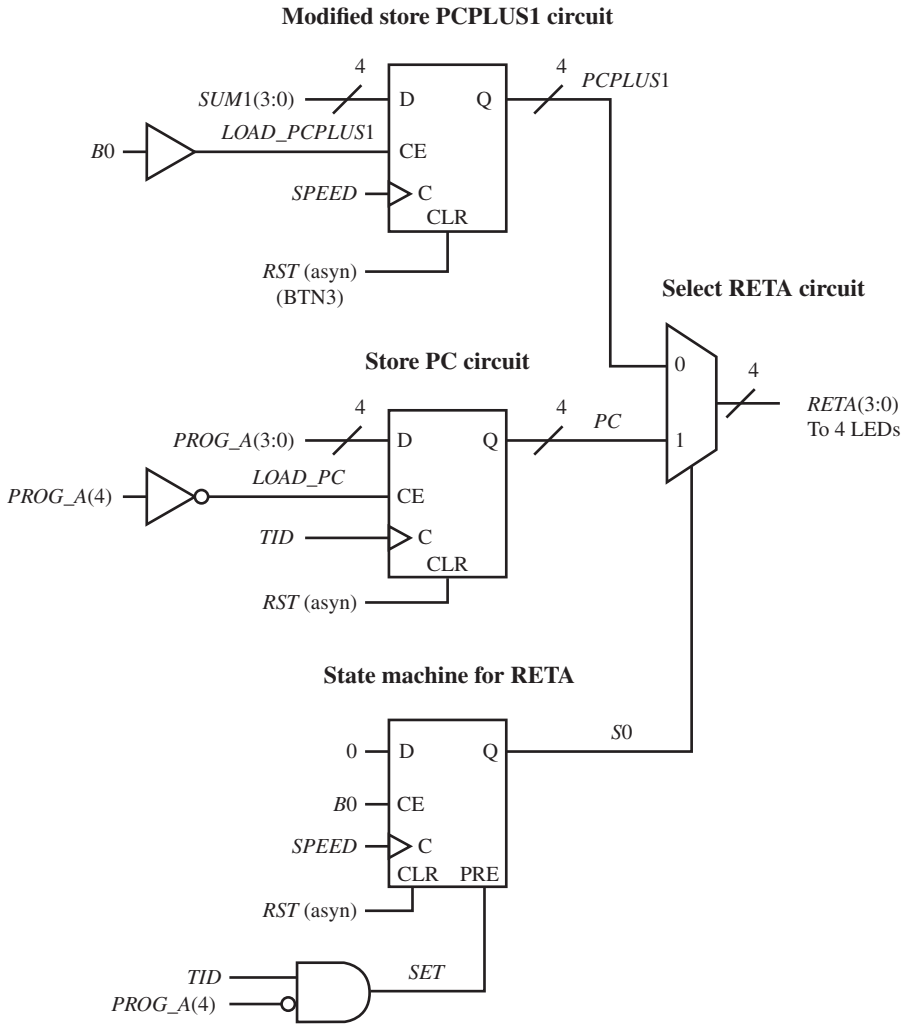


FIGURE 25.2 Circuits for displaying the signal *RETA* (return address) for VBC1-E

Things you should notice about the circuit for displaying the signal *RETA* for VBC1-E in Figure 25.2:

- The select RETA circuit steers the signal *PCPLUS*1 to the output signal *RETA* when the INT instruction is executed so that the signal *PCPLUS*1 is used as the return address when the IRET instruction is executed.
- The store PC circuit in Figure 25.2 stores the current value of the signal *PC*—that is, signal *PROG_A*(3:0) when a hardware interrupt is caused by pressing push-button BTN3 on the button module. The select RETA circuit steers the signal *PC* to the output signal *RETA*, which satisfies the RETA(3:0) ← PC(3:0) part of the transfer function for a hardware interrupt.
- The select input to the 2-to-1 MUX for the select RETA circuit is provided by the state machine for RETA. When the output signal *S*0 of the state machine is 0, the signal *RETA* follows the signal *PCPLUS1*, which provides the correct return address for the INT

instruction. When the output signal *S*0 of the state machine is 1, the signal *RETA* follows the signal *PC*, which provides the correct return address for a hardware interrupt.

- When the state machine for RETA provides a 0 output, *RETA* displays the return address for a software interrupt via four LEDs as shown in Figure 25.2.
- Notice that the store PC circuit can only store *PROG_A*(3:0) when *LOAD_PC* is 1 or *PROG_A*(4) is 0, and the state machine for RETA can only provide a 1 at its output when *TID* is 1 and *PROG_A*(4) is 0. When the state machine for RETA provides a 1 output, *RETA* displays the return address for a hardware interrupt via four LEDs.

Listing 25.3 shows a complete VHDL design for the store PC circuit for VBC1-E.

LISTING 25.3
Complete VHDL design for the store PC circuit for VBC1-E (project: Store_PC_Circuit)

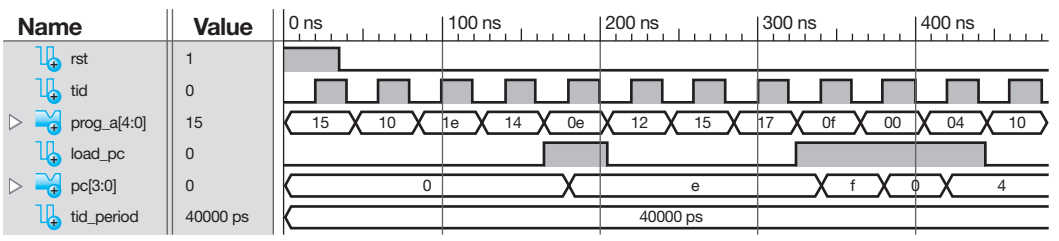
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Store_PC_Circuit is port (
     rst, tid : in STD_LOGIC;
     prog_a : in STD_LOGIC_VECTOR (4 downto 0);
     load_pc : inout STD_LOGIC;
     pc : out STD_LOGIC_VECTOR (3 downto 0)
     );
end Store_PC_Circuit;

architecture Mixed of Store_PC_Circuit is
begin
     load_pc <= not prog_a(4);
process(rst, tid)
begin
     if rst = '1' then PC <= "0000";
     elsif rising_edge (tid) and load_pc = '1' then
          pc <= prog_a(3 downto 0);
     end if;
end process;
end Mixed;
```

Things that you should notice about the complete VHDL design for the store PC circuit for VBC1-E in Listing 25.3:

- The VHDL code is written using a mixed design style—that is, a Boolean equation is used and a process with an if statement is used, which represents two different design styles.
- Signal *LOAD_PC* is used in the entity and not as an internal signal so that the signal *LOAD_PC* will be displayed in the simulation waveform diagram for design entity Store_PC_circuit.

Waveform 25.3 shows a simulation waveform diagram with the correct functionality of design entity Store_PC_Circuit.

WAVEFORM 25.3
Simulation waveform diagram with the correct functionality of design entity Store_PC_Circuit



Things you should notice about the waveforms in Waveform 25.3:

- Observe that the only times that the signal *PC* stores the signal *PROG_A*(3:0) is when the CE (clock enable) input to the loadable register is asserted or equal to 1. This only occurs when *PROG_A*(4) is 0, which results in the signal *LOAD_PC* = 1, because *LOAD_PC* = $\overline{PROG\_A(0)}$. This occurs four times in Waveform 25.3. The first time occurs when *PROG_A*(4:0) is 0E (0 1110) and *PC*(3:0) stores E (1110). The second time occurs when *PROG_A*(4:0) is 0F (0 1111) and *PC*(3:0) stores F (1111). The third time occurs when *PROG_A*(4:0) is 00 (0 0000) and *PC*(3:0) stores 0 (0000). The fourth time occurs when *PROG_A*(4:0) is 04 (0 0100) and *PC*(3:0) stores 4 (0100).
- Notice when the signal *PROG_A*(4) is 1 and the signal *LOAD_PC* is 0, the signal *PC* does not store the signal *PROG_A*(3:0).
- Waveform 25.3 shows that design entity Store_PC_Circuit is functionally correct.

The state machine for RETA selects the value of *RETA*—that is, the return address for the INT instruction (a software interrupt) or the return address for a hardware interrupt when push-button BTN3 on the button module is manually pressed. To select the return address for the INT instruction in the select RETA circuit, the select input to the 2-to-1 MUX must be 0. To select the return address for a hardware interrupt for the select RETA circuit, the select input to the 2-to-1 MUX must be 1. The output signal *S*0 of the state machine for RETA supplies the signal for the select input to the 2-to-1 MUX.

Figure 25.3 shows the state diagram for the state machine for RETA.

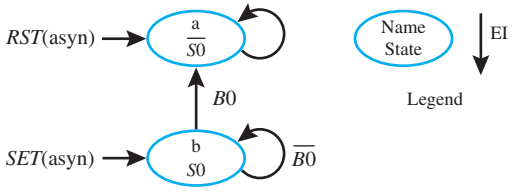


FIGURE 25.3 State diagram for the state machine for RETA

In Figure 25.3, when the input signal *RST* is 1 the state machine is cleared, so the output signal *S*0 is 0. When the code for VBC1-E is initially downloaded, the state machine is also cleared, so the output signal *S*0 is 0. The output signal *S*0 remains 0 until a hardware interrupt occurs, which sets the state machine, so the output signal *S*0 is 1. The output signal *S*0 remains 1 until a software interrupt occurs or until the state machine is cleared by the signal *RST*.

The Set OR Hold 1 equation can be used to write the D execution equation for the state diagram in Figure 25.3. The D execution equation is $D = S0 \cdot \overline{B0}$.

The state machine for RETA can also be preset by the signal *SET*, when a hardware interrupt occurs that is caused by $TID \cdot \overline{PROG\_A(4)}$, so $SET = TID \cdot \overline{PROG\_A(4)}$. These relationships can be used to draw a circuit for the state machine for RETA as shown in Figure 25.4.

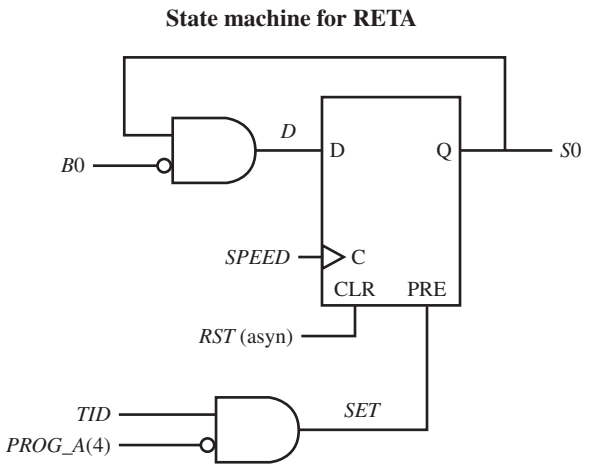


FIGURE 25.4 Circuit for the state machine for RETA

The state machine for RETA shown in Figure 25.2 is equivalent to the circuit shown in Figure 25.4. Notice that the state machine for RETA in Figure 25.2 has a clock enable (CE) input, while the circuit in Figure 25.4 does not have a CE input.

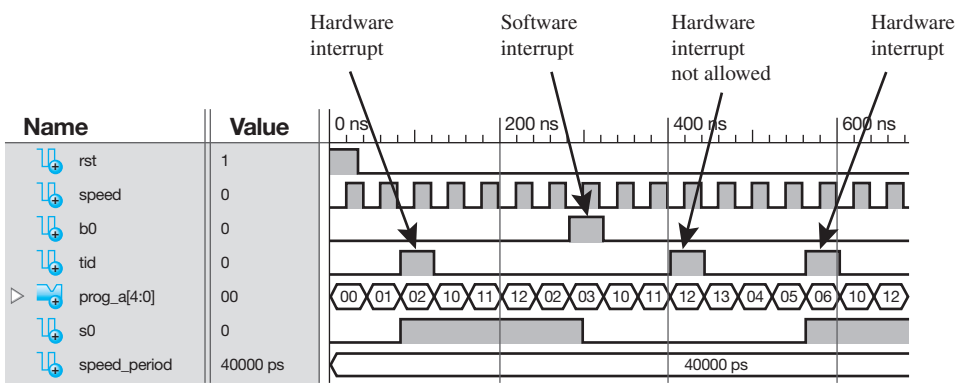
Listing 25.4 shows a complete VHDL design for the state machine for RETA in Figure 25.2 for VBC1-E.

LISTING 25.4
Complete VHDL design for the state machine for RETA in Figure 25.2 for VBC1-E (project: SMFR_with_CE)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SMFR_with_CE is port (
     rst, speed, b0, tid : in STD_LOGIC;
     prog_a : in STD_LOGIC_VECTOR (4 downto 0);
     s0 : out STD_LOGIC
     );
end SMFR_with_CE;

architecture Mixed of SMFR_with_CE is
     signal set :std_logic;
begin
     set <= tid and not prog_a(4);
process (rst, set, speed)
begin
     if rst = '1' then s0 <= '0';
     elsif set = '1' then s0 <= '1';
     elsif rising_edge (speed) and b0 = '1' then s0 <= '0';
     end if;
end process;
end Mixed;
```

Waveform 25.4 shows a simulation waveform diagram with the correct functionality of design entity SMFR_with_CE for VBC1-E.

WAVEFORM 25.4
Simulation waveform diagram with the correct functionality of design entity SMFR_with_CE for VBC1-E



Things you should notice about the waveforms in Waveform 25.4:

- The *RST* pulse sets *S*0 to 0.
- A hardware interrupt is initiated when a *TID* pulse is generated.
- Observe that the state machine for RETA only responds to a *TID* pulse when *PROG_A*(4) is 0, which sets *S*0 to 1. *S*0 remains 1 until the software interrupt occurs as shown in Waveform 25.4, which loads 0 into the state machine for RETA, so *S*0 goes to 0.

- A hardware interrupt is initiated by another *TID* pulse, but this does not cause a hardware interrupt because *PROG_A*(4) is 1.
- When a hardware interrupt is initiated by another *TID* pulse and *PROG_A*(4) is 0, state machine for RETA sets *S*0 to 1.
- Waveform 25.4 shows that design entity SMFR_with_CE is functionally correct.

25.4 DESIGNING CIRCUITS TO PROVIDE A LOADER FOR INSTRUCTION MEMORY FOR VBC1-E

Experiment 25L is an experiment designed to provide the capability to load a program into the total instruction memory of VBC1-E via a file created by the Save button in EASY1-E. Additional circuitry must be added to VBC1-E to create VBC1-EL. After VBC1-EL is downloaded into the FPGA on a BASYS 2 board or on a NEXYS 2 board, the machine code for a program can be loaded into the total instruction memory via the USB connector on the board. Experiment 25L is self-contained and shows the additional circuitry necessary to design VBC1-EL.

In order to load the total instruction memory contents via the USB connector, a computer software program is required. From this textbook's website download the installer for the VBC1-L (VBC1-EL) Memory Loader to your computer, and install the software.

The machine code file that is generated by EASY1-E can be entered into the VBC1-L (VBC1-EL) Memory Loader. Then, the machine code can be automatically loaded into the instruction memory and the extended instruction memory of VBC1-EL by clicking on the Load Memory button. For more information, see Appendix E, Section E.3, Loading Memory via the Memory Loader Program.

PROBLEMS

Section 25.2 Designing a Debounced One-Pulse Trigger Interrupt Circuit and Modifying the RPC Circuit for VBC1-E

25.1 In Figure 25.1, what manually operated push-button switch is used to generate the signal *TRIG_INT*?

25.2 List the values of the two signals that cause the signal *PRE* to preset the RPC to the address 10000.

25.3 Is the signal *PRE* used as a synchronous signal or an asynchronous signal in the RPC?

25.4 What is the first address of the extended instruction memory (section 1) in binary, hexadecimal, and decimal?

25.5 Figure P25.5 shows a PS/NS table for a complex state machine. *RST* is an asynchronous reset input and *S* is an external input. Show the D excitation equations and the output equation *OP* for the state machine, and draw a circuit diagram for the circuit using D flip-flops. What is the name of the circuit? (Hint: The equations can be written by inspection.)

25.6 Complete the state diagram shown in Figure P25.6 on the next page that is equivalent to the PS/NS table in Figure P25.5. Label the transition input signal $\overline{S}$ for $S = 0$ and S for $S = 1$. Show the D excitation equations and the output equation. (Hint: To check your state diagram, write the D excitation equations and the output equation for your state diagram. The equations should be identical to those obtained via the PS/NS table in Figure P25.5).

25.7 Write the transfer function form for a hardware interrupt for VBC1-E.

25.8 Write an equivalent expression for PC(3:0) ← 0000, IPROC ← 1.

RST	*S*	*Q*1	*Q*2	*Q*3	$Q1^+$	$Q2^+$	$Q3^+$	*OP*
1	X	X	X	X	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	1	0	0	0	1	0	0
0	0	1	0	1	0	1	0	0
0	0	1	1	0	0	1	1	1
0	0	1	1	1	0	1	1	0
0	1	0	0	0	1	0	0	0
0	1	0	0	1	1	0	0	0
0	1	0	1	0	1	0	1	0
0	1	0	1	1	1	0	1	0
0	1	1	0	0	1	1	0	0
0	1	1	0	1	1	1	0	0
0	1	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1	0

FIGURE P25.5

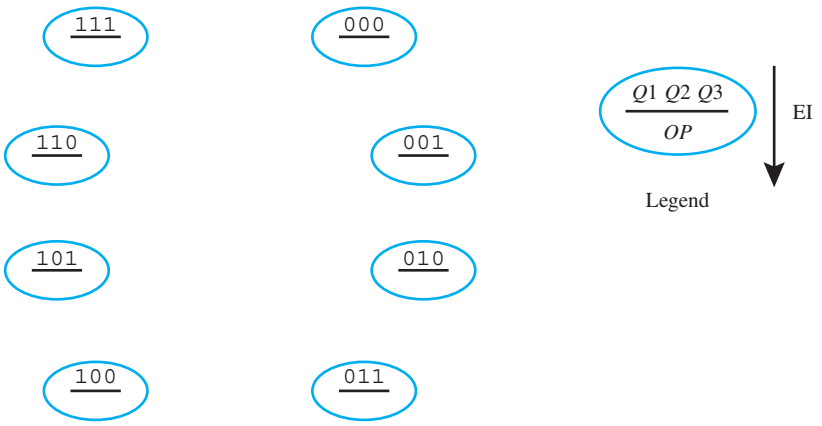


FIGURE P25.6

25.9 Listing P25.9 shows a process for the running program counter in Figure P25.1. Why is the VHDL statement **elsif** pre = '1' **then** prog_a(4 **downto** 0) <= "10000"; placed before the rising_edge (speed)?

```
process (rst, speed, pre)
begin
     if rst = '1' then prog_a(4 downto
              0) <= "00000";
     elsif pre = '1' then prog_a(4
              downto 0) <= "10000";
     elsif rising_edge (speed) then
          if s_clr = '1' then prog_a(4
              downto 0) <= "00000";
          elsif s_pre = '1' then
              prog_a(4 downto 0) <=
              "10000";
          else prog_a(4 downto 0) <=
              sum2(4 downto 0);
          end if;
     end if;
end process;
```

LISTING P25.9

25.10 In Waveform 25.2 for the design entity RPC_Circuit_MOD, does the first trigger interrupt signal *TID* in the waveform diagram cause a hardware interrupt? Give an explanation for your answer.

25.11 In Waveform 25.2 for the design entity RPC_Circuit_MOD, does the second trigger interrupt signal *TID* in the waveform diagram cause a hardware interrupt? Give an explanation for your answer.

Section 25.3 Designing Circuits for Displaying the Signal *RETA* for VBC1-E

25.12 Show complete VHDL code for the store PC circuit in Figure 25.2. Use a Boolean equation for the signal *LOAD_PC* and a conditional signal assignment for the D flip-flop array in the architecture. Show a simulation for your design to verify that your VHDL code is functionally correct. Use signal *LOAD_PC* as an internal signal so it does not show up in the simulation. Name the design entity Store_PC_Circuit_CSA.

25.13 Show complete VHDL code for the state machine for RETA in Figure P25.13. Use Boolean equations for the signals *D* and *SET*, and use a conditional signal assignment for the D flip-flop in the architecture. Show a simulation for your design to verify that your VHDL code is functionally correct. Name the design entity SM_for_RETA.

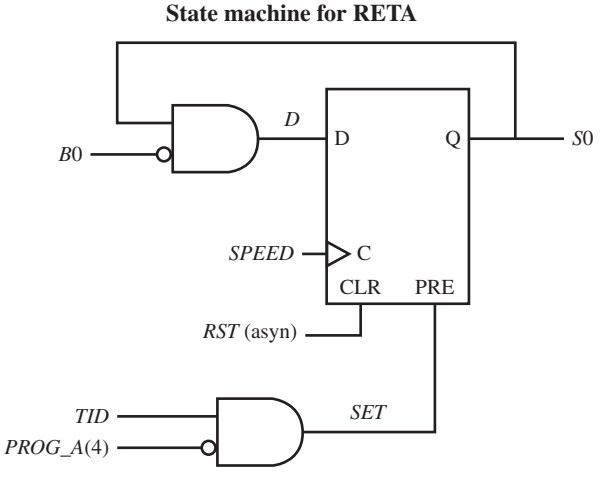


FIGURE P25.13

25.14 What is the return address when a hardware interrupt occurs, and where is it stored?

25.15 What is the return address when a software interrupt is executed, and where is it stored?

25.16 For Figure 25.3, write the D excitation equation for the state diagram by inspection using the Set OR Hold 1 equation. Is the signal *SET* a synchronous input or an asynchronous input to the D flip-flop?

25.17 Show an equivalent state sequence diagram for the State Machine for RETA in Figure 25.3. Write the D excitation equation for your state sequence diagram by inspection using the Set OR Hold 1 equation.

25.18 Show an equivalent PS/NS table for the state machine for RETA in Figure 25.3. Write the D excitation equation for your PS/NS table by inspection using the Set OR Hold 1 equation. (Hint: Remember that $D = S0^+$ for a D flip-flop.)

A Appendix

Laboratory Experiments

Experiments 1A through 25L (34 experiments)

EXPERIMENT 1A: DESIGNING AND SIMULATING GATES

1. Learning Objectives

In this experiment, you will practice working with VHDL. First, you will be guided through the design of a 3-input AND gate using VHDL. Second, you will be guided through the simulation of a 3-input AND gate to verify that the design works as predicted or expected. Then, you will be introduced to some useful tools associated with the Xilinx® software. This is summarized as follows:

1. Design a 3-input AND gate for the Boolean function $F = A \cdot B \cdot C$.
2. Simulate the design of a 3-input AND gate to verify that it works.
3. Learn some useful tools.

After your learn how to design and simulate a 3-input AND gate, you will design and simulate a few additional gates on your own.

2. Designing a 3-input AND Gate

Before we get started, you should be aware that the experiments in this book were written to support either the BASYS 2 board shown in Figure E1A.1 or the NEXYS 2 board shown in Figure E1A.2 that are manufactured by Digilent (the Digilent website is digilentinc.com). The cost of the BASYS 2 board is about $49, and the cost of the NEXYS 2 board is about $99. You will need to purchase one of these boards to perform the experiments.

Instructors and students are encouraged to use the free version of the Xilinx Integrated Software Environment (ISE) called ISE WebPACK™ found on the following Xilinx website: http://www.xilinx.com/support/download/.

You can download the latest ISE WebPACK version or an earlier version. Be sure to place different versions in different directories on your computer if you elect to have more than one version on your computer at the same time. We used ISE version 9 when we first started developing the experiments for this book. At this time, ISE version 12 is currently available. All versions can be migrated to work with the latest version. For each new version, Xilinx generally speeds up the compilation time and adds new features. Updating a project to a new ISE version is handled automatically, but projects are not backward compatible—that is, once a project is updated, it will not be recognized with the older version of the ISE software. The source code for a new ISE version can be recompiled using the older ISE version, and this is sometimes helpful to know if you are trying to use both an older version and a new version. We have migrated our VHDL code from the older ISE version 9 to the newer ISE versions 10, 11, and 12 without any problems. We recommend using the current latest version and sticking with it for all the experiments.

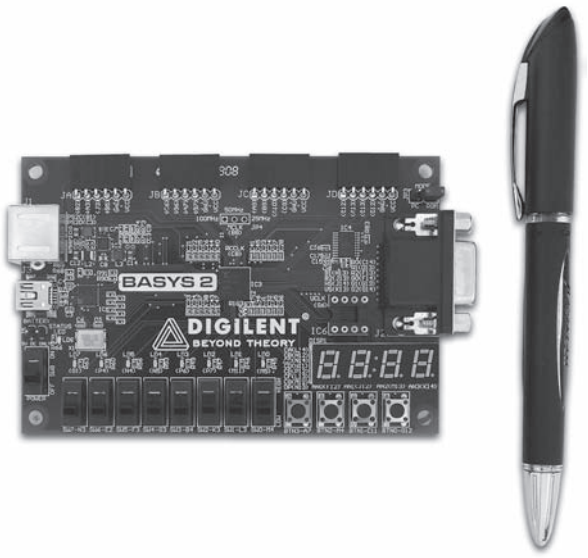


FIGURE E1A.1 BASYS 2 board
Photo courtesy of Diligent, Inc.

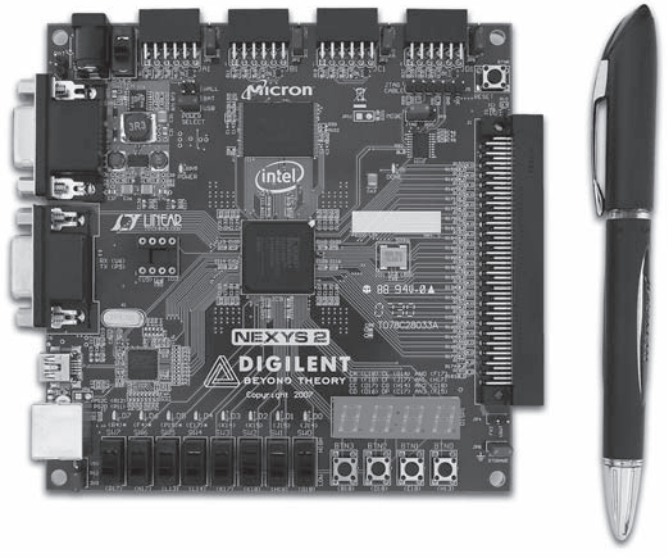


FIGURE E1A.2 NEXYS 2 board
Photo courtesy of Diligent, Inc.

First you must learn how to use the Xilinx ISE Project Navigator to enter a VHDL design.

Note: **Click** means to click the **left** mouse button, while **right click** means to click the **right** mouse button.

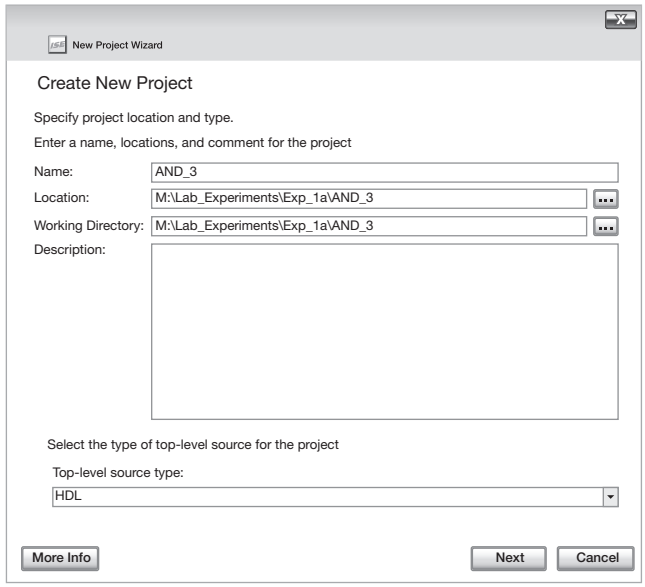
To begin, double-click the Xilinx ISE icon on the desktop. This opens up the Xilinx ISE Project Navigator. A window opens up called Tip of the Day. You can turn this off or leave it on via the check box at the bottom of the window. Click OK. Click File on the menu bar, then under the drop-down window, click New Project. This opens up the New Project Wizard window called Create New Project as shown in Window E1A.1. The Project Name, Project Location, and Top-Level Source Type are filled in as discussed later.

Be sure to choose a project name that starts with a letter and has no spaces in it. If you want to include two words, bump them together like AndGate, or you may use an underscore to separate the two words such as And_Gate. Names are case insensitive (not case sensitive) just like signals in VHDL. Names like OR2 and AND3 for gates can cause problems because these names are also used by some Xilinx library of parts and may be reserved. If you want to use similar names then use an underscore—that is, OR_2 and AND_3 work just fine. To help us identify the project in the future, we used the name AND_3. AND_3 is used, because this is our first VHDL design for a 3-input AND gate. Choose your Project Location such as the C drive on your computer in a folder such as Lab_Experiments, and a subfolder Exp_1a, e.g., C:\Lab_Experiments\Exp_1a\AND_3. *Note: Do not leave any spaces in the Project Location path*. We chose our Project Location in Window E1A.1 as our external hard drive called M:. You could also use an external USB memory stick (flash drive) if you prefer. If you elect to use a hard drive for your projects, we recommend that you also always provide a backup for your projects, which may be done using a USB memory stick. Select HDL for the Top-Level Source Type, then click Next.

This brings up the New Project Wizard window called Project Settings as shown in Window E1A.2.

Be sure to choose the Project Settings for the project as shown in the Window E1A.2. If you elect to use the BASYS 2 board, use Window E1A.2a (BASYS 2 board). If you elect to use the NEXYS 2 board, use Window E1A.2b (NEXYS 2 board). Click Next, and then click Finish.

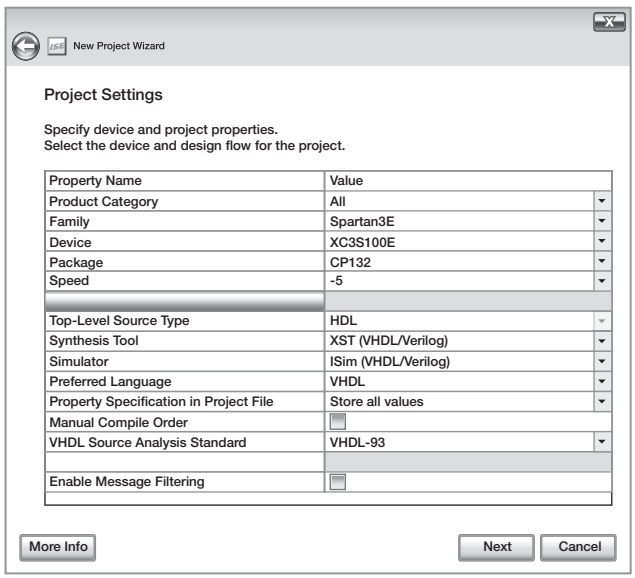
WINDOW E1A.1 New Project Wizard window called Create New Project filled in as discussed in the text



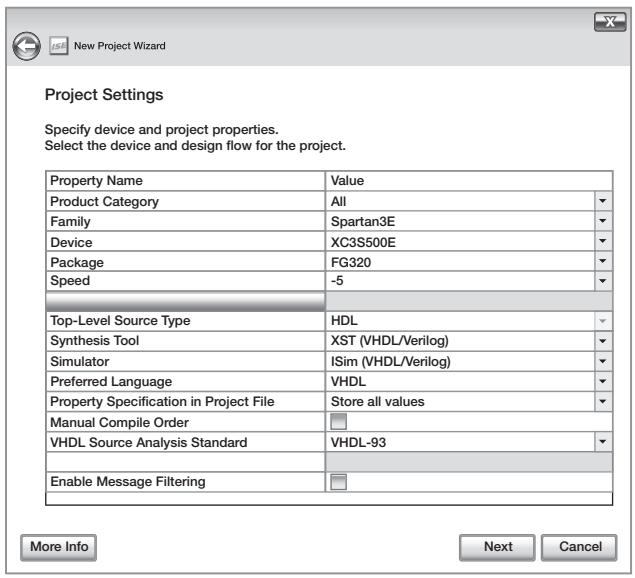
The Family, Device, and Package information can be obtain by reading the information directly off of the FPGA chip on a BASY 2 board or a NEXYS 2 board or by using the information in Appendix C.

To add a project to your design, click Project on the menu bar, then under the drop-down list, click New Source. When the New Source Wizard window called Select Source Type appears, select (click) VHDL Module in the list, and type in a File name, which will be the name of the entity in your VHDL design as shown in Window E1A.3.

We use the name AND_3 for the entity in our design. Be sure to choose a file name that starts with a letter and has no spaces in it. Click Next to close the current window and reopen a new New Source Wizard window called Define Module as shown in Window ElA.4.

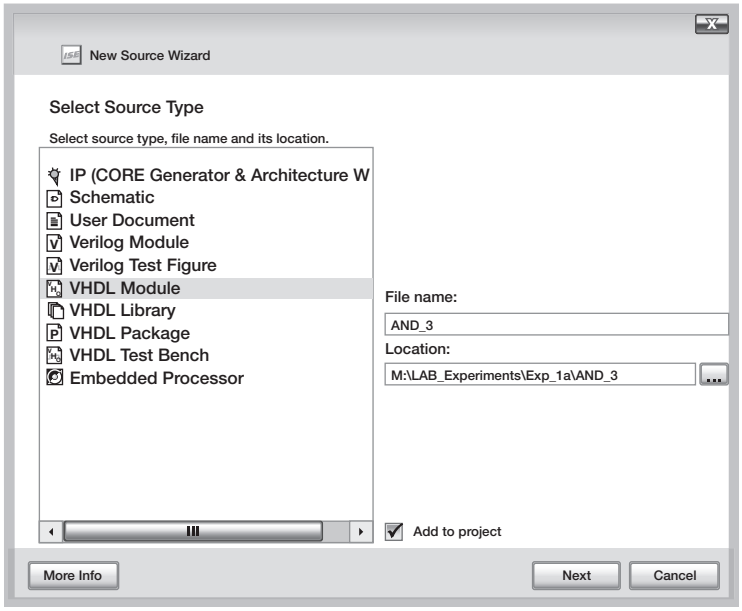


(a) BASYS 2 board



(b) NEXYS 2 board

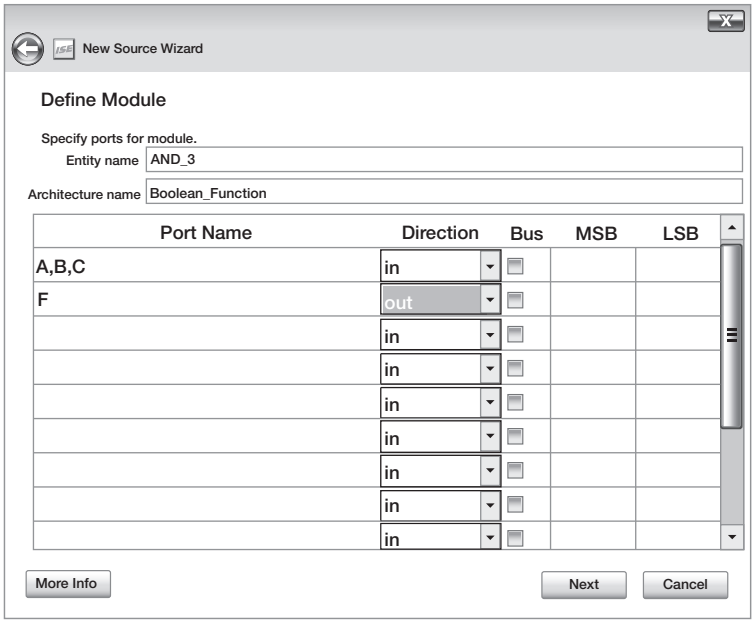
WINDOW E1A.2 (a) New Project Wizard window called Project Settings for the BASYS 2 board; (b) device properties window for the NEXYS 2 board



WINDOW E1A.3 New Source Wizard window called Select Source Type filled in as discussed in the text

The File name you typed, in the previous New Source Wizard window, is listed under the Entity Name. Change the Architecture Name, which is listed as Behavioral to Boolean_ function. Remember, do not leave spaces in a name. The Architecture Name change is not necessary, but we want to make the name meaningful for the design style we are using—that is, a simple Boolean function.

Under Port Name in the New Source Wizard window, type A, B, C on the first line. Under Direction, the mode for these signals is **in**, which is what we want it to be. Under Port Name, type F on the second line. Under Direction, use the down arrow to select (click) the mode for this signal as **out**. Click Next then Finish; this opens up a template of your design in the edit window on the ISE Project Navigator.



WINDOW E1A.4 New Source Wizard window called Define Module filled in as discussed in the text

To simplify the VHDL template, we recommend that you only retain the lines of code needed in the design and arrange the entity as shown in Listing E1A.1. Arranging the entity in the form shown in Listing E1A.1 is not necessary, but this format often helps students make fewer mistakes when writing their own VHDL code.

LISTING E1A.1
Template for the VHDL design for the 3-input AND gate (project: AND_3)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND_3 is Port (
     A,B,C : in STD_LOGIC;
     F : out STD_LOGIC
     );
end AND_3;

architecture Boolean_function of AND_3 is

begin

end Boolean_function;
```

The only thing that is missing in the design is the Boolean assignment statement for the 3-input AND gate, F <= A **and** B **and** C, which must be added in the **architecture** between **begin** and **end** Boolean_function.

Add the Boolean assignment statement to your design, then click the Save icon to save your design. Next double-click Synthesize—XST (Xilinx® Synthesis Technology) in the Processes window or right click Synthesize—XST, which opens up a drop-down list, then click run. You should get a syntax error because a semicolon was not provided at the end of the Boolean assignment statement. To see the error we are referring to, you must have the Console window open at the bottom of the ISE Project Navigator. If you got the error, then add the semicolon after C—that is, F <= A **and** B **and** C;. Click the Save icon to save your correction, and rerun Synthesize—XST and the error should go away.

Errors show up in the Console window or Error window at the bottom of the ISE Projector Navigator if they occur. If there are no errors you get the following text: Process "Synthesize—XST" completed successfully. In some cases you may have warnings, but the syntax is correct. The warnings are only reported by double-clicking Synthesize—XST. These warnings may result in a circuit that performs incorrectly, so after you remove all syntax errors you should double-click Synthesize—XST to see if there are warnings that may need to be corrected. We will discuss warnings that need to be corrected later. Your VHDL design must be error free before you can obtain a simulation for your design. Do not try to obtain a simulation if you have errors in your VHDL design until you fix the errors.

3. Simulating a 3-Input AND Gate

Simulation is used to verify that the VHDL design for the 3-input AND gate is functionally correct. By functionally correct, we mean that the simulation result satisfies the truth table for a 3-input AND gate.

Don't attempt to simulate a VHDL design until Synthesize—XST is run successfully.

Note: To obtain a simulation for your design, see Appendix B for help.

4. Useful Tools

For a small design, it is useful to observe the black box and the circuit created for the design. In the upper part of the Design window, select Implementation if it is not selected.

Under Synthesize—XST in the Processes window, double-click View Technology Schematic, then click OK when the Set RTL/Tech Viewer Startup Mode window appears. Follow the instructions provided in that window. The black box for the design is shown in the Edit window. Double-click inside the black box, and a box labeled LUT is shown in the Edit window. You will learn more about IBUFs (input BUFFERs), OBUFs (output BUFFERs), and LUTs (look-up tables) later, so don't worry about them now. Double-click inside the box labeled LUT, and the LUT Dialog window appears with the Schematic tab selected so that you can view the schematic that was created for the design.

The input signals I2, I1, and I0 and the output signal O are internal signals generated by the Xilinx software. The gates are what interest us because two AND2 gates are connected together to form an AND3 gate, which is what we wanted to create with our design. Select (click) the Truth Table tab, and you will see a truth table for the AND3 gate. Observe that when all three inputs are 1 the output is 1, which is correct for an AND3 gate, and the output is 0 for all other combinations of the inputs.

Notice that there is a tab labeled Karnaugh Map in the LUT Dialog window. You will learn about Karnaugh maps later.

Tasks:

1. Open a new project and name it Compare. Use VHDL to design a circuit that compares two values applied to its inputs to see if they are equal or the same. Make the input signals *A* and *B* for the circuit. Make the output *F_OUT* for the circuit. The architecture for your design will contain an assignment statement for the function *F_OUT*. [Hint: Read Chapter 1, Section 1.4.7 (VHDL Design for an XNOR function).]
2. Write the truth table for an XNOR function with inputs *A* and *B* and output *F_OUT*.
3. Add a New Source to your project to simulate your project Compare. Make the stimulus inputs count up from 00, 01, to 11 with the signal *A* the MSB and signal *B* the LSB. Run Simulate Behavioral Model. Write a truth table for the circuit. Verify that your truth table output and your simulation output are the same. If the output values are not the same, you have a mistake that you must find and correct.
4. Open a new project and name it Two_gates. Use VHDL to design a circuit for a 3-input NOR gate and also a 3-input XOR gate in the same project. Make the input signals *X*, *Y*, and *Z* for both the NOR gate and the XOR gate. Make the output *F_NOR* for the NOR gate and make the output *F_XOR* for the XOR gate. The architecture for your design will contain an assignment statement for the function *F_NOR* and an assignment statement for the function *F_XOR*. [Hint: Read Chapter 1, Section 1.4.6 (VHDL Design for a NOR Function), and Chapter 1, Section 1.4.4 (VHDL Design for an XOR Function).]
5. Write the truth table for the functions NOR and XOR with inputs *X*, *Y*, and *Z* and outputs *F_NOR* and *F_XOR*.
6. Add a New Source to your project to simulate your project Two_gates. Make the stimulus inputs count up from 000, 001, 010, 011, up to 111 with the signal *X* the MSB and signal Z the LSB. Run Simulate Behavioral Model. Write the truth table for the circuit. Verify that your truth table outputs and your simulation outputs are the same. If the output values are not the same, you have a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is Task 6 (project Two_gates).

2. Include the complete VHDL code for Task 1 (project Compare).
3. Include the single block diagram (the black box for the entity) for project Compare.
4. Include the truth table for project Compare.
5. Include a printout of the simulation waveform diagram for project Compare.
6. Include the complete VHDL code for Task 4 (project Two_gates).
7. Include the single block diagram (the black box for the entity) for project Two_gates.
8. Include the truth table for project Two_gates.
9. Include a printout of the simulation waveform diagram for project Two_gates.
10. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
11. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 1B: COMPLETING THE DESIGN CYCLE

1. Learning Objectives

In this experiment, you will practice completing the VHDL design cycle. First, you will be guided through the steps necessary to assign the package pins for all the port signals of the design for the 3-input AND gate that you designed in Experiment 1A. Second, you will be guided through the steps necessary to generate a programming file for the 3-input AND gate. Third, you will be guided through the steps to download the programming file for the 3-input AND gate into the FPGA on a BASYS 2 board or a NEXYS 2 board. Then, you will verify that your design works in hardware. This is summarized as follows:

1. Assign package pins for all the port signals in the entity for your 3-input AND gate design.
2. Generate a programming file for your 3-input AND gate, then check to see if your VHDL code needs to be corrected based on reported errors and warnings,
3. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
4. Verify that your design works in hardware.

After you learn how to assign package pins, generate a programming file, download the programming file, and verifying that your design works in hardware, you will repeat the process for a few additional gates on your own.

2. Assigning Package Pins for the 3-Input AND Gate

You must learn more about the Xilinx ISE Project Navigator to assign the package pins for a design.

Note: **Click** means to click the **left** mouse button, while **right click** means to click the **right** mouse button.

To begin, double-click the Xilinx ISE icon on the desktop. This opens up the Xilinx ISE Project Navigator. The project that you worked on last will appear in the Project Navigator. If you want a different project, click File on the menu bar, then under the drop-down window, double-click Open Project. The Open Project window appears. Browse through the directory and locate the AND_3 project folder or whatever name you gave the 3-input AND gate folder when you set up the project in Experiment 1A. In the folder for the 3-input AND gate, click AND_3.xise or <the name you used>.xise. This opens up the 3-input AND gate project.

To assign the package pins for the port signals for your design, click Project on the menu bar, then under the drop-down window, click New Source. When the New Source Wizard window appears, select (click) Implementation Constraints File in the list, and type in a File name. We use the name **apps** to assign the package pins. Be sure to choose a file name that starts with

a letter and has no spaces in it. Click Next and then click Finish. The file you just created called apps.ucf is saved under AND_3.vhd or <the name you used>.vhd in the Sources widow.

Click on the apps.ucf file to highlight it. In the Processes window, open up User Constraints by clicking the plus sign. Double-click Edit Constraints (Text). This opens up the ISE Project Navigator Editor window. Each signal name or NET in the entity of your VHDL design must be assigned a location (LOC), which represents an FPGA pin connection. The Edit Constraints (Text) file is where that information is placed.

Input signals in the entity will be assigned to slide switches or push-button switches, while output signals will be assigned to single LED or to an LED segment or input control signal of DISP1 (the 7-segment display).

Both the BASYS 2 board and the NEXYS 2 board have eight slide switches, four push-button switches, eight single LEDs, a four 7-segment display, and four peripheral connectors. The eight slide switches are labeled SW7 down to SW0, the four push-button switches are labeled BTN3 down to BTN0, the eight single LEDs are labeled LD7 down to LD0, the four 7-segment display is just labeled DISP1, and the four peripheral connectors are labeled JA, JB, JC, and JD. You should be able to easily identify each of these I/O devices on a BASYS 2 or on a NEXYS 2 board. Each of these I/O devices is connected to an FPGA pin except for the four 7-segment display, which is connected to multiple pins. Appendix C provides a handy reference for the FPGA pin connections on the BASYS 2 board and the FPGA pin connections NEXYS 2 board.

If you elect to use the BASYS 2 board, see Appendix C (FPGA Pin Connections), Section C.1. Assign input signal *A* to SW2 (K3), input signal *B* to SW1 (L3), and input signal *C* to SW0 (P11) so that you can use the switches to provide inputs for the 3-input AND gate. Assign output *F* to LD0 (M5) so you can observe when the output is 1 or true when the single LED is lighted or the output is 0 or false when the signal LED is not lighted.

The Edit Constraints (Text) file must be typed in as shown in Figure E1B.1 for the BASYS 2 board for project AND_3.

```
# a comment can appear after the symbol "#"
NET "A" LOC = K3   ;
NET "B" LOC = L3   ;
NET "C" LOC = P11  ;
NET "F" LOC = M5   ;
```

FIGURE E1B.1 Edit Constraints (Text) file for the BASYS 2 board for project AND_3

If you elect to use the NEXYS 2 board, see Appendix C(FPGA Pin Connections), Section C.2. Assign input signal *A* to SW2 (K18), input signal *B* to SW1 (H18), and input signal *C* to SW0 (G18) so that you can use the switches to provide inputs for the 3-input AND gate. Assign output *F* to LD0 (J14) so you can observe when the output is 1 or true when the single LED is lighted or the output is 0 or false when the signal LED is not lighted.

The Edit Constraints (Text) file must be typed in as shown in Figure E1B.2 for the NEXYS 2 board for project AND_3.

```
# a comment can appear after the symbol "#"
NET "A" LOC = K18  ;
NET "B" LOC = H18  ;
NET "C" LOC = G18  ;
NET "F" LOC = J14  ;
```

FIGURE E1B.2 Edit Constraints (Text) file for the NEXYS 2 board for project AND_3

After you fill in all the I/O locations for the 3-input AND gate design, click the Save icon.

An alternate way to assign package pins is to open up User Constraints in the Processes window and click on I/O Pin Planning (PlanAhead)—Post-Synthesis. Select a signal in the I/O Port window and that signal will appear in the I/O Port Properties window. Simply fill in the Site location with the FPGA pin for the selected signal and click Apply. This process must be repeated for all the I/O port signals. After you complete entering all the FPGA pins for all the I/O port signals, click Save Design and close the current window. The PlanAhead program

automatically generates the .ucf file for the design and lists the NETs and locations (LOCs) with the proper signals and FPGA pins as shown in Figures E1B.1 and E1B.2.

3. Generate a Programming File for the 3-Input AND Gate

To generate a programming file, click the file name for your design project to highlight it (AND_3 in this case), then double-click Generate Programming File in the Processes window. The ISE software reruns the Synthesize—XST process to include the user constraints file. It starts the Implement Design process, which translates the VHDL design into an internal netlist format. It maps the design by optimizing the number of gates, and finally it assigns gates on the FPGA chip and their interconnections are routed through the programmable switch matrices on the FPGA chip. From a user standpoint, this is unimportant. If the Implement Design process completes successfully, then the ISE software begins the Generate Programming File process, which creates a stream of bits called the .bit file. If errors or warnings are reported after the Generate Programming File process is completed, you *must* correct all errors in your VHDL code. Only certain warnings need to be corrected in your VHDL code. These will be discussed later—for example, clock skew, latches, and gated clocks. If certain warnings are not corrected, your VHDL code may not always run correctly.

The .bit file can be downloaded into the FPGA chip to carry out the logic function(s) described in the VHDL code. The name that is used for the entity of your VHDL file is the prefix for the .bit file that is created for the design. For the 3-input AND gate, the .bit file that is created is named AND_3.bit.

4. Download the Programming File into the FPGA

To download the programming file into the FPGA either on the BASYS 2 board or the NEXYS 2 board, you must connect the USB cable to the computer and also to the USB connector on the BASYS 2 board or the NEXYS 2 board. For the BASYS 2 board, set the switch SW8 (the Power Switch) to on and observe that the red POWER LED turns on brightly. On the NEXYS 2 board for the POWER SELECT header move the jumper to USB, then set the switch labeled POWER SWITCH to on and observe that the red POWER LED turns on brightly. On the three-pin header identified with the labels MODE, move the jumper to PC on the BASYS 2 board or move the jumper to JTAG on the NEXYS 2 board.

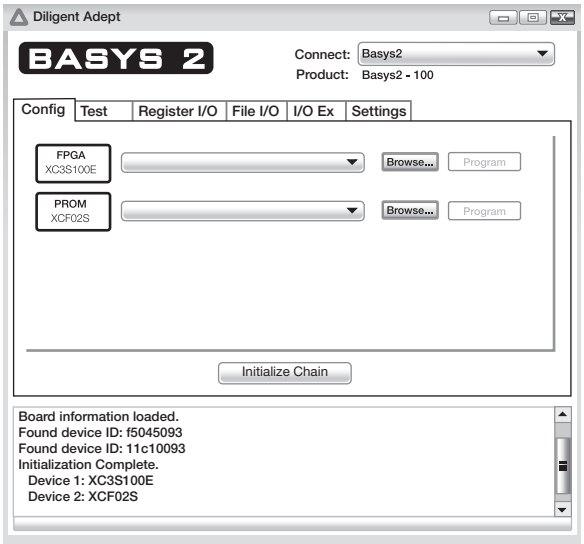
Open the program Digilent Adept* wherever it is installed on your computer. The Adept program identifies the board connected to the USB connector as shown in Window E1B.1 for the BASYS 2 board and Window E1B.2 for the NEXYS 2 board.

When the Digilent Adept window appears, browse to find your folder with the AND_3 gate design, then double-click AND_3.bit or <the name you used>.bit. A Warning window appears, which is normal, so click Yes. Then click Program to download the bit file from your computer into the FPGA. The message Programming Successful appears when the FPGA has been programmed. Later you may want to download the bit file from your computer into the PROM on the FPGA board so your design will begin running as soon as power is supplied to the board. The PROM is a simple flash memory that stores the .bit file.

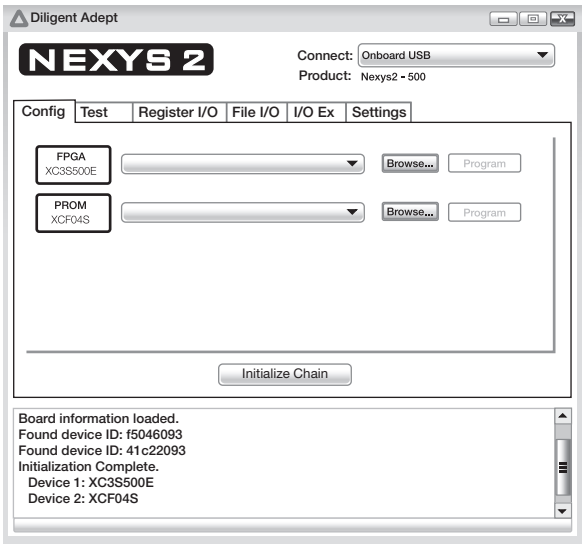
You can now check to see if the 3-input AND gate is correctly working in hardware using the slide switches SW2 down to SW0. The single LED LD0 should only light when all three slide switches are pushed forward to their 1 position. Be sure to check this out on your BASYS 2 board or your NEXYS 2 board to verify that your 3-input AND gate works in hardware as expected.

Tasks:

1. Open project Compare for the 2-input comparator circuit, that you designed in Experiment 1A, and complete the design cycle by doing the following:



WINDOW E1B.1 Digilent Adept window for the BASYS 2 board



WINDOW E1B.2 Digilent Adept window for the NEXYS 2 board

   a. Assign package pins for all the port signals in the entity (assign *A* to SW1, *B* to SW0, and *F_OUT* to LD0).
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
2. Check to see if your 2-input comparator works in hardware—that is, when SW1 and SW0 are both pushed forward or both pulled back, LD0 should light; otherwise, LD0 should not light. If your design does not perform in this manner, you have made a mistake that you must find and correct.
3. Open project Two_gates for the 3-input NOR gate and the 3-input XOR gate circuits that you designed in Experiment 1A, and complete the design cycle by doing the following:
   a. Assign package pins for all the port signals in the entity (assign *X* to SW7, *Y* to SW6, *Z* to SW5, *F_NOR* to LD7, and *F_XOR* to LD0).
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
4. Check to see if your 3-input NOR gate and your 3-input XOR gate work in hardware. When SW7, SW6, and SW5 are all pulled back, LD7 should light; otherwise LD7 should not light. LD0 should light when an odd number of the slide switches SW7, SW6, and SW5 are all pushed forward, and should not light otherwise. If your design does not perform in this manner, you have made a mistake that you must find and correct.
5. Write an alternate assignment statement that can be substituted for the assignment statement for F_NOR in your VHDL design for the 3-input NOR gate. Also write an alternate assignment statement the can be substituted for the assignment statement for F_XOR in your VHDL design for the 3-input XOR gate. (Hint: Consider using DeMorgan's theorem

to obtain the alternate assignment statement for the 3-input NOR gate, and consider using an SOP form for the alternate assignment statement for the 3-input XOR gate.)

6. Open project Two_gates and substitute the alternate assignment statement you obtained in Task 5 in you VHDL code (don't get rid of the previous assignments, simply comment them out), then complete the design cycle by doing the following:
   a. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   b. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. Check to see if your 3-input NOR gate and your 3-input XOR gate work in hardware the same way they did in task 4. When SW7, SW6, and SW5 are all pulled back, LD7 should light; otherwise LD7 should not light. LD0 should light when an odd number of the slide switches SW7, SW6, and SW5 are all pushed forward, and should not light otherwise. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is Task 7 (project Two_gates).
2. Include the complete VHDL code for project Compare.
3. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for project Compare. The Edit Constraints (Text) is an entry under User Constraints in the Processes window. Double-click Edit Constraints (Text) to show the text in the right-hand side of the Project Navigator window. To make a copy of it, click File, then click Print.
4. Include the alternate assignment statement for *F_NOR* for the 3-input NOR gate and the alternate assignment statement for *F_XOR* for the 3-input XOR gate you obtained in task 5.
5. Include the complete VHDL code for project Two-gates that includes the alternate assignment statements for *F_NOR* and *F_XOR*.
6. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for project Two_gates.
7. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
8. Your lab instructor may add additional requirements for this experiment.

---

* The Adept software enables you to use the USB cable. Here is how to install the Adept software on your personal computer. The procedure may be slightly different for different operating systems and even newer version of Adept.

1. Launch Internet Explorer Browser. In the Address space for the Browser, type http://digilentinc.com/Products/Detail.cfm?NavPath=2,66,69&Prod=ADEPT then click Download! to Download Digilent ADEPT.
2. When the Adept window appears, click Save to save the file DAS(x86)2-0-10.exe on your desktop, in case you want to execute it later; otherwise, click Run. When the download is complete, click Run, then click Run again, then click Next.
3. Click the radio button "I accept the terms of the license agreement," then click Next.
4. When the Adept Runtime—InstallShield Wizard, window appears, click Next, click Next, then click Install, then click Finish.
5. When you get the window Adept—Install Shield Wizard, click Install, then click Finish.
6. You can click on Adept and pin it to the start menu if you prefer.

EXPERIMENT 2: DESIGNING AND TESTING A KEYPAD ENCODER SYSTEM

1. Learning Objectives

In this experiment, you will practice working with an encoder system for a keypad. First, you will learn how to design a keypad encoder and simulate it to verify that it works. Second, you will learn how to design a decimal display decoder and simulate it to verify that it works. Third, you will learn how to combine designs with some additional modules to obtain a flat design for a complete system. Then, you will download and test your design in hardware to verify that the system works. This is summarized as follows:

1. Design a keypad encoder and simulate it to verify that it works.
2. Design a decimal display decoder and simulate it to verify that it works.
3. Obtain a flat design for the complete keypad encoder system.
4. Download and test your design in hardware.

2. Keypad Encoder System with Its Inputs and Outputs

Figure E2.1 shows an annotated schematic for the keypad encoder system.

Notice that the four push buttons on the BASYS 2 board or the NEXYS 2 board are used as a simple 4-input keypad. The push-button switches are connected to module 1, the keypad encoder. The truth table for the keypad encoder is shown in Table E2.1.

TABLE E2.1 Truth table for the keypad encoder

*B*3	*B*2	*B*1	*B*0	*E*2	*E*1	*E*0	
0	0	0	1	0	0	0	Only press *B*0 so output *E*2 *E*1 *E*0 is 000 or 0
0	0	1	0	0	0	1	Only press *B*1 so output *E*2 *E*1 *E*0 is 001 or 1
0	1	0	0	0	1	0	Only press *B*2 so output *E*2 *E*1 *E*0 is 010 or 2
1	0	0	0	0	1	1	Only press *B*3 so output *E*2 *E*1 *E*0 is 011 or 3
0	0	0	0	1	1	1	*E*2 is 1 to indicate no button is pressed (We elected to make *E*1 and *E*0 also 1s because it doesn't matter what their values are when *E*2 is 1)

Notice that a keypad encoder has a restriction placed on its inputs just like the restriction placed on the inputs of telephone keypads and computer keyboards. The restriction is simply that you are only allowed to press one key at a time.

Notice in Figure E2.1 that the outputs of the keypad encoder are connected to the inputs of the decimal display decoder. The truth table for a decimal display decoder with active high outputs is shown in Table E2.2.

TABLE E2.2 Truth table for the decimal display decoder with active high outputs

*E*2	*E*1	*E*0	*S*7	*S*6	*S*5	*S*4	*S*3	*S*2	*S*1	*S*0	
0	0	0	0	0	1	1	1	1	1	1	7-segment display
0	0	1	0	0	0	0	0	1	1	0	
0	1	0	0	1	0	1	1	0	1	1	
0	1	1	0	1	0	0	1	1	1	1	
1	1	1	0	0	0	0	0	0	0	0	Blank the display (turn off all segments)

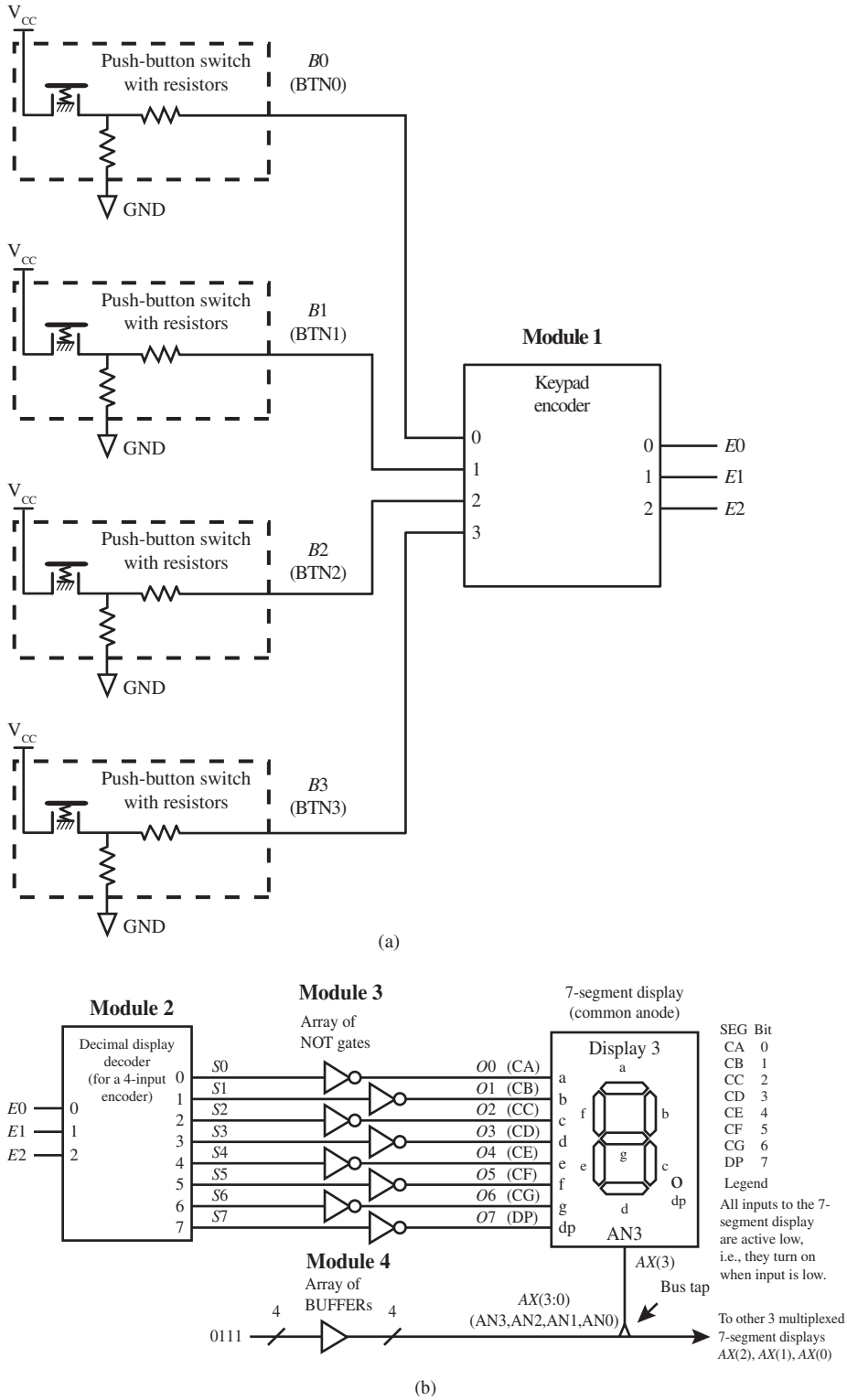


FIGURE E2.1 Annotated schematic for the keypad encoder system: (a) push-button inputs and keypad encoder; (b) decimal display decoder, array of NOT gate, array of buffers, and 7-segment display

Recommended Pre-Lab:

1. Tasks 1 through 4.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms, etc.

Tasks:

1. Write each Boolean function output for the keypad encoder, in the simplest compact (minterm or maxterm) form—that is, use the fewest number of 0s or 1s. Write a canonical form for each compact form. Use each canonical form to write an assignment statement for the output of the Keypad Encoder in VHDL.
2. Create a new project and write complete VHDL code for Figure E2.1a for the keypad encoder (module 1). Simulate the design to verify that it follows the keypad encoder truth table. [Note: see Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If the outputs of the keypad encoder do not follow the truth table, then you know that the VHDL code for module 1 has an error. You must find the error or errors and fix them.
3. Write each Boolean function output of the decimal display decoder in the simplest compact minterm or maxterm) form—that is, use the fewest number of 0s or 1s. Write a canonical form for each compact form. Use each canonical form to write an assignment statement for the output of the decimal display decoder in VHDL.
4. Create a new project and write complete VHDL code for Figure E2.1b for the decimal display decoder (module 2). Simulate the design to verify that it follows the decimal display decoder truth table. If the outputs of the decimal display decoder do not follow the truth table, then you know that the VHDL code for module 2 has an error. You must find the error or errors and fix them.
5. Create a new project and write complete VHDL code for the keypad encoder system using a flat design approach. (Hint: See Chapter 2, Section 2.4.5, for discussion of a flat design approach.) Use the designs for modules 1 and 2 and include them in a single architecture, then add the VHDL code for modules 3 and 4. Be sure that all external input and output signals are declared in the entity and that all internal signals are declared between **architecture** and the first **begin**. Document your VHDL code as shown in the design in Chapter 2, Section 2.4.5, Listing 2.1.
6. Complete the design cycle for your keypad encoder system by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. Check to see if your keypad encoder system works in hardware—that is, when BTN0 is pressed the 7-segment display should display a 0, when BTN1 is pressed, the 7-segment display should display a 1; when BTN2 is pressed, the 7-segment display should display a 2; and when BTN3 is pressed, the 7-segment display should display a 3; and when no push button is pressed, the 7-segment display should be blank. The push buttons may be pressed in any order, but only press one push button at a time. If your keypad encoder system design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information:

course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 7 (Keypad Encoder System).

2. Include the simplest compact (minterm or maxterm) forms for the Boolean equations for task 1 (Keypad Encoder design).
3. Include the complete VHDL code for your keypad encoder design.
4. Include a printout of the simulation waveform diagram for your keypad encoder design. Identify the inputs on the waveform diagram using minterm designators.
5. Include the simplest compact (minterm or maxterm) forms for the Boolean equations for your decimal display decoder design.
6. Include the complete VHDL code for your decimal display decoder design.
7. Include a printout of the simulation waveform diagram for your decimal display decoder design. Identify the inputs on the waveform diagram using minterm designators.
8. Include the complete VHDL code for your keypad encoder system.
9. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your keypad encoder system.
10. If the truth table outputs were made active low for the decimal display decoder, explain what changes would have to be made to the keypad encoder system so that the design would work the same way in hardware as before. (Hint: Verify your suggestions; make the changes to see if your hardware design still works the same as before.)
11. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
12. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 3: DESIGNING AND TESTING A CHECK GATES SYSTEM

1. Learning Objectives

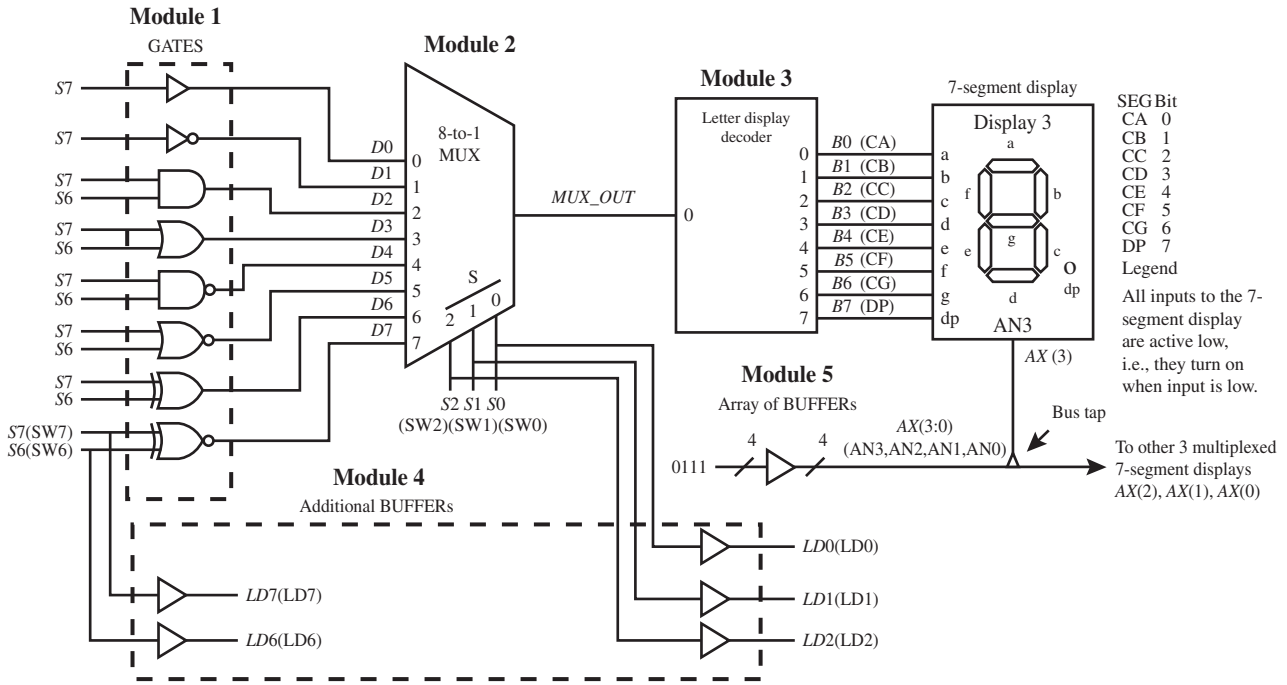
In this experiment, you will practice working with an 8-to-1 MUX (multiplexer) and all the basic gates. First, you will learn how to design an 8-to-1 MUX and simulate it to verify that it works. Second, you will learn how to design a letter display decoder with active low outputs and simulate it to verify that it works. Third, you will learn how to combine designs with some additional modules to obtain a flat design for a complete system. Then, you will download and test your design in hardware to verify that the system works. This is summarized as follows:

1. Design an 8-to-1 MUX and simulate it to verify that it works.
2. Design a letter display decoder and simulate it to verify that it works.
3. Obtain a flat design for the complete check gates system.
4. Download and test your design in hardware.

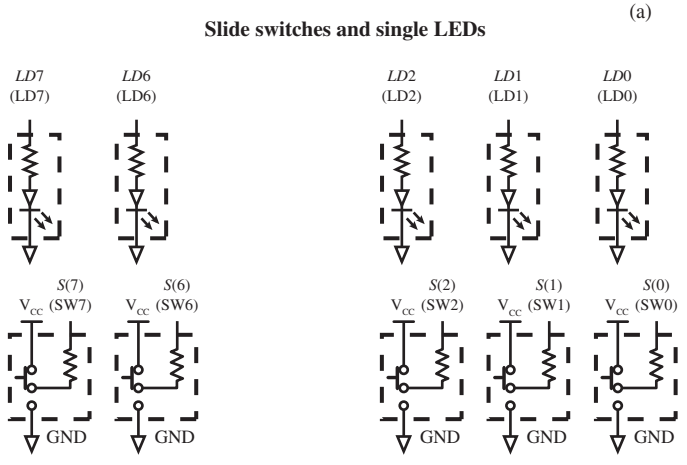
2. Check Gates System with its Inputs and Outputs

Figure E3.1 shows an annotated schematic for the check gates system.

Notice that slide switches SW7 and SW6 are used to provide the inputs to the gates. The single LEDs LD7 and LD6 show the status of these switches—that is, when a slide switch is pushed forward for a 1, the corresponding LED lights. Slide switches SW2, SW1, and SW0 are used to provide the select inputs to the 8-to-1 MUX. The single LEDs LD2, LD1, and LD0 show the status of these switches. The value of the output of each gate is shown on the 7-segment display as an L when the value is 0 and as an H when the value is 1.



(a)



(b)

FIGURE E3.1 Annotated schematic for the check gates system; (a) complete system minus the slide switches and single LEDs; (b) slide switches and single LEDs

Table E3.1 shows a partially filled in compressed truth table for the 8-to-1 MUX.

A compressed truth table is often used for a large MUX because the full-blown truth table is quite large as a result of the number of inputs (11 to be exact), which would result in a truth table with $2^{11} = 2{,}048$ lines. To keep things simple, you will just be required to verify that your simulation waveform for *MUX_OUT* agrees with the 16 input combinations that are shown in the partial truth table in Table E3.2 for the 8-to-1 MUX.

TABLE E3.1 Partially filled in compressed truth table for the 8-to-1 MUX

*S*2	*S*1	*S*0	*MUX_OUT*
0	0	0	*D*0
0	0	1	*D*1
0	1	0	*D*2
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

TABLE E3.2 Partial truth table for the 8-to-1 MUX with 8 input combinations for testing

*S*2	*S*1	*S*0	*D*7	*D*6	*D*5	*D*4	*D*3	*D*2	*D*1	*D*0	*MUX_OUT*
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	1
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	1
1	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0	1
1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	1
1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	1

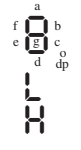
Table E3.3 shows a partially filled in truth table for the letter display decoder with active low outputs.

TABLE E3.3 Partially filled in truth table for the letter display decoder with active low outputs

Letter display decoder (active low outputs)

MUX_OUT	*B*7	*B*6	*B*5	*B*4	*B*3	*B*2	*B*1	*B*0
0	1	1	0	0	0	1	1	1
1								

7-segment display



Recommended Pre-Lab:

1. Tasks 1 through 6.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Fill in the compressed truth table in Table E3.1 for the 8-to-1 MUX.
2. Write the Boolean function for the 8-to-1 MUX in SOP form. (Hint: See Chapter 3, Section 3.8, Figure 3.21b, for a 4-to-1 MUX and use extrapolation.) Do not write the Boolean

function for the 8-to-1 MUX using Table E3.2, because the function will contain many more literals than necessary.

3. Create a new project and write complete VHDL code for the 8-to-1 MUX. Simulate the design for the 8 input combinations shown in Table E3.2 to verify that your design follows the partial truth tables. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If the output of the 8-to-1 MUX does not follow these truth tables, then you know that the VHDL code for module 2 has an error. You must find the error or errors and fix them.
4. Fill in the truth table in Table E3.3 for the letter display decoder with active low outputs.
5. Write the Boolean function outputs for the letter display decoder.
6. Create a new project and write complete VHDL code for the letter display decoder. Simulate the design to verify that it follows the truth table for the letter display decoder. If the outputs of the letter display decoder do not follow the truth table for the letter display decoder, then you know that the VHDL code for module 3 has an error. You must find the error or errors and fix them.
7. Create a new project and write complete VHDL code for the check gates system using a flat design approach. (Hint: See Section 2.4.5 in Chapter 2 for a discussion of a flat design approach.) Use the designs for modules 2 and 3 and include them in a single architecture; then add the VHDL code for modules 1, 4, and 5. Be sure that all external input and output signals are declared in the entity and that all internal signals are declared between **architecture** and the first **begin**. Document your VHDL code as shown in the design in Chapter 2, Section 2.4.5, Listing 2.1.
8. Complete the design cycle for your check gates system by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
9. Check to see if your check gates system works in hardware—that is, when slide switches SW2, SW1, and SW0 select a respective gate, the gate can be checked to see it works according to its truth table using slide switches SW7 and SW6. When a gate output is 0, the 7-segment display will show an L (for low). When the gate output is 1, the 7-segment display will show an H (for high). The single LEDs are used to show the status of the inputs to the gates and also the status of the inputs to the 8-to-1 MUX. If your check gates system design does not verify the correct functionality of each gate, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 9 (check gates system).
2. Include the filled in compressed truth table for the 8-to-1 MUX.
3. Included the Boolean function for the 8-to-1 MUX in SOP form.
4. Include the complete VHDL code for your 8-to-1 MUX design.
5. Include a printout of the simulation waveform diagram for your 8-to-1 MUX.
6. Include the filled-in truth table for the letter display decoder.
7. Include the Boolean function outputs for the letter Display Decoder.
8. Include the complete VHDL code for your letter display decoder design.

9. Include a printout of the simulation waveform diagram for your letter display decoder design.
10. Include the complete VHDL code for your check gates system design.
11. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your check gates system design.
12. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
13. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 4: DESIGNING AND TESTING A CUSTOM DECIMAL DISPLAY DECODER SYSTEM

1. Learning Objectives

In this experiment, you will practice working with a custom decimal display decoder system. First, you will learn how to design a display decimal decoder using vectors in VHDL and simulate it to verify that it works. Second, you will learn how to design a MUX array and simulate it to verify that it works. Third, you will learn how to combine designs with some additional modules to obtain a flat design for the complete system. Then, you will download and test your design in hardware to verify that the system works. This is summarized as follows:

1. Design a decimal display decoder and simulate it to verify that it works.
2. Design a MUX array and simulate it to verify that it works.
3. Obtain a flat design for the complete custom decimal display decoder system.
4. Download and test your design in hardware.

2. Custom Decimal Display Decoder System with Inputs and Outputs

Figure E4.1 shows an annotated schematic for the custom decimal display decoder system.

Notice that the four slide switches SW3, SW2, SW1, and SW0 are used for the BCD inputs and one push-button switch BTN3 is used for the lamp test (LT) input. Notice that SW7 is used to switch between Disp 3 (display 3) and Disp 0 (display 0) of the 7-segment display.

Table E4.1 shows a partially filled in truth table for the decimal display decoder with active low outputs. When the BCD value is 0 through 9 and the lamp test input is inactive, the BCD value should be displayed on the 7-segment display. When the lamp test input is active, signal *S*4 is 1, all of the segments for the 7-segment display are turned on to test if the segments are good—that is, not burned out. When a BCD value is not entered (binary values 10 through 15 are entered) and the lamp test is inactive, the display should be blank—that is, all segments turned off.

Table E4.2 shows a template for a compressed truth table for the array of MUXs. When the input signal *SEL* is 0 and 1, you must fill in the outputs provided by the array of MUXs (module 2). Keep in mind that the output of the array of MUXs is a bus that has the data type std_logic_vector in VHDL.

Recommended Pre-Lab:

1. Tasks 1 through 4.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

TABLE E4.1 Partially filled-in truth table for the decimal display decoder with active low outputs

Decimal display decoder												7-segment display
S(4)	*S*(3)	*S*(2)	*S*(1)	*S*(0)	*D*(6)	*D*(5)	*D*(4)	*D*(3)	*D*(2)	*D*(1)	*D*(0)	
0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	0	0	1	1
0	0	0	1	0								2
0	0	0	1	1								3
0	0	1	0	0								4
0	0	1	0	1								5
0	0	1	1	0								6
0	0	1	1	1								7
0	1	0	0	0								8
0	1	0	0	1								9
0	1	0	1	0								Blank*
0	1	0	1	1								Blank
0	1	1	0	0								Blank
0	1	1	0	1								Blank
0	1	1	1	0								Blank
0	1	1	1	1								Blank
1	X	X	X	X								8

* All segments turned off

TABLE E4.2 Template for a compressed truth table for the array of MUXs

SEL	*AX*(3:0)
0	
1	

Tasks:

1. Fill in the rest of the truth table in Table E4.1 for the decimal display decoder.
2. Create a new project and write complete VHDL code for the decimal display decoder (module 1). Using a dataflow design style with a conditional signal assignment (when-else statement). Simulate the design to verify that it follows the truth table that you created just for the inputs *S*(4) *S*(3) *S*(2) *S*(1) *S*(0) = 00000 to 1XXXX. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If the outputs of the decimal display decoder do not follow the truth table, then you know that the VHDL code for module 1 has an error. You must find the error or errors and fix them.
3. Fill in the truth table in Table E4.2 for the array of MUXs.
4. Create a new project and write complete VHDL code for the array of MUXs (module 2). Use a dataflow design style with a conditional signal assignment (when-else statement).

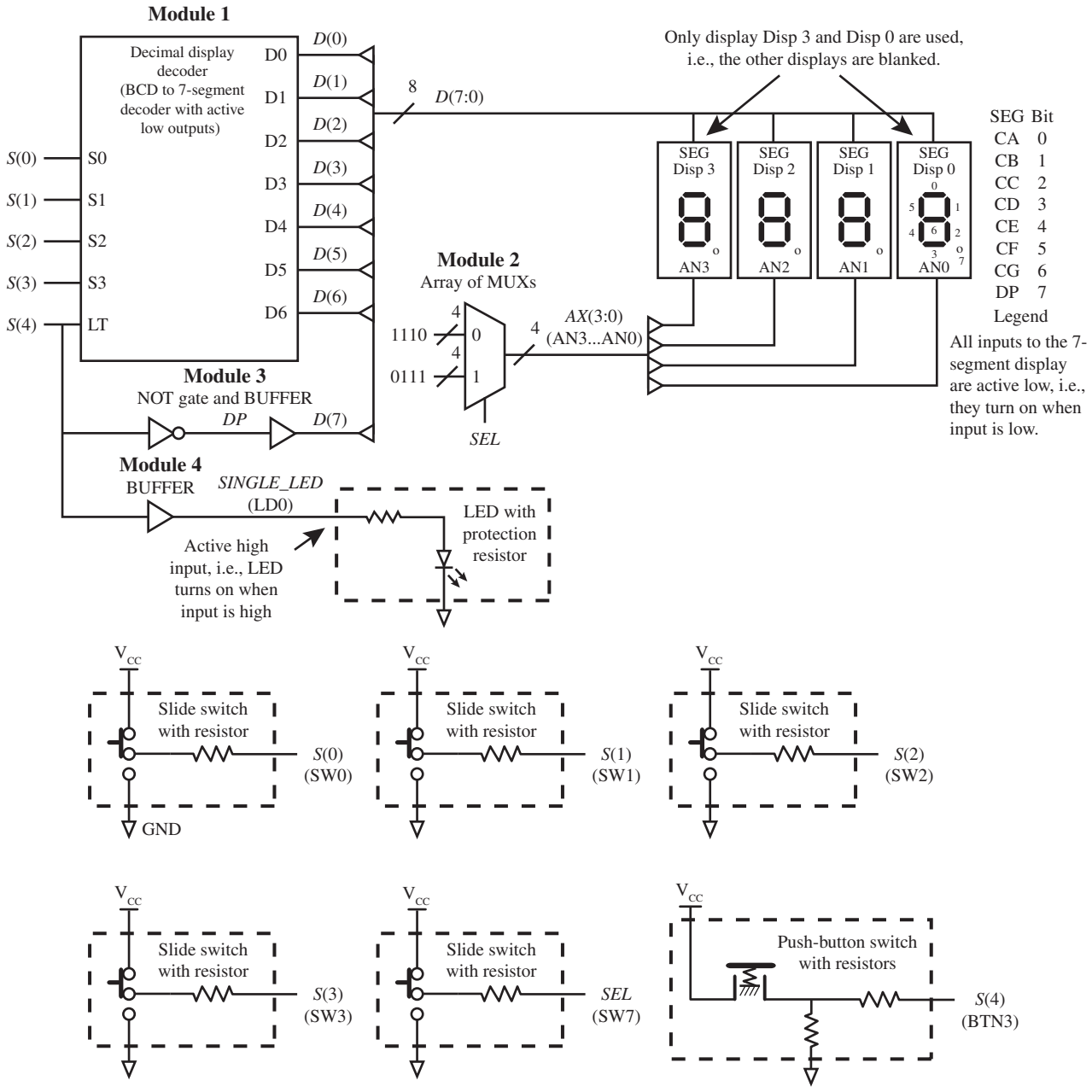


FIGURE E4.1 Annotated schematic for the custom decimal display decoder system

Simulate the design to verify that it follows the truth table that you created. If the outputs of the array of MUXs do not follow the truth table, then you know that the VHDL code for module 2 has an error. You must find the error or errors and fix them.

5. Create a new project and write complete VHDL code for the custom decimal display decoder system using a flat design approach. Use the designs for modules 1 and 2 and include them in a single architecture; then add the VHDL code for modules 3 and 4. Be sure that all external input and output signals are declared in the entity and that all internal signals are declared between **architecture** and the first **begin** (Hint: See Section 2.4.5 in Chapter 2 or the end of Section 4.8 in Chapter 4 for a discussion of a flat design approach.)

Use Boolean equations to write the assignments for *DP*, *D*(7), and *SINGLE_LED*. (Hint: To form the bus for *D*, use either an aggregate or the concatenation operator.)

6. Complete the design cycle for your custom decimal display decoder system by doing the following:
   a. Assign package pins for all the port signals in the entity for your design. Note: Bus signals such as *S*(0), *S*(1), etc, must be listed in the Edit Constraints (Text) file as S<0>, S<1>, etc.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. Check to see if your custom decimal display decoder system works in hardware. Verify that each binary coded decimal (BCD) number entered by the four slide switches is displayed in decimal on the 7-segment display and non-BCD numbers will result in a blank display. When the lamp test push button is pressed (lamp test input is 1), all of the segments of the 7-segment display, the decimal point of the 7-segment display, and the single LED LD0 turns on. When slide switch SW7 is pulled back, DSP0 is enabled and when slide switch SW7 is pushed forward, DSP3 is enabled. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 7 (Custom Decimal Display Decoder System).
2. Include the filled-in truth table for the decimal display decoder.
3. Include the complete VHDL code for your decimal display decoder design.
4. Include a printout of the simulation waveform diagram for your decimal display decoder design.
5. Include the filled-in truth table for the array of MUXs.
6. Include the complete VHDL code for your array of MUXs design.
7. Include a printout of the simulation waveform diagram for the array of MUXs design.
8. Include the complete VHDL code for your custom decimal display decoder system.
9. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your custom decimal display decoder system.
10. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
11. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 5A: DESIGNING AND TESTING A D LATCH AND A D FLIP-FLOP WITH A CLR INPUT

1. Learning Objectives

In this experiment, you will practice working with a D latch and D flip-flops with a *CLR* (CLEAR) input. First, you will learn how to design a D latch in VHDL and run a post-route

simulation for the design to verify that it works. Second, you will download and test your design in hardware to verify that the D latch works. Third, you will learn how to design a positive edge-triggered D flip-flop with a *CLR* input and run a post-route simulation for the design to verify that it works. Fourth, you will download and test your design in hardware to verify that the D flip-flop works. This is summarized as follows:

1. Design a D latch in VHDL and run a post-route simulation for the design to verify that it works.
2. Download and test your design in hardware to verify that the D latch works.
3. Design a positive edge-triggered D flip-flop with a *CLR* (*CLEAR*) input and run a post-route simulation for the design to verify that it works.
4. Download and test your design in hardware to verify that that the D flip-flop works.

2. D Latch with Inputs and Output

Figure E5A.1 shows an annotated schematic for a D latch circuit.

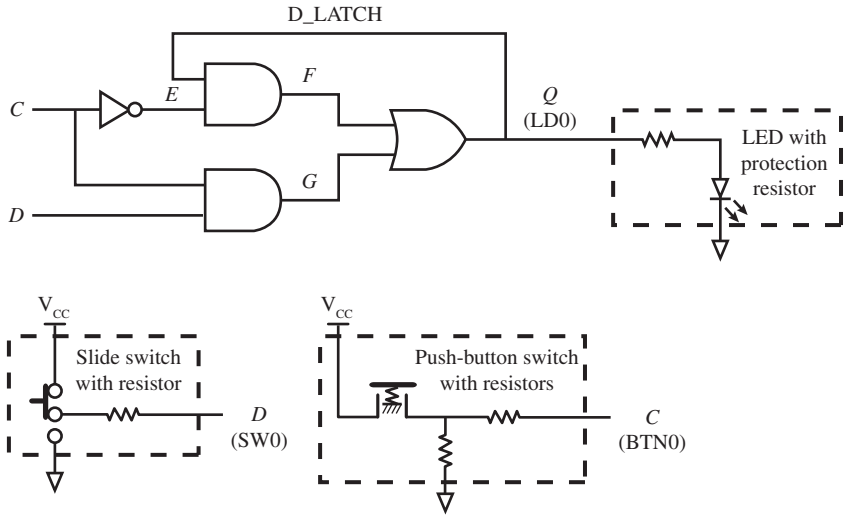


FIGURE E5A.1 Annotated schematic for a D latch circuit

Notice that one slide switch SW0 provides the *D* input value, one push-button switch BTN0 provides the *C* input value, and a single LED output LD0 provides the *Q* output value.

3. D Flip-Flop with a CLR Input with Inputs and Output

Figure E5A.2 shows an annotated schematic for a D flip-flop circuit with a *CLR* input.

Notice that one slide switch SW0 provides the *D* input value, one push-button switch BTN0 provides the *C* input value, one push-button switch BTN3 provides the *CLR* input value, and a single LED output LD0 provides the *Q* output value.

Recommended Pre-Lab:

1. Tasks 1 and 5.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Create a new project named D_LATCH, and write complete VHDL code for the D latch circuit shown in Figure E5A.1. Use a dataflow design style with Boolean equations. Obtain

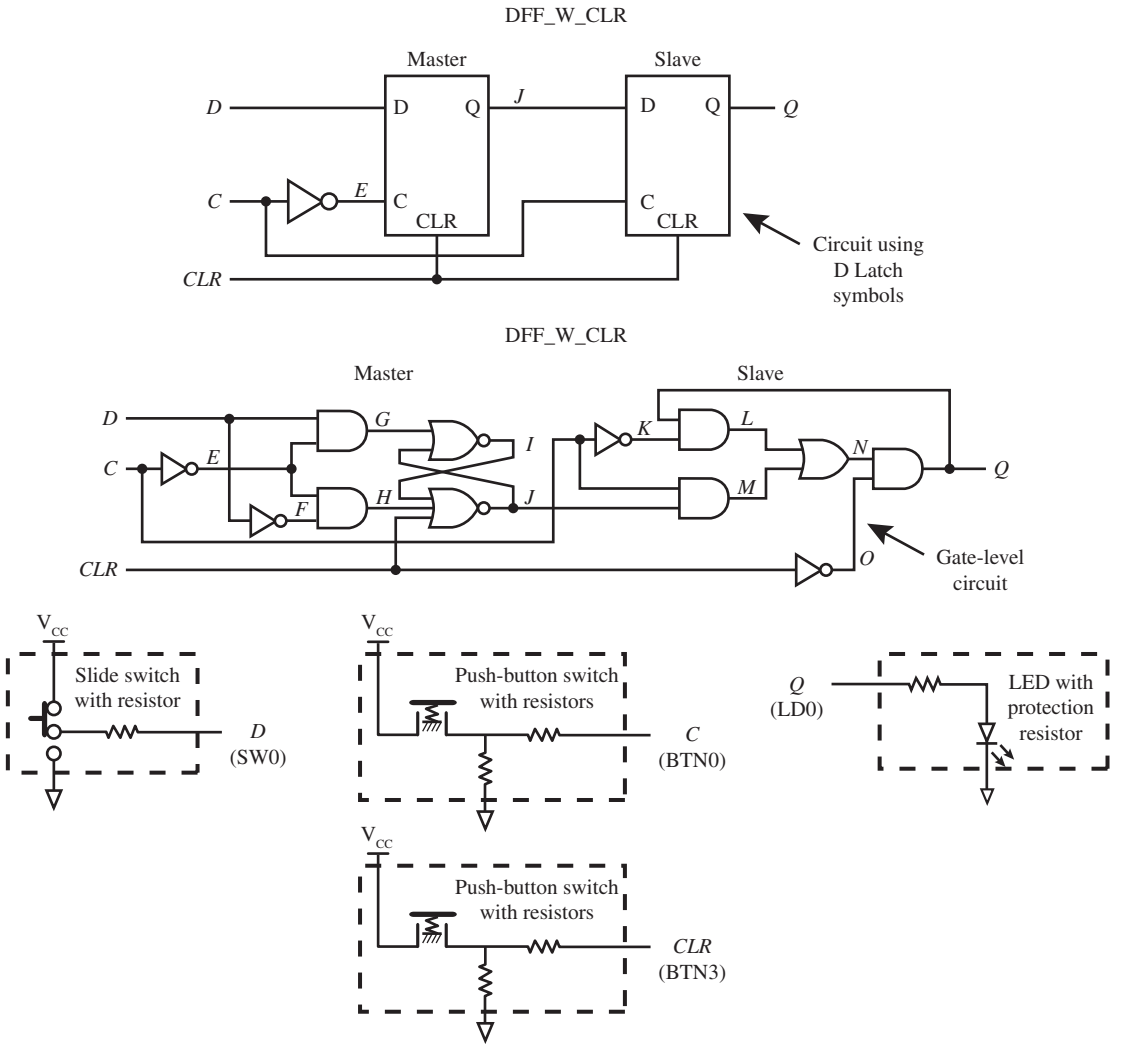


FIGURE E5A.2 Annotated schematic for a D flip-flop circuit with a CLR input

a post-route simulation for the design to verify that it follows the characteristic table for the D latch as provided in the text in Chapter 5, Section 5.5.2. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If the output of your D latch does not follow the characteristic table, then you know that the VHDL code for your D latch has an error. You must find the error or errors and fix them.

2. Complete the design cycle for your D latch by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
3. Check to see if your D latch works in hardware. Push and hold down the push-button switch BTN0. While the push-button switch is being held down, push the slide switch SW0 forward then back and observe the single LED output LD0. Observe that the output LD0, which represents *Q*, follows the value of SW0, which represents D. When you release the

push-button switch observe the last value that you provided for the *D* input is captured by *Q*. When *C* is 0 (push-button switch not pressed), observe that *D* input has no effect on the output *Q*.

4. Fill in the following PS/NS table for your D latch design. First use the *C* and *D* inputs to establish the present-state value of *Q* in the first row of the table. Next change the value of *C* and *D* to the values required for the first row of the table and record the next-state value of *Q*—that is, Q^+. (Note: In the table when *C* is 0, the push-button switch is not pressed, and when *C* is 1, the push-button switch is pressed. Use this procedure to fill in each row of the table. Draw the logic symbol for the D latch represented by the PS/NS table that you filled in. Use the PS/NS table to obtain the characteristic table for the D latch.

C	*D*	*Q*	Q^+
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

5. Create a new project named DFF_W_CLR, and write complete VHDL code for the D flip-flop with a *CLR* input, shown in Figure E5A.2. Use a dataflow design style with Boolean equations. Obtain a post-route simulation for the design to verify that it follows the characteristic table for the D flip-flop as provided in the text in Chapter 5, Section 5.6. If the output of your D flip-flop does not follow the characteristic table, then you know that the VHDL code for your D flip-flop has an error. You must find the error or errors and fix them.
6. Complete the design cycle for your D flip-flop with a *CLR* input, by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. Check to see if your D flip-flop with a CLR input works in hardware. Observe the single LED output LD0 for the following conditions: Move slide switch SW0 forward and press and release BTN0; then move slide switch SW0 back and press and release BTN0. Observe that the output LD0, which represents *Q*, follows the value of SW0, which represents *D*, each time BTN0 is pressed, which represents *C*. You may release BTN0 any time after you press it. The value of *D* is stored by the D flip-flop when BTN0 is pressed. Observe what happens when the value of the slide switch SW0 is changed and BTN0 is not pressed. You will observe that this action has no effect on the output LD0. Also observe that any time the output LD0 is 1 and BNT3 is pressed that the output LD0 turns off or *Q* is cleared to 0. If your D flip-flop circuit does not act in this manner, you have a problem that you must find and fix.
8. Fill in the following PS/NS table for your D flip-flop with a *CLR* input. First use the *CLR*, *C*, and *D* inputs to establish the present-state value of *Q* in the first row of the table. Next change the value of *CLR*, *C*, and *D* to the values required for the first row of the table and record the next-state value of *Q*—that is, Q^+. The symbol ↑ means to press BTN0, which represents *C*, to generate a rising edge, or $0 \rightarrow 1$ transition. You may release BTN0 any time

after you press it. Use this procedure to fill in each row of the table. Draw the logic symbol for your D flip-flop with a *CLR* input represented by the PS/NS table that you filled in. Use the PS/NS table to obtain the characteristic table for the D flip-flop with a *CLR* input.

CLR	*C*	*D*	*Q*	Q^+
1	×	×	×	
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	↑	0	0	
0	↑	0	1	
0	↑	1	0	
0	↑	1	1	

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working designs and get them signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working designs. Your final working designs for this experiment are task 3 (D Latch) and task 7 (D flip-flop with a *CLR* input).
2. Include the complete VHDL code for your D latch design.
3. Include a printout of the post-route simulation waveform diagram for your D latch design.
4. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your D latch design.
5. Include the filled in PS/NS table for the D latch.
6. Include the logic symbol for the D Latch.
7. Include the characteristic table for the D latch.
8. Include the complete VHDL code for your D flip-flop with a *CLR* input design.
9. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your D flip-flop with a *CLR* input design.
10. Include the filled in PS/NS table for D flip-flop with a *CLR* input.
11. Include the logic symbol for D flip-flop with a *CLR* input.
12. Include the characteristic table for D flip-flop with a *CLR* input.
13. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
14. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 5B: DESIGNING AND TESTING AN 8-BIT REGISTER AND A D FLIP-FLOP WITH A PRE INPUT

1. Learning Objectives

In this experiment, you will practice working with an 8-bit register and a D flip-flop with a *PRE* (PRESET) input. First, you will learn how to design an 8-bit register and run a post-route simulation for the design to verify that it works. Second, you will download and test your design

in hardware to verify that the 8-bit register works. Third, you will learn how to design a positive edge-triggered D flip-flop with a PRE input and run a post-route simulation for the design to verify that it works. Fourth, you will download and test your design in hardware to verify that the D flip-flop works. This is summarized as follows:

1. Design an 8-bit register and run a post-route simulation for the design to verify that it works.
2. Download and test your design in hardware to verify that the 8-bit register works.
3. Design a positive edge-triggered D flip-flop with a *PRE* (PRESET) input and run a post-route simulation for the design to verify that it works.
4. Download and test your design in hardware to verify that that the D flip-flop works.

2. 8-bit Register with Inputs and Output

Figure E5B.1 shows an annotated schematic for an 8-bit register circuit with inputs and outputs and a D latch circuit.

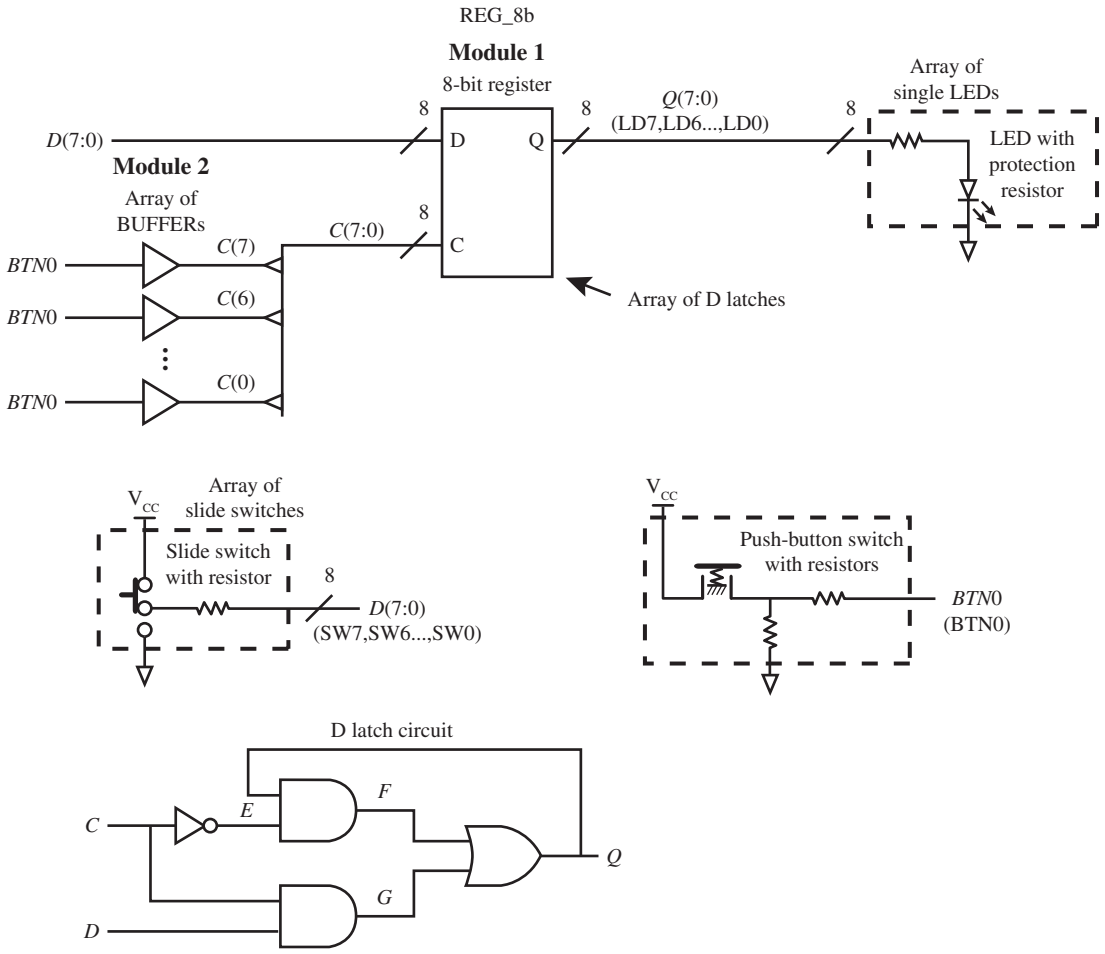


FIGURE E5B.1 Annotated schematic for an 8-bit register circuit and D latch circuit

Notice that an array of slide switch SW7, SW6. . . , SW0 provide the *D* input values, one push-button switch BTN0 provides the *C* input value, and an array of single LED outputs LD7, LD6, . . ., LD0 provide the *Q* output values.

3. D Flip-Flop with a PRE Input with Inputs and Output

Figure E5B.2 shows an annotated schematic for a D flip-flop circuit with a *PRE* input.

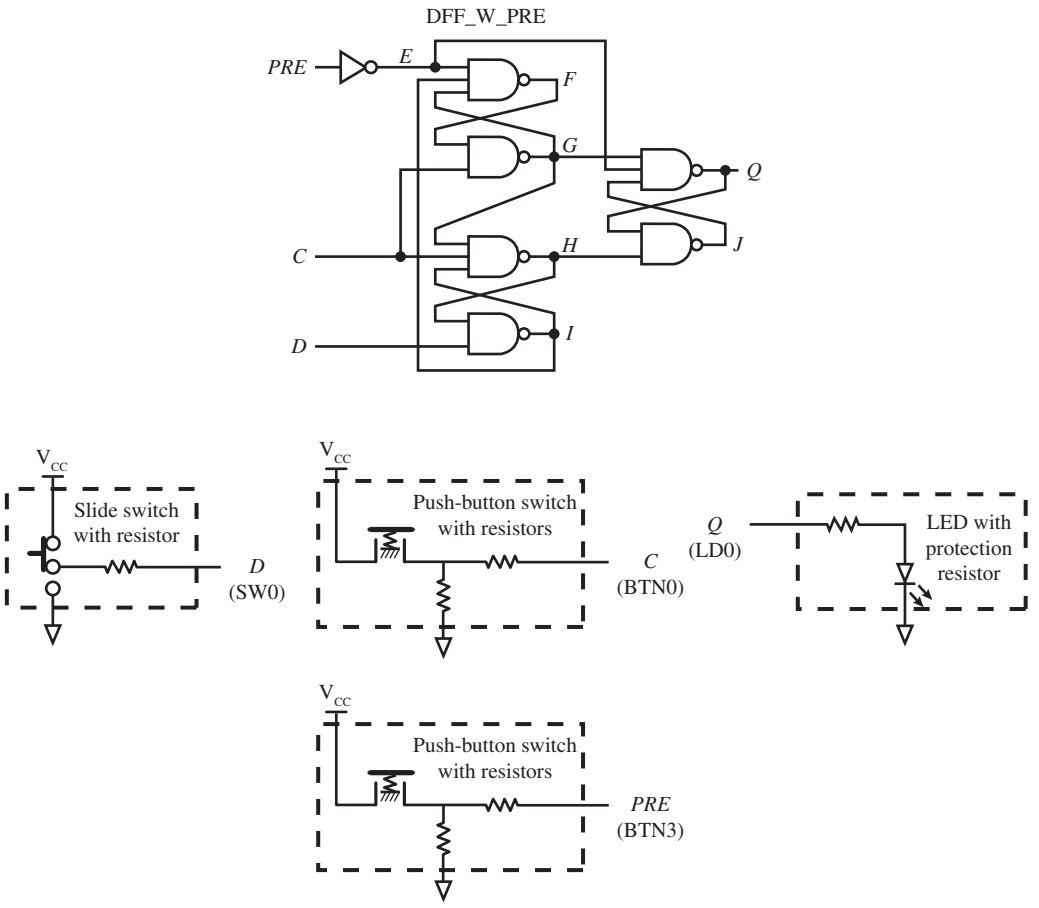


FIGURE E5B.2
Annotated schematic for a D flip flop with a PRE input

Notice that one slide switch SW0 provides the *D* input value, one push-button switch BTN0 provides the *C* input value, one push-button switch BTN3 provides the *PRE* input value, and a single LED output LD0 provides the *Q* output value.

Recommended Pre-Lab:

1. Tasks 1 and 5.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Create a new project named REG_8b, and write complete VHDL code for the 8-bit register circuit shown in Figure E5B.1. To generate the 8-bit register, use an array of eight D latch circuits. To create the array, simply declare the port names and the internal signals as vectors for the D latch circuit in Figure E5B.1. Use a dataflow design style with Boolean equations. Obtain a post-route simulation for the design to verify that it follows the characteristic table for each of the D latch circuits in the array as provided in the text in Chapter 5, Section 5.5.2.

[Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If the output of your 8-bit register does not follow the characteristic table for each of the D latch circuits in the array, then you know that the VHDL code for your 8-bit register has an error. You must find the error or errors and fix them.

2. Complete the design cycle for your 8-bit register by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
3. Check to see if your 8-bit register works in hardware. Select an 8-bit value by pushing forward or pulling back the slide switches SW7, SW6. . ., SW0. Press and release BTN0, which represents *C*. This captures or stores that value of the slide switches. After you release the push-button switch BTN0, you should observe that you can change the slide switches and this does not change the previous value that was captured. If your design for the 8-bit register doesn't work in this manner, then you know that the VHDL code has an error. You must find the error or errors and fix them.
4. Fill in the following truth table to confirm that you can capture the following binary values at the slide switch inputs when you push and then release push-button switch BTN0:

Switch settings								Value captured							
SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0	LD7	LD6	LD5	LD4	LD3	LD2	LD1	LD0
1	0	1	0	1	0	1	0								
1	1	0	0	1	1	0	0								
1	1	1	0	0	0	1	1								
1	1	1	1	0	0	0	0								
0	0	1	1	0	0	1	1								
0	1	0	1	0	1	0	1								
0	0	0	0	1	1	1	1								

5. Create a new project named DFF_W_PRE, and write complete VHDL code for the D flip-flop with a *PRE* input, shown in Figure E5B.2. Use a dataflow design style with Boolean equations. Obtain a post-route simulation for the design to verify that it follows the characteristic table for the D flip-flop as provided in the text in Chapter 5, Section 5.6.1. If the output of your D Flip-Flop does not follow the characteristic table, then you know that the VHDL code for your D flip-flop has an error. You must find the error or errors and fix them.
6. Complete the design cycle for your D flip-flop with a *PRE* input, by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. Check to see if your D flip-flop with a *PRE* input works in hardware. Observe the single LED output LD0 for the following conditions: Move slide switch SW0 forward and press and release BTN0; then move slide switch SW0 back and press and release BTN0.

Observe that the output LD0, which represents *Q*, follows the value of SW0, which represents *D*, each time BTN0 is pressed, which represents *C*. You may release BTN0 any time after you press it. The value of *D* is stored by the D flip-flop when BTN0 is pressed. Observe what happens when the value of the slide switch SW0 is changed and BTN0 is not pressed. You will observe that this action has no effect on the output LD0. Also observe that any time the output LD0 is 0 and BNT3 is pressed, the output LD0 turns on or *Q* is set to 1. If your D flip-flop circuit does not act in this manner, you have a problem that you must find and fix.

8. Fill in the following PS/NS table for your D flip-flop with a *PRE* input. First use the *PRE*, *C*, and *D* inputs to establish the present-state value of *Q* in the first row of the table. Next change the value of *PRE*, *C*, and *D* to the values required for the first row of the table and record the next-state value of *Q*, i.e., Q^+. The symbol ↑ means to press BTN0, which represents *C*, to generate a rising edge, or $0 \rightarrow 1$ transition. You may release BTN0 any time after you press it. Use this procedure to fill in each row of the table. Draw the logic symbol for your D flip-flop with a *PRE* input represented by the PS/NS table that you filled in. Use the PS/NS table to obtain the characteristic table for the D flip-flop with a *PRE* input.

PRE	*C*	*D*	*Q*	Q^+
1	×	×	×	
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	↑	0	0	
0	↑	0	1	
0	↑	1	0	
0	↑	1	1	

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working designs and get them signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working designs. Your final working designs for this experiment are task 3 (8-bit register) and task 7 (D flip-flop with a *PRE* input).
2. Include the complete VHDL code for your 8-bit register design.
3. Include a printout of the post-route simulation waveform diagram for your 8-bit register design.
4. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your 8-bit register design.
5. Include the filled-in truth table for the 8-bit register.
6. Include the logic symbol for the 8-bit register.
7. Include the characteristic table for the 8-bit register.
8. Include the complete VHDL code for your D flip-flop with a *PRE* input design.
9. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your D Flip-Flop with a *PRE* input.

10. Include the filled-in truth table for the D flip-flop with a *PRE* input.
11. Include the logic symbol for the D flip-flop with a *PRE* input.
12. Include the characteristic table for the D flip-flop with a *PRE* input.
13. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
14. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 6A: DESIGNING AND TESTING A SIMPLE COUNTER SYSTEM—A ONE-HOT UP COUNTER WITH 8 BITS

1. Learning Objectives

In this experiment, you will practice designing and testing a simple counter system—that is, a one-hot up counter with 8 bits. First you will learn how to design a one-hot up counter with 8 bits and simulate it to verify that it works. Second, you will learn how to design a frequency divider to divide the frequency of 50 MHz down to approximately 3 Hz and verify that the design works by observing the output supplied to a single LED. Third, you will learn how to combine the frequency divider and the one-hot up counter via a flat design approach. Fourth, you will download and test your design in hardware to verify that the system works. This is summarized as follows:

1. Design a one-hot up counter with 8 bits and simulate it to verify that the design works.
2. Design a frequency divider to divide the frequency of 50 MHz down to approximately 3 Hz and verify that the design works via a single LED.
3. Combine the frequency divider and the one-hot up counter to form a system via a flat design approach.
4. Download and test your design in hardware to verify that the system works.

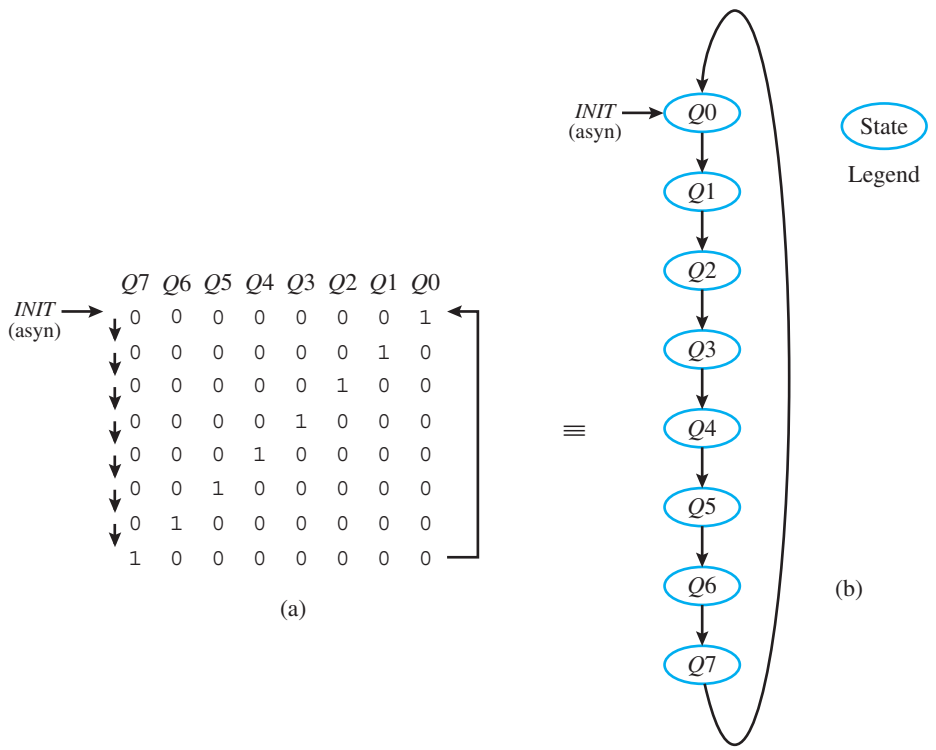


FIGURE E6A.1 Design specification for a one-hot up counter: (a) counting sequence diagram; (b) equivalent state diagram

2. One-Hot Up Counter with 8 Bits

The design specification for a one-hot up counter with 8 bits is shown in Figure E6A.1. Figure E6A.1a shows the counting sequence diagram, and Figure E6A.1b shows an equivalent state diagram for the one-hot up counter. Each flip-flop output that is turned on represents a state for a one-hot counter.

Figure E6A.2 shows a template for a one-hot up counter with 8 bits.

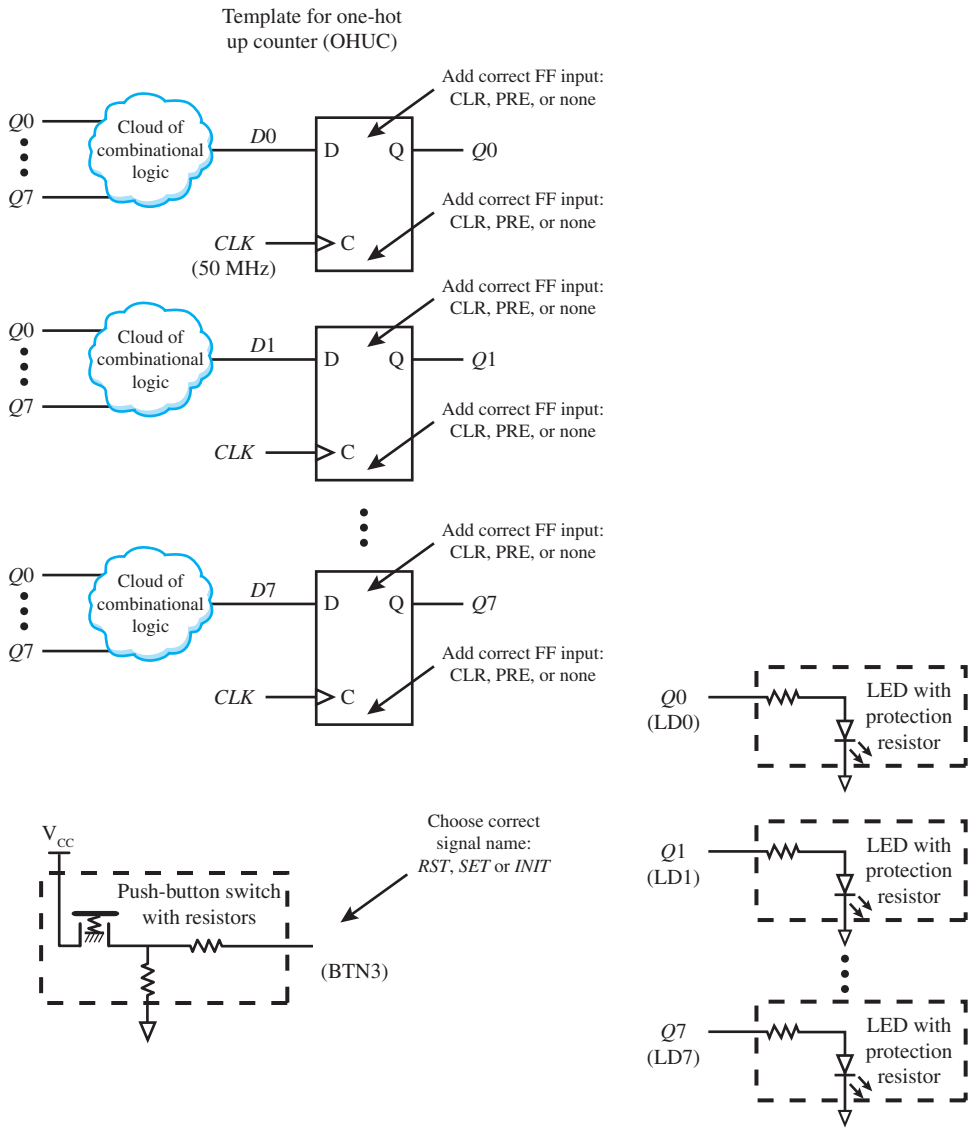


FIGURE E6A.2
Template for a one-hot up counter with 8 bits

Notice in Figure E6A.2 that the push-button switch BTN3 provides the input value and eight separate single LED outputs LD0 through LD7 provide the *Q* output values. You must determine the eight clouds of combinational logic, the correct FF (flip-flop) inputs for clearing and/or presetting the FFs, and the correct signal name for the push-button switch based on the design specification.

3. Frequency Divider with Input and Output

Figure E6A.3 shows a logic symbol for a frequency divider (FD) that will divide a 50-MHz frequency down to approximately 3 Hz connected to a single LED.

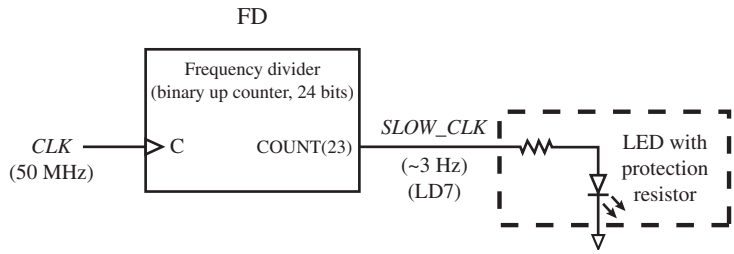


FIGURE E6A.3 Frequency divider connected to a single LED

For this design, a binary counter with 24 bits that range from 23 down to 0 is used to divide the frequency of the clock from 50 MHz down to $50/2^{24}$ MHz or 2.98 Hz. The signal name for the output of the frequency divider is *COUNT*, and it is an internal signal.

Recommended Pre-Lab:

1. Tasks 1 and 2.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Obtain the design for the one-hot up counter specification in Figure E6A.1 using the algorithmic equation method. First, use the Set OR Hold 1 equation to obtain the D excitation equations for the one-hot up counter in Figure E6A.1. Draw the circuit for the one-hot up counter using the template in Figure E6A.2. Create a new project named OHUC, and write complete VHDL code for the one-hot up counter. Make the *Q* output a std_logic_vector data type. Write the VHDL code using a dataflow design style with an aggregate. (Hint: See Chapter 6, Section 6.6, Listing 6.4, which shows how to use an aggregate.)
2. Simulate the design for the one-hot up counter to verify that it follows the counting sequence diagram or the equivalent state diagram in Figure E6A.1. [Note: see Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If your design does not follow the counting sequence diagram or the equivalent state diagram, then you know that the VHDL code has an error. You must find the error or errors and fix them.
3. Create a new project named FD, and write complete VHDL code for the logic symbol FD connected to a single LED as shown in Figure E6A.3. [Note: Do not use FD as the name of the design entity because this name is reserved as a keyword in the Xilinx library. FD1 will work if you want to use this name.) Write the VHDL code using a behavioral design style via the Arithmetic Method. (Hint: See Chapter 6, Section 6.8, Listing 6.6.)
4. Complete the design cycle for your FD circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
5. Check to see if your FD circuit works in hardware. Verify that LD7 blinks at a frequency of ~3 Hz. If your design does not perform in this manner, you have made a mistake that you must find and correct.
6. Use a flat design approach to combine the frequency divider and one-hot up counter so that the one-hot up counter uses the *SLOW_CLK* signal provided by the frequency divider as its

clock input. Use the input (push-button switch) and outputs (single LEDs) for your system design as shown below the template for the one-hot up counter in Figure E6A.2. (Hint: See Section 2.4.5 in Chapter 2 or the end of Section 4.8 in Chapter 4 for a discussion of a flat design approach.) First, create a new project named FD_OHUC and write the complete VHDL code for the system. You can use the architecture parts of the VHDL code for the frequency divider and the one-hot up counter as modules 1 and 2 in your flat design.

7. After you create the new project, click Project, then click Add Copy of Source to add the VHDL code—that is, the .vhd files, for FD and OHUC to your new project FD_OHUC. Open up the .vhd files FD and OHUC in turn and copy and paste the architecture parts you need in your flat design. After you copy and paste the architecture parts you need, delete the .vhd files for FD and OHUC. You must remember to include *SLOW_CLK* as an internal signal in the architecture of your flat design and also to use *SLOW_CLK* as the clock signal that drives your one-hot up counter.
8. Complete the design cycle for your FD_OHUC design by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
9. Check to see if your FD_OHUC design works in hardware. Verify that LD7 down to LD0 count in the one-hot sequence 00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000 and repeats, where 0 represents an LED that is off and 1 represents an LED that is on. When push button BTN3 is pressed and held down LD0 goes to 1 and all the other LED go to 0 until the push button is released. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 9 (FD_OHUC design).
2. Include the D excitation equations for your one-hot up counter.
3. Include your circuit drawing for the one-hot up counter.
4. Include your complete VHDL code for your OHUC design.
5. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your OHUC design.
6. Include a printout of the simulation waveform diagram for the OHUC design.
7. Include the complete VHDL code for your FD design.
8. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your FD design.
9. Include the complete VHDL code for your FD_OHUC design.
10. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your FD_OHUC design.
11. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
12. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 6B: DESIGNING AND TESTING A SIMPLE COUNTER SYSTEM—A GRAY CODE COUNTER WITH 2 BITS

1. Learning Objectives

In this experiment, you will practice designing and testing simple counters—that is, a Gray code counter. First, you will learn how to design a Gray code counter with 2 bits and simulate it to verify that it works. Second, you will learn how to design a frequency divider to divide the frequency of 50 MHz down to approximately 1 Hz and verify that the design works by observing the output supplied to a single LED. Third, you will learn how to combine the frequency divider and the Gray code counter to form a system via a hierarchal design approach. Fourth, you will download and test your design in hardware to verify that the system works. This is summarized as follows:

1. Design a Gray-code counter with 2 bits and simulate it to verify that the design works.
2. Design a frequency divider to divide the frequency of 50 MHz down to approximately 1 Hz and verify that the design works via a single LED.
3. Combine the frequency divider and the Gray code counter to form a system via a hierarchal design approach.
4. Download and test your design in hardware to verify that the system works.

2. Gray Code Counter with 2 Bits

The design specification for a Gray code counter with 2 bits is shown in Figure E6B.1. Figure E6B.1a shows the counting sequence diagram, and Figure E6B.1b shows an equivalent state diagram for the Gray code counter.

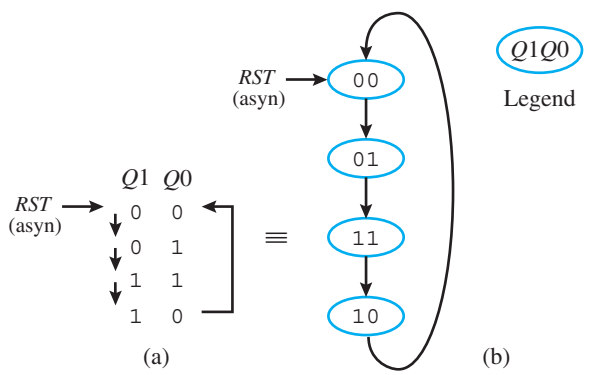


FIGURE E6B.1 Design specification for a Gray code counter: (a) counting sequence diagram; (b) equivalent state diagram

Figure E6B.2 shows the template for a synchronous Gray code counter with 2 bits.

Notice in Figure E6B.2 that the push-button switch BTN0 provides the input value, and two separate single LED outputs LD0 and LD1 provide the *Q* output values. You must determine the two clouds of combinational logic, the correct FF (flip-flop) inputs for clearing and/or presetting the FFs, and the correct signal name for the push-button switch based on the design specification.

3. Frequency Divider with Input and Output

Figure E6B.3 shows a logic symbol for a frequency divider (FD) that will divide a 50-MHz frequency down to approximately 1 Hz connected to a single LED.

For this design, a binary counter with 26 bits that range from 25 down to 0 is used to divide the frequency of the clock from 50 MHz down to $50/2^{26}$ MHz or 0.7451 Hz. The signal name for the output of the frequency divider is *COUNT*, and it is an internal signal.

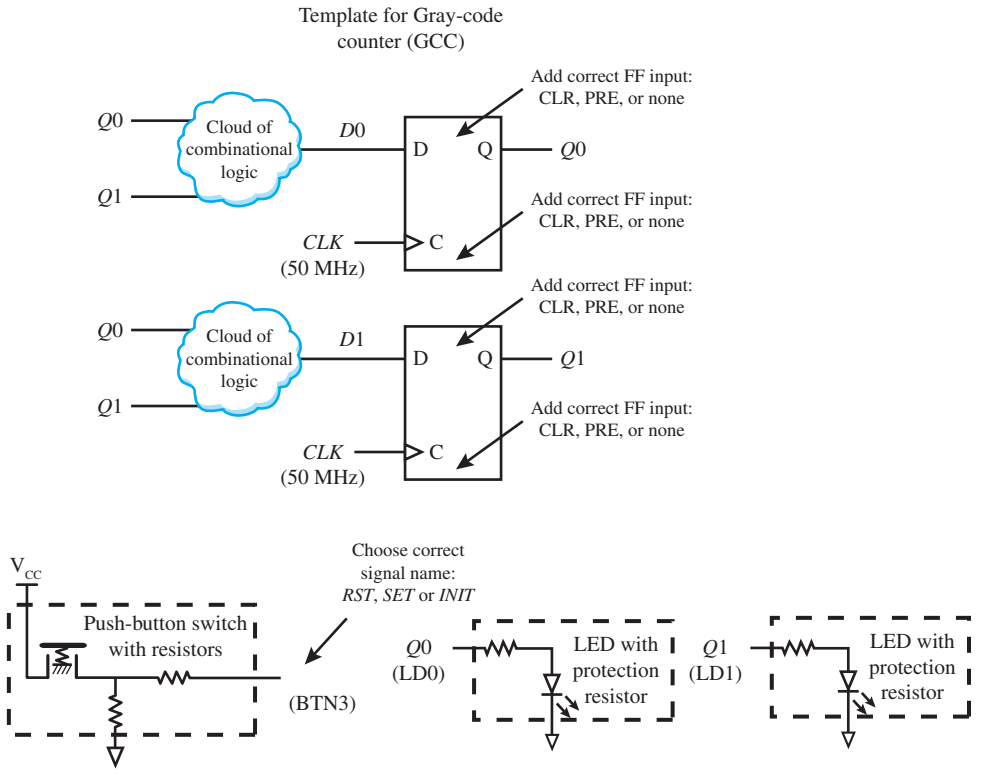


FIGURE E6B.2 Template for a Gray code counter with 2 bits

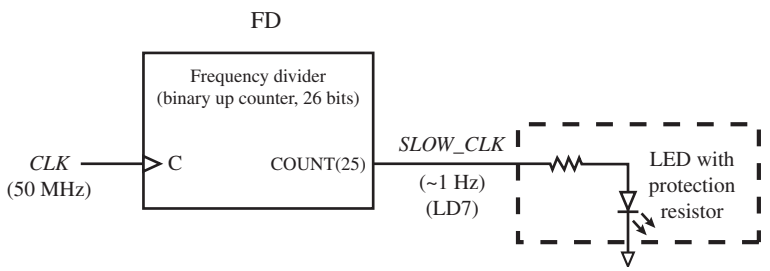


FIGURE E6B.3 Frequency divider connected to a single LED

Recommended Pre-Lab:

1. Task 1.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Obtain the design for the Gray code counter specification in Figure E6B.1 using the algorithmic equation method. First, use the Set OR Hold 1 equation to obtain the D excitation equations for the Gray code counter in Figure E6B.1. Draw the circuit for the Gray code counter using the template in Figure E6B.2. Create a new project named GCC, and write

complete VHDL code for the Gray code counter. Write the VHDL code using a behavioral design style with either one or two processes. Simulate the design to verify that it follows the counting sequence diagram or the equivalent state diagram in Figure E6B.1. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If your design does not follow the counting sequence diagram or the equivalent state diagram, then you know that the VHDL code has an error. You must find the error or errors and fix them.

2. Create a new project named FD, and write complete VHDL code for the logic symbol FD connected to a single LED as shown in Figure E6B.3. (Note: Do not use FD as the name of the design entity because this name is reserved as a keyword in the Xilinx library. FD1 will work if you want to use this name.) Write the VHDL code using a behavioral design style via the arithmetic method. (Hint: See Chapter 6, Section 6.8, Listing 6.6.)
3. Complete the design cycle for your FD circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
4. Check to see if your FD circuit works in hardware. Verify that LD7 blinks at a frequency of ~1 Hz. If your design does not perform in this manner, you have made a mistake that you must find and correct.
5. Use a hierarchal design approach to combine the frequency divider and Gray code counter so that the Gray code counter uses the *SLOW_CLK* signal provided by the frequency divider as its clock input. Use the input (push-button switch) and outputs (single LEDs) for your system design as shown below the template for the Gray code counter in Figure E6B.2. (Hint: See Chapter 4, Section 4.8 Structural Design Style.) First, create a new project named FD_GCC and write the complete VHDL code for the system. You can use the complete VHDL code for the frequency divider and the Gray code counter as the definitions for components 1 and 2 in your hierarchal design. After you create the new project, click Project, then click Add Copy of Source to add the VHDL code—that is, the .vhd files, for FD and GCC to your new project FD_GCC.
6. Complete the design cycle for your FD_GCC design by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. Check to see if your FD_GCC design works in hardware. Verify that LD1 and LD0 count in the Gray code sequence 00, 01, 11, 10 and repeats, where 0 represents an LED that is off and 1 represents an LED that is on. When push button BTN3 is pressed and held down LD1 and LD0 go to 00 until the push button is released. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 7 (FD_GCC design).

2. Include the D excitation equations for your Gray code counter.
3. Include your circuit drawing for the Gray code counter.
4. Include your complete VHDL code for your GCC design.
5. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your GCC design.
6. Include a printout of the simulation waveform diagram for the GCC design.
7. Include the complete VHDL code for your FD design.
8. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your FD design.
9. Include the complete VHDL code for your FD_GCC design.
10. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your FD_GCC design.
11. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
12. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 6C: DESIGNING AND TESTING A SIMPLE NONCONVENTIONAL COUNTER SYSTEM—A ROBOT EYE CIRCUIT

1. Learning Objectives

In this experiment, you will practice designing and testing a simple nonconventional counter system—that is, the design for a robot eye circuit. First, you will learn how to design a robot eye circuit (REC) and simulate it. Second, you will learn how to design an exact frequency divider to divide the frequency of 50 MHz down to exactly 8 Hz and verify that the design works by observing the output supplied to a single LED. Third, you will learn how to design a robot eye system that contains the exact frequency divider and the robot eye circuit via a flat design approach. Fourth, you will download and test your design to verify that the system works. This is summarized as follows:

1. Design a robot eye circuit and simulate it.
2. Design an exact frequency divider to divide the frequency of 50 MHz down to exactly 8 Hz and verify that the design works via a single LED.
3. Design a robot eye system that contains the exact frequency divider and the robot eye circuit via a flat design approach.
4. Download and test your design in hardware to verify that the system works.

2. Robot Eye Circuit

The design specification for a robot eye circuit is shown in Figure E6C.1. Figure E6C.1a shows the state sequence diagram, and Figure E6C.1b shows an equivalent state diagram for the robot eye circuit.

Notice that this is a **nonconventional counter** because it has repeating states.

To differentiate between the repeating state values of 0010 and 0100, you must add an additional state variable $Q4$ (an additional flip-flop $Q4$) to the state sequence diagram to remove the repeating state values.

Figure E6C.2 shows an annotated logic symbol for the robot eye circuit.

Notice in Figure E6C.2 that Q is an internal signal and QN is a port signal.

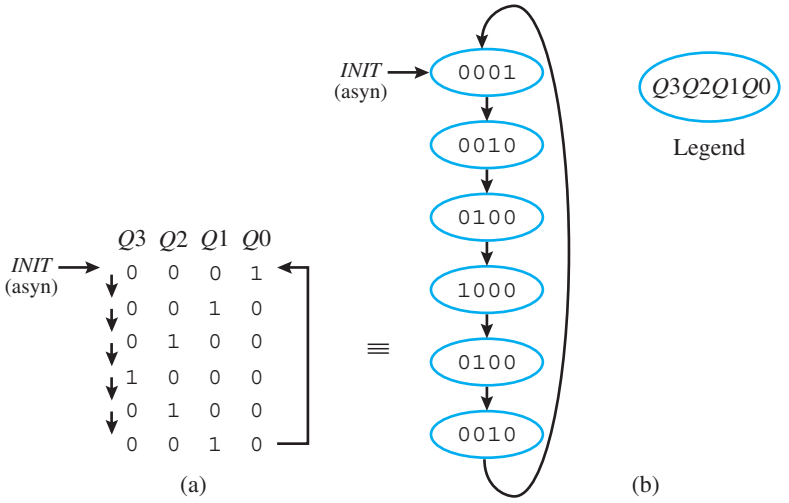


FIGURE E6C.1 Design specification for a robot eye circuit: (a) state sequence diagram; (b) equivalent state diagram

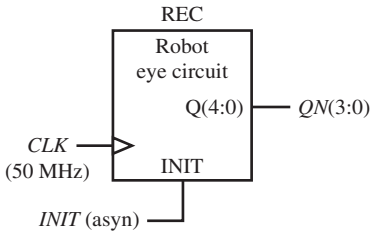
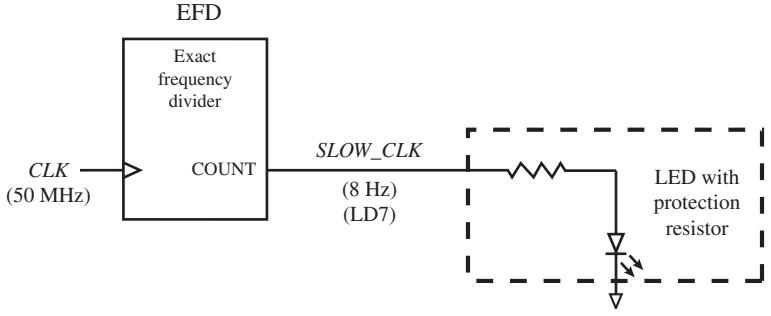


FIGURE E6C.2 Annotated logic symbol for the robot eye circuit

3. Exact Frequency Divider with Input and Output

Figure E6C.3 shows an annotated logic symbol for an exact frequency divider (EFD) that will divide a 50-MHz frequency down to exactly 8 Hz connected to a single LED.

FIGURE E6C.3 Exact frequency divider connected to a single LED



The ratio of 50 MHz and 8 Hz is 6,250,000. To provide an exact frequency divider with an output of 8 Hz, use the arithmetic method to count the number of clock cycles that must occur for half the period—that is, $T/2 = (1/f)/2 = (1/(8\text{ Hz}))/2 = 0.0625\text{s}$ (3,125,000 clock cycles) via an internal signal named *COUNT*. Toggling the signal *SLOW_CLK* between 1 and 0 at 0.0625s internals will produce the frequency 8 Hz.

4. Robot Eye System

Figure E6C.4 shows an annotated schematic for a robot eye system. The system consists of the three components or modules EFD, REC, and NOT_ARRAY.

Recommended Pre-Lab:

1. Task 1.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

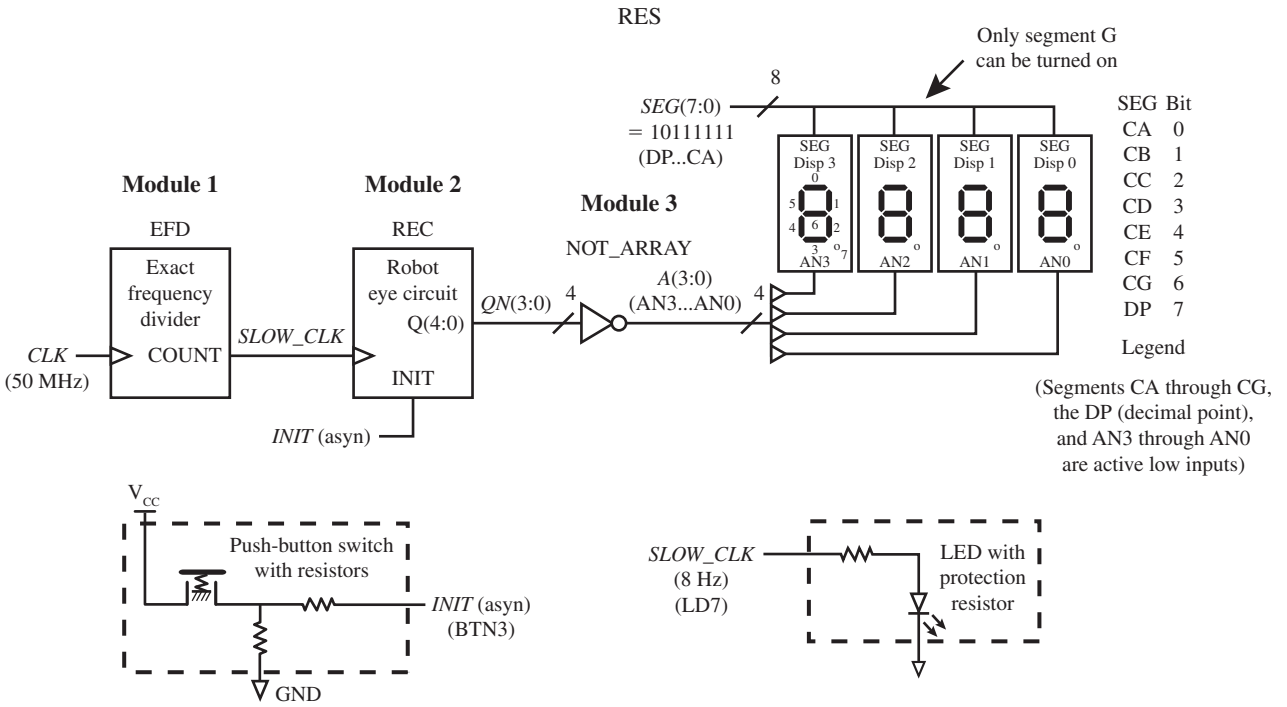


FIGURE E6C.4 Annotated schematic for a robot eye system

Tasks:

1. Obtain the design for the robot eye circuit design specification in Figure E6C.1 using the PS/NS Tabular Method. First, add the required minimum number of FFs (flip-flops) to the design specification in Figure E6C.1 to modify the state sequence diagram and also the equivalent state diagram to remove the repeating state values. (Hint: See Chapter 6, Section 6.10.) Create a new project named REC, and write complete VHDL code for the robot eye circuit. Write the VHDL code using a behavioral design style with an if statement and a case statement. Simulate the design to verify that it follows your modified state sequence diagram and modified equivalent state diagram. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If your design does not follow your modified state sequence diagram and your modified equivalent state diagram, then you know that the VHDL code has an error. You must find the error or errors and fix them.
2. Create a new project named EFD, and write complete VHDL code for the logic symbol EFD connected to a single LED as shown in Figure E6C.3. Write the VHDL code using a behavioral design style via the arithmetic method. (Hint: See Chapter 6, Section 6.8, Listing 6.7.)
3. Complete the design cycle for your EFD circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
4. Check to see if your EFD circuit works in hardware. Verify that LD7 blinks at a frequency of about 8 Hz. If your design does not perform in this manner, you have made a mistake that you must find and correct.

5. Use a flat design approach to obtain the robot eye system in Figure E6C.4. (Hint: See Section 2.4.5 in Chapter 2 or the end of Section 4.8 in Chapter 4 for discussion of a flat design approach.) First, create a new project named RES, and write the complete VHDL code for the system. You can use the architecture parts of the VHDL code for the exact frequency divider and the robot eye circuit as modules 1 and 2 in your flat design. After you create the new project, click Project, then click Add Copy of Source to add the VHDL code—that is, the .vhd files, for EFD and REC to your new project RES. Open up the .vhd files EFD and REC and copy and paste the architecture parts you need in your flat design. After you copy and paste the architecture parts you need, delete the .vhd files for EFD and REC from your RES project. You must remember to include *SLOW_CLK* as an internal signal in the architecture of your flat design and also to use *SLOW_CLK* as the clock signal that drives your robot eye circuit. For module 3 simply add the architecture part you need for the NOT_ARRAY. (Note: Include SEG as an output in the NOT_ARRAY module.)
6. Complete the design cycle for your RES design by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. Check to see if the RES system works in hardware. Verify that the center segment of each of the four 7-segment displays generate the roving eye effect at the rate of 8 Hz, as observed by the single blinking LED at LD7. When push button BTN3 is pressed and held down, only the center segment of the 7-segment display on the extreme right is turned on, which initializes the robot eye system to 0001 until the push button is released. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design(s). Your final working design for this experiment is task 8 (RES design).
2. Include your modified state sequence diagram and modified equivalent state diagram for your REC design.
3. Include the complete VHDL code for your REC design.
4. Include a printout of the simulation waveform diagram for your REC design.
5. Include the complete VHDL code for your EFD design.
6. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your EFD design.
7. Include the complete VHDL code for your RES design.
8. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your RES design.
9. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
10. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 6D: DESIGNING AND TESTING A SIMPLE NONCONVENTIONAL COUNTER—A SMILEY FACE CIRCUIT

1. Learning Objectives

In this experiment, you will practice designing and testing a simple nonconventional counter—that is, the design for a smiley face circuit. First, you will learn how to design a smiley face circuit (SFC) and simulate it to verify that it works. Second, you will learn how to design an exact frequency divider to divide the frequency of 50 MHz down to exactly 5 Hz and verify that the design works by observing the output supplied to a single LED. Third, you will learn how to design a smiley face system that contains the exact frequency divider and the smiley face circuit via a hierarchal design approach. Fourth, you will download and test your design in hardware to verify that the system works. This is summarized as follows:

1. Design a smiley face circuit (SFC) and simulate it to verify that it works.
2. Design an exact frequency divider to divide the frequency of 50 MHz down to exactly 5 Hz and verify that the design works via a single LED.
3. Design a smiley face system that contains the exact frequency divider and the smiley face circuit via a hierarchal design approach.
4. Download and test your design in hardware to verify that the system works.

2. Smiley Face Circuit

The design specification for a smiley face circuit is shown in Figure E6D.1. Figure E6D.1a shows the state sequence diagram, and Figure E6D.1b shows an equivalent state diagram for the smiley face circuit.

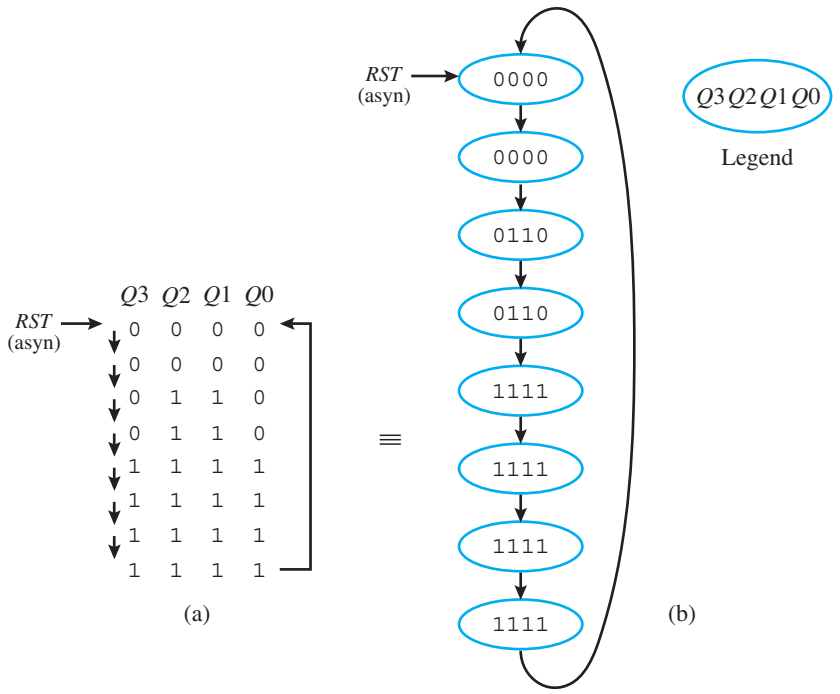


FIGURE E6D.1 Design specification for a smiley face circuit: (a) state sequence diagram; (b) equivalent state diagram

Notice that this is a **nonconventional counter** because it has repeating states. To differentiate between the repeating state values of 0000, 0110, and 1111, you must add two additional state

variables, $Q4$ and $Q5$, (two additional flip-flops $Q4$ and $Q5$) to the state sequence diagram (or the equivalent state diagram) to remove the repeating state values.

Figure E6D.2 shows an annotated logic symbol for the smiley face circuit.

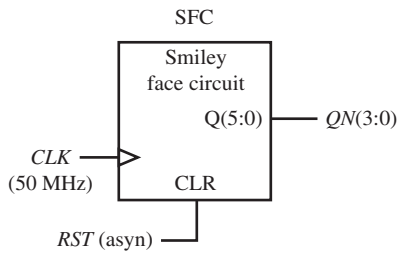


FIGURE E6D.2 Annotated logic symbol for the smiley face circuit

Notice in Figure E6D.2 that Q is an internal signal and QN is a port signal.

3. Exact Frequency Divider with Input and Output

Figure E6D.3 shows an annotated logic symbol for an exact frequency divider (EFD) that will divide a 50-MHz frequency down to exactly 5 Hz connected to a single LED.

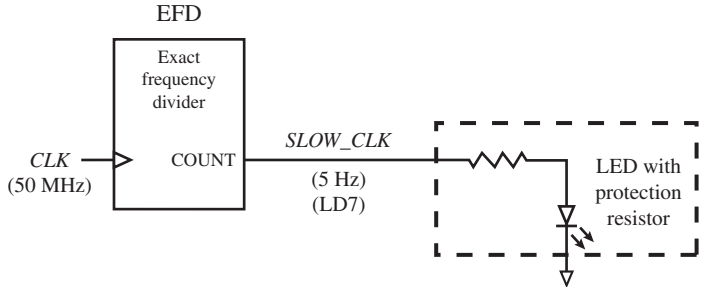


FIGURE E6D.3 Exact frequency divider connected to a single LED

The ratio of 50 MHz and 5 Hz is 10,000,000. To provide an exact frequency divider with an output of 5 Hz, use the arithmetic method to count the number of clock cycles that must occur for half the period—that is, $T/2 = (1/f)/2 = (1/(5\text{ Hz}))/2 = 0.1\text{s}$ (5,000,000 clock cycles) via an internal signal named *COUNT*. Toggling the signal *SLOW_CLK* between 1 and 0 at 0.1s intervals will produce the frequency 5 Hz.

4. Smiley Face System

Figure E6D.4 shows an annotated schematic for a smiley face system. The system consists of the three components or modules EFD, SFC, and NOT_ARRAY.

Recommended Pre-Lab:

1. Task 1.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Obtain the design for the smiley face circuit design specification in Figure E6D.1 using the PS/NS Tabular Method. First, add the required minimum number of FFs (Flip-Flops) to the design specification in Figure E6D.1 to modify the state sequence diagram and also the equivalent state diagram to remove the repeating state values. (Hint: See Chapter 6, Sec-

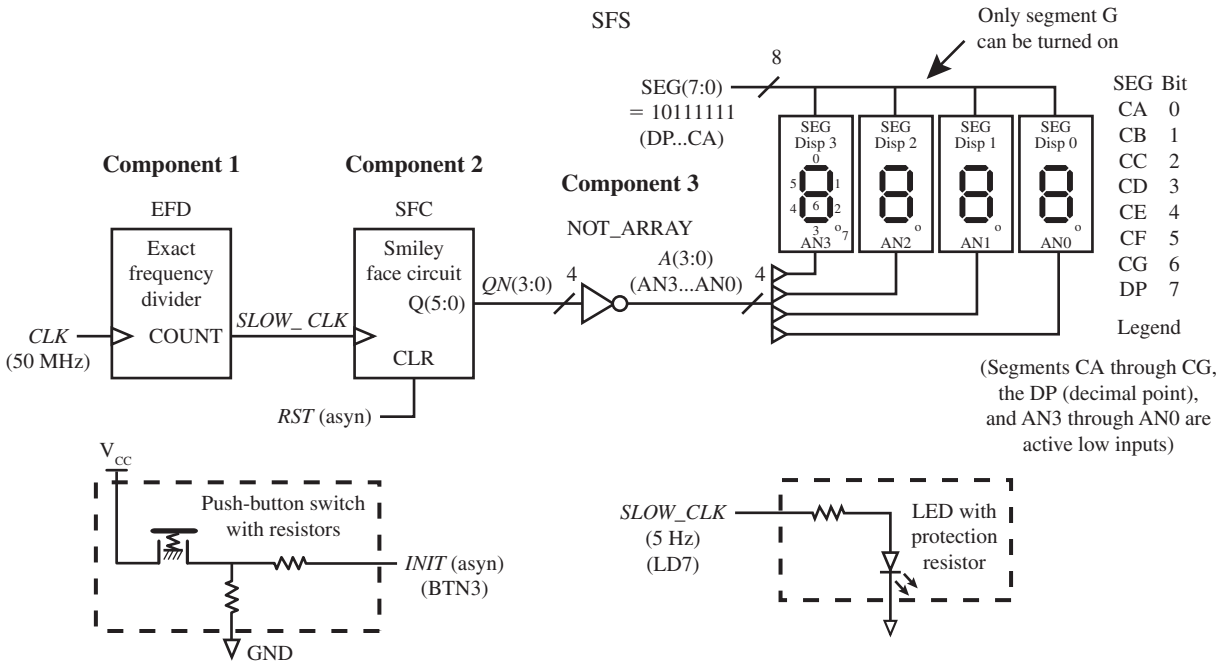


FIGURE E6D.4 Annotated schematic for a smiley face system

tion 6.10.) Create a new project named SFC, and write complete VHDL code for the smiley face circuit. Write the VHDL code using a behavioral design style with an if statement and a case statement. Simulate the design to verify that it follows your modified state sequence diagram and modified equivalent state diagram. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If your design does not follow your modified state sequence diagram and your modified equivalent state diagram, then you know that the VHDL code has an error. You must find the error or errors and fix them.

2. Create a new project named EFD, and write complete VHDL code for the logic symbol EFD connected to a single LED as shown in Figure E6D.3. Write the VHDL code using a behavioral design style via the arithmetic method. (Hint: See Chapter 6, Section 6.8, Listing 6.7.)
3. Complete the design cycle for your EFD circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
4. Check to see if your EFD circuit works in hardware. Verify that LD7 blinks at a frequency of about 5 Hz. If your design does not perform in this manner, you have made a mistake that you must find and correct.
5. Use a hierarchal design approach to obtain the smiley face system in Figure E6D.4. (Hint: See Chapter 4, Section 4.8, Structural Design Style.) First, create a new project named SFS and write the complete VHDL code for the system. You can use the complete VHDL code for the exact frequency divider and the smiley face circuit as the definitions for components 1 and 2 in your hierarchal design. After you create the new project, click Project, then click Add Copy of Source to add the VHDL code—that is, the .vhd files, for EFD and SFC to

your new project SFS. Create a project named NOT_ARRAY to define the NOT_ARRAY as component 3 in your hierarchal design for project SFS. (Note: Include SEG as an output in the NOT_ARRAY component.)

6. Complete the design cycle for your SFS design by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. Check to see if your SFS design works in hardware. Verify that the center segment of each of the four 7-segment displays generate the smiley face at the rate of 5 Hz, as observed by the single blinking LED at LD7. When push button BTN3 is pressed and held down, the 7-segment display is turned off, which resets or clears the smiley face system to 0000 until the push button is released. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design(s). Your final working design for this experiment is task 7 (SFS design).
2. Include your modified state sequence diagram and modified equivalent state diagram for the smiley face circuit.
3. Include the complete VHDL code for your SFC design.
4. Include a printout of the simulation waveform diagram for your SFC design.
5. Include the complete VHDL code for your EFD design.
6. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your EFD design.
7. Include the complete VHDL code for your SFS design.
8. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your SFS design.
9. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
10. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 7A: DESIGNING AND TESTING A SIMPLE ERROR DETECTION SYSTEM USING A FLAT DESIGN APPROACH

1. Learning Objectives

In this experiment, you will practice designing and testing a special combinational logic system. First, you will learn how to design a 4-bit parity generator (PG) for an even function and simulate it to verify that it works. Second, you will learn how to design a simple error detection system to transmit and receive odd parity with 4 data bits that will detect a single-bit error in the transmitted bits received at the destination, via a flat design approach. Third you will download and test your design in hardware to verify that the system works. This is summarized as follows:

1. Design a 4-bit parity generator (PG) for an even function and simulate it to verify that it works.
2. Design a simple error detection system to transmit and receive odd parity with 4 data bits that will detect a single-bit error in the transmitted bits received at the destination via a flat design approach.
3. Download and test your design in hardware to verify that the system works.

2. Design a 4-bit Parity Generator for an Even Function

Figure E7A.1 shows a logic symbol for a 4-bit parity generator for an even function.

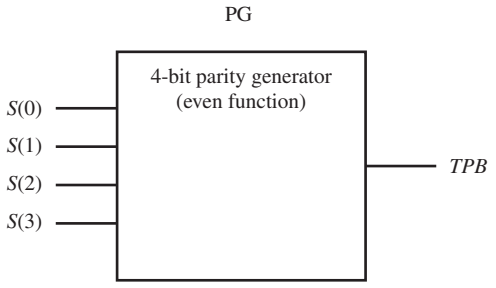


FIGURE E7A.1 4-bit parity generator (even function)

Note: The output *TPB* (transmitted parity bit) for this design requires that you to obtain the Boolean equation for an even function for the inputs *S*(0), *S*(1), *S*(2), and *S*(3).

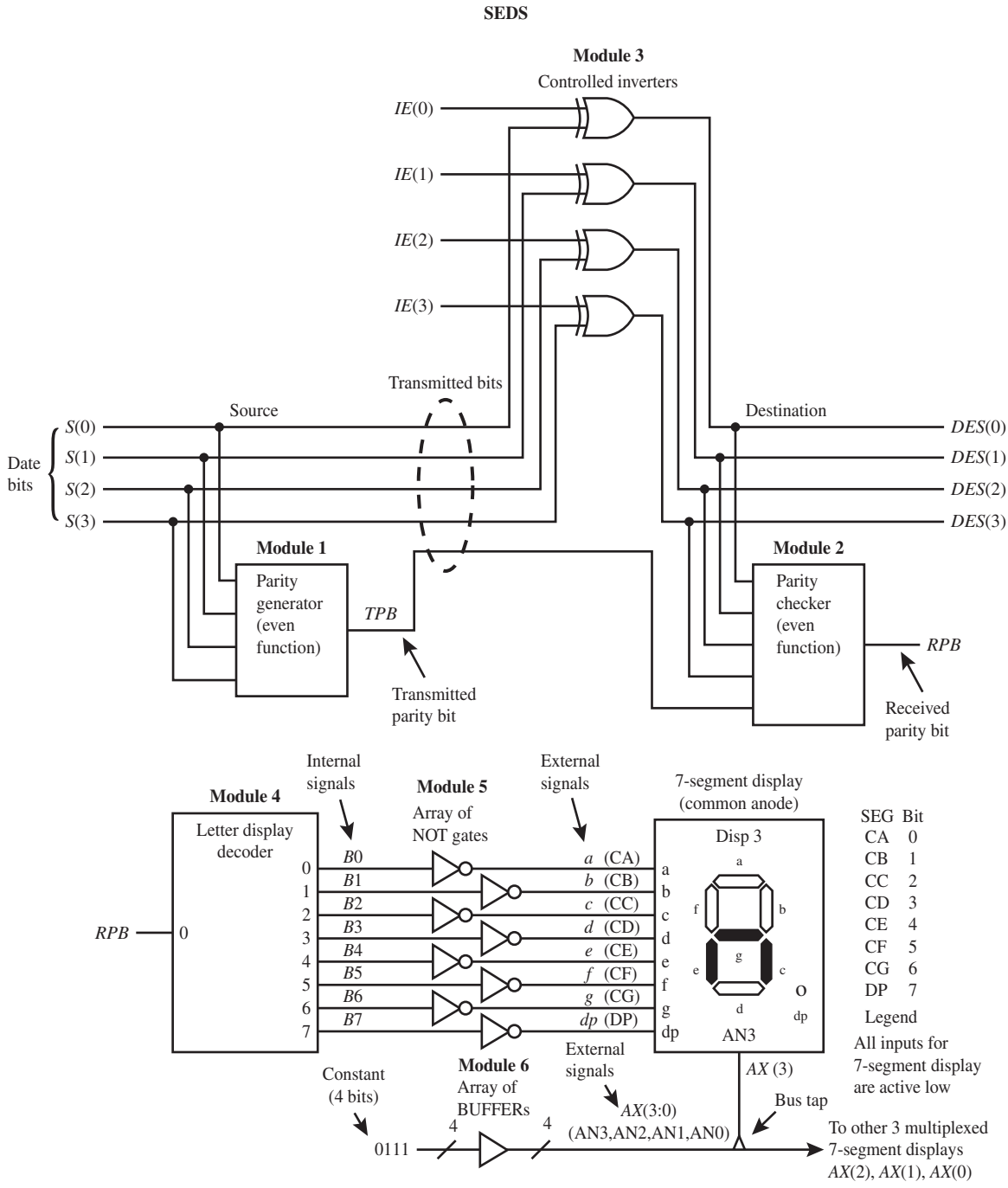
3. Design a Simple Error Detection System

Figure E7A.2 shows a simple error detection system (SEDS) to transmit and receive odd parity with 4 data bits with the capability to insert a single-bit error, a two (or double)-bit error, a three-bit error, and a four-bit error, via inputs *IE*(0) through *IE*(3). Normally, over a short distance from the source to the destination on the FPGA chip, there will be no errors generated on any of the data bit lines or the parity bit line. Module 3 is an array of XOR gates with each XOR gate performing the function of a controlled inverter. When *IE*(0) (insert error) is 1, $IE(0) \oplus S(0) = 1 \oplus S(0) = \overline{S(0)}$, *S*(0) is inverted, which causes an error in the transmitted bit for *S*(0). When *IE*(0) (insert error) is 0, $IE(0) \oplus S(0) = 0 \oplus S(0) = S(0)$, *S*(0) is not inverted, which causes no error in the transmitted bit for *S*(0). To cause a single-bit error, set only one of the four *IE* bits to 1. To cause a two-bit error, set any two of the four *IE* bits to 1. To cause a three-bit error, set any three of the four *IE* bits to 1. To cause a four-bit error, set all four of the four *IE* bits to 1. In practice, only single-bit and double-bit errors generally occur.

Table E7A.1 shows a template and a partially filled-in truth table for the letter display decoder with active low outputs. The input for the letter display decoder is received parity bit (*RPB*).

TABLE E7A.1 Template and partially filled-in truth table for the letter display decoder with active high outputs

	Letter display decoder (active high outputs)								7-segment display
RPB	*B*7	*B*6	*B*5	*B*4	*B*3	*B*2	*B*1	*B*0	
0	0	1	0	1	0	1	0	0	n
1									E



(a)

FIGURE E7A.2 Annotated schematic for a simple error detection system: (a) complete system minus the slide switches and single LEDs

The 7-segment display will indicate no error occurs with the symbol for an "n", and it will indicate that an error occurs with the symbol "E".

Recommended Pre-Lab:

1. Task 1.

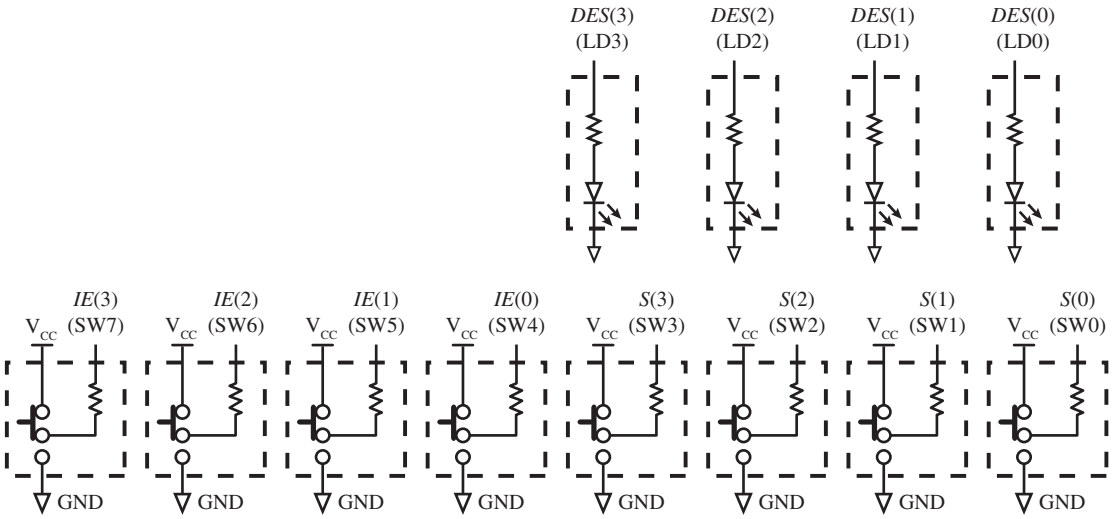


FIGURE E7A.2 Annotated schematic for a simple error detection system: (b) slide switches and single LEDs

2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Obtain the Boolean equation for the even function parity generator. Create a new project named PG, and write complete VHDL code for the logic symbol PG in Figure E7A.1. Write the VHDL code using a dataflow design style via a Boolean equation. (Hint: See Chapter 7, Section 7.4.1, for the definition of an even function.) Simulate the design to verify that *TPB* has a value of 1 when the input string has an even number of 1s (0, 2, and 4), else *TPB* has a value of 0. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] Create a truth table for the inputs *S*(3) through *S*(0) and output *TPB*, and fill in the truth table to verify that your PG design has a value of 1 when the input string has an even number of 1s and has a value of 0 otherwise. If your design does not generate *TPB* as an even function, you know that the VHDL code has an error. You must find the error or errors and fix them.
2. Fill in the truth table in Table E7A.1 for the letter display decoder with active high outputs.
3. Write the Boolean output equations for the letter display decoder.
4. Use a flat design approach to obtain the simple error detection system in Figure E7A.2. (Hint: See Section 2.4.5 in Chapter 2 or the end of Section 4.8 in Chapter 4 for discussion of a flat design approach.) Create a new project named SEDS and write complete VHDL code for the system. You can use the architecture part of the VHDL code for parity generator as module 1 in your flat design. Write the VHDL code for modules 2 through 5 using Boolean equations via a dataflow design style.
5. Complete the design cycle for your SEDS circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.

c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.

6. Check to see if your SEDS circuit works in hardware. Set all four of the four *IE* bits to 0. Verify that outputs LD3 through LD0 match the inputs SW3 through SW0 for all 16 different combinations of the switches and that the 7-segment display shows the symbol "n", for no error. If this is not true when you test your circuit in hardware, you know that the VHDL code has an error. You must find the error or errors and fix them. Next check to see if your SEDS circuit works in hardware for a single-bit error. Set each of the four *IE* bits to 1, one at a time. Verify that outputs LD3 through LD0 match the inputs SW3 through SW0, except for the *IE* bit you set to 1 for all 16 different combinations of the switches and that the 7-segment display shows the symbol "E", for error. If this is not true when you test your circuit in hardware, you know that the VHDL code has an error. You must find the error or errors and fix them.
7. Does your SEDS circuit detect double-bit errors? Show examples by filling in the following partial truth table. Use 1 for an LED turned on and 0 for an LED turned off.

SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0	LD3	LD2	LD1	LD0	DSP3
0	0	1	1	0	0	0	0					
0	1	1	0	0	1	0	1					
1	0	0	1	1	1	1	1					

8. Does your SEDS circuit detect three-bit errors? Show examples by filling in the following partial truth table. Use 1 for an LED turned on and 0 for an LED turned off.

SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0	LD3	LD2	LD1	LD0	DSP3
0	1	1	1	0	0	0	0					
1	1	1	0	0	1	0	1					
1	0	1	1	1	1	1	1					

9. Does your SEDS circuit detect a four-bit error? Show an example by filling in the following partial truth table. Use 1 for an LED turned on and 0 for an LED turned off.

SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0	LD3	LD2	LD1	LD0	DSP3
1	1	1	1	0	0	0	0					

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working designs. Your final working design for this experiment is task 6 (SEDS design).
2. Include the Boolean equation for the even function parity generator.
3. Include the complete VHDL code for your design PG design.
4. Include a printout of the simulation waveform diagram for your PG design.
5. Include your filled-in truth table for your PG design.
6. Include your filled-in truth table for the letter display decoder.
7. Include the Boolean output equations for the letter display decoder.
8. Include the complete VHDL code for your SEDS design.

9. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your SEDS design.
10. Include your answer to the question and your filled-in partial truth table obtained by testing your SEDS design in task 7.
11. Include your answer to the question and your filled-in partial truth table obtained by testing your SEDS design in task 8.
12. Include your answer to the question and your filled-in partial truth table obtained by testing your SEDS design in task 9.
13. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered in your solutions. You may include any helpful hints and improvements you may think of for this experiment.
14. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 7B: DESIGNING AND TESTING A 4-BIT SIMPLE ADDER-SUBTRACTOR SYSTEM USING A HIERARCHAL DESIGN APPROACH

1. Learning Objectives

In this experiment, you will practice designing and testing a special combinational logic system. First, you will learn how to design a full adder and simulate it to verify that it works. Second, you will learn how to design an XOR gate and simulate it to verify that it works. Third, you will design a 4-bit simple adder-subtractor system using indirect subtraction by addition via a hierarchal design approach. Fourth, you will download and test your design in hardware to verify that the system works. This is summarized as follows:

1. Design a full adder and simulate it to verify that it works.
2. Design an XOR gate and simulate it to verify that it works.
3. Design a 4-bit simple adder-subtractor system using indirect subtraction by addition via a hierarchal design approach.
4. Download and test your design in hardware to verify that the system works.

2. Design a Full Adder

Table E7B.1 shows a partially filled-in truth table for a full adder.

TABLE E7B.1 Partially filled-in truth table for a full adder

A	*B*	*CI*	*CO*	*S*
0	0	0		
0	0	1	0	1
0	1	0		
0	1	1	1	0
1	0	0		
1	0	1		
1	1	0		
1	1	1		

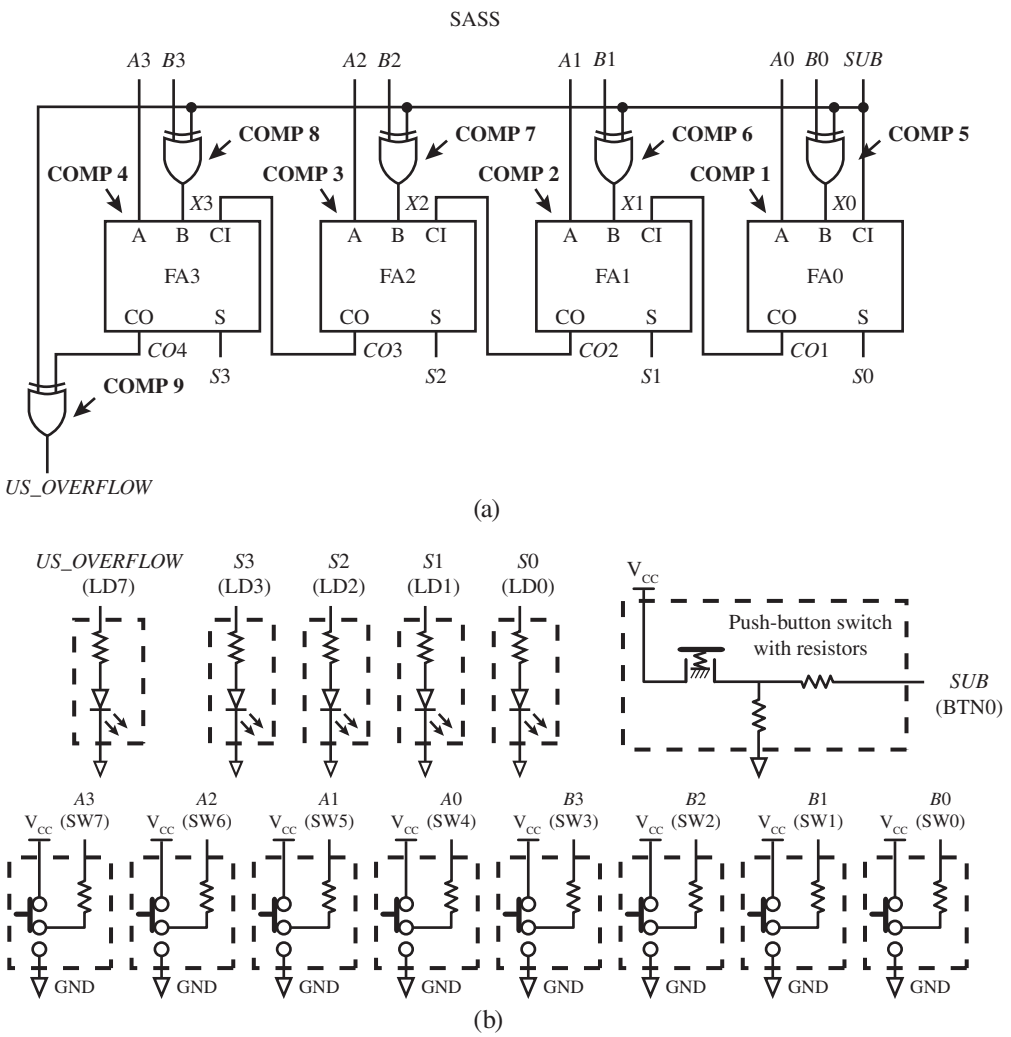
For this design you will be required to fill in the truth table and obtain the Boolean equations for *S* and *CO*.

3. Design a 4-bit Simple Adder-Subtractor System

Figure E7B.1 shows the design specification for a 4-bit simple adder-subtractor system (SASS) to perform unsigned addition of $A + B$ and $A - B$. *US_OVERFLOW* is the unsigned overflow bit. When adding two 4-bit operands, *US_OVERFLOW* = 0 if the 4-bit result *S*3 *S*2 *S*1 *S*0 is correct, and *US_OVERFLOW* = 1 if the 4-bit result *S*3 *S*2 *S*1 *S*0 is incorrect, because the result will not fit in just 4 bits.

When subtracting two 4-bit operands, $A - B$ for this design, *US_OVERFLOW* = 0 if the 4-bit result *S*3 *S*2 *S*1 *S*0 is correct, and *US_OVERFLOW* = 1 if the 4-bit result *S*3 *S*2 *S*1 *S*0 is incorrect, because the result will not fit in just 4 bits.

FIGURE E7B.1
Annotated schematic for a 4-bit simple adder subtractor system: (a) complete system minus the switches and LEDs; (b) switches and LEDs



To make the design process easier, we use a modular design technique to carry out the addition in the circuit. The resulting circuit is called a ripple-carry adder (RCA), because the carry output of component 1 (COMP 1) ripples through all the other adder components (COMP 2 then COMP 3 then COMP 4) via the circuit connections shown in Figure E7B.1.

Subtractor circuits are not generally used in computers because subtraction can be performed with an adder circuit using indirect subtraction by addition as shown with the XOR gates in Figure E7B.1a—that is, components 5 through 8.

Recommended Pre-Lab:

1. Tasks 1 and 2.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Fill in the partially filled-in truth table in Table E7B.1 for a full adder. Write the Boolean equations for outputs *S* and *CO* in sum of products form for the full adder. Create a new project named FA, and write the VHDL code using a dataflow design style with Boolean equations for the full adder. Simulate the design to verify that it follows the full adder truth table. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If your design does not follow the full adder truth table, then you know that the VHDL code has an error. You must find the error or errors and fix them.
2. Create a new project named X_OR. Write the VHDL code using a behavioral design style with a process for a single 2-input XOR gate with the inputs i1 and i2 and the output o1. Simulate the design to verify that it follows the truth table for a 2-input XOR gate. If your design does not follow the truth table for a 2-input XOR gate, then you know that the VHDL code has an error. You must find the error or errors and fix them.
3. Use a hierarchal design approach to design the 4-bit simple adder-subtractor system shown in Figure E7B.1. (Hint: See Chapter 4, Section 4.8 Structural Design Style.) First, create a new project named SASS and write the complete VHDL code for the system. You can use the complete VHDL code for the full adder for the definition for COMP1 through COMP4—that is, components 1 through 4—and the XOR gate for the definition for COMP5 through COMP9—that is, components 5 through 9. After you create the new project click Project, then click Add Copy of Source to add the VHDL code—that is, the .vhd files, for FA and X_OR to your new project SASS.
4. Complete the design cycle for your SASS design by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
5. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board. Check to see if your SASS design works in hardware. Add 1 to the numbers 1 through 15, in binary, to verify that the result is correct at the output LEDs LD3 LD2 LD1 LD0 (or *S*3 *S*2 *S*1 *S*0). LD7 (or *US_OVERFLOW*) should be 0 for all the additions except for 1 added to 15, which should indicate an error. If your design does not add these numbers correctly, then you know that the VHDL code has an error. You must find the error or errors and fix them. Subtract 0 through 7 from the number 7 to verify that the result is correct at the output LEDs LD3 LD2 LD1 LD0 (or *S*3 *S*2 *S*1 *S*0). LD7 (or *US_OVERFLOW*) should be 0 for all the subtractions. Subtract 8 through 15 from the number 7 to verify that the result is incorrect at the output LEDs LD3 LD2 LD1 LD0 (or *S*3 *S*2 *S*1 *S*0). LD7 (or *US_OVERFLOW*) should be 1 for all the subtractions. If your design does not perform in this manner, you have made a mistake or mistakes that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 5 (SASS design).
2. Include your filled-in truth table for a full adder.
3. Include the Boolean equations for a full adder.
4. Include the complete VHDL code for your X_OR design.
5. Include a printout of the simulation waveform diagram for your X_OR design.
6. Include the complete VHDL code for your SASS design.
7. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your SASS design.
8. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered in your solutions. You may include any helpful hints and improvements you may think of for this experiment.
9. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 8: DESIGNING AND TESTING A LUT DESIGN SYSTEM USING A FLAT DESIGN APPROACH

1. Learning Objectives

In this experiment, you will practice designing and testing a LUT design system via a flat design approach. First, you will learn how to design a LUT (look-up table) and simulate it to verify that it works. Second, you will learn how to design a hexadecimal display decoder to display the output of the LUT in hexadecimal. Third, you will design a LUT design system via a flat design approach. Fourth, you will download and test your system design in hardware to verify that the system works. This is summarized as follows:

1. Design a LUT (or PROM) and simulate it to verify that it works.
2. Design a hexadecimal display decoder to display the output of the LUT in hexadecimal.
3. Design a LUT design system via a flat design approach.
4. Download and test your system design in hardware to verify that the system works.

2. LUT Truth Table

Figure E8.1 shows the truth table for a simple LUT and the corresponding fuse map.

FIGURE E8.1 Truth table for a simple LUT and the corresponding fuse map

A	*B*	*F*1	*F*2	*F*3	*F*4
0	0	1	0	1	0
0	1	1	1	1	1
1	0	0	1	1	1
1	1	0	1	0	0

≡

ADDRESS (Hexadecimal)	*DATA* (Hexadecimal)
0	A
1	F
2	7
3	4

3. LUT Logic Circuit and Logic Symbol

A logic circuit for the LUT is shown in Figure E8.2 using a 2-to-4 decoder and a set of OR gates. Figure E8.3 shows a logic symbol for the LUT in Figure E8.2.

Note: In VHDL, the signal *ADDRESS* can be expressed as either ADDRESS <= A&B or ADDRESS <= (A,B), and the signal *DATA* can be expressed as either DATA <=

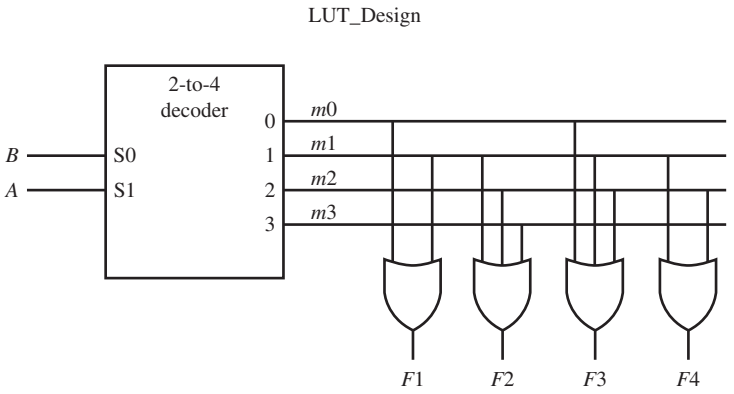


FIGURE E8.2 Logic circuit for the LUT using a 2-to-4 decoder and a set of OR gates

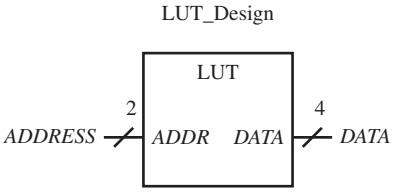


FIGURE E8.3 Logic symbol for the PROM or LUT in Figure E8.2

F1&F2&F3&F4 or DATA <= (F1,F2,F3,F4). The ampersand character is the concatenation operator in VHDL, and the list of signals in parentheses forms an aggregate.

4. Hexadecimal Display Decoder Truth Table and Logic Symbol

Figure E8.4 shows a partially filled-in truth table for the hexadecimal display decoder with active low outputs. In this truth table for the digits 0 and 1, observe that the output values for D represent active low outputs—that is, 0 represents a segment that is turned on and 1 represents a segment that is turned off.

Hexadecimal display decoder

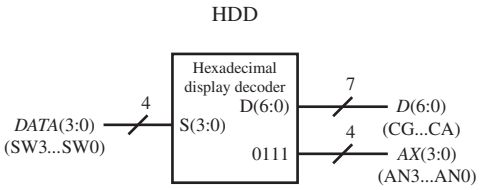
S(3)	*S*(2)	*S*(1)	*S*(0)	*D*(6)	*D*(5)	*D*(4)	*D*(3)	*D*(2)	*D*(1)	*D*(0)
0	0	0	0	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	0	0	1
0	0	1	0							
0	0	1	1							
0	1	0	0							
0	1	0	1							
0	1	1	0							
0	1	1	1							
1	0	0	0							
1	0	0	1							
1	0	1	0							
1	0	1	1							
1	1	0	0							
1	1	0	1							
1	1	1	0							
1	1	1	1							

7-segment display

FIGURE E8.4 Partially filled-in truth table for the hexadecimal display decoder with active low outputs

Figure E8.5 shows a logic symbol for the hexadecimal display decoder in Figure E8.4 with an *AX* output added to drive the enable inputs of the 7-segment display.

FIGURE E8.5 Logic symbol for the hexadecimal display decoder in Figure E8.4 with an *AX* output added to drive the enable inputs of the 7-segment display



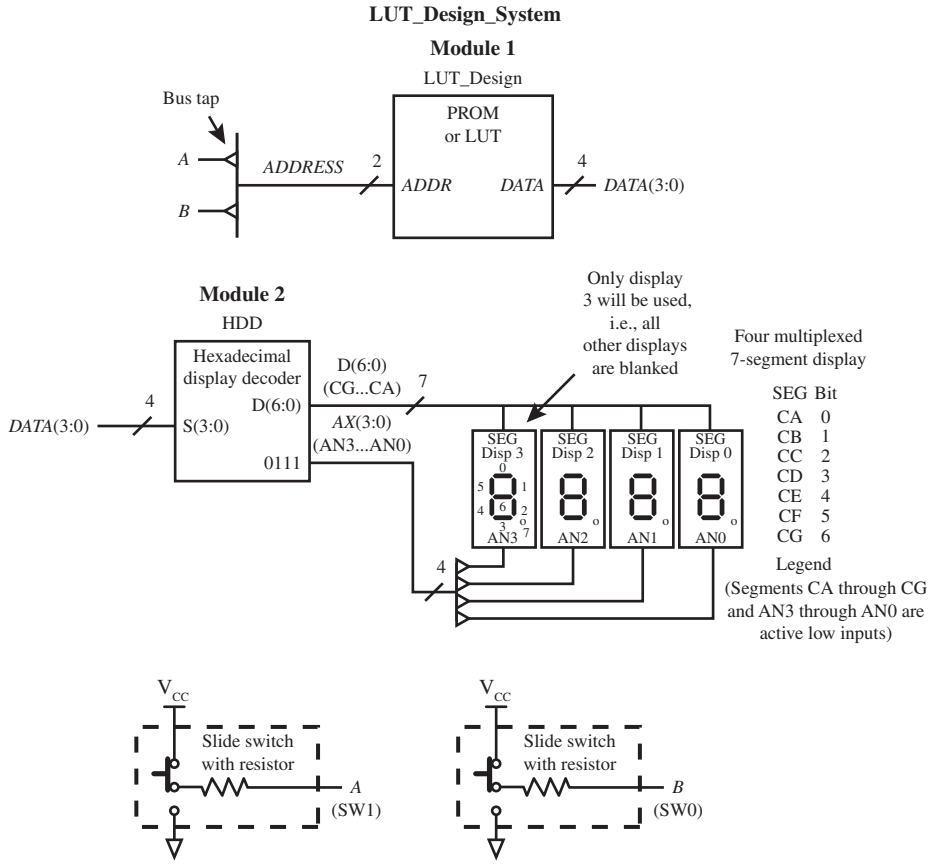
Things you should notice about the input signals and the output signals for the hexadecimal display decoder in Figure E8.5:

- The input signals *DATA*(3:0) are connected to the switch inputs SW3 downto SW0, respectively.
- The output signals *D*(6:0) are connected to the 7-segment display inputs CG downto CA, respectively.
- The output signal *AX*(3:0) are connected to the 7-segment display inputs AN3 downto AN0, respectively.

5. LUT Design System

Figure E8.6 shows an annotated schematic for a LUT design system (LUT_Design_System).

FIGURE E8.6 Annotated schematic for a LUT design system (LUT_Design_System)



Things you should notice about the annotated schematic in Figure E8.6:

- The small triangles (called bus taps) represent connections to individual signals on the buses.
- The internal signals—that is, the signals that are placed between the **architecture** in VHDL and the first **begin**, are *m*0, *m*1, *m*2, *m*3, F1, F2, F3, F4, *ADDRESS* (1:0) and *DATA* (3:0).
- The 7-segment display displays the hexadecimal values of the LUT design system only on display 3.

Recommended Pre-Lab:

1. Tasks 1 and 2.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Create a new project named LUT_Design, and write complete VHDL code for the logic circuit in Figure E8.2. Write the VHDL code using Boolean equations for the minterms *m*0 through *m*3 and also for the functions *F*1 through *F*4. Simulate the design to verify that it follows the LUT truth table and the corresponding fuse map. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If your simulation does not follow the LUT truth table and the corresponding fuse map, then you know that the VHDL code has an error. You must find the error or errors and fix them.
2. Fill in the remaining outputs for the partially filled-in hexadecimal display decoder truth table in Figure E8.4. Remember that the outputs are active low.
3. Create a new project named HDD and write complete VHDL code for the hexadecimal display decoder for Figure E8.5 using your complete filled-in truth table for the hexadecimal display decoder. Write the VHDL code using a selected signal assignment.
4. Complete the design cycle for your HDD circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
5. Check to see if your HDD circuit works in hardware. Verify for input values 0000 through 1111 that your circuit provided the hexadecimal output values 0 through F. If your design does not perform in this manner, you have made a mistake that you must find and correct.
6. Use a flat design approach to combine the LUT_Design and the hexadecimal display decoder (HDD). (Hint: See Section 2.4.5 in Chapter 2 or the end of Section 4.8 in Chapter 4 for a discussion of a flat design approach.) First, create a new project named LUT_Design_System and write the complete VHDL code for the system. You can use the architecture parts of the VHDL code for LUT_Design and HDD as modules 1 and 2 in your flat design. After you create the new project, click Project, then click Add Copy of Source to add the VHDL code—that is, the .vhd files, for LUT_Design and HDD to your new project LUT_Design_System. Open up the .vhd files for LUT_Design and HDD and copy and paste the architecture parts you need in your flat design. After you copy and paste the architecture parts you need, delete the .vhd files for LUT_Design and HDD.
7. Complete the design cycle for your LUT_Design_System by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.

c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.

8. Check to see if your LUT_Design_System works in hardware. Remember that SW1 provides input signal *A* and SW0 provides input signal *B*. When *A B* = 00, your hexadecimal output should be A, When *A B* = 01, your hexadecimal output should be F. When *A B* = 10, your hexadecimal output should be 7. When *A B* = 11, your hexadecimal output should be 4. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 8 (LUT_Design_System).
2. Include the complete VHDL code for your LUT_Design.
3. Include a printout of the simulation waveform diagram for your LUT_Design.
4. Include your filled-in truth table for the hexadecimal display decoder.
5. Include the complete VHDL code for your HDD design.
6. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your HDD design.
7. Include the complete VHDL code for your LUT_Design_System.
8. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your LUT_Design_System.
9. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
10. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 9A: DESIGNING AND TESTING A ONE-HOT UP/DOWN COUNTER SYSTEM USING A FLAT DESIGN APPROACH

1. Learning Objectives

In this experiment, you will practice designing and testing a one-hot up/down counter system via a flat design approach. First, you will learn how to design a one-hot up/down counter with six states and simulate it to verify that it works. Second, you will learn how to design a frequency divider to divide the frequency of 50 MHz down to approximately 6 Hz and verify that the design works by observing the output supplied to a single LED. Third, you will design a one-hot up/down counter system via a flat design approach. Fourth, you will download and test your system design in hardware to verify that the system works. This is summarized as follows:

1. Design a one-hot up/down counter with six States and simulate it to verify that it works.
2. Design a frequency divider to divide the frequency of 50 MHz down to approximately 6 Hz and verify that the design works via a single LED.
3. Design a one-hot up/down counter system via a flat design approach.
4. Download and test your system design in hardware to verify that the system works.

2. One-Hot Up/Down Counter with Six States

Figure E9A.1 shows the state sequence diagram for a one-hot up/down counter(OHUDC).

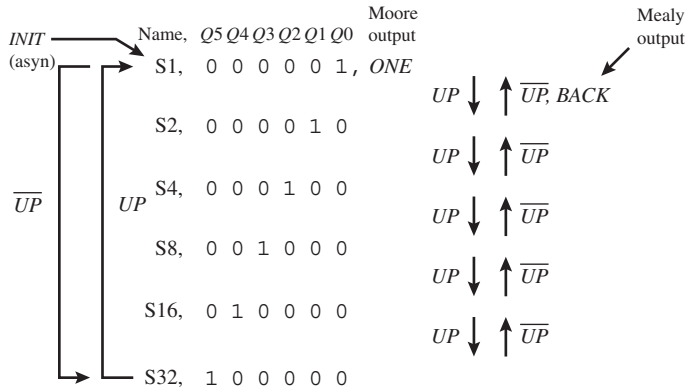


FIGURE E9A.1 State sequence diagram for the one-hot up/down counter (OHUDC)

This is a complex state machine design because the state sequence is controlled by the external input signal *UP*. When *INIT* is asserted or *INIT* = 1, it overrides the clock and forces the counter to go to state 1 (S1). *INIT* is not asserted for the following discussion. When *UP* is asserted or *UP* = 1, the counter counts up. A Moore output signal called *ONE* is used to indicate when the counter is in state 1 (S1)—that is, *ONE* = 1. If the counter is not in state 1, then *ONE* = 0. When $\overline{UP}$ is asserted or *UP* = 0, the counter counts down. A Mealy output signal called *BACK* is used to indicate when the counter counts down (or backward) from state 2 to state 1, so the signal *BACK* is a function of the state of the machine and also a function of the inputs signal *UP*. *BACK* = 1 only when $\overline{UP}$ is asserted in state 2; otherwise *BACK* = 0.

A logic symbol for the one-hot up/down counter with six states is shown in Figure E9A.2.

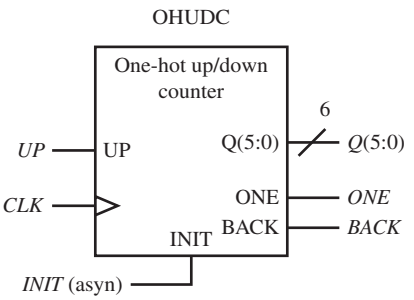


FIGURE E9A.2 Logic symbol for a one-hot up/down counter with six states

3. Frequency Divider with Input and Output

Figure E9A.3 shows a logic symbol for a frequency divider (FD) connected to a single LED that divides a 50-MHz frequency down to approximately 6 Hz.

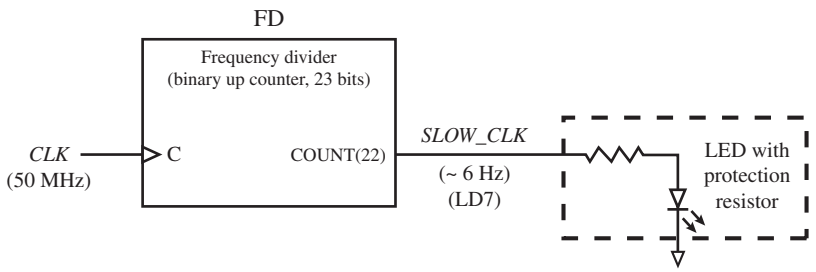


FIGURE E9A.3 Frequency divider connected to a single LED

For this design, a binary counter with 23 bits that range from 22 down to 0 is used to divide the frequency of the clock from 50 MHz down to $50/2^{23}$ MHz or 5.9605 Hz. The signal name for the output of the frequency divider is *COUNT*, and it is an internal signal.

4. One-Hot Up/Down Counter System

Figure E9A.4 shows an annotated schematic for a one-hot up/down counter system that includes a frequency divider, a one-hot up/down counter, a decimal display decoder, and a buffer array.

Things you should notice about the schematic in Figure E9A.4:

- The small triangles (called bus taps) represent connections to individual signals on the buses.
- The internal input signals—that is, the signals that are placed between the **architecture** in VHDL and the first **begin**, are *COUNT*(22:0), *SLOW_CLK*, and *Q*(5:0).
- The signal *INIT* (asyn) is used to initialize the counter system to state 1 (S1) at any time.
- The frequency divider is designed to output *SLOW_CLK* at approximately 6 Hz.
- *ONE* is a Moore output that is true only when the state machine is in state 1.
- *BACK* is a Mealy output that is true only when the state machine is in state 2 and the external input signal $\overline{UP}$ is true at the same time.
- The 7-segment display should display the state of the counter for states 1, 2, 4, and 8, and it should be turned off for states 16 and 32. The six LEDs with protection resistors LD5 through LD0 display the state of the counter in binary for all states.

Recommended Pre-Lab:

1. Tasks 1 and 2.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Use the state sequence diagram in Figure E9A.1 to draw an equivalent state diagram for the one-hot up/down counter. Use the true where shown signal convention for the Moore output *ONE*, and also for the Mealy output *BACK*. To simplify the state diagram drawing, use the state variable *Q*0 in state 1, *Q*1 in state 2, . . ., and *Q*5 in state 32.
2. Create a new project named OHUDC, and write complete VHDL code for the one-hot up/down counter in Figure E9A.1. Write the VHDL code using the two-process PS/NS method. (Hint: See Chapter 9, Section 9.8, Listing 9.6.) Simulate the design to verify that it follows the state sequence diagram in Figure E9A.1 and your equivalent state diagram. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If your simulation does not follow the state sequence diagram or the equivalent state diagram, then you know that the VHDL code has an error. You must find the error or errors and fix them.
3. Create a new project named FD, and write complete VHDL code for the logic symbol FD connected to a single LED shown in Figure E9A.3. (Note: Do not use FD as the name of the design entity because this name is reserved as a keyword in the Xilinx library. FD1 will work if you want to use this name.) Write the VHDL code using a conditional signal assignment via the arithmetic method. (Hint: See Chapter 6, Section 6.8, Listing 6.6.)
4. Complete the design cycle for your FD circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.

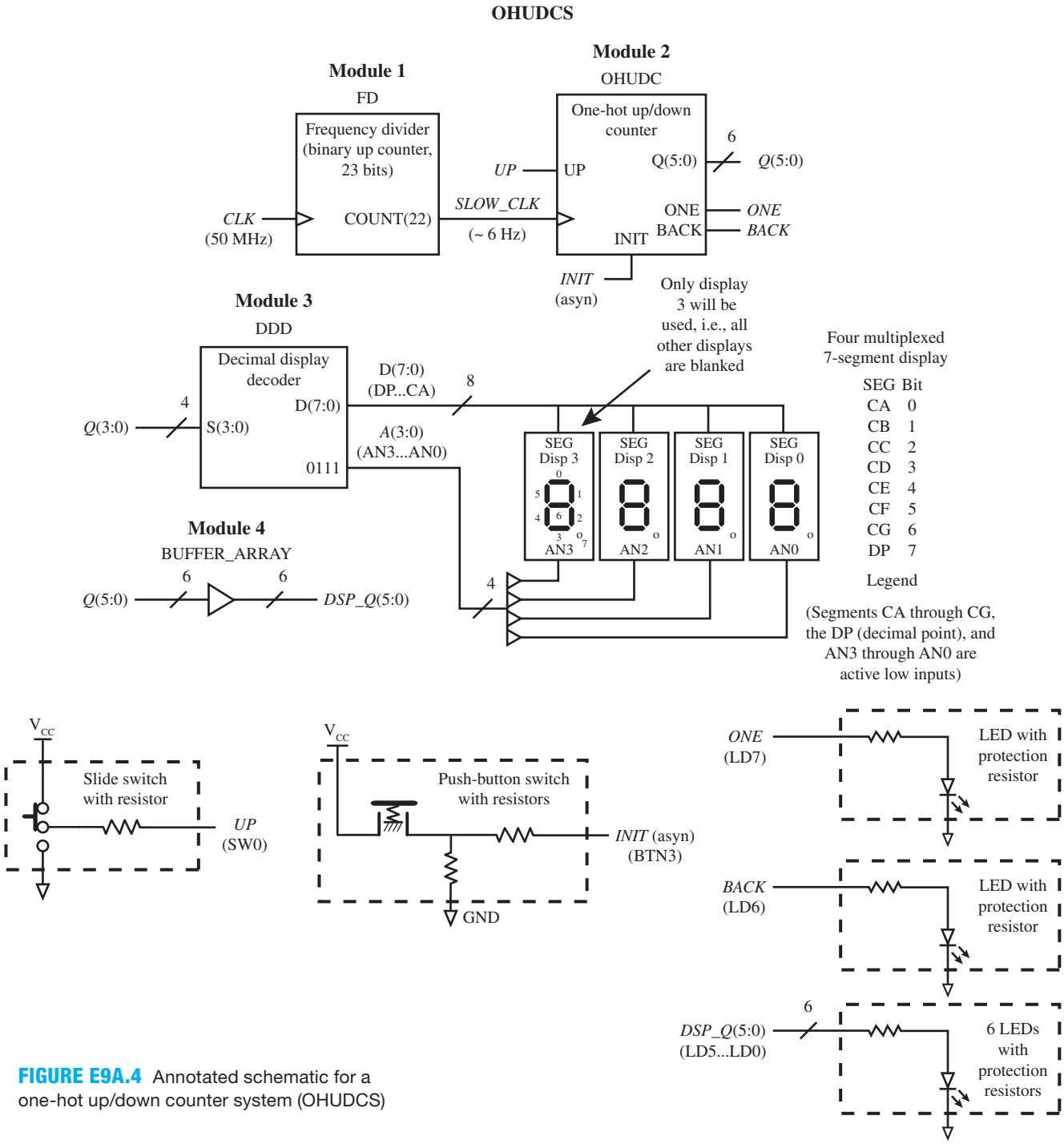


FIGURE E9A.4 Annotated schematic for a one-hot up/down counter system (OHUDCS)

5. Check to see if your FD circuit works in hardware. Verify that LD7 blinks at a frequency of approximately 6 Hz. If your design does not perform in this manner, you have made a mistake that you must find and correct.
6. Use a flat design approach to combine the frequency divider, one-hot up/down counter, decimal display decoder, and BUFFER_ARRAY so that the one-hot up/down counter uses

the signal *SLOW_CLK* provided by the frequency divider as its clock input. (Hint: See Section 2.4.5 in Chapter 2 or the end of Section 4.8 in Chapter 4 for a discussion of a flat design approach.) First, create a new project named OHUDCS and write the complete VHDL code for the system. You can use the architecture parts of the VHDL code for FD and the one-hot up/down counter as modules 1 and 2 in your flat design. After you create the new project, click Project, then click Add Copy of Source to add the VHDL code—that is, the .vhd files, for FD and OHUDC to your new project OHUDCS. Open up the .vhd files for FD and OHUDC and copy and paste the architecture parts you need in your flat design. After you copy and paste the architecture parts you need, delete the .vhd files for FD and OHUDC. You must remember to include *SLOW_CLK* as an internal signal in the architecture of your flat design and also to use *SLOW_CLK* as the clock signal that drives your one-hot up/down counter.

7. Complete the design cycle for your OHUDCS design by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
8. Check to see if your OHUDCS design works in hardware. When SW0 is pushed forward—that is, SW0 = 1—verify that LD5 downto LD0 counts in the one-hot sequence 000001, 000010, 000100, 001000, 010000, 100000, and repeats, where 0 represents an LED that is off and 1 represents an LED that is on. When push button BTN3 is pressed and held down, LD0 and LD7 go to 1 and all the other LEDs go to 0 until the push button is released. When SW0 is pulled back—that is, SW0 = 0—verify that LD5 downto LD0 counts in the one-hot sequence 100000, 010000, 001000, 000100, 000010, 000001, and repeats. During the time that the counter is in state 2 and UP = 0, the Mealy output BACK = 1. BACK = 0 at all other times. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 8 (OHUDCS design).
2. Include your equivalent state diagram for the one-hot up/down counter.
3. Include the complete VHDL code for your OHUDC design.
4. Include a printout of the simulation waveform diagram for OHUDC design.
5. Include the complete VHDL code for your FD design.
6. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your FD design.
7. Include the complete VHDL code for your OHUDCS design.
8. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your OHUDCS design.
9. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
10. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 9B: DESIGNING AND TESTING A 10-STATE COUNTER SYSTEM USING A HIERARCHAL DESIGN APPROACH

1. Learning Objectives

In this experiment, you will practice designing and testing a 10-state counter system via a hierarchal design approach. First, you will learn how to design an up(0-4)/down(9-5) 10-state counter and simulate it to verify that it works. Second, you will learn how to design a frequency divider to divide the frequency of 50 MHz down to approximately 1.5 Hz and verify that the design works by observing the output supplied to a single LED. Third, you will design a 10-State Counter System via a hierarchal design approach. Fourth, you will download and test your system design in hardware to verify that the system works. This is a summarized as follows:

1. Design an up(0-4)/down(9-5) 10-state counter and simulate it to verify that it works.
2. Design a frequency divider to divide the frequency of 50 MHz down to approximately 1.5 Hz and verify that the design works via a single LED.
3. Design a 10-state counter system via a hierarchal design approach.
4. Download and test your system design in hardware to verify that the system works.

2. Up(0-4)/Down(9-5) 10-State Counter

Figure E9B.1 shows a state sequence diagram for an up(0-4)/down(9-5) 10-state counter.

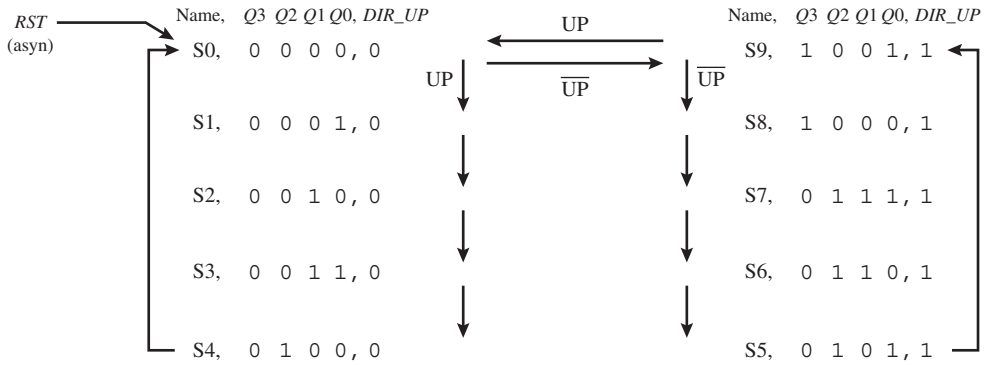


FIGURE E9B.1 State sequence for the up(0-4)/down(9-5) 10-state counter

This is a complex state machine design because the state sequence is controlled by the external input signal *UP*. When *RST* is asserted or *RST* = 1, it overrides the clock and forces the counter to go to state 0 (S0). *RST* is not asserted for the following discussion. When the counter is in state 0 (S0) and *UP* is asserted or *UP* = 1, the counter counts up to state 1 (S1). When the counter is in State 9 (S9) and $\overline{UP}$ is asserted or *UP* = 0, the counter counts down to state 8 (S8). The state sequence in the state sequence diagram can only be changed at states 0 and 9. The Moore output *DIR_UP* is a function of the state of the machine and not a function of the input signal *UP*.

A logic symbol for the up(0-4)/down(9-5) 10-state counter (TSC) is shown in Figure E9B.2.

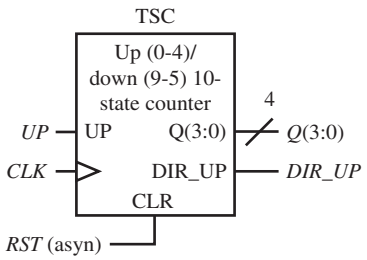


FIGURE E9B.2 Logic symbol for an up(0-4)/down(9-5) 10-state counter

3. Frequency Divider with Input and Output

Figure E9B.3 shows a logic symbol for a frequency divider (FD) connected to a single LED that divide a 50-MHz frequency down to approximately 1.5 Hz.

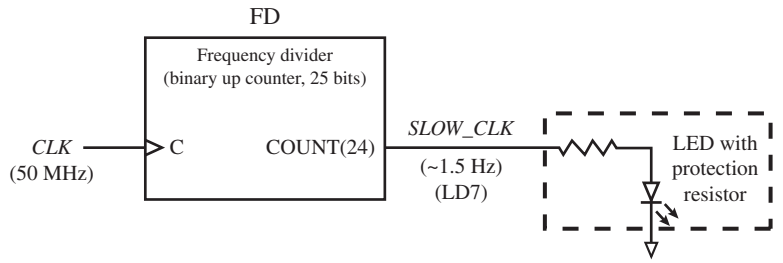


FIGURE E9B.3 Frequency divider connected to a single LED

For this design, a binary counter with 25 bits that range from 24 down to 0 is used to divide the frequency of the clock from 50 MHz down to $50/2^{25}$ MHz or 1.4901 Hz. The signal name for the output of the frequency divider is *COUNT*, and it is an internal signal.

4. 10-State Counter System

Figure E9B.4 shows an annotated schematic for a 10-state counter system that includes a frequency divider, an up(0-4)/down(9-5) 10-state counter, a decimal display decoder, and a buffer array.

Things you should notice about the schematic in Figure E9B.4:

- The small triangles (called bus taps) represent connections to individual signals on the buses.
- The internal input signals—that is, the signals that are placed between the **architecture** in VHDL and the first **begin**, are *COUNT*(24:0), *SLOW_CLK*, and *Q*(3:0).
- The signal *RST* (asyn) is used to reset the counter system to state 0 (S0) at any time.
- The frequency divider is designed to output *SLOW_CLK* at approximately 1.5 Hz.
- The signal *UP* is used to select the sequence—that is, either up or down $\overline{UP}$. When the counter counts up, the Moore output signal *DIR_UP* turns on the decimal point DP, and when the counter counts down the decimal point DP turns off.
- *DSP_Q*(3:0) turns on four single LEDs to display the sequence of the counter in binary.
- The 7-segment LED display, Disp 3, displays the sequence of the counter in decimal.

Recommended Pre-Lab:

1. Tasks 1 and 2.
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Use the state sequence diagram in Figure E9B.1 to draw an equivalent state diagram for the up(0-4)/down(9-5) 10-state counter. Use the true where shown signal convention for the Moore output *DIR_UP*.
2. Create a new project named TSC, and write complete VHDL code for the up(0-4)/down(9-5) 10-state counter in Figure E9B.1. Write the VHDL code using the two-process PS/NS method. (Hint: See Chapter 9, Section 9.5, Listing 9.2.) Simulate the design to verify that it follows the state sequence diagram in Figure E9B.1 and your equivalent state diagram. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.] If your simulation does not follow the state sequence diagram or the equivalent state

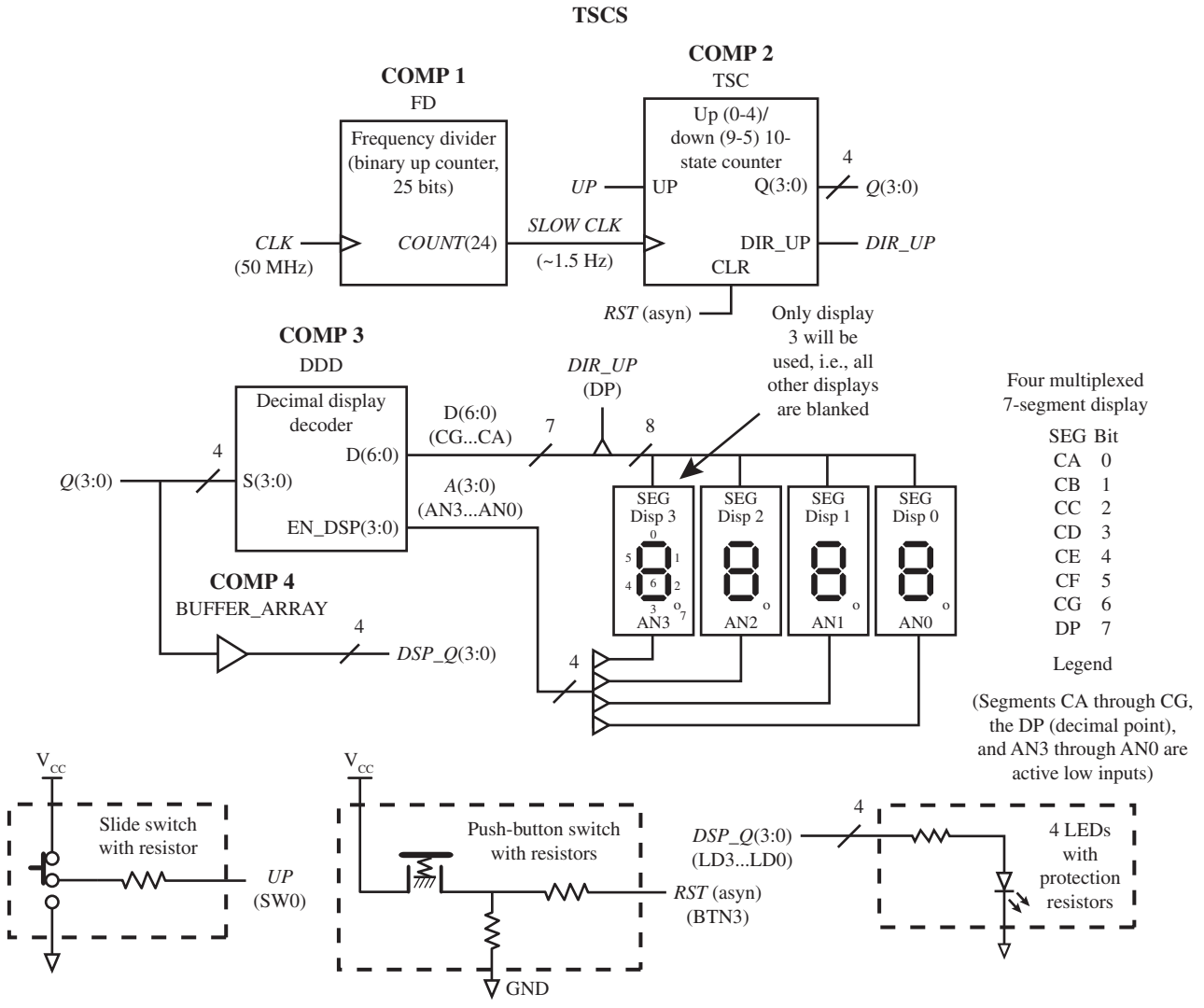


FIGURE E9B.4 Annotated schematic for a 10-state counter system (TSCS)

diagram, then you know that the VHDL code has an error. You must find the error or errors and fix them.

3. Create a new project named FD, and write complete VHDL code for the logic symbol FD connected to a single LED shown in Figure E9B.3. (Note: Do not use FD as the name of the design entity because this name is reserved as a keyword in the Xilinx library. FD1 will work if you want to use this name.) Write the VHDL code using a conditional signal assignment via the arithmetic method. (Hint: See Chapter 6, Section 6.8, Listing 6.6.)
4. Complete the design cycle for your FD circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.

5. Check to see if your FD circuit works in hardware. Verify that LD7 blinks at a frequency of approximately 1.5 Hz. If your design does not perform in this manner, you have made a mistake that you must find and correct.
6. Use a hierarchal design approach to combine the frequency divider, up(0-4)/down(9-5) 10-state counter, decimal display decoder, and BUFFER_ARRAY so that the up(0-4)/down(9-5) 10-state counter uses the signal *SLOW_CLK* provided by the frequency divider as its clock input. First, create a new project named TSCS and write the complete VHDL code for the system. You can use the complete VHDL code for the frequency divider and the up(0-4)/down(9-5) 10-state counter as the definitions for components 1 and 2 in your hierarchal design. After you create the new project, click Project, then click Add Copy of Source to add the VHDL code—that is, the .vhd files, for FD and TSC to your new project TSCS.
7. Complete the design cycle for your TSCS design by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
8. Check to see if your TSCS design works in hardware. Verify that *RST* clears Disp 3 to 0. When SW0 is pushed forward, the counter counts up from 0 through 4 and repeats this sequence, and when SW0 is pulled back, Disp 3 counts down from 9 through 5 and repeats this sequence. The decimal point of Disp 3 will be on only when Disp 3 counts up and off when Disp 3 counts down. The change over from up to down will only occur at states 0 and 9. The signal *DIR_UP* that lights the decimal point is not a function of the input signal *UP* but a function of the state of the machine because it is a Moore output. If your design does not perform in this manner, you have made a mistake that you must find and correct.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design for this experiment is task 8 (TSCS design).
2. Include your equivalent state diagram for the up(0-4)/down(9-5) 10-state counter.
3. Include the complete VHDL code for your TSC design.
4. Include a printout of the simulation waveform diagram for your TSC design.
5. Include the complete VHDL code for your FD design.
6. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your FD design.
7. Include the complete VHDL code for your TSCS design.
8. Include a printout of the Edit Constraints (Text), which shows the package pin assignments for your TSCS design.
9. Write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
10. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 10: WORKING WITH EASY1 (EDITOR/ASSEMBLER/SIMULATOR) FOR VBC1

In this experiment, you will practice working with EASY1 (Editor/Assembler/Simulator) for VBC1. Figure E10.1 shows a block diagram for VBC1.

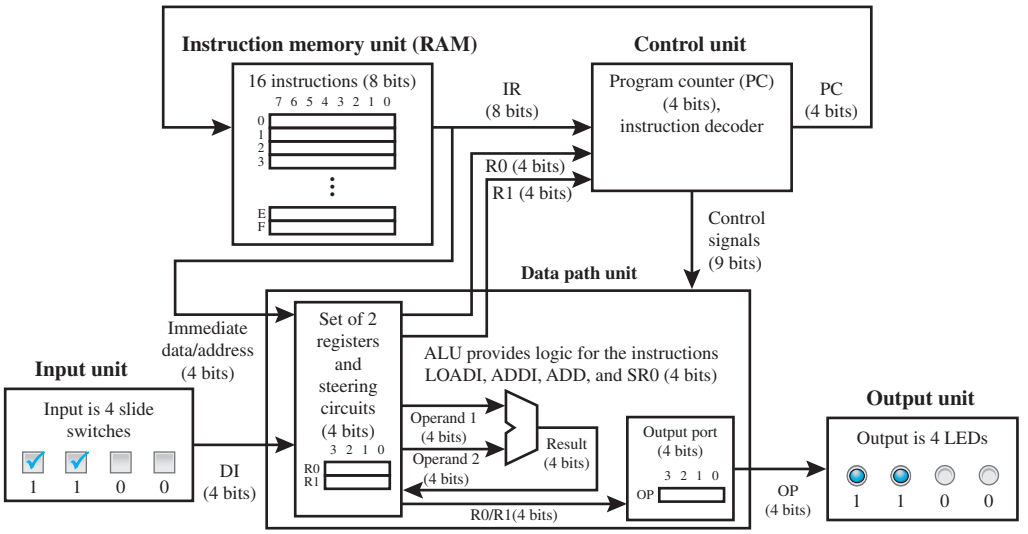


FIGURE E10.1 Block diagram for VBC1

Things you should notice about the block diagram in Figure E10.1:

- The instruction memory unit (RAM) stores the instructions and provides the current contents of the instruction memory at the output IR (instruction register).
- The control unit provides the PC (program counter), which points to the next instruction to be executed.
- The input unit supplies the input via four slide switches.
- The data path unit with registers R0 and R1 and output port provides the capability to execute the IN, OUT, and MOV instructions.
- The control unit provides the capability to execute the JNZ instruction.
- The ALU provides the capability to execute the LOADI, ADDI, ADD, and SR0 instructions.
- The output unit provides the output via 4 LED.

In Figure E10.1 observe that a checkmark at an input represents a logic 1 and no checkmark represents a logic 0. Also observe a dot inside a circle at an output represents a logic 1 and no dot represents a logic 0.

Open the program EASY1 and observe the Screen (or GUI).

Notice the following on the GUI for EASY1:

- The instruction memory is shown on the right of the GUI and is labeled Instruction Memory.
- The current contents of the instruction memory is IR, which is shown via 8 bits on the GUI.
- The PC is shown via 4 bits on the GUI, which is the address indicated by the instruction pointer.
- The input is shown via four squares. When you click on one of the squares it shows a checkmark, and when you click on it again the checkmark is removed. For VBC1, a checkmark represents a slide switch that is pushed forward (this represents a logic 1 or high), while no checkmark represents a slide switch that is pulled back (this represents a logic 0 or low).
- Registers R0 and R1 are shown via 4 bits on the GUI.
- The output is shown via four circles on the GUI. For VBC1, a dot inside a circle presents a lighted LED (this represents a logic 1 or high), while no circle represents an LED that is not lighted (this represents a logic 0 or low).

Recommended Pre-Lab:

1. Read Appendix D, Sections D.1 through D.6, for additional information about EASY1 for this experiment.
2. Write an assembly language program that runs in an infinite loop and bounces the instruction pointer for EASY1 forward, from the first instruction at 0000 to the last instruction at 0100; then bounces the instruction pointer backward, from the last instruction at 0100 to the first instruction at 0000; and then repeats in this manner. Provide a comment for each instruction that explains how the program works and not just what each instruction does. (Hint: This program can be written using only the instructions LOADI and JNZ.) Remember that the instruction JNZ only jumps when the value in the destination register (DR) is not equal to 0. If DR is equal to 0, then the JNZ instruction falls through to the instruction at the next address. You can make the JNZ instructions jump through memory going forward or backward based on the value of DR. For each JNZ instruction you write, use a label for the address in the instruction.
3. Your instructor will provide you with additional information about what should be submitted for your **pre-lab report requirements**—that is, assembled printout of the assembly language program with comments, a copy of your assembly language program with comments in a text file, and the like.

Tasks:

1. Using EASY1, click on the first box in the instruction memory, which has the address 0000, and change the contents of the box to the value 10000000. Be sure that the contents is only 8 bits. Fill in the rest of the instruction memory as shown here:

Instruction Memory

0000	10000000
0001	11000000
0010	11100000
0011	11110000
0100	11111000
0101	11111100
0110	11111110
0111	11111111
1000	00000000

2. Click the Reset button, which moves the instruction pointer (the lighted dot) back to the first address of the first location in the instruction memory, which is 0000. Reset will not reset the instruction memory. Observe that the PC now has the value 0000 and the IR, which represents the current instruction in memory, has the value 10000000. Now click the Step button once and notice that the lighted dot in the instruction memory moves to the address 0001 and the PC has the address 0001. Also notice that the IR has the value of 11000000. Click the Step button repeatedly and observe that the PC changes to the value of the address in memory that is represented by the instruction pointer (the lighted dot) and the IR changes to the values of the current instruction in memory at the lighted dot.
3. Click the Reset button and then click the Run button. Observe that the instruction pointer moves down through the memory, indicating that the instructions in memory are attempting to be executed; however, there are no instructions to execute because we have not placed any instructions into the Assembly area to assemble and load into memory. This exercise

should provide you with the idea that when instructions are actually placed into the Assembly area and assembled and loaded into memory, EASY 1 will execute those instructions.

4. Click the Assemble and Load button, and this will clear the instruction memory, because there is no assembly language program loaded into the Assembly area of instruction memory.
5. Enter Program E10.1 in the Assembly area of the GUI of EASY1, then click Assemble and Load. This will assemble the program (convert the assembly language program to 1s and 0s) and will also load the 1s and 0s for each instruction into instruction memory.

```
;VBC1 Exp 10 Program 1
start: IN R0          ; input 15 via the Input squares
       OUT R0         ; output contents of R0
       MOV R1,R0      ; move contents of R0 to R1
       JNZ R0,start   ; jump to start
```

PROGRAM E10.1 Simple assembly language program for VBC1 using four instructions

6. After the program is loaded into instruction memory, the PC is at 0000, which is the first instruction to be executed, which is in r0, as shown just to the right of the instruction memory in Figure E10.2.
7. As shown in Program E10.1, a semicolon is used to list comments in an assembly language program. The semicolon followed by the comment "input 15 via the Input squares" tells you to click each of the four squares to provide the input of 15—that is, place a checkmark in each of the squares, which represents 1111 = 15, as shown in Figure E10.2.

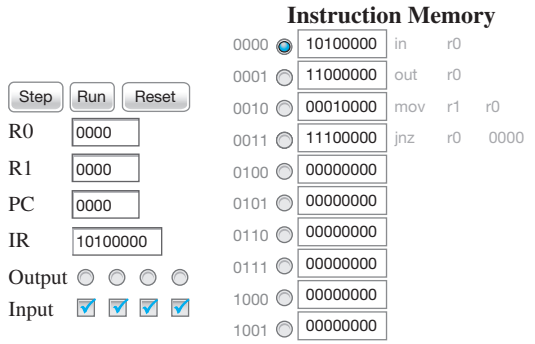
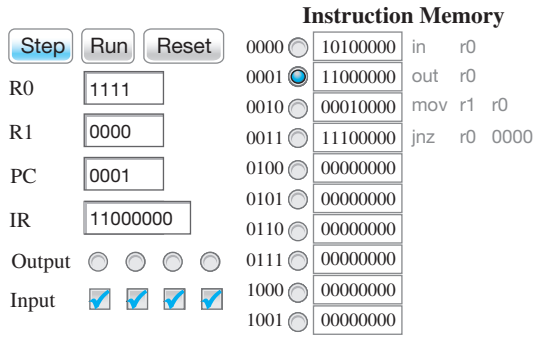
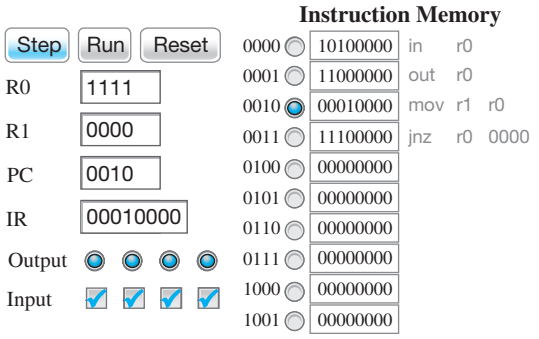


FIGURE E10.2 Program 1 assembled and loaded; input is set to 1111.

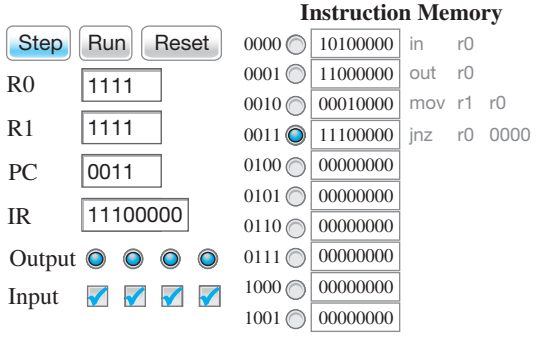
8. Click Step to single step through the program to observe how the program works.
9. Here is what you should observe in the sequence that occurs:



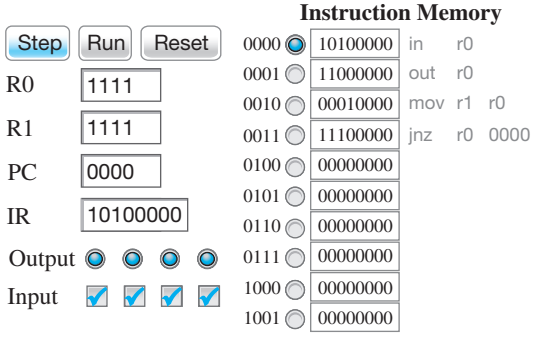
Click Step button first time.



Click Step button second time.



Click Step button third time.



Click Step button fourth time.

10. You should now begin to understand what the instructions IN, OUT, MOV, and JNZ in Program E10.1 actually do as they are executed.
11. To see the program run at a reasonable speed, click Reset and then click Run. This causes the instructions to run at a predetermined speed.
12. Click Reset and change the value of the input from 15 to 3, and single step through the program; also run the program to observe how the values for registers R0 and R1 change and the output changes as the program is executed via single stepping through the program or running the program.
13. Click Reset and change the value of the input from 3 to 0, and observe how the values of registers R0 and R1 change and how the output changes as the program is executed via single stepping through the program or running the program. Explain why the program does not jump to start when the JNZ instruction is executed. Explain what happens when the program reaches the last address in instruction memory. What is the last address in instruction memory?

14. Write an assembly language program—that is, Program 2, which will display an output value that is twice the value that is applied at the input. This will only be true for input values in the range of 1 through 7. Your program should run continuously through its instructions. Enter the program into EASY1, and run the program to verify that it works. Provide a comment for each instruction that explains how the program works and not just what each instruction does.
15. Explain why the restriction must be placed on the range of the values at the input in task 14.
16. Write an assembly language program—that is, Program 3, which will display the output 4-bit LED sequence for a bouncing lights program for VBC1, as shown in Figure E10.3. Your program should run continuously through its instructions. Enter the program into EASY1, and run the program to verify that it works. Provide a comment for each instruction that explains how the program works and not just what each instruction does.

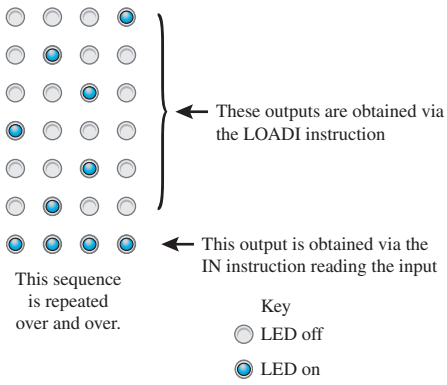


FIGURE E10.3 Output 4-bit LED sequence for a bouncing lights program for VBC1

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is the execution of the assembly language program, Program 2, in task 14 via EASY1.
2. Explain how instruction memory can be cleared via the EASY1 GUI with the click of a button, when there is no assembly language program loaded into the Assembly area of EASY1.
3. For task 12, list the value of the output in binary after single stepping or running the program.
4. For task 13, list the value of the output in binary after single stepping or running the program. Also include the answers to the three questions in task 13.
5. For task 14, include an assembled printout of the assembly language program with comments in Program 2 on the screen or GUI (graphical user interface) for EASY1. Make sure that the total GUI for EASY1 is available on your printout.
6. Include the answer to the question in task 15.
7. For task 16, include an assembled printout of the assembly language program with comments in Program 3 on the screen or GUI (graphical user interface) for EASY1. Make sure that the total GUI for EASY1 is available on your printout.
8. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
9. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 11: WRITING AND SIMULATING PROGRAMS FOR VBC1 WITH EASY1

In this experiment, you will practice writing and simulating programs for VBC1 with EASY1. Figure E11.1 shows a summary of all the assembly language instruction for VBC1 in alphabetical order.

FIGURE E11.1 Summary of all the assembly language instruction for VBC1 in alphabetical order

Assembly language instruction	Comment
ADD DR,SR	DR (destination register) either R0 or R1 SR (source register) either R0 or R1
ADDI DR,Data	DR (destination register) either R0 or R1 Data (data included in the instruction) 0 through 15
IN DR	DR (destination register) either R0 or R1
JNZ DR,Address	DR (destination register) either R0 or R1 Address 0 through 15
LOAD DR,Data	DR (destination register) either R0 or R1 Data (data included in the instruction) 0 through 15
MOV DR,SR	DR (destination register) either R0 or R1 SR (source register) either R0 or R1
OUT DR	DR (destination register) either R0 or R1
SR0 DR,SR	DR (destination register) either R0 or R1 SR (source register) either R0 or R1

Figure E11.2 shows the output 4-bit LED sequence for a smiley face program for VBC1.

FIGURE E11.2 Output 4-bit LED sequence for a smiley face program

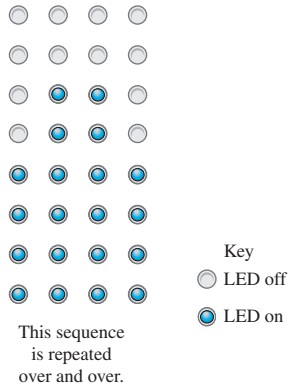


Figure E11.3 shows the output 4-bit LED sequence for a robot eye program for VBC1.

FIGURE E11.3 Output 4-bit LED sequence for a robot eye program

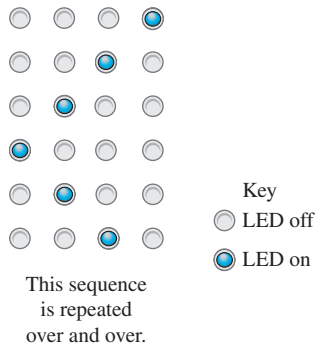


Figure E11.4 shows a state sequence diagram for a 4-bit stoppable one-hot up counter. $Q3$ $Q2$ $Q1$ $Q0$ are displayed on the output LEDs 3 downto 0, respectively, and S is provided by the status of the input slide switches. $S = 0$ is switch inputs 0000, and $S = 1$ is switch inputs that are any value except 0000—that is, 0001, 0010, 0011, etc.

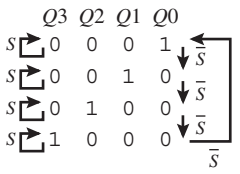


FIGURE E11.4 State sequence diagram for a 4-bit stoppable one-hot up counter

Recommended Pre-Lab:

1. Read Appendix D, Sections D.7 and D.8, for additional information about EASY1 for this experiment. Also read Chapter 11, Section 11.11, Programming Examples and Techniques for VBC1.
2. Write an assembly language program that simultaneously blinks the two least significant output bits five times then simultaneously blinks the two most significant output bits five times and repeats this sequence over and over. Be sure that your program does not run amuck. (Hint: Use two internal loops.) Provide a comment for each instruction that explains how the program works and not just what each instruction does.
3. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—that is, assembled printout of the assembly language program with comments, a copy of your assembly language program with comments in a text file, and the like.

Tasks:

For each program that you write, provide a comment for each instruction that explains how the program works and not just what each instruction does.

1. Write an assembly language program for VBC1 that will produce the smiley face sequence shown in Figure E11.2 as an inline program. Be sure that your program does not run amuck. Enter the program into EASY1, and run the program to verify that it works. Record the number of instructions that are required for your inline program.
2. Write an assembly language program for VBC1 that will produce the smiley face sequence shown in Figure E11.2 as a program that contains at least one internal loop to reduce repeated instructions. Be sure that your program does not run amuck. Enter the program into EASY1, and run the program to verify that it works. Record the number of instructions that are required for your program with an internal loop(s).
3. Write an assembly language program for VBC1 that will produce the robot eye sequence shown in Figure E11.3 as an inline program. Be sure that your program does not run amuck. Enter the program into EASY1, and run the program to verify that it works. Record the number of instructions that are required for your inline program.
4. Write an assembly language program for VBC1 that will produce the robot eye sequence shown in Figure E11.3 as a program that contains at least one internal loop to reduce repeated instructions. Be sure that your program does not run amuck. Enter the program into EASY1, and run the program to verify that it works. Record the number of instructions that are required for your program with an internal loop(s).
5. Write an assembly language program for VBC1 that will produce the 4-bit stoppable one-hot up counter shown in Figure E11.4 as an inline program. Be sure that your program does

not run amuck. Enter the program into EASY1, and run the program to verify that it works. Record the number of instructions that are required for your inline program.

6. Write an assembly language program for VBC1 that will produce the 4-bit stoppable one-hot up counter shown in Figure E11.4 as a program that contains at least one internal loop to reduce repeated instructions. Be sure that your program does not run amuck. Enter the program into EASY1, and run the program to verify that it works. Record the number of instructions that are required for your program with an internal loop(s).
7. Fill in the following table:

Program	Number of instructions, inline	Number of instructions, internal loop(s)
Smiley face		
Robot eye		
4-bit stoppable one-hot up counter		

In general, which programming technique takes longer to execute the complete program sequence one time, an inline program or a program with a loop counter(s) or an internal loop(s)? Why? (Explain your answer.)

8. Place the programs for tasks 1 through 6 in a text file for easy reference.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working programs and get them signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working programs. Your final working programs for this experiment will be chosen at random.
2. Include the filled in table in task 7 and your answers to the questions.
3. Ask your instructor if you should include the copy of your programs in the text file in your report, or if you should e-mail the copy of your programs in your text file to him/her as an attachment.
4. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
5. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 12: DESIGNING AND TESTING VBC1 (DATA PATH UNIT)

In this experiment, you will practice working with bus steering circuits and loadable register circuits.

This is the first in a series of projects to design and test VBC1 (Very Basic Computer 1). Experiment 12 consists of the annotated schematic shown in Figure E12.1.

Figure E12.1 shows the data path unit for VBC1. The schematic shows the external input switch circuits (both slide switches and push-button switches) and the external output circuit for four LEDs. Labels for all the switches and LEDs are included in parentheses—for example, SW3. . .SW0, BTN3, LD0, DP. . .CA, and AN3. . .AN0. These labels are used to obtain the pin assignments for the Digilent board you elect to use—for example, a BASYS 2 board or a NEXYS 2 board.

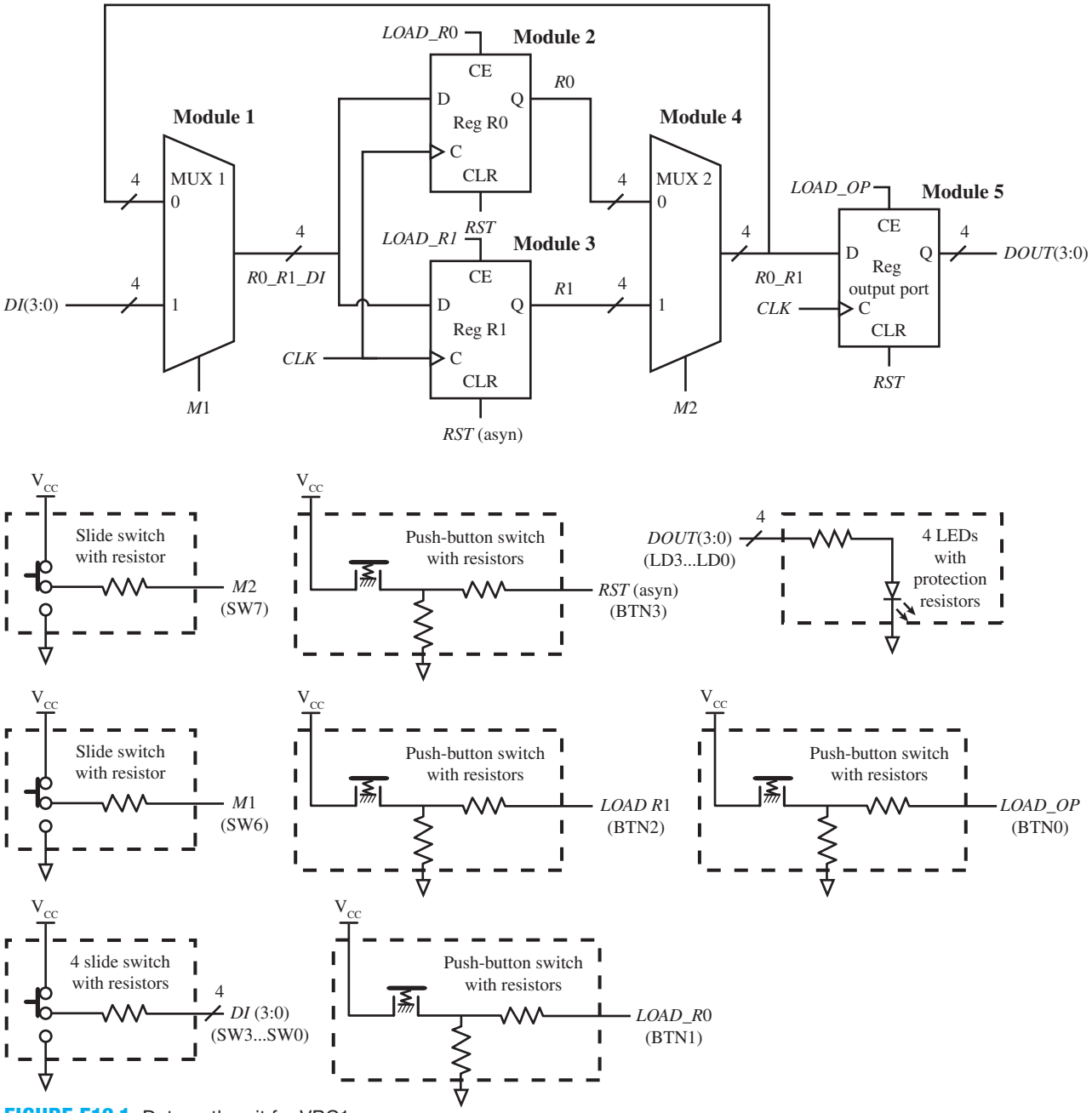


FIGURE E12.1 Data path unit for VBC1

Things you should notice about the schematic in Figure E12.1:

- The schematic represents a portion of the data path unit for VBC1, which performs the instructions IN, OUT, and MOV by manually selecting the control signals *M*1, *M*2, *LOAD_R*0, *LOAD_R*1, and *LOAD_OP* and the input signal *DI*.
- The modules numbered 1 through 5 are used to simplify the design of VBC1 by dividing the design into small circuits.

- The external input and output signals—that is, the port signals for the entity in VHDL—are *DI*(3:0), *M*1, *CLK*, *RST*, *LOAD_R*0, . . . (you should be able to list the rest of them).
- The internal signals, for the architecture in VHDL, are *R*0_*R*1_*DI*, *R*0, *R*1, *R*0_*R*1.
- Modules 1, 2, and 3 provide a data input port driven from four slide switches.
- Modules 4 and 5 provide a data output port, which drives four LEDs.
- The select inputs to modules 1 and 4 are driven by two slide switches.
- The clear inputs to the loadable registers are driven by a push-button switch.
- The clock enable inputs to the loadable registers are driven by push-button switches.
- The control signals *M*1, *M*2, *LOAD_R*0, *LOAD_R*1, and *LOAD_OP* must be supplied manually via slide switches and push-button switches.
- The control signals *M1*, *LOAD_R*0, and *LOAD_R*1 allow the instruction IN to be performed—that is, IN R0 and IN R1.
- The control signals *M*2 and *LOAD_OP* allow the instruction OUT to be performed—that is, OUT R0, and OUT R1.
- The control signals *M*1, *M*2, *LOAD_R*0, and *LOAD_R*1 allow the instruction MOV to be performed—that is, MOV R0,R0, MOV R0,R1, MOV R1,R0, and MOV R1,R1.

If you design and test the data path unit for VBC1 in Figure E12.1, you can manually single step through the assembly language program shown in Program E12.1 to verify that your circuit executes the program correctly—that is, executing one instruction at a time. You must supply the proper values for the control signals *M*1, *M*2, *LOAD_R*0, *LOAD_R*1, and *LOAD_OP*, and also supply the proper values for the input data signal *DI*. *Note*: The result you observe as you single step through the program by manually supplying the control signal values and input values shown in the comments should provide the same result you observe as you single step through the program using EASY1.

PROGRAM E12.1
Simple assembly language program for VBC1 using the instructions IN, OUT, and MOV

```
;VBC1 Exp 12, 13, and 15 Program 1
;Testing IN, OUT, and MOV instructions
IN R0          ; input 9 via 4 slide switches
OUT R0         ; output contents of R0 to LEDs
IN R0          ; input 15 via 4 slide switches
MOV R1,R0      ; move contents of R0 to R1
OUT R1         ; output contents of R1 to LEDs
IN R1          ; input 6 via 4 slide switches
OUT R1         ; output contents of R1 to LEDs
OUT R0         ; output contents of R0 to LEDs
```

The control signals are normally supplied by an instruction decoder and therefore do not have to be provided manually in the final design. The reason for supplying these signals manually, in this experiment, is to learn how VBC1 must supply these signals to perform the instructions IN, OUT, and MOV.

Read Chapter 12, Section 12.3 (Designing bus steering circuits), to learn how to design a bus steering circuit. Read Chapter 12, Section 12.4 (Designing loadable register circuits), to learn how to design loadable register circuits.

To gain an understanding of VBC1, read Chapter 10. To gain an understanding of the instructions IN, OUT, and MOV, read the sections that cover these instructions in Chapter 11. To learn how to edit, assemble, and simulate programs for VBC1, read the EASY1 Tutorial in Appendix D, which shows you how to use EASY1.

VBC1 has only eight instructions. The data path unit in this experiment will execute the following three VBC1 instructions:

`IN DR ; 4 SWs`	input data (DI) from 4 SWs (switches) to R0 or to R1
`OUT DR ; 4 LEDs`	output data (DOUT) to 4 LEDs from R0 or from R1
`MOV DR,SR`	move R0 to R0, or move R0 to R1, or move R1 to R0, or move R1 to R1

Note: A semicolon precedes a comment; DR stands for destination register; SR stands for source register.

In this experiment, you will run the assembly language program shown in Program E12.1 to see if your circuit executes the program correctly as you supply the proper values for the control signals *M*1, *M*2, *LOAD_R*0, *LOAD_R*1, and *LOAD_OP*, and the input values shown in the comments.

Recommended Pre-Lab:

1. Create a new project named MUX_1, and write complete VHDL code for a single 2-to-1 MUX using a conditional signal assignment. Make the input and output signal names the same as those used by module 1 in Figure E12.1 using just scalar signals, not vector signals. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Create a new project named Reg_R0, and write complete VHDL code for a single loadable D flip-flop using a conditional signal assignment. Make the input and output signal names the same as those used by module 2 in Figure E12.1 using just scalar signals, not vector signals. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them.
3. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code for the data path unit for VBC1 shown in Figure E12.1 using only conditional signal assignments via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
3. Analyze the data path unit to determine the values that must be supplied by the control signals *M*1, *M*2, *LOAD_R*0, *LOAD_R*1, and *LOAD_OP* to execute each of the following instructions: IN R0, IN R1, OUT R0, OUT R1, MOV R0, R0, MOV R0,R1, MOV R1,R0, and MOV R1,R1. *Examples*: The IN R0 instruction is executed only when $M1 = 1$ and $LOAD\_R0 = 1$. The MOV R1,R0 instruction is executed only when $M2 = 0$, $M1 = 0$, and $LOAD\_R1 = 1$.
4. You need to verify that your VHDL design for the data path unit works correctly for each of the instructions IN, OUT, and MOV as you manually supply the proper control signal values and input values. Fill in Table E12.1 for each of the instructions in the assembly language program shown in Program E12.1 and repeated in Table E12.1.

TABLE E12.1

Board input →	SW7	SW6	SW3	SW2	SW1	SW0	BTN3	BTN2	BTN1	BTN0
Circuit signal →	*M*2	*M*1		*DI*(3:0)			*RST*	*LOAD_R1*	*LOAD_R0*	*LOAD_OP*
Instruction ↓										
`IN R0 ; input 9`										
`OUT R0 ; to LEDs`										
`IN R0 ; input 15`										
`MOV R1,R0`										
`OUT R1 ; to LEDs`										
`IN R1 ; input 6`										
`OUT R1 ; to LEDs`										
`OUT R0 ; to LEDs`										

Use the following notation for your entries in Table E12.1: (a) use a 1 to represent a slide switch that is pushed forward; (b) use a 0 to represent a slide switch that is pulled back; (c) use a don't care (X) for each slide switch input that does not contribute to the execution of the instruction; (d) use a 1 to represent a push-button switch that must be pressed and released to perform the instruction; (e) use a 0 to represent a push-button switch that is not pressed to perform the instruction. Your final task is to verify that your VHDL circuit design for VBC1 provides the same results you observed when you single stepped through the instructions in Program E12.1 using EASY1. If your hardware design fails to provide the same results as EASY1, then you either made a mistake assigning pins or you may have design errors in your VHDL code that you must find and fix. To start, press and release BTN3, which resets the loadable D flip-flops. If you make a mistake entering values, simply press and release BTN3, then single step through the instructions from the beginning.

5. Program/Table E12.2 shows a program for VBC1 for a robot eye. For practice, hand assemble all the instructions in Program/Table E12.2. Write the values in equation form for just the required inputs and control signals (I/C signals) for each instruction in Program E12.2 for your data path unit design for VBC1.
6. Enter Program E12.2 into EASY 1 and single step through the program to see how it works. Check your hand-assembled machine code with that produced by EASY1.
7. Using the switches and push buttons for the required inputs and control signals, single step through Program E12.2 via your VBC1 data path unit design to confirm that your hardware design and EASY1 perform in the same way.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is the execution of the simple program in Table E12.1 with your VBC1 design.
2. Include Table E12.1 filled in correctly.
3. Include the complete VHDL code for your VBC1 design.
4. Include a printout of the Edit Constraints (Text) for your VBC1 design, which is generated by running Edit Constraints (Text).

PROGRAM/TABLE E12.2 Robot eye program, hand assembly, and just required I/C signals

; VBC1 Exp 12 Program 2 Robot Eye	Hand assembly	Just required inputs and control signals
in r0 ; input 0001 via 4 slide switches	10100000	$DI = 0001, M1 = 1, LOAD\_R0 = 1$
mov r1,r0 ; copy r0 into r1	00010000	$M1 = M2 = 0, LOAD\_R1 = 1$
out r1 ; output 0001 to LEDs		
in r1 ; input 0010 via 4 slide switches		
out r1 ; output 0010 to LEDs		
in r0 ; input 0100 via 4 slide switches		
mov r1,r0 ; copy r0 into r1		
out r1 ; output 0100 to LEDs		
in r1 ; input 1000 via 4 slide switches		
out r1 ; output 1000 to LEDs		
out r0 ; output 0100 to LEDs		
in r1 ; input 0010 via 4 slide switches		
out r1 ; output 0010 to LEDs		
in r1 ; input 0001 via 4 slide switches		
mov r0,r1 ; copy r1 into r0		
out r0 ; output 0001 to LEDs		

5. Include an assembled printout of the assembly language program with comments in Program E12.1 on the screen or GUI (graphical user interface) for EASY1 in task 4. Make sure that the total GUI for EASY1 is available on your printout.
6. Include the hand assembly and just the required input and control signals for each instruction in Program/Table E12.2 for the robot eye program for VBC1 in task 5.
7. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
8. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 13: DESIGNING AND TESTING VBC1 (INSTRUCTION MEMORY UNIT)

In this experiment, you will practice working with instruction memory and a program counter.

This is the second in a series of projects to design and test VBC1. This is a stand-alone project—that is, do not add Experiment 12 to this design. Experiment 13 consists of the annotated schematic shown in Figure E13.1.

Figure E13.1 shows the instruction memory unit and part of the control unit for VBC1. The switch input circuits are not shown in the circuit diagram to simplify the drawing. Graphic symbols are shown for the single LEDs and the 7-segment display. Labels for all the switches and LEDs are included in parentheses—for example, SW7. . .SW0, BTN3, BTN1, BTN0, LD7. . .LD0, DP. . .CA, and AN3. . .AN0. These labels are used to obtain the pin assignments for the Digilent board you elect to use—for example, a BASYS 2 board or a NEXYS 2 board.

Things you should notice about the schematic in Figure E13.1:

- The schematic represents the instruction memory unit and a part of the control unit for VBC1. This much of the circuit for VBC1 allows you to load data (or instructions) into

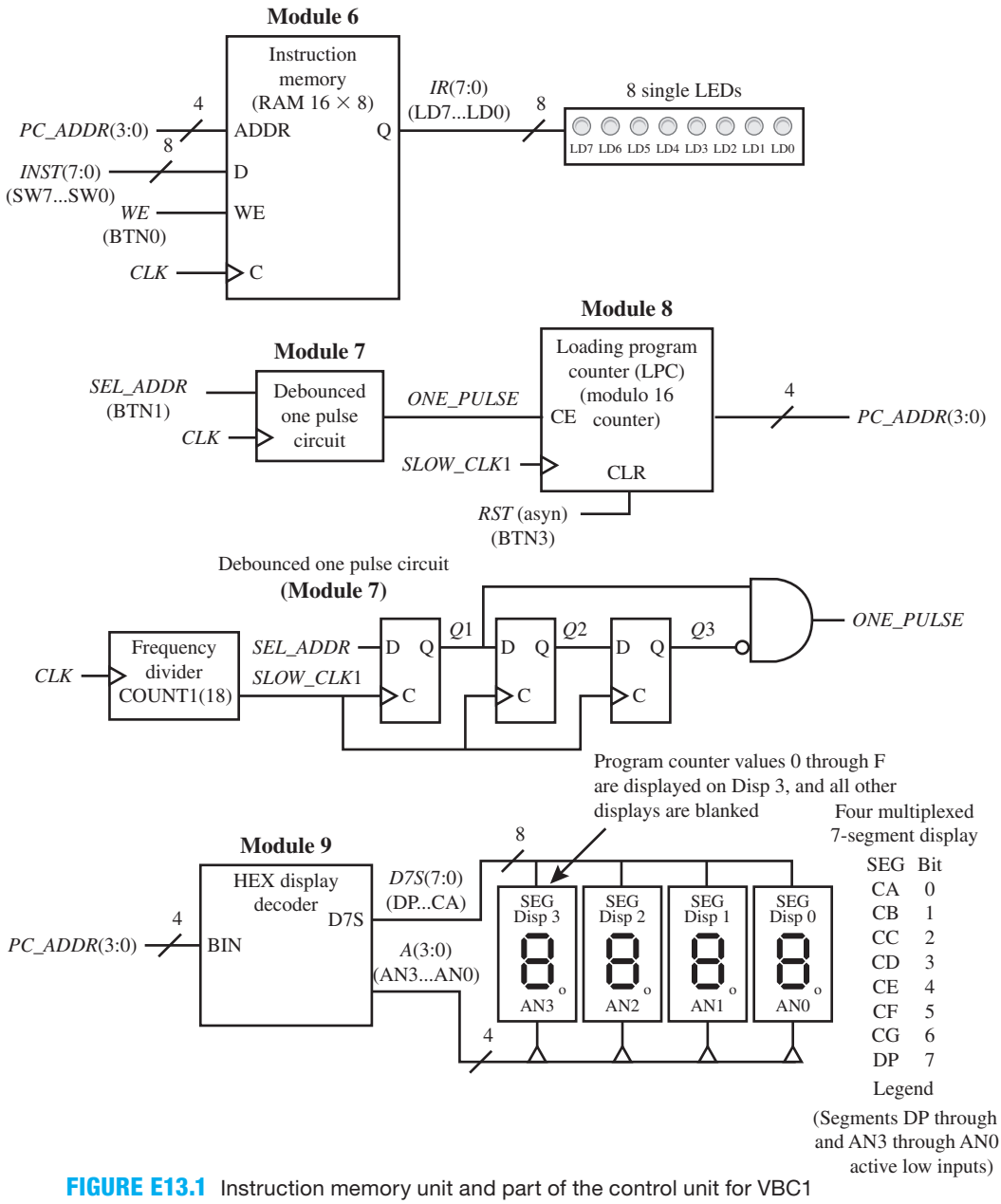


FIGURE E13.1 Instruction memory unit and part of the control unit for VBC1

instruction memory and verify that the data (or instructions) are stored at the appropriate address locations.

- The modules are numbered 6 through 9 in this experiment because the modules in the data path unit for VBC1 in Experiment 12 will be added to this experiment later, and this will prevent us from having to change the module numbers later.
- The external input and output signals—that is, the port signals for the entity in VHDL—are *INST*(7:0), *WE*, *CLK*, *SEL_ADDR*, . . . (you should be able to list the rest of them).
- The internal signals for the architecture in VHDL are *PC_ADDR*, *ONE_PULSE*, *COUNT*1, *SLOW_CLK*1, *Q*1, *Q*2, and *Q*3.

- Module 6 is the instruction memory for storing instructions for VBC1. Instructions are loaded via eight slide switches at the address specified by *PC_ADDR* when *WE* is enabled by a push-button switch. The output of the instruction memory—that is, the instruction register—is driving eight single LEDs.
- Module 7 provides a single pulse at its output—that is, signal *ONE_PULSE*—each time a push-button switch is pressed at its input—that is, signal *SEL_ADDR*.
- Module 8 is the loading program counter (LPC) for VBC1 using the CE input to prevent a gated clock circuit; that is, in general, if a clock net is sourced by a combinational pin, the circuit is called a gated clock. Using a gated clock is not considered a good design practice.
- Module 9 is a HEX display decoder for displaying the address of the loading program counter via a 7-segment display.
- After instructions are loaded, the contents of instruction memory may be verified by selecting the address of the LPC and observing the contents of the instruction memory via the eight single LEDs.

If you design the instruction memory unit and part of the control unit for VBC1 in Figure E13.1, you can manually load instructions (write instruction into memory) at addresses 0 through F. You can then reset the LPC and read the instruction at the respective addresses 0 through F.

Program E13.1 can be loaded into instruction memory (even though it will not execute properly for this limited design of VBC1). You may elect to hand assemble the program to obtain the machine code or use EASY1 to obtain the assembled program—that is, the machine code.

PROGRAM E13.1

```
;VBC1 Exp 12, 13, and 15 Program 1
;Testing IN, OUT, and MOV instructions
IN R0       ; input 9 via 4 slide switches
OUT R0      ; output contents of R0 to LEDs
IN R0       ; input 15 via 4 slide switches
MOV R1,R0   ; move contents of R0 to R1
OUT R1      ; output contents of R1 to LEDs
IN R1       ; input 6 via 4 slide switches
OUT R1      ; output contents of R1 to LEDs
OUT R0      ; output contents of R0 to LEDs
```

After Program E13.1 is loaded into instruction memory, you can verify that the program is loaded by resetting the LPC and incrementing the LPC via the signal *SEL_ADDR*.

To test all addresses, you can load the hexadecimal values 00 (binary value 0000 0000), 11 (binary value 0001 0001), 22, 33, . . . EE, FF in the memory locations 0 through F. After loading the values, you can verify that these values are at the specified locations.

To investigate the design of the instruction memory using a process, read Chapter 13, Section 13.2 (Designing an Instruction Memory).

To investigate the design of flip-flop circuits using conditional signal assignments, read Chapter 12, Section 12.4 (Designing Loadable Register Circuits).

Recommended Pre-Lab:

1. Create a new project named LPC, and write complete VHDL code for the loading program counter (a modulo 16 counter) using a conditional signal assignment with the arithmetic method. Make the input and output signal names the same as those used by module 8 in Figure E13.1 except for the signals *ONE_PULSE* and *SLOW_CLK*1. Change *SLOW_CLK*1 to *CLK* and remove the signal *ONE_PULSE* because this will be a stand-alone design—that is, the frequency divider and the debounced one-pulse circuit will not be used in this project. Run a simulation to verify your design works. If your design does not work, you

must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code for the memory unit and part of the Control Unit for VBC1 using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4 for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Use a process for the instruction memory (module 6).
3. Use a separate conditional signal assignment to design each flip-flop for the debounced one-pulse circuit (module 7), and use a Boolean equation to provide the gate output *ONE_PULSE*.
4. Use a conditional signal assignment with the arithmetic method to design the loading program counter (module 8).
5. Use a selected signal assignment to design the display hexadecimal decoder (module 9).
6. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. To verify that your design works, reset the loading program counter (LPC), and you should see 0 on Disp 3 of the 7-segment display. Each time you press and release BTN1 (SEL_ADDR), your design should cycle through all the LPC addresses—that is, 0 through F. If your hardware design does not operate in this manner, then you either made a mistake assigning pins or you may have design errors in your VHDL code that you must find and fix.
8. For information on how to manually load a program into instruction memory, see Appendix E. To load the memory with instructions, you can enter machine code (8 bits) for an instruction (*INST*) at each memory address by setting the slide switches and pressing and releasing BTN0 (*WE* or Write Enable). Press and release BTN1 (*SEL_ADDR*) to increment the PC to the next address. You may hand assemble the machine code for Program E13.1, or you may use EASY1 to obtain the machine code for Program E13.1. Manually load Program E13.1 into Instruction Memory.
9. After you manually load Program E13.1 in your VBC1 design, press the Reset button and verify that the program is stored in the instruction memory. This requires pressing BTN3 to reset VBC1 and then pressing BTN1 in succession to observe the contents of instruction memory via the single LEDs, LD7 . . . LD0).
10. To test all addresses in your instruction memory, load the hexadecimal values 00 (binary value 0000 0000), 11 (binary value 0001 0001), 22, 33, . . . EE, FF in the memory locations 0 through F. When you press and release BTN3 (*RST*) and then cycle through all the LPC addresses by pressing and releasing BTN1 (*SEL_ADDR*), you should observe that the values 00, 11, 22, 33, . . . EE, FF are stored in the memory locations 0 through F, respectively.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information:

course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is to demonstrate task 9 with your VBC1 design.

2. Print out your explanation of the purpose of the enumerated data type that is necessary in the design of the instruction memory.
3. Include the complete VHDL code for your VBC1 design.
4. Include a printout of the Edit Constraints (Text) for your VBC1 design, which is generated by running Edit Constraints (Text).
5. For task 9, include an assembled printout of the assembly language program with comments in Program E13.1 on the screen or GUI (graphical user interface) for EASY1. Make sure that the total GUI for EASY1 is available on your printout.
6. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
7. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 14: DESIGNING AND TESTING VBC1 (MONITOR SYSTEM)

In this experiment, you will practice working with a multiplexed display system.

This is the third in a series of projects to design and test VBC1. Experiment 14 consists of the annotated schematics shown in Figures E14.1 and E14.2. You should design and test the circuit in Figure E14.1 first. After you get the circuit in Figure E14.1 working, add the circuit in Figure E14.2 to your project.

Figure E14.1 shows the monitor system for VBC1. The push-button switch input circuit is not shown in the circuit diagram to simplify the drawing. A graphic symbol is shown for the 7-segment LED display. Labels for the push-button switch and LEDs are included in parentheses—for example, BTN3, DP. . .CA, and AN3. . .AN0. These labels are used to obtain the pin assignments for the Digilent board you elect to use—for example, a BASYS 2 board or a NEXYS 2 board.

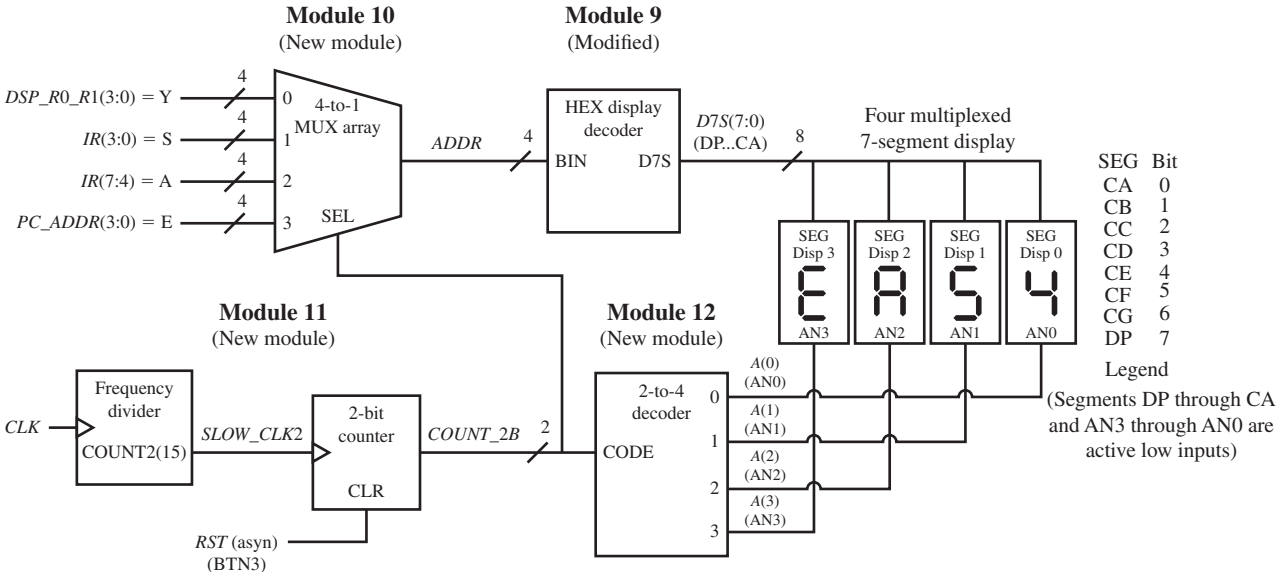


FIGURE E14.1 Monitor system for VBC1

Things you should notice about the schematic in Figure E14.1:

- The schematic represents the monitor system for VBC1. This circuit for VBC1 allows you to display and monitor the PC on Disp 3, the IR on Disp 2 and Disp 1, and either Register 0 or Register 1 on Disp 0. For testing purposes, the values for PC, IR, and the register are supplied as 4-bit constants to generate EA54, which can be interpreted as the word EASY by considering that each character represents a letter.
- The logic modules are numbered consecutively 10 through 12 continuing from Figure E13.1 in Experiment 13.
- The external input and output signals—that is, the port signals for the entity in VHDL—are *CLK*, *RST*, *D7S*(7:0), and *A*(3:0).
- The internal signals for the architecture in VHDL are *ADDR*, *COUNT*2, *SLOW_CLK*2, and *COUNT_2B*.
- Module 9 is the HEX display decoder that provides the decoded values to the LED segments of the 7-segment display. Bit 7 is used to turn off the decimal point DP.
- Module 10 is the 4-to-1 MUX array for selecting the character that will be displayed.
- Module 11 is a 2-bit counter driven by a slowed-down (lower frequency) version of the clock *CLK*. The counter selects the character to be displayed via the MUX array and also selects the particular 7-segment LED display—that is, Disp 3, Disp 2, Disp 1 or Disp 0—that displays the character via the decoder.
- Module 12 is a 2-to-4 decoder, which enables the appropriate 7-segment LED display in the sequence Disp 0, Disp 1, Disp 2, Disp 3, Disp 0 . . . via the 2-bit counter.

If you design the multiplexed display system for VBC1 in Figure E14.1, you can display any 4-bit word that consists of the characters 0 through F. For this project, the 4-bit word EASY is displayed. To change to different characters, simply modify your code by supplying the desired characters as constants.

Figure E14.2 shows an addition to the instruction memory unit and part of the control unit for VBC1 covered in Experiment 13. The switch input circuits are not shown in the circuit diagram to simplify the drawing. Labels for all the switches are included in parentheses—for example, SW7. . .SW0, BTN0, BTN1, and BTN3. Remember, these labels are used to obtain the pin assignments for the Digilent board you elect to use.

Things you should notice about the schematic in Figure E14.2:

- The schematic represents an addition to the instruction memory unit and part of the control unit for VBC1.
- The new logic module is numbered 13, where the number is continued consecutively from Figure E14.1. The VHDL code for modules 6, 7, and 8 may be copied from Experiment 13.
- The external input and output signals—that is, the port signals for the entity in VHDL—are *SW*(7:0), *LOAD_MEM*, *WEI*, . . . (you should be able to list the rest of them).
- The internal signals for the architecture in VHDL are *PC_ADDR*, *INST*, *WE*, and *ONE_PULSE*.
- Module 6 is the instruction memory for storing instructions for VBC1. Instructions are loaded via *INST* by seven slide switches at the address specified by *PC_ADDR* when *WE* is enabled via *LOAD_MEM* by a push-button switch and *WEI* by a push-button switch.
- Module 7 provides a single pulse at its output—that is, signal *ONE_PULSE*—each time a push-button switch is pressed at its input—that is, signal *SEL_ADDR*.
- Module 8 is the loading program counter for VBC1 using the CE input to prevent a gated clock; that is, in general, if a clock net is sourced by a combinational pin, the circuit is called a gated clock. Using a gated clock is not considered a good design practice.
- Module 13 is a load memory circuit that requires the user to press and hold push-button switch BTN2 to provide the signal *LOAD_MEM* that selects an instruction from the slide switches *SW*(7:0) while simultaneously pressing push-button switch BTN0 to provide the

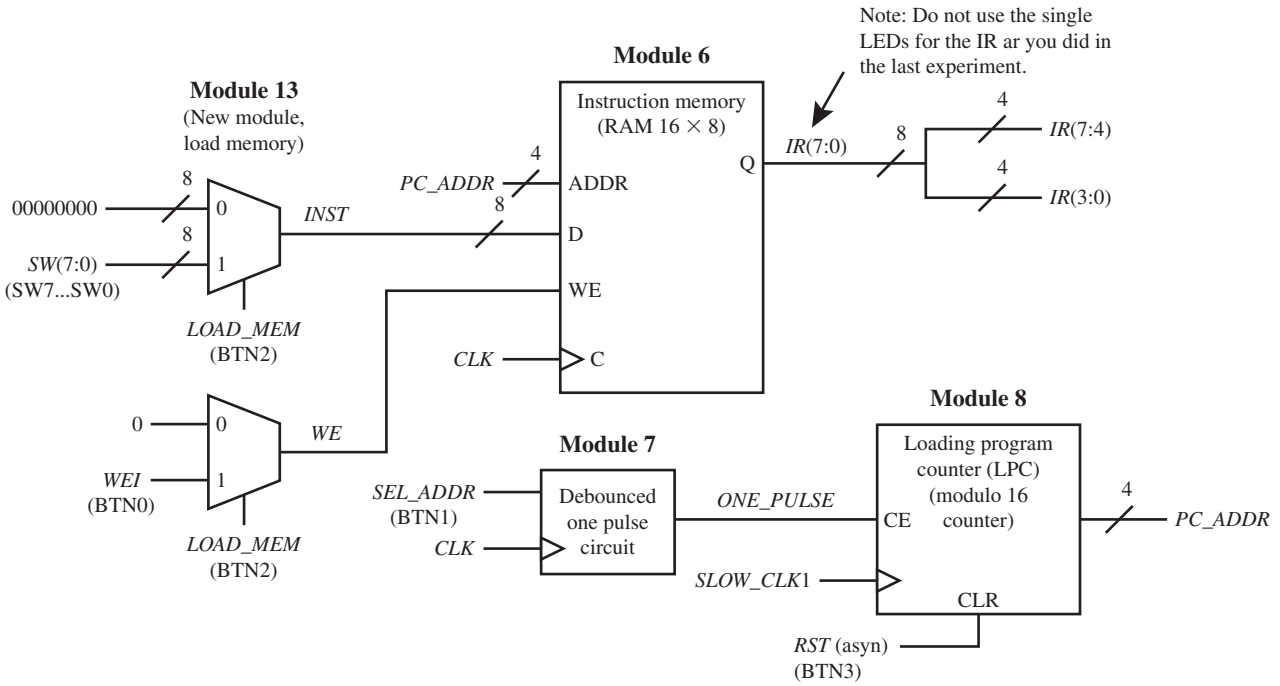


FIGURE E14.2 Addition to the instruction memory unit and part of the control unit for VBC1

signal *WEI* that writes the instruction into instruction memory. When the push-button switch BTN2 is not pressed, this frees up the slide switches *SW*(7:0) to be used for other input requirements for VBC1.

If you add the circuit in Figure E14.2 to this project, the PC address will be displayed in hexadecimal on Disp 3, and the *IR* will be displayed as two hexadecimal numbers on Disp 2 (MSD) and Disp 1 (LSD). To cause this to occur be sure to change inputs 3, 2, and 1 of the 4-to-1 MUX array to *PC_ADDR*(3:0), *IR*(7:4), and *IR*(3:0), respectively. Signal *DSP_R*0(3:0) is to remain the constant value of 4; modify the 2-to-4 decoder (module 12) so that Disp 0 is turned off.

Recommended Pre-Lab:

1. Create a new project named Decoder_2_4, and write complete VHDL code for the 2-to-4 Decoder with active high outputs using a conditional signal assignment. Make the input and output signal names the same as those used by module 12 in Figure E14.1. This will be a stand-alone design—that is, module 11 will not be used. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL program. Be sure to place all modules in the code section in numerical order.

2. Start a new project and add module 9 from Experiment 13 to the new project. Modify module 9 by removing output signal *A*(3:0).
3. Add module 10 (4-to-1 MUX array) using a conditional signal assignment with the fixed input values E at MUX array input 3, A at MUX array input 2, 5 (5 is the closest character that resembles an "S") at MUX array input 1, and 4 (4 is the closest character that resembles a "Y") at MUX array input 0.
4. Add module 11 (frequency divider/2-bit counter). For both the frequency divider and the 2-bit counter, use a conditional signal assignment with the arithmetic method. Use a Boolean equation for signal *SLOW_CLK*2.
5. Add module 12 (2-to-4 Decoder) using a conditional signal assignment.
6. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
7. You should observe that the four 7-segment multiplexed display shows the word EASY without flickering and without bleeding. If your design does not operate in this manner, then you either made a mistake assigning pins or you may have design errors in your VHDL code that you must find and fix.
8. Change *SLOW_CLK*2 to *COUNT*2(18), then repeat tasks 6(b) and 6(c). You should observe that the four 7-segment multiplexed display flickers as the word EASY is displayed.
9. Change *SLOW_CLK*2 to *COUNT*2(2), then repeat tasks 6(b) and 6(c). You should observe that the four 7-segment multiplexed display bleeds as the word EASY is being displayed.
10. Change *SLOW_CLK*2 back to *COUNT*2(15), and start adding Figure E14.2 to your project.
11. Add module 6 (instruction memory) from Experiment 13 to your project.
12. Add module 7 (debounced one-pulse circuit) from Experiment 13 to your project.
13. Add module 8 (loading program counter) from Experiment 13 to your project.
14. Add module 13 (load memory) to your project.
15. Note that you must make *INST* and *WE* internal signals, and you must make *SW*, *LOAD_MEM*, and *WEI* external signals and provide their pin numbers in the .ucf file.
16. Change the fixed input values for module 10 (4-to-1 MUX array) to the VHDL signals *PC_ADDR* at MUX input 3, *IR*(7:4) at MUX input 2, and *IR*(3:0) at MUX input 1. Modify the 2-to-4 decoder (module 12) so that Disp 0 is turned off. With these changes, the PC address will be displayed in hexadecimal on Disp 3, and the IR will be displayed as two hexadecimal numbers on Disp 2 (MSD) and Disp 1 (LSD).
17. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
18. For information on how to manually load a program into instruction memory, see Appendix E. To test your VBC1 design, load the hexadecimal values 00 (binary value 0000 0000), 11 (binary value 0001 0001), 22 (binary value 0010 0010), 33, . . . EE, FF in the memory locations 0 through F. When you press and release BTN3 (*RST*) and then cycle through all the PC addresses by pressing and releasing BTN1 (*SEL_ADDR*), you should observe that the values 00, 11, 22, 33, . . . EE, FF are stored in the memory locations 0 through F, respectively. If your design does not operate in this manner, then you either

made a mistake assigning pins or you may have design errors in your VHDL code that you must find and fix.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is to demonstrate task 18 with your VBC1 design.
2. Briefly discuss what causes flickering and what causes bleeding of the symbols being displayed on the multiplexed display in this experiment.
3. Include the complete VHDL code for your VBC1 design.
4. Include a printout of the Edit Constraints (Text) for your VBC1 design, which is generated by running Edit Constraints (Text).
5. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
6. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 15: DESIGNING AND TESTING VBC1 (INSTRUCTION DECODER)

In this experiment, you will practice working with modules to form a system controlled by an instruction decoder .

This is the fourth in a series of projects to design and test VBC1. Experiment 15 consists of the annotated schematics shown in Figures E15.1 and E15.2.

Figure E15.1 shows the modified data path unit, the modified monitor system, the modified instruction memory unit, and part of the control unit for VBC1. Modifications are shown by arrows in Figure E15.1. Switch input circuits are not shown in the circuit diagram to simplify the drawing. A graphic symbol is shown for the 7-segment LED display. Labels for the switches and LEDs are included in parentheses—for example, BTN3, LD3. . .LD0, DP. . .CA, SW7. . .SW0, BTN2, BTN0, BTN1, and AN3. . .AN0. These labels are used to obtain the pin assignments for the Digilent board you elect to use—for example, a BASYS 2 board or a NEXYS 2 board.

Figure E15.2 shows the instruction decoder, speed, select R0 or R1, data input, and blank display circuits for VBC1. Switch input circuits are not shown in the circuit diagram to simplify the drawing. Labels for the switches are included in parentheses—for example, BTN2, BNT3, SW4, and SW3. . .SW0. These labels are used to obtain the pin assignments for the Digilent board you elect to use.

In Figure E15.2, module 14 (instruction decoder) is used to decode the instructions IN, OUT, and MOV. Module 15 (speed) allows you to load the instruction memory or single step through the instruction memory. Module 16 (select R0 or R1) allows you to select either R0 or R1 and display the value on Disp 0 of the four 7-segment display. Module 16 gives you the ability to observe the contents of register R0 or R1 as you single step through a sequence of instructions to check proper instruction execution. Module 17 (data input) provides a buffer array for the data input for the four slide switches with the signal name *SW*(3:0). Module 18 (blank display) momentarily blanks (shuts off) the display when VBC1 is reset.

When the circuits in Figures E15.1 and E15.2 are combined, you can load the instruction memory, execute the IN, OUT, and MOV instructions, and single step through a program that only uses these instructions.

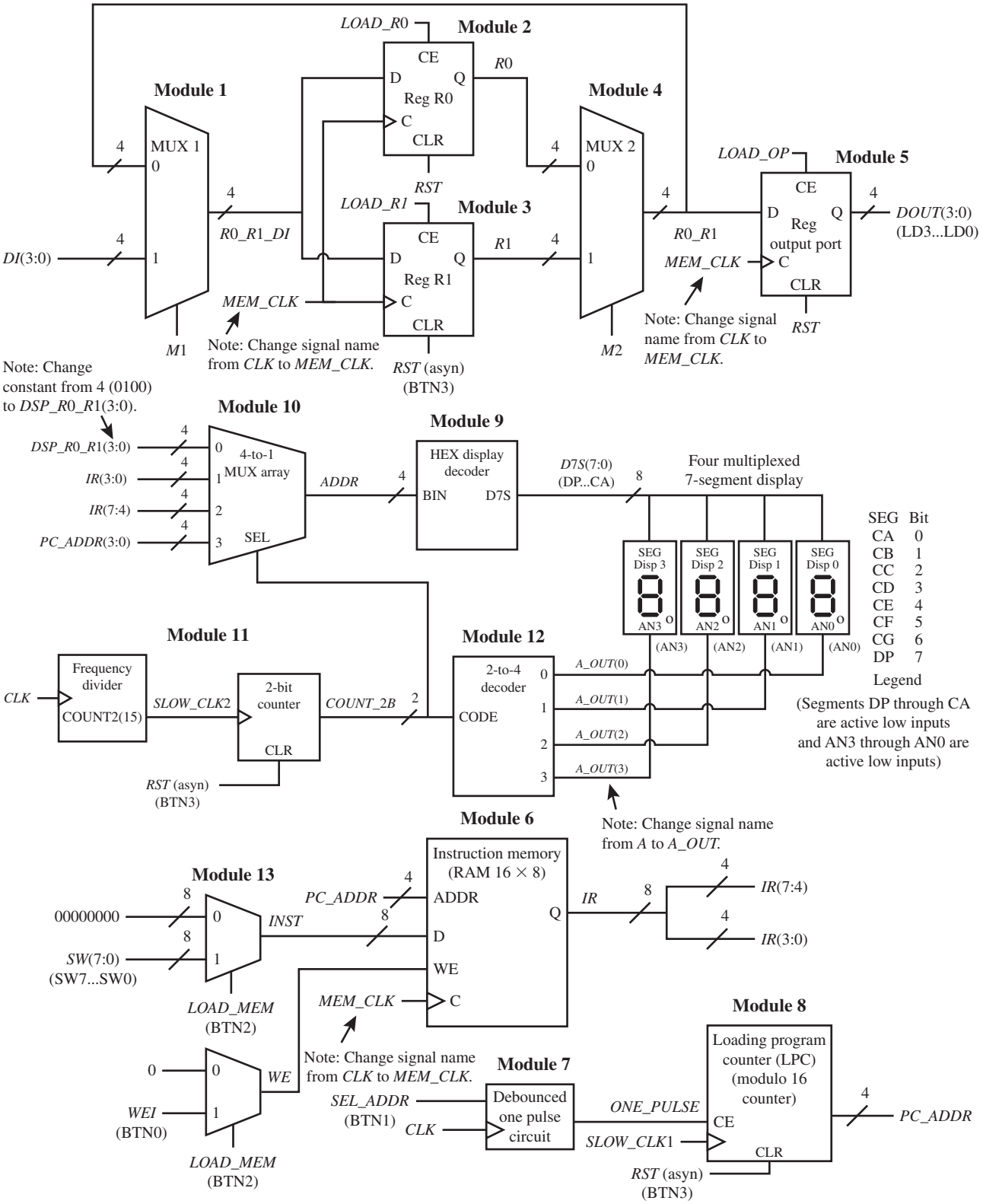


FIGURE E15.1 Modified data path unit, modified monitor system, modified instruction memory unit, and part of the control unit for VBC1

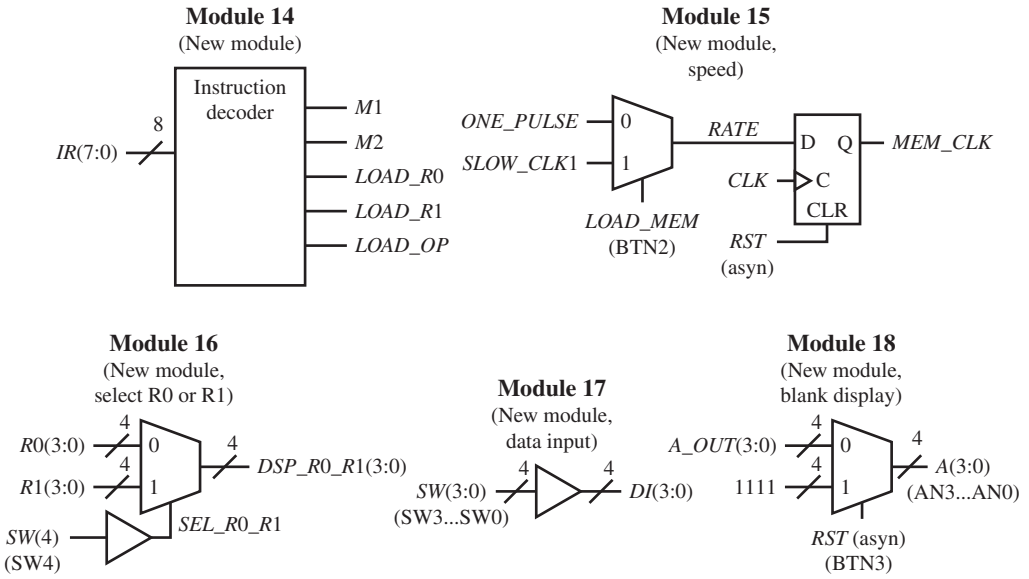


FIGURE E15.2
Instruction decoder, speed, select R0 or R1, data input, and blank display circuits for VBC1

Recommended Pre-Lab:

1. Create a new project named INST_DECODER, and write complete VHDL code for the instruction decoder using a process with a case statement. Make the input and output signal names the same as those used by module 14 in Figure E15.2. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project, and add modules 1 through 5 in Experiment 12 and modules 6 through 13 in Experiment 14 to the new project. Make the modifications indicated by the arrows in the annotated schematic in Figure E15.1.
3. Add module 14 (instruction decoder) using Procedure ID as presented in Chapter 15, Section 15.4. For this experiment, only include the control signals in the instruction decoder that are required for the instructions IN, OUT, and MOV.
4. Add module 15 (speed) using conditional signal assignments. This module allows VBC1 to load instructions or to single step through a set of instructions.
5. Add module 16 (select *R*0 or *R*1) using a conditional signal assignment and a Boolean equation. This module allows VBC1 to display either R0 or R1.
6. Add module 17 (data input) using a Boolean equation. This module allows VBC1 to input four bits of data via four slide switches.
7. Add module 18 (blank display) using a conditional signal assignment. This module allows VBC1 to blank the display when reset is asserted.

8. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
9. Use the EASY1 editor to enter the instructions shown in Program E15.1. Use the EASY1 simulator to single step through the program so that you understand what it does.

PROGRAM E15.1

```
;VBC1 Exp 12, 13, and 15 Program 1
;Testing IN, OUT, and MOV instructions
IN R0       ; input 9 via 4 slide switches
OUT R0      ; output contents of R0 to LEDs
IN R0       ; input 15 via 4 slide switches
MOV R1,R0   ; move contents of R0 to R1
OUT R1      ; output contents of R1 to LEDs
IN R1       ; input 6 via 4 slide switches
OUT R1      ; output contents of R1 to LEDs
OUT R0      ; output contents of R0 to LEDs
```

10. For information on how to manually load a program into memory, see Appendix E. Manually load the machine code for the instructions in Program E15.1 into the memory of VBC1, and execute the instruction sequence by single stepping in the same manner that you did with EASY1. If VBC1 does not provide the same results as EASY1 for each instruction, then you either made a mistake in entering the machine code or you may have design errors in your VHDL code that you must find and fix.
11. For information on how to initialize memory at startup, see Appendix E. This process loads the memory of VBC1 at startup, so you do not have to manually load it. Single step through the memory of VBC1 to verify that the machine code is properly loaded for Program E15.1 in the addresses 0 through 15 just as they are in the memory of EASY1.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is the execution of Program E15.1 with your VBC1 design.
2. Write and include a conditional signal assignment for the control signal *M*2 in the instruction decoder truth tables just for the instructions IN, OUT, and MOV. This is an alternate way of generating the control signal *M*2.
3. Include the complete VHDL code for your VBC1 design.
4. Include a printout of the Edit Constraints (Text) for your VBC1 design, which is generated by running Edit Constraints (Text).
5. Explain the purpose of the D flip-flop in module 15.
6. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
7. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 16: DESIGNING AND TESTING VBC1 (ARITHMETIC LOGIC UNIT)

In this experiment, you will practice working with modules to form the complete data path for VBC1.

This is the fifth in a series of projects to design and test VBC1. Experiment 16 consists of the annotated schematics shown in Figures E16.1 and E16.2.

Figure E16.1 shows module modifications made to improve VBC1 by controlling the monitor system—that is, the multiplexed display system. Switch input circuits are not shown in the circuit diagram to simplify the drawing. Labels for the switches are included in parentheses—for example, SW5, BTN2, BTN3, and SW6. These labels are used to obtain the pin assignments for the Digilent board you elect to use—for example, a BASYS 2 board or a NEXYS 2 board.

Module 11 (redirected 2-bit counter) is modified to only display the PC and the IR when you load the machine code, otherwise you can elect to turn Disp 0 on or off via SW5. Module 12 (2-to-4 decoder) is modified so that you can elect to turn the monitor system on or off via SW6, when single stepping through a program or when running a program in the next experiment at an observable clock speed.

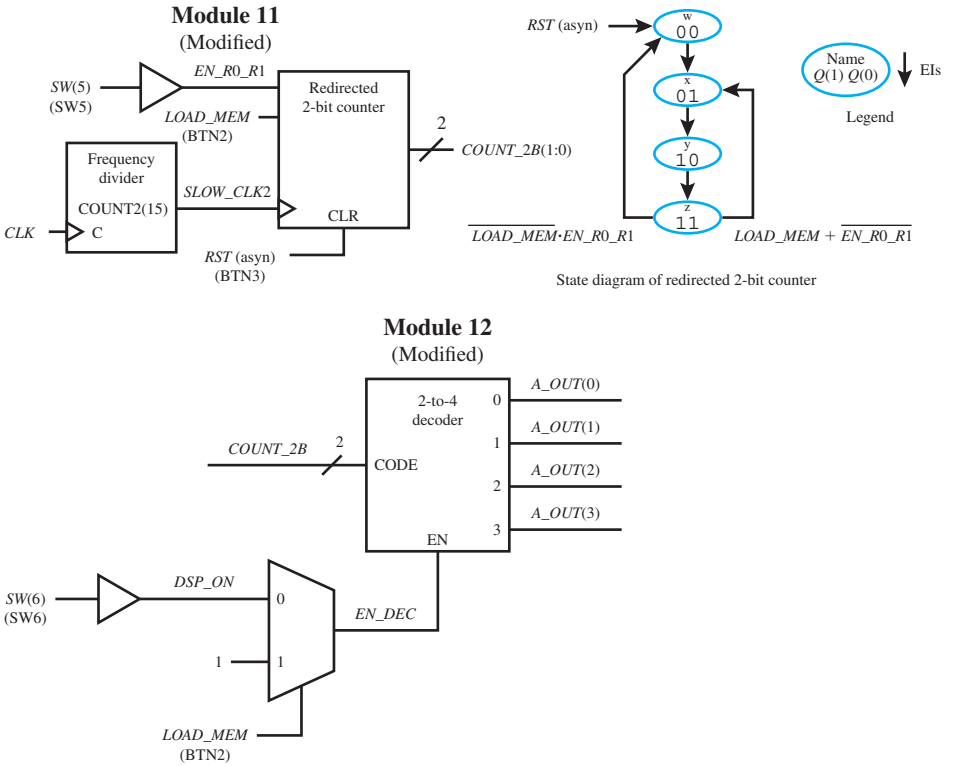


FIGURE E16.1 Module modifications to improve VBC1 by controlling the monitor system

The circuit diagram in Figure E16.2 shows the complete data path circuit for VBC1. This circuit contains a simple ALU (module 19) that performs the LOADI, ADDI, ADD, and SR0 instructions. Modules 20, 21, and 22 provide the necessary data bus steering for the data path circuit.

Listing E16.1 shows a VHDL template for the ALU instructions LOADI and ADDI.

When the circuits in Figures E16.1 and E16.2 are completed and the VHDL code for the additional instructions are included in the instruction register, you can load the instruction memory and execute the IN, OUT, MOV, LOADI, ADDI, ADD, and SR0 instructions and single step through a program that only uses these instructions. You can also shut off the monitor system via SW6, while

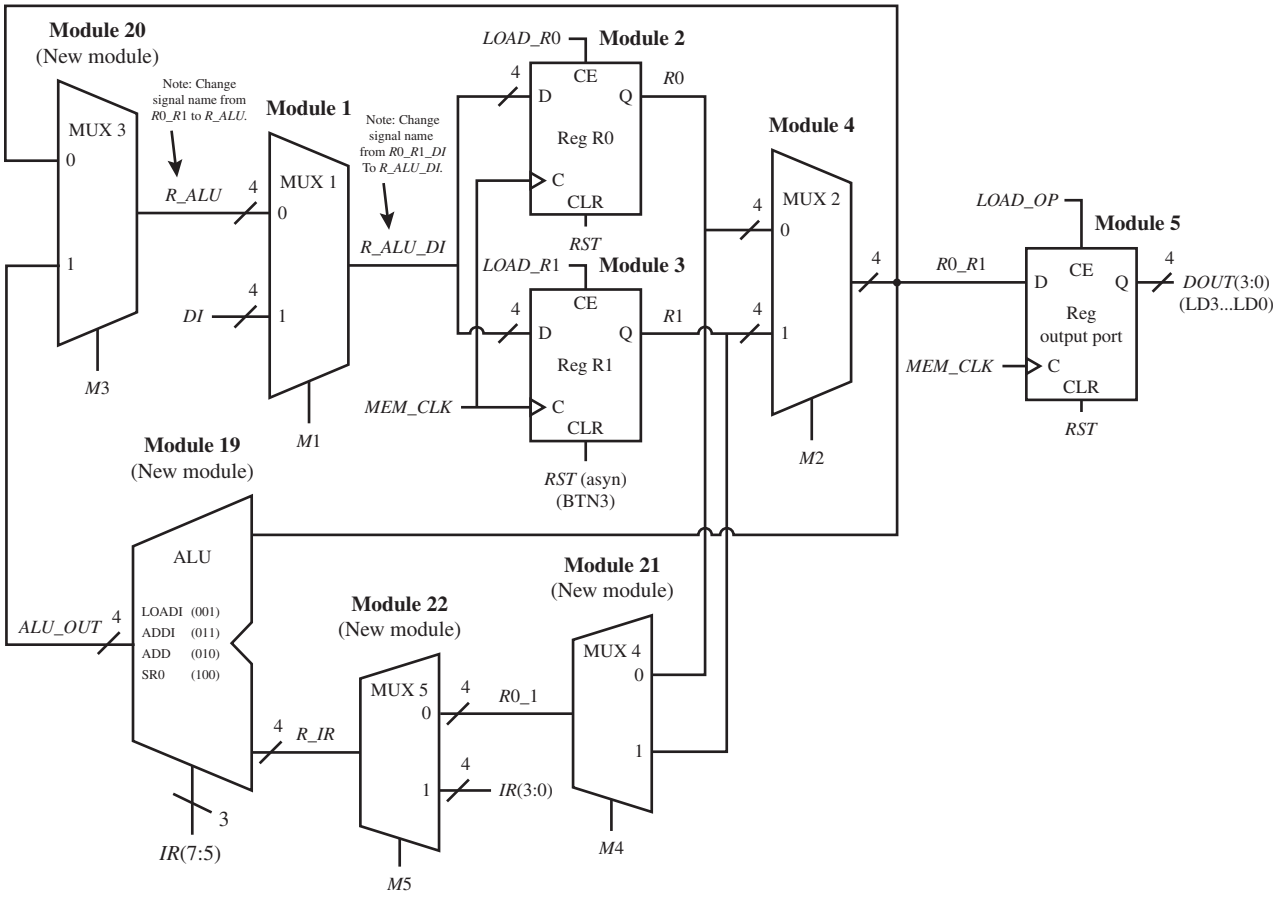


FIGURE E16.2 Complete data path circuit for VBC1

LISTING E16.1 A VHDL template for the ALU instructions LOADI and ADDI

```
--Module 19 code, process for a simple ALU
process (ir(7 downto 5), r0_r1, r_ir)
begin
   --put default ALU output value here, i.e., alu_out <= "0000"
   --default to prevent creating latches
   case ir(7 downto 5) is -- the OPCODE for each ALU instruction is
                          -- in bits (7:5) in the IR

      --the LOADI instructions have the OPCODE 001
      when "001" => -- determine the ALU output equation for the
                    -- LOADI instruction and enter it here

      --the ADDI instructions have the OPCODE 011
      when "011" => -- determine the ALU output equation for the
                    -- ADDI instruction and enter it here

      when others => null;
   end case;
end process;
```

single stepping through a program. Only the PC and the IR are displayed by the monitor system when you load the machine code, otherwise you can elect to turn Disp 0 on or off via SW5.

Recommended Pre-Lab:

1. Create a new project named Redirected_Counter1, and write complete VHDL code for the redirected 2-bit counter, using the two process PS/NS method. Make the input and output signal names the same as those used by module 11(modified) in Figure E16.1 except for *SLOW_CLK*2. Change *SLOW_CLK*2 to *CLK* because this will be a stand-alone design—that is, the frequency divider that provides the signal for *SLOW_CLK*2 and the buffer for the switch input *SW*(5) will not be used. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Create a new project named Redirected_Counter2, and write complete VHDL code for the redirected 2-bit counter, using the AE method (algorithmic equation method). For the AE Method, write the D excitation equations by inspection using the state diagram via the Set OR Hold 1 equation, then draw the circuit and obtain the VHDL code from the circuit (see Chapter 9, Section 9.9, for a review). Make the input and output signal names the same as those used by module 11(modified) in Figure E16.1 except for *SLOW_CLK*2. Change *SLOW_CLK*2 to *CLK* since this will be a stand-alone design—that is, the frequency divider that provides the signal for *SLOW_CLK*2 and the buffer for the switch input *SW*(5) will not be used. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
3. The simulation waveform for your design for redirected counter 1 and your simulation waveform for redirected counter 2 should be identical when supplied by the same stimulus inputs. You should also observe that your design using the two-process PS/NS method requires roughly three times more lines of VHDL code than your design using the AE method.
4. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use Documentation Style M introduced in Chapter 12, Section 12.5.1, Listing 12.4 for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project and add Experiment 15 to the new project. Modify module 11 (redirected 2-bit counter) in your VBC1 design. Use the two-process PS/NS method or use the AE method to obtain the counter design for module 11 (redirected 2-bit counter).
3. Modify module 12 (2-to-4 decoder) in your VBC1 design by adding an enable input (EN) that enables the decoder's output when EN = 1, but disables the decoder's output when EN = 0. When BTN2 is pressed to load memory, the decoder is enabled via a 2-to-1 MUX. When you are not loading memory, the decoder's output is controlled by slide switch SW6.
4. Add module 19 (ALU) to your current design of VBC1. Use the template in Listing E16.1 for the ALU for LOADI and ADDI. Be sure to use the *R_IR* input to the ALU as the input for the immediate data for the LOADI instruction. This is a requirement imposed by the circuit design since *IR*(3:0) is supplied to the ALU via module 22.
5. Add module 20 (MUX 3), module 21 (MUX 4), and module 22 (MUX 5) to your design. Remember to change the signal name at the output of module 20 (MUX 3) from *R*0_*R*1 to *R_ALU*. Also change the signal name at the output of module 1 (MUX 1) from *R*0_*R*1_*DI* to *R_ALU_DI*.

6. Use the instruction decoder truth tables for the LOADI and ADDI instructions to obtain the equations for the required control signals *M*1, *M*2, *M*3, *M*4, *M*5, *LOAD_R*0, *LOAD_R*1 and *LOAD_OP*, and add these to module 14 (instruction decoder). Include default values for all control signals.
7. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
8. To test your design for the LOADI and ADDI instructions, use EASY1 to obtain the machine code for the assembly language program in Program E16.1.

PROGRAM E16.1

```
;VBC1 Exp 16 Program 1
;Testing LOADI and ADDI instructions
LOADI R0,3
OUT R0
ADDI R0,9
MOV R1,R0
OUT R1
```

Simulate the program so that you understand what it does. For information on how to manually load a program into memory, see Appendix E. Manually load the machine code into the memory of VBC1, and execute the instruction sequence by single stepping in the same manner that you do with EASY1. If VBC1 does not execute the instructions and provide the same results as EASY1, then you either made a mistake in entering the machine code or you may have design errors in your VHDL code that you must find and fix before continuing.

9. Modify module 19 (ALU) to support the ADD and the SR0 instructions. Modify module 14 (instruction decoder) to support the ADD and SR0 instructions.
10. Do the following: Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File. Download the programming file.
11. To verify that your design works for all of the instructions except JNZ, use EASY1 to obtain the machine code for the bouncing lights program in Program E16.2.

PROGRAM E16.2

```
;VBC1 Exp 16 Program 2
;Bouncing lights program, 2 left then 2 right
loadi r1,1
addi r1,3
mov r0,r1
out r0
add r0,r0
out r0
sr0 r0,r0
sr0 r0,r0
out r0
in r0    ; input 1 via 4 slide switches
out r0
```

Simulate the program so that you understand what it does. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E16.2 into the memory of VBC1 by initializing memory. Execute the instruction sequence by single stepping in the same manner that you do with EASY1. If VBC1 does not execute the instructions and provide the same results as EASY1, then you either made a mistake in loading the machine code or you may have design errors in your VHDL code that you must find and fix.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is the execution of the bouncing lights program in Program E16.2 with your VBC1 design.
2. Include the complete VHDL code for your VBC1 design.
3. Include a printout of the Edit Constraints (Text) for your VBC1 design, which is generated by running Edit Constraints (Text).
4. For task 11, include an assembled printout of the assembly language program with comments in Program E16.2 on the screen or GUI (graphical user interface) for EASY1. Make sure that the total GUI for EASY1 is available on your printout.
5. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
6. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 17: DESIGNING AND TESTING VBC1 (FINAL HARDWARE DESIGN FOR VBC1)

In this experiment, you will practice working with modules to form the complete circuit for VBC1.

This is the last in a series of projects to design and test VBC1. Experiment 17 consists of the annotated schematics shown in Figures E17.1 and E17.2.

In Figures E17.1 and E17.2, the switch input circuits are not shown in the circuit diagrams to simplify the drawings. Labels for the switches are included in parentheses—for example, SW7, BTN1, BTN2, and BTN3. These labels are used to obtain the pin assignments for the Digilent board you elect to use—for example, a BASYS 2 board or a NEXYS 2 board.

In the circuit in Figure E17.1, module 23 [running program counter (RPC)] is necessary for executing the JNZ instruction, as well as all the other instructions for VBC1. All the instructions for VBC1 are fetched and executed in just one clock cycle of the frequency of SPEED. Module 24 (proper address) selects the proper address for loading instructions via *PC_ADDR* or executing instructions via *PROG_A* (short for program address).

In the circuit in Figure E17.2, module 25 (run frequency) provides a frequency of *SLOW_CLK*3 to execute instructions at a speed that can be observed on the output LEDs. Module 15 (speed) allows VBC1 to either single step through a set of instructions (set SW7 = 1) or run a set of instructions (set SW7 = 0) at the frequency of *SLOW_CLK*3. The frequency of *SPEED* is the same as the frequency of *SLOW_CLK*3.

When VBC1 is running a set of instructions at the speed of *SLOW_CLK*3, it is easy to determine the execution time of each instruction or set of instructions. Knowing the frequency of *CLK* and *SLOW_CLK*3, you can calculate how long it takes each instruction to execute and therefore how long it takes a series of instructions to execute.

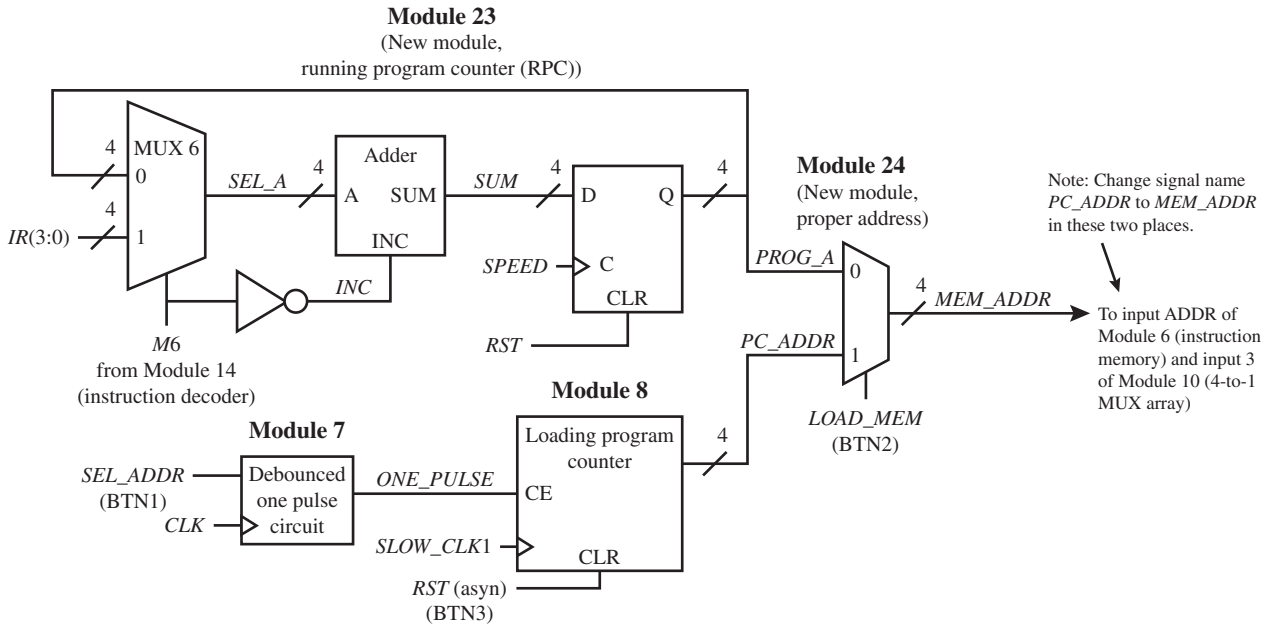


FIGURE E17.1 Modules added to provide the proper address for loading and executing instructions for VBC1

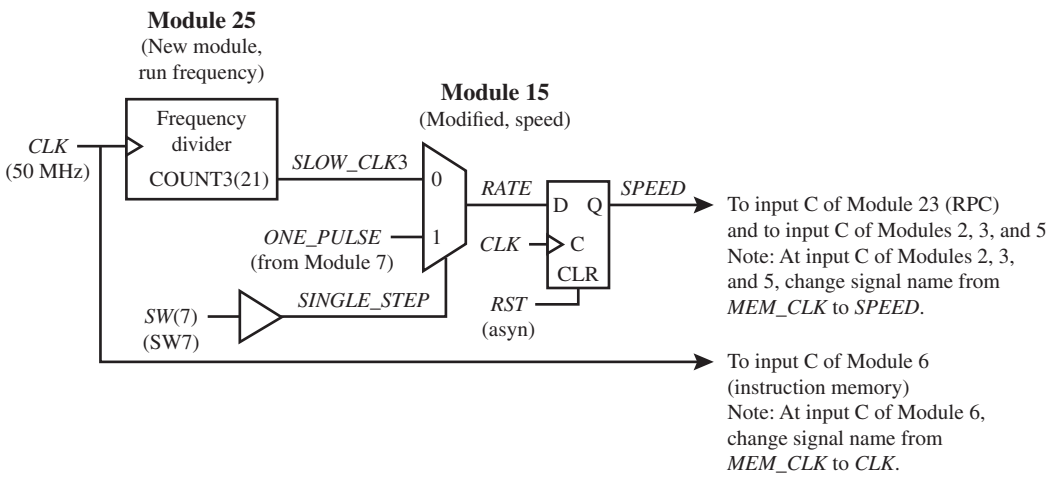


FIGURE E17.2 The run frequency and state machine modules for VBC1

Execution time for one instruction = Period for *SLOW_CLK3* = T_{SLOW_CLK3}

Execution time for a series of instructions = T_{SLOW_CLK3} × Number of instructions in the series

Note: If there is a loop in the series of instructions, the instructions inside the loop are executed multiple times.

To execute the JNZ instruction, control bit *M*6 must be added to the instruction decoder. This can be done by adding one more default option and one more condition to the case statement for the instruction decoder.

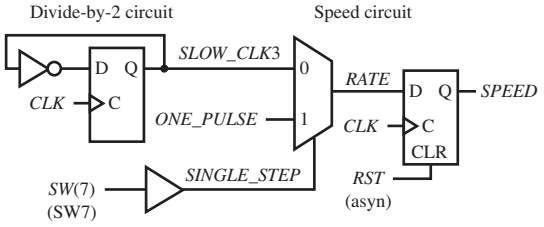
When the circuits in Figures E17.1 and E17.2 are added to your design of VBC1 and the VHDL code for the JNZ instruction is included in the instruction decoder, you can load the instruction memory and execute all the instructions for VBC1. You can single step through a program or run a program at the frequency of *SLOW_CLK*3. To single step through a program,

set slide switch SW7 = 1, then press and release push button BTN1 to fetch and execute each instruction. To run a program at the frequency of *SLOW_CLK3* (or frequency of *SPEED*), set slide switch SW7 to 0.

Recommended Pre-Lab:

1. Create a new project named Speed_Circuit_D2 shown in Figure E17.3, and write complete VHDL code for the speed circuit with a divide-by-2 circuit using a Boolean equation for the MUX, a Boolean equation for the buffer, a conditional signal assignment for the D flip-flop, and a conditional signal assignment for the divide-by-2 circuit. Make the input and output signal names the same as shown in Figure E17.3 except for *SW*(7). Change *SW*(7) to *SW7*. Figure E17.3 is the same as module 15 except for the divide-by-2 circuit that is added to it. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]

Speed circuit with a divide-by-2 circuit **FIGURE E17.3**



2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project and add Experiment 16 to the new project. Add module 23 (running programmable counter) using conditional signal assignments and Boolean equations. Include *M*6 as an internal output signal in module 14 (instruction decoder) and add m6 <= '0' to the list of default signals. This is necessary so that the PC is normally incremented to the next address except when a JNZ instruction occurs, which will be added later.
3. Add module 24 (proper address) using a conditional signal assignment. Be sure to route or supply the output signal to the inputs as shown in the schematic.
4. Add module 25 (run frequency) using a conditional signal assignment and a Boolean equation.
5. Modify module 15 (speed)—that is, use a Boolean equation for 2-to-1 MUX and a Boolean equation for the buffer.
6. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.

7. To verify that your design works for all the instructions except JNZ, use EASY1 to obtain the machine code for the bouncing lights program in Program E17.1. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E17.1 by initializing memory. Set SW7 = 1 to single step through the program by pressing BNT1 to execute each instruction, and then set SW7 = 0 to run the program at the speed of *SLOW_CLK*3.

PROGRAM E17.1
Bouncing lights program

```
;VBC1 Exp 17 Program 1
;bouncing lights program, 2 left then 2 right
loadi r1,1
addi r1,3
mov r0,r1
out r0
add r0,r0
out r0
sr0 r0,r0
sr0 r0,r0
out r0
in r0    ; input 1 via 4 slide switches
out r0
```

Do not continue until you get your design working for single stepping and running at the speed of *SLOW_CLK*3. If VBC1 does not execute the instructions and provide the same results as EASY1, then you either made a mistake in loading the machine code or you may have design errors in your VHDL code that you must find and fix.

8. Add the JNZ instruction to Module 14 (instruction decoder).
9. Complete the design cycle for your circuit by doing the following:
   a. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   b. Download the programming file into the FPGA on a BASYS 2 board or on a NEXYS 2 board.
10. Add the assembly language instruction JNZ R0,0 to the end of the bouncing lights program. To check the jump instruction, manually load the jump instruction to the end of Program E17.1 (see Appendix E, for manually loading memory, if you need help). Set SW7 = 0 to single step the program to verify that it works correctly when it reaches and executes the jump instruction. After you verify that Program E17.1 works correctly, set SW7 = 0 to run your VBC1 design at the speed of *SLOW_CLK*3, and observe the bouncing lights pattern.
11. To fully exercise the jump instruction for your VBC1 design, use EASY1 to obtain the machine code for the robot eye program in Program E17.2. Load the machine code for Program E17.2 by initializing memory (see Appendix E, for manually initializing memory, if you need help). Remove or comment out Program E17.1 in your VHDL code before regenerating the VHDL programming file. Set SW7 = 1 to single step through the program by pressing BNT1 to execute each instruction, and then set SW7 = 0 to run the program at the speed of *SLOW_CLK*3.

PROGRAM E17.2
Robot eye program

```
;VBC1 Exp 17 Program 2
;robot eye program with loops that execute
;the JNZ instruction several times!
x:   loadi r1,13  ;load immediate 16-3 (or 13) into r1
     loadi r0,1   ;load immediate 1 into r0
     out r0       ;output r0 to leds
y:   add r0,r0    ;add r0 to r0 (shift left)
     mov r0,r0    ;time delay
     mov r0,r0    ;time delay
     out r0       ;output r0 to leds
     addi r1,1    ;add immediate 1 to r1
     jnz r1,y     ;jump to y if r1 is not zero
     loadi r1,14  ;load immediate 16-2 (or 14) into r1
z:   sr0 r0,r0    ;shift right r0 (w0 fill) into r0
     out r0       ;output r0 to leds
     addi r1,1    ;add immediate 1 to r1
     mov r0,r0    ;time delay
     jnz r1,z     ;jump to z if r1 is not zero
     jnz r0,x     ;jump to x if r0 is not zero
```

Your VBC1 design should now work for single stepping and running Program E17.2 at the speed of *SLOW_CLK*3. If VBC1 does not execute the instructions and provide the same results as EASY1, then you either made a mistake in loading the machine code or you may have design errors in your VHDL code that you must find and fix.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is the execution of the robot eye program in Program E17.2 with your VBC1 design, both single stepping and running at the speed of *SLOW_CLK*3.
2. Include your calculation for the time it takes to execute one VBC1 instruction at the clock frequency of the signal *SLOW_CLK*3. Also include your calculation for how long it takes to execute all the instructions in the program in Program E17.2 down to, but not including, the instruction loadi r1,14 at the clock frequency of the signal *SLOW_CLK*3.
3. Include the complete VHDL code for your VBC1 design.
4. Include a printout of the Edit Constraints (Text) for your VBC1 design, which is generated by running Edit Constraints (Text).
5. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
6. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 17L: DESIGNING A LOADER FOR INSTRUCTION MEMORY FOR VBC1

by Scott M. Marshall and Richard S. Sandige

17L.1 Background

In this experiment, you will learn how to design a loader for instruction memory for VBC1 to create VBC1-L. When completing this experiment, focus on applying existing skills to new situations and learning good design techniques for interfacing an FPGA with other circuit components.

This experiment provides an extension to VBC1 (which was completed in Experiment 17) and is not required for VBC1 to function correctly.

The memory loader works by communicating with a USB communication circuit placed on a BASYS 2 board or on a NEXYS 2 board by Digilent Inc. The VBC1-L Memory Loader software program uses a library provided by Digilent Inc. to communicate with the FPGA via the USB circuit.

17L.2 Module Overview

The heart of the additional hardware designed for communicating with the USB circuit is a CFSM (complex finite state machine). CFSMs were previously discussed in Chapter 9. Four synchronizer circuits (sync circuits), each of which consists of two cascaded D flip-flops (not a single D flip-flop), serve as USB I/O synchronizers for the CFSM and form a new module, module L1. The CFSM, and a single loadable register, explained in Chapter 12, Section 12.4, form a new module, module L2. In addition, an OR gate is added as module L3. Figure E17L.1 shows the block diagrams for modules L1, L2, and L3.

Each sync circuit in module L1 is a synchronizer circuit that consists of two D flip-flops connected as shown in the Figure E17L.1A.

Things to note regarding Figures E17L.1 and E17L.1A:

- Figure E17L.1 shows block diagrams of modules L1, L2, and L3, added to VBC1.
- The external input signals for module L1 are *CLK*, *A_USB_ASTB*, *A_USB_DSTB*, and *A_USB_WRITE*, while the external output signal is *USB_WAIT_S*. *Note*: The notation *A_* is used for asynchronous external input signals coming into the FPGA—for example, *A_USB_ASTB*—and the notation *_S* is used for synchronous external output signals leaving the FPGA—for example, *USB_WAIT_S*. The remaining signals are internal signals.
- The external input signals for module L2 are *CLK* and *RST*, while signal *USB_DATA*(7:0) is of type inout, because it is used for bidirectional communication with the USB circuit. The remaining signals are internal signals.
- Module L3 has the external input signal *LOAD_MEM_IN* and internal signals *MEMLOAD_WE* and *LOAD_MEM*.
- Signals for communication with the USB chip have *USB* as a part of the name, while signals pertinent to loading the instruction memory have names that are prefixed with *MEMLOAD*.
- Adjustments to the VHDL entity and internal signal declarations to account for applicable new or modified input, output, and internal signals must be made. The external asynchronous input signals (names with *A_*), synchronous output signals (names ending with *_S*), and bidirectional *USB_DATA* bus are new I/O signals for the FPGA, while the change of *LOAD_MEM* to *LOAD_MEM_IN* for BTN2 is the only change to the existing I/O signals.
- Reset inputs to the the sync circuits are not required.

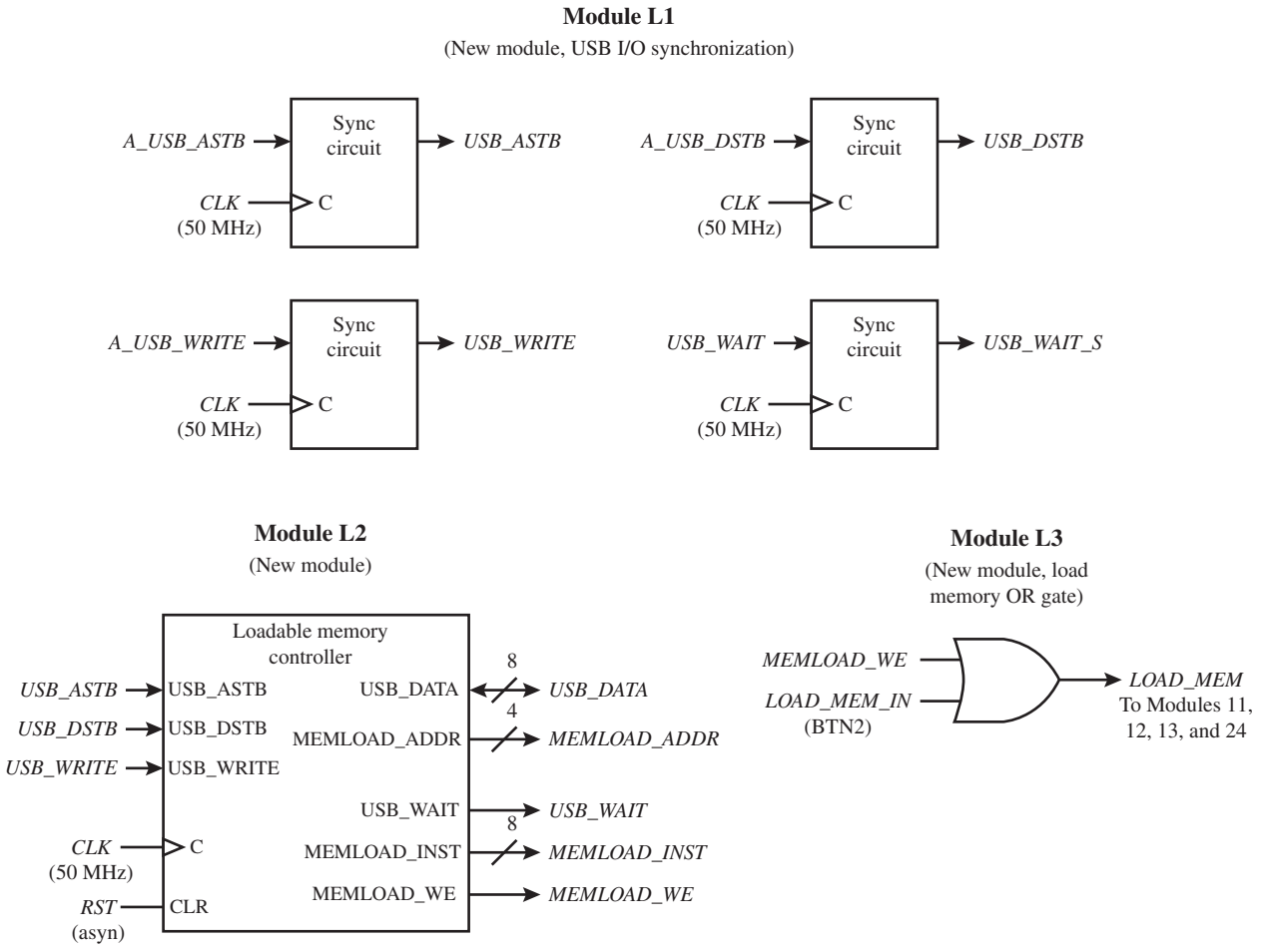


FIGURE E17L.1 Overview block diagrams for new modules L1, L2, and L3

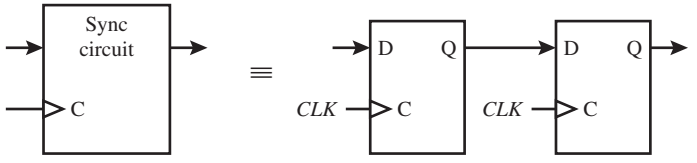


FIGURE E17L.1A Synchronizer circuit

17L.3 Memory Loader and CFSM Detail

Figure E17L.2 shows a detailed block diagram of the subcomponents of module L2.

Things to note regarding Figure E17L.2:

- Figure E17L.2 represents a block diagram of the subcomponents of module L2, including the loadable register.
- The loadable register subcomponent stores the current memory address the memory loader CFSM is acting upon, which is provided on the *MEMLOAD_ADDR* output signal.

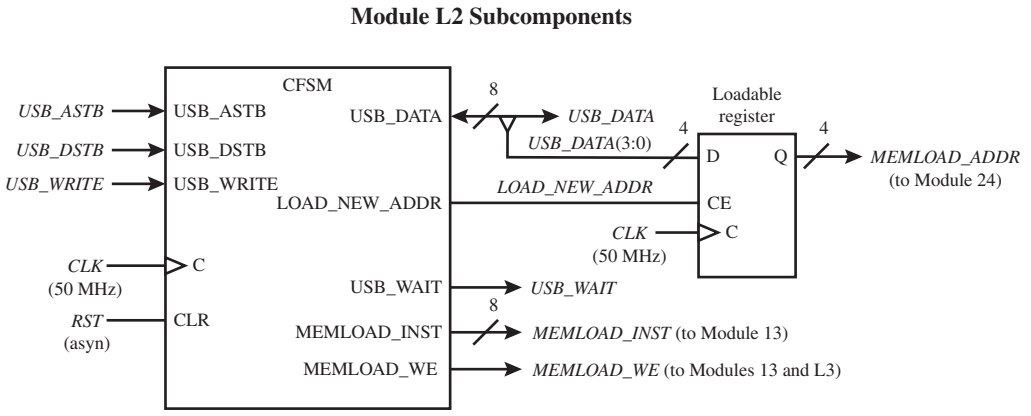


FIGURE E17L.2
Detailed block diagram for the subcomponents of module L2

- The signal *LOAD_NEW_ADDR* joins an output of the CFSM to the CE (clock-enable) input of the loadable register.
- The *MEMLOAD_ADDR* output signal is 4 bits wide and provides the address of the instruction to load, while the *MEMLOAD_INST* output signal is 8 bits wide and provides the instruction to load, and the *MEMLOAD_WE* output signal provides a write-enable signal for the instruction memory.

The CFSM in module L2 should be designed using the two-process PS/NS method. A state diagram detailing the proper function of the CFSM is provided in Figure E17L.3.

Things to note regarding Figure E17L.3:

- Figure E17L.3 is a state diagram for the CFSM in module L2, and uses the show all values convention, rather than the show where true convention, both explained in Chapter 9, Section 9.7.
- Where the values 0, 00000000, and ZZZZZZZZ are specified as the Moore outputs in the state diagram, use these as the default values that precede the case statement. Values or variables other than these should be specified for states within the case statement.
- While the *USB_DATA* signal is an output, it also appears as an input in the WRITE_INST state because it is a bidirectional signal. Data can both be written to and read from the bidirectional *USB_DATA* bus. Data can be read when *USB_DATA* is set to ZZZZZZZZ.
- The signals *USB_ASTB* and *USB_DSTB* are active low signals and are labeled accordingly in the state diagram. These two signals are strobe signals driven by clock-synchronization D flip-flops in Module L1, which are driven by the USB circuit. These strobe signals are pulsed to trigger the start of either address or instruction communication with the USB circuit.
- The signal *USB_WRITE* is active low, such that when the signal is low, data are written to the FPGA.
- The READ states are reached when the USB circuit (and thus the computer) is reading the current address or instruction from the CFSM and instruction memory, while the WRITE states are reached when the USB circuit (and thus the computer) is writing a new address or instruction to the CFSM and instruction memory.
- Situations where no state change occurs (such as when READY points to itself) should not be described using specific conditions in if or elsif clauses, but rather, should be described by else clauses. The else clauses act as catch-all states and will prevent the generation of any latches.
- The value of 00000000 for the signal *USB_DATA* in the READ_ INST state is used because reading the instruction memory on the computer is not supported in VBC1-L.

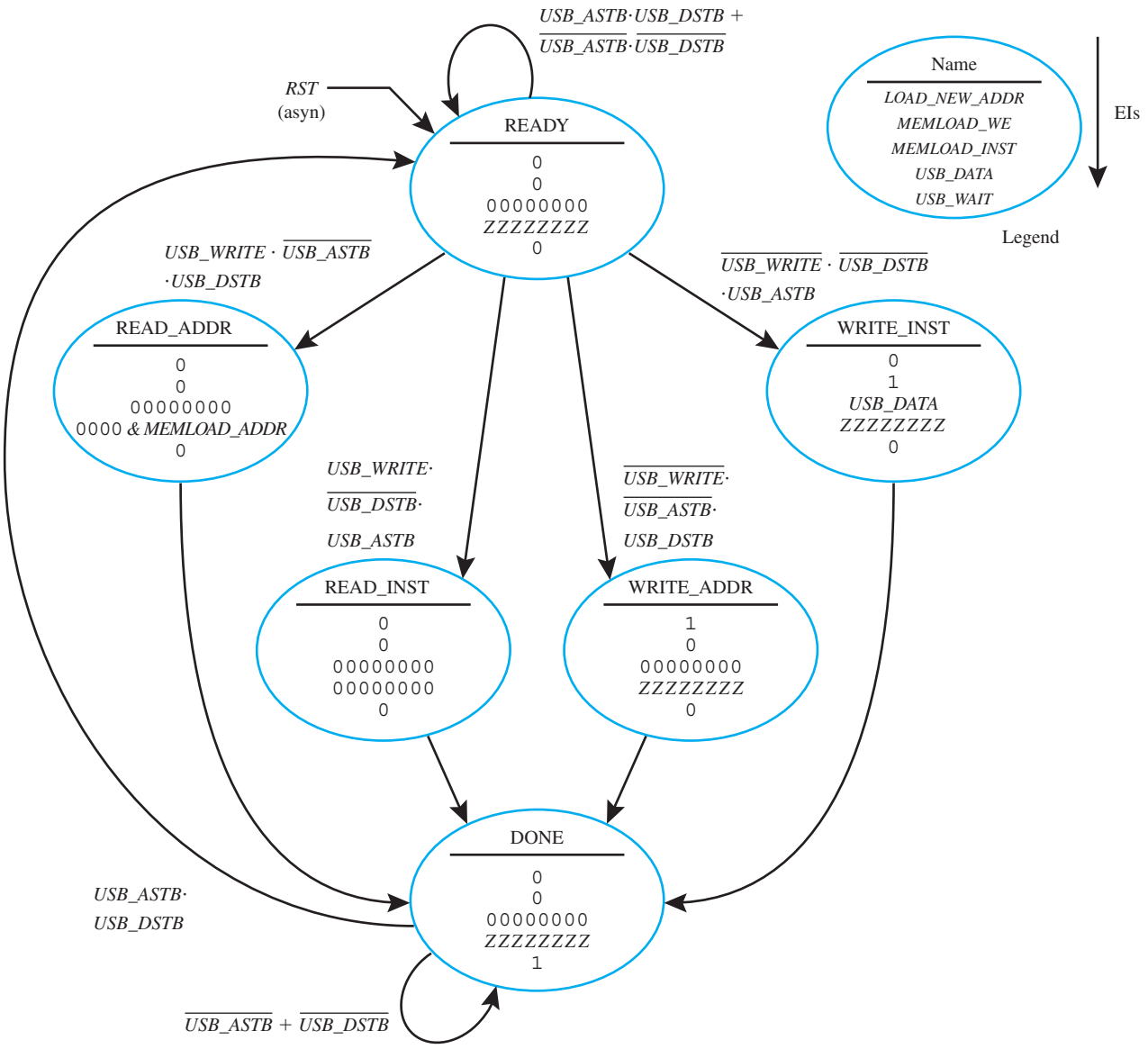


FIGURE E17L.3 State diagram for module L2 CFSM

- For more details on Digilent Inc.'s communication protocol, see the document "Digilent Parallel Interface Model Reference Manual" at http://www.digilentinc.com/Data/Products/ADEPT/DpimRef%20programmers%20manual.pdf.

It is important to use a default output value of ZZZZZZZZ (make Zs uppercase, not lowercase) for the *USB_DATA* signal to prevent bus contention or driver fight, as previously discussed in Chapter 7, Section 7.3. Setting the *USB_DATA* signal to ZZZZZZZZ places the FPGA pins in a high impedance state, which can be thought of as being disconnected. This allows the USB circuit to safely send data to the FPGA without hardware damage, the FPGA to read the data, and the *USB_DATA* signal to work as a bidirectional bus. For more information on data bus sharing and the high impedance state, see Chapter 7, Sections 7.2 and 7.3.

17L.4 Good Design Practices: Interfacing with Other Components

It is important to realize that a goal of this experiment is not only to review previously learned VHDL design concepts and to create an easy and fast way to load VBC1 instruction memory, but to also learn new techniques and best practices when interfacing a FPGA with other components.

Push-button and switch events are asynchronous, because a user can trigger a push button or switch independent of the clock signal *CLK*. The asynchronous behavior of other components, such as the USB circuit, may not be as apparent, however; they too are asynchronous because a *CLK* signal is not transferred between them and the FPGA.

Asynchronous signals can cause problems within the FPGA if their levels change when the *CLK* signal reaches the triggering (rising and/or falling) edge. When this occurs, the FPGA may not use the intended value of the asynchronous signal and can enter a metastable state.

To prevent problems, it is good design practice to synchronize asynchronous signals, such as inputs from other circuit components, as well as outputs of combinational logic, which can become asynchronous even if originally generated by synchronous logic due to propagation delays. It is considered a good design practice to synchronize a signal by feeding it through two cascaded D flip-flop as shown in Chapter 9, Section 9.10.

In the case of VBC1-L, the asynchronous FPGA input signals *A_USB_ASTB*, *A_USB_DSTB*, and *A_USB_WRITE* should all be synchronized, resulting in the signals *USB_ASTB*, *USB_DSTB*, and *USB_WRITE*. In addition, the asynchronous combinational logic output signal *USB_WAIT* from module L2 should be synchronized to form the FPGA output signal *USB_WAIT_S*. Module L1 serves this purpose.

Note that best design practices would also synchronize the *USB_DATA* signal. However, synchronizing *USB_DATA*, which is a bidirectional tri-stated bus, is an advanced technique that will not be covered here.

17L.5 Signal Routing Modifications

To allow the instruction memory to be loaded using either the push buttons and switches or the memory loader (module L2), the *LOAD_MEM* signal, which is high when memory is being loaded, must be changed to include the *MEMLOAD_WE* signal from module L2. To do so, the *LOAD_MEM* signal is renamed to *LOAD_MEM_IN* in both the entity and the .ucf (implementation constraints) file and must be connected to BTN2. Then, a new internal signal by the name of *LOAD_MEM* is created and is assigned as the OR combination of *LOAD_MEM_IN* and *MEMLOAD_WE*, forming the OR gate in module L3. All existing modules should continue to use the *LOAD_MEM* signal, not *LOAD_MEM_IN*. Now, when either BTN2 is pressed or when *MEMLOAD_WE* is high, memory will be loaded.

In order to accommodate the additional address, instruction, and write-enable signals from the memory loader, modules 13 (load memory) and 24 (proper address) must be modified. A single 8-bit-wide 2-to-1 MUX array along with a single 2-to-1 MUX must be added to module 13, along with the internal signals *LOAD_INST*(7:0) and *LOAD_WE*. A single 4-bit-wide 2-to-1 MUX array, along with the internal signal *LOAD_ADDR*(3:0), must be added to module 24. The modified versions of modules 13 and 24 are detailed in Figures E17L.4 and E17L.5, respectively.

Things to note regarding Figure E17L.4:

- The *LOAD_INST*(7:0) and *LOAD_WE* internal signals are added, along with an 8-bit 2-to-1 MUX array and a single 2-to-1 MUX.
- The signal *LOAD_MEM_IN* chooses between the memory loader and the manual push buttons and switches for loading memory. When *LOAD_MEM_IN* is high, *LOAD_MEM* is also high (from module L3), and memory is loaded manually.
- The signal *LOAD_MEM* indicates when memory is being loaded.

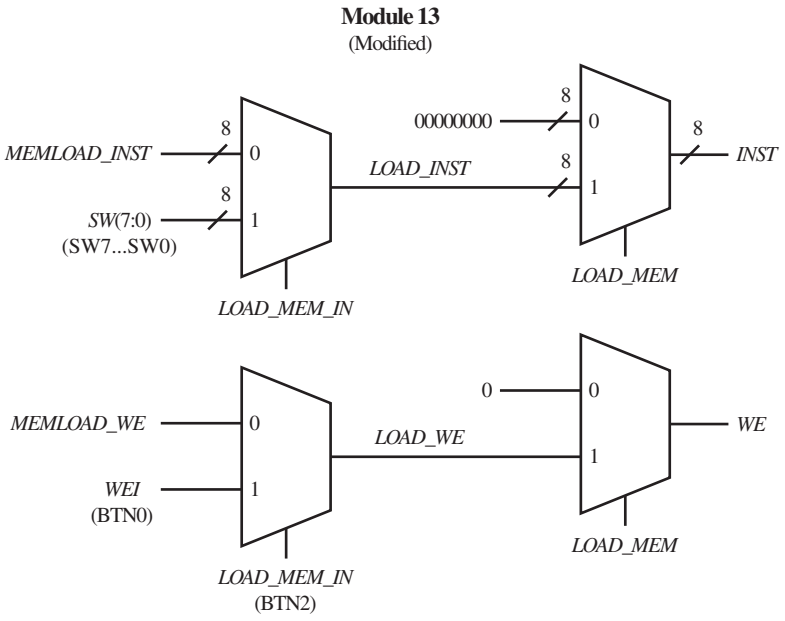


FIGURE E17L.4 Block diagram of modified module 13

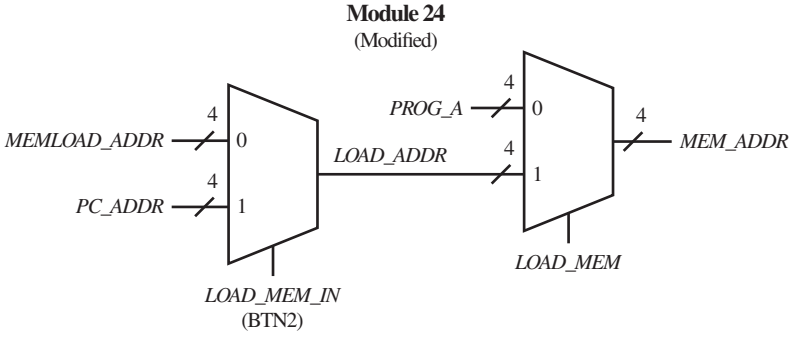


FIGURE E17L.5 Block diagram of modified module 24

Things to note regarding Figure E17L.5:

- The *LOAD_ADDR*(3:0) internal signal is added along with a 4-bit 2-to-1 MUX array.
- The signals *LOAD_MEM_IN* and *LOAD_MEM* act similarly to the modified version of module 13, detailed in Figure E17L.4.

Upon completion of the additions of modules L1, L2, and L3 and the modifications of modules 13 and 24, the design of VBC1-L is complete. The memory loader (module L2) can now control the instruction memory just as if memory is being loaded manually. Program E17.2 in Experiment 17 can be used to verify completion of VBC1-L. For information on loading programs with VBC1-L and on how to use the VBC1-L (VBC1-EL) memory loader program, see Appendix E.

Tasks:

1. Write the complete VHDL design for the modifications and additions described using the design styles you prefer via a flat design approach. Use documentation style M, introduced

in Chapter 12, Section 12.5.1, Listing 12.4, and be sure that the new modules are added in numerical order in the code section.
2. Start a new project with an entity that resembles that of Experiment 17, but make the needed modifications for modules L1, L2, and L3. Copy the existing VHDL design from Experiment 17 to use as a starting base for the needed additions and modifications.
3. Add the internal signals needed for modules 13, 24, L1, L2, and L3. When using documentation style M, remember that the signal *LOAD_MEM* is connected to modules L3, 11, 12, 13, and 24.
4. Modify modules 13 and 24 as described in Figures E17L.4 and E17L.5. Also notice in Figures E17L.4 and E17L.5 that signal *LOAD_MEM_IN* must be connected to push button BTN2, because the signal *LOAD_MEM* is used as an internal signal for this design.
5. Add module L1 using the block diagrams in Figure E17L.1.
6. Add module L2 using the CFSM state diagram in Figure E17L.3 and the block diagrams in Figures E17L.1 and E17L.2.
7. Add module L3 using the block diagram in Figure E17L.1.
8. Modify the .ucf (implementation constraints) file to account for the change of *LOAD_MEM* to *LOAD_MEM_IN* for BTN2 as well as the new I/O connections for modules L1 and L2. See Appendix C for the pin connections for all the signals that communicate with the USB circuit (these signals contain the letters USB).
9. For information on loading memory via the memory loader program, see Appendix E. With the design of VBC1-L complete, the VBC1-L (VBC1-EL) memory loader program can be used to automatically load memory. Verify that the completed design works by using the VBC1-L (VBC1-EL) memory loader program to load Program E17.2 in Experiment 17 and run the program successfully on a BASYS 2 board or on a NEXYS 2 board.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First, print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is demonstrating that Program E17.2 in Experiment 17 can be loaded using the VBC1-L memory loader software program and run successfully on a BASYS 2 board or on a NEXYS 2 board with your VBC1-L design.
2. Explain the purpose of the synchronizer circuits (sync circuits) in module L1.
3. Explain why the state machine in module L2 is considered complex.
4. Explain the purpose of the loadable register subcomponent in module L2.
5. Explain the purpose of module L3.
6. Include the complete VHDL code for your VBC1-L design.
7. Include a text copy of the .ucf (implementation constraints) file.
8. Write a short paragraph explaining the work you did for this experiment, along with a description of any problems you may have encountered and their solutions. Include helpful hints and suggestions of improvement for this experiment.
9. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 18 WRITING ASSEMBLY LANGUAGE PROGRAMS AND RUNNING THEM ON VBC1

In this experiment, you will practice writing assembly language programs and running them on your completed hardware version of VBC1. You will also practice writing assembly language programs for VBC1-E. This is summarized as follows:

1. Write the assembly language programs that are provided to gain practice in writing assembly language programs for VBC1. You must write the programs and simulate them using EASY1 to verify that they work properly. You must load the programs into VBC1 and verify that the programs perform correctly on your completed hardware version of VBC1.
2. Write the assembly language programs that are provided for VBC1-E and verify that they perform correctly using EASY1-E.
3. Be sure that your programs are well documented so that others can understand them. Copy each of your assembly language programs for tasks 1, 2, and so on, in a text file to keep them together. This will also allow you to easily copy and paste them into EASY1/EASY1-E to show that they work correctly in the lab. Store your programs under the titles shown in Text File E18.1.

```
;Program 1 for VBC1
;Program 2 for VBC1
;Program 3 for VBC1
;Program 4 for VBC1
;Program 5 for VBC1
;Program 6 for VBC1-E
;Program 7 for VBC1-E
;Program 8 for VBC1-E
;Program 9 for VBC1-E
;Program(s) for extra lab credit for VBC1-E
```

TEXT FILE E18.1 Store your programs under these titles

Recommended Pre-Lab:

1. Read Appendix D, Sections D.9 and D.10, for additional information about EASY1-E for this experiment. Also read Chapter 18, Section 18.8, More about Interrupts and Assembler Directives.
2. Write an assembly language program for VBC1-E that blinks the two least significant instruction pointer bits waiting on a hardware interrupt. An interrupt service routine counts the number of times the push button named Trigger Interrupt is pressed and delays the process of displaying the number at OP1 for 33 machine cycles. Be sure that your program does not run amuck. Provide a comment for each instruction that explains how the program works and not just what each instruction does.
3. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, assembled printout of the assembly language program with comments, a copy of your assembly language program with comments in a text file, and the like.

Tasks:

VBC1 Programs:

1. **Program 1**: Write an assembly language program for VBC1 that does the following:
   - The output LEDs follow the same sequence as a one-hot up counter, which is shown as follows:

```
LD3  LD2  LD1  LD0
Off  Off  Off  On
Off  Off  On   Off
Off  On   Off  Off
On   Off  Off  Off
Off  Off  Off  On   starts repeating
```

- When LD0 lights, the sequence rapidly moves to light LD1.
- When LD1 lights, it stays on for a long delay time (determined by a single loop counter) for approximately 17 machine cycles (or instruction cycles). A machine cycle is the time it takes to execute each assembly language instruction.
- After the delay for LD1, LD2 turns on, and the sequence rapidly moves to Light LD3.
- When LD3 lights, it stays on for the same delay time as LD1.
- After LD3 turns off, the sequence starts all over again.

Use EASY1 to write, assemble, and run your VBC1 program to verify that it performs correctly. Load your program into VBC1 and run your program. You should observe that your program performs the same way as it does with EASY1. See Figure D.3 in Appendix D for the instruction set and aliases for VBC1.

2. **Program 2**: Modify Program 1 so that the input switches can be used to select the delay time. After you verify proper operation of your program using EASY1, load your modified program into VBC1 and run your program. You should observe that your program performs the same way as it does with EASY1.
3. **Program 3**: Modify Program 1 to rapidly move (without any built-in delay time) through the sequence lighting the LEDs LD0, LD1, LD2, and LD3. Make the length of time that LD3 stays on be approximately 256 machine cycles. (Hint: $16 \times 16 = 256$.) After you verify proper operation of your program using EASY1, load your modified program into VBC1 and run your program. You should observe that your program performs the same way as it does with EASY1. (Hint: Use a loop counter embedded within a loop counter.)
4. **Program 4**: Modify Program 3 so that the input switches can be used to select the delay time. After you verify proper operation of your program using EASY1, load your modified program into VBC1 and run your program. You should observe that your program performs the same way as it does with EASY1.
5. **Program 5**: Write a program for a stoppable one-hot up counter for VBC1 that has a delay time of approximately 16 machine cycles for each of the output LEDs using a loop counter. When the input switches are all 0s, the counter will count through its sequence over and over. When the input switches are not all 0s, the counter will stop at the maximum count of 1000.

VBC1-E Programs:

6. **Program 6:** Write an assembly language program for VBC1-E that uses the INT and IRET instructions to do the following:

   - The output LEDs for Port 1 follow the same sequence as a one-hot down counter, which is shown as follows:

```
LD3   LD2   LD1   LD0
On    Off   Off   Off
Off   On    Off   Off
Off   Off   On    Off
Off   Off   Off   On
On    Off   Off   Off   starts repeating
```

   - Provide a delay time of approximately 16 machine cycles for each of the output LED, using a loop counter.

   Use EASY1-E to write, assemble, and run your VBC1-E program to verify that it performs correctly. See Table 18.10 in Chapter 18 for the instruction set for VBC1-E. Each alias for VBC1 provided in Figure D.3 in Appendix D for VBC1 is the same for VBC1-E.

7. **Program 7**: Modify Program 6 so that the input switches for ports 3 downto 0 can be used to independently select the delay times for LD3, LD2, LD1 and LD0, respectively, during program execution. Verify proper operation of your program using EASY1-E.

8. **Program 8**: Write a program for a modulo 16 binary up counter, and write a interrupt service routine that lights the LED sequence LD0, LD2, LD1, LD3, LD2, LD0, and all off. Make port 1 the output port for the modulo 16 counter, and make port 2 the output port for the LED sequence. Use a hardware interrupt to interrupt the modulo 16 counter and run the LED sequence. Verify proper operation of your programs using EASY1-E.
9. **Program 9**: Write a program that compares two 4-bit numbers at input ports 1 and 2, and if they are in ascending order, display them in ascending order at output ports 1 and 2. If the same two 4-bit numbers at input ports 1 and 2 are out of ascending order, then sort them and display them in ascending order at output ports 1 and 2.
10. **Program(s) for extra lab credit**:

    Example: You can write a program for VBC1-E that multiplies two 4-bit numbers supplied at the input ports 1 and 2 and provides the output at output ports 1 and 2. Your program must run correctly using EASY1-E.

    Extra credit will be given for writing other interesting programs for VBC1-E. Your programs must run correctly using EASY1-E.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working programs for VBC1 and get them signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working programs. Your final working programs for this experiment will be chosen at random.
2. Ask your instructor if you should include the copy of your programs in the text file in your report, or if you should e-mail the copy of your programs in your text file to him/her as an attachment.
3. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
4. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 19: DESIGNING AND TESTING VBC1-E (IN, OUT, AND UNCHANGED INSTRUCTIONS)

In this experiment, you will practice working with modules to provide the IN and OUT instructions for VBC1-E. Beginning with your final design of VBC1 (Experiment 17), you will modify the design to include the new IN and OUT instructions and the unchanged instructions (LOADI, ADDI, and JNZ) for VBC1-E.

Figure E19.1 shows a BASYS 2 board with the necessary peripheral modules (three LED modules, one button module, and one switch module) necessary to perform Experiments 19 through 25 in a stripped-down version of VBC1-E. A NEXYS 2 board can also be used to perform Experiments 19 through 25 in a stripped down version of VBC1-E. The cost of the peripheral modules for the BASYS 2 board or the NEXYS 2 board is about $55.

Figure E19.2 shows a NEXYS 2 board with the necessary peripheral modules (five LED modules, one button module, and three switch modules) and the FX2 MIB (module interface board) necessary to perform Experiments 19 through 25 in a full-blown version of VBC1-E. The cost of the FX2 MIB is about $20. The cost of the additional peripheral modules (two additional switch modules and two additional LED modules) is about $40.

This is the first in a series of projects to design and test VBC1-E. Experiment 19 consists of the annotated schematics shown in Figures E19.3 and E19.4.

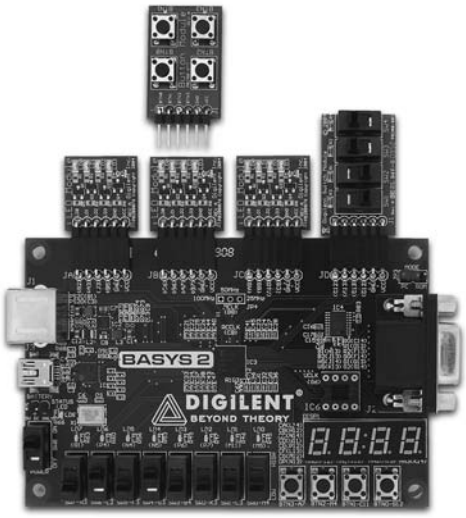


FIGURE E19.1 A BASYS 2 board with the necessary peripheral modules to design a stripped-down version of VBC1-E

Photo courtesy of Diligent, Inc.

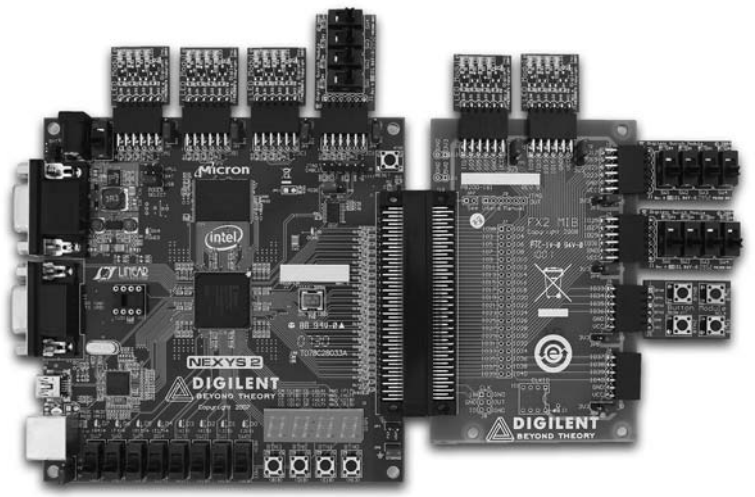


FIGURE E19.2 A NEXYS 2 board with the necessary peripheral modules to design a full-blown version of VBC1-E

Photo courtesy of Diligent, Inc.

Figure E19.3 shows the modified data path unit for VBC1-E with its new input circuit. Switch input circuits are not shown in the circuit diagram to simplify the drawing. The switches labeled *SW*(3:0) (SW3 . . . SW0), *SWP*1 (3:0) (JD-1, JD-2, JD-3, JD-4) (P stands for port), *SWP*2(3:0) (J3-1, J3-2, J3-3, J3-4), and *SWP*3(3:0) (J4-1, J4-2, J4-3 J4-4) are used for the full-blown version. The Pmod connector labeled JD shown in parentheses is used to obtain the pin assignments for the NEXYS 2 board. The Pmod connectors labels J3 and J4 shown in parentheses are used to obtain the pin assignments for the FX2 MIB (module interface board). The FX2 MIB is an add-on board on the right side of the NESYS 2 board that provides additional I/O (input/output) capability for the NEXYS 2 board.

The switches labeled *SW*(3:0) (SW3 . . . SW0) and *SWP*1 (3:0) (JD-1, JD-2, JD-3, JD-4) (P stands for port) are used for the BASYS 2 board or the NEXYS 2 board for the stripped-down version. The Pmod connector labeled JD shown in parentheses is used to obtain the pin assignments for the BASYS 2 board or the NEXYS 2 board. For a complete set of FPGA pin connections for the BASYS 2 board, the NEXYS 2 board, and the FX2 MIB see Appendix C (FPGA Pin Connections—Handy Reference).

Figure E19.4 shows the modified data path unit for VBC1-E with its new output circuit. LED output circuits are not shown in the circuit diagram to simplify the drawing. The LEDs labeled *LD*(3:0) (LD3 . . . LD0), *LDP*1(3:0) (JC-1, JC-2, JC-3 JC-4) (P stands for port), *LDP*2(3:0) (J1-1, J1-2, J1-3, J1-4), and *LDP*3(3:0) (J2-1, J2-2, J2-3, J2-4) are used for the full-blown version. The Pmod Connector labeled JC shown in parentheses is used to obtain the pin assignments for the NEXYS 2 board. The Pmod Connectors labels J1 and J2 shown in parentheses are used to obtain the pin assignments for the FX2 MIB (module interface board).

The LEDs labeled *LD*(3:0) (LD3 . . . LD0) and *LDP*1(3:0) (JC-1, JC-2, JC-3, JC-4) (P stands for port) are used for the BASYS 2 board or the NEXYS 2 board for the stripped-down version. The Pmod connector labeled JC shown in parentheses is used to obtain the pin assignments for the BASYS 2 board or the NEXYS 2 board.

To design the instruction decoder use Procedure ID. This procedure was first presented in Chapter 15, Section 15.4, and repeated in Chapter 19, Section 19.6. For this experiment, only include the control signals required for the instructions IN, OUT, LOADI, ADDI, and JNZ.

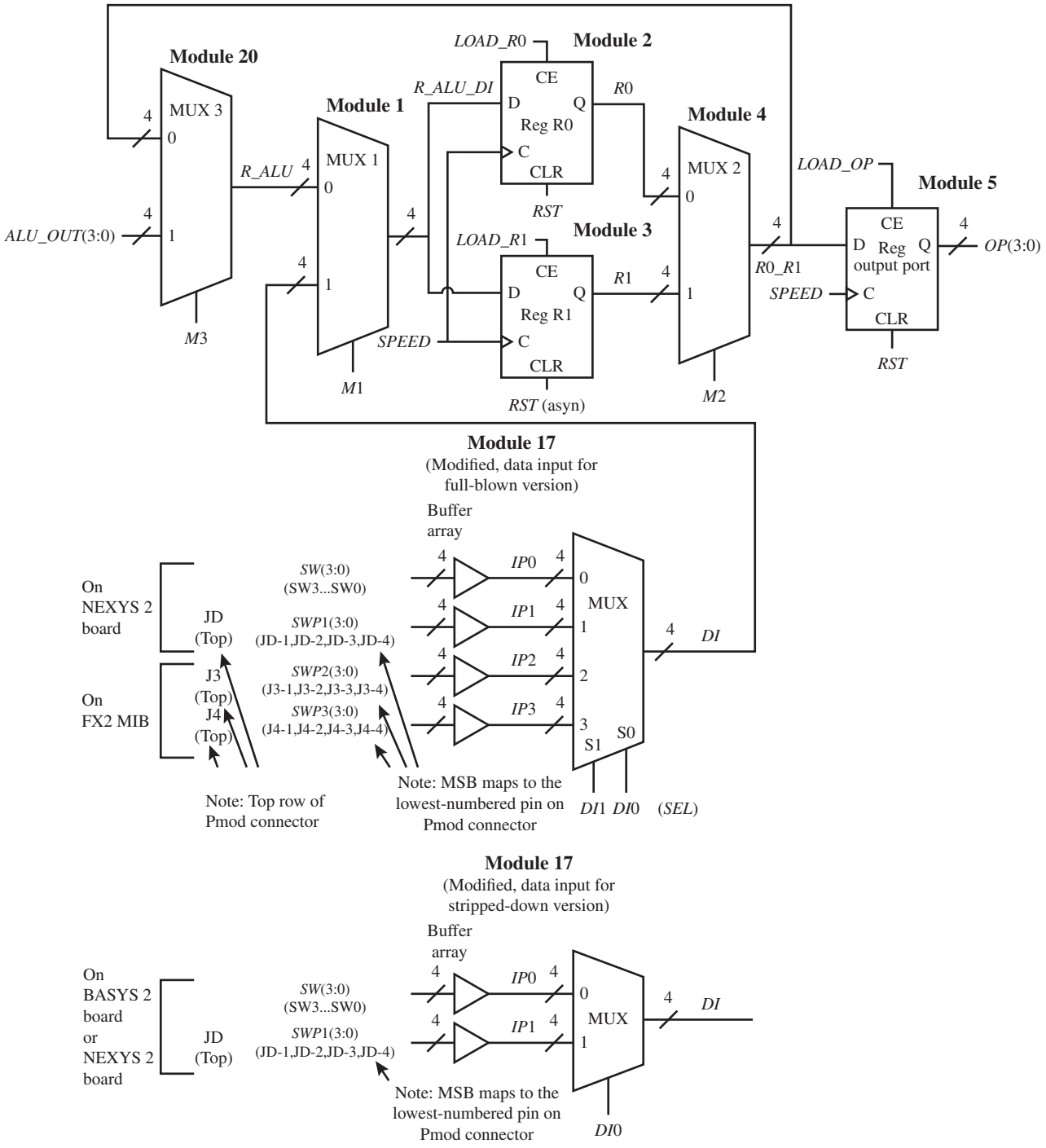


FIGURE E19.3 Modified data path unit for VBC1-E with its new input circuit

When the circuits in Figure E19.4 are incorporated in the design, you can load the instruction memory and execute the IN, OUT, LOADI, ADDI, and JNZ instructions to single step through a program or to run a program that only uses these instructions.

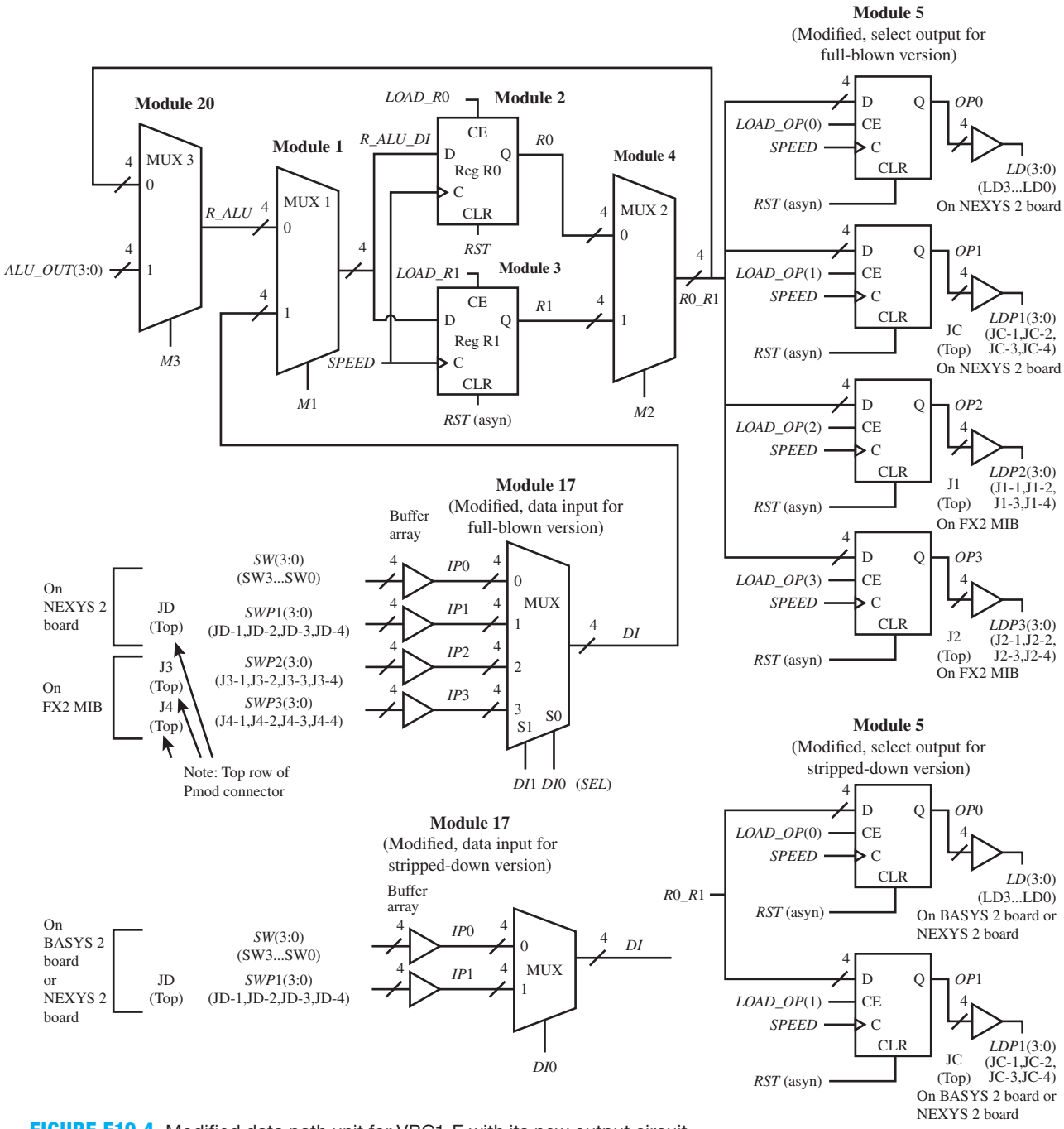


FIGURE E19.4 Modified data path unit for VBC1-E with its new output circuit

Recommended Pre-Lab:

1. Create a new project named Data_Input, and write complete VHDL code for the data input for the full-blown version using a conditional signal assignment for the MUX and Boolean equations for the buffers. Make the input and output signal names the same as those used by module 17 in Figure E19.4. Run a simulation to verify your design works. If your design

does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project and add your code for Experiment 17 or Experiment 17L to the new project. Modify module 17 (data input) as shown in Figure E19.3 or Figure E19.4. Use a conditional signal assignment for the MUX and Boolean equations for the buffers.
3. Modify module 5 (select output) as shown in Figure E19.4. Use a conditional signal assignment for each D flip-flop and Boolean equations for the buffers.
4. Use the instruction decoder truth tables for IN and OUT to obtain the Boolean equations for the control signals. Modify the instruction decoder (module 14), to handle only the instructions IN, OUT, LOADI, ADDI, and JNZ. Use the select signal ir (7 downto 5) in the case statement and select ir (3 downto 2) using an if statement for the IN and OUT instructions.
5. Modify the ALU (module 19) to handle only the instructions LOADI and ADDI.
6. If you initialized memory (module 6), remove the initialization instructions.
7. If you designed the full-blown version of VBC1-E, first plug the FX2 MIB into the NEXYS 2 board. Plug three switch modules into the Pmod connectors JD, J3, J4, and plug in three LED modules into the Pmod connectors JC, J1, and J2 as specified in Figure E19.4. If you designed the stripped-down version of VBC1-E, plug one switch module into Pmod connector JD and one LED module into Pmod connector JC as specified in Figure E19.4.
8. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.
9. Use the EASY1-E editor to enter the instructions shown for FB in Program E19.1, if you designed the full-blown version of VBC1-E. If you designed the stripped-down version of VBC1-E use Program E19.1 with the title "VBC1-E Exp 19 Program 1 Stripped." Use the EASY1-E simulator to single step through the program so that you understand what it does.

PROGRAM E19.1
Full-blown version and stripped-down version

```
;VBC1-E Exp 19 Program 1 FB
;Executing IN, OUT, and Unchanged instructions
;at all 4 input/output ports
start:
in r0,0       ;set input port 0 to 3
out r0,0
in r0,1       ;set input port 1 to 5
out r0,1
in r1,2       ;set input port 2 to 10
out r1,2
in r1,3       ;set input port 3 to 12
out r1,3
```

(Continued)

```
loadi r0,14
addi r0,1
jnz r0,start
```

```
;VBC1-E Exp 19 Program 1 Stripped
;Executing IN, OUT and Unchanged instructions
;only at input/output ports 0 and 1
start:
in r0,0        ;set input port 0 to 3
out r0,0
in r0,1        ;set input port 1 to 5
out r0,1
loadi r0,14
addi r0,1
jnz r0,start
```

10. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E19.1 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and also run program E19.1. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you either made a mistake assigning pins or you may have design errors in your VHDL code that you must find and fix.
11. If you used the memory loader circuitry from Experiment 17L to design VBC1-E, use the memory loader program to load Program E19.1 into memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and also run Program E19.1. VBC1-E should provide the same results as EASY1-E for each instruction.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is the execution of Program E19.1 with your VBC1-E design.
2. Include the complete VHDL code for your VBC1-E design.
3. Include a printout of the Edit Constraints (Text) for your VBC1-E design, which is generated by running Edit Constraints (Text).
4. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
5. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 20: DESIGNING AND TESTING VBC1-E (MOV AND DATA MEMORY INSTRUCTIONS)

In this experiment, you will practice working with modules to provide the STORE and FETCH instructions for VBC1-E. You will also provide the MOV instruction for VBC1-E.

This is the second in a series of projects to design and test VBC1-E. Experiment 20 consists of the annotated schematics shown in Figures E20.1 and E20.2.

Figure E20.1 shows a circuit for the 4 × 4 data memory.

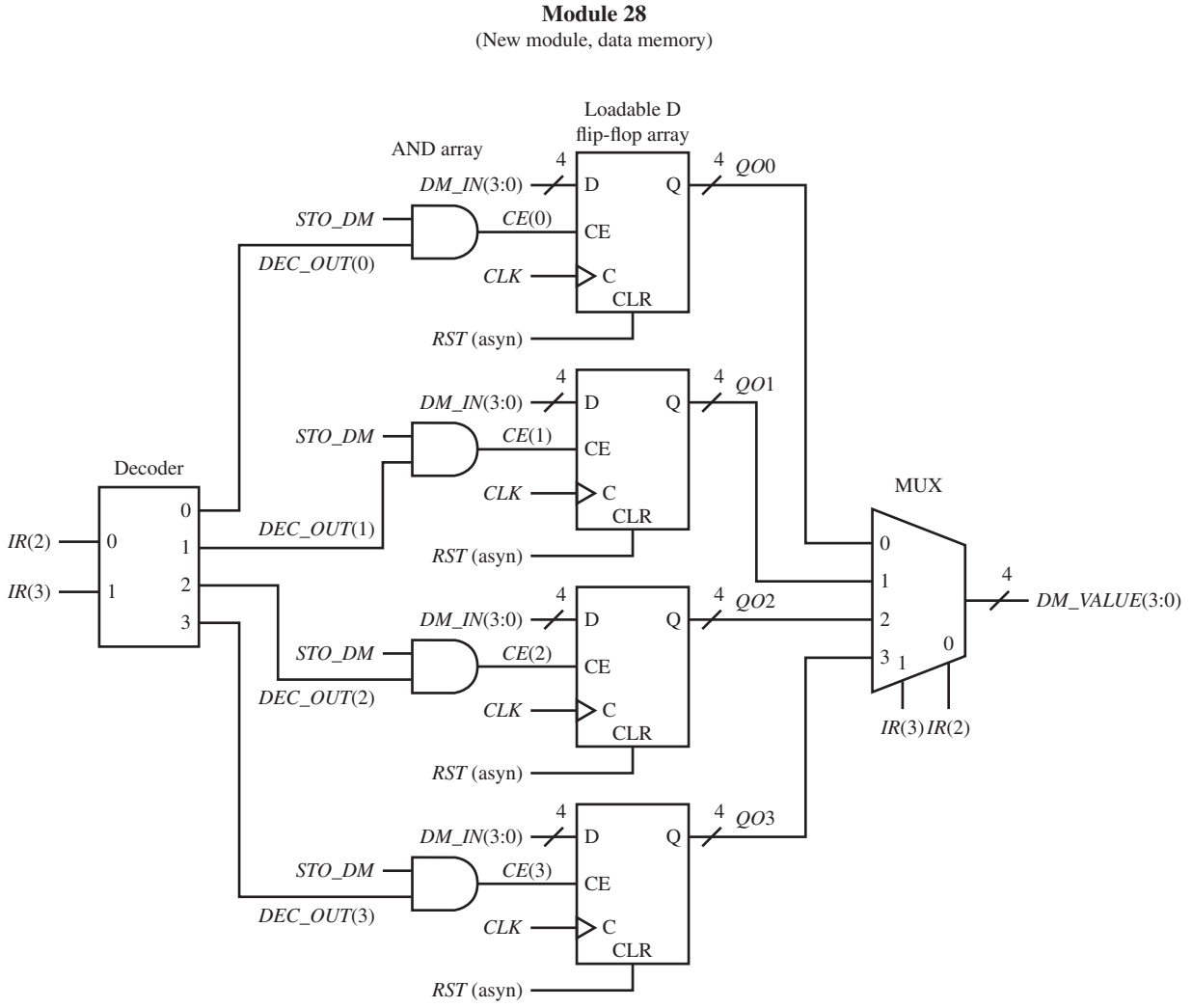


FIGURE E20.1 Circuit for the 4 × 4 data memory

Figure E20.2 shows the circuit that is used to perform the STORE and FETCH instructions. LED output circuits are not shown in the circuit diagram to simplify the drawing. The signal for the four LEDs is *DM*(3:0) (JB-1, JB-2, JB-3, JB-4). The labels in parentheses are used to obtain the pin assignments for the BASYS 2 board or the NEXYS 2 board.

Hint: For designing the Instruction Decoder: (1) use a process with a case statement to select each of the instructions by their OPCODE, (2) specify the default instruction decoder output values before the case statement, and (3) use Boolean equations for the control bits for each instruction within the case statement. Add the control signals required for the instructions STORE, FETCH, and MOV.

When the circuits in Figure E20.1 and E20.2 are incorporated in the design, you can load the instruction memory and execute the STORE, FETCH, and MOV instructions in addition to the instructions IN, OUT, LOADI, ADDI, and JNZ to single step through a program or to run a program that only uses these instructions.

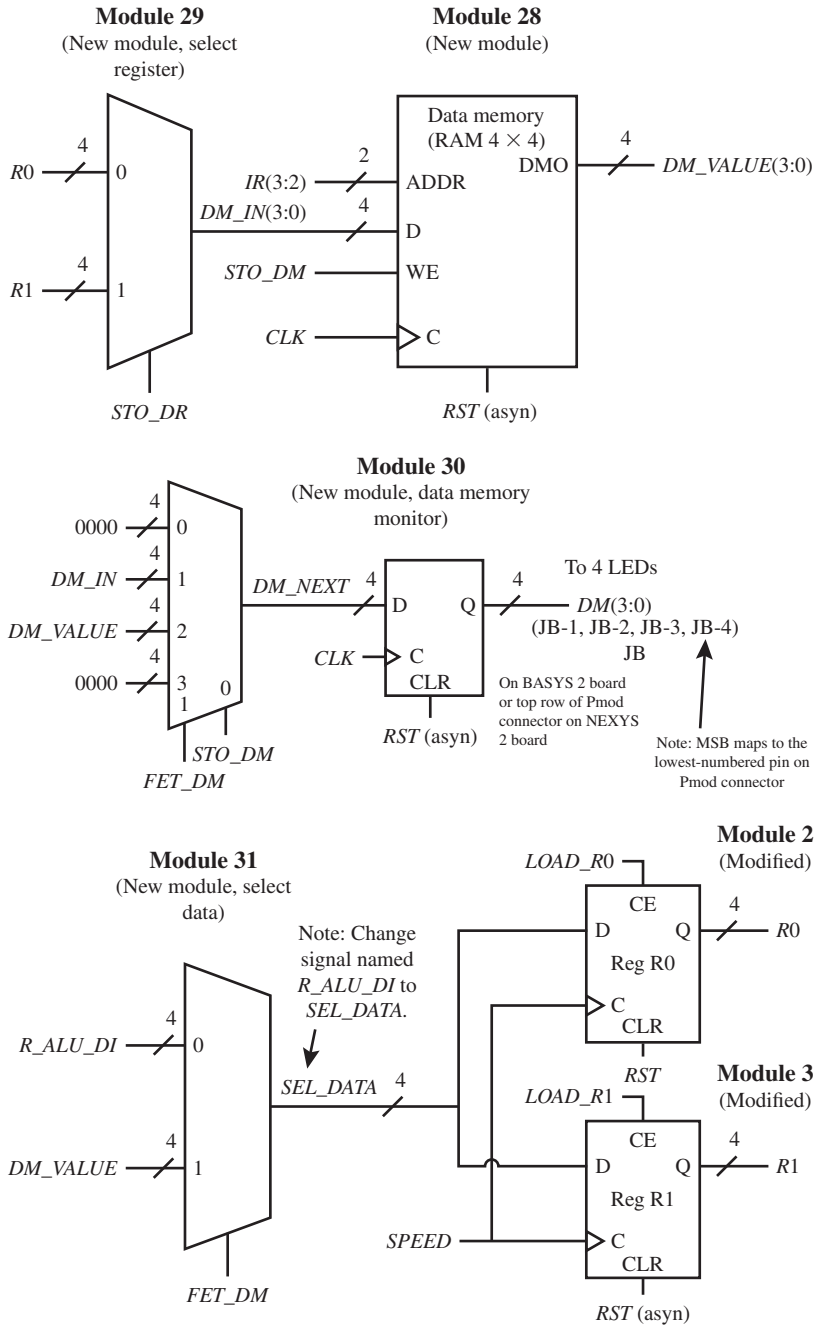


FIGURE E20.2 Modified data path unit for VBC1-E with it new output circuit

Recommended Pre-Lab:

1. Create a new project named Data_Memory, and write complete VHDL code for the data memory using a conditional signal assignment for each loadable D flip-flop in the array. Make the input and output signal names the same as those used by module 28 in Figure E20.1. You may use any design style you prefer for the decoder, the AND array, and the MUX for the data memory design. Run a simulation to verify your design works. If your

design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project and add your code for Experiment 19 to the new project. Add module 28 (data memory) as shown in Figure E20.1. Use a conditional signal assignment for each of the loadable D flip-flops.
3. Add module 29 (select register) as shown in Figure E20.2. Use a conditional signal assignment for the select register MUX.
4. Add module 30 (data memory monitor) as shown in Figure E20.2. Use conditional signal assignments for the data memory monitor.
5. Add module 31 (select data) as shown in Figure E20.2.
6. Modify modules 2 and 3 by changing the signal name *R_ALU_DI* to *SEL_DATA*.
7. Use the instruction decoder truth tables for STORE and FETCH to obtain the Boolean equations for the control signals. Modify the instruction decoder (module 14) to handle the instructions STORE, FETCH, and MOV.
8. VBC1-E requires a data memory. To test your design requires an additional peripheral module board—that is, an LED module. Plug the LED module into the Pmod connector JB on a BASYS 2 board or on a NEXYS 2 board as indicated in Figure E20.2 to observe the value of the data memory signal *DM*.
9. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.
10. Use the EASY1-E editor to enter the instructions shown for FB in Program E20.1, if you designed the full-blown version. If you designed the stripped-down version of VBC1-E use Program E20.1 with the title "VBC1-E Exp 20 Program 1 Stripped." Use the EASY1-E simulator to single step through the program so that you understand what it does.

PROGRAM E20.1
Full-blown version and stripped-down version

```
;VBC1-E Exp 20 Program 1 FB
;Executing store and fetch instructions
;at all 4 input/output ports
in r0,0       ;set input port 0 to 1
store r0,0
fetch r1,0
out r1,0
in r1,1       ;set input port 1 to 2
store r1,1
fetch r0,1
out r0,1
in r0,2       ;set input port 2 to 4
```

(Continued)

```
store r0,2
fetch r1,2
out r1,2
in r1,3       ;set input port 3 to 8
store r1,3
fetch r0,3
out r0,3

;VBC1-E Exp 20 Program 1 Stripped
;Executing store and fetch instructions
;only at input/output ports 0 and 1
back:
in r0,0       ;set input port 0 to 1
store r0,0
fetch r1,0
out r1,0
in r1,1       ;set input port 1 to 2
store r1,1
fetch r0,1
out r0,1
loadi r0,1
jnz r0,back
```

11. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E20.1 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and also run Program E20.1. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you either made a mistake assigning pins or you may have design errors in your VHDL code that you must find and fix.
12. If you used the memory loader circuitry from Experiment 17L to design VBC1-E, use the memory loader program to load Program E20.1 into memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and also run Program E20.1. VBC1-E should provide the same results as EASY1-E for each instruction.
13. Use the EASY1-E editor to enter the instructions shown for FB in Program E20.2, if you designed the full-blown version. If you designed the stripped-down version of VBC1-E, use Program E20.2 with the title "VBC1-E Exp 20 Program 2 Stripped." Use the EASY1-E simulator to single step through the program so that you understand what it does.

PROGRAM E20.2
Full-blown version and stripped-down version

```
;VBC1-E Exp 20 Program 2 FB
;Executing move instruction
;at all 4 input/output ports
IN R0,0     ; input 9 via 4 slide switches at port 0
OUT R0,0    ; output contents of R0 to LEDs at port 0
IN R0,1     ; input 15 via 4 slide switches at port 1
MOV R1,R0   ; move contents of R0 to R1
OUT R1,1    ; output contents of R1 to LEDs at port 1
IN R1,2     ; input 6 via 4 slide switches at port 2
OUT R1,2    ; output contents of R1 to LEDs at port 2
```

```
OUT R0,3    ; output contents of R0 to LEDs at port 3
IN R1,3     ; input 5 via 4 slide switches at port 3
MOV R0,R1   ; move contents of R1 to R0
OUT R1,3    ; output contents of R1 to LEDs at port 3

;VBC1-E Exp 20 Program 2 Stripped
;Executing move instruction
;only at input/output ports 0 and 1
back:
IN R0,0     ; input 9 via 4 slide switches at port 0
OUT R0,0    ; output contents of R0 to LEDs at port 0
IN R0,1     ; input 15 via 4 slide switches at port 1
MOV R1,R0   ; move contents of R0 to R1
OUT R1,1    ; output contents of R1 to LEDs at port 1
loadi r0,1
jnz r0,back
```

14. Comment out Program E20.1 in your VHDL design. Load the machine code from Program E20.2 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15 just as they are in the memory of EASY1-E. Single step and also run Program E20.2. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you may have design errors in your VHDL code that you must find and fix.
15. If you used the memory loader circuitry from Experiment 17L to design VBC1-E, use the memory loader program to load Program E20.2 into memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and also run Program E20.2. VBC1-E should provide the same results as EASY1-E for each instruction.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working designs. Your final working designs are the execution of Programs E20.1 and E20.2 with your VBC1-E design.
2. Include the complete VHDL code for your VBC1-E design.
3. Include a printout of the Edit Constraints (Text) for your VBC1-E design, which is generated by running Edit Constraints (Text).
4. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
5. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 21: DESIGNING AND TESTING VBC1-E (ALMOST ALL INSTRUCTIONS)

In this experiment, you will practice working with modules to provide all the arithmetic, logic, shift, rotate, and jump instructions—that is, almost all instructions for VBC1-E except the interrupt and interrupt return instructions.

This is the third in a series of projects to design and test VBC1-E. Experiment 21 consists of the annotated schematics shown in Figures E21.1 and E21.2.

Figure E21.1 shows the data path circuit for VBC1-E.

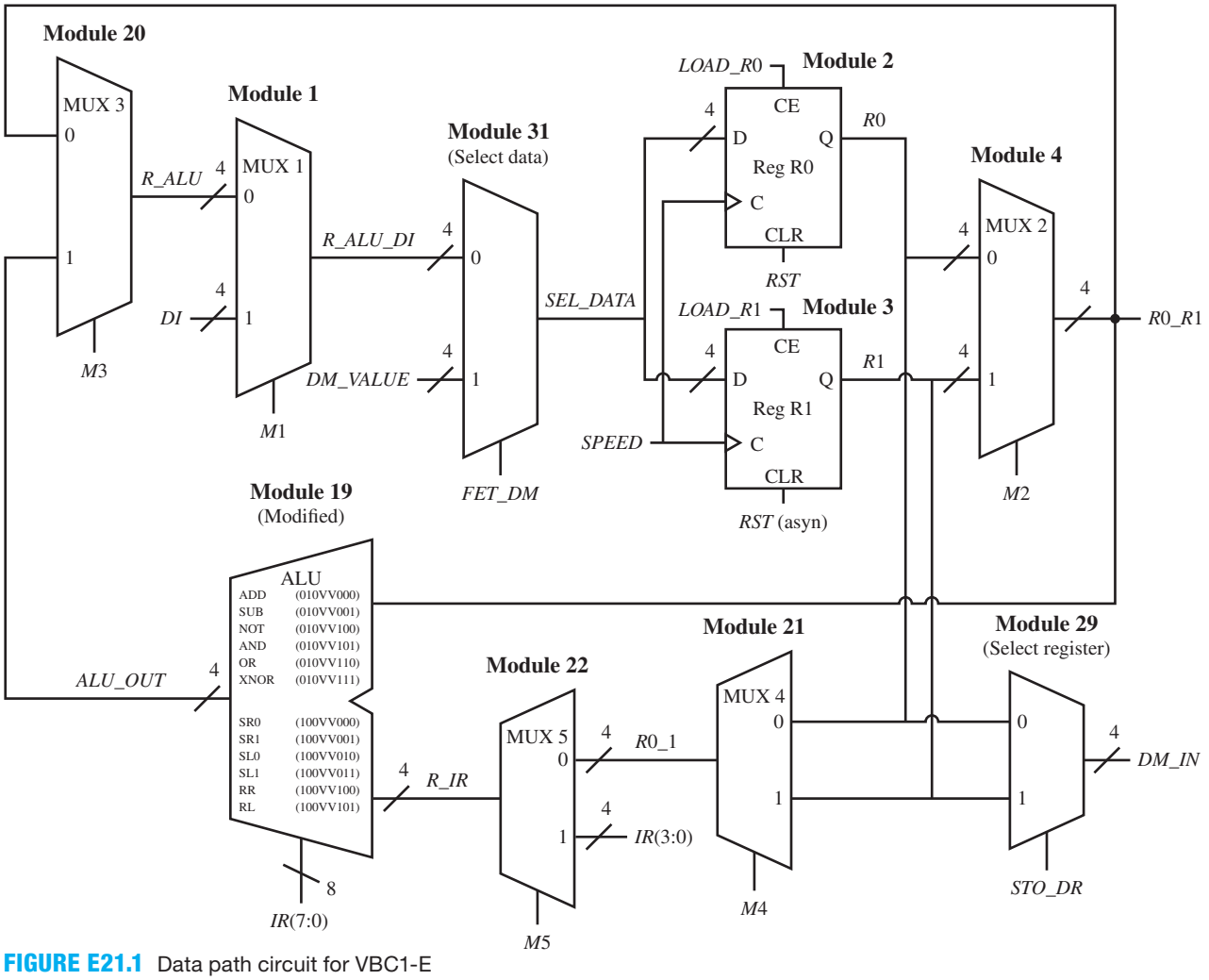


FIGURE E21.1 Data path circuit for VBC1-E

Figure E21.2 shows the circuits for the JMP and JMPR instructions for VBC1-E.

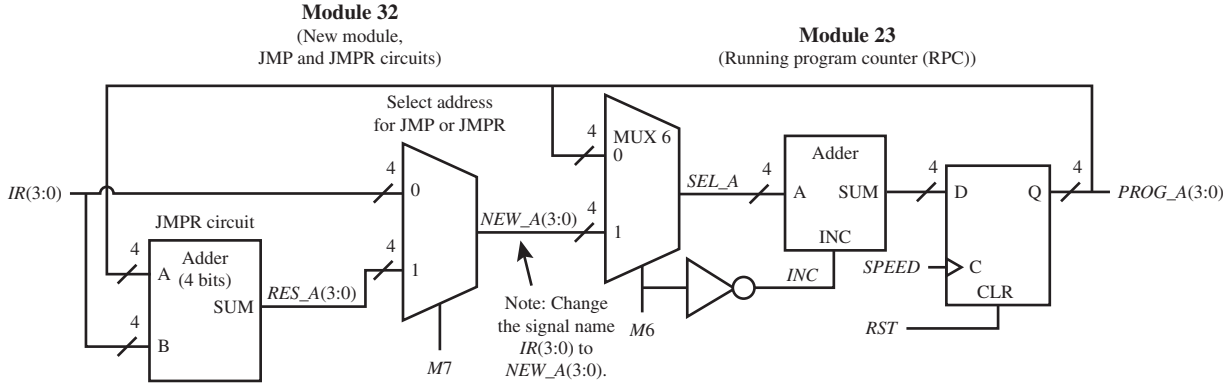


FIGURE E21.2 Circuits for the JMPR and JMP instructions for VBC1-E

When the circuits in Figures E21.1 and E21.2 are incorporated in the design, you can load the instruction memory and execute all the arithmetic, logic, shift, and rotate instructions and also the JMP, JMPR, and HALT instructions for VBC1-E.

Recommended Pre-Lab:

1. Create a new project named ALU, and write complete VHDL code for the ALU using a process with a case statement. Add the lines of code that are required to perform the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR. Make the input and output signal names the same as those used by module 19 in Figure E21.1. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for ecample, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project and add your code for Experiment 20 to the new project.
3. Add the lines of VHDL code to module 19 (ALU) that are required to perform the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR.
4. In Figure E21.1 for the ALU, make the operand signal R0_R1 the destination register (DR) and operand signal R_IR the source register (SR) for all of the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR for VBC1-E. With these operand assignments, obtain the Boolean equations for the control signals for the arithmetic and logic instructions ADD, SUB, NOT, AND, OR, and XNOR. Modify the instruction decoder (module 14) to handle the arithmetic and logic instructions.
5. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.
6. Use the EASY1-E editor to enter the instructions shown for FB in Program E21.1, if you designed the full-blown version of VBC1-E. If you designed the stripped-down version of VBC1-E use Program E21.1 with the title "VBC1-E Exp 21 Program 1 Stripped." Use the EASY1-E simulator to single step through the program so that you understand what it does.

PROGRAM E21.1
Full-blown version and stripped-down version

```
;VBC1-E Exp 21 Program 1 FB
;Executing arithmetic and logic instructions
;at all 4 input/output ports
in r0,0 ;set input port 0 to 1
out r0,0
loadi r1,2
out r1,1
add r1,r0
out r1,2
```

(Continued)

```
sub r1,r0
out r1,3
not r1,r1
out r1,0
and r1,r0
out r1,1
loadi r1,2
xnor r0,r1
or r0,r1
out r0,3      ;roll over

;VBC1-E Exp 21 Program 1 Stripped
;Executing arithmetic and logic instructions
;only at input/output ports 0 and 1
begin:
in r0,0       ;set input port 0 to 1
out r0,0
loadi r1,2
out r1,1
add r1,r0
sub r1,r0
not r1,r1
out r1,0
and r1,r0
out r1,1
loadi r1,2
xnor r0,r1
or r0,r1
out r0, 1
```

7. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E21.1 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and also run Program E21.1. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you either made a mistake assigning pins or you may have design errors in your VHDL code that you must find and fix.
8. Add the lines of VHDL code to module 19 (ALU) that are required to perform the shift and rotate instructions SR0, SR1, SL0, SL1, RR, and RL.
9. In Figure E21.1 for the ALU, make the operand signal R0_R1 the destination register (DR) and operand signal R_IR the source register (SR) for all of the shift and rotate instructions SR0, SR1, SL0, SL1, RR, and RL for VBC1-E. With these operand assignments, obtain the Boolean equations for the control signals for the shift and rotate instructions SR0, SR1, SL0, SL1, RR, and RL. Modify the instruction decoder (module 14) to handle the shift and rotate instructions.
10. Do the following: Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File. Download the programming file.
11. Use the EASY1-E editor to enter the instructions shown for FB in Program E21.2, if you designed the full-blown version of VBC1-E. If you designed the stripped-down version of

VBC1-E use Program E.21.2 with the title "VBC1-E Exp 21 Program 2 Stripped." Use the EASY1-E simulator to single step through the program so that you understand what it does.

PROGRAM E21.2
Full-blown version and stripped-down version

```
;VBC1-E Exp 21 Program 2 FB
;Executing shift and rotate instructions
;at all 4 input/output ports
start: in r0,0    ;set input port 0 to 1
     out r0,0
     sl0 r1,r0
     out r1,1
     sl1 r1,r0
     out r1,2
     sr1 r1,r0
     out r1,3
     sr0 r1,r1
     out r1,0
     loadi r1,14
     rl r0,r1
     out r0,2
     rr r0,r0
     out r0,3
     jnz r0,start
```

```
;VBC1-E Exp 21 Program 2 Stripped
;Executing shift and rotate instructions
;only at input/output ports 0 and 1
start: in r0,0    ;set input port 0 to 1
       out r0,0
       sl0 r1,r0
       out r1,1
       sl1 r1,r0
       sr1 r1,r0
       sr0 r1,r1
       out r1,0
       loadi r1,14
       rl r0,r1
       rr r1,r1
       rr r0,r1
       out r0,1
       jnz r0,start
```

12. Comment out Program E21.1 in your VHDL design. Load the machine code for Program E21.2 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and run Program E21.2. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you may have design errors in your VHDL code that you must find and fix.
13. If you used the memory loader circuitry from Experiment 17L to design VBC1-E, use the memory loader program to load Program E21.2 into memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0

through 15, just as they are in the memory of EASY1-E. Single step and run Program E21.2. VBC1-E should provide the same results as EASY1-E for each instruction.

14. Add module 32 (JMP and JMPR circuits) as shown in Figure E21.2. Use an arithmetic expression for the JMPR circuit and a conditional signal assignment for the select address for JMP or JMPR.
15. Use the instruction decoder truth tables for the JMP and JMPR instructions to obtain the Boolean equations for the control signals. Modify the instruction decoder (module 14) to handle the JMP and JMPR instructions.
16. Modify module 23 (RPC) in Figure E21.2 by changing the signal name *IR*(3 **downto** 0) to *NEW_A*(3 **downto** 0).
17. Do the following: Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File. Download the programming file.
18. Use the EASY1-E editor to enter the instructions shown in Program E21.3. This program works on the full-blown version and also on the stripped-down version of VBC1-E. Use the EASY1-E simulator to single step through the program so that you understand what it does.

PROGRAM E21.3
Executing jump instructions

```
;VBC1-E Exp 21 Program 3
;Executing jump instructions
;no inputs required, outputs at ports 0 and 1
         loadi r0,1
repeat1: out r0,0
         sl0 r0,r0
         jnz r0,repeat1
         xnor r0,r0
         out r0,1
         loadi r1,4
         jmpr 1
repeat2: out r1,0
         sr0 r1,r1
         jnz r1,repeat2
         xnor r1,r1
         not r1,r1
         out r1,1
         jmp 0
```

19. Comment out Program E21.2 in your VHDL design. Load the machine code for Program E21.3 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and run Program E21.3. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you may have design errors in your VHDL code that you must find and fix.
20. If you used the memory loader circuitry from Experiment 17L to design VBC1-E, use the memory loader program to load Program E21.3 into memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Single step and run Program E21.3. VBC1-E should provide the same results as EASY1-E for each instruction.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working designs. Your final working design is the execution of Program E21.3 with your VBC1-E design.
2. Include the complete VHDL code for your VBC1-E design.
3. Include a printout of the Edit Constraints (Text) for your VBC1-E design, which is generated by running Edit Constraints (Text).
4. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
5. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 22: DESIGNING AND TESTING VBC1-E (MODIFIED MANUAL LOADING)

In this experiment, you will practice working with a module to modify manual loading. This provides fewer distractions from flashing outputs as a program is manually loaded into instruction memory or manually stepped through instruction memory to observe its contents.

This is the fourth in a series of projects to design and test VBC1-E. Experiment 22 consists of the annotated schematic shown in Figure E22.1.

Figure E22.1 shows a circuit that prevents instruction execution during manual loading for VBC1-E.

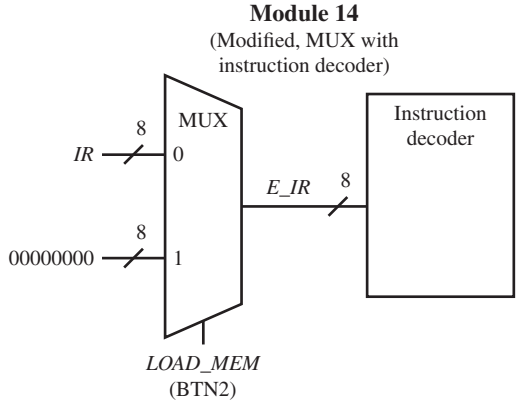


FIGURE E22.1
Circuit that prevents instruction execution during manual loading for VBC1-E

When the circuit in Figure E22.1 is incorporated in the design, instruction memory can be loaded with all the instructions except INT and IRET, which have not been implemented yet. When BTN2 is pressed to allow manual loading or when BTN2 is pressed to observe the contents at an address in instruction memory, you will not observe the execution of an OUT instruction during the loading or observing process. This will provide fewer distractions when you manually load or view the contents of instruction memory due to output ports that may be lighted.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking

each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.

2. Start a new project and add your code for Experiment 21 to the new project.
3. Modify module 14 so that eight 0s are supplied to the instruction decoder when BTN2 is pressed as shown in Figure E22.1. Use a conditional signal assignment for the MUX in Figure E22.1. Be sure to define E_IR in module 14 as an internal signal between **architecture** and the first **begin**.
4. In the instruction decoder process change IR to E_IR. This is most easily done by placing the cursor just before IR in the process and click Edit then Replace on the Project Navigator menu bar. When the Replace window appears, type IR after "Find what," and type E_IR after "Replace with." *Do not* click on "Replace All." Click on "Find Next" and then click "Replace" repeatedly, until you replace every IR just in the instruction decoder process of module 14 and then stop.
5. Complete the design cycle for your circuit by doing the following:
   a. Generate a programming file.
   b. Check to see if your design needs to be fixed based on any errors or warnings. If so, make the fixes, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.
6. Use the EASY1-E editor to enter the instructions shown for FB in Program E22.1, if you designed the full-blown version of VBC1-E. If you designed the stripped-down version of VBC1-E use Program E22.1 with the title "VBC1-E Exp 22 Program 1 Stripped." Use the EASY1-E simulator to single step through the program or run the program so that you understand what it does.

PROGRAM E22.1
Full-blown version and stripped-down version

```
;VBC1-E Exp 22 Program 1 FB
;Turn all port lights on and off
loadi r0,5
out r0,0
out r0,1
out r0,2
out r0,3
loadi r0,10
out r0,0
out r0,1
out r0,2
out r0,3
loadi r0,0
out r0,0
out r0,1
out r0,2
out r0,3
jmp 0

;VBC1-E Exp 22 Program 1 Stripped
;only turns port lights 0 and 1 on and off
loadi r0,5
out r0,0
out r0,1
loadi r0,10
```

```
out r0,0
out r0,1
loadi r0,0
out r0,0
out r0,1
jmp 0
```

7. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E22.1 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 15, just as they are in the memory of EASY1-E. Run Program E22.1. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you either made a mistake assigning pins or you may have design errors in your VHDL code that you must find and fix.
8. Perform the following test on VBC1-E. Set SW7 to 1 to single step then reset VBC1-E by pressing and releasing BTN3. Hold down BTN2 and press BTN1 repeatedly to observe the contents at each address in instruction memory. Observe that the output ports do not light as they do when you single step or run the program. This is the correct operation of VBC1-E due to the modification of module 14. If the output ports light, then you may have made a mistake with your VHDL code that needs to be fixed so the output ports do not light. When you get your program working correctly, change the 8 bits (eight 0s) provided at the 1 input to the MUX in module 14 to the signal *IR*. This will cause VBC1-E to execute instructions when you are manually loading a program or manually observing a program in instruction memory.
9. Complete the design cycle for your circuit by doing the following:
   a. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   b. Download the programming file into the FPGA on a BASYS 2 or a NEXYS 2 board.
10. Perform the following test on VBC1-E. Set SW7 to 1 to single step then reset VBC1-E by pressing and releasing BTN3. Hold down BTN2 and press BTN1 repeatedly to observe the contents at each address in instruction memory. The output ports should light as they do when you single step or run the program, which can be distracting, if you elect to manually load a program. After you finish this test, change the signal *IR* at the 1 input of the MUX in module 14 back to 8 bits (eight 0s) to agree with Figure E22.1.
11. Complete the design cycle for your circuit by doing the following:
   a. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   b. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.
12. Set SW7 to 1 to single step then reset VBC1-E by pressing and releasing BTN3. Hold down BTN2 and press BTN1 repeatedly to observe the contents at each address in instruction memory. Observe that the output ports do not light.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working

design is the execution of Program E22.1 with your VBC1-E design. You also need to show that the output ports do not light as you manually load a program or as you hold down BTN2 and press BTN1 repeatedly to observe the contents of instruction memory.
2. Include the complete VHDL code for your VBC1-E design.
3. Include a printout of the Edit Constraints (Text) for your VBC1-E design, which is generated by running Edit Constraints (Text).
4. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
5. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 23: DESIGNING AND TESTING VBC1-E (ADD EXTENDED INSTRUCTION MEMORY)

In this experiment, you will practice working with modules to add extended instruction memory. Adding extended instruction memory (module 6) is easy, but this also requires modifying the 4-to-1 MUX array circuit for the multiplexed display system (module 10), the running program counter circuit (module 23), the proper address circuit (module 24), the loading program counter circuit (module 8), and the JMPR circuit (module 32).

This is the fifth in a series of projects to design and test VBC1-E. Experiment 23 consists of the annotated schematics shown in Figures E23.1 and E23.2.

Figure E23.1 shows the modified circuits for the total instruction memory and the 4-to-1 MUX array for the multiplexed display system for VBC1-E.

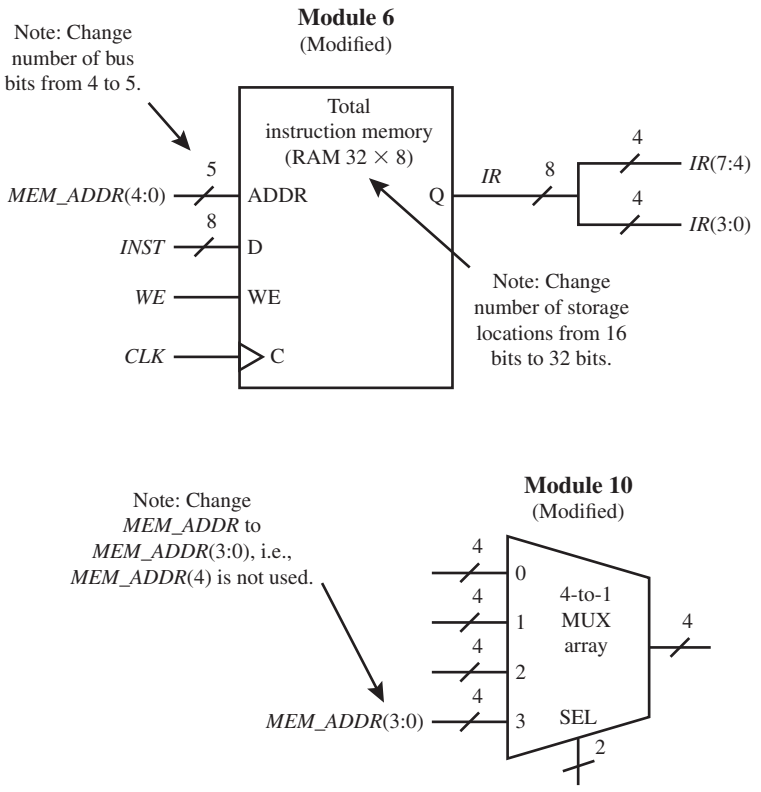


FIGURE E23.1 Modified circuits for the total instruction memory and the 4-to-1 MUX array for the multiplexed display system for VBC1-E

Figure E23.2 shows the modified circuits for the running program counter, the proper address, the loading program counter, and the JMPR circuit.

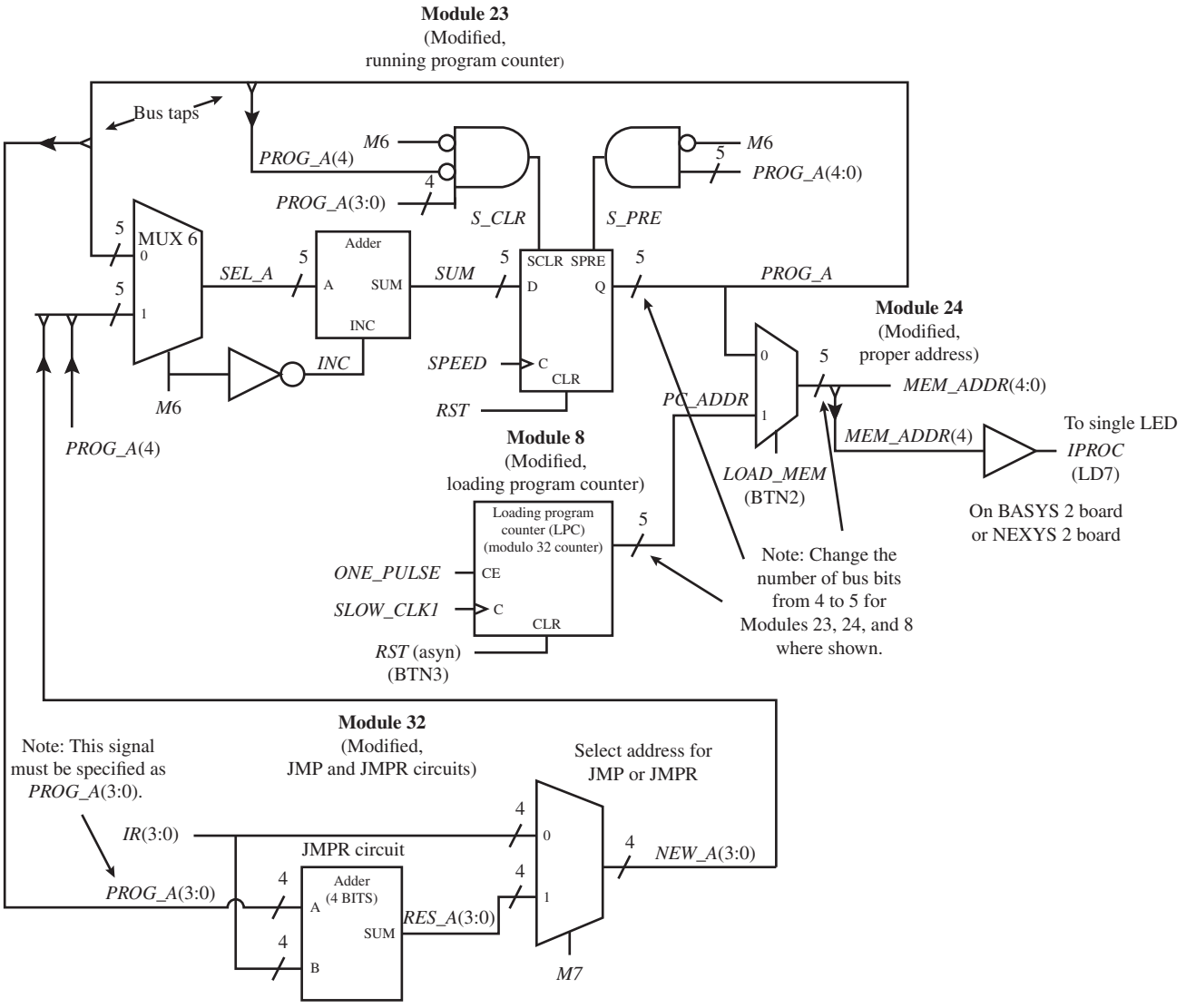


FIGURE E23.2 Modified circuits for the running program counter, the proper address, the loading program counter, and the JMPR circuit for VBC1-E

When the circuits in Figure E23.1 and E23.2 are incorporated in the design, program memory can be loaded with 32 instructions. The first 16 memory locations represent instruction memory and the last 16 memory locations represent the extended instruction memory. The instruction memory and the extended instruction memory can be loaded either manually or by including the desired bits as initialization bits for the memory in the VHDL code. At this point in the design of VBC1-E, only instructions that are placed in instruction memory (addresses 0 through 15) can be executed. In the next experiment, the instructions INT and IRET will be added, which will allow you to execute instructions in both instruction memory (addresses 0 through 15) and extended instruction memory (addresses 16 through 31).

Recommended Pre-Lab:

1. Create a new project named JMP_JMPR, and write complete VHDL code for the JMP and JMPR circuits. Make the input and output signal names the same as those used by module 32 in Figure E23.2. *RES_A*(3:0) is an internal signal between the JMPR circuit and the select address for JMP or JMPR. Run a simulation to verify your design works. If your design does not work, you must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project and add your code for Experiment 22 (or Experiment 21 if you skipped Experiment 22) to the project.
3. Modify the internal signal for modules 6, 10 and 24 so that there are 5 bus bits for the signal MEM_ADDR as shown in Figures E23.1 and E23.2.
4. Modify the definition of the instruction memory (module 6) so that there are 32 memory storage locations as shown in Figure E23.1.
5. Modify the code for module 10 to include the bus for the signal *MEM_ADDR*(3:0). This is important because only bits 3 downto 0 are used in the circuit for module 10 as shown in Figure E23.1.
6. Modify the internal signals for module 23 so that there are 5 bus bits for the signals *SEL_A* and *SUM* as shown in Figure E23.2.
7. Modify the code for module 23 by changing the conditional signal assignment at input 1 of MUX 6 from *NEW_A* to *PROG_A*(4) & *NEW_A* as shown in Figure E23.2.
8. Modify the code for module 23 by changing the conditional signal assignment for the loadable D flip-flop to a process. The process for the D flip-flop must contain the synchronous clear signal *S_CLR* and the synchronous preset *S_PRE* as shown in Figure E23.2. These synchronous signals must be placed in the code after the rising edge of the clock signal *SPEED*.
9. Modify the code for module 24 to include the output signal *IPROC* as shown in Figure E23.2. The output signal *IPROC* is connected to a single LED. When the LED is not lighted, the address range supplied to the total instruction memory (module 6) is 00000 through 01111. When the LED is lighted, the address range supplied to the total instruction memory is 10000 through 11111.
10. Modify the internal signal for modules 23 and 24 so that there are 5 bus bits for the signal *PROG_A* as shown in Figure E23.2.
11. Modify the internal signal for modules 8 and 24 so that there are 5 bus bits for the signal *PC_ADDR* as shown in Figure E23.2.
12. Modify the code for module 32 (the JMPR circuit) to include the bus signal *PROG_A*(3:0): This is important because only bits 3 down to 0 are used in the circuit for module 32.
13. VBC1-E requires a single LED for the output signal *IPROC*. Use LD7 on the BASYS 2 board or on the NEXYS 2 board for this LED as indicated in Figure E23.2.
14. Complete the design cycle for your circuit by doing the following:
    a. Assign package pins for all the port signals in the entity for your design.
    b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings. [Note: The warning: FF/Latch <prog_a_4> (with-

out init value) has a constant value of 0 in block <ProjectName> can be safely ignored because prog_a(4) is not assigned a value of 1 in this design, which is what the warning is telling you.] If there are other errors and warnings that need to be corrected, correct your VHDL code, then rerun Generate Programming File.

c. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.

15. For information on how to manually load a program into memory, see Appendix E. To verify that you can load and read back the contents of the instruction memory and the extended instruction memory, manually load 3 into all the locations of the instruction memory (IM), then manually load C into all the locations of the extended instruction memory (EIM). The LED for the signal *IPROC* will turn off for all the locations of the instruction memory, and it will turn on for all the locations of the extended instruction memory. If you can load both the IM and the EIM and read back the information that you stored, then your circuit is probably working correctly. If your circuit is not working correctly, you either made a mistake assigning the pin for the signal *IPROC*, or you may have design errors in you VHDL code that you must find and fix.
16. Use the EASY1-E editor to enter the instructions shown in Program E23.1. This program works on the full-blown version and also on the stripped-down version of VBC1-E. Use the EASY1-E simulator to single step through the program or run the program so that you understand what it does.

PROGRAM E23.1
Limited multiply program

```
;VBC1-E Exp 23 Program 1
;Limited multiply program
;Multiply 2 4-bit numbers at input ports 0 and 1
;provides correct 4-bit result at output port 1
;result cannot be larger than 15 for this program

in r0, 0
in r1, 1
store r0, 0
store r1, 3
repeat:
fetch r0, 2
fetch r1, 0
add r0, r1
store r0, 2

fetch r1, 3
addi r1, 15
store r1, 3
jnz r1, repeat

out r0, 1
halt
```

17. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E23.1 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 31, just as they are in the memory of EASY1-E. Run

Program E23.1. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you may have design errors in your VHDL code that you must find and fix.

18. Manually load 0 into all locations of the memory, both IM and EIM of VBC1-E, then manually load the machine code for Program E23.1 into the memory of VBC1-E. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 31, just as they are in the memory of EASY1-E. Single step and also run Program E23.1. VBC1-E should provide the same results as EASY1-E for each instruction.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is the execution of Program E23.1 with your VBC1-E design. You also need to show that you can store 3 in every location of IM and C in every location of EIM and read this information back.
2. Include the complete VHDL code for your VBC1-E design.
3. Include a printout of the Edit Constraints (Text) for your VBC1-E design, which is generated by running Edit Constraints (Text).
4. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
5. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 24: DESIGNING AND TESTING VBC1-E (INT AND IRET INSTRUCTIONS)

In this experiment, you will practice working with modules to provide the INT (software interrupt) and IRET (interrupt return) instructions. Adding the INT and IRET instructions requires adding two new circuits called the select circuit (module 33) and the store PCPLUS1 circuit (module 34). The running program counter (module 23) is also modified and divided into two modules named modules 23a and 23b.

This is the sixth in a series of projects to design and test VBC1-E. Experiment 24 consists of the annotated schematics shown in Figures E24.1 and E24.2.

Figure E24.1 shows the modified circuit for the running program counter and the select circuit for VBC1-E.

Figure E24.2 shows the store PCPLUS1 circuit for VBC1-E.

When the circuits in Figures E24.1 and E24.2 are incorporated in the VBC1-E design, you can write a program using any of the 25 instructions for VBC1-E, including the INT and IRET instructions. Circuits that are required to perform a hardware interrupt will be included in the next experiment.

Recommended Pre-Lab:

1. Create a new project named Store_PCPLUS1, and write complete VHDL code for the store PCPLUS1 circuit using simple Boolean equations and a conditional signal assignment. Make the input and output signal names the same as those used by module 34 in Figure E24.2. Run a simulation to verify your design works. If your design does not work, you

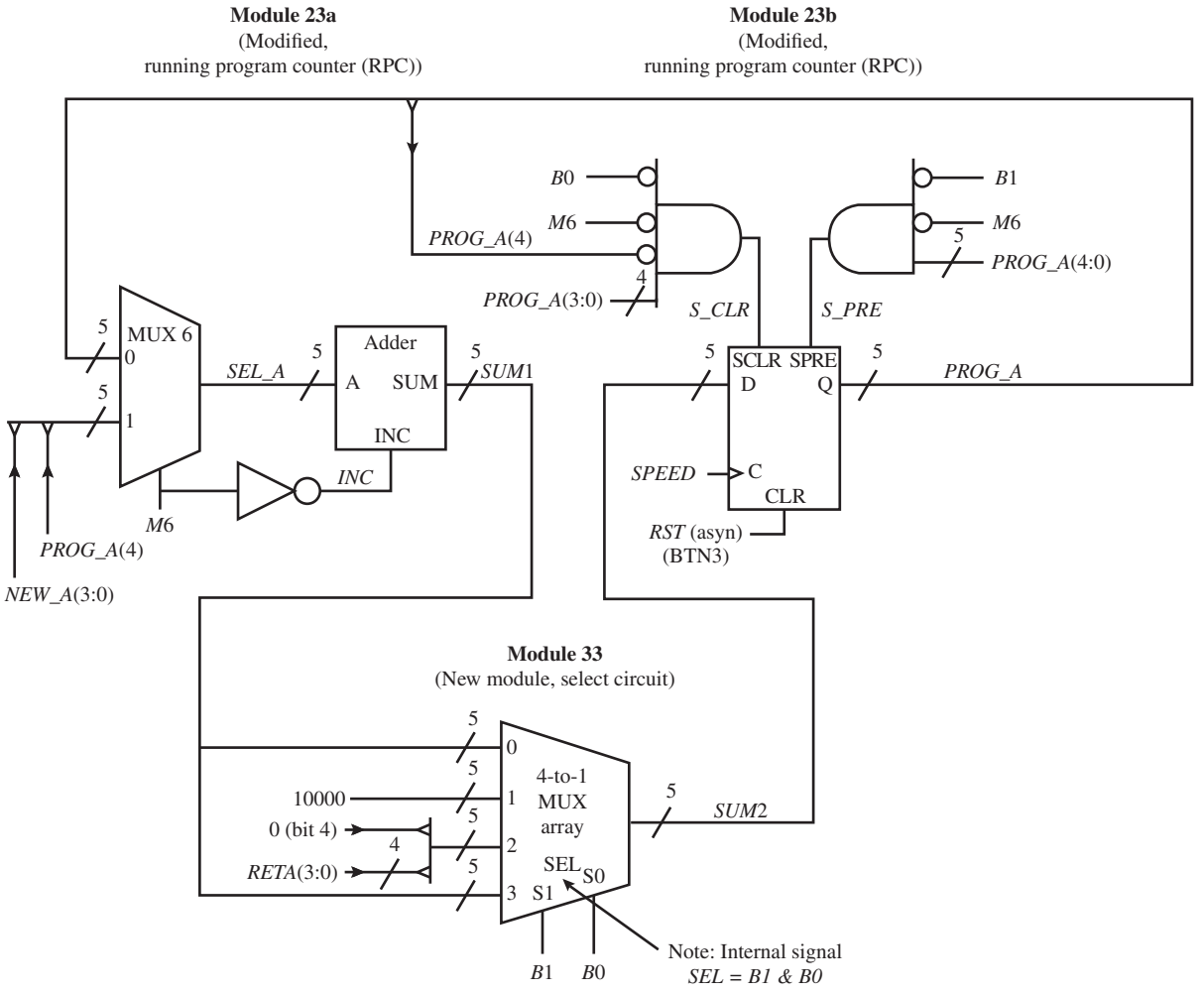


FIGURE E24.1 Modified circuit for the running program counter and the select circuit for VBC1-E

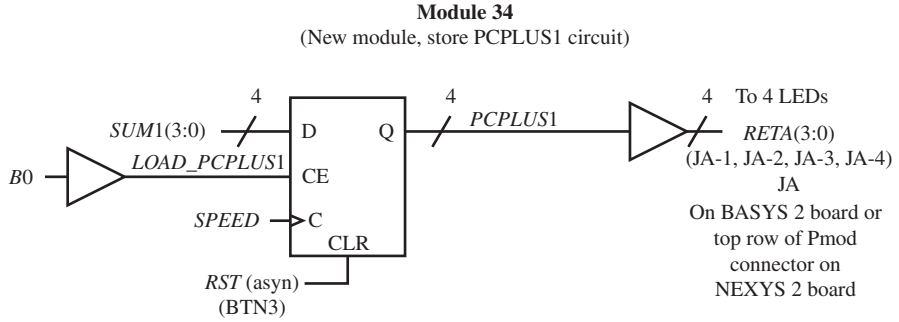


FIGURE E24.2 Store PCPLUS1 circuit for VBC1-E

must find the error or errors and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]

2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project and add your code for Experiment 23 to the new project.
3. Add module 33 to your design using a conditional signal assignment. Divide module 23 as shown in Figure E24.1 so that SUM1 is an output of module 23a and SUM2 is an input to module 23b.
4. Modify module 23b by including the signals *B*0 and *B*1 to the two AND gates as shown in Figure E24.1.
5. Add module 34 to your design using simple Boolean equations and a conditional signal assignment.
6. Use the instruction decoder truth tables for the INT and IRET instructions to obtain the Boolean equations for the control signals. Modify the instruction decoder (module 14) to handle the INT and IRET instructions.
7. VBC1-E requires four LEDs for the output signal *RETA*(3:0). To test your design requires an additional peripheral module board—that is, an LED module. Plug the LED module into the Pmod connector JA on a BASYS 2 board or on a NEXYS 2 board to observe the value of the output signal *RETA*.
8. Complete the design cycle for your circuit by doing the following:
   a. Assign package pins for all the port signals in the entity for your design.
   b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
   c. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.
9. To verify that VBC1-E indicates the correct signal for *RETA*, manually load the machine code for the instruction INT, which is "a4," at address 00000, and manually load the machine code for the instruction JMP 0, which is "c0," at address 00001. Also load the machine code for IRET, which is "ac," at address 10000. Use the EASY1-E editor to enter the instructions shown in Program E24.1. This program works on the full-blown version and also on the stripped-down version of VBC1-E.

PROGRAM E24.1
Test program for four LEDs for RETA at addresses 0, 1, and 16

```
;VBC1-E Exp 24 Program 1
;Test program for 4 LEDs for RETA
;at addresses 0, 1, and 16
int
jmp 0

biproc
iret
```

Single step through Program E24.1 with EASY1-E and observe the value of the signal *RETA*. Reset VBC1-E and single step through Program E24.1 to verify that VBC1-E works the same way as EASY1-E. If the signal *RETA* does not provide the same value via the four LEDs on VBC1-E as indicated with EASY-E, then you either made a mistake assigning pins, entering the code into VBC1-E manually, or you may have design errors in your VHDL code that you must find and fix.

10. Use the EASY1-E editor to enter the instructions shown in Program E24.2. This program works on the full-blown version and also on the stripped-down version of VBC1-E. Use the EASY1-E simulator to single step through the program so that you understand what it does.

PROGRAM E24.2
Test program for four LEDs for RETA at addresses 0, 14, 15, 16, and 31

```
;VBC1-E Exp 24 Program 2
;Test program for 4 LEDs for RETA
;at addresses 0, 14, 15, 16, and 31
jmp 14
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
int
jmp 0

biproc
jmp 15
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
mov r0,r0
iret
```

11. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E24.2 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 31, just as they are in the memory of EASY1-E. Single step and also run Program E24.2. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you must find and fix your error or errors.
12. Use the EASY1-E editor to enter the instructions shown in Program E24.3. This program works on the full-blown version and also on the stripped-down version of VBC1-E. Use the EASY1-E simulator to single step through the program or run the program so that you understand what it does.

PROGRAM E24.3
This program works on the full-blown version and also on the stripped-down version of VBC1-E

```
;VBC1-E Exp 24 Program 3
;Binary up counter program in Instruction Memory, op 0
;Software interrupt to Extended Instruction Memory
;robot eye program in Extended Instruction Memory, op 1
loadi r0,1
out r0,0
addi r0,1
int
jmpr 13

biproc
loadi r1,1
out r1,1
loadi r1,2
out r1,1
loadi r1,4
out r1,1
loadi r1,8
out r1,1
loadi r1,4
out r1,1
loadi r1,2
out r1,1
loadi r1,1
out r1,1
iret
```

13. Comment out Program E24.2 in your VHDL design. Load the machine code for Program E24.3 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 31, just as they are in the memory of EASY1-E. Single step and also run Program E24.3. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you must find and fix your error or errors.
14. Manually load 0 into all locations of the memory, both IM and EIM of VBC1-E, then manually load the machine code for Program E24.3 into the memory of VBC1-E. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 31, just as they are in the memory of EASY1-E. Single step and also run Program E24.3. VBC1-E should provide the same results as EASY1-E for each instruction.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is the execution of Program E24.3 with your VBC1-E design.
2. Include the complete VHDL code for your VBC1-E design.
3. Include a printout of the Edit Constraints (Text) for your VBC1-E design, which is generated by running Edit Constraints (Text).

4. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
5. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 25: DESIGNING AND TESTING VBC1-E (FINAL HARDWARE DESIGN FOR VBC1-E)

In this experiment, you will practice working with modules to provide a hardware interrupt. Adding a hardware interrupt requires adding modules (modules 35, 36, 37, and 38) and modifying two existing modules (modules 23b and 34).

This is the seventh and final project in the series of projects to design and test VBC1-E. Experiment 25 consists of the annotated schematics shown in Figures E25.1 and E25.2.

Figure E25.1 shows the module for a debounced one-pulse trigger interrupt circuit (module 35) and the modified circuit for the running program counter (module 23b) for VBC1-E.

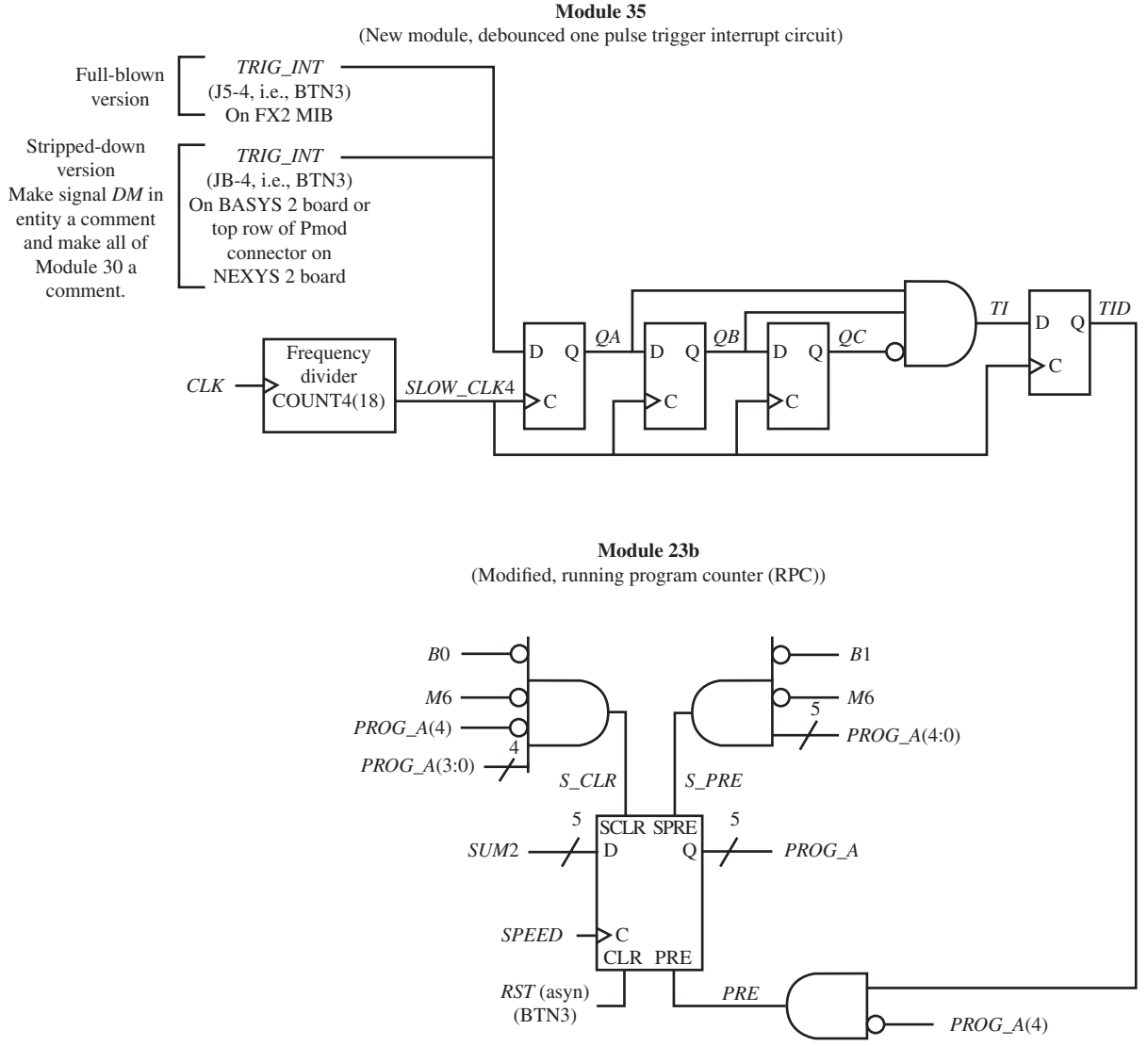


FIGURE E25.1 Debounced one-pulse trigger interrupt circuit and modified circuit for the running program counter for VBC1-E

Figure E25.2 shows circuits for displaying the signal *RETA* (return address) for both a hardware interrupt and a software interrupt. For a software interrupt, *RETA* must contain the address of the instruction following the INT instruction. For a hardware interrupt, *RETA* must contain the address of the instruction that was interrupted and was not completed.

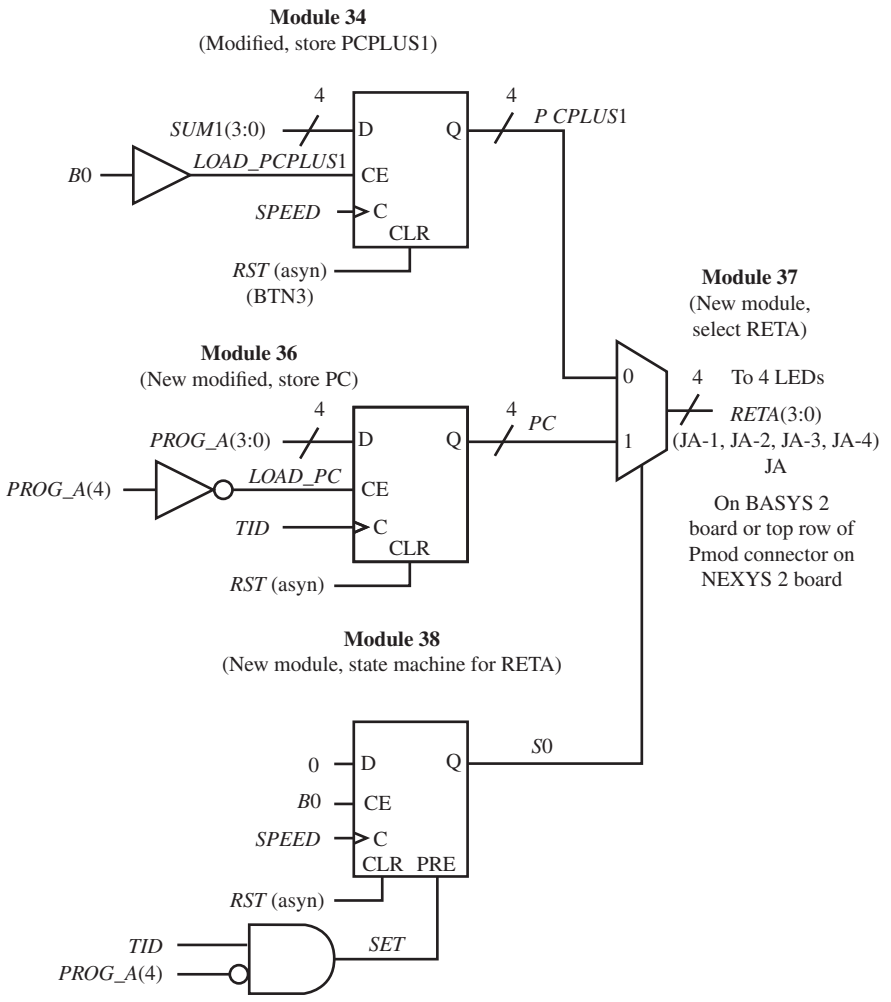


FIGURE E25.2 Circuits for displaying the signal *RETA* (return address) for VBC1-E

When the circuits in Figures E25.1 and E25.2 are incorporated in the design, a program that provides a return for a hardware interrupt can be written that will execute instructions that are placed in extended instruction memory. *RETA* provides the return address of the hardware interrupt that occurs when the push button that generates the signal *TRIG_INT* is pressed. *RETA* also provides the correct return address for the instruction *INT* (the software interrupt instruction) if it is executed in instruction memory.

Recommended Pre-Lab:

1. Create a new project named SM_RETA, and write complete VHDL code for the state machine for RETA using a Boolean equation for the AND operation and a conditional signal assignment for the loadable D flip-flop. Make the input and output signal names the same as those used by module 38 in Figure E25.2 except for *PROG_A*(4). Change *PROG_A*(4) to *PROG_A*4, because this will be a stand-alone design. Run a simulation to verify your design works. If your design does not work, you must find the error or errors

and fix them. [Note: See Appendix B (Obtaining Simulations via the VHDL Test Bench Program) for help.]
2. Your instructor will provide you with additional information about what should be submitted for your pre-lab report requirements—for example, VHDL code, test bench code, and simulation waveforms.

Tasks:

1. Write complete VHDL code using the specified design style via a flat design approach. Use documentation style M introduced in Chapter 12, Section 12.5.1, Listing 12.4, for marking each internal signal and each section of code in your VHDL design. Be sure to place all modules in the code section in numerical order.
2. Start a new project and add your code for Experiment 24 to the new project.
3. Add module 35 to your design. Because module 35 has some of the same circuits as module 7, you can copy the code for module 7, and modify it to obtain module 35.
4. For the stripped-down version, be sure to make signal *DM* a comment and make module 30 a comment. This effectively removes the data memory monitor from the design of the stripped-down version of VBC1-E. The full-blown version retains the data memory monitor because the FX2 MIB provides a Pmod connector for the signal *TRIG_INT*, so that module 30 does not have to be removed from the design.
5. Modify module 23b by adding the circuit for the signal *PRE* (preset).
6. Modify module 34 by removing the buffer and the signal *RETA* that was output by the buffer. The output for module 34 is the signal *PCPLUS*1, as shown in Figure E25.2.
7. Add module 36 to your design. Use a Boolean equation for the NOT operation and a conditional signal assignment for the loadable D flip-flop.
8. Add module 37 to your design using a conditional signal assignment for the MUX.
9. Add module 38 to your design.
10. VBC1-E requires a push-button switch for the signal *TRIG_INT*. For the full-blown version of VBC1-E, plug a button module into Pmod connector J5. For the stripped-down version of VBC1-E, remove the LED module plugged into Pmod connector JB and plug a button module into Pmod connector JB. See Figures E19.1 and E19.2.
11. Complete the design cycle for your circuit by doing the following:
    a. Assign package pins for all the port signals in the entity for your design.
    b. Generate a programming file, then check to see if your VHDL code needs to be corrected based on reported errors and warnings; if so, correct your VHDL code, then rerun Generate Programming File.
    c. Download the programming file into the FPGA on a BASYS 2 or NEXYS 2 board.
12. Use the EASY1-E editor to enter the instructions shown for FB in Program E25.1, if you designed the full-blown version of VBC1-E. If you designed the stripped-down version of VBC1-E, use Program E25.1 with the title "VBC1-E Exp 25 Program 1 Stripped." Use the EASY1-E simulator to single step through the program or run the program so that you understand what it does.

PROGRAM E25.1
Full-blown version and stripped-down version

```
;VBC1-E Exp 25 Program 1 FB
;Executes store and fetch instructions
;Uses all input and output ports
;hardware interrupt, displays 0, 15, and 5
in r0,0       ;set input port 0 to any value
store r0,0
fetch r1,0
out r1,0      ;output port 0 follows input port 0
                                                    (Continued)
```

```
in r1,1       ;set input port 1 to any value
store r1,1
fetch r0,1
out r0,1      ;output port 1 follows input port 1
in r0,2       ;set input port 2 to any value
store r0,2
fetch r1,2
out r1,2      ;output port 2 follows input port 2
in r1,3       ;set input port 3 to any value
store r1,3
fetch r0,3
out r0,3      ;output port 3 follows input port 3

biproc
loadi r1,0
out r1,0
out r1,1
out r1,2
out r1,3
loadi r1,15
out r1,0
out r1,1
out r1,2
out r1,3
loadi r1,5
out r1,0
out r1,1
out r1,2
out r1,3
iret

;VBC1-E Exp 25 Program 1 Stripped
;Executes store and fetch instructions
;only uses input and output ports 0 and 1
;hardware interrupt, displays 0, 15, and 5
in r0,0       ;set input port 0 to any value
store r0,0
fetch r1,0
out r1,0      ;output port 0 follows input port 0
in r1,1       ;set input port 1 to any value
store r1,1
fetch r0,1
out r0,1      ;output port 1 follows input port 1

biproc
loadi r1,0
out r1,0
out r1,1
loadi r1,15
out r1,0
```

```
out r1,1
loadi r1,5
out r1,0
out r1,1
iret
```

13. For information on how to initialize memory at startup, see Appendix E. Load the machine code for Program E25.1 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 31, just as they are in the memory of EASY1-E. Single step and also run Program E25.1. If VBC1-E does not provide the same results as EASY1-E for each instruction, then you made a mistake assigning the pin J5-4 for BTN3 (signal *TRIG_INT*) for the full-blown version, made a mistake assigning the pin JB-4 for BTN3 (signal *TRIG_INT*) for the stripped-down version, or you may have design errors in your VHDL code that you must find and fix.
14. Use the EASY1-E editor to enter the instructions shown in Program E25.2. This program works on the full-blown version and also on the stripped-down version of VBC1-E. Use the EASY1-E simulator to single step through the program or run the program so that you understand what it does.

PROGRAM E25.2
This program works on the full-blown version and also on the stripped-down version of VBC1-E

```
;VBC1-E Exp 25 Program 2
;Binary up counter program in Instruction Memory, op 0
;Hardware interrupt to Extended Instruction Memory
;Lights in progressive left sequence in Extended Instruction
;Memory, op 1
loadi r0,1
back:
out r0,0
addi r0,1
jmp back

biproc
loadi r1,1
out r1,1
loadi r1,2
out r1,1
loadi r1,4
out r1,1
loadi r1,8
out r1,1
loadi r1,0
out r1,1
iret
```

15. Comment out Program E25.1 in your VHDL design. Load the machine code for Program E25.2 into the memory of VBC1-E by initializing memory. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 31, just as they are in the memory of EASY1-E. Run Program E25.2. If VBC1-E

does not provide the same results as EASY1-E for each instruction, then you must find and fix your error or errors.

16. Manually load 0 into all locations of the memory, both IM and EIM of VBC1-E, then manually load the machine code for Program E25.2 into the memory of VBC1-E. Single step through the memory of VBC1-E to verify that the machine code is properly loaded in the addresses 0 through 31, just as they are in the memory of EASY1-E. Single step and also run Program E25.2. VBC1-E should provide the same results as EASY1-E for each instruction.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working designs and get them signed off by your lab instructor. First print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working designs. Your final working designs are the execution of Programs E25.1 and E25.2 with your VBC1-E design.
2. Include the complete VHDL code for your VBC1-E design.
3. Include a printout of the Edit Constraints (Text) for your VBC1-E design, which is generated by running Edit Constraints (Text).
4. Be sure to write a short paragraph summarizing the work you did for this experiment, and describe any problems you may have encountered while obtaining your solutions. You may include any helpful hints and improvements you may think of for this experiment.
5. Your lab instructor may add additional requirements for this experiment.

EXPERIMENT 25L: DESIGNING A LOADER FOR INSTRUCTION MEMORY FOR VBC1-E

by Scott M. Marshall and Richard S. Sandige

25L.1 Background

In this experiment, you will learn how to design a loader for the total instruction memory for VBC1-E to create VBC1-EL. When completing this experiment, focus on applying existing skills to new situations and learning good design techniques for interfacing a FPGA with other circuit components.

This experiment provides an extension to VBC1-E (which was completed in Experiment 25) and is not required for VBC1-E to function correctly.

You do not have to have completed Experiment 17L to complete this experiment. If you have already completed Experiment 17L, most of the following details will be review, and only a few bus width changes will be required to convert module L2 to module EL2, while modules L1 and L3 will remain unchanged to form modules EL1 and EL3, respectively.

The memory loader works by communicating with a USB communication circuit placed on a BASYS 2 board or on a NEXYS 2 board by Digilent Inc. The VBC1-L memory loader software program uses a library provided by Digilent Inc. to communicate with the FPGA via the USB circuit.

25L.2 Module Overview

The heart of the additional hardware designed for communicating with the USB circuit is a CFSM (complex finite state machine). CFSMs were previously discussed in Chapter 9. Four synchronizer circuits (sync circuits), each of which consist of two cascaded D flip-flops (not a

single D flip-flop), serve as USB I/O synchronizers for the CFSM and form a new module, module EL1. The CFSM and a single loadable register, explained in Chapter 12, Section 12.4, form a new module, module EL2. In addition, an OR gate is added as module EL3. Figure E25L.1 shows the block diagrams for modules EL1, EL2, and EL3.

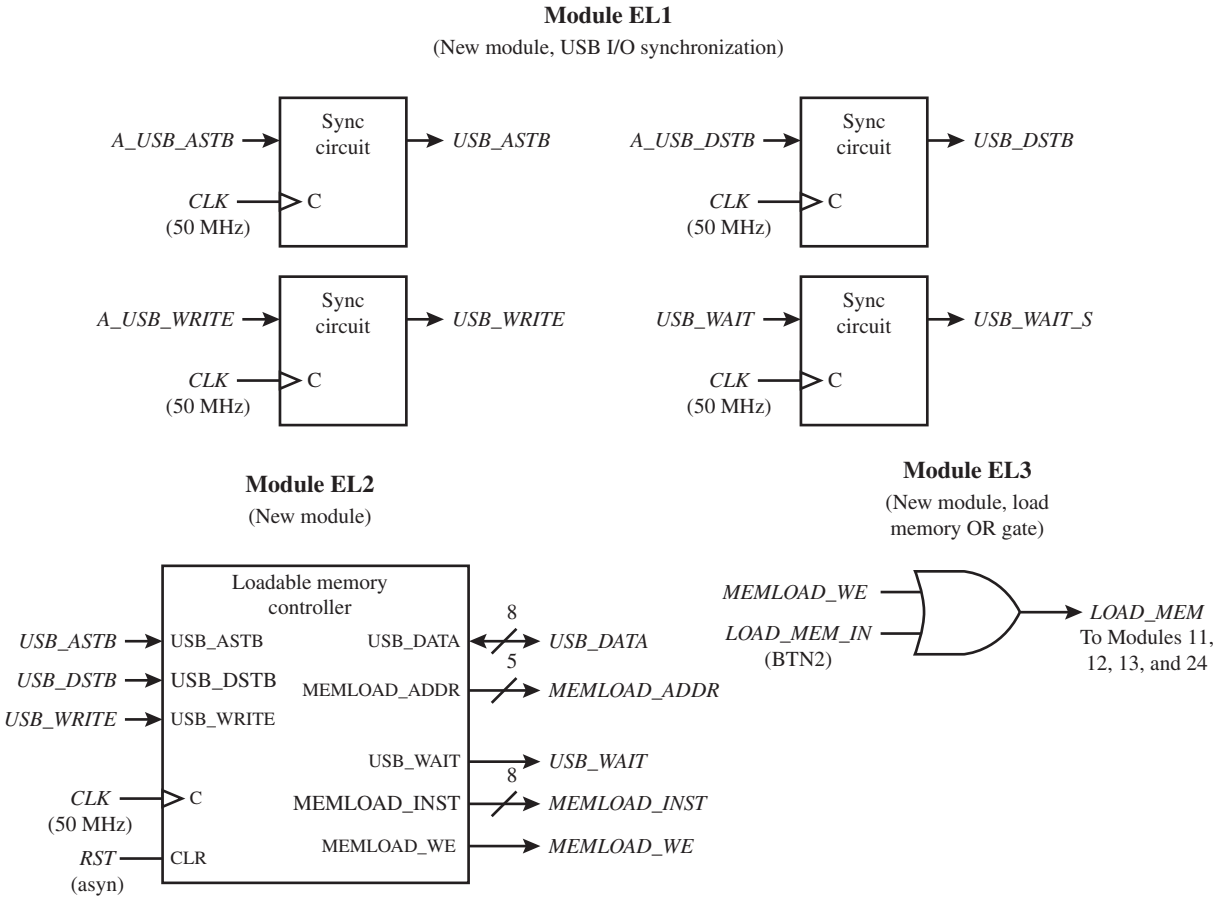


FIGURE E25L.1 Overview block diagrams for new modules EL1, EL2, and EL3

Each sync circuit in module EL1 is a synchronizer circuit that consists of two D flip-flops connected as shown in the Figure E25L.1A.

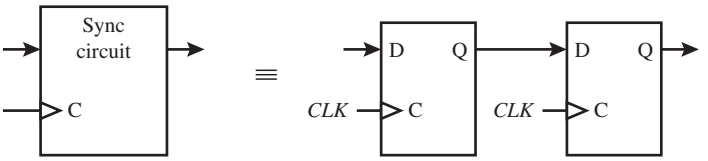


FIGURE E25L.1A Synchronizer circuit

Things to note regarding Figures E25L.1 and E25L.1A:

- Figure E25L.1 shows block diagrams of the new modules EL1, EL2, and EL3 added to VBC1-E.
- The external input signals for module EL1 are *CLK*, *A_USB_ASTB*, *A_USB_DSTB*, and *A_USB_WRITE*, while the external output signal is *USB_WAIT_S*. *Note*: The notation *A_* is used for asynchronous external input signals coming into the FPGA—for example, *A_USB_ASTB*—and the notation *_S* is used for synchronous external output signals leaving the FPGA—for example, *USB_WAIT_S*. The remaining signals are internal signals.

- The external input signals for module EL2 are *CLK* and *RST*, while signal *USB_DATA*(7:0) is of type inout, because it is used for bidirectional communication with the USB circuit. The remaining signals are internal signals.
- Module EL3 has the external internal signal *LOAD_MEM_IN* and internal signals *MEMLOAD_WE* and *LOAD_MEM*.
- Signals for communication with the USB chip have *USB* as a part of the name, while signals pertinent to loading the instruction memory have names that are prefixed with *MEMLOAD*.
- Adjustments to the VHDL entity and internal signal declarations to account for applicable new or modified input, output, and internal signals must be made. The external asynchronous input signals (names with *A_*), synchronous output signals (names with *_S*), and bidirectional *USB_DATA* bus are new I/O signals for the FPGA, while the change of *LOAD_MEM* to *LOAD_MEM_IN* for BTN2 is the only change to the existing I/O signals.
- Reset inputs to the the sync circuits are not required.

25L.3 Memory Loader and CFSM Detail

Figure E25L.2 shows a detailed block diagram of the subcomponents of module EL2.

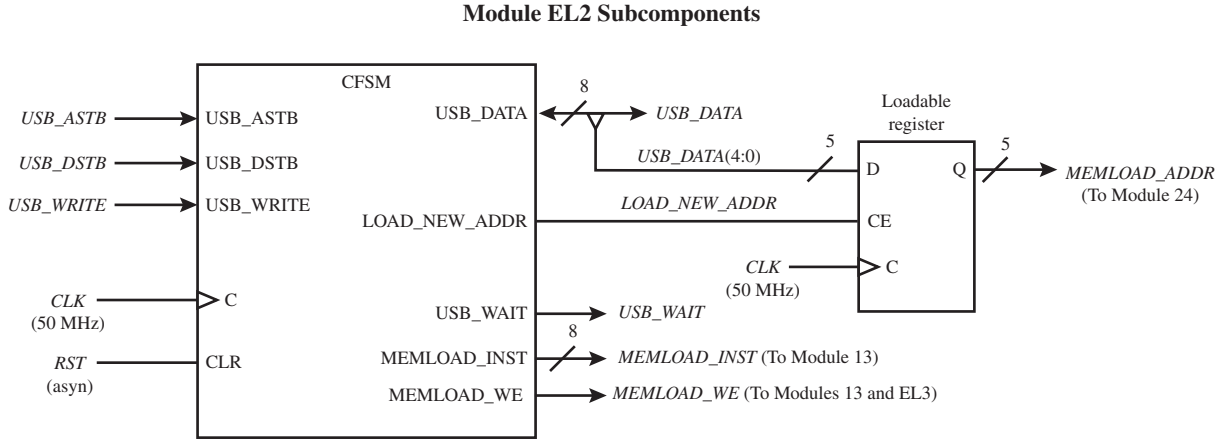


FIGURE E25L.2 Detailed block diagram for the subcomponents of module EL2

Things to note regarding Figure E25L.2:

- Figure E25L.2 represents a block diagram of the subcomponents of module EL2, including the loadable register.
- The loadable register subcomponent stores the current memory address the memory loader CFSM is acting upon, which is provided on the *MEMLOAD_ADDR* output signal.
- The signal *LOAD_NEW_ADDR* joins an output of the CFSM to the CE (clock-enable) input of the loadable register.
- The *MEMLOAD_ADDR* output signal is 5 bits wide and provides the address of the instruction to load, while the *MEMLOAD_INST* output signal is 8 bits wide and provides the instruction to load, and the *MEMLOAD_WE* output signal provides a write-enable signal for the instruction memory.

The CFSM in module EL2 should be designed using the two-process PS/NS method. A state diagram detailing the proper function of the CFSM is provided in Figure E25L.3.

Things to note Regarding Figure E25L.3:

- Figure E25L.3 is a state diagram for the CFSM in module EL2, and uses the show all values convention, rather than the show where true convention, both explained in Chapter 9, Section 9.7.

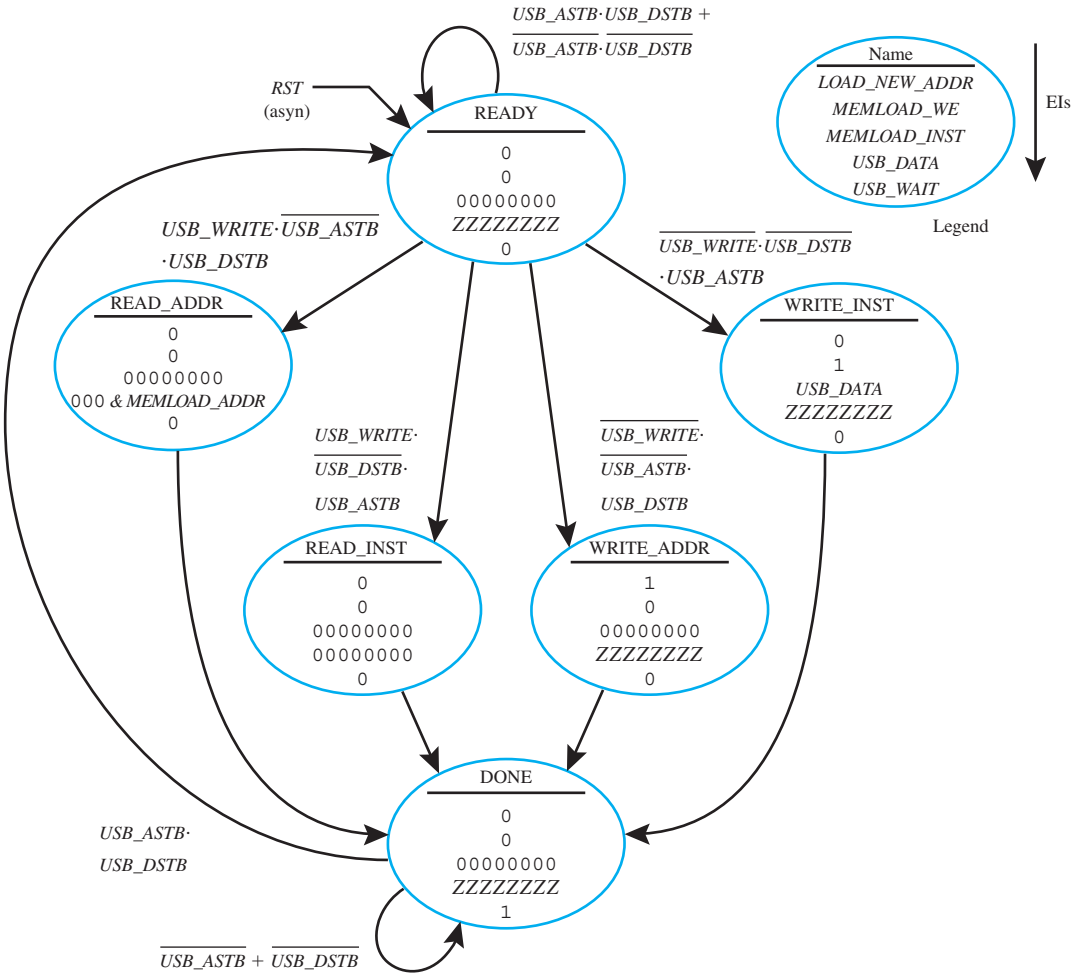


FIGURE E25L.3 State diagram for module EL2 CFSM

- Where the values 0, 00000000, and ZZZZZZZZ are specified as the Moore outputs in the state diagram, use these as the default values that precede the case statement. Values or variables other than these should be specified for states within the case statement.
- While the *USB_DATA* signal is an output, it also appears as an input in the WRITE_INST state because it is a bidirectional signal. Data can both be written to and read from the bidirectional *USB_DATA* bus. Data can be read when *USB_DATA* is set to ZZZZZZZZ.
- The signals *USB_ASTB* and *USB_DSTB* are active low signals and are labeled accordingly in the state diagram. These two signals are strobe signals driven by clock-synchronization D flip-flops in module EL1, which are driven by the USB circuit. These strobe signals are pulsed to trigger the start of either address or instruction communication with the USB circuit.
- The signal *USB_WRITE* is active low, such that when the signal is low, data are written to the FPGA.
- The READ states are reached when the USB circuit (and thus the computer) is reading the current address or instruction from the CFSM and instruction memory, while the WRITE states are reached when the USB circuit (and thus the computer) is writing a new address or instruction to the CFSM and instruction memory.
- Situations where no state change occurs (such as when READY points to itself) should not be described using specific conditions in if and elsif clauses, but rather, should be described

by else clauses. The else clauses act as catch-all states and will prevent the generation of any latches.

- The value of 00000000 for the signal *USB_DATA* in the READ_INST state is used because reading the instruction memory on the computer is not supported in VBC1-EL.
- For more details on Digilent Inc.'s communication protocol, see the document "Digilent Parallel Interface Model Reference Manual" at http://www.digilentinc.com/Data/Products/ADEPT/DpimRef%20programmers%20manual.pdf.

It is important to use a default output value of ZZZZZZZZ (make Zs uppercase, not lowercase) for the *USB_DATA* signal to prevent bus contention or driver fight, as previously discussed in Chapter 7, Section 7.3. Setting the *USB_DATA* signal to ZZZZZZZZ places the FPGA pins in a high impedance state, which can be thought of as being disconnected. This allows the USB circuit to safely send data to the FPGA without hardware damage, the FPGA to read the data, and the *USB_DATA* signal to work as a bidirectional bus. For more information on data bus sharing and the high impedance state, see Chapter 7, Sections 7.2 and 7.3.

25L.4 Good Design Practices: Interfacing with Other Components

It is important to realize that a goal of this experiment is not only to review previously learned VHDL design concepts and to create an easy and fast way to load VBC1-E instruction memory, but to also learn new techniques and best practices when interfacing an FPGA with other components.

Push-button and switch events are asynchronous, because a user can trigger a push button or switch independent of the clock signal *CLK*. The asynchronous behavior of other components, such as the USB circuit, may not be as apparent; however, they too are asynchronous because a *CLK* signal is not transferred between them and the FPGA.

Asynchronous signals can cause problems within the FPGA if their levels change when the *CLK* signal reaches the triggering (rising and/or falling) edge. When this occurs, the FPGA may not use the intended value of the asynchronous signal and can enter a metastable state.

To prevent problems, it is good design practice to synchronize asynchronous signals, such as inputs from other circuit components, as well as outputs of combinational logic, which can become asynchronous even if originally generated by synchronous logic due to propagation delays. It considered good design practice to synchronize a signal by feeding it through two cascaded D flip-flop as shown in Chapter 9, Section 9.10.

In the case of VBC1-EL, the asynchronous FPGA input signals *A_USB_ASTB*, *A_USB_DSTB*, and *A_USB_WRITE* should all be synchronized, resulting in the signals *USB_ASTB*, *USB_DSTB*, and *USB_WRITE*. In addition, the asynchronous combinational logic output signal *USB_WAIT* from module EL2 should be synchronized to form the FPGA output signal *USB_WAIT_S*. Module EL1 serves this purpose.

Note that best design practices would also synchronize the *USB_DATA* signal. However, synchronizing *USB_DATA*, which is a bidirectional tri-stated bus, is an advanced technique that will not be covered here.

25L.5 Signal Routing Modifications

To allow the instruction memory to be loaded using either the push buttons and switches or the memory loader (module EL3), the *LOAD_MEM* signal, which is high when memory is being loaded, must be changed to include the *MEMLOAD_WE* signal from module EL2. To do so, the *LOAD_MEM* signal is renamed to *LOAD_MEM_IN* in both the entity and the .ucf (implementation constraints) file and must be connected to BTN2. Then, a new internal signal by the name of *LOAD_MEM* is created and is assigned as the OR combination of *LOAD_MEM_IN* and *MEMLOAD_WE*, forming the OR gate in module EL3. All existing modules should continue to use the *LOAD_MEM* signal, not *LOAD_MEM_IN*. Now, when either BTN2 is pressed or when *MEMLOAD_WE* is high, memory will be loaded.

In order to accommodate the additional address, instruction, and write-enable signals from the memory loader, modules 13 (load memory) and 24 (proper address) must be modified. A single 8-bit-wide 2-to-1 MUX array along with a single 2-to-1 MUX must be added to module 13, along with the internal signals *LOAD_INST*(7:0) and *LOAD_WE*. A single 5-bit-wide 2-to-1 MUX array, along with the internal signal *LOAD_ADDR*(4:0), must be added to module 24. The modified versions of modules 13 and 24 are detailed in Figures E25L.4 and E25L5, respectively.

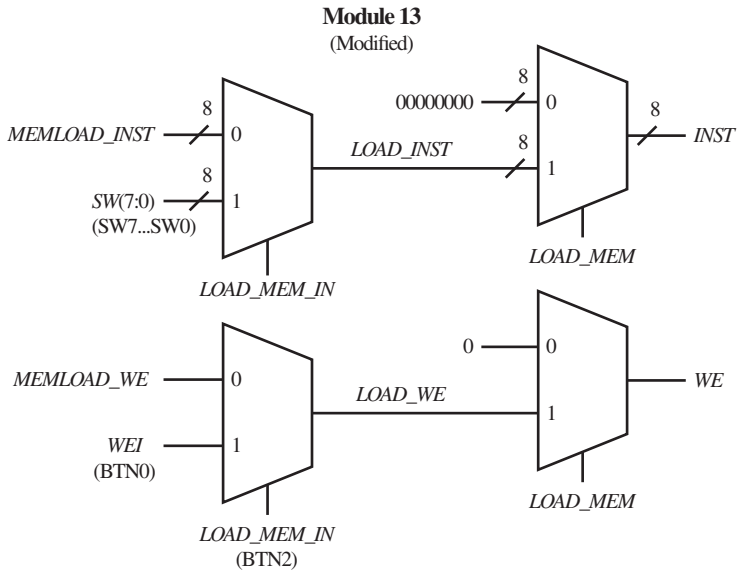


FIGURE E25L.4 Block diagram of modified module 13

Things to note regarding Figure E25L.4:

- The *LOAD_INST*(7:0) and *LOAD_WE* internal signals are added, along with an 8-bit 2-to-1 MUX array and a single 2-to-1 MUX.
- The signal *LOAD_MEM_IN* chooses between the memory loader and the manual push buttons and switches for loading memory. When *LOAD_MEM_IN* is high, *LOAD_MEM* is also high (from module EL3), and memory is loaded manually.
- The signal *LOAD_MEM* indicates when memory is being loaded.

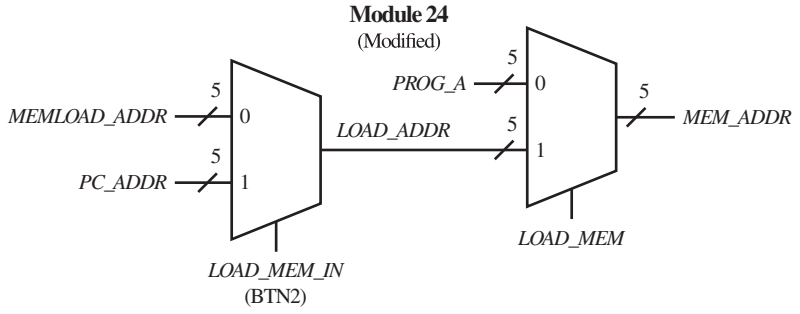


FIGURE E25L.5 Block diagram of modified module 24

Things to note regarding Figure E25L.5:

- The *LOAD_ADDR*(4:0) internal signal is added along with a 5-bit 2-to-1 MUX array.
- The signals *LOAD_MEM_IN* and *LOAD_MEM* act similarly as they do in the modified version of module 13, detailed in Figure E25L.4.

Upon completion of the additions of modules EL1, EL2, and EL3 and the modifications of modules 13 and 24, the design of VBC1-EL is complete. The memory loader (module EL2) can

now control the instruction memory just as if memory is being loaded manually. Program E25.2 in Experiment 25 can be used to verify completion of VBC1-EL. For information on loading programs with VBC1-EL and on how to use the VBC1-L (VBC1-EL) memory loader program, see Appendix E.

Tasks:

1. Write the complete VHDL design for the modifications and additions described using the design styles you prefer via a flat design approach. Use documentation style M, introduced in Chapter 12, Section 12.5.1, Listing 12.4 and be sure that the new modules are added in numerical order in the code section.
2. Start a new project with an entity that resembles that of Experiment 25, but make the needed modifications for modules EL1, EL2, and EL3. Copy the existing VHDL design from Experiment 25 to use as a starting base for the needed additions and modifications.
3. Add the internal signals needed for modules 13, 24, EL1, EL2, and EL3. When using documentation style M, remember that the signal *LOAD_MEM* is connected to modules EL3, 11, 12, 13, and 24.
4. Modify modules 13 and 24 as described in Figures E25L.4 and E25L.5. Also notice in Figures E25L.4 and E25L.5 that signal *LOAD_MEM_IN* must be connected to BTN2, because the signal *LOAD_MEM* is used as an internal signal for this design.
5. Add module EL1 using the block diagrams in Figure E25L.1.
6. Add module EL2 using the CFSM state diagram in Figure E25L.3 and the block diagrams in Figures E25L.1 and E25L.2.
7. Add module EL3 using the block diagram in Figure E25L.1.
8. Modify the .ucf (implementation constraints) file to account for the change of *LOAD_MEM* to *LOAD_MEM_IN* for BTN2 as well as the new I/O connections for modules EL1 and EL2. See Appendix C for the pin connections for all the signals that communicate with the USB circuit (these signals contain the letters USB).
9. For information on loading memory via the memory loader program, see Appendix E. With the design of VBC1-EL complete, the VBC1-L (VBC1-EL) memory loader program can be used to automatically load memory. Verify that the completed design works by using the VBC1-L (VBC1-EL) memory loader program to load Program E25.2 in Experiment 25 and run the program successfully on a BASYS 2 board or on a NEXYS 2 board.

Lab Report Requirements:

1. To receive full credit, you must demonstrate your final working design and get it signed off by your lab instructor. First, print out a cover page with only the following information: course title, experiment number, your name, and your partner's name(s). Then invite your lab instructor to come to your bench to observe your final working design. Your final working design is demonstrating that Program E25.2 in Experiment 25 can be loaded using the VBC1-L memory loader software program and run successfully on a BASYS 2 board or on a NEXYS 2 board with your VBC1-EL design.
2. Explain the purpose of the synchronizer circuits (sync circuits) in module EL1.
3. Explain why the state machine in module EL2 is considered complex.
4. Explain the purpose of the loadable register subcomponent in module EL2.
5. Explain the purpose of module EL3.
6. Include the complete VHDL code for your VBC1-EL design.
7. Include a text copy of the .ucf (implementation constraints) file.
8. Write a short paragraph explaining the work you did for this experiment, along with a description of any problems you may have encountered and their solutions. Include helpful hints and suggestions of improvement for this experiment.
9. Your lab instructor may add additional requirements for this experiment.

Appendix B

Obtaining Simulations via the VHDL Test Bench Program

B.1 INTRODUCTION

The purpose of Appendix B is to show how to use the VHDL Test Bench program to obtain simulations. A Test Bench Waveform program was available in ISE up through Version 10, in addition to a VHDL Test Bench program. Starting with ISE Version 11, Xilinx® elected to remove the Test Bench Waveform program, but they continued to support the VHDL Test Bench program. To obtain simulations with the latest version of ISE, one must learn how to use the VHDL Test Bench program. The test bench code generated by the VHDL Test Bench program is portable—that is, it can be used with the ISE compiler and also with other compilers. The Test Bench Waveform program that was removed could only be used with the ISE compiler, which was probably the reason it was removed.

The following examples provide you with enough information to obtain simulations. The simulation waveforms that are shown in the following examples were obtained using ISE Version 12 or later. The output simulation waveforms for different versions of ISE may look slightly different. In ISE Versions 11 and 12, portions of the waveforms that represent a 1 are shaded, as you will see later when the waveforms are shown. This improvement makes it a little easier to distinguish between 1s and 0s in the simulation waveforms.

A design project can be updated from older ISE versions (Versions 9 or 10), which have the extension .ise, to a newer ISE version (Version 11 or newer), which have the extension .xise. First, open the ISE Project Navigator for the newer version of ISE. Second, click on the down arrow at the bottom right-hand corner of the screen to change from ISE Project Files (*.xise) to Old ISE Project files (*.ise). Third, click on the old ISE Project file, then click "Migrate Only" to convert the older .ise file to the newer .xise file.

B.2 EXAMPLE 1—COMBINATIONAL LOGIC DESIGN (PROJECT: AND_3)

Important Note: For other combinational logic designs, follow the same procedure presented in this section.

Listing B.1 shows a complete VHDL design for a three input AND gate—that is, this is the first design in Experiment 1A in Appendix A.

LISTING B.1
Complete VHDL design for a 3-input AND gate (project: AND_3)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity AND_3 is port (
     A,B,C : in STD_LOGIC;
     F : out STD_LOGIC
     );
end AND_3;

architecture Boolean_function of AND_3 is
begin
     F <= A and B and C;
end Boolean_function;
```

Follow the procedure in Experiment 1A, in Section 2, to enter this VHDL design into the Xilinx ISE Project Navigator if you have not already done so. Double-click Synthesize—XST, to see if there are any errors. If there are any errors, you must correct them before you continue.

To simulate this design, click Project on the menu bar, then under the drop-down window click New Source. When the New Source Wizard window appears, select (click) VHDL Test Bench in the list, and type in a file name. We used the name sim, which is short for simulation. Be sure to choose a file name that starts with a letter and has no spaces in it. Click Next, Next, Finish.

Your screen should now show the test bench code for AND_3 that is shown in Template B.1:

TEMPLATE B.1 Test bench code for the 3-input AND gate design in Listing B.1 (project: AND_3)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

--Uncomment the following library declaration if using
--arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY sim IS
END sim;

ARCHITECTURE behavior OF sim IS

     --Component Declaration for the Unit Under Test (UUT)

COMPONENT AND_3
PORT(
     A : IN std_logic;
     B : IN std_logic;
     C : IN std_logic;
     F : OUT std_logic
     );
END COMPONENT;

--Inputs
signal A : std_logic := '0';
signal B : std_logic := '0';
signal C : std_logic := '0';
```

```
--Outputs
signal F : std_logic;

--No clocks detected in port list. Replace <clock> below with
--appropriate port name

constant <clock>_period : time := 10 ns;
BEGIN

     --Instantiate the Unit Under Test (UUT)
uut: AND_3 PORT MAP (
     A => A,
     B => B,
     C => C,
     F => F
     );

--Clock process definitions
<clock>_process :process
begin
     <clock> <= '0';
     wait for <clock>_period/2;
     <clock> <= '1';
     wait for <clock>_period/2;
end process;

--Stimulus process
stim_proc: process
begin
     --hold reset state for 100 ns.
     wait for 100 ns;

     wait for <clock>_period*10;

     --insert stimulus here

     wait;
end process;

END;
```

The test bench code in Template B.1 was generated by WebPACK ISE Version 12. Because this is a combinational logic design, there is no clock. Observe the comment "No clocks detected in port list." Comment out or remove the section of the test bench code that contains the name clock, including the clock process following the comment "Clock process definitions."

Replace the stimulus process with the code shown in Stimulus Process B.1, then click Save to save the test bench code.

After you add the stimulus process to the template, you have created fully functional test bench code for the 3-input AND gate design in Listing B.1.

Test bench code is simply a program that supplies the information necessary to run a simulation. Because we are using VHDL, the test bench code is patterned after VHDL and supplies

STIMULUS PROCESS B.1 Code for the stimulus inputs for the 3-input AND gate design in Listing B.1 (project: AND_3)

```
--Stimulus Process
process
begin
     --stimulus inputs
     A <= '0'; B <= '0'; C <= '0';     wait for 100 ns;
     A <= '0'; B <= '0'; C <= '1';     wait for 100 ns;
     A <= '0'; B <= '1'; C <= '0';     wait for 100 ns;
     A <= '0'; B <= '1'; C <= '1';     wait for 100 ns;
     A <= '1'; B <= '0'; C <= '0';     wait for 100 ns;
     A <= '1'; B <= '0'; C <= '1';     wait for 100 ns;
     A <= '1'; B <= '1'; C <= '0';     wait for 100 ns;
     A <= '1'; B <= '1'; C <= '1';     wait for 100 ns;
     A <= '0'; B <= '0'; C <= '0';     wait;
end process;
```

the stimulus inputs necessary to run a simulation for a VHDL design. When you first begin to use test bench code, it is helpful to analyze the code to see how it is written. As you see more test bench code, you will begin to understand how to write your own code. It is easy to use the VHDL Test Bench template and modify it to provide your final test bench code for a combinational logic design, because you only need to learn how to remove the clock signals and supply the stimulus process, which contains the stimulus inputs for your design. Notice that the stimulus inputs show all the possible combinations of the input signals *A*, *B*, and *C*.

Things you should notice about the template for the test bench code for Template B.1 that is generated for your design:

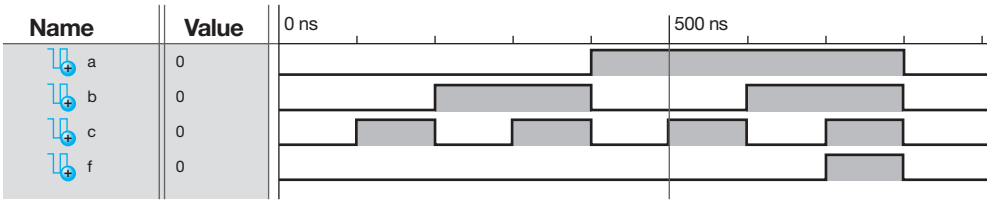
- The library part has a comment concerning arithmetic functions. Uncomment the USE clause if you are using an arithmetic function in your design.
- The entity declaration is empty because it requires no inputs or outputs.
- In the architecture declaration, the design is declared as a component with the same inputs and outputs as specified in the unit under test (UUT)—that is, AND_3 for this example.
- All the input port signals and the output port signals in the entity of the UUT are declared as internal signals. All the internal signals are initialized to 0 via := '0'.
- The UUT is instantiated after the first **begin** in the **architecture**.
- The stimulus process cannot contain a sensitivity list because a wait statement is used in the process.
- The stimulus inputs are placed after **begin** in the stimulus process. The stimulus inputs are applied for the period of time given by the evaluation of the time expression in each wait statement when the simulate behavioral model is run. The last wait statement stops the simulation.

Don't get discouraged when you obtain a simulation for a design. Simply use the template and add the stimulus process. If you change the stimulus inputs, you can see how this affects the simulation result—that is, via trial and error. Simply change the stimulus inputs, save the sim file, and rerun the simulation. To obtain the simulation waveform, click Simulation in the Sources window of the ISE Project Navigator to turn on the radio button beside Simulation. Next, click the VHDL Test Bench file sim.vhd to highlight it. If you elected to use a different file name for the simulation file, then click your elected file name to highlight it. Do not click the uut - AND_3 file to highlight it, because this file is not the simulation file.

Open up the ISim Simulator in the Processes window, by clicking the plus sign. In the Processes window in the ISE Project Navigator, double-click Behavioral Check Syntax or right-click on Behavioral Check Syntax, which opens up a drop-down window, then click run. This checks to see if there are any errors in the syntax for the test bench code. Click on Console in the Console window at the bottom of the ISE Project Navigator to observe if the process "Behavioral Check Syntax" was successfully completed. If the process was not successfully completed, you must find and fix your error or errors.

If the syntax for the test bench code is correct, double-click Simulate Behavioral Model or right-click Simulate Behavioral Model, which opens up a drop-down window, then click run. This starts the ISE simulator. After the simulation waveforms are generated and appears on the screen, click on the "Zoom to Full View" icon.

The simulation result that is provided in the ISim window is shown in Waveform B.1.



WAVEFORM B.1
Simulation result

The scale for Waveform B.1 is shown above the waveforms. Observe that ns stands for nanoseconds, which is 10^{-9} seconds. Notice that all values of 1 in Waveform B.1 are highlighted.

When the simulation is run, the stimulus inputs for the signals *A*, *B*, and *C* are applied to the VHDL code for the design and the output signal *F* is generated. Because the output signal *F* in Waveform B.1 follows the truth table for a 3-input AND gate, we know that the VHDL code for the design is correct.

If the output signal *F* did not follow the truth table for a 3-input AND gate, then we would know that the VHDL code for the design has an error. Entering the wrong assignment statement for the Boolean function will result in an error. For example, if you would enter the assignment statement for the Boolean function of the 3-input AND gate as $F <= A$ **and** B **or** C or as $F <= A$ **nand** B **and** C, then you would observe that the output signal *F* does not follow the truth table for a 3-input AND gate, so the VHDL design would have an error.

If you were writing your own test bench code, your test bench code could also have an error. If you have a problem obtaining a simulation, look closely at both your VHDL design and the test bench code that you entered for the design to find and correct any errors that may exist.

B.3 EXAMPLE 2—SYNCHRONOUS SEQUENTIAL LOGIC DESIGN (PROJECT: DFF)

Important Note: For other synchronous sequential logic designs, follow the same procedure presented in this section.

Listing B.2 shows a complete VHDL design for a DFF, using a dataflow design style with a conditional signal assignment (CSA).

LISTING B.2
Complete VHDL design for a DFF (project: DFF)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DFF is port (
    d, clk, rst: in std_logic;
```

(Continued)

```
    q: inout std_logic
    );
end DFF;
architecture dataflow of DFF is
begin
    q <= '0' when rst = '1' else
         d when rising_edge (clk) else
         q; --note: "else q" is inferred (so it can be removed)
end dataflow;
```

Enter this VHDL design into the Xilinx ISE Project Navigator, and then double-click Synthesize—XST, to see if there are any errors. If there are any errors, you must correct them before you continue.

To simulate this design, click Project on the menu bar, then under the drop-down window click New Source. When the New Source Wizard window appears, select (click) VHDL Test Bench in the list, and type in a file name. We use the name sim, which is short for simulation. Be sure to choose a file name that starts with a letter and has no spaces in it. Click Next, Next, Finish.

Your screen should now show the test bench code for AND_3 that is shown in Template B.2:

TEMPLATE B.2 Test bench code for the DFF design in Listing B.2 (project: DFF)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

--Uncomment the following library declaration if using
--arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY sim IS
END sim;

ARCHITECTURE behavior OF sim IS
    --Component Declaration for the Unit Under Test (UUT)

    COMPONENT DFF
    PORT(
        d : IN std_logic;
        clk : IN std_logic;
        rst : IN std_logic;
        q : INOUT std_logic
        );
    END COMPONENT;

    --Inputs
    signal d : std_logic := '0';
    signal clk : std_logic := '0';
    signal rst : std_logic := '0';

    --BiDirs
    signal q : std_logic;
```

```
    --Clock period definitions
    constant clk_period : time := 10 ns;

BEGIN

    --Instantiate the Unit Under Test (UUT)
    uut: DFF PORT MAP (
         d => d,
         clk => clk,
         rst => rst,
         q => q
         );

    --Clock process definitions
    clk_process :process
    begin
         clk <= '0';
         wait for clk_period/2;
         clk <= '1';
         wait for clk_period/2;
end process;

--Stimulus process
stim_proc: process
begin
    --hold reset state for 100 ns.
    wait for 100 ns;

    wait for clk_period*10;

    --insert stimulus here

    wait;
end process;

END;
```

The test bench code in Template B.2 was generated by WebPACK ISE Version 12. Because this is a synchronous sequential logic design, there is a clock. Following the comment "Clock period definitions," change "time := 10 ns" to "time := 50 ns" because we want to display the clock period as 50 ns (20 MHz), rather than the clock period as10 ns (100 MHz).

Replace the stimulus process with the code shown in Stimulus Process B.2, then click Save to save the test bench code.

```
--Stimulus Process (Make time := 50 ns)
process
begin
    --stimulus inputs
    rst <= '1'; wait for 50 ns;
```

(Continued)

STIMULUS PROCESS B.2 Code for the stimulus inputs for the DFF design in Listing B.2 (project: DFF)

```
    rst <= '0'; wait for 50 ns;
    d <= '1';   wait for 100 ns;
    d <= '0';   wait for 400 ns;
    d <= '1';   wait for 50 ns;
    rst <= '1'; wait;
end process;
```

After you add the stimulus process to the template, you have created fully functional test bench code for the DFF design in Listing B.2.

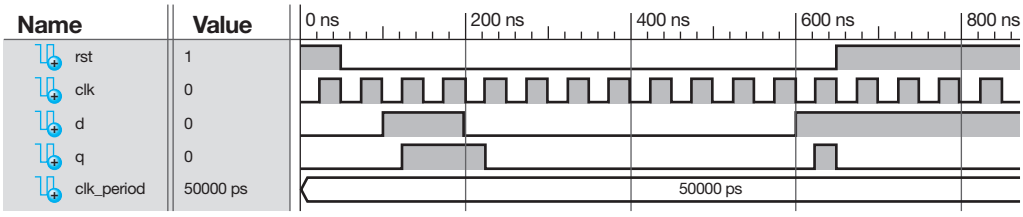
To obtain the simulation waveform, click Simulation in the Sources window of the ISE Project Navigator to turn on the radio button beside Simulation. Next, click the VHDL Test Bench file sim.vhd to highlight it. If you elected to use a different file name for the simulation file, then click your elected file name to highlight it. Do not click the uut - DFF file to highlight it because this file is not the simulation file.

Open up the ISim Simulator in the Processes window, by clicking the plus sign. In the Processes window in the ISE Project Navigator, double-click Behavioral Check Syntax or right-click on Behavioral Check Syntax, which opens up a drop-down window, then click run. This checks to see if there are any errors in the syntax for the test bench code. Click on Console in the Console window at the bottom of the ISE Project Navigator to observe if the process "Behavioral Check Syntax" was successfully completed. If the process was not successfully completed, you must find and fix your error or errors.

If the syntax for the test bench code is correct, double-click Simulate Behavioral Model or right-click Simulate Behavioral Model, which opens up a drop-down window, then click run. This starts the simulator. After the simulation waveforms are generated and appears on the screen, click on the "Zoom to Full View" icon.

The simulation result for the DFF design is displayed in the ISim window as shown in Waveform B.2.

WAVEFORM B.2
Simulation result for DFF



Observe that the simulation result in Waveform B.2 follows the characteristic table for the positive edge triggered DFF with a *CLR* input, which is the complete VHDL DFF design in Listing B.2. If the output signal *Q* did not follow the characteristic table, then we would know that the design contains an error that must be found and fixed. If you have an error in your design, you must find the error or errors and fix them and rerun the Simulate Behavioral Model to verify the fix or fixes.

Appendix C

FPGA Pin Connections—Handy Reference

C.1 BASYS 2 BOARD

FPGA pin connections for the BASYS 2 board: CLK is 25, 50, or 100 MHz (use FPGA Pin B8)
Family: Spartan3E Device: XC3S100E Package: CP132
or Device: XC3S250E Package: CP132

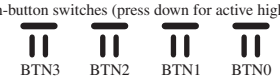
Slide switch	FPGA pin
SW0	P11
SW1	L3
SW2	K3
SW3	B4
SW4	G3
SW5	F3
SW6	E2
SW7	N3

Push-button switch	FPGA pin
BTN0	G12
BTN1	C11
BTN2	M4
BTN3	A7

Single LED	FPGA pin
LD0	M5
LD1	M11
LD2	P7
LD3	P6
LD4	N5
LD5	N4
LD6	P4
LD7	G1

Slide switches (push forward = high output and pull back = low output)

Push-button switches (press down for active high output)



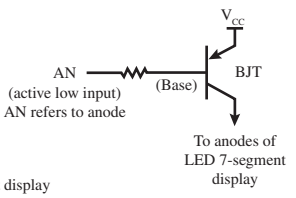
Single LEDs with active high inputs

DISP1

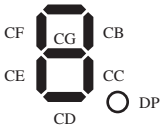
LED cathode	FPGA pin
CA	L14
CB	H12
CC	N14
CD	N11
CE	P12
CF	L13
CG	M12
DP	N13

BJT base	FPGA pin
AN0	F12
AN1	J12
AN2	M13
AN3	K14

AN0 is right-most 7-segment display
AN3 is left-most 7-segment display



Common Anode



Segments CA through CG and DP are active low inputs

CA refers to cathode for segment A,
CB refers to cathode for segment B, etc.,
DP refers to cathode for segment DP.

Peripheral connectors

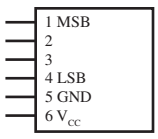
Pmod JA I/O	FPGA pin
JA-4(LSB)	B5
JA-3	J3
JA-2	A3
JA-1(MSB)	B2

Pmod JB I/O	FPGA pin
JB-4(LSB)	B7
JB-3	C5
JB-2	B6
JB-1(MSB)	C6

Pmod JC I/O	FPGA pin
JC-4(LSB)	C9
JC-3	A10
JC-2	B9
JC-1(MSB)	A9

Pmod JD I/O	FPGA pin
JD-4(LSB)	D12
JD-3	C13
JD-2	A13
JD-1(MSB)	C12

Connector circuit



Plug Pmod in here

C.2 NEXYS 2 BOARD

FPGA pin connections for the NEXYS 2 board: CLK is 50 MHz (use FPGA Pin B8)
Family: Spartan3E Device: XC3S500E Package: FG320

Slide switch	FPGA pin
SW0	G18
SW1	H18
SW2	K18
SW3	K17
SW4	L14
SW5	L13
SW6	N17
SW7	R17

Push-button switch	FPGA pin
BTN0	B18
BTN1	D18
BTN2	E18
BTN3	H13

Single LED	FPGA pin
LD0	J14
LD1	J15
LD2	K15
LD3	K14
LD4	E17
LD5	P15
LD6	F4
LD7	R4

Slide switches (push forward = high output and pull back = low output)

Push-button switches (press down for high output)

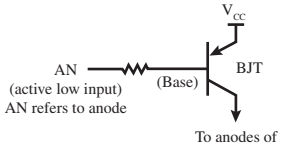
Single LEDs with active high inputs

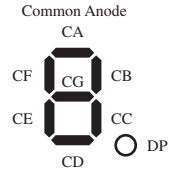
DISP1

LED cathode	FPGA pin
CA	L18
CB	F18
CC	D17
CD	D16
CE	G14
CF	J17
CG	H14
DP	C17

BJT base	FPGA pin
AN0	F17
AN1	H17
AN2	C18
AN3	F15

AN0 is right-most 7-segment display
AN3 is left-most 7-segment display





Segments CA through CG and DP are active low inputs

CA refers to cathode for segment A,
CB refers to cathode for segment B, etc.,
DP refers to cathode for segment DP.

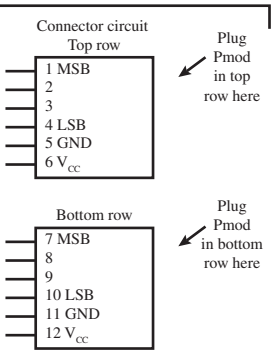
Peripheral connectors

Pmod JA I/O	FPGA pin
Top row	
JA-4(LSB)	M15
JA-3	L17
JA-2	K12
JA-1(MSB)	L15
Bottom row	
JA-10(LSB)	M16
JA-9	M14
JA-8	L16
JA-7(MSB)	K13

Pmod JB I/O	FPGA pin
Top row	
JB-4(LSB)	T17
JB-3	R15
JB-2	R18
JB-1(MSB)	M13
Bottom row	
JB-10(LSB)	U18
JB-9	T18
JB-8	R16
JB-7(MSB)	P17

Pmod JC I/O	FPGA pin
Top row	
JC-4(LSB)	H16
JC-3	G13
JC-2	J16
JC-1(MSB)	G15
Bottom row	
JC-10(LSB)	J12
JC-9	G16
JC-8	F14
JC-7(MSB)	H15

Pmod JD I/O	FPGA pin
Top row	
JD-4(LSB)	P18
JD-3	N18
JD-2	M18
JD-1(MSB)	J13



Note: To provide voltage to pin V_{CC} you must use a power supply jumper between the two headers closest to the middle of the Nexsys 2 board for each Pmod connector.

C.3 MEMORY LOADER I/O PIN CONNECTIONS FOR THE FPGAS ON THE BASYS 2 AND NEXYS 2 BOARD

Signal	BASYS 2 Board Spartan3E XC3S100E-CP132 or XC3S250E-CP132	NEXYS 2 Board Spartan3E XC3S500E-FG320
A_USB_ASTB	F2	V14
A_USB_DSTB	F1	U14
USB_WAIT_S	D2	N9
A_USB_WRITE	C2	V16
USB_DATA 0	N2	R14
USB_DATA 1	M2	R13
USB_DATA 2	M1	P13
USB_DATA 3	L1	T12
USB_DATA 4	L2	N11
USB_DATA 5	H2	R11
USB_DATA 6	H1	P10
USB_DATA 7	H3	R10

C.4 FX2 MIB (MODULE INTERFACE BOARD)—ADD-ON BOARD FOR NEXYS 2

Peripheral connectors

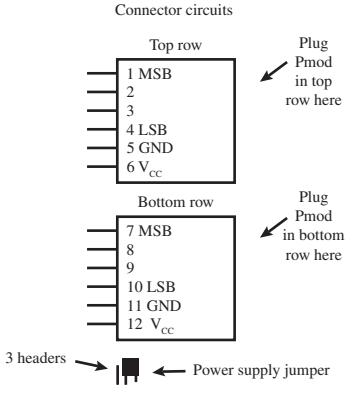
Top of FX2 MIB

Pmod J1

100-pin connector	I/O	FPGA pin
Top row		
FX2-1O7	J1-4(LSB)	C5
FX2-1O5	J1-3	B6
FX2-1O3	J1-2	C3
FX2-1O1	J1-1(MSB)	B4
Bottom row		
FX2-1O8	J1-10(LSB)	F7
FX2-1O6	J1-9	D5
FX2-1O4	J1-8	C4
FX2-1O2	J1-7(MSB)	A4

Pmod J2

100-pin connector	I/O	FPGA pin
Top row		
FX2-1O15	J2-4(LSB)	E9
FX2-1O13	J2-3	D7
FX2-1O11	J2-2	C7
FX2-1O9	J2-1(MSB)	E7
Bottom row		
FX2-1O16	J2-10(LSB)	C9
FX2-1O14	J2-9	E8
FX2-1O12	J2-8	F8
FX2-1O10	J2-7(MSB)	A6



Note: To provide voltage to pin V_{CC} you must use a power supply jumper between the two headers closest to the edge of the FX2 MIB for each Pmod connector.

Pmod J3

100-pin connector	I/O	FPGA pin
Top row		
FX2-1O23	J3-4(LSB)	A11
FX2-1O21	J3-3	A10
FX2-1O19	J3-2	F9
FX2-1O17	J3-1(MSB)	A8
Bottom row		
FX2-1O24	J3-10(LSB)	D11
FX2-1O22	J3-9	B10
FX2-1O20	J3-8	D10
FX2-1O18	J3-7(MSB)	G9

Pmod J4

	I/O	FPGA pin
Top row		
FX2-1O31	J4-4(LSB)	F12
FX2-1O29	J4-3	F11
FX2-1O27	J4-2	C11
FX2-1O25	J4-1(MSB)	E10
Bottom row		
FX2-1O32	J4-10(LSB)	A13
FX2-1O30	J4-9	E12
FX2-1O28	J4-8	E11
FX2-1O26	J4-7(MSB)	B11

Pmod J5

	I/O	FPGA pin
FX2-1O36	J5-4(LSB)	C14
FX2-1O35	J5-3	A14
FX2-1O34	J5-2	E13
FX2-1O33	J5-1(MSB)	B13

Pmod J6

	I/O	FPGA pin
FX2-1O40	J6-4(LSB)	B16
FX2-1O39	J6-3	A16
FX2-1O38	J6-2	B14
FX2-1O37	J6-1(MSB)	D14

Right side of FX2 MIB

Appendix D

EASY1 Tutorial

D.1 INTRODUCTION

EASY1 is a simple editor, assembler, and simulator to test assembly language programs written for VBC1. EASY1 can be used to generate the machine code for each program written for VBC1. EASY1 can also be used to simulate a program for VBC1. The simulation of a program for VBC1 using EASY1 and the execution of the same program via the hardware design for VBC1 should be the same. The simulator allows the user to either single step a program or to run a program at various speeds that are selected by the user. The programmer's register model can be observed as each instruction is executed by single stepping the program. This allows a programmer to observe if a program is performing correctly.

D.2 EASY1 SCREEN OR GUI

The editor/assembler/simulator 1 (EAS1) for VBC1, which we will refer to as EASY1, is used for writing an assembly language program, assembling the program, and simulating the program via a programmer's register model displayed on the screen. A form of the programmer's register model for VBC1 was presented in Chapter 10, Section 10.7, and provided the essential background for this tutorial.

From this textbook's website, download the program EASY1 on your computer. Figure D.1 shows the screen or graphical user interface (GUI) for EASY1. EASY1 is the program that runs when the EASY1-E check box is unchecked. EASY1-E is the program that runs when the EASY1-E check box is checked. EASY1-E is the editor/assembler/simulator 1-E for VBC1-E, which is the enhanced version of VBC1.

D.3 EASY1 LAYOUT

EASY1 is divided into three parts: The assembly part on the left side is where you enter or write an assembly language program. The assembly part is designed as a simple editor. The programmer's register model in the middle is used to run your program—that is, simulate your program after you assemble your program by pressing Assemble and Load. The instruction memory part on the right side is where the machine code for your program is placed by the assembler. EASY1 performs the following three functions: (1) provides an editor for writing and correcting your program, (2) assembles your program to provide the machine code necessary for program execution, and (3) simulates your program to verify that it performs as you expect it to perform.

If there are syntax errors in your program, you are notified by a pop-up window when you click the Assemble and Load button with the mouse. If there are no syntax errors, the program is assembled and the machine code is loaded in the instruction memory. You have the option

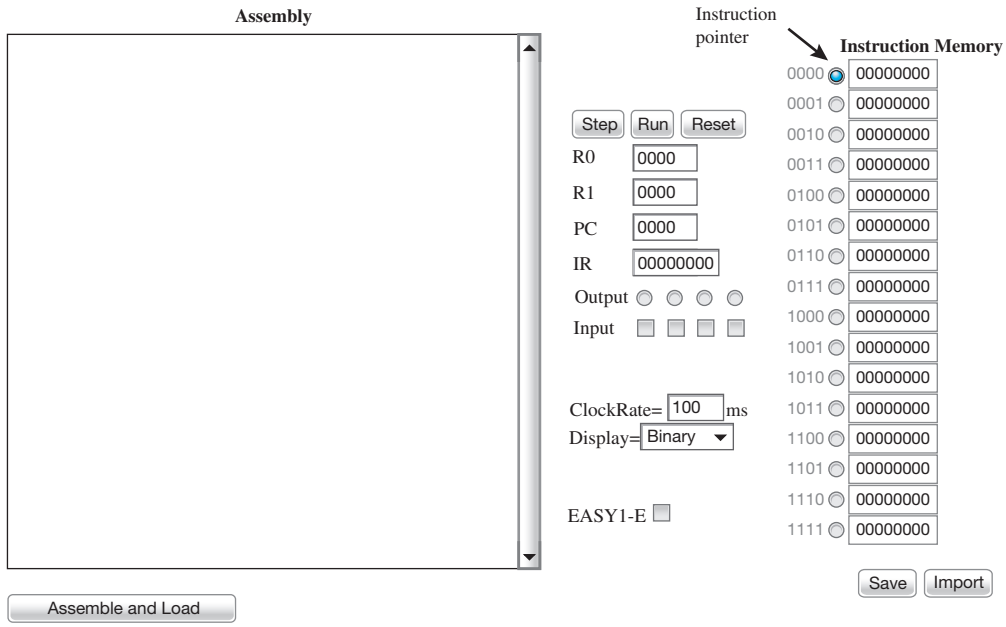


FIGURE D.1 The screen (or GUI) for EASY1

of stepping through the program one line at a time, running the program at the 100-ms default clock rate (you can change this rate to run the program faster or slower), or resetting the program when you click the Step, Run, or Reset buttons, respectively. R0 displays the contents of register zero for bits (3:0), R1 displays the contents of register one for bits (3:0), PC displays the contents of the program counter for bits (3:0), and IR displays the contents of the instruction register for bits (7:0). All four registers must be read from left to right, because the MSB is placed on the left and the LSB is placed on the right.

An output is processed in the programmer's register model by an instruction called OUT that is placed in the program. A clear circle represents a logic 0 output, and a circle with a dot in it represents a logic 1 output. In your mind, picture the four circles for the output as LED (3:0) or light emitting diodes, from 3 down to 0, read from the left to the right.

An input is processed in the programmer's register model by an instruction called IN that is placed in the program. A clear square represents a logic 0 input, and a checked square (click the mouse on a square to enter and/or remove a check mark) represents a logic 1 input. In your mind, picture the four squares for the input as SW (3:0) or slide switches, 3 down to 0, read from the left to the right.

Program assembly—that is, converting the assembly language program into machine code—is automatically performed by the assembler. If there are no errors in your assembly language program, the machine code for the program is loaded into the instruction memory when you click the Assemble and Load button with the mouse. To the left of the instruction memory is a column of numbers ranging from 0000 to 1111 in binary. This column represents all the available memory addresses for the instruction memory of VBC1. You can change this column to express the memory addresses in decimal or in hexadecimal via the selection arrow following Display. In the column between the memory addresses and the instruction memory is a set of vertical circles. The circle with a dot indicates the current value of the program counter (PC) and the current value in the IR. The instruction memory is initially loaded with all 0s. The first column to the right of the instruction memory is reserved for the mnemonic field and the second and third columns are reserved for the operands field for each assembly language instruction. These fields are filled with the instructions after you write a program and click the Assemble and Load button. Each instruction is placed next to its machine code as a handy reference.

The two buttons labeled Save and Import, located just below the instruction memory, are handy buttons to use when writing assembly language programs. The Save button allows you to save the contents of the instruction memory in a file. The Import button allow you to transfer any file that has been previously saved via the Save button back into the EASY1 editor. These two buttons allow you to save a program you are working on, but that you may or may not have finished, and then import the assembly language program back into the EASY1 editor. The Save button and the Import button only work on a locally saved version of EASY1—that is, on EASY1 that is running from your computer and not over a network.

D.4 HOW TO USE EASY1

The best way to use EASY1 is to simply write some assembly language programs and run them using the simulator via the Step button or the Run button. You will quickly and easily gain a complete understanding of EASY1 and how it works using this approach. Figure D.2 is a slightly modified block diagram of VBC1. This figure was presented earlier in Chapter 10, Section 10.5, Figure 10.4. The figure has been modified to include the squares for the 4 slide switches at the input and the circles for the 4 LEDs at the output.

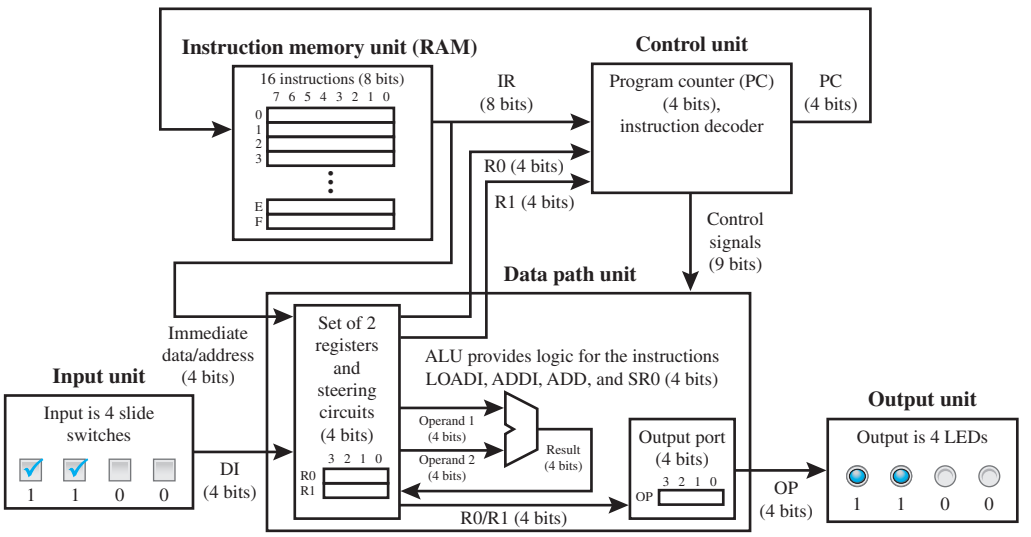


FIGURE D.2 Block diagram for VBC1

VBC1 has eight different instructions with 22 variations. Figure D.3 shows an alphabetical listing of the instruction set and the aliases supported by EASY1 for VBC1. The editor/assembler/simulator (EASY1) automatically generates the machine code for each instruction, thus relieving you of performing that task.

Note: The operands Data and Address in the instructions and aliases can be expressed in decimal, binary, or hexadecimal—for example, 10 or 10d in decimal, 1010b in binary, or ah in hexadecimal.

D.5 EXAMPLE 1—A SIMPLE INPUT/OUTPUT PROGRAM

Write an assembly language program to perform the following tasks: (1) input the decimal number 12 into register R0, via the input that is represented by the squares (remember to think of these as slide switches); (2) move the decimal number 12 to register R1; and (3) output the

```
Instruction     Comment

ADD R0,R0
ADD R0,R1
ADD R1,R0
ADD R1,R1

ADDI R0,Data    ; where Data is a decimal value 0 through 15
ADDI R1,Data    ; where Data is a decimal value 0 through 15

IN R0           ; from input switches
IN R1           ; from input switches

JNZ R0,Address  ; where Address is a decimal value 0 through 15
JNZ R1,Address  ; where Address is a decimal value 0 through 15

LOADI R0,Data   ; where Data is a decimal value 0 through 15
LOADI R1,Data   ; where Data is a decimal value 0 through 15

MOV R0,R0
MOV R0,R1
MOV R1,R0
MOV R1,R1

OUT R0          ; to output LEDs
OUT R1          ; to output LEDs

SR0 R0,R0
SR0 R0,R1
SR0 R1,R0
SR0 R1,R1

Aliases         Comment

DEC R0          ; ADDI R0,15
DEC R1          ; ADDI R1,15

INC R0          ; ADDI R0,1
INC R1          ; ADDI R1,1

NOP             ; MOV R0,R0

SUBI R0,0       ; ADDI R0,0
SUBI R1,0       ; ADDI R1,0

SUBI R0,Data    ; ADDI R0, (16 – Data), where Data is a decimal value 1 though 15
SUBI R1,Data    ; ADDI R1, (16 – Data), where Data is a decimal value 1 though 15
```

FIGURE D.3 Instruction set and the aliases supported by EASY1 for VBC1 listed alphabetically

content of register R1 to the output that is represented by the circles (remember to think of these as LEDs). Set the input switches to 12 or 1100b prior to single stepping or running the program as shown in the block diagram in Figure D.2. After single stepping or running the program, compare your results with the output shown in the block diagram in Figure D.2.

Program D.1 shows an assembly language program that performs the input/output tasks for Example 1.

PROGRAM D.1
Assembly language program for Example 1

```
; Program D.1, a simple input/output program
; first read the input switch values
; then output the switch values to the LEDs
IN R0        ; input 12 to R0 from switches
MOV R1,R0    ; move R0 to R1
OUT R1       ; output switch values to LEDs
```

Comments are shown in Program D.1 to document the algorithm—that is, the step-by-step procedure. It is good programming practice to use comments for remembering how the program works at a later date or for someone else who is trying to understand your program for the first time.

Figure D.4 shows the program in Program D.1 entered in the editor of EASY1. The first assembly language instruction in line 4 of the program has an intentional error. The error is Ro when it should be R0. The letter o is mistakenly typed in for the number 0. This is a common mistake you need to avoid.

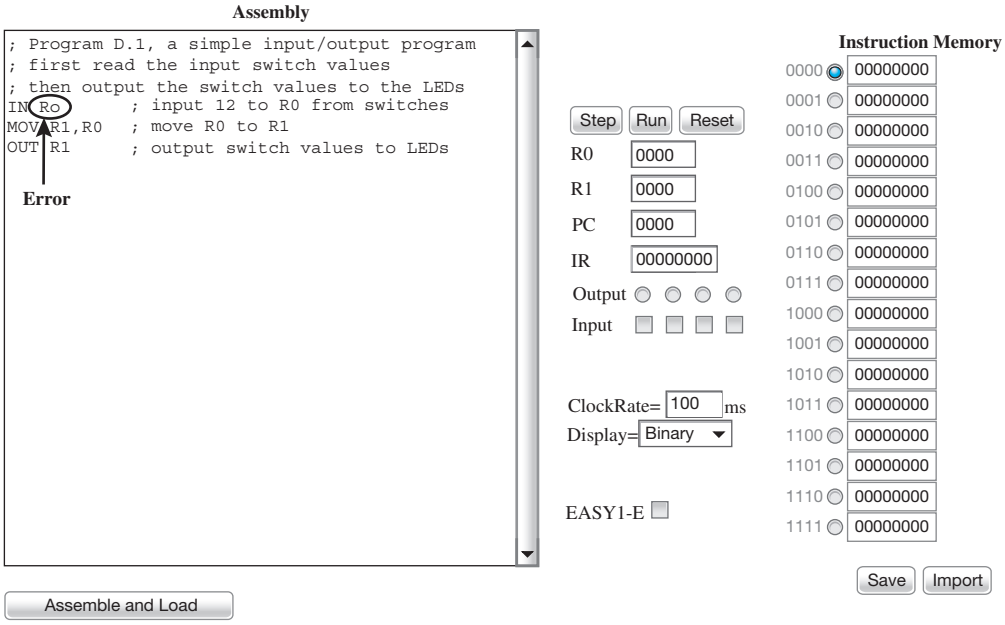


FIGURE D.4 EASY1 screen after entering Program D.1 with an error

To assemble the program and load the machine code into instruction memory, click the **Assemble and Load** button. After clicking the **Assemble and Load** button, an error message pops up in the EASY1 screen as shown in Figure D.5, indicating a syntax error in line 4 of the program. The instruction memory is not loaded until all syntax errors in the program are identified and corrected.

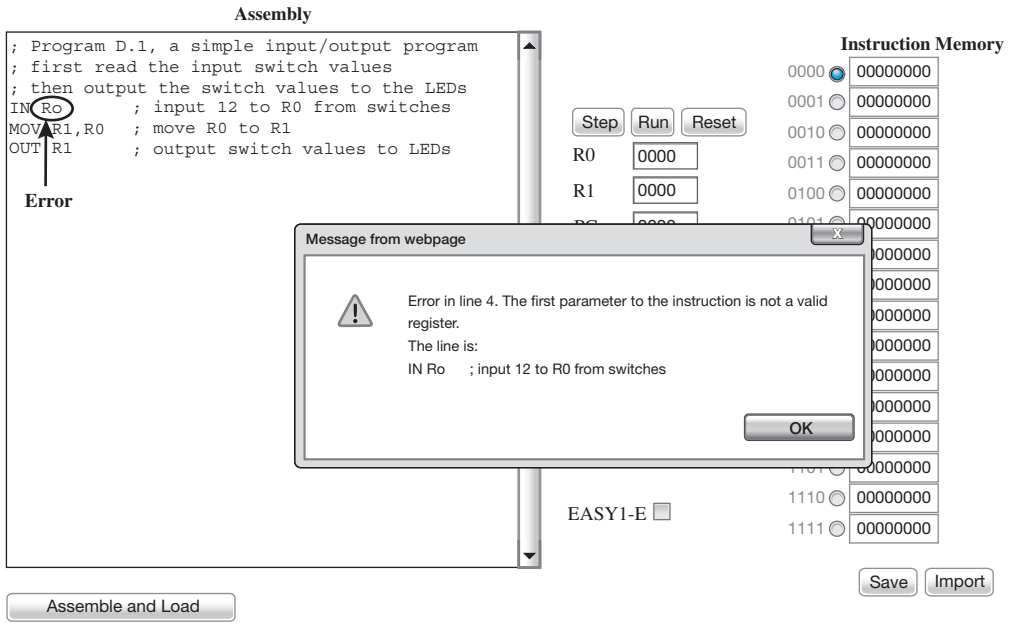


FIGURE D.5 EASY1 screen after clicking Assemble and Load without removing the error

Notice that the first three lines in the assembly language program are comments, the fourth line is the first instruction in the program, and the pop-up window says that this line has an error. The error in the assembly language program is corrected by clicking the OK button in the pop-up window and changing Ro (R followed by the letter o) to R0 (R followed by the number 0). Figure D.6 shows the EASY1 screen, after correcting the error and clicking the Assemble and Load button.

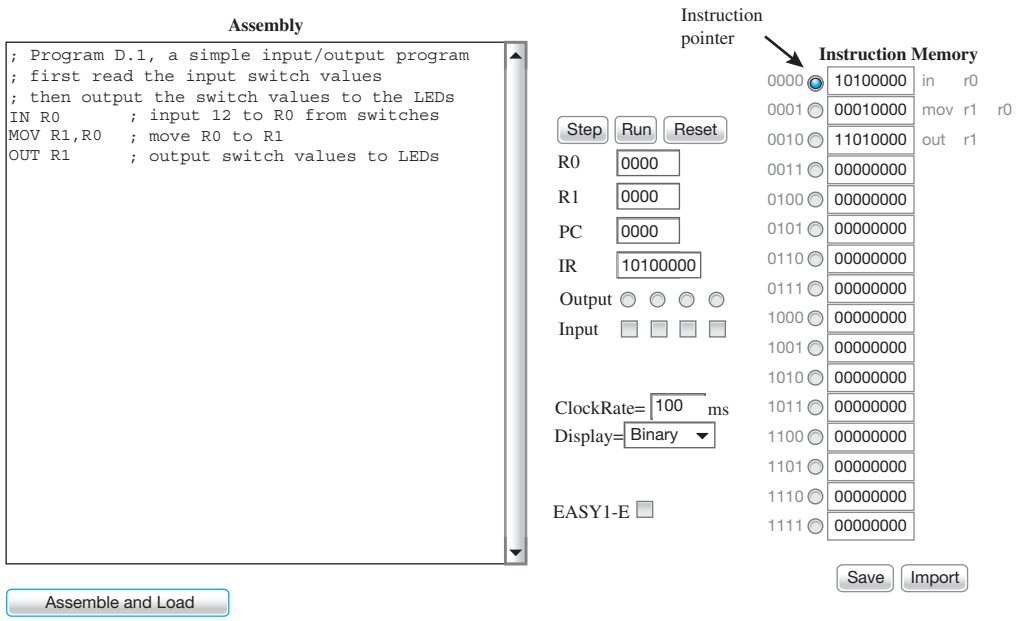


FIGURE D.6 EASY1 screen after correcting the error and clicking Assemble and Load for Program D.1

Notice in Figure D.6 that the assembly language program has been assembled (the assembly language program has been converted to machine code), and the machine code has been loaded into the instruction memory. The mnemonic field and the operands field in the assembly language program are shown beside the machine code in the instruction memory, as reference.

Notice that the first instruction is at address 0 (as indicated by the instruction pointer), while the second instruction is at address 1—that is, the n'th instruction is at address n − 1

because 0 is the reference for numbering the addresses. If 1 were the reference for numbering the addresses, then the address 0 would be wasted.

Prior to running the program, supply an input of decimal 12 on the input slide switches as shown in Figure D.7 by the check marks placed in the input boxes.

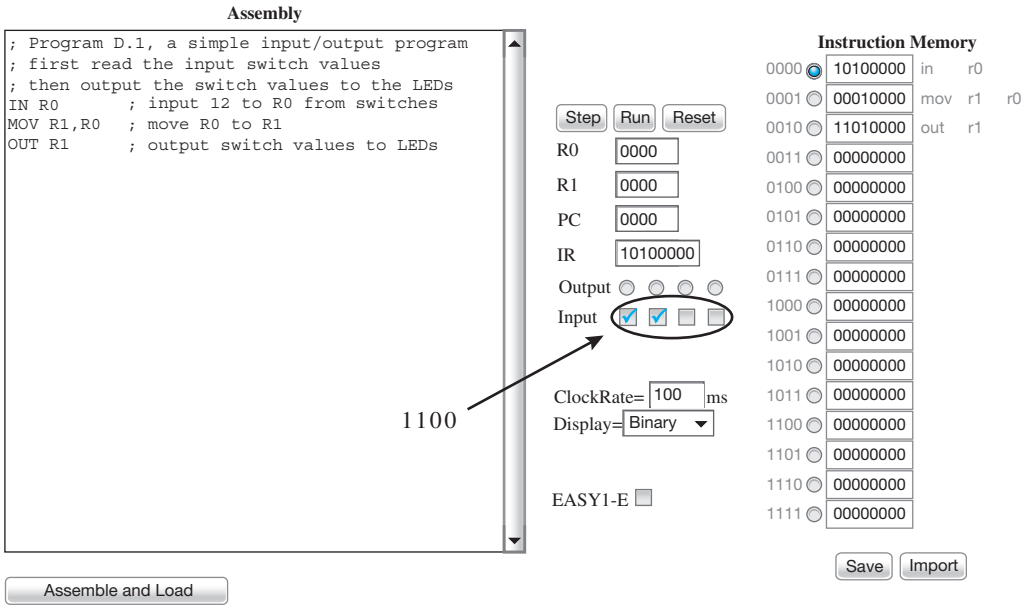


FIGURE D.7 EASY1 screen after inputting decimal 12 on the slide switches for Program D.1

To single step through the assembly language program to see the result of each instruction after it is executed, simply click the Step button once for each instruction. Figure D.8 shows the EASY1 screen after clicking the Step button one time to execute the first instruction which is at address 0.

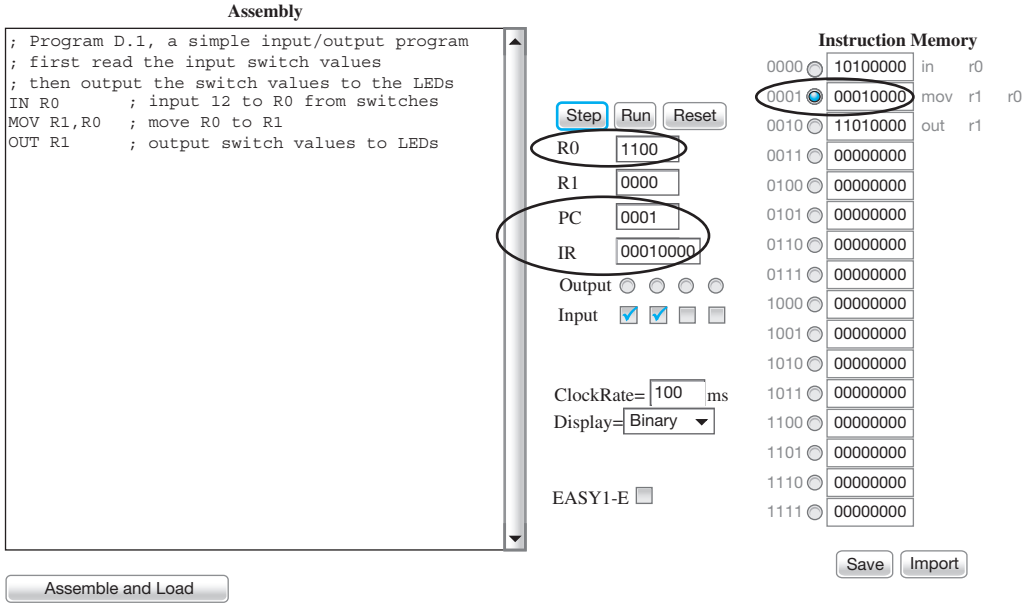


FIGURE D.8 EASY1 screen after clicking the Step button one time to execute the first instruction for Program D.1

Observe in Figure D.8 that R0 contains 12, as it should after executing the instruction IN R0 because the input slide switches were initially set to 12 prior to executing the instruction. Notice that the PC now points to (or shows) the address of the next instruction or the second instruction,

which is at address 1 as indicated by the instruction pointer. Also observe in Figure D.8 that the IR contains the machine code for the next instruction that will be executed next, which is the second instruction at address 1.

Figure D.9 shows the EASY1 screen after clicking the Step button a second time to execute the second instruction at address 1.

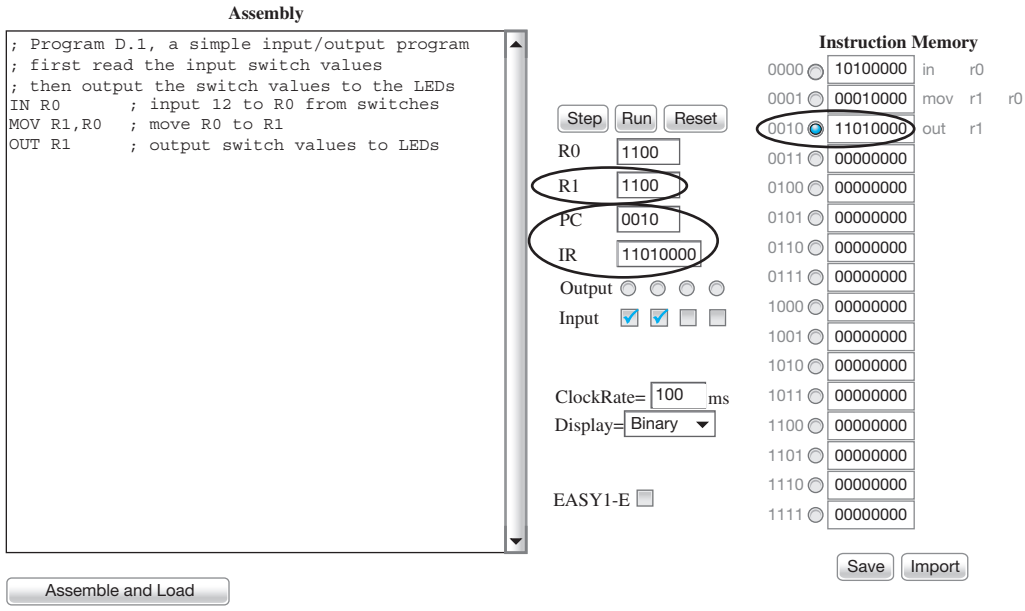


FIGURE D.9 EASY1 screen after clicking the Step button the second time to execute the second instruction for Program D.1

Observe in Figure D.9 that R1 contains 12, as it should after executing the MOV R1,R0 instruction. The PC now points to the address of the next instruction or the third instruction, which is at address 2. Observe that IR contains the machine code for the third instruction at address 2, which will be executed next.

Figure D.10 shows the EASY1 screen after clicking the Step button a third time to execute the third instruction at address 2.

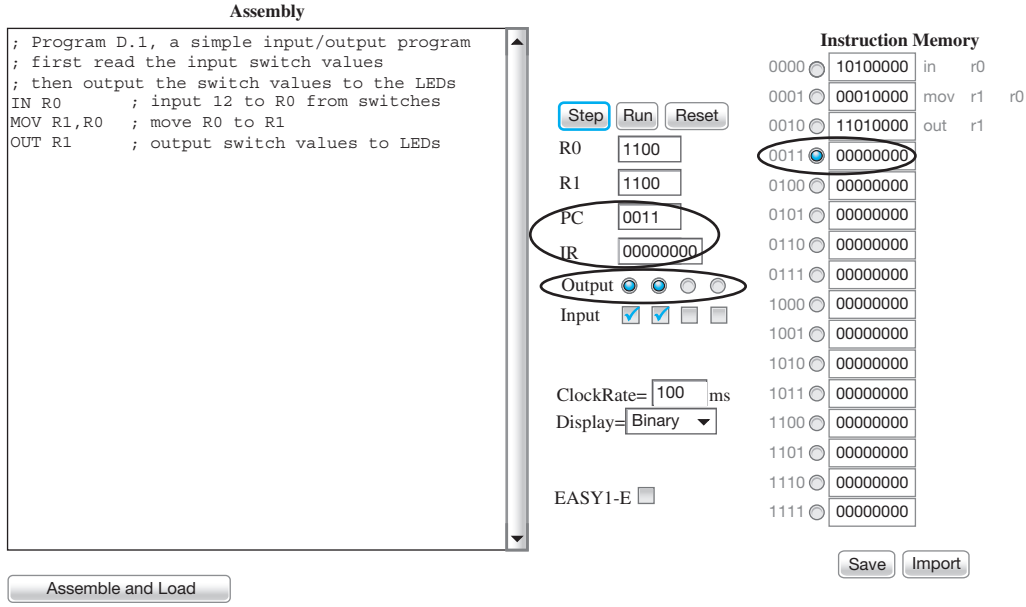


FIGURE D.10 EASY1 screen after clicking the Step button the third time to execute the third instruction for Program D.1

Observe in Figure D.10 that the output is 1100 or 12, as it should be after executing the OUT R1 instruction. Observe that the PC now points to the address of the next instruction, which would be the fourth instruction at address 3, but there is none. Because there is no fourth instruction, the IR contains all 0s, which is the default value of the instruction memory when no instruction has been loaded.

D.6 EXAMPLE 2—INPUT/OUTPUT PROGRAM MODIFIED TO RUN CONTINUOUSLY

The simple input/output program in this example can be modified to run continuously through its instructions. The assembly language program for doing this is shown in Program D.2.

```
; Program D.2, a simple input/output program
; first read the input switch values
; then output the switch values to the LEDs
repeat:  IN R0          ; input 12 to R0 from switches
         MOV R1,R0      ; move R0 to R1
         OUT R1         ; output switch values to LEDs
         LOADI R1,1     ; set R1 to 1 to force an
                        ; unconditional jump
         JNZ R1,repeat  ; jump back to repeat
```

PROGRAM D.2
Assembly language program for Example 1 modified to run continuously through its instructions

Notice that the program is written to read the input switch values into R0, move the contents of R0 to R1, output the contents of R1 to the output LEDs, and then unconditionally jump back to the label repeat to start the process over again.

Figure D.11 shows the assembly language program in Program D.2 entered in the editor of EASY1 and assembled by clicking on Assemble and Load and setting the input to 12 (1100).

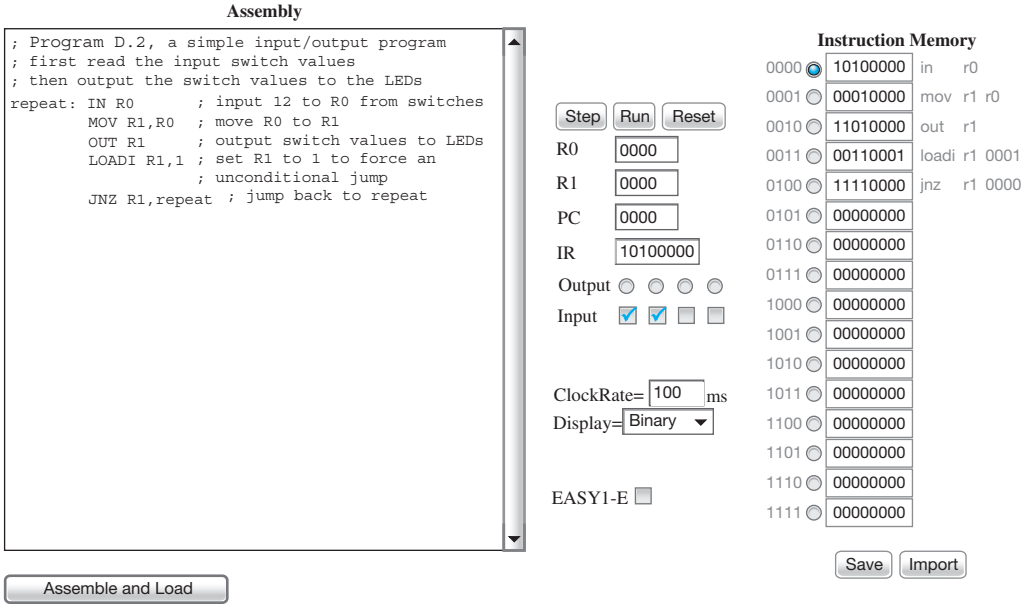


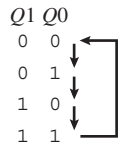
FIGURE D.11 EASY1 screen after entering Program D.2 and clicking Assemble and Load and setting the input to 12 (1100)

To verify that the program in Figure D.11 works as intended, click the Run button. Observe that the program executes the instructions in sequence from addresses 0 through 4 and repeats this sequence continuously. Enter a value from 0 to 15 via the input slide switches, and observe that the output LEDs follow the value that is selected for the input. Because there is no unconditional jump instruction for VBC1, the two instructions LOADI R1,1 and JNZ R1,repeat are equivalent to a nonexistent unconditional jump instruction JMP repeat.

D.7 EXAMPLE 3—A SIMPLE STATE MACHINE PROGRAM

Write an assembly language program for the 2-bit binary-up counter circuit represented in Figure D.12, where $Q1$ $Q0$ are displayed on the output LEDs 1 and 0, respectively.

FIGURE D.12 2-bit binary-up counter circuit state sequence diagram



Quite often, there are many different ways to write an assembly language program for a problem. An inline program would be simple to write and would require eight instructions plus an additional jump instruction to prevent the program from running amuck, for a total of nine instructions. One solution for the 2-bit binary-up counter (four states) using a loop counter is shown in Program D.3. Observe that this program only requires eight instructions.

PROGRAM D.3 Assembly language program for a 2-bit binary-up counter (four states) using a loop counter

```
; Program D.3, 2-bit binary up counter (4 states)
start: LOADI R0,0     ; initialize output value
loop:  OUT R0         ; output R0 to LEDs
       INC R0         ; increment loop counter
       MOV R1,R0      ; place test value in R1
       ADDI R1,12     ; 12 = 16 − (# of loops) = 16 − 4
       JNZ R1,loop    ; test for loop completion
       LOADI R1,1     ; set R1 to 1 to force an
                      ; unconditional jump
       JNZ R1,start   ; jump to start
```

Figure D.13 on the facing page shows Program D.3 entered in the editor of EASY1 and assembled by clicking on Assemble and Load. For this program, no input is required via the switches.

Click the Run button to observe that the program works as expected. Observe that the output LEDs 0 and 1 count up in binary from 00 to 11 and roll over—that is, go back to 00—and repeat this task continuously. Observe that Display in the EASY1 screen shows Hex. This converts the displayed values in the R0, R1, PC, and IR to hexadecimal as well as each displayed address and machine code in instruction memory to hexadecimal.

D.8 EXAMPLE 4—A COMPLEX STATE MACHINE PROGRAM

Write an assembly language program for the stoppable 4-bit binary-up counter circuit represented by the state sequence diagram in Figure D.14 shown on the facing page. $Q3$, $Q2$, $Q1$, and $Q0$ are displayed on the output LEDs 3 down to 0, respectively and S is provided by the status of the input slide switches. $S = 0$ is switch inputs 0000, and $S = 1$ is switch inputs that are any value except 0000—that is, 0001, 0010, 0011, etc.

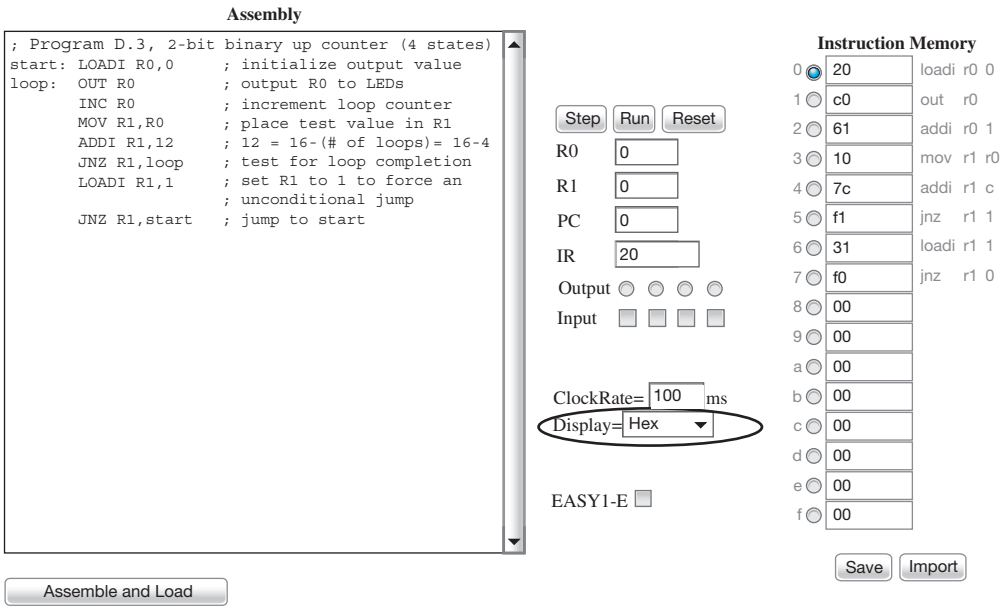


FIGURE D.13 EASY1 screen after entering Program D.3 and clicking Assemble and Load

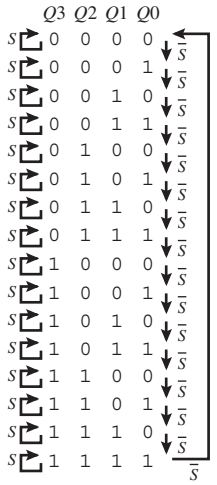


FIGURE D.14 State sequence diagram for a stoppable 4-bit binary-up counter circuit

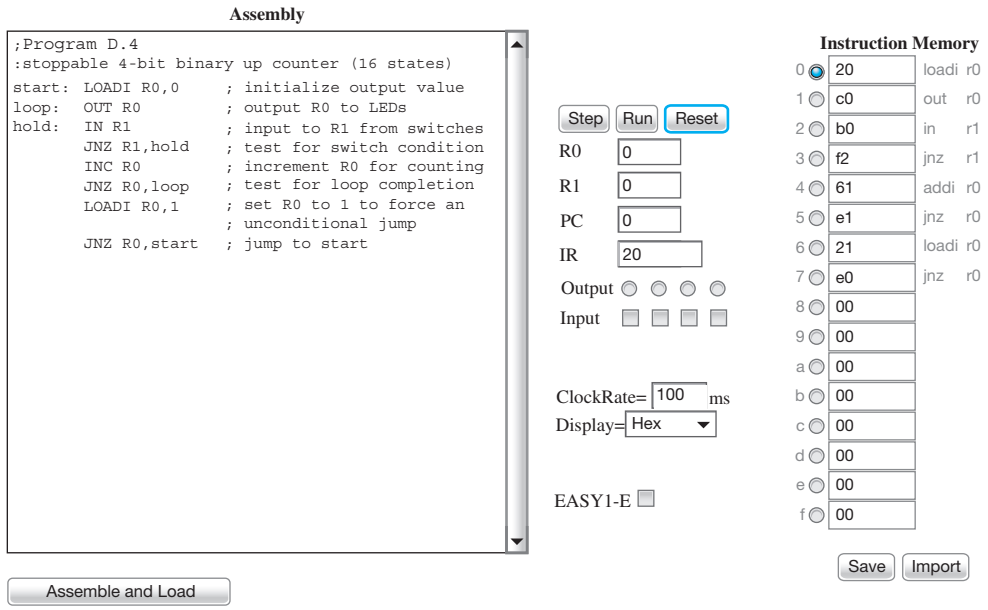
Program D.4 shows one solution for the stoppable 4-bit binary-up counter (16 states) in Figure D.14.

PROGRAM D.4 Assembly language program for a stoppable 4-bit binary-up counter (16 states)

```
; Program D.4
; stoppable 4-bit binary-up counter (16 states)
start:   LOADI R0,0    ; initialize output value
loop:    OUT R0        ; output R0 to LEDs
hold:    IN R1         ; input to R1 from switches
         JNZ R1,hold   ; test for switch condition
         INC R0        ; increment R0 for counting
         JNZ R0,loop   ; test for loop completion
         LOADI R0,1    ; set R0 to 1 to force an unconditional jump
         JNZ R0,start  ; jump to start
```

Figure D.15 shows the assembly language program in Program D.4 entered in the editor of EASY1 and assembled by clicking on Assemble and Load.

FIGURE D.15 EASY1 screen after entering Program D.4 and clicking Assemble and Load

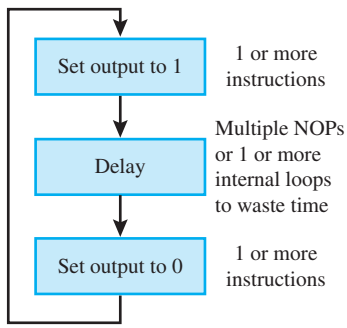


To observe the simulation of the program, click the Run button to verify that the output LEDs 3 down to 0 count up in binary from 0000 to 1111 and roll over and repeat this task continuously when $S = 0$. When $S = 1$, the counter marks time at the current count or present state until $S = 0$ and then continues counting.

D.9 EXAMPLE 5—GENERATING TIME DELAYS

Figure D.16 shows a simplified flow chart for generating a time delay.

FIGURE D.16 Simplified flow chart for a time delay



Write an assembly language program for a time delay that turns the least significant bit of the output on for nine machine cycles (a machine cycle or instruction cycle is the time it take to execute each assembly language instruction in VBC1) and then turns the output off and repeats this sequence over and over. The delay for nine machine cycles can be easily generated via an internal loop using a NOP. Either MOV R0, R0 or its alias NOP can be used to waste time.

Program D.5A shows one solution for the time delay as an inline program.

PROGRAM D.5A
Time delay as an inline program

```
; Program D.5A, Time Delay for 9 machine cycles
; inline program
start:  LOADI R0,1      ; set R0 to 1
        OUT R0          ; set LSB of Output to 1
        NOP             ; waste time
        NOP             ; waste time
        NOP             ; waste time
        NOP             ; waste time
        NOP             ; waste time
        NOP             ; waste time
        NOP             ; waste time
        LOADI R0,0      ; set R0 to 0
        OUT R0          ; set LSB of Output to 0
        LOADI R1,1      ; force unconditional jump
        JNZ R1,start    ; repeat sequence
```

After the LSB of output 1 is turned on, the seven NOPs in Program D.5A generates seven machine cycles. The instructions LOADI R0,0 and OUT R0, after the seven NOPs, generate two more machine cycles, for a total of nine machine cycles, thus keeping the LSB of output 1 turned on for nine machine cycles or nine instruction cycles.

Program D.5B shows an equivalent solution for the time delay as a program with an internal loop.

PROGRAM D.5B
Time delay as an internal loop program

```
; Program D.5B, Time Delay for 9 machine cycles
; internal loop
start:  LOADI R0,1      ; set R0 to 1
        OUT R0          ; set LSB of Output to 1
        LOADI R1,3      ; begin delay loop
back:   DEC R1          ; repeat the loop 3 times
        JNZ R1,back     ; test for loop completion
        LOADI R0,0      ; set R0 to 0
        OUT R0          ; set LSB of Output to 0
        LOADI R1,1      ; force unconditional jump
        JNZ R1,start    ; repeat sequence
```

After the LSB of output 1 is turned on, the internal loop in Program D.5B generates six machine cycles. The instruction LOADI R1,3, before the internal loop, and the instructions LOADI R0,0 and OUT R0, after the internal loop, generate three more machine cycles, for a total of nine machine cycles, thus keeping the LSB of output 1 turned on for nine machine cycles or nine instruction cycles.

D.10 USING EASY1 TO GENERATE MACHINE CODE FOR VBC1

After you write an assembly language program for VBC1 and assemble and load the program via EASY1, you can simulate the program to check for program functionality. If the program performs as you expected, you can then use the machine code generated by the assembler to load the instruction memory of VBC1. After loading the program, you can either single step the

program or run the program at the frequency of the signal *SPEED*, which allows you to observe the output LEDs during program execution.

You can generate the machine code for an assembly language program for VBC1 manually, but that takes time. Using the machine code generated by EASY1 usually takes less time and may result in fewer errors. After you enter the machine code into the instruction memory of VBC1, you can also test your program to see if it performs as expected via the hardware implementation of VBC1.

You may recall that the **Save** button allows you to save the contents of the instruction memory in a file. The saved file can then be copied and pasted into your VHDL code to initialize the memory of VBC1 or VBC1-E at startup. The saved file can also be read by a memory loader program, which allows the machine code to be loaded directly into the memory of VBC1 or VBC1-E via the USB connector on your FPGA board. See Appendix E for more information on loading instructions into memory. To use the memory loader program, you must complete Experiments 17 and 17L in Appendix A, which provide the VHDL design for VBC1 and the memory loader circuitry on your FPGA board. The memory loader program can also be used after you complete Experiments 25 and 25L in Appendix A, which provide the VHDL design for VBC1-E and the memory loader circuitry on your FPGA board.

Appendix E

Three Methods for Loading Instructions into Memory

The three methods for loading instructions into memory are:

1. Loading memory manually.
2. Initializing memory at startup.
3. Loading memory via the memory loader program.

Loading memory manually allows for manually loading the instructions in machine code form into memory by slide switches and push buttons after the VHDL design is downloaded into the FPGA. This method is the most tedious and also the most time consuming.

Initializing memory at startup allows for loading the instructions in machine code form into memory by placing the instructions in the VHDL code before the VHDL code is downloaded into the FPGA. This method requires no I/O support, which means it may be used without slide switches and push buttons. This method is more efficient than loading memory manually, but it does require that the VHDL design be synthesized and a new .bit file be generated each time the initialized memory contents change.

Loading memory via the memory loader program allows for loading the instructions in machine code form over the USB cable with a click of a button after the VHDL design is downloaded into the FPGA. This method requires a software program that controls the USB controller on the board, in addition to interface circuitry that must be added to the VHDL code. This method can also clear memory with a click of a button. This method is much more efficient than initializing memory at startup and only takes a little time.

E.1 LOADING MEMORY MANUALLY

For each of the following, use EASY1 or EASY1-E to assemble the program that you want to manually load into memory. Set the display mode in the EASY1 or EASY1-E GUI (graphical user interface) to binary or hex (hexadecimal), so that you can easily enter instructions via the slide switches SW7 … SW0. Make a copy of the GUI to help provide less chance for errors by reading a hard copy for the addresses and corresponding instructions, rather than reading the computer screen.

For Experiment 13, do the following:

1. Press and release BTN3 to reset VBC1 to address 0, as shown on Disp 3 (display 3 of the 7-segment display).
2. To load an instruction into memory, enter the instruction via the slide switches SW7 … SW0, then simply press and release BTN0.
3. Press and release BTN1 to increment to the next address.

4. Load the next instruction in the same manner. Each time an instruction is loaded, remember to press and release BTN1 to increment to the next address.
5. Continue in this manner until all instructions you want to load are loaded into the memory of VBC1.

For Experiments 14, 15, and 16, do the following:

1. Press and release BTN3 to reset VBC1 to address 0, as shown on Disp 3 (display 3 of the 7-segment display).
2. To load an instruction into memory, enter the instruction via the slide switches SW7 … SW0, then simultaneously press push buttons BTN2 and BTN0, then release them.
3. Press and release BTN1 to increment to the next address.
4. Load the next instruction in the same manner. Each time an instruction is loaded, remember to press and release BTN1 to increment to the next address.
5. Continue in this manner until all 16 instructions are loaded into the memory of VBC1.

For Experiments 17, 17L, 19, 20, 21, 22, 23, 24, 25, and 25L, do the following:

1. Press and hold BTN2, then press and release BTN3 to reset VBC1 to address 0, as shown on Disp 3 (display 3 of the 7-segment display).
2. Do not release push button BTN2.
3. To load an instruction into memory, enter the instruction via the slide switches SW7 … SW0, then press and release BTN0, but not BTN2.
4. As you continue to hold down BTN2, press and release BTN1 to increment to the next address.
5. Load the next instruction in the same manner. Each time an instruction is loaded, remember to press and release BTN1 to increment to the next address.
6. Continue in this manner until all 16 instructions (for VBC1) or 32 instructions (for VBC1-E) are loaded into the memory.

E.2 INITIALIZING MEMORY AT STARTUP

For Experiments 14, 15, 16, 17, and 17L, do the following:

1. Change "**signal** mem: mem_type;" to "**signal** mem: mem_type := (" under the declaration for mem_type for module 6 (instruction memory).
2. Assemble your VBC1 program via EASY1.
3. Use the EASY1 Save button to copy the machine code shown in the instruction memory of EASY1 into a file (choose a text file).
4. Open the text file and paste its contents after "**signal** mem: mem_type := (".
5. Regenerate the VHDL programming file and download it into the FPGA on a BASYS 2 board or on a NEXYS 2 board. This loads instruction memory at startup with an assembly language program.

For Experiments 19, 20, 21, and 22, do the following:

1. Change "**signal** mem: mem_type;" to "**signal** mem: mem_type := (" under the declaration for mem_type for module 6 (instruction memory).
2. Assemble your VBC1-E program via EASY1-E.
3. Use the EASY1-E Save button to copy the machine code shown in the instruction memory into a file (choose a text file).
4. Open the text file and paste its contents after "**signal** mem: mem_type := (".

5. After the 16th instruction, change the ";" to ");", to end the instruction list, then comment out the 16 remaining instructions.
6. Regenerate the VHDL programming file and download it into the FPGA on a BASYS 2 board or on a NEXYS 2 board. This loads instruction memory at startup with an assembly language program.

For Experiments 23, 24, 25, and 25L, do the following:

1. Change "**signal** mem: mem_type;" to "**signal** mem: mem_type := (" under the declaration for mem_type for module 6 (instruction memory).
2. Assemble your VBC1-E program via EASY1-E.
3. Use the EASY1-E Save button to copy the machine code shown in the instruction memory and the extended instruction memory into a file (choose a text file).
4. Open the text file and paste its contents after "**signal** mem: mem_type := (".
5. Regenerate the VHDL programming file and download it into the FPGA on a BASYS 2 board or on a NEXYS 2 board. This loads instruction memory and extended instruction memory at startup with an assembly language program.

E.3 LOADING MEMORY VIA THE MEMORY LOADER PROGRAM

The VBC1-L (VBC1-EL) memory loader program does the following:

- It will automatically load instruction memory after a VHDL design for VBC1-L has been downloaded into the FPGA on either a BASYS 2 board or a NEXYS 2 board.
- It will automatically load instruction memory and extended instruction memory after a VHDL design for VBC1-EL has been downloaded into the FPGA on either a BASYS 2 board or a NEXYS 2 board.
- It will overwrite instructions that have been manually loaded or instructions that have been loaded via initializing memory.

To use VBC1-L or VBC1-EL and the associated loader software, do the following:

1. From this textbook's website, download the installer for the VBC1-L (VBC1-EL) memory loader to your computer, and install the software.
2. Launch the VBC1-L (VBC1-EL) memory loader program.
3. Open the software program's built-in help by selecting the "Help" entry from the Help menu.
4. Select the "Getting Started" section of the software program's help and follow the step-by-step directions that appear.

This page intentionally left blank

Index

C

D

E

M

N

R

S

T

U

V

W

X

Z

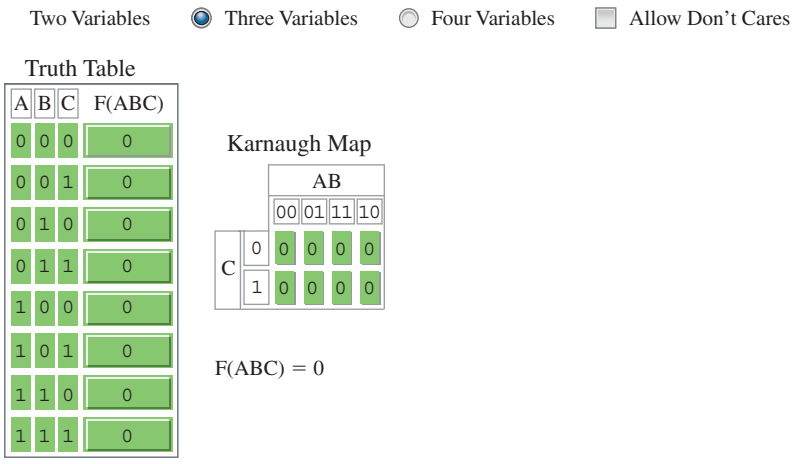


FIGURE 2.7 The screen (or GUI) for Karnaugh Map Explorer for three variables

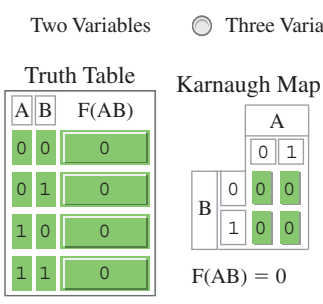


FIGURE 2.8 The screen (or GUI) for Karnaugh Map Explorer for two variables

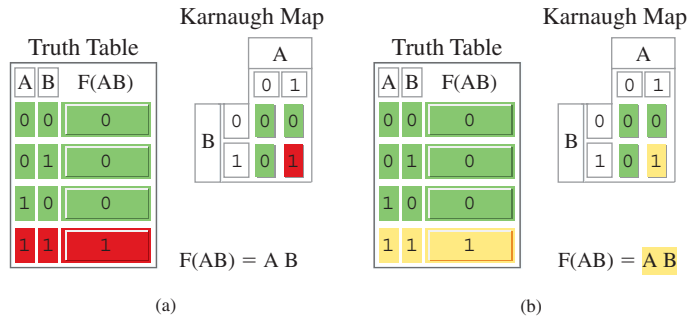


FIGURE 2.9 (a) The screen (or GUI) for Karnaugh Map Explorer for a 2-input AND gate; (b) highlighted areas in yellow in the truth table and K-map relate to the value of the function

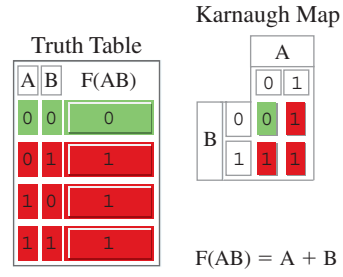


FIGURE 2.10 The screen (or GUI) for Karnaugh Map Explorer for a 2-input OR gate

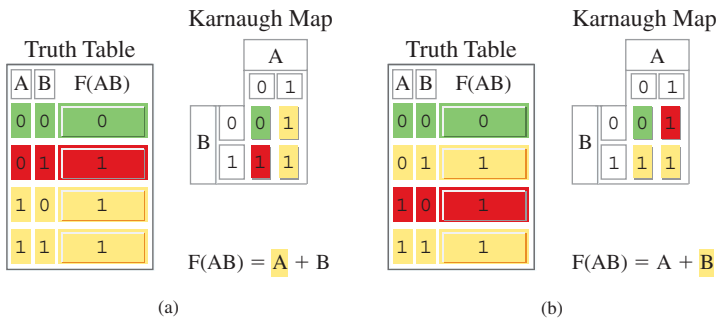


FIGURE 2.11 (a) Minterms that generate variable *A*; (b) minterms that generate variable *B*

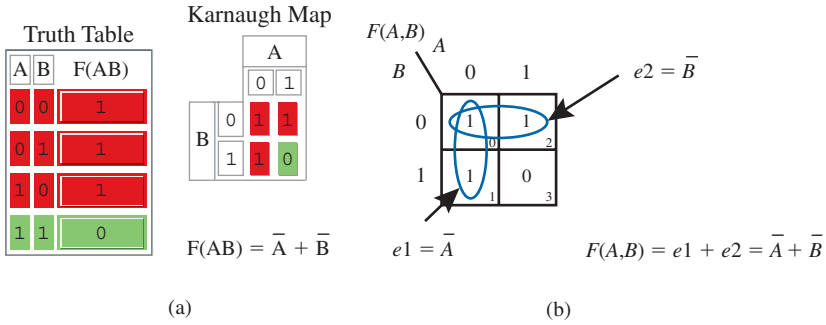


FIGURE 2.13 (a) The screen (or GUI) for Karnaugh Map Explorer for a 2-input NAND gate; (b) manual K-map reduction for a 2-input NAND gate

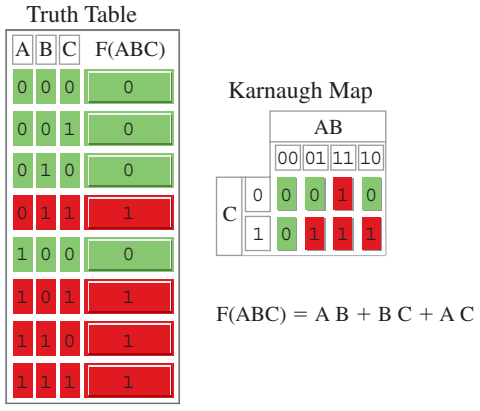


FIGURE 2.14 The screen (or GUI) for Karnaugh Map Explorer for a 3-input majority function

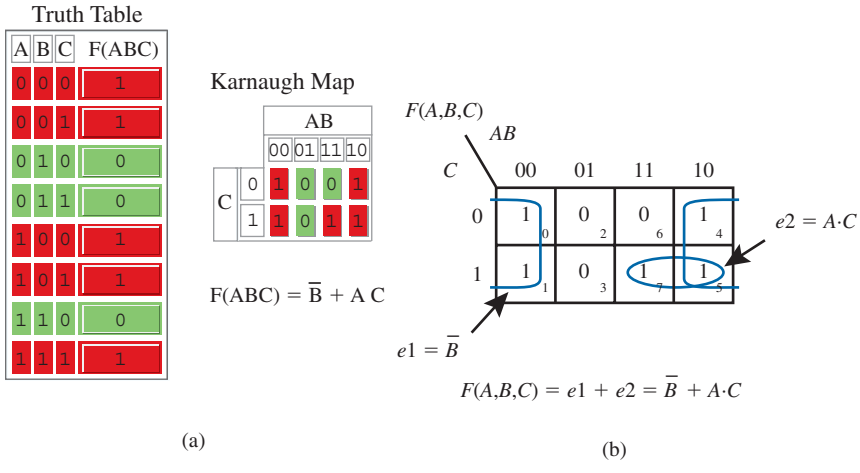


FIGURE 2.16 (a) The screen (or GUI) for Karnaugh Map Explorer with a group of four adjacent cells; (b) manual K-map reduction

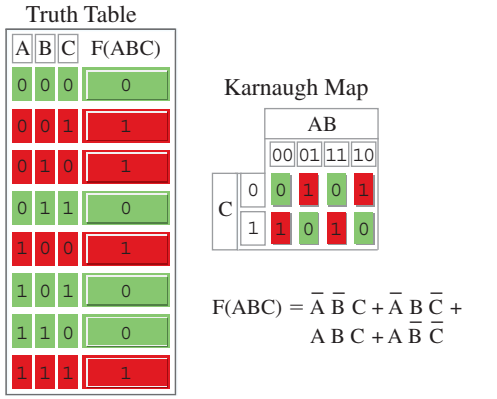
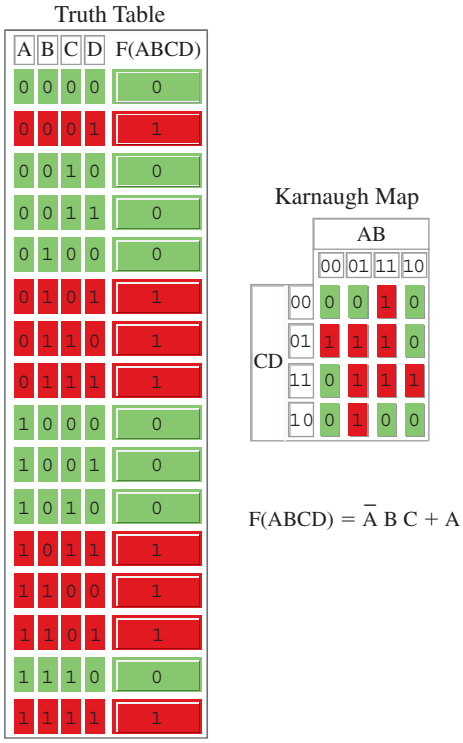


FIGURE 2.17 The screen (or GUI) for Karnaugh Map Explorer for a 3-input XOR gate

FIGURE 2.18 The screen (or GUI) for Karnaugh Map Explorer for a redundant larger group of cells



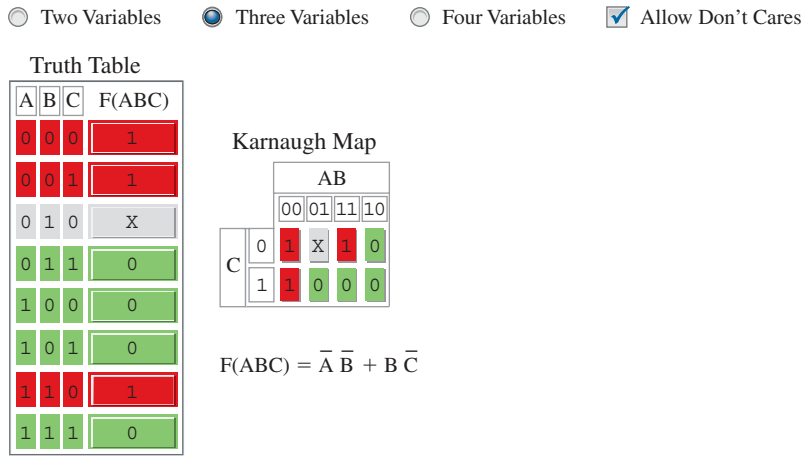


FIGURE 2.20 The screen (or GUI) for Karnaugh Map Explorer for a function with an incompletely specified, or don't-care, output

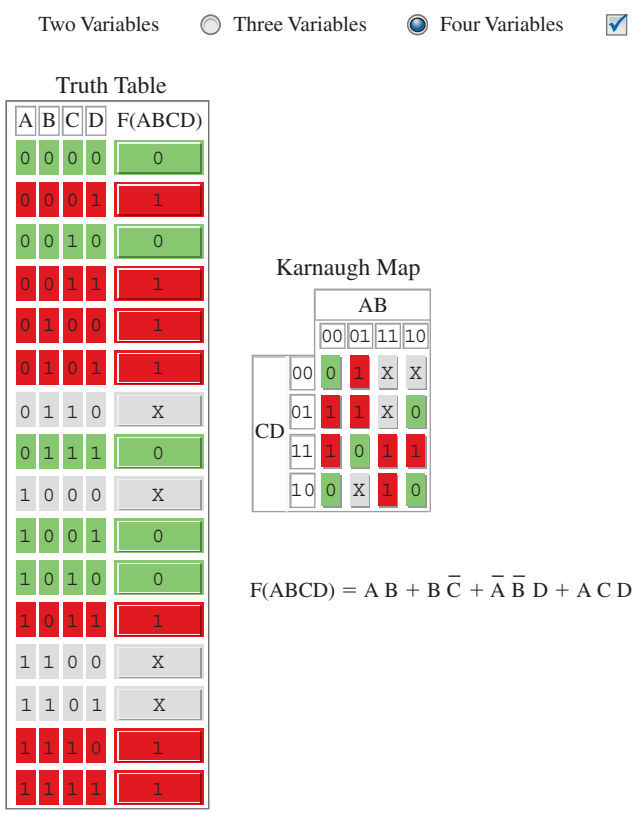


FIGURE 2.21 The screen (or GUI) for Karnaugh Map Explorer for a function with several incompletely specified, or don't-care, outputs