

D I G I T A L

S Y S T E M S

D E S I G N

U S I N G

VHDL

*Charles H. Roth, Jr.*

# DIGITAL SYSTEMS DESIGN USING VHDL®

---

**Charles H. Roth, Jr**

*The University of Texas at Austin*



**PWS Publishing Company**

I(T)P *An International Thomson Publishing Company*

Boston • Albany • Bonn • Cincinnati • London • Madrid • Melbourne  
Mexico City • New York • Paris • San Francisco • Tokyo • Toronto • Washington



PWS Publishing Company  
20 Park Plaza, Boston, MA 02116-4324

Copyright © 1998 by PWS Publishing Company, a division of International Thomson Publishing Inc.  
**All rights reserved.** No part of this book may be reproduced, stored in a retrieval system, or transcribed in any form or by any means — electronic, mechanical, photocopying, recording, or otherwise — without the prior written permission of PWS Publishing Company.

ITP™

International Thomson Publishing  
The trademark ITP is used under license. 5/11/1993

For more information, contact:

PWS Publishing Company  
20 Park Plaza  
Boston, MA 02116-4324

99  
99  
ST 350 V67 R245

International Thomson Publishing Europe  
Berkshire House 168–173  
High Holborn  
London WC1V 7AA  
England

Thomas Nelson Australia  
102 Dodds Street  
South Melbourne, 3205  
Victoria, Australia

Nelson Canada  
1120 Birchmount Road  
Scarborough, Ontario  
Canada M1K 5G4

Printed and bound in the United States of America.

98 99 00 01—10 9 8 7 6 5 4 3 2

Sponsoring Editor: Bill Barter  
Market Development Manager: Nathan Wilbur  
Assistant Editor: Suzanne Jeans  
Editorial Assistant: Tricia Kelly  
Production Editor: Pamela Rockwell  
Manufacturing Manager: Andrew Christensen  
Text Printer: Quebecor-Fairfield  
Cover Printer: Phoenix Color Corp.

International Thomson Editores  
Campos Eliseos 385, Piso 7  
Col. Polanco  
11560 Mexico C.F., Mexico

International Thomson Publishing GmbH  
Konigswinterer Strasse 418  
53227 Bonn, Germany

International Thomson Publishing Asia  
221 Henderson Road  
#05–10 Henderson Building  
Singapore 0315

International Thomson Publishing Japan  
Hirakawacho Kyowa Building, 31  
2-2-1 Hirakawacho  
Chiyoda-ku, Tokyo 102  
Japan

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

Roth, Charles H.

Digital system design using VHDL / by Charles H. Roth.

p. cm.

ISBN 0-534-95099-X (alk. paper)

1. Electronic digital computers—Circuits—Design and construction—Data processing. 2. VHDL (Hardware description language) 3. System design—Data processing. I. Title.

TK7888.4.R667 1997

621.39'2—dc21

97-24246

CIP

# CONTENTS

---

## PREFACE

VII

<b>CHAPTER 1 REVIEW OF LOGIC DESIGN FUNDAMENTALS</b>	<b>1</b>
1.1 Combinational Logic	1
1.2 Boolean Algebra and Algebraic Simplification	3
1.3 Karnaugh Maps	7
1.4 Designing with NAND and NOR Gates	10
1.5 Hazards in Combinational Networks	13
1.6 Flip-flops and Latches	14
1.7 Mealy Sequential Network Design	17
1.8 Design of a Moore Sequential Network	23
1.9 Equivalent States and Reduction of State Tables	25
1.10 Sequential Network Timing	28
1.11 Setup and Hold Times	29
1.12 Synchronous Design	31
1.13 Tristate Logic and Busses	35
<b>CHAPTER 2 INTRODUCTION TO VHDL</b>	<b>43</b>
2.1 VHDL Description of Combinational Networks	44
2.2 Modeling Flip-flops using VHDL Processes	50
2.3 VHDL Models for a Multiplexer	54

2.4 Compilation and Simulation of VHDL Code	56
2.5 Modeling a Sequential Machine	58
2.6 Variables, Signals, and Constants	65
2.7 Arrays	68
2.8 VHDL Operators	70
2.9 VHDL Functions	72
2.10 VHDL Procedures	74
2.11 Packages and Libraries	76
2.12 VHDL Model for a 74163 Counter	78
 <b>CHAPTER 3 DESIGNING WITH PROGRAMMABLE LOGIC DEVICES</b>	 <b>85</b>
3.1 Read-only Memories	85
3.2 Programmable Logic Arrays (PLAs)	89
3.3 Programmable Array Logic (PALs)	96
3.4 Other Sequential Programmable Logic Devices (PLDs)	101
3.5 Design of a Keypad Scanner	109
 <b>CHAPTER 4 DESIGN OF NETWORKS FOR ARITHMETIC OPERATIONS</b>	 <b>121</b>
4.1 Design of a Serial Adder with Accumulator	121
4.2 State Graphs for Control Networks	123
4.3 Design of a Binary Multiplier	124
4.4 Multiplication of Signed Binary Numbers	132
4.5 Design of a Binary Divider	144
 <b>CHAPTER 5 DIGITAL DESIGN WITH SM CHARTS</b>	 <b>161</b>
5.1 State Machine Charts	161
5.2 Derivation of SM Charts	167
5.3 Realization of SM Charts	178
5.4 Implementation of the Dice Game	180

---

5.5 Alternative Realizations for SM Charts Using Microprogramming	184
5.6 Linked State Machines	190
<b>CHAPTER 6 DESIGNING WITH PROGRAMMABLE GATE ARRAYS AND COMPLEX PROGRAMMABLE LOGIC DEVICES</b>	<b>201</b>
6.1 XILINX 3000 Series FPGAs	201
6.2 Designing with FPGAs	211
6.3 XILINX 4000 Series FPGAs	219
6.4 Using a One-Hot State Assignment	229
6.5 Altera Complex Programmable Logic Devices (CPLDs)	231
6.6 Altera FLEX 10K Series CPLDs	236
<b>CHAPTER 7 FLOATING-POINT ARITHMETIC</b>	<b>243</b>
7.1 Representation of Floating-Point Numbers	243
7.2 Floating-Point Multiplication	244
7.3 Other Floating-Point Operations	259
<b>CHAPTER 8 ADDITIONAL TOPICS IN VHDL</b>	<b>265</b>
8.1 Attributes	265
8.2 Transport and Inertial Delays	269
8.3 Operator Overloading	270
8.4 Multivalued Logic and Signal Resolution	272
8.5 IEEE-1164 Standard Logic	276
8.6 Generics	280
8.7 Generate Statements	282
8.8 Synthesis of VHDL Code	283
8.9 Synthesis Examples	289
8.10 Files and TEXTIO	295

---

<b>CHAPTER 9 VHDL MODELS FOR MEMORIES AND BUSSES</b>	<b>303</b>
9.1 Static RAM Memory	303
9.2 A Simplified 486 Bus Model	316
9.3 Interfacing Memory to a Microprocessor Bus	325
 <b>CHAPTER 10 HARDWARE TESTING AND DESIGN FOR TESTABILITY</b>	 <b>339</b>
10.1 Testing Combinational Logic	339
10.2 Testing Sequential Logic	344
10.3 Scan Testing	347
10.4 Boundary Scan	351
10.5 Built-In Self-Test	361
 <b>CHAPTER 11 DESIGN EXAMPLES</b>	 <b>373</b>
11.1 UART Design	373
11.2 Description of the MC68HC05 Microcontroller	387
11.3 Design of Microcontroller CPU	394
11.4 Completion of the Microcontroller Design	411
 <b>APPENDIX A VHDL LANGUAGE SUMMARY</b>	 <b>419</b>
<b>APPENDIX B BIT PACKAGE</b>	<b>425</b>
<b>APPENDIX C TEXTIO PACKAGE</b>	<b>435</b>
<b>APPENDIX D BEHAVIORAL VHDL CODE FOR M6805 CPU</b>	<b>437</b>
<b>APPENDIX E M6805 CPU VHDL CODE FOR SYNTHESIS</b>	<b>443</b>
<b>APPENDIX F PROJECTS</b>	<b>453</b>
 <b>REFERENCES</b>	 <b>459</b>
 <b>INDEX</b>	 <b>463</b>

# PREFACE

---

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

Students using this textbook should have completed a course in the fundamentals of logic design, including both combinational and sequential networks. Although no previous knowledge of VHDL is assumed, students should have programming experience using a modern higher-level language such as Pascal or C. A course in assembly language programming and basic computer organization is also very helpful, especially for Chapters 9 and 11.

Because students typically take their first course in logic design two years before this course, most students need a review of the basics. For this reason, Chapter 1 includes a review of logic design fundamentals. Most students can review this material on their own, so it is unnecessary to devote much lecture time to this chapter. However, a good understanding of timing in sequential networks and the principles of synchronous design is essential to the digital system design process.

Chapter 2 introduces the basics of VHDL, and this hardware description language is used throughout the rest of the book. Additional features of VHDL are introduced on an as-needed basis, and more advanced features are covered in Chapter 8. From the start, we relate the constructs of VHDL to the corresponding hardware. Some textbooks teach VHDL as a programming language and devote many pages to teaching the language syntax. Instead, our emphasis is on how to use VHDL in the digital design process. The language is very complex, so we do not attempt to cover all its features. We emphasize the basic features that are necessary for digital design and omit some of the less-used features.

VHDL is very useful in teaching top-down design. We can design a system at a high level and express the algorithms in VHDL. We can then simulate and debug the designs at this level before proceeding with the detailed logic design. However, no design is complete until it has actually been implemented in hardware and the hardware has been tested. For this reason, we recommend that the course include some lab exercises in which designs are implemented in hardware. We introduce simple programmable logic devices (PLDs) in Chapter 3 so that real hardware can be used early in the course if desired. The first part

of this chapter will be review for some students, but the second part of the chapter contains some important design examples. These examples illustrate the concept of fitting a design to a particular type of hardware. Chapter 3 also introduces the use of a *test bench* written in VHDL, which tests a design described in VHDL. The material in this chapter also serves as an introduction to more complex PLDs, which are discussed in Chapter 6.

Chapter 4 has two purposes. First, it presents some of the techniques used for computer arithmetic, including the design of systems for multiplication and division of signed binary numbers. Second, it shows how VHDL can be used to describe and simulate these systems. Use of a state machine for sequencing the operations in digital systems is an important concept presented in this chapter.

Use of sequential machine charts (SM charts) as an alternative to state graphs is presented in Chapter 5. We show how to write VHDL code based on SM charts and how to use programmable logic arrays for hardware implementation of SM charts. Transformation of SM charts leads to alternative hardware realizations of digital systems, and use of linked state machines facilitates the decomposition of complex systems into simpler ones. The design of a dice-game simulator is used to illustrate these techniques.

Chapter 6 describes two types of hardware devices that are widely used to implement digital system designs. First, field-programmable gate arrays (FPGAs) manufactured by XILINX are described, and techniques for designing with FPGAs are discussed. The dice-game simulator design from Chapter 5 is completed using FPGAs. We conclude the chapter with a description of Altera CPLDs. Use of FPGAs and CPLDs allows rapid prototyping of digital designs. Students can implement their designs in hardware without spending a lot of time in lab wiring up ICs.

Basic techniques for floating-point arithmetic are described in Chapter 7. A floating-point multiplier provides a complete design example, which is carried through starting with development of the basic algorithm, then simulating the system using VHDL, and finally implementing the system using an FPGA.

By the time students reach Chapter 8, they should be thoroughly familiar with the basics of VHDL. At this point we introduce some of the more advanced features of VHDL and illustrate their use. The use of multivalued logic, including the IEEE-1164 standard logic, is one of the important topics covered. In this chapter we also introduce the use of CAD tools for automatic synthesis of digital hardware from a VHDL description. We have deliberately delayed introduction of synthesis tools to this point, because we feel that it is very important for students to understand the basic principles of digital system design before they start using synthesis tools. Intelligent use of synthesis tools requires a good understanding of the underlying hardware because the way in which the VHDL code is written influences the efficiency of the resulting hardware implementation.

Chapter 9 describes the use of VHDL to model RAM memories and bus interfaces. Many of the VHDL features introduced in Chapter 8 are used in the examples in this chapter. This chapter emphasizes the use of VHDL simulation for checking to see that timing specifications for the memory and bus interface are satisfied. A system composed of static RAM chips interfaced to a 486 microprocessor bus is designed and tested to illustrate these techniques.

The important topics of hardware testing and design for testability are covered in Chapter 10. This chapter introduces the basic techniques for testing combinational and sequential logic. Then scan design and boundary-scan techniques, which facilitate the

testing of digital systems, are described. The chapter concludes with a discussion of built-in self-test (BIST). VHDL code for a boundary-scan example and for a BIST example is included. The topics in this chapter play an important role in digital system design, and we recommend that they be included in any course on this subject. Chapter 10 can be covered any time after the completion of Chapter 8.

Chapter 11 presents two complete design examples that illustrate the use of VHDL synthesis tools. The first example, a serial communications receiver-transmitter, should easily be understood by any student who has completed the material through Chapter 8. The final example is the complete design of a microcontroller, including the CPU and input-output interfaces. This example is fairly complex and requires some understanding of the basics of assembly language programming and computer organization.

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

All of the VHDL code in this textbook, including the bit library from Appendix B, is available on the world-wide web. The URL is <http://www.pws.com/ee/roth.html>.

## ACKNOWLEDGMENTS

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

Fredrick M. Cady, *Montana State University*

Tri Caohuu, *San Jose State University*

Gabriel Castelino, *GMI Engineering and Management Institute*

Maciej Ciesielski, *University of Massachusetts–Amherst*

Mike D. Ciletti, *University of Colorado at Colorado Springs*

Sura Lekhakul, *St. Cloud State University*

Figures 6-2, 6-3, 6-4, 6-8, 6-10, 6-11, 6-12, 6-13, 6-20, 6-21, 6-22, 6-24, and 6-25 are reprinted in this textbook with permission of Xilinx, Inc. Copyright ©1994 by Xilinx, Inc. All rights reserved. Xilinx, XACT, all XC-prefix product designations, LCA, and Logic Cell are trademarks of Xilinx.

Figures 6-29, 6-30, 6-31, 6-32, 6-33, 6-34, 6-35, 6-36, 6-37, and 6-38 are reprinted in this textbook with permission from Altera Corporation. Copyright © 1996 by Altera Corporation. All rights reserved. Altera is a trademark and service mark of Altera Corporation in the United States and other countries. Altera products are the intellectual property of Altera Corporation and are protected by copyright law and one or more U.S. and foreign patents and patent applications.

*C. H. Roth, Jr.*

# CHAPTER 1

## REVIEW OF LOGIC DESIGN FUNDAMENTALS

---

This chapter reviews many of the logic design topics normally taught in a first course in logic design. Some of the review examples that follow are referenced in later chapters of this text. For more details on any of the topics discussed in this chapter, the reader should refer to a standard logic design textbook such as *Fundamentals of Logic Design*, 4th ed. (Boston: PWS Publishing Company, 1995).

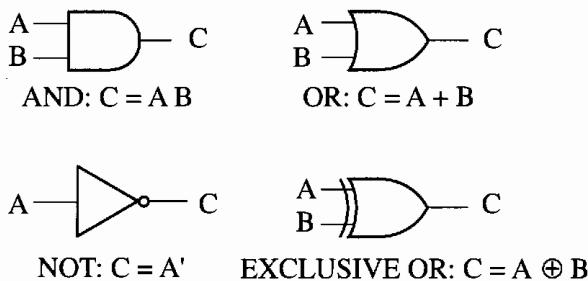
First, we review combinational logic and then sequential logic. Combinational logic has no memory, so the present output depends only on the present input. Sequential logic has memory, so the present output depends not only on the present input but also on the past sequence of inputs. The sections on sequential network timing and synchronous design are particularly important, since a good understanding of timing issues is essential to the successful design of digital systems.

### 1.1 COMBINATIONAL LOGIC

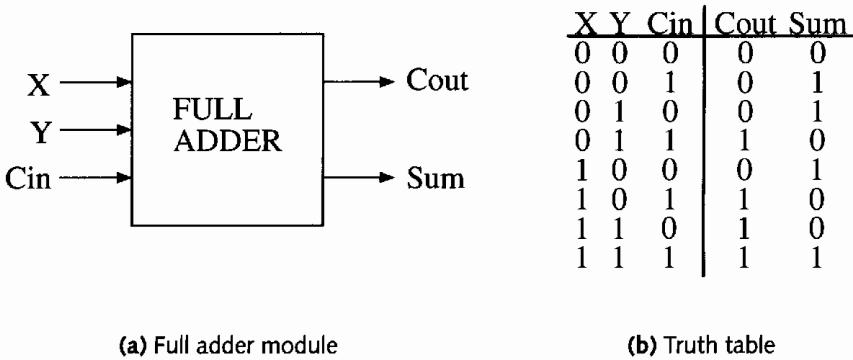
Some of the basic gates used in logic networks are shown in Figure 1-1. Unless otherwise specified, all the variables that we use to represent logic signals will be two-valued, and the two values will be designated 0 and 1. We will normally use positive logic, for which a low voltage corresponds to a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

For the AND gate of Figure 1-1, the output  $C = 1$  if and only if the input  $A = 1$  and the input  $B = 1$ . We will use a raised dot or simply write the variables side by side to indicate the AND operation; thus  $C = A \cdot B = AB$ . For the OR gate, the output  $C = 1$  if and only if the input  $A = 1$  or the input  $B = 1$  (inclusive OR). We will use + to indicate the OR operation; thus  $C = A + B$ . The NOT gate, or inverter, forms the complement of the input; that is, if  $A = 1$ ,  $C = 0$ , and if  $A = 0$ ,  $C = 1$ . We will use a prime ('') to indicate the complement (NOT) operation, so  $C = A'$ . The exclusive-OR (XOR) gate has an output  $C = 1$  if  $A = 1$  and  $B = 0$  or if  $A = 0$  and  $B = 1$ . The symbol  $\oplus$  represents exclusive OR, so we write

$$C = AB' + A'B = A \oplus B \quad (1-1)$$

**Figure 1-1 Basic Gates**

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

**Figure 1-2 Full Adder**

We will derive algebraic expressions for  $Sum$  and  $Cout$  from the truth table. From the table,  $Sum = 1$  when  $X = 0$ ,  $Y = 0$ , and  $Cin = 1$ . The term  $X'Y'Cin$  equals 1 only for this combination of inputs. The term  $X'YCin'$  = 1 only when  $X = 0$ ,  $Y = 1$ , and  $Cin = 0$ . The term  $XY'Cin'$  is 1 only for the input combination  $X = 1$ ,  $Y = 0$ , and  $Cin = 0$ . The term  $XYCin$  is 1 only when  $X = Y = Cin = 1$ . Therefore,  $Sum$  is formed by ORing these four terms together:

$$Sum = X'Y'Cin + X'YCin' + XY'Cin' + XYCin \quad (1-2)$$

Each of the terms in this expression is 1 for exactly one combination of input values. In a similar manner,  $Cout$  is formed by ORing four terms together:

$$Cout = X'Y\text{Cin} + XY'\text{Cin} + XY\text{Cin}' + XY\text{Cin} \quad (1-3)$$

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

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

$$Cout = m_3 + m_5 + m_6 + m_7 = \Sigma m(3, 5, 6, 7)$$

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

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

$$Cout = (X + Y + \text{Cin})(X + Y + \text{Cin}')(X + Y' + \text{Cin})(X' + Y + \text{Cin}) \quad (1-4)$$

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

$$Cout = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \prod M(0, 1, 2, 4)$$

where the decimal numbers correspond to the truth table rows for which *Cout* = 0.

## 1.2 BOOLEAN ALGEBRA AND ALGEBRAIC SIMPLIFICATION

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

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

**Example**

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

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

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

**Table 1-1 Laws and Theorems of Boolean Algebra**

Operations with 0 and 1:

$$X + 0 = X \quad (1-5) \quad X \cdot 1 = X \quad (1-5D)$$

$$X + 1 = 1 \quad (1-6) \quad X \cdot 0 = 0 \quad (1-6D)$$

Idempotent laws:

$$X + X = X \quad (1-7) \quad X \cdot X = X \quad (1-7D)$$

Involution law:

$$(X')' = X \quad (1-8)$$

Laws of complementarity

$$X + X' = 1 \quad (1-9) \quad X \cdot X' = 0 \quad (1-9D)$$

Commutative laws:

$$X + Y = Y + X \quad (1-10) \quad XY = YX \quad (1-10D)$$

Associative laws:

$$\begin{aligned} (X + Y) + Z &= X + (Y + Z) \\ &= X + Y + Z \end{aligned} \quad (1-11) \quad (XY)Z = X(YZ) = XYZ \quad (1-11D)$$

Distributive laws:

$$X(Y + Z) = XY + XZ \quad (1-12) \quad X + YZ = (X + Y)(X + Z) \quad (1-12D)$$

Simplification theorems:

$$XY + XY' = X \quad (1-13) \quad (X + Y)(X + Y') = X \quad (1-13D)$$

$$X + XY = X \quad (1-14) \quad X(X + Y) = X \quad (1-14D)$$

$$(X + Y')Y = XY \quad (1-15) \quad XY' + Y = X + Y \quad (1-15D)$$

DeMorgan's laws:

$$(X + Y + Z + \dots)' = X'Y'Z' \dots \quad (1-16) \quad (XYZ \dots)' = X' + Y' + Z' + \dots \quad (1-16D)$$

$$[f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot)]' = f(X_1', X_2', \dots, X_n', 1, 0, \cdot, +) \quad (1-17)$$

Duality:

$$(X + Y + Z + \dots)^D = XYZ \dots \quad (1-18) \quad (XYZ \dots)^D = X + Y + Z + \dots \quad (1-18D)$$

$$[f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot)]^D = f(X_1, X_2, \dots, X_n, 1, 0, \cdot, +) \quad (1-19)$$

Theorem for multiplying out and factoring:

$$(X + Y)(X' + Z) = XZ + X'Y \quad (1-20) \quad XY + X'Z = (X + Z)(X' + Y) \quad (1-20D)$$

Consensus theorem:

$$\begin{aligned} XY + YZ + X'Z &= XY + X'Z \\ &= (X + Y)(X' + Z) \end{aligned} \quad (1-21) \quad (1-21D)$$

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

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

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

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

$$\begin{aligned} Cout &= (X'YCin + XYCin) + (XY'Cin + XYCin) + (XYCin' + XYCin) \\ &= YCin + XCin + XY \end{aligned} \quad (1-22)$$

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

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

$$\begin{aligned} (A + BC)(D + E') + A'(B' + C')(D + E') &= D + E' \\ [X = D + E', Y = A + BC, Y' = A'(B' + C')] \end{aligned}$$

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

$$A'B + A'BC = A'B \quad [X = A'B]$$

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

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

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

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

4. *Adding redundant terms.* Redundant terms can be introduced in several ways, such as adding  $XX'$ , multiplying by  $(X + X')$ , adding  $YZ$  to  $XY + X'Z$  (consensus theorem), or adding  $XY$  to  $X$ . When possible, the terms added should be chosen so that they will combine with or eliminate other terms. For example,

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

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

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

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

The following theorems apply to exclusive-OR:

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

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

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

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

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

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

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

$$(X \oplus Y)' = X \oplus Y' = X' \oplus Y = XY + X'Y' \quad (1-30)$$

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

$$\begin{aligned} \text{Sum} &= X'(Y'Cin + YCin') + X(Y'Cin' + YCin) \\ &= X'(Y \oplus Cin) + X(Y \oplus Cin)' = X \oplus Y \oplus Cin \end{aligned} \quad (1-31)$$

### 1.3 KARNAUGH MAPS

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

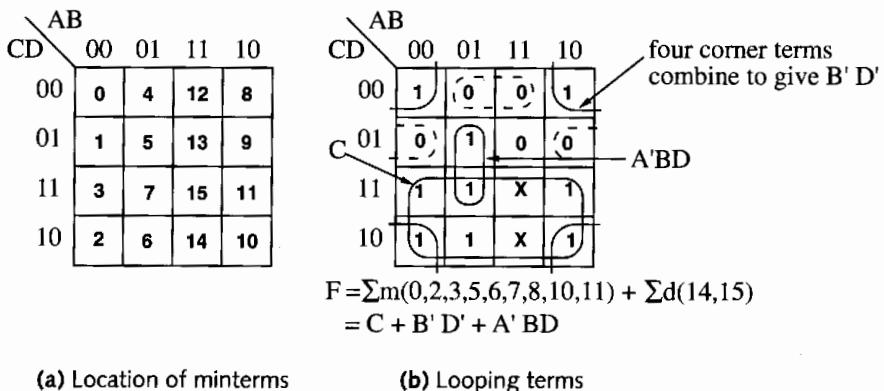
The variable values along the edge of the map are ordered so that adjacent squares on the map differ in only one variable. The first and last columns and the top and bottom rows of the map are considered to be adjacent. Two 1s in adjacent squares can be combined by eliminating one variable using  $xy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

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

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

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

Figure 1-3 Four-Variable Karnaugh Maps

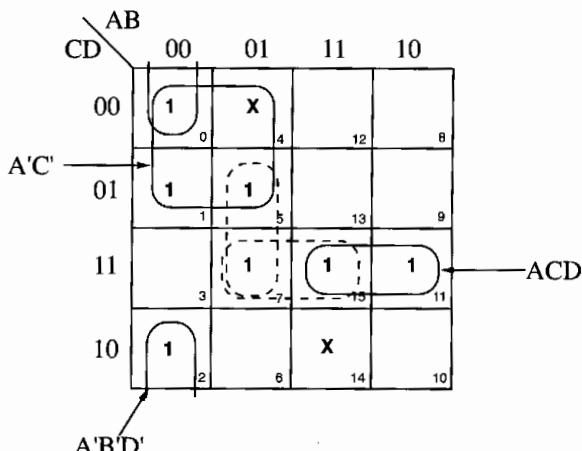


The minimum sum-of-products representation of a function consists of a sum of prime implicants. A group of one, two, four, or eight adjacent 1s on a map represents a prime implicant if it cannot be combined with another group of 1s to eliminate a variable. A prime implicant is essential if it contains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

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

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

Figure 1-4 Selection of Prime Implicants



When don't cares (Xs) are present on the map, the don't cares are treated like 1s when forming prime implicants, but the Xs are ignored when finding a minimum set of prime implicants to cover all the 1s. The following procedure can be used to obtain a minimum sum of products from a Karnaugh map:

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

To find minimum product of sums from a Karnaugh map, loop the 0s instead of the 1s. Since the 0s of  $F$  are the 1s of  $F'$ , looping the 0s in the proper way gives the minimum sum of products for  $F'$ , and the complement is the minimum product of sums for  $F$ . For Figure 1-3, we can first loop the essential prime implicants of  $F'$  ( $BC'D'$  and  $B'C'D$ , indicated by dashed loops), and then cover the remaining 0 with  $ABC'$  or  $AC'D$ . Thus one minimum sum for  $F'$  is

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

from which the minimum product of sums for  $F$  is

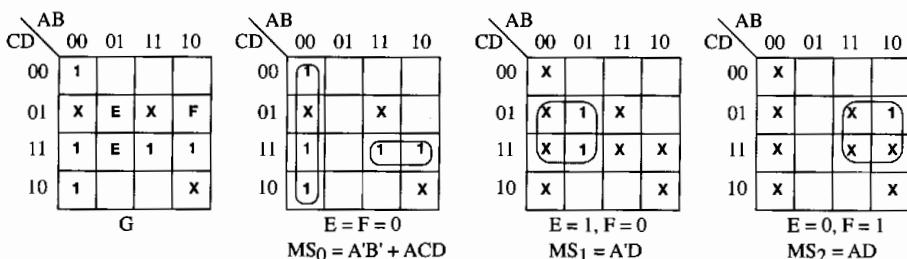
$$F = (B' + C + D)(B + C + D')(A' + B' + C)$$

By using map-entered variables, Karnaugh map techniques can be extended to simplify functions with more than four or five variables. Figure 1-5 shows a 4-variable map with two additional variables entered in the squares in the map. When  $E$  appears in a square, this means that if  $E = 1$ , the corresponding minterm is present in the function  $G$ , and if  $E = 0$ , the minterm is absent. Thus, the map represents the 6-variable function

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

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

**Figure 1-5 Simplification Using Map-Entered Variables**



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

$$F = MS_0 + P_1 MS_1 + P_2 MS_2 + \dots \quad (1-32)$$

where

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

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

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

For the example of Figure 1-5, maps for finding  $MS_0$ ,  $MS_1$ , and  $MS_2$  are shown, where  $E$  corresponds to  $P_1$  and  $F$  corresponds to  $P_2$ . The resulting expression is a minimum sum of products for  $G$ :

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

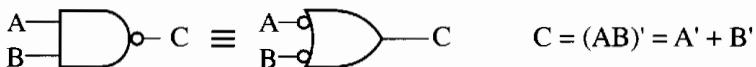
After some practice, it should be possible to write the minimum expression directly from the original map without first plotting individual maps for each of the minimum sums.

## 1.4 DESIGNING WITH NAND AND NOR GATES

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

**Figure 1-6** NAND and NOR Gates

NAND:



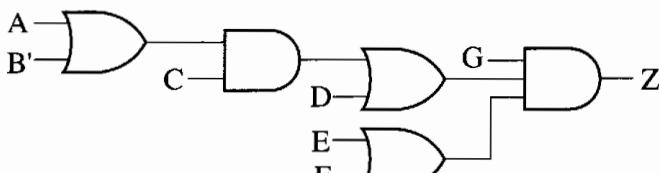
NOR:



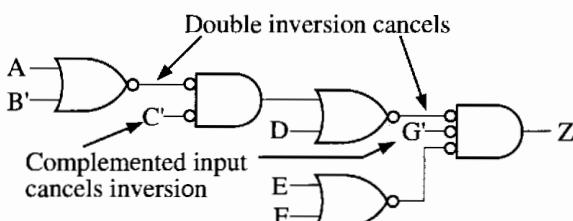
Conversion from networks of OR and AND gates to networks of all NOR gates or all NAND gates is straightforward. To design a network of NOR gates, start with a product-of-sums representation of the function (circle 0s on the Karnaugh map). Then find a network of OR and AND gates that has an AND gate at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

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

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

**Figure 1-7** Conversion to NOR Gates

(a) AND-OR network



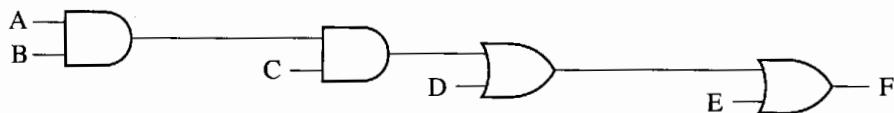
(b) Equivalent NOR-gate network

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

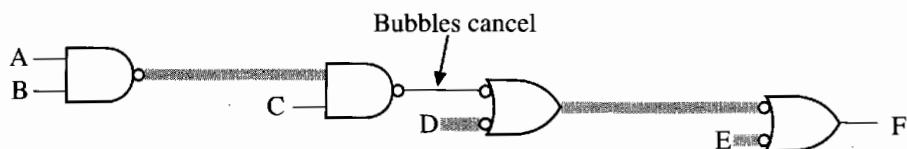
1. Convert all AND gates to NAND gates by adding an inversion bubble at the output. Convert OR gates to NAND gates by adding inversion bubbles at the inputs. (To convert to NOR, add inversion bubbles at all OR gate outputs and all AND gate inputs.)
2. Whenever an inverted output drives an inverted input, no further action is needed, since the two inversions cancel.
3. Whenever a non-inverted gate output drives an inverted gate input or vice versa, insert an inverter so that the bubbles will cancel. (Choose an inverter with the bubble at the input or output, as required.)
4. Whenever a variable drives an inverted input, complement the variable (or add an inverter) so the complementation cancels the inversion at the input.

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

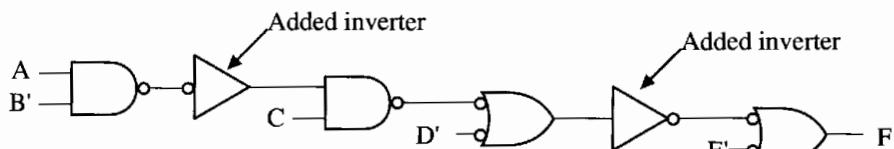
**Figure 1-8 Conversion of AND-OR Network to NAND Gates**



(a) AND-OR network



(b) First step in NAND conversion



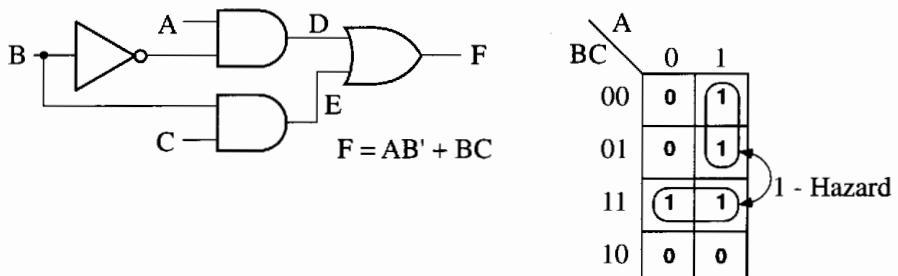
(c) Completed conversion

## 1.5 HAZARDS IN COMBINATIONAL NETWORKS

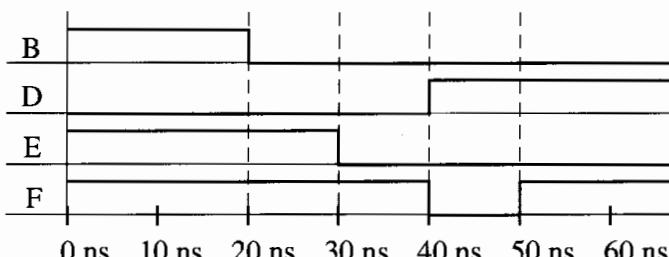
When the input to a combinational network changes, unwanted switching transients may appear in the output. These transients occur when different paths from input to output have different propagation delays. If, in response to an input change and for some combination of propagation delays, a network output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

Figure 1-9(a) illustrates a network with a static 1-hazard. If  $A = C = 1$ , the output should remain a constant 1 when  $B$  changes from 1 to 0. However, as shown in Figure 1-9(b), if each gate has a propagation delay of 10 ns,  $E$  will go to 0 before  $D$  goes to 1, resulting in a momentary 0 (a 1-hazard appearing in the output  $F$ ). As seen on the Karnaugh map, there is no loop that covers both minterm  $ABC$  and  $AB'C$ . So if  $A = C = 1$  and  $B$  changes, both terms can momentarily go to 0, resulting in a glitch in  $F$ . If we add a loop to the map and add the corresponding gate to the network (Figure 1-9(c)), this eliminates the hazard. The term  $AC$  remains 1 while  $B$  is changing, so no glitch can appear in the output.

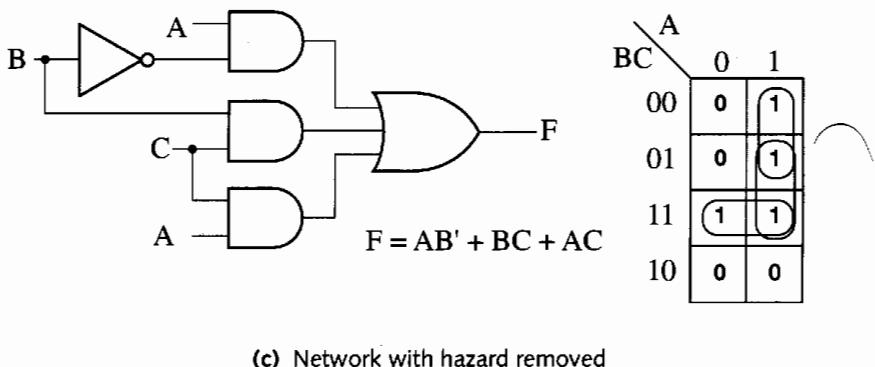
**Figure 1-9 Elimination of 1-Hazard**



(a) Network with 1-hazard



(b) Timing chart



(c) Network with hazard removed

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

1. Find a sum-of-products expression ( $F'$ ) for the output in which every pair of adjacent 1s is covered by a 1-term. (The sum of all prime implicants will always satisfy this condition.) A two-level AND-OR network based on this  $F'$  will be free of 1-, 0-, and dynamic hazards.
2. If a different form of network is desired, manipulate  $F'$  to the desired form by simple factoring, DeMorgan's laws, etc. Treat each  $x_i$  and  $x'_i$  as independent variables to prevent introduction of hazards.

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

## 1.6 FLIP-FLOPS AND LATCHES

Sequential networks commonly use flip-flops as storage devices. Figure 1-10 shows a clocked D flip-flop. This flip-flop can change state in response to the rising edge of the clock input. The next state of the flip-flop after the rising edge of the clock is equal to the D input before the rising edge. The characteristic equation of the flip-flop is therefore  $Q^+ = D$ , where  $Q^+$  represents the next state of the Q output after the active edge of the clock and D is the input before the active edge.

Figure 1-10 Clocked D Flip-flop with Rising-edge Trigger

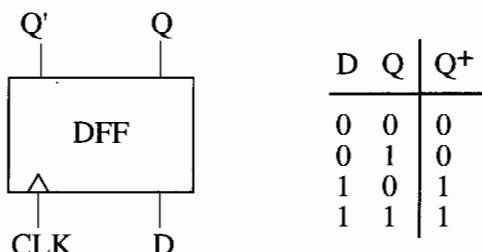
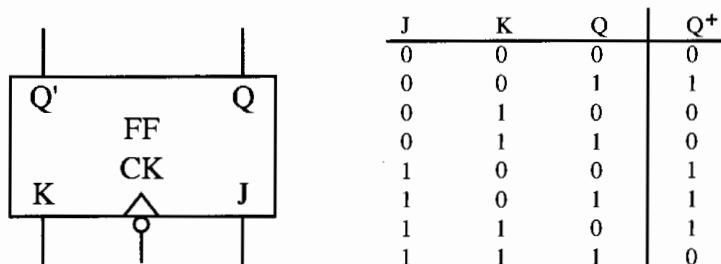


Figure 1-11 shows a clocked J-K flip-flop and its truth table. Since there is a bubble at the clock input, all state changes occur following the falling edge of the clock input. If  $J = K = 0$ , no state change occurs. If  $J = 1$  and  $K = 0$ , the flip-flop is set to 1, independent of the present state. If  $J = 0$  and  $K = 1$ , the flip-flop is always reset to 0. If  $J = K = 1$ , the flip-flop changes state. The characteristic equation, derived from the truth table in Figure 1-11, using a Karnaugh map is

$$Q^+ = JQ' + K'Q. \quad (1-33)$$

**Figure 1-11 Clocked J-K Flip-flop**

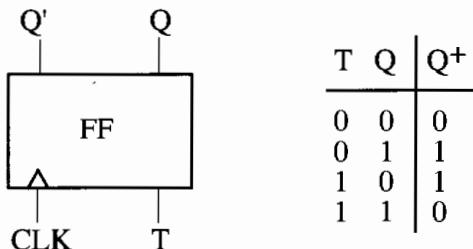


A clocked T flip-flop (Figure 1-12) changes state following the active edge of the clock if  $T = 1$ , and no state change occurs if  $T = 0$ . T flip-flops are particularly useful for designing counters. The characteristic equation for the T flip-flop is

$$Q^+ = QT' + Q'T = Q \oplus T \quad (1-34)$$

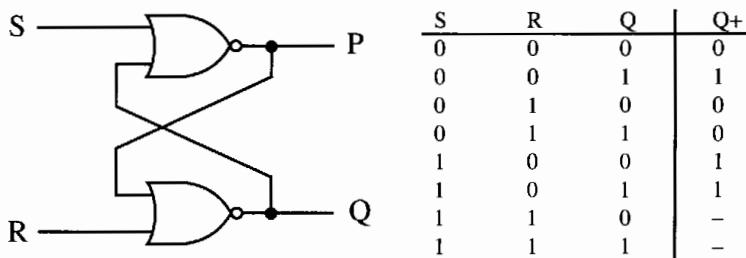
A J-K flip-flop is easily converted to a T flip-flop by connecting  $T$  to both  $J$  and  $K$ . Substituting  $T$  for  $J$  and  $K$  in (1-33) yields (1-34).

**Figure 1-12 Clocked T Flip-flop**



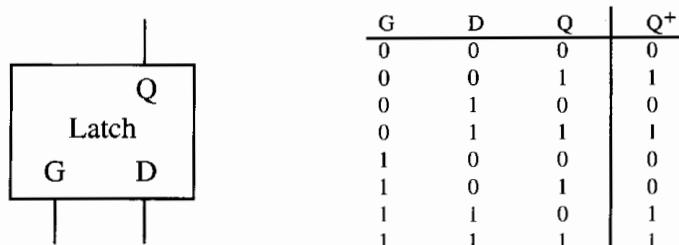
Two NOR gates can be connected to form an unclocked S-R (set-reset) flip-flop, as shown in Figure 1-13. An unclocked flip-flop of this type is often referred to as an S-R latch. If  $S = 1$  and  $R = 0$ , the  $Q$  output becomes 1 and  $P = Q'$ . If  $S = 0$  and  $R = 1$ ,  $Q$  becomes 0 and  $P = Q'$ . If  $S = R = 0$ , no change of state occurs. If  $R = S = 1$ ,  $P = Q = 0$ , which is not a proper flip-flop state, since the two outputs should always be complements. If  $R = S = 1$  and these inputs are simultaneously changed to 0, oscillation may occur. For this reason,  $S$  and  $R$  are not allowed to be 1 at the same time. For purposes of deriving the characteristic equation, we assume the  $S = R = 1$  never occurs, in which case  $Q^+ = S + R'Q$ . In this case,  $Q^+$  represents the state after any input changes have propagated to the  $Q$  output.

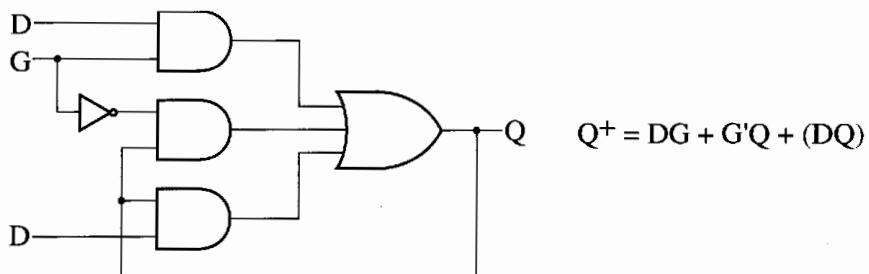
**Figure 1-13 S-R Latch**



A gated D latch (Figure 1-14), also called a transparent D latch, behaves as follows: If  $G = 1$ , then the  $Q$  output follows the  $D$  input ( $Q^+ = D$ ). If  $G = 0$ , then the latch holds the previous value of  $Q$  ( $Q^+ = Q$ ). The characteristic equation for the D latch is  $Q^+ = GD + G'Q$ . Figure 1-15 shows an implementation of the D latch using gates. Since the  $Q^+$  equation has a 1-hazard, an extra AND gate has been added to eliminate the hazard.

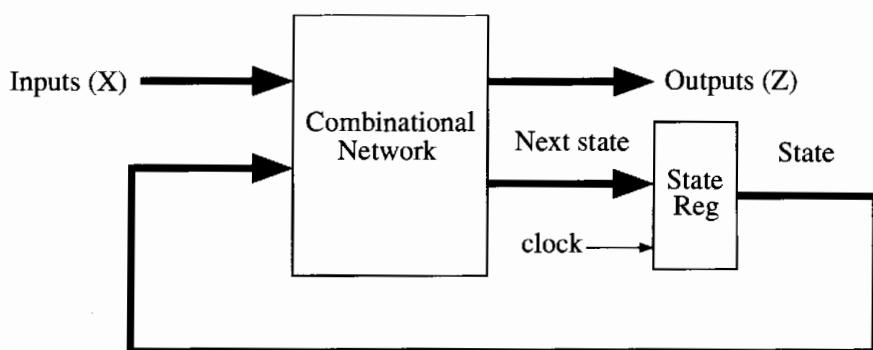
**Figure 1-14 Transparent D Latch**



**Figure 1-15 Implementation of D Latch**

## 1.7 MEALY SEQUENTIAL NETWORK DESIGN

The two basic types of sequential networks are Mealy and Moore. In a Mealy network, the outputs depend on both the present state and the present inputs. In a Moore network, the outputs depend only on the present state. A general model of a Mealy sequential network consists of a combinational network, which generates the outputs and the next state, and a state register, which holds the present state (see Figure 1-16). The state register normally consists of D flip-flops. The normal sequence of events is (1) the  $X$  inputs are changed to a new value, (2) after a delay, the corresponding  $Z$  outputs and next state appear at the output of the combinational network, and (3) the next state is clocked into the state register and the state changes. The new state feeds back into the combinational network, and the process is repeated.

**Figure 1-16 General Model of Mealy Sequential Machine**

As an example of a Mealy sequential network, we will design a code converter that converts an 8-4-2-1 binary-coded-decimal (BCD) digit to an excess-3-coded decimal digit. The input ( $X$ ) and output ( $Z$ ) will be serial with the least significant bit first. Table 1-2 lists the desired inputs and outputs at times  $t_0$ ,  $t_1$ ,  $t_2$ , and  $t_3$ . After receiving four inputs, the network should reset to its initial state, ready to receive another BCD digit.

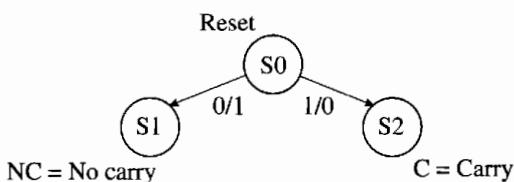
**Table 1-2** Code Converter

<b>X Input (BCD)</b>				<b>Z Output (excess -3)</b>			
$t_3$	$t_2$	$t_1$	$t_0$	$t_3$	$t_2$	$t_1$	$t_0$
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

We now construct a state graph for the code converter (Figure 1-17(a)). The excess-3 code is formed by adding 0011 to the BCD digit. For example,

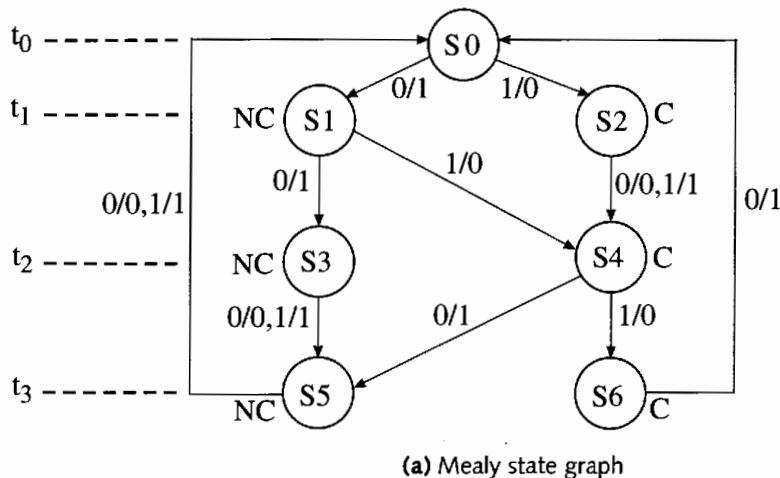
$$\begin{array}{r} 0100 \\ +0011 \\ \hline 0111 \end{array} \quad \begin{array}{r} 0101 \\ +0011 \\ \hline 1000 \end{array}$$

At  $t_0$ , we add 1 to the least significant bit, so if  $X = 0$ ,  $Z = 1$  (no carry), and if  $X = 1$ ,  $Z = 0$  (carry = 1). This leads to the following partial state graph:



S0 is the reset state, S1 indicates no carry after the first addition, and S2 indicates a carry of 1. At  $t_1$ , we add 1 to the next bit, so if there is no carry from the first addition (state S1),  $X = 0$  gives  $Z = 0 + 1 + 0 = 1$  and no carry (state S3), and  $X = 1$  gives  $Z = 1 + 1 + 0 = 0$  and a carry (state S4). If there is a carry from the first addition (state S2), then  $X = 0$  gives  $Z = 0 + 1 + 1 = 0$  and a carry (S4), and  $X = 1$  gives  $Z = 1 + 1 + 1 = 1$  and a carry (S4). At  $t_2$ , 0 is added to X, and transitions to S5 (no carry) and S6 are determined in a similar manner. At  $t_3$ , 0 is again added to X, and the network resets to S0.

Figure 1-17 State Graph and Table for Code Converter



(a) Mealy state graph

PS	NS		Z	
	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	-	1	-

(b) State table

Figure 1-17(b) gives the corresponding state table. (*Fundamentals of Logic Design*, pp. 429–430 gives an alternative way of deriving this state table.) At this point, we should verify that the table has a minimum number of states before proceeding (see Section 1-9). Since the state table has seven states, three flip-flops will be required to realize the table. The next step is to make a state assignment that relates the flip-flop states to the states in the table. The best state assignment to use depends on a number of factors. In many cases, we should try to find an assignment that will reduce the amount of required logic. For some types of programmable logic, a straight binary state assignment will work just as well as any other. For programmable gate arrays, a one-hot assignment (see Section 6.4) may be preferred.

In order to reduce the amount of logic required, we will make a state assignment using the following guidelines (see *Fundamentals of Logic Design*, p. 412):

- I. States that have the same next state (NS) for a given input should be given adjacent assignments (look at the columns of the state table).
- II. States that are the next states of the same state should be given adjacent assignments (look at the rows).
- III. States that have the same output for a given input should be given adjacent assignments.

Using these guidelines tends to clump 1s together on the Karnaugh maps for the next state and output functions. The guidelines indicate that the following states should be given adjacent assignments:

- I. (1, 2), (3, 4), (5, 6) (in the  $X = 1$  column,  $S_1$  and  $S_2$  both have NS  $S_4$ ; in the  $X = 0$  column,  $S_3$  and  $S_4$  have NS  $S_5$ , and  $S_5$  and  $S_6$  have NS  $S_0$ )
- II. (1, 2), (3, 4), (5, 6) ( $S_1$  and  $S_2$  are NS of  $S_0$ ;  $S_3$  and  $S_4$  are NS of  $S_1$ ; and  $S_5$  and  $S_6$  are NS of  $S_4$ )
- III. (0, 1, 4, 6), (2, 3, 5)

Figure 1-18(a) gives an assignment map, which satisfies the guidelines, and the corresponding transition table. Since state 001 is not used, the next state and outputs for this state are don't cares. The next state and output equations are derived from this table in Figure 1-19. Figure 1-20 shows the realization of the code converter using NAND gates and D flip-flops.

**Figure 1-18**

		$Q_1$	
		0	1
$Q_2 Q_3$	0	$S_0$	$S_1$
01		$S_2$	
11		$S_5$	$S_3$
10		$S_6$	$S_4$

(a) Assignment map

$Q_1 Q_2 Q_3$	$Q_1^+ Q_2^+ Q_3^+$		Z	
	X = 0	X = 1	X = 0	X = 1
000	100	101	1	0
100	111	110	1	0
101	110	110	0	1
111	011	011	0	1
110	011	010	1	0
011	000	000	0	1
010	000	xxx	1	x
001	xxx	xxx	x	x

(b) Transition table

If J-K flip-flops are used instead of D flip-flops, the input equations for the J-K flip-flops can be derived from the next state maps. Given the present state flip-flop ( $Q$ ) and the desired next state ( $Q^+$ ), the  $J$  and  $K$  inputs can be determined from the following table, which was derived from the truth table in Figure 1-11:

$Q$	$Q^+$	$J$	$K$	
0	0	0	$X$	(No change in $Q$ ; $J$ must be 0, $K$ may be 1 to reset $Q$ to 0.)
0	1	1	$X$	(Change to $Q = 1$ ; $J$ must be 1 to set or toggle.)
1	0	$X$	1	(Change to $Q = 0$ ; $K$ must be 1 to reset or toggle.)
1	1	$X$	0	(No change in $Q$ ; $K$ must be 0, $J$ may be 1 to set $Q$ to 1.)

Figure 1-19 Karnaugh Maps for Figure 1-17

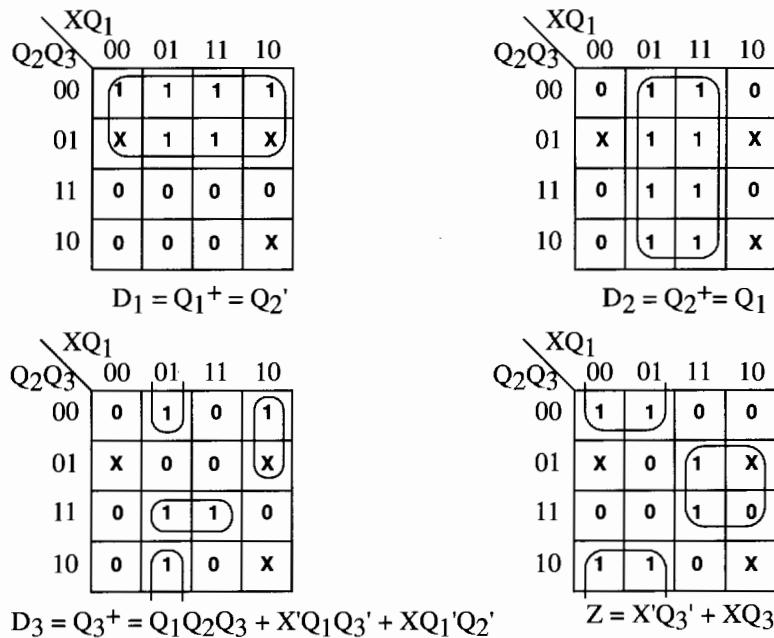


Figure 1-20 Realization of Code Converter

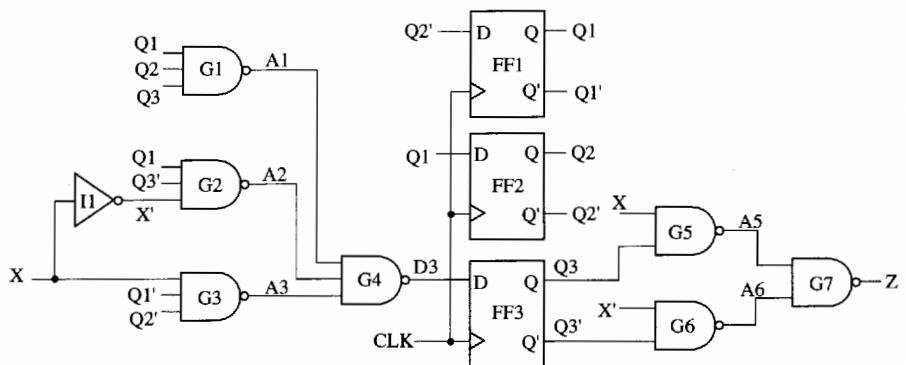
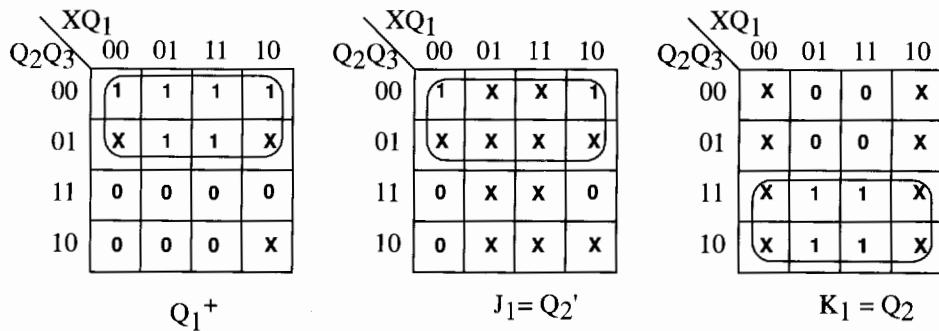
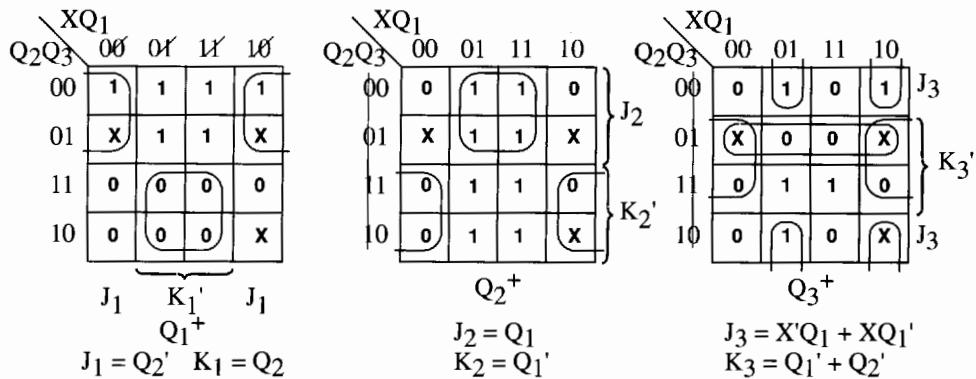


Figure 1-21 shows derivation of J-K flip-flops for the state table of Figure 1-17 using the state assignment of Figure 1-18. First, we derive the J-K input equations for flip-flop  $Q_1$  using the  $Q_1^+$  map as the starting point. From the preceding table, whenever  $Q_1$  is 0,  $J = Q_1^+$  and  $K = X$ . So, we can fill in the  $Q_1 = 0$  half of the  $J_1$  map the same as  $Q_1^+$  and the  $Q_1 = 0$  half of the  $K_1$  map as all Xs. When  $Q_1$  is 1,  $J_1 = X$  and  $K_1 = (Q_1^+)^*$ . So, we can fill in the  $Q_1 = 1$  half of the  $J_1$  map with Xs and the  $Q_1 = 1$  half of the  $K_1$  map with the complement of the  $Q_1^+$ . Since half of every J and K map is don't cares, we can avoid drawing separate J and K maps and read the Js and Ks directly from the  $Q^+$  maps, as illustrated in Figure 1-21(b). This shortcut method is based on the following: If  $Q = 0$ , then  $J = Q^+$ , so loop the 1s on the  $Q = 0$  half of the map to get J. If  $Q = 1$ , then  $K = (Q^+)^*$ , so loop the 0s on the  $Q = 1$  half of the map to get K. The J and K equations will be independent of  $Q$ , since  $Q$  is set to a constant value (0 or 1) when reading J and K. To make reading the Js and Ks off the map easier, we cross off the  $Q$  values on each map. In effect, using the shortcut method is equivalent to splitting the four-variable  $Q^+$  map into two three-variable maps, one for  $Q = 0$  and one for  $Q = 1$ .

**Figure 1-21 Derivation of J-K Input Equations**



(a) Derivation using separate J-K maps



(b) Derivation using the shortcut method

The following summarizes the steps required to design a sequential network:

1. Given the design specifications, determine the required relationship between the input and output sequences. Then find a state graph and state table.
2. Reduce the table to a minimum number of states. First eliminate duplicate rows by row matching; then form an implication table and follow the procedure in Section 1.9.
3. If the reduced table has  $m$  states ( $2^{n-1} < m \leq 2^n$ ),  $n$  flip-flops are required. Assign a unique combination of flip-flop states to correspond to each state in the reduced table.
4. Form the transition table by substituting the assigned flip-flop states for each state in the reduced state tables. The resulting transition table specifies the next states of the flip-flops and the output in terms of the present states of the flip-flops and the input.
5. Plot next-state maps and input maps for each flip-flop and derive the flip-flop input equations. Derive the output functions.
6. Realize the flip-flop input equations and the output equations using the available logic gates.
7. Check your design using computer simulation or another method.

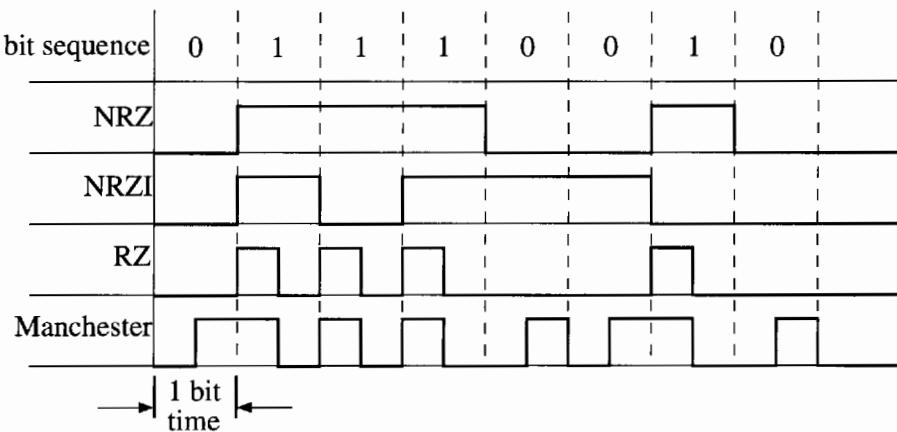
Steps 2 through 7 may be carried out using a suitable CAD program.

## 1.8 DESIGN OF A MOORE SEQUENTIAL NETWORK

As an example of designing a Moore sequential machine, we will design a converter for serial data. Binary data is frequently transmitted between computers as a serial stream of bits. Figure 1-22 shows four different coding schemes for serial data. The example shows transmission of the bit sequence 0, 1, 1, 1, 0, 0, 1, 0. With the NRZ (nonreturn-to-zero) code, each bit is transmitted for one bit time without any change. With the NRZI (nonreturn-to-zero-inverted) code, data is encoded by the presence or absence of transitions in the data signal. For each 0 in the original sequence, the bit transmitted is the same as the previous bit transmitted. For each 1 in the original sequence, the bit transmitted is the complement of the previous bit transmitted. For the RZ (return-to-zero) code, a 0 is transmitted as 0 for one full bit time, but a 1 is transmitted as a 1 for the first half of the bit time, and then the signal returns to 0 for the second half. For the Manchester code, a 0 is transmitted as 0 for the first half of the bit time and a 1 for the second half, but a 1 is transmitted as a 1 for the first half and a 0 for the second half. Thus, the Manchester encoded bit always changes in the middle of the bit time.

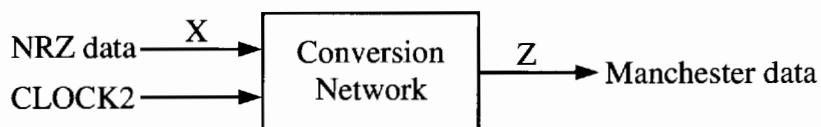
We will design a Moore sequential network that converts an NRZ-coded bit stream to a Manchester-coded bit stream (Figure 1-23). In order to do this, we will use a clock (*CLOCK2*) that is twice the frequency of the basic bit clock. If the NRZ bit is 0, it will be 0 for two *CLOCK2* periods, and if it is 1, it will be 1 for two *CLOCK2* periods. Thus, starting in the reset state ( $S_0$ ), the only two possible input sequences are 00 and 11, and the corresponding output sequences are 01 and 10. When a 0 is received, the network goes to  $S_1$  and outputs a 0; when the second 0 is received, it goes to  $S_2$  and outputs a 1. Starting in  $S_0$ , if a 1 is received, the network goes to  $S_3$  and outputs a 1, and when the second 1 is received, it must go to a state with a 0 output. Going back to  $S_0$  is appropriate since  $S_0$  has

Figure 1-22 Coding Schemes for Serial Data Transmission

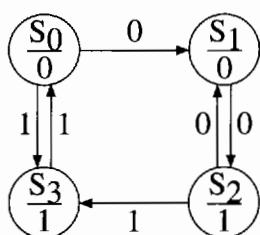


a 0 output and the network is ready to receive another 00 or 11 sequence. When in  $S_2$ , if a 00 sequence is received, the network can go to  $S_1$  and back to  $S_2$ . If a 11 sequence is received in  $S_2$ , the network can go to  $S_3$  and then back to  $S_0$ . The corresponding Moore state table has two don't cares, which correspond to input sequences that cannot occur.

Figure 1-23 Moore network for NRZ-to-Manchester Conversion



(a) Conversion network



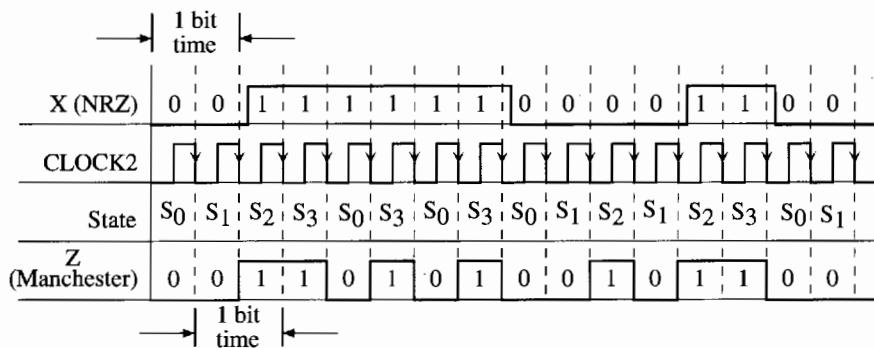
(b) State graph

Present State	Next State		Present Output (Z)
	X = 0	X = 1	
S <sub>0</sub>	S <sub>1</sub>	S <sub>3</sub>	0
S <sub>1</sub>	S <sub>2</sub>	-	0
S <sub>2</sub>	S <sub>1</sub>	S <sub>3</sub>	1
S <sub>3</sub>	-	S <sub>0</sub>	1

(c) State table

Figure 1-24 shows the timing chart for the Moore network. Note that the Manchester output is shifted one clock time with respect to the NRZ input. This shift occurs because a Moore network cannot respond to an input until the active edge of the clock occurs. This is in contrast to a Mealy network, for which the output can change after the input changes and before the next clock.

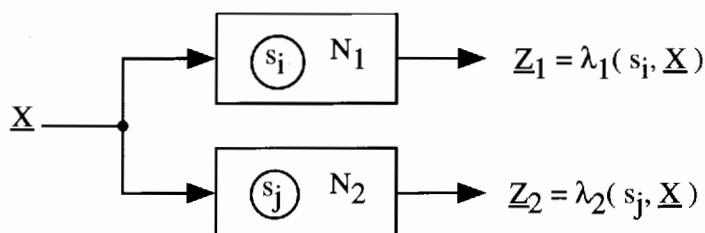
**Figure 1-24 Timing for Moore Network**



## 1.9 EQUIVALENT STATES AND REDUCTION OF STATE TABLES

The concept of equivalent states is important for the design and testing of sequential networks. Two states in a sequential network are said to be *equivalent* if we cannot tell them apart by observing input and output sequences. Consider two sequential networks,  $N_1$  and  $N_2$  (see Figure 1-25).  $N_1$  and  $N_2$  could be copies of the same network.  $N_1$  is started in state  $s_i$ , and  $N_2$  is started in state  $s_j$ . We apply the same input sequence,  $\underline{X}$ , to both networks and observe the output sequences,  $\underline{Z}_1$  and  $\underline{Z}_2$ . (The underscore notation indicates a sequence.) If  $\underline{Z}_1$  and  $\underline{Z}_2$  are the same, we reset the networks to states  $s_i$  and  $s_j$ , apply a different input sequence, and observe  $\underline{Z}_1$  and  $\underline{Z}_2$ . If the output sequences are the same for all possible input sequences, we say the  $s_i$  and  $s_j$  are equivalent ( $s_i \equiv s_j$ ). Formally, we can define equivalent states as follows:  $s_i \equiv s_j$  if and only if, for every input sequence  $\underline{X}$ , the output sequences  $\underline{Z}_1 = \lambda_1(s_i, \underline{X})$  and  $\underline{Z}_2 = \lambda_2(s_j, \underline{X})$  are the same. This is not a very practical way to test for state equivalence since, at least in theory, it requires input sequences of infinite length. In practice, if we have a bound on number of states, then we can limit the length of the test sequences.

**Figure 1-25 Sequential Networks**



A more practical way to determine state equivalence uses the state equivalence theorem:  $s_i \equiv s_j$  if and only if for every single input  $X$ , the outputs are the same and the next states are equivalent. When using the definition of equivalence, we must consider all input sequences, but we do not need any information about the internal state of the system. When using the state equivalence theorem, we must look at both the output and next state, but we need to consider only single inputs rather than input sequences.

The table of Figure 1-26(a) can be reduced by eliminating equivalent states. First, observe that states  $a$  and  $h$  have the same next states and outputs when  $X = 0$  and also when  $X = 1$ . Therefore,  $a \equiv h$  so we can eliminate row  $h$  and replace  $h$  with  $a$  in the table. To determine if any of the remaining states are equivalent, we will use the state equivalence theorem. From the table, since the outputs for states  $a$  and  $b$  are the same,  $a \equiv b$  if and only if  $c \equiv d$  and  $e \equiv f$ . We say that  $c-d$  and  $e-f$  are implied pairs for  $a-b$ . To keep track of the implied pairs, we make an implication chart, as shown in Figure 1-26(b). We place  $c-d$  and  $e-f$  in the square at the intersection of row  $a$  and column  $b$  to indicate the implication. Since states  $d$  and  $e$  have different outputs, we place an  $\times$  in the  $d-e$  square to indicate that  $d \not\equiv e$ . After completing the implication chart in this way, we make another pass through the chart. The  $e-g$  square contains  $c-e$  and  $b-g$ . Since the  $c-e$  square has an  $\times$ ,  $c \not\equiv e$ , which implies  $e \not\equiv g$ , so we  $\times$  out the  $e-g$  square. Similarly, since  $e \not\equiv f$ , we  $\times$  out the  $f-g$  square. On the next pass through the chart, we  $\times$  out all the squares that contain  $e-f$  or  $f-g$  as implied pairs (shown on the chart with dashed  $\times$ s). In the next pass, no additional squares are  $\times$ ed out, so the process terminates. Since all the squares corresponding to non-equivalent states have been  $\times$ ed out, the coordinates of the remaining squares indicate equivalent state pairs. From the first column,  $a \equiv b$ ; from third column,  $c \equiv d$ ; and from the fifth column,  $e \equiv f$ .

The implication table method of determining state equivalence can be summarized as follows:

1. Construct a chart that contains a square for each pair of states.
2. Compare each pair of rows in the state table. If the outputs associated with states  $i$  and  $j$  are different, place an  $\times$  in square  $i-j$  to indicate that  $i \not\equiv j$ . If the outputs are the same, place the implied pairs in square  $i-j$ . (If the next states of  $i$  and  $j$  are  $m$  and  $n$  for some input  $x$ , then  $m-n$  is an implied pair.) If the outputs and next states are the same (or if  $i-j$  implies only itself), place a check ( $\checkmark$ ) in square  $i-j$  to indicate that  $i \equiv j$ .
3. Go through the table square by square. If square  $i-j$  contains the implied pair  $m-n$ , and square  $m-n$  contains an  $\times$ , then  $i \not\equiv j$ , and an  $\times$  should be placed in square  $i-j$ .
4. If any  $\times$ s were added in step 3, repeat step 3 until no more  $\times$ s are added.
5. For each square  $i-j$  that does not contain an  $\times$ ,  $i \equiv j$ .

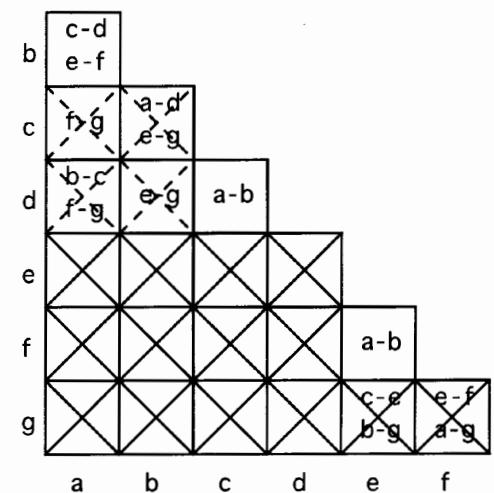
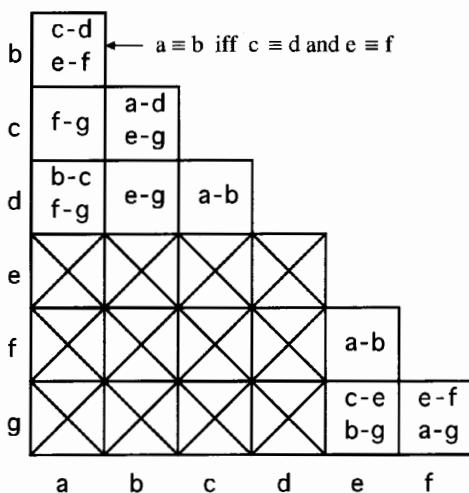
If desired, row matching can be used to partially reduce the state table before constructing the implication table. Although we have illustrated this procedure for a Mealy table, the same procedure applies to a Moore table.

Two sequential networks are said to be equivalent if every state in the first network has an equivalent state in the second network, and vice versa.

Figure 1-26 State Table Reduction

Present State	Next State		Present Output	
	X = 0	1	X = 0	1
a	c	f	0	0
b	d	e	0	0
c	<del>a</del>	g	0	0
d	b	g	0	0
e	e	b	0	1
f	f	a	0	1
g	c	g	0	1
<del>h</del>	<del>e</del>	<del>f</del>	0	0

(a) State table reduction by row matching



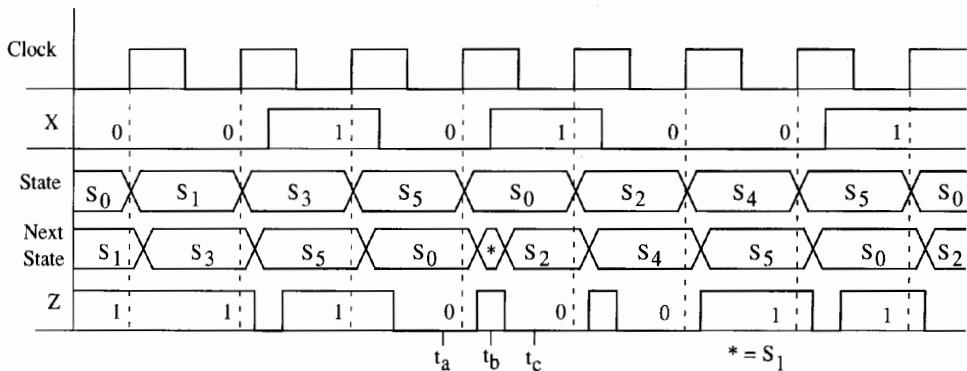
	X =	0	1	X =	0	1
a		c	e		0	0
c		a	g		0	0
e		e	a		0	1
g		c	g		0	1

(d) Final reduced table

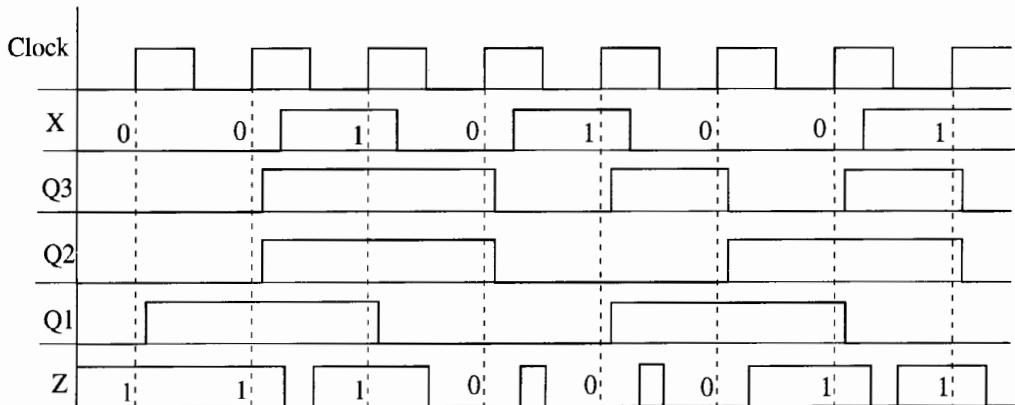
### 1.10 SEQUENTIAL NETWORK TIMING

If the state table of Figure 1-17(b) is implemented in the form of Figure 1-16, the timing waveforms are as shown in Figure 1-27. Propagation delays in the network have been neglected. In this example, the input sequence is 00101001, and  $X$  is assumed to change in the middle of the clock pulse. At any given time, the next state and  $Z$  output can be read from the next state table. For example, at time  $t_a$ , State =  $S_5$  and  $X = 0$ , so Next State =  $S_0$  and  $Z = 0$ . At time  $t_b$  following the rising edge of the clock, State =  $S_0$  and  $X$  is still 0, so Next State =  $S_1$  and  $Z = 1$ . Then  $X$  changes to 1, and at time  $t_c$  Next State =  $S_2$  and  $Z = 0$ . Note that there is a *glitch* (sometimes called a false output) at  $t_b$ . The  $Z$  output momentarily has an incorrect value at  $t_b$ , because the change in  $X$  is not exactly synchronized with the active edge of the clock. The correct output sequence, as indicated on the waveform, is 1 1 0 0 0 1 1. Several glitches appear between the correct outputs; however, these are of no consequence if  $Z$  is read at the right time. The glitch in the next state at  $t_b$  ( $S_1$ ) also does not cause a problem, because the next state has the correct value at the active edge of the clock.

Figure 1-27 Timing Diagram for Code Converter

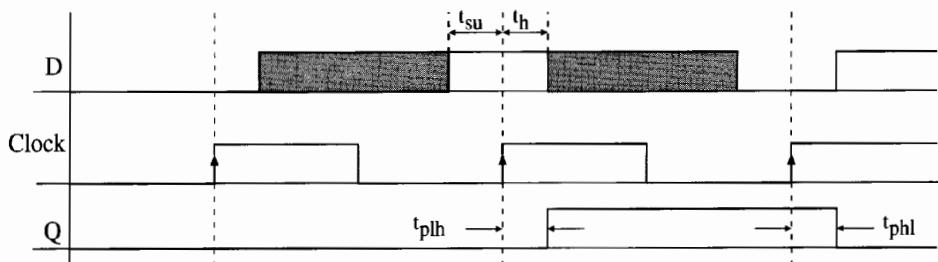


The timing waveforms derived from the network of Figure 1-20 are shown in Figure 1-28. They are similar to the general timing waveforms given in Figure 1-27 except that State has been replaced with the states of the three flip-flops, and a propagation delay of 10 ns has been assumed for each gate and flip-flop.

**Figure 1-28 Timing Diagram for Figure 1-20**

## 1.11 SETUP AND HOLD TIMES

For an ideal D flip-flop, if the  $D$  input changed at exactly the same time as the active edge of the clock, the flip-flop would operate correctly. However, for a real flip-flop, the  $D$  input must be stable for a certain amount of time before the active edge of the clock (called the setup time). Furthermore,  $D$  must be stable for a certain amount of time after the active edge of the clock (called the hold time). Figure 1-29 illustrates setup and hold times for a  $D$  flip-flop that changes state on the rising edge of the clock.  $D$  can change at any time during the shaded region on the diagram, but it must be stable during the time interval  $t_{su}$  before the active edge and for  $t_h$  after the active edge. If  $D$  changes at any time during the forbidden interval, it cannot be determined whether the flip-flop will change state. Even worse, the flip-flop may malfunction and output a short pulse or even go into oscillation.

**Figure 1-29 Setup and Hold Times for D Flip-flop**

The propagation delay from the time the clock changes to the time the  $Q$  output changes is also indicated in Figure 1-29. The propagation delay for a low-to-high change in  $Q$  is  $t_{plh}$ , and for a high-to-low change it is  $t_{phl}$ . Minimum values for  $t_{su}$  and  $t_h$  and maximum values for  $t_{plh}$  and  $t_{phl}$  can be read from manufacturers' data sheets.

The maximum clock frequency for a sequential network depends on several factors. For a network of the form of Figure 1-16, assume that the maximum propagation delay through the combinational network is  $t_{cmax}$  and the maximum propagation delay from the time the clock changes to the flip-flop output changes is  $t_{pmax}$ , where  $t_{pmax}$  is the maximum of  $t_{ph}$  and  $t_{phl}$ . Then the maximum time from the active edge of the clock to the time the change in  $Q$  propagates back to the D flip-flop inputs is  $t_{pmax} + t_{cmax}$ . If the clock period is  $t_{ck}$ , the D inputs must be stable  $t_{su}$  before the end of the clock period. Therefore,

$$t_{pmax} + t_{cmax} \leq t_{ck} - t_{su}$$

and

$$t_{ck} \geq t_{pmax} + t_{cmax} + t_{su}$$

For example, for the network of Figure 1-20, if the maximum gate delay is 15 ns,  $t_{pmax}$  for the flip-flops is 15 ns, and  $t_{su}$  is 5 ns, then

$$t_{ck} \geq 2 \times 15 + 15 + 5 = 50 \text{ ns.}$$

The maximum clock frequency is then  $1/t_{ck} = 20 \text{ MHz}$ . Note that the inverter is not in the feedback loop.

A hold-time violation could occur if the change in  $Q$  fed back through the combinational network and caused D to change too soon after the clock edge. The hold time is satisfied if

$$t_{pmin} + t_{cmin} \geq t_h$$

When checking for hold-time violations, the worst case occurs when the timing parameters have their minimum values. Since  $t_{pmin} > t_h$  for normal flip-flops, a hold-time violation due to  $Q$  changing does not occur. However, a setup or hold-time violation could occur if the X input to the network changes too close to the active edge of the clock.

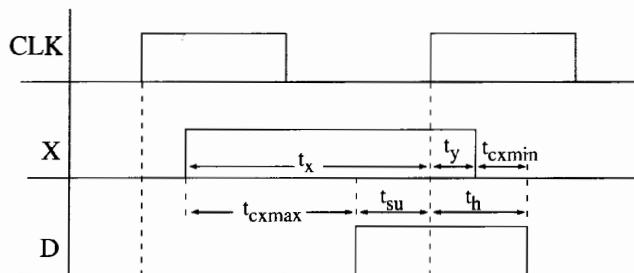
When the X input to a sequential network changes, we must make sure that the input change propagates to the flip-flop inputs such that the setup time is satisfied before the active edge of the clock. If X changes at time  $t_x$  before the active edge of the clock (see Figure 1-30), then the setup time is satisfied if

$$t_x \geq t_{cxmax} + t_{su}$$

where  $t_{cxmax}$  is the maximum propagation delay from X to the flip-flop input. In order to satisfy the hold time, we must make sure that X does not change too soon after the clock. If X changes at time  $t_y$  after the active edge of the clock, then the hold time is satisfied if

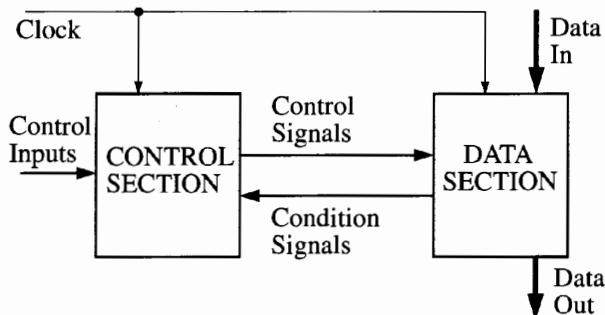
$$t_y \geq t_h - t_{cxmin}$$

where  $t_{cxmin}$  is the minimum propagation delay from X to the flip-flop input. If  $t_y$  is negative, X can change before the active clock edge and still satisfy the hold time.

**Figure 1-30 Setup and Hold Timing for Changes in X**

## 1.12 SYNCHRONOUS DESIGN

One of the most commonly used digital design techniques is *synchronous design*. This type of design uses a clock to synchronize the operation of all flip-flops, registers, and counters in the system. All state changes will occur immediately following the active edge of the clock. The clock period must be long enough so that all flip-flop and register inputs will have time to stabilize before the next active edge of the clock.

**Figure 1-31 Synchronous Digital System**

A typical digital system can be divided into a control section and a data section, as shown in Figure 1-31. A common clock synchronizes the operation of the control and data sections. The data section may contain data registers, arithmetic units, counters, etc. The control section is a sequential machine that generates control signals to control the operation of the data section. For example, if the data section contains a shift register, the control section may generate signals  $Ld$  and  $Sh$ , which determine when the register is to be loaded and when it is to be shifted. The data section may generate condition signals that effect the control sequence. For example, if a data operation produces an arithmetic overflow, then the data section might generate a condition signal  $V$  to indicate an overflow.

**Figure 1-32 Timing Chart for System with Falling-Edge Devices**

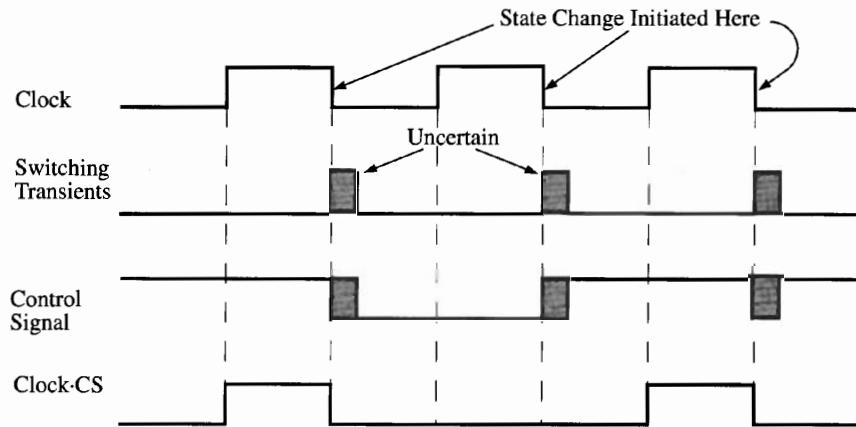


Figure 1-32 illustrates the operation of a digital system, which uses devices that change state on the falling edge of the clock. Several flip-flops may change state in response to this falling edge. The time at which each flip-flop changes state is determined by the propagation delay for that flip-flop. The changes in flip-flop states in the control section will propagate through the combinational network that generates the control signals, and some of the control signals may change as a result. The exact times at which the control signals change depend on the propagation delays in the gate networks that generate the signals as well as the flip-flop delays. Thus, after the falling edge of the clock, there is a period of uncertainty during which control signals may change. Glitches and spikes may occur in the control signals due to hazards. Furthermore, when signals are changing in one part of the circuit, noise may be induced in another part of the circuit. As indicated by the cross-hatching in Figure 1-32, there is a time interval after each falling edge of the clock in which there may be noise in a control signal ( $CS$ ), and the exact time at which the control signal changes is not known.

If we want a device in the data section to change state on the falling edge of the clock only if the control signal  $CS = 1$ , we can AND the clock with  $CS$ , as shown in Figure 1-33(a). The  $CLK$  input to the device will be a clean signal, and except for a small delay in the AND gate, the transitions will occur in synchronization with the clock. The  $CLK$  signal is clean because the clock is 0 during the time interval in which the switching transients occur in  $CS$ .

**Figure 1-33 Gated Control Signal**

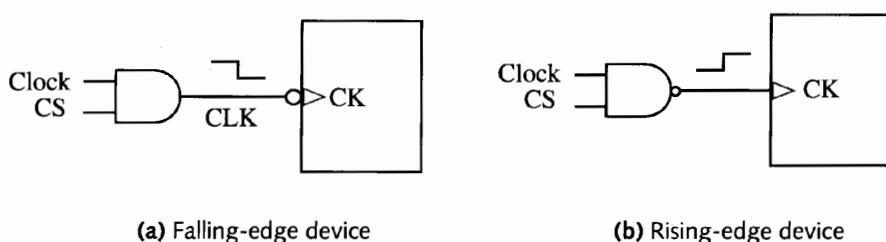
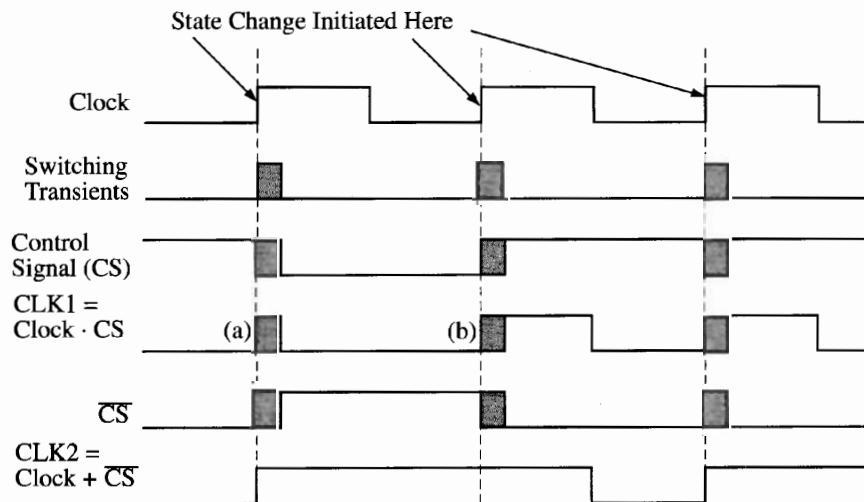
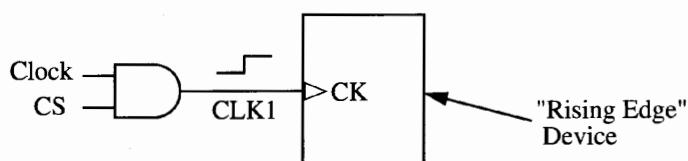


Figure 1-34 illustrates the operation of a digital system that uses devices that change state on the rising edge of the clock. In this case, the switching transients that result in noise and uncertainty will occur following the rising edge of the clock. The cross-hatching indicates the time interval in which the control signal  $CS$  may be noisy. If we want a device to change state on the rising edge of the clock when  $CS = 1$ , it is tempting to AND the clock with  $CS$ , as shown in Figure 1-35. The resulting signal, which goes to the  $CK$  input of the device, may be noisy and timed incorrectly. In particular, the  $CLK1$  pulse at (a) will be short and noisy. It may be too short to trigger the device, or it may be noisy and trigger the device more than once. In general, it will be out of sync with the clock, because the control signal does not change until after some of the flip-flops in the control network have changed state. The rising edge of the pulse at (b) again will be out of sync with the clock, and it may be noisy. But even worse, the device will trigger near point (b) when it should not trigger there at all. Since  $CS = 0$  at the time of the rising edge of the clock, triggering should not occur until the next rising edge, when  $CS = 1$ .

**Figure 1-34 Timing Chart for System with Rising-Edge Devices**



**Figure 1-35 Incorrect Design for Rising-Edge Device**

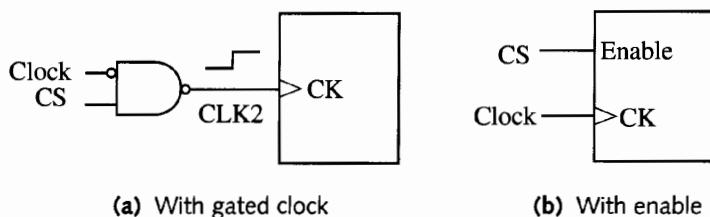


If we move the bubble in Figure 1-33(a) from the device input to the gate output (see Figure 1-33(b)), the timing will be the same as in Figure 1-32. We now have a rising-edge device, but it will trigger on the falling edge of the system clock. To get around this problem, we can invert the clock, as indicated by the added bubble in Figure 1-36(a). The *CK* input to the device is then

$$CLK2 = (CS \cdot \text{clock}')' = CS' + \text{clock}$$

As shown in Figure 1-34, the *CLK2* signal will be free of noise, and when *CS* = 1, *CLK2* will change from 0 to 1 at the same time as the clock.

**Figure 1-36 Correct Design for Rising-Edge Device**



Many registers, counters, and other devices used in synchronous systems have an enable input (see Figure 1-36(b)). When enable = 1, the device changes state in response to the clock, and when enable = 0, no state change occurs. Use of the enable input eliminates the need for a gate on the clock input, and the associated timing problems are avoided.

In summary, synchronous design is based on the following principles:

- Method: All clock inputs to flip-flops, registers, counters, etc., are driven directly from the system clock or from the clock ANDed with a control signal.
- Result: All state changes occur immediately following the active edge of the clock signal.
- Advantage: All switching transients, switching noise, etc., occur between clock pulses and have no effect on system performance.

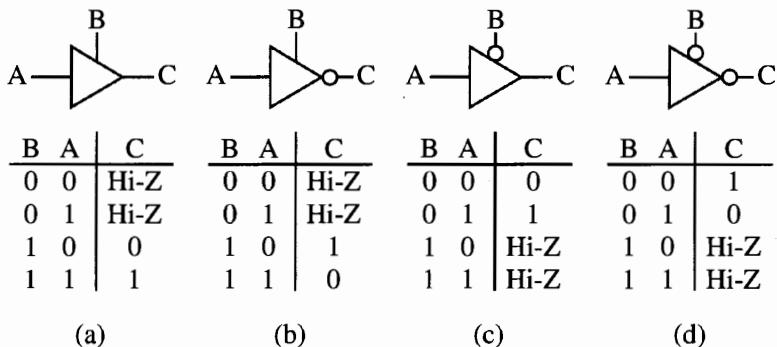
Asynchronous design (see *Fundamentals of Logic Design*, Chapters 23–27) is generally more difficult than synchronous design. Since there is no clock to synchronize the state changes, problems may arise when several state variables must change at the same time. A race occurs if the final state depends on the order in which the variables change. Asynchronous design requires special techniques to eliminate problems with races and hazards. On the other hand, synchronous design has several disadvantages: In high-speed circuits where the propagation delay in the wiring is significant, the clock signal must be carefully routed so that it reaches all the clock inputs at essentially the same time. The maximum clock rate is determined by the worst-case delay of the longest path. The system inputs may not be synchronized with the clock, so use of synchronizers may be required.

## 1.13 TRISTATE LOGIC AND BUSSES

In digital systems, transferring data back and forth between several system components is often necessary. In this section, we introduce the concept of tristate buffers and show how tristate busses can be used to facilitate data transfers between registers.

Figure 1-37 shows four kinds of tristate buffers.  $B$  is a control input used to enable or disable the buffer output. When a buffer is enabled, the output ( $C$ ) is equal to the input ( $A$ ) or its complement. When a buffer is disabled, the output is in a high-impedance, or hi-Z, state, which is equivalent to an open circuit. Normally, if we connect the outputs of two gates or flip-flops together, the circuit will not operate properly. However, we can connect two tristate buffer outputs, provided that only one output is enabled at a time.

Figure 1-37 Four Kinds of Tristate Buffers



(a)

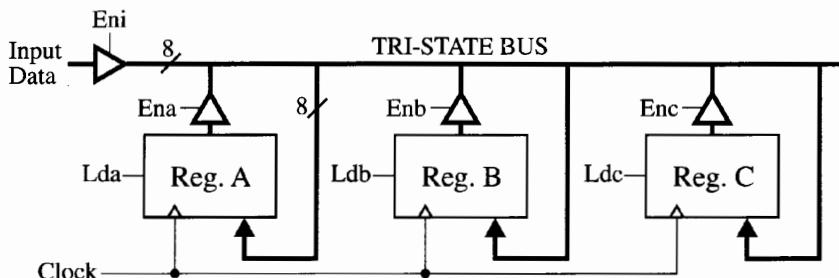
(b)

(c)

(d)

Figure 1-38 shows a system with three registers connected to a tristate bus. Each register is 8 bits wide, and the bus consists of 8 wires connected in parallel. Each tristate buffer symbol in the figure represents 8 buffers operating in parallel with a common enable input. Only one group of buffers is enabled at a time. For example, if  $Eni = 1$ , the register  $B$  output is driven onto the bus. The data on the bus is routed to the inputs of register  $A$ , register  $B$ , and register  $C$ . However, data is loaded into a register only when its load input is 1 and the register is clocked. Thus, if  $Eni = Ldc = 1$ , the data in register  $B$  will be copied into register  $C$  when the active edge of the clock occurs. If  $Eni = Lda = Ldb = 1$ , the input data will be loaded in registers  $A$  and  $B$  when the registers are clocked.

Figure 1-38 Data Transfer Using Tristate Bus



## Problems

- 1.1** Write out the truth table for the following equation.

$$F = (A \oplus B) \cdot C + A' \cdot (B' \oplus C)$$

- 1.2** A full subtracter computes the difference of three inputs  $X$ ,  $Y$ , and  $B_{in}$ , where  $Diff = X - Y - B_{in}$ . When  $X < (Y + B_{in})$ , the borrow output  $B_{out}$  is set. Fill in the truth table for the subtracter and derive the sum-of-products and product-of-sums equations for  $Diff$  and  $B_{out}$ .

- 1.3** Simplify  $Z$  using a 4-variable map with map-entered variables.  $ABCD$  represents the state of a control network. Assume that the network can never be in state 0100, 0001, or 1001.

$$Z = BC'DE + ACDF' + ABCD'F' + ABC'D'G + B'CD + ABC'D'H'$$

- 1.4** For the following functions, find the minimum sum of products using 4-variable maps with map-entered variables. In (a) and (b),  $m_i$  represents a minterm of variables  $A$ ,  $B$ ,  $C$ , and  $D$ .

(a)  $F(A, B, C, D, E) = \sum m(0, 4, 6, 13, 14) + \sum d(2, 9) + E(m_1 + m_{12})$

(b)  $Z(A, B, C, D, E, F, G) = \sum m(2, 5, 6, 9) + \sum d(1, 3, 4, 13, 14) + E(m_{11} + m_{12}) + F(m_{10}) + G(m_0)$

(c)  $H = A'B'CDF' + A'CD + A'B'CD'E + BCDF'$

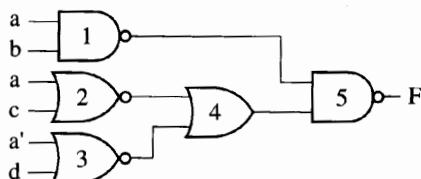
(d)  $G = C'E'F + DEF + AD'E'F' + BC'E'F + AD'EF'$

*Hint:* Which variables should be used for the map sides and which variables should be entered into the map?

### 1.5

- (a) Find all the static hazards in the following network. For each hazard, specify the values of the input variables and which variable is changing when the hazard occurs. For one of the hazards, specify the order in which the gate outputs must change.

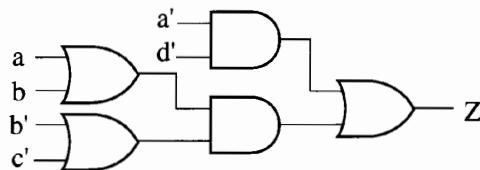
- (b) Design a NAND-gate network that is free of static hazards to realize the same function.



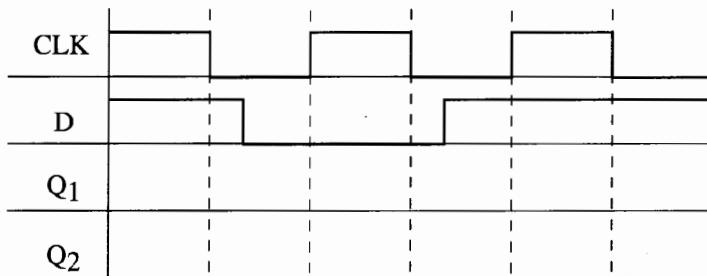
### 1.6

- (a) Find all the static hazards in the following network. State the condition under which each hazard can occur.

- (b) Redesign the network so that it is free of static hazards. Use gates with at most three inputs.



- 1.7** Construct a clocked D flip-flop, triggered on the rising edge of  $CLK$ , using two transparent D latches and any necessary gates. Complete the following timing diagram, where  $Q_1$  and  $Q_2$  are latch outputs. Verify that the flip-flop output changes to  $D$  after the rising edge of the clock.



- 1.8** A synchronous sequential network has one input and one output. If the input sequence 0101 or 0110 occurs, an output of two successive 1s will occur. The first of these 1s should occur coincident with the last input of the 0101 or 0110 sequence. The network should reset when the second 1 output occurs. For example,

input sequence:  $X = 010011101010101101 \dots$

output sequence:  $Z = 000000000011000011 \dots$

- (a) Derive a Mealy state graph and table with a minimum number of states (6 states).  
 (b) Try to choose a good state assignment. Realize the network using J-K flip-flops and NAND gates. Repeat using NOR gates. (Work this part by hand.)  
 (c) Check your answer to (b) using the *LogicAid* program. Also use the program to find the NAND solution for two other state assignments.

- 1.9** A sequential network has one input ( $X$ ) and two outputs ( $Z_1$  and  $Z_2$ ). An output  $Z_1 = 1$  occurs every time the input sequence 010 is completed provided that the sequence 100 has never occurred. An output  $Z_2 = 1$  occurs every time the input sequence 100 is completed. Note that once a  $Z_2 = 1$  output has occurred,  $Z_1 = 1$  can never occur, but *not* vice versa.

- (a) Derive a Mealy state graph and table with a minimum number of states (8 states).  
 (b) Try to choose a good state assignment. Realize the network using J-K flip-flops and NAND gates. Repeat using NOR gates. (Work this part by hand.)  
 (c) Check your answer to (b) using the *LogicAid* program. Also use the program to find the NAND solution for two other state assignments.

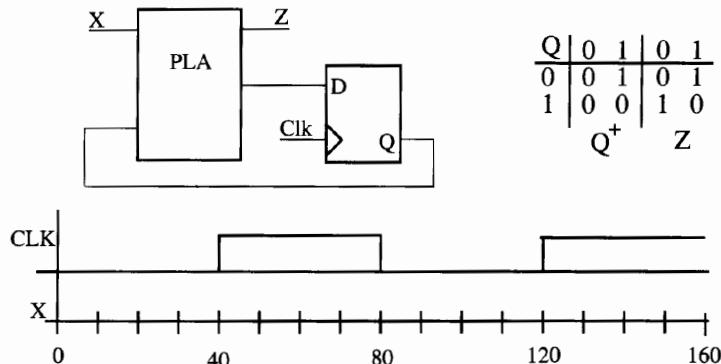
**1.10** A sequential network has one input ( $X$ ) and two outputs ( $S$  and  $V$ ).  $X$  represents a 4-bit binary number  $N$ , which is input least significant bit first.  $S$  represents a 4-bit binary number equal to  $N + 2$ , which is output least significant bit first. At the time the fourth input occurs,  $V = 1$  if  $N + 2$  is too large to be represented by 4 bits; otherwise,  $V = 0$ . The value of  $S$  should be the proper value, not a don't care, in both cases. The network always resets after the fourth bit of  $X$  is received.

- (a) Derive a Mealy state graph and table with a minimum number of states (6 states).
- (b) Try to choose a good state assignment. Realize the network using J-K flip-flops and NAND gates. Repeat using NOR gates. (Work this part by hand.)
- (c) Check your answer to (b) using the *LogicAid* program. Also use the program to find the NAND solution for two other state assignments.

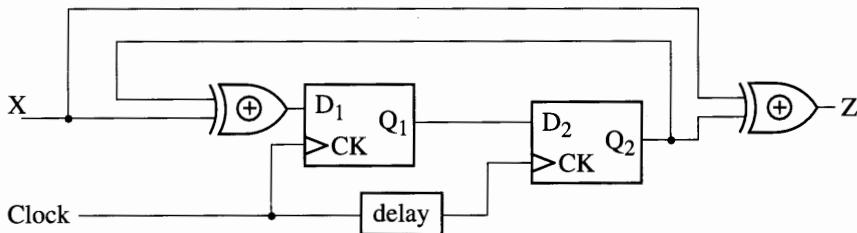
**1.11** A sequential network has one input ( $X$ ) and two outputs ( $D$  and  $B$ ).  $X$  represents a 4-bit binary number  $N$ , which is input least significant bit first.  $D$  represents a 4-bit binary number equal to  $N - 2$ , which is output least significant bit first. At the time the fourth input occurs,  $B = 1$  if  $N - 2$  is negative; otherwise,  $B = 0$ . The network always resets after the fourth bit of  $X$  is received.

- (a) Derive a Mealy state graph and table with a minimum number of states (6 states).
- (b) Try to choose a good state assignment. Realize the network using J-K flip-flops and NAND gates. Repeat using NOR gates. (Work this part by hand.)
- (c) Check your answer to (b) using the *LogicAid* program. Also use the program to find the NAND solution for two other state assignments.

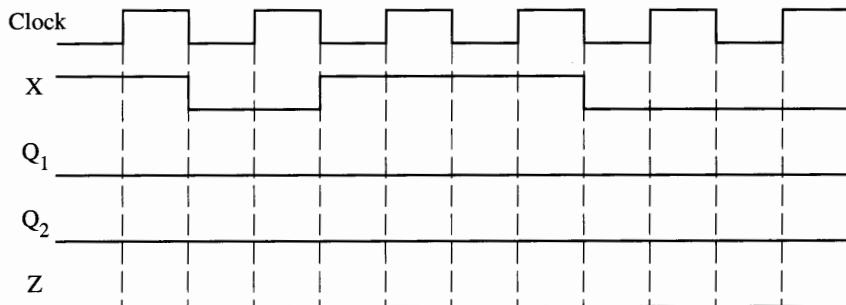
**1.12** A sequential network has the following form. The delay through the combinational network is in the range  $5 \leq t_c \leq 20$  ns. The propagation delay from the rising edge of the clock to the change in the flip-flop output is in the range  $5 \leq t_p \leq 10$  ns. The required setup and hold times for the flip-flop are  $t_{su} = 10$  ns and  $t_h = 5$  ns. Indicate on the diagram the times at which  $X$  is allowed to change.



**1.13** A Mealy sequential network is implemented using the network shown below. Assume that if the input  $X$  changes, it changes at the same time as the **falling** edge of the clock. Assume the following delays: XOR gate, 5 to 15 ns; flip-flop propagation delay, 5 to 15 ns; setup time, 10 ns; hold time, 5 ns. Initially assume that the “delay” is 0 ns.



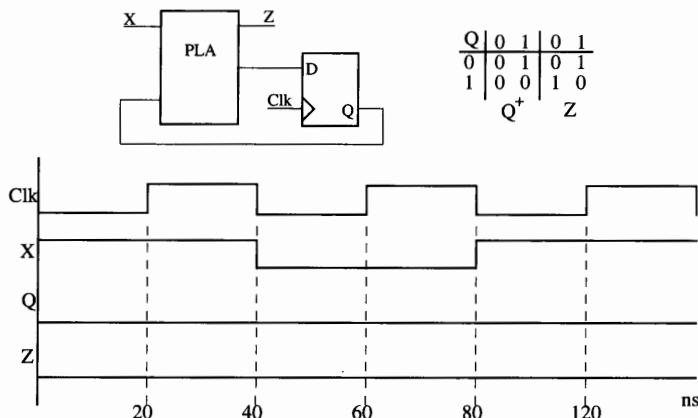
- (a) Determine the maximum clock rate for proper synchronous operation.
- (b) Assume a clock period of 100 ns. What is the maximum value that “delay” can have and still achieve proper synchronous operation?
- (c) Complete the following timing diagram. Indicate the proper times to read the output ( $Z$ ). Assume that “delay” is 0 ns and that the propagation delay for the flip-flop and XOR gate has a nominal value of 10 ns.



**1.14** A sequential network consists of a PLA and a D flip-flop, as shown.

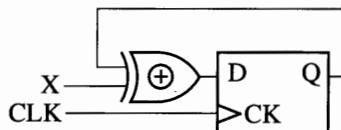
- (a) Complete the timing diagram assuming that the propagation delay for the PLA is in the range 5 to 10 ns, and the propagation delay from clock to output of the D flip-flop is 5 to 10 ns. Use cross-hatching on your timing diagram to indicate the intervals in which  $Q$  and  $Z$  can change, taking the range of propagation delays into account.

- (b) Assuming that  $X$  always changes at the same time as the falling edge of the clock, what is the maximum setup and hold time specification that the flip-flop can have and still maintain proper operation of the network?



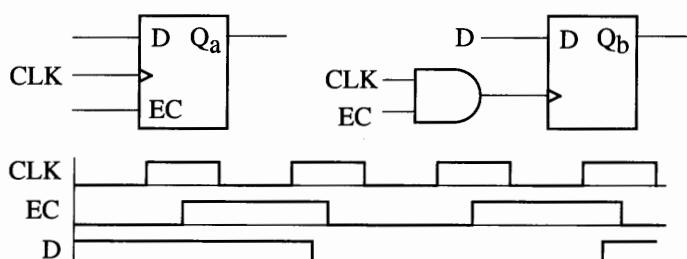
**1.15** A D flip-flop has a setup time of 4 ns, a hold time of 2 ns, and a propagation delay from the rising edge of the clock to the change in flip-flop output in the range of 6 to 12 ns. The XOR gate delay is in the range of 1 to 8 ns.

- (a) What is the minimum clock period for proper operation of the following network?  
 (b) What is the earliest time after the rising clock edge that  $X$  is allowed to change?



**1.16**

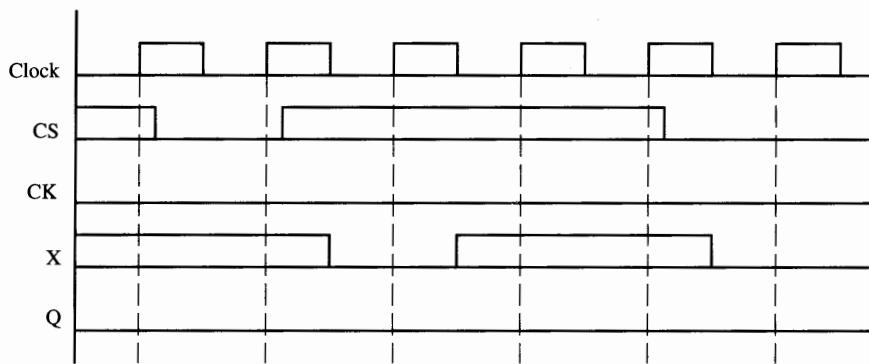
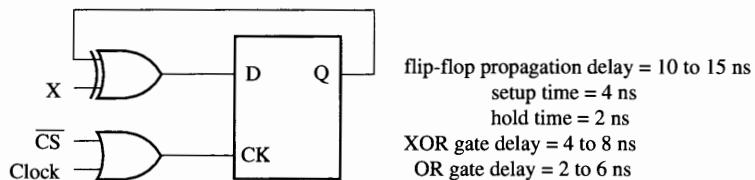
- (a) Do the following two networks have essentially the same timing?  
 (b) Draw the timing for  $Q_a$  and  $Q_b$  given the timing diagram.  
 (c) If your answer to (a) is no, show what change(s) should be made in the second network so that the two networks have essentially the same timing (do not change the flip-flop).



**1.17**

Assume that CS (and also  $\overline{CS}$ ) change 2 ns after the rising edge of the clock.

- (a) Plot CK and Q on the timing diagram. A precise plot is not required; just show the relative times at which the signals change.
- (b) If X changes at the falling edge of Clock, as shown, what is the maximum clock frequency?
- (c) With respect to the rising edge of Clock, what is the earliest that X can change and still satisfy the hold-time requirement?



# CHAPTER 2

## INTRODUCTION TO VHDL

---

As integrated circuit technology has improved to allow more and more components on a chip, digital systems have continued to grow in complexity. As digital systems have become more complex, detailed design of the systems at the gate and flip-flop level has become very tedious and time consuming. For this reason, use of hardware description languages in the digital design process continues to grow in importance. A hardware description language allows a digital system to be designed and debugged at a higher level before conversion to the gate and flip-flop level. Use of synthesis computer-aided design tools to do this conversion is becoming more widespread. This is analogous to writing software programs in a high-level language such as C and then using a compiler to convert the programs to machine language. The two most popular hardware description languages are VHDL and Verilog.

VHDL is a hardware description language used to describe the behavior and structure of digital systems. The acronym VHDL stands for VHSIC Hardware Description Language, and VHSIC in turn stands for Very High Speed Integrated Circuit. However, VHDL is a general-purpose hardware description language that can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. VHDL was originally developed for the military to allow a uniform method for specifying digital systems. The VHDL language has since become an IEEE standard, and it is widely used in industry.

VHDL can describe a digital system at several different levels—behavioral, data flow, and structural. For example, a binary adder could be described at the behavioral level in terms of its function of adding two binary numbers, without giving any implementation details. The same adder could be described at the data flow level by giving the logic equations for the adder. Finally, the adder could be described at the structural level by specifying the interconnections of the gates that comprise the adder.

VHDL leads naturally to a top-down design methodology, in which the system is first specified at a high level and tested using a simulator. After the system is debugged at this level, the design can gradually be refined, eventually leading to a structural description closely related to the actual hardware implementation. VHDL was designed to be technology independent. If a design is described in VHDL and implemented in today's technology, the same VHDL description could be used as a starting point for a design in some future technology.

In this chapter, we describe the basic features of VHDL and illustrate how we can describe simple combinational and sequential networks using VHDL. We will use VHDL in later chapters to design more complex digital systems. In Chapter 8, we introduce some of the more advanced features of VHDL, and we discuss the use of CAD software tools for automatic synthesis from VHDL descriptions.

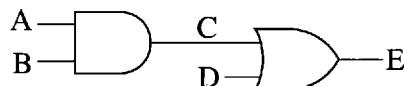
## 2.1 VHDL DESCRIPTION OF COMBINATIONAL NETWORKS

We start by describing a simple gate network in VHDL. If each gate in the network of Figure 2-1 has a 5-ns propagation delay, the network can be described as follows:

```
C <= A and B after 5 ns;
E <= C or D after 5 ns;
```

where  $A$ ,  $B$ ,  $C$ ,  $D$ , and  $E$  are signals. A signal in VHDL usually corresponds to a signal in a physical system. The symbol " $<=$ " is the signal assignment operator, which indicates the value computed on the right side is assigned to the signal on the left side. When these statements are simulated, the first statement will be evaluated any time  $A$  or  $B$  changes, and the second statement will be evaluated any time  $C$  or  $D$  changes. Suppose that initially  $A = 1$ , and  $B = C = D = E = 0$ . If  $B$  changes to 1 at time 0,  $C$  will change to 1 at time = 5 ns. Then  $E$  will change to 1 at time = 10 ns.

**Figure 2-1** Gate Network



VHDL signal assignment statements, like the ones in the preceding example, are called *concurrent statements* when they are not contained in a VHDL process or block. The VHDL simulator monitors the right-hand side of each concurrent statement, and any time a signal changes, the expression on the right-hand side is immediately re-evaluated. The new value is assigned to the signal on the left-hand side after an appropriate delay.

When we initially describe a network, we may not be concerned about propagation delays. If we write

```
C <= A and B;
E <= C or D;
```

this implies that the propagation delays are 0 ns. In this case, the simulator will assume an infinitesimal delay referred to as  $\Delta$  (delta). For this example, if  $B$  is changed to 1 at time = 0, then  $C$  will change at time  $0 + \Delta$  and  $E$  will change at time  $0 + 2\Delta$ .

Unlike a sequential program, the order of the preceding statements is unimportant. If we write

```
E <= C or D;
C <= A and B;
```

the simulation results would be exactly the same as before. Even if a VHDL program has no explicit loops, concurrent statements may execute repeatedly as if they were in a loop. The VHDL statement

```
CLK <= not CLK after 10 ns;
```

will generate a clock waveform with a half-period of 10 ns. If *CLK* is initially '0', it will change to '1' after 10 ns. When *CLK* changes to '1', the statement will be executed again, and *CLK* will change back to '0' after another 10 ns. This process will continue indefinitely. On the other hand, the concurrent statement

```
CLK <= not CLK;
```

will cause a run-time error during simulation. Since there is 0 delay, the value of *CLK* will change at times  $0 + \Delta$ ,  $0 + 2\Delta$ ,  $0 + 3\Delta$ , etc., and real time will never advance.

In general, VHDL is not case sensitive; that is, capital and lowercase letters are treated the same by the compiler and simulator. Thus the statements

```
Clk <= NOT clk After 10 NS;
and CLK <= not CLK after 10 ns;
```

are treated exactly the same. Signal names and other VHDL identifiers may contain letters, numbers, and the underscore character (\_). An identifier must start with a letter, and it cannot end with an underscore. Thus C123, and ab\_23 are legal identifiers, but 1ABC and ABC\_ are not. Every VHDL statement must be terminated with a semicolon.

## Entity-Architecture Pairs

To write a complete VHDL program, we must declare all the input and output signals and specify the type of each signal. As an example, we will describe the full adder of Figure 1-2(a). A complete description must include an entity declaration and an architecture declaration. The entity declaration specifies the inputs and outputs of the adder module:

```
entity FullAdder is
  port (X, Y, Cin: in bit;           -- Inputs
        Cout, Sum: out bit);         -- Outputs
end FullAdder;
```

The words **entity**, **is**, **port**, **in**, **out**, and **end** are reserved words (or keywords), which have a special meaning to the VHDL compiler. In this text, we will put all reserved words in boldface type. Anything that follows a double dash (--) is a VHDL comment. The port declaration specifies that *X*, *Y*, and *Cin* are input signals of type bit and that *Cout* and *Sum* are output signals of type bit. Each signal in this example is of type bit, which means it can assume only values of '0' or '1'.

The operation of the full adder is specified by an architecture declaration:

```
architecture Equations of FullAdder is
begin                                     -- Concurrent Assignments
    Sum  <= X xor Y xor Cin after 10 ns;
    Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

In this example, the architecture name (Equations) is arbitrary, but the entity name (FullAdder) must match the name used in the associated entity declaration. The VHDL assignment statements for *Sum* and *Cout* represent the logic equations for the full adder (Equations 1-31 and 1-22). Several other architectural descriptions, such as a truth table or an interconnection of gates, could have been used instead. In the *Cout* equation, parentheses are required around (*X and Y*), since VHDL does not specify an order of precedence for the logic operators.

When we describe a system in VHDL, we must specify an entity and an architecture at the top level, and also specify an entity and architecture for each of the component modules that are part of the system (see Figure 2-2). Each entity declaration includes a list of interface signals that can be used to connect to other modules or to the outside world. We will use entity declarations of the form

```
entity entity-name is
    [port (interface-signal-declaration);]
end [entity] [entity-name];
```

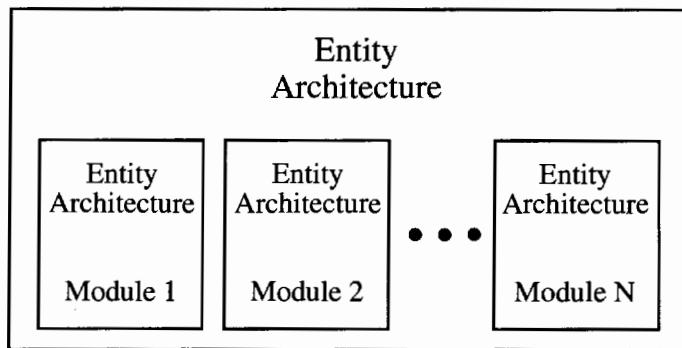
The items enclosed in brackets are optional. The interface-signal-declaration normally has the following form:

```
list-of-interface-signals: mode type [: initial-value]
{; list-of-interface-signals: mode type [: initial-value]}
```

The curly brackets indicate zero or more repetitions of the enclosed clause. Input signals are of mode **in**, output signals are of mode **out**, and bidirectional signals are of mode **inout**. So far, we have used only type bit; other types are described in Sections 2.6 and 2.7. The optional initial value is used to initialize the signals on the associated list; otherwise, the default initial value is used for the specified type. For example, the port declaration

```
port (A, B: in integer := 2; C, D: out bit);
```

indicates that *A* and *B* are input signals of type integer, which are initially set to 2, and *C* and *D* are output signals of type bit, which are initialized by default to '0'.

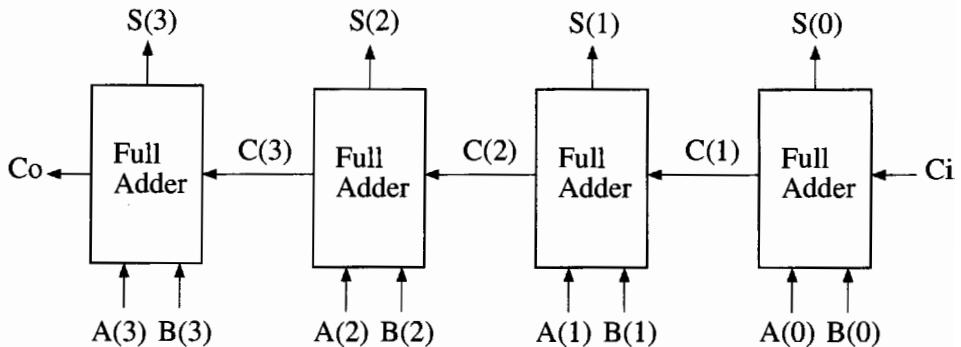
**Figure 2-2** VHDL Program Structure

Associated with each entity is one or more architecture declarations of the form

```
architecture architecture-name of entity-name is
    [declarations]
begin
    architecture body
end [architecture] [architecture-name];
```

### Four-bit Full Adder

Next we will show how to use the FullAdder module defined earlier as a component in a system that consists of four full adders connected to form a 4-bit binary adder (see Figure 2-3). We first declare the 4-bit adder as an entity (see Figure 2-4). Since the inputs and the sum output are 4 bits wide, we declare them as bit\_vectors, which are dimensioned 3 **downto** 0. (We could have used the range 1 **to** 4 instead).

**Figure 2-3** Four-bit Binary Adder

Next we specify the FullAdder as a component within the architecture of Adder4 (Figure 2-4). The component specification is very similar to the entity declaration for the full adder, and the input and output port signals correspond to those declared for the full adder. Following the component statement, we declare a 3-bit internal carry signal  $C$ .

In the body of the architecture, we create several instances of the FullAdder component. (In CAD jargon, we “instantiate” four copies of the FullAdder.) Each copy of FullAdder has a name (such as FA0) and a port map. The signal names following the port map correspond one-to-one with the signals in the component port. Thus,  $A(0)$ ,  $B(0)$ , and  $Ci$  correspond to the inputs  $X$ ,  $Y$ , and  $Cin$ , respectively.  $C(1)$  and  $S(0)$  correspond to the  $Cout$  and  $Sum$  outputs.

**Figure 2-4 Structural Description of 4-bit Adder**

```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
        S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;

architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit;           -- Inputs
        Cout, Sum: out bit);         -- Outputs
end component;
signal C: bit_vector(3 downto 1);
begin
  -- instantiate four copies of the FullAdder
  FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
  FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
  FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
  FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

In preparation for simulation, we can place the entity and architecture for the FullAdder and for Adder4 together in one file and compile. Alternatively, we could compile the FullAdder separately and place the resulting code in a library that is linked in when we compile Adder4.

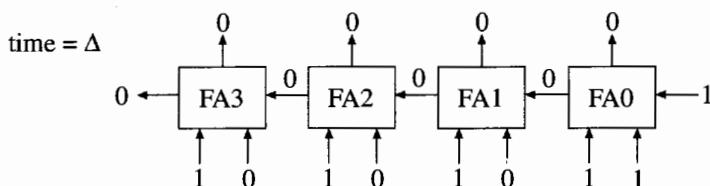
All the simulation examples in this text use the V-System/Windows simulator from Model Tech. Most other VHDL simulators use similar command files and can produce output in a similar format. We use the following simulator commands to test Adder4:

```
list A B Co C Ci S -- put these signals on the output list
force A 1111      -- set the A inputs to 1111
force B 0001      -- set the B inputs to 0001
force Ci 1        -- set Ci to 1
run 50            -- run the simulation for 50 ns
force Ci 0
force A 0101
force B 1110
run 50
```

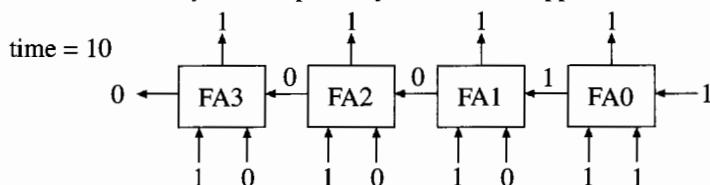
We have chosen to run the simulation for 50 ns, since this is long enough for the carry to propagate through all the full adders. The simulation results for the above command list are

ns	delta	a	b	co	c	ci	s
0	+0	0000	0000	0	000	0	0000
0	+1	1111	0001	0	000	1	0000
10	+0	1111	0001	0	001	1	1111
20	+0	1111	0001	0	011	1	1101
30	+0	1111	0001	0	111	1	1001
40	+0	1111	0001	1	111	1	0001
50	+0	0101	1110	1	111	0	0001
60	+0	0101	1110	1	110	0	0101
70	+0	0101	1110	1	100	0	0111
80	+0	0101	1110	1	100	0	0011

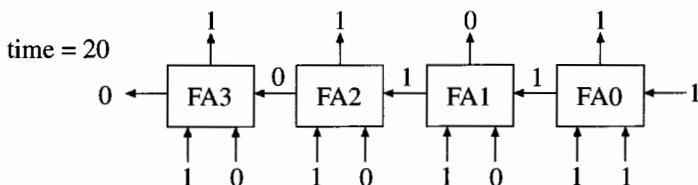
The listing shows how the carry propagates one position every 10 ns. The full adder inputs change at time =  $\Delta$ :



The sum and carry are computed by each FA and appear at the FA outputs 10 ns later:



Since the inputs to FA1 have changed, the outputs change 10 ns later:



The final simulation results are

$$1111 + 0001 + 1 = 0001 \text{ with a carry of } 1 \text{ (at time = 40 ns)} \text{ and}$$

$$0101 + 1110 + 0 = 0011 \text{ with a carry of } 1 \text{ (at time = 80 ns)}$$

The simulation stops at 80 ns, since no further changes occur after that time.

## 2.2 MODELING FLIP-FLOPS USING VHDL PROCESSES

A common way of modeling sequential logic in VHDL uses a *process*. A process may have the form

```
process(sensitivity-list)
begin
    sequential-statements
end process;
```

Whenever one of the signals in the *sensitivity list* changes, the sequential statements in the process body are executed in sequence one time.

The following example illustrates the difference in the way sequential and concurrent statements are executed. A VHDL program has signals *A*, *B*, *C*, and *D* of type integer. The signals are initialized to *A* = 1, *B* = 2, *C* = 3, and *D* = 0. The program contains the following concurrent statements:

```
A <= B;      -- statement 1
B <= C;      -- statement 2
C <= D;      -- statement 3
```

Assume that *D* changes to 4 at time = 10. The following sequence of events then occurs: Since *D* has changed, statement 3 executes, and *C* is changed to 4 at time  $10 + \Delta$ . Next, since *C* has changed, statement 2 executes, and *B* is updated at time  $10 + 2\Delta$ . Then the change in *B* triggers execution of statement 1, and *A* is updated at time  $10 + 3\Delta$ . Since *A* does not appear on the right-hand side of any statement, no further execution is triggered.

time	delta	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	
0	+0	1	2	3	0	
10	+0	1	2	3	4	(statement 3 executes first)
10	+1	1	2	4	4	(then statement 2 executes)
10	+2	1	4	4	4	(then statement 1 executes)
10	+3	4	4	4	4	(no further execution occurs)

Now consider a program with the same statements placed in a process:

```
process (B, C, D)
begin
    A <= B;      -- statement 1
    B <= C;      -- statement 2
    C <= D;      -- statement 3
end process;
```

Assume that *A*, *B*, *C*, and *D* are initialized as before and *D* changes to 4 at time = 10. Since *D* has changed, and *D* is on the sensitivity list, the process begins execution. Statements 1, 2, and 3 are executed in sequence; then the process goes back to the top and waits until a signal on the sensitivity list changes. Execution of the three statements takes place

instantaneously at time = 10; not even delta time is required to execute the statements. However, since  $A$ ,  $B$ , and  $C$  are signals, their values are not updated until time  $10 + \Delta$ . Therefore, the old values of  $B$ ,  $C$ , and  $D$  are used when the statements are executed. Signals  $A$ ,  $B$ , and  $C$  will change value at time  $10 + \Delta$ , so the process will execute again. As a result,  $A$  and  $B$  will change at time  $10 + 2\Delta$ , and the process will execute a third time. The sequence of events is summarized as follows:

time	delta	A	B	C	D	
0	+0	1	2	3	0	
10	+0	1	2	3	4	(statements 1,2,3 execute; then update A,B,C)
10	+1	2	3	4	4	(statements 1,2,3 execute; then update A,B,C)
10	+2	3	4	4	4	(statements 1,2,3 execute; then update A,B,C)
10	+3	4	4	4	4	(no further execution occurs)

This example shows how signal assignment statements can be used as sequential statements in a process. Another commonly used sequential statement is the **if** statement. The basic **if** statement has the form

```
if condition then
    sequential statements1
else sequential statements2
end if;
```

The condition is a Boolean expression, which evaluates to TRUE or FALSE. If it is TRUE, sequential statements1 are executed; otherwise, sequential statements2 are executed.

Next we use a VHDL process to model a simple D flip-flop (Figure 1-10), which changes state on the rising edge of the clock input. The signal  $QN$  represents the  $Q'$  output of the flip-flop. In the entity declaration (see Figure 2-5),  $QN$  is explicitly initialized to '1', since it must be the complement of  $Q$ , and bit signals are initialized to '0' by default. VHDL requires that bit values such as '0' and '1' be enclosed in single quotes. The architecture name, SIMPLE, is an arbitrary choice. Since the flip-flop can change state only when the clock changes, we define a process that is executed only when  $CLK$  changes. Thus  $CLK$  is the only signal in the sensitivity list. The clock signal is also tested within the process, and if  $CLK = '1'$ , this means that a rising edge has just occurred on  $CLK$ . In this case,  $Q$  is set equal to  $D$ , and  $QN$  is set to the complement of  $D$ . The 10-ns delay represents the propagation delay between the time the clock changes and the flip-flop outputs change.

**Figure 2-5 D Flip-flop Model**

```

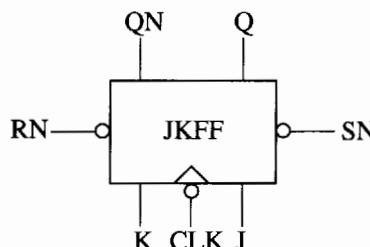
entity DFF is
  port (D, CLK: in bit;
        Q: out bit; QN: out bit := '1');
-- initialize QN to '1' since bit signals are initialized to '0' by default
end DFF;

architecture SIMPLE of DFF is
begin
  process (CLK)           -- process is executed when CLK changes
  begin
    if CLK = '1' then      -- rising edge of clock
      Q <= D after 10 ns;
      QN <= not D after 10 ns;
    end if;
  end process;
end SIMPLE;

```

Next, we model a J-K flip-flop (Figure 2-6) that has active-low direct set (*SN*) and reset (*RN*) inputs and changes state on the falling edge of the clock. In this chapter, we have used a suffix *N* to indicate an active-low (negative-logic) signal. For simplicity, we assume that the condition  $SN = RN = 0$  does not occur. Later, we discuss a more complete model that takes this case into account. The VHDL code for the J-K flip-flop is given in Figure 2-7. The next state of the flip-flop is determined by its characteristic equation:

$$Q^+ = JQ' + K'Q$$

**Figure 2-6 J-K Flip-flop**

Since the process is executed whenever *SN*, *RN*, or *CLK* changes, we must determine when the falling edge of *CLK* has occurred. Simply checking for *CLK* = '0' would not work if the process were activated by a change in *SN* or *RN*. Instead, we have used `(elsif CLK = '0' and CLK'event)`. *CLK'event* (read as *CLK* tick event) evaluates to TRUE if *CLK* has just changed value. Thus, `(CLK = '0' and CLK'event)` is TRUE only if a falling edge of *CLK* has just occurred. *CLK'event* is an example of a signal attribute, and attributes are discussed in detail in Section 8.1.

**Figure 2-7 J-K Flip-flop Model**

```

entity JKFF is
  port (SN, RN, J, K, CLK: in bit;                      -- inputs
        Q: inout bit; QN: out bit := '1');                  -- see Note 1
end JKFF;

architecture JKFF1 of JKFF is
begin
  process (SN, RN, CLK)                                     -- see Note 2
  begin
    if RN = '0' then Q<= '0' after 10 ns;                  -- RN=0 will
                                                               clear the FF
    elsif SN = '0' then Q<= '1' after 10 ns;                -- SN=0 will
                                                               set the FF
    elsif CLK = '0' and CLK'event then -- see Note 3
      Q <= (J and not Q) or (not K and Q) after 10 ns;       -- see Note 4
    end if;
  end process;
  QN <= not Q;                                            -- see Note 5
end JKFF1;

```

Note 1: Q is declared as inout (rather than out) because it appears on both the left and right sides of an assignment within the architecture.

Note 2: The flip-flop can change state in response to changes in SN, RN, and CLK, so these 3 signals are in the sensitivity list.

Note 3: The condition (CLK = '0' and CLK'event) is TRUE only if CLK has just changed from '1' to '0'.

Note 4: Characteristic equation that describes behavior of J-K flip-flop.

Note 5: Every time Q changes, QN will be updated. If this statement were placed within the process, the old value of Q would be used instead of the new value.

The preceding example introduces the use of **elsif**, which is an alternative way of writing nested **if** statements. The most general form of the **if** statement is

```

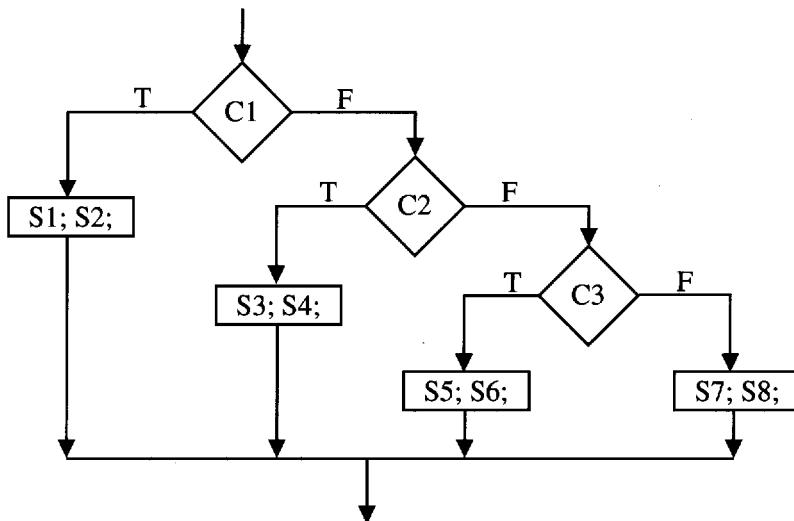
if condition then
  sequential statements
  {elsif condition then sequential statements }
    -- 0 or more elsif clauses may be included
  [else sequential statements]
end if;

```

The curly brackets indicate that any number of **elsif** clauses may be included, and the square brackets indicate that the **else** clause is optional. The example of Figure 2-8 shows how a flowchart can be represented using nested **ifs** or the equivalent using **elsifs**. In this

example, C1, C2, and C3 represent conditions that can be true or false, and S1, S2, . . . , S8 represent sequential statements. Each if requires a corresponding **end if**, but **elsif** does not.

**Figure 2-8 Equivalent Representations of a Flowchart Using Nested Ifs and Elsifs**



```

if (C1) then S1; S2;
  elsif (C2) then S3; S4;
    elsif (C3) then S5; S6;
      else S7; S8;
    end if;
  end if;
end if;

```

```

if (C1) then S1; S2;
  elsif (C2) then S3; S4;
    elsif (C3) then S5; S6;
      else S7; S8;
    end if;
  end if;
end if;

```

## 2.3 VHDL MODELS FOR A MULTIPLEXER

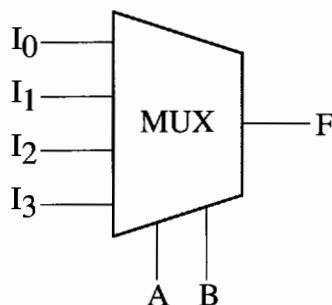
Figure 2-9 shows a 4-to-1 multiplexer (MUX) with four data inputs and two control inputs, A and B. The control inputs select which one of the data inputs is transmitted to the output. The logic equation for the 4-to-1 MUX is

$$F = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

Thus, one way to model the MUX is with the VHDL statement

```
F <= (not A and not B and I0) or (not A and B and I1) or
(A and not B and I2) or (A and B and I3);
```

**Figure 2-9 4-to-1 Multiplexer**



To model the MUX at the behavioral level, we can use a *conditional assignment statement*. This statement has the form

```

signal_name <= expression1 when condition1
    else expression2 when condition2
    ...
    [else expressionN];
    
```

This concurrent statement is executed whenever an event occurs on a signal used in one of the expressions or conditions. If condition1 is true, signal\_name is set equal to the value of expression1, else if condition2 is true, signal\_name is set equal to the value of expression2, etc.

We can also model the 4-to-1 MUX of Figure 2-9 using a *selected signal assignment statement*:

```

F <= I0 when Sel = 0
    else I1 when Sel = 1
    else I2 when Sel = 2
    else I3;
    
```

In the above concurrent statement, *Sel* represents the integer equivalent of a 2-bit binary number with bits *A* and *B*.

If a MUX model is used inside a process, a concurrent statement cannot be used. As an alternative, the MUX can be modeled using a *case* statement:

```

case Sel is
    when 0 => F <= I0;
    when 1 => F <= I1;
    when 2 => F <= I2;
    when 3 => F <= I3;
end case;
    
```

The **case** statement has the general form

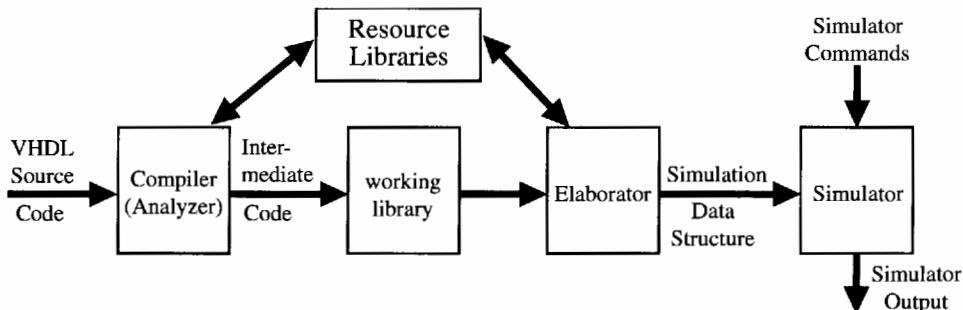
```
case expression is
    when choice1 => sequential statements1
    when choice2 => sequential statements2
    .
    .
    [when others => sequential statements]
end case;
```

The “expression” is evaluated first. If it is equal to “choice1”, then “sequential statements1” are executed; if it is equal to “choice2”, then “sequential statements2” are executed; etc. All possible values of the expression must be included in the choices. If all values are not explicitly given, a “**when others**” clause is required in the case statement.

## 2.4 COMPILE AND SIMULATION OF VHDL CODE

After describing a digital system in VHDL, simulation of the VHDL code is important for two reasons. First, we need to verify the VHDL code correctly implements the intended design; second, we need to verify that the design meets its specifications. Before the VHDL model of a digital system can be simulated, the VHDL code must first be compiled (see Figure 2-10). The VHDL compiler, also called an analyzer, first checks the VHDL source code to see that it conforms to the syntax and semantic rules of VHDL. If there is a syntax error such as a missing semicolon, or if there is a semantic error such as trying to add two signals of incompatible types, the compiler will output an appropriate error message. The compiler also checks to see that references to libraries are correct. If the VHDL code conforms to all the rules, the compiler generates intermediate code, which can be used by a simulator or by a synthesizer. (Synthesis of digital logic from VHDL code is discussed in Chapter 8.)

Figure 2-10 Compilation, Elaboration, and Simulation of VHDL Code



In preparation for simulation, the VHDL intermediate code must be converted to a form that can be used by the simulator. This step is referred to as *elaboration*. During elaboration, ports are created for each instance of a component, memory storage is allocated

for the required signals, the interconnections among the port signals are specified, and a mechanism is established for executing the VHDL processes in the proper sequence. The resulting data structure represents the digital system being simulated. After an initialization phase, the simulator enters the execution phase. The simulator accepts simulation commands, which control the simulation of the digital system and specify the desired simulator output.

As an example of simulation, we trace execution of the VHDL code shown in Figure 2-11. The keyword **transport** specifies the type of delay, as discussed in Section 8.2. During elaboration, a *driver* is created for each signal. Each driver holds the current value of a signal and a queue of future signal values. Each time a signal is scheduled to change in the future, the new value is placed in the queue along with the time at which the change is scheduled.

**Figure 2-11 VHDL Code for Simulation Example**

```
entity simulation_example is
end simulation_example;

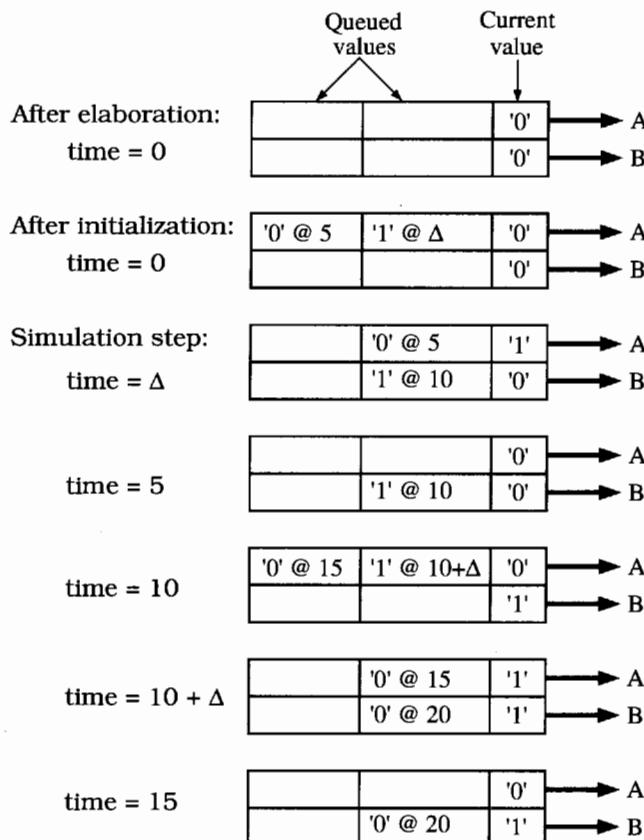
architecture test1 of simulation_example is
  signal A,B: bit;
begin
  P1: process(B)
  begin
    A <= '1';
    A <= transport '0' after 5 ns;
  end process P1;

  P2: process(A)
  begin
    if A = '1' then B <= not B after 10 ns; end if;
  end process P2;
end test1;
```

Figure 2-12 shows the drivers for the signals *A* and *B* as the simulation progresses. After elaboration is finished, each driver holds '0', since this is the default initial value for a bit. When simulation begins, initialization takes place. Both processes are executed simultaneously one time through, and then the processes wait until a signal on the sensitivity list changes. When process *P1* executes in zero time, two changes in *A* are scheduled (*A* changes to '1' at time  $\Delta$  and back to '0' at time = 5 ns). Meanwhile, process *P2* executes, but no change in *B* occurs, since *A* is still '0' during execution at time 0 ns. Time advances to  $\Delta$ , and *A* changes to '1'. The change in *A* causes process *P2* to execute, and since *A* = '1', *B* is scheduled to change to '1' at time 10 ns. The next scheduled change occurs at time = 5 ns, when *A* changes to '0'. This change causes *P2* to execute, but *B* does not change. *B* changes to '1' at time = 10 ns. The change in *B* causes *P1* to execute, and two changes in *A* are scheduled. When *A* changes to '1' at time  $10 + \Delta$ , process *P2* executes, and *B* is scheduled to change at time 20 ns. Then *A* changes at time 15 ns, and the simulation continues in this manner until the run time limit is reached.

VHDL simulators use event-driven simulation, as illustrated in the preceding example. A change in a signal is referred to as an *event*. Each time an event occurs, any processes that have been waiting on the event are executed in zero time, and any resulting signal changes are queued up to occur at some future time. When all the active processes are finished executing, simulation time is advanced to the time for which the next event is scheduled, and the simulator processes that event. This continues until either no more events have been scheduled or the simulation time limit is reached.

**Figure 2-12 Signal Drivers for Simulation Example**



## 2.5 MODELING A SEQUENTIAL MACHINE

In this section we discuss several ways of writing VHDL descriptions for sequential machines. First, we write a behavioral model for a Mealy sequential network based on the state table of Figure 1-17. As shown in Figure 1-16, the Mealy machine consists of a combinational network and a state register. The VHDL model of Figure 2-13 uses two processes to represent these two parts of the network. At the behavioral level, we will represent the state and next state of the network by integer signals initialized to 0. The first

process represents the combinational network. Since the network outputs, *Z* and *Nextstate*, can change when either the *State* or *X* changes, the sensitivity list includes both *State* and *X*. The case statement tests the value of *State*, and depending on the value of *X*, *Z* and *Nextstate* are assigned new values. The second process represents the state register. Whenever the rising edge of the clock occurs, *State* is updated to the value of *Nextstate*, so *CLK* appears in the sensitivity list.

In Figure 2-13, *State* is an integer. Since only seven integer values of *State* are explicit choices, the statement **when others => null** is included. The **null** implies no action, which is appropriate, since the other values of *State* should never occur.

**Figure 2-13 Behavioral Model for Figure 1-17**

```
-- This is a behavioral model of a Mealy state machine (Figure 1-17)
-- based on its state table. The output (Z) and next state are
-- computed before the active edge of the clock. The state change
-- occurs on the rising edge of the clock.
entity SM1_2 is
  port(X, CLK: in bit;
       Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
  signal State, Nextstate: integer := 0;
begin
  process(State,X)                                --Combinational Network
  begin
    case State is
      when 0 =>
        if X='0' then Z<='1'; Nextstate<=1; end if;
        if X='1' then Z<='0'; Nextstate<=2; end if;
      when 1 =>
        if X='0' then Z<='1'; Nextstate<=3; end if;
        if X='1' then Z<='0'; Nextstate<=4; end if;
      when 2 =>
        if X='0' then Z<='0'; Nextstate<=4; end if;
        if X='1' then Z<='1'; Nextstate<=4; end if;
      when 3 =>
        if X='0' then Z<='0'; Nextstate<=5; end if;
        if X='1' then Z<='1'; Nextstate<=5; end if;
      when 4 =>
        if X='0' then Z<='1'; Nextstate<=5; end if;
        if X='1' then Z<='0'; Nextstate<=6; end if;
      when 5 =>
        if X='0' then Z<='0'; Nextstate<=0; end if;
        if X='1' then Z<='1'; Nextstate<=0; end if;
      when 6 =>
        if X='0' then Z<='1'; Nextstate<=0; end if;
      when others => null;                         -- should not occur
    end case;
  end process;
```

```

process(CLK)                                -- State Register
begin
  if CLK='1' then                         -- rising edge of clock
    State <= Nextstate;
  end if;
end process;
end Table;

```

A simulator command file that can be used to test Figure 2-13 is as follows:

```

wave CLK X State NextState Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600

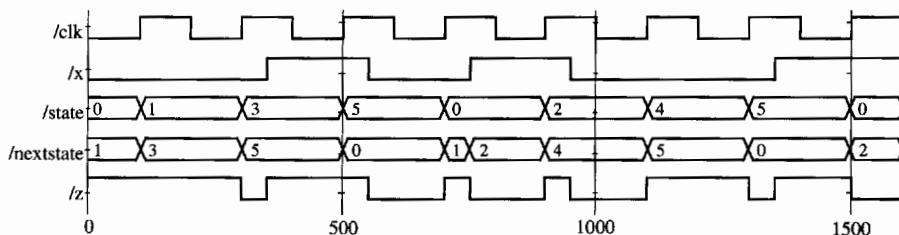
```

The first command specifies the signals that are to be included in the waveform output. The next command defines a clock with period of 200 ns. *CLK* is '0' at time 0 ns, is '1' at time 100 ns, and repeats every 200 ns. In a command of the form

```
force signal_name v1 t1, v2 t2, ...
```

*signal\_name* gets the value *v1* at time *t1*, the value *v2* at time *t2*, etc. *X* is '0' at time 0 ns, changes to '1' at time 350 ns, changes to '0' at time 550 ns, etc. The *X* input corresponds to the sequence 0010 1001, and only the times at which *X* changes are specified. These changes occur at the same times relative to the clock, as shown in Figure 1-27. Execution of the preceding command file produces the waveforms shown in Figure 2-14, which are similar to those in Figure 1-27.

Figure 2-14 Waveforms for Figure 2-13



The data-flow VHDL model of Figure 2-15 is based on the next-state and output equations, which are derived in Figure 1-19. The flip-flops are updated in a process that is sensitive to *CLK*. When the rising edge of the clock occurs, *Q1*, *Q2*, and *Q3* are all assigned new values. A 10-ns delay is included to represent the propagation delay between the active edge of the clock and the change of the flip-flop outputs. Even though the assignment statements in the process are executed sequentially, *Q1*, *Q2*, and *Q3* are all scheduled to be updated at the same time,  $T + \Delta$ , where  $T$  is the time at which the rising edge of the clock occurred. Thus, the old value of *Q1* is used to compute  $Q2^+$ , and the old values of *Q1*, *Q2*, and *Q3* are used to compute  $Q3^+$ . The concurrent assignment statement for *Z* causes *Z* to be updated whenever a change in *X* or *Q3* occurs. The 20-ns delay represents two gate delays.

**Figure 2-15 Sequential Machine Model Using Equations**

```
-- The following is a description of the sequential machine of
-- Figure 1-17 in terms of its next state equations.
-- The following state assignment was used:
-- S0-->0; S1-->4; S2-->5; S3-->7; S4-->6; S5-->3; S6-->2

entity SM1_2 is
  port(X,CLK: in bit;
       Z: out bit);
end SM1_2;

architecture Equations1_4 of SM1_2 is
  signal Q1,Q2,Q3: bit;
begin
  process(CLK)
  begin
    if CLK='1' then
      Q1<=not Q2 after 10 ns; -- rising edge of clock
      Q2<=Q1 after 10 ns;
      Q3<=(Q1 and Q2 and Q3) or ((not X) and Q1 and (not Q3)) or
        (X and (not Q1) and (not Q2)) after 10 ns;
    end if;
  end process;
  Z<=((not X) and (not Q3)) or (X and Q3) after 20 ns;
end Equations1_4;
```

Figure 2-16 shows a structural VHDL representation of the network of Figure 1-20. Seven NAND gates, three D flip-flops, and one inverter are used. All these elements are defined in a library named BITLIB. The element definitions are contained in a package called *bit\_pack*. (See Appendix B for a listing of *bit\_pack*.) The **library** and **use** statements are explained in Section 2.11. Since *Q1*, *Q2*, and *Q3* are initialized to '0', the complementary flip-flop outputs (*Q1N*, *Q2N*, and *Q3N*) are initialized to '1'. *G1* is a 3-input NAND gate with inputs *Q1*, *Q2*, *Q3*, and output *A1*. *FF1* is a D flip-flop (see Figure 2-5) with the *D* input connected to *Q2N*. All the gates and flip-flops in *bit\_pack* have a default delay of

10 ns. Executing the simulator command file given below produces the waveforms of Figure 2-17, which are very similar to Figure 1-28.

```
wave CLK X Q1 Q2 Q3 Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

**Figure 2-16 Structural Model of Sequential Machine**

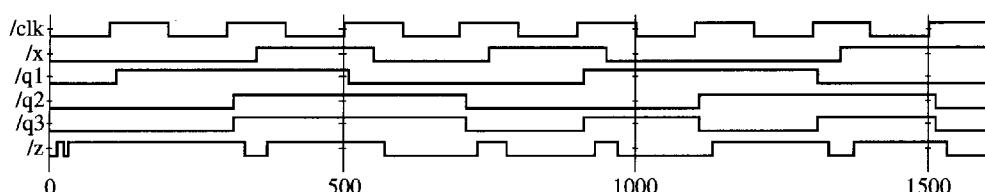
```
-- The following is a STRUCTURAL VHDL description of
-- the network of Figure 1-20.
```

```
library BITLIB;
use BITLIB.bit_pack.all;

entity SM1_2 is
  port(X,CLK: in bit;
       Z: out bit);
end SM1_2;

architecture Structure of SM1_2 is
  signal A1,A2,A3,A5,A6,D3: bit:='0';
  signal Q1,Q2,Q3: bit:='0';
  signal Q1N,Q2N,Q3N, XN: bit:='1';
begin
  I1: Inverter port map (X,XN);
  G1: Nand3 port map (Q1,Q2,Q3,A1);
  G2: Nand3 port map (Q1,Q3N,XN,A2);
  G3: Nand3 port map (X,Q1N,Q2N,A3);
  G4: Nand3 port map (A1,A2,A3,D3);
  FF1: DFF port map (Q2N,CLK,Q1,Q1N);
  FF2: DFF port map (Q1,CLK,Q2,Q2N);
  FF3: DFF port map (D3,CLK,Q3,Q3N);
  G5: Nand2 port map (X,Q3,A5);
  G6: Nand2 port map (XN,Q3N,A6);
  G7: Nand2 port map (A5,A6,Z);
end Structure;
```

**Figure 2-17 Waveforms for Figure 2-16**



An alternative form for a process uses wait statements instead of a sensitivity list. A process cannot have both wait statement(s) and a sensitivity list. A process with wait statements may have the form

```
process
begin
    sequential-statements
    wait-statement
    sequential-statements
    wait-statement
    .
    .
    .
end process;
```

This process will execute the sequential-statements until a wait statement is encountered. Then it will wait until the specified wait condition is satisfied. It will then execute the next set of sequential-statements until another wait is encountered. It will continue in this manner until the end of the process is reached. Then it will start over again at the beginning of the process.

Wait statements can be of three different forms:

- wait on** sensitivity-list;
- wait for** time-expression;
- wait until** boolean-expression;

The first form waits until one of the signals on the sensitivity list changes. The second form waits until the time specified by time expression has lapsed. If **wait for** 5 ns is used, the process waits for 5 ns before continuing. If **wait for** 0 ns is used, the wait is for one delta time. For the third form, the boolean-expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to TRUE. For example,

```
wait until A = B;
```

will wait until either  $A$  or  $B$  changes. Then  $A = B$  is evaluated, and if the result is TRUE, the process will continue; else the process will continue to wait until  $A$  or  $B$  changes again and  $A = B$  is TRUE.

The following example (Figure 2-18) uses a single process with wait statements to model the behavior of a Mealy sequential network. The case statement in this process is the same as in Figure 2-13. After exiting the case statement, the process waits for either the clock or  $X$  to change. If the rising edge of the clock has occurred, the state is updated. Since updating the state requires a delta time, **wait for** 0 ns ensures that the state is updated before the case statement is executed again.

For Figure 2-18, if  $X$  changes at the same time as the rising edge of the clock, the new value of  $X$  will be used to compute the values of  $Nextstate$  and  $Z$ , and the timing waveforms will be correct. If  $X$  changes after the rising edge of the clock,  $Nextstate$  and  $Z$  will have already been computed using the old value of  $X$ . The change in  $X$  will cause  $Nextstate$  and  $Z$  to be updated again, and the timing waveforms will also be correct. In general, the two-

process model for a state machine is preferable to the one-process model, since the former more accurately models the actual hardware. When we use CAD tools for automatic synthesis, constructs like **wait for** 0 ns are not allowed.

In order to modify the code in Figure 2-18 to account for propagation delay in updating the state register, we could replace the **if** statement with

```
if rising_edge(CLK) then
    state <= Nextstate after delay1;
    wait for delay1;
end if;
```

However, the process then would not respond to any change in X that occurred while waiting for delay1. The state machine would still function correctly, but we might miss some glitches that occur in the output waveform. This may not be important if the behavioral model of the sequential machine is used by itself just to check the output sequence; however, if the model is used as part of a larger system, correct timing may become very important.

**Figure 2-18 Behavioral Model for Figure 1-17 Using a Single Process**

```
-- This is a behavioral model of a Mealy state machine based on
-- its state table. The output (Z) and next state are computed
-- on the rising edge of the clock OR when the input (X) changes.
-- The state change occurs on the rising edge of the clock.

library BITLIB;
use BITLIB.Bit_pack.all;

entity SM1_2 is
    port(X, CLK: in bit;
         Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
    signal State, Nextstate: integer := 0;
begin
    process
    begin
        case State is
            when 0 =>
                if X='0' then Z<='1'; Nextstate<=1; end if;
                if X='1' then Z<='0'; Nextstate<=2; end if;
            when 1 =>
                if X='0' then Z<='1'; Nextstate<=3; end if;
                if X='1' then Z<='0'; Nextstate<=4; end if;
            when 2 =>
                if X='0' then Z<='0'; Nextstate<=4; end if;
                if X='1' then Z<='1'; Nextstate<=4; end if;
```

```

when 3 =>
    if X='0' then Z<='0'; Nextstate<=5; end if;
    if X='1' then Z<='1'; Nextstate<=5; end if;
when 4 =>
    if X='0' then Z<='1'; Nextstate<=5; end if;
    if X='1' then Z<='0'; Nextstate<=6; end if;
when 5 =>
    if X='0' then Z<='0'; Nextstate<=0; end if;
    if X='1' then Z<='1'; Nextstate<=0; end if;
when 6 =>
    if X='0' then Z<='1'; Nextstate<=0; end if;
when others => null; -- should not occur
end case;
wait on CLK, X;
if (rising_edge(CLK)) then      -- rising_edge function is in BITLIB
    State <= Nextstate;
    wait for 0 ns;           -- wait for State to be updated
end if;
end process;
end table;

```

## 2.6 VARIABLES, SIGNALS, AND CONSTANTS

Up to this point, we have used only signals in processes and have not used variables. Variables may be used for local storage in processes, procedures, and functions. A variable declaration has the form

```
variable list_of_variable_names : type_name [ := initial_value];
```

Variables must be declared within the process in which they are used and are local to that process. (An exception to this rule is *shared* variables, which are not discussed in this text.) Signals, on the other hand, must be declared outside of a process. Signals declared at the start of an architecture can be used anywhere within that architecture. A signal declaration has the form

```
signal list_of_signal_names : type_name [ := initial_value];
```

A common form of constant declaration is

```
constant constant_name : type_name := constant_value;
```

A constant *delay1* of type time having the value of 5 ns can be defined as

```
constant delay1 : time := 5 ns;
```

Constants declared at the start of an architecture can be used anywhere within that architecture, but constants declared within a process are local to that process.

Variables are updated using a variable assignment statement of the form

```
variable_name := expression;
```

When this statement is executed, the variable is instantaneously updated with no delay, not even a delta delay. In contrast, consider a signal assignment of the form

```
signal_name <= expression [after delay];
```

The expression is evaluated when this statement is executed, and the signal is scheduled to change after delay. If no delay is specified, then the signal is scheduled to be updated after a delta delay.

The examples in Figures 2-19 and 2-20 illustrate the difference between using variables and signals in a process. The variables must be declared and initialized inside the process, whereas the signals must be declared and initialized outside the process. In Figure 2-19, if trigger changes at time = 10, *Var1*, *Var2*, and *Var3* are computed sequentially and updated instantly, and then *Sum* is computed using the new variable values. The sequence is *Var1* =  $2 + 3 = 5$ , *Var2* = 5, *Var3* = 5. Then *Sum* =  $5 + 5 + 5$  is computed. Since *Sum* is a signal, it is updated  $\Delta$  time later, so *Sum* = 15 at time =  $10 + \Delta$ . In Figure 2-20, if trigger changes at time = 10, signals *Sig1*, *Sig2*, *Sig3*, and *Sum* are all computed at time 10, but the signals are not updated until time  $10 + \Delta$ . The old values of *Sig1* and *Sig2* are used to compute *Sig2* and *Sig3*. Therefore, at time =  $10 + \Delta$ , *Sig1* = 5, *Sig2* = 1, *Sig3* = 2, and *Sum* = 6.

**Figure 2-19 Process Using Variables**

```
entity dummy is
end dummy;

architecture var of dummy is
  signal trigger, sum: integer:=0;
begin
  process
    variable var1: integer:=1;
    variable var2: integer:=2;
    variable var3: integer:=3;
  begin
    wait on trigger;
    var1 := var2 + var3;
    var2 := var1;
    var3 := var2;
    sum <= var1 + var2 + var3;
  end process;
end var;
```

**Figure 2-20 Process Using Signals**

```

entity dummy is
end dummy;

architecture sig of dummy is
    signal trigger, sum: integer:=0;
    signal sig1: integer:=1;
    signal sig2: integer:=2;
    signal sig3: integer:=3;
begin
    process
    begin
        wait on trigger;
        sig1 <= sig2 + sig3;
        sig2 <= sig1;
        sig3 <= sig2;
        sum <= sig1 + sig2 + sig3;
    end process;
end sig;

```

Variables, signals, and constants can have any one of the predefined VHDL types or they can have a user-defined type. Some of the predefined types are

bit	'0' or '1'
boolean	FALSE or TRUE
integer	an integer in the range $-(2^{31}-1)$ to $+(2^{31}-1)$ (some implementations support a wider range)
real	floating-point number in the range $-1.0E38$ to $+1.0E38$
character	any legal VHDL character including upper- and lowercase letters, digits, and special characters (each printable character must be enclosed in single quotes; e.g., 'd','7','+')
time	an integer with units fs, ps, ns, us, ms, sec, min, or hr

Note that the integer range for VHDL is symmetrical, even though the range for a 32-bit 2's complement integer is  $-2^{31}$  to  $+(2^{31}-1)$ .

A common user-defined type is the enumeration type in which all of the values are enumerated. For example, the declarations

```

type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type := S1;

```

define a signal called *state* that can have any one of the values S0, S1, S2, S3, S4, or S5, and that is initialized to S1. If no initialization is given, the default initialization is the leftmost element in the enumeration list, S0 in this example. VHDL is a strongly typed language, so signals and variables of different types generally cannot be mixed in the same assignment statement, and no automatic type conversion is performed. Thus the statement *A <= B or C* is valid only if A, B, and C all have the same type or closely related types.

## 2.7 ARRAYS

In order to use an array in VHDL, we must first declare an array type and then declare an array object. For example, the following declaration defines a one-dimensional array type named SHORT\_WORD:

```
type SHORT_WORD is array (15 downto 0) of bit;
```

An array of this type has an integer index with a range from 15 downto 0, and each element of the array is of type bit.

Next, we declare array objects of type SHORT\_WORD:

```
signal DATA_WORD: SHORT_WORD;
variable ALT_WORD: SHORT_WORD := "0101010101010101";
constant ONE_WORD: SHORT_WORD := (others => '1');
```

*DATA\_WORD* is a signal array of 16 bits, indexed 15 downto 0, which is initialized (by default) to all '0' bits. *ALT\_WORD* is a variable array of 16 bits, which is initialized to alternating 0s and 1s. *ONE\_WORD* is a constant array of 16 bits; all bits are set to 1 by (**others** => '1'). We can reference individual elements of the array by specifying an index value. For example, *ALT\_WORD*(0) accesses the rightmost bit of *ALT\_WORD*. We can also specify a portion of the array by specifying an index range: *ALT\_WORD*(5 **downto** 0) accesses the low-order 6 bits of *ALT\_WORD*, which have an initial value of 010101.

The array type and array object declarations illustrated here have the general forms

```
type array_type_name is array index_range of element_type;
signal array_name: array_type_name [ := initial_values ];
```

In the preceding declaration, **signal** may be replaced with **variable** or **constant**.

Multidimensional array types may also be defined with two or more dimensions. The following example defines a two-dimensional array variable, which is a matrix of integers with four rows and three columns:

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;
variable matrixA: matrix4x3 := ((1, 2, 3), (4, 5, 6),
(7, 8, 9), (10, 11, 12));
```

The variable *matrixA*, will be initialized to

1	2	3
4	5	6
7	8	9
10	11	12

The array element *matrixA*(3, 2) references the element in the third row and second column, which has a value of 8.

When an array type is declared, the dimensions of the array may be left undefined.

This is referred to as an *unconstrained array type*. For example,

```
type intvec is array (natural range <>) of integer;
```

declares intvec as an array type that defines a one-dimensional array of integers with an unconstrained index range of natural numbers. The default type for array indices is integer, but another type may be specified. Since the index range is not specified in the unconstrained array type, the range must be specified when the array object is declared. For example,

```
signal intvec5: intvec(1 to 5) := (3,2,6,8,1);
```

defines a signal array named *intvec5* with an index range of 1 to 5 that is initialized to 3, 2, 6, 8, 1. The following declaration defines matrix as a two-dimensional array type with unconstrained row and column index ranges:

```
type matrix is array (natural range <>, natural range <>) of integer;
```

The VHDL code in Figure 2-21 is a behavioral model for the sequential machine of Figure 1-17(a) using arrays to represent the state and output tables. Two two-dimensional array types are defined—an integer array for the state table and a bit array for the output table. In both cases, the first index is an integer of unconstrained range, and the second index is a bit of unconstrained range. When the actual state and output tables are declared as constants, the actual index ranges are specified as 0 to 6 for the rows and '0' to '1' for the two columns. Every time *X* or *State* changes, the state and output tables are read by the concurrent statements to determine the *NextState* and *Z* values. The *State* is updated on the rising edge of the clock by the process.

Figure 2-21 Sequential Machine Model Using State Table

```
entity SM1_2 is
  port (X, CLK: in bit;
        Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
  type StateTable is array (integer range <>, bit range <>) of integer;
  type OutTable is array (integer range <>, bit range <>) of bit;
  signal State, NextState: integer := 0;
  constant ST: StateTable (0 to 6, '0' to '1') :=
    ((1,2), (3,4), (4,4), (5,5), (5,6), (0,0), (0,0));
  constant OT: OutTable (0 to 6, '0' to '1') :=
    (('1','0'), ('1','0'), ('0','1'), ('0','1'), ('1', '0'), ('0','1'),
     ('1','0'));
begin
  NextState <= ST(State,X);
  Z <= OT(State,X);
  -- concurrent statements
  -- read next state from state table
  -- read output from output table

```

```

process (CLK)
begin
    if CLK = '1' then                      -- rising edge of CLK
        State <= NextState;
    end if;
end process;
end Table;

```

Predefined unconstrained array types in VHDL include `bit_vector` and `string`, which are defined as follows:

```

type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;

```

The characters in a string literal must be enclosed in double quotes. For example, “This is a string.” is a string literal. The following example declares a constant `string1` of type `string`:

```

constant string1: string(1 to 29) :=  
    "This string is 29 characters."

```

A `bit_vector` literal may be written either as a list of bits separated by commas or as a string. For example, `('1','0','1','1','0')` and `"10110"` are equivalent forms. The following declares a constant `A` that is a `bit_vector` with a range 0 to 5.

```
constant A : bit_vector(0 to 5) := "101011";
```

After a type has been declared, a related subtype can be declared to include a subset of the values specified by the type. For example, the type `SHORT_WORD`, which was defined at the start of this section, could have been defined as a subtype of `bit_vector`:

```
subtype SHORT_WORD is bit_vector (15 downto 0);
```

Two predefined subtypes of type `integer` are `POSITIVE`, which includes all positive integers, and `NATURAL`, which includes all positive integers and 0.

## 2.8 VHDL OPERATORS

Predefined VHDL operators can be grouped into seven classes:

1. Binary logical operators: `and` `or` `nand` `nor` `xor` `xnor`
2. Relational operators: `=` `/=` `<=` `>=`
3. Shift operators: `sll` `srl` `sla` `sra` `rol` `ror`
4. Adding operators: `+` `-` `&` (concatenation)
5. Unary sign operators: `+` `-`
6. Multiplying operators: `*` `/` `mod` `rem`
7. Miscellaneous operators: `not` `abs` `**`

When parentheses are not used, operators in class 7 have highest precedence and are applied first, followed by class 6, then class 5, etc. Class 1 operators have lowest precedence and are applied last. Operators in the same class have the same precedence and are applied from left to right in an expression. The precedence order can be changed by using parentheses. In the following expression, A, B, C, and D are bit\_vectors:

```
(A & not B or C ror 2 and D) = "110010"
```

The operators are applied in the order:

**not, &, ror, or, and, =**

If A = "110", B = "111", C = "011000", and D = "111011", the computation proceeds as follows:

**not B = "000"** (bit-by-bit complement)

**A & not B = "110000"** (concatenation)

**C ror 2 = "000110"** (rotate right 2 places)

**(A & not B) or (C ror 2) = "110110"** (bit-by-bit or)

**(A & not B or C ror 2) and D = "110010"** (bit-by-bit and)

**[(A & not B or C ror 2) and D] = "110010"] = TRUE** (the parentheses force the equality test to be done last and the result is TRUE)

The binary logical operators (class 1) as well as **not** can be applied to bits, booleans, bit\_vectors, and boolean\_vectors. The class 1 operators require two operands of the same type, and the result is of that type.

The result of applying a relational operator (class 2) is always a boolean (FALSE or TRUE). Equals (=) and not equals (/=) can be applied to almost any type. The other relational operators can be applied to any numeric or enumerated type as well as to some array types. For example, if A = 5, B = 4, and C = 3, the expression **(A >= B) and (B <= C)** evaluates to FALSE.

The shift operators can be applied to any bit\_vector or boolean\_vector. In the following examples, A is a bit\_vector equal to "10010101":

**A sll 2 is "01010100"** (shift left logical, filled with '0')

**A srl 3 is "00010010"** (shift right logical, filled with '0')

**A sla 3 is "10101111"** (shift left arithmetic, filled with right bit)

**A sra 2 is "11100101"** (shift right arithmetic, filled with left bit)

**A rol 3 is "10101100"** (rotate left)

**A ror 5 is "10101100"** (rotate right)

The + and - operators can be applied to integer or real numeric operands. The & operator can be used to concatenate two vectors (or an element and a vector, or two elements) to form a longer vector. For example, "010" & '1' is "0101" and "ABC" & "DEF" is "ABCDEF".

The \* and / operators perform multiplication and division on integer or floating-point operands. The rem and mod operators calculate the remainder and modulus for integer operands. The \*\* operator raises an integer or floating-point number to an integer power, and abs finds the absolute value of a numeric operand.

## 2.9 VHDL FUNCTIONS

A function executes a sequential algorithm and returns a single value to the calling program. When the following function is called, it returns a bit vector equal to the input bit vector (reg) rotated one position to the right:

```
function rotate_right (reg: bit_vector)
    return bit_vector is
begin
    return reg ror 1;
end rotate_right;
```

A function call can be used anywhere that an expression can be used. For example, if  $A = "10010101"$ , the statement

```
B <= rotate_right (A);
```

would set  $B$  equal to "11001010", and leave  $A$  unchanged.

The general form of a function declaration is

```
function function-name (formal-parameter-list)
    return return-type is
    [declarations]
begin
    sequential statements -- must include return return-value;
end function-name;
```

The general form of a function call is

```
function_name (actual-parameter-list)
```

The number and type of parameters on the actual-parameter-list must match the formal-parameter-list in the function declaration. The parameters are treated as input values and cannot be changed during execution of the function.

The function defined in Figure 2-22 uses a **for** loop. The general form of a **for** loop is

```
[loop-label:] for loop-index in range loop
    sequential statements
end loop [loop-label];
```

The loop-index is automatically defined when the loop is entered, and it should not explicitly be declared. It is initialized to the first value in the range and then the sequential statements are executed. The loop-index can be used within the sequential statements, but it cannot be changed within the loop. When the end of the loop is reached, the loop-index is set to the next value in the range and the sequential statements are executed again. This process continues until the loop has been executed for every value in the range, and then the loop

terminates. After the loop terminates, the loop-index is no longer available. In Figure 2-22, the loop index (*i*) will be initialized to 0 when the **for** loop is entered, and the sequential statements will be executed. Execution will be repeated for *i* = 1, *i* = 2, and *i* = 3; then the loop will terminate.

An exit statement of the form

```
exit; or exit when condition;
```

may be included in the loop. The loop will terminate when the exit statement is executed, provided that in the second case, the condition is TRUE.

**Figure 2-22 Add Function**

```
-- This function adds 2 4-bit vectors and a carry.
-- It returns a 5-bit sum

function add4 (A,B: bit_vector(3 downto 0); carry: bit)
  return bit_vector is

  variable cout: bit;
  variable cin: bit := carry;
  variable sum: bit_vector(4 downto 0):="00000";
  begin

    loop1: for i in 0 to 3 loop
      cout := (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin);
      sum(i) := A(i) xor B(i) xor cin;
      cin := cout;
    end loop loop1;
    sum(4):= cout;
    return sum;
  end add4;
```

If *A*, *B*, and *C* are integers, the statement  $C \leftarrow A + B$  will set *C* equal to the sum of *A* and *B*. However, if *A*, *B*, and *C* are bit\_vectors, this statement will not work, since the "+" operation is not defined for bit\_vectors. However, we can write a function to perform bit\_vector addition. The function given in Figure 2-22 adds two 4-bit vectors plus a carry and returns a 5-bit vector as the sum. The function name is *add4*, the formal parameters are *A*, *B*, and *carry*, and the return-type is bit\_vector. Variables *cout* and *cin* are defined to hold intermediate values during the calculation. The variable *sum* is used to store the value to be returned. When the function is called, *cin* will be initialized to the value of the carry. The **for** loop adds the bits of *A* and *B* serially in the same manner as a serial adder. The first time through the loop, *cout* and *sum*(0) are computed using *A*(0), *B*(0), and *cin*. Then the *cin* value is updated to the new *cout* value, and execution of the loop is repeated. The second time through the loop, *cout* and *sum*(1) are computed using *A*(1), *B*(1), and the new *cin*. After four times through the loop, all values of *sum*(*i*) have been computed and *sum* is returned. The total simulation time required to execute the *add4* function is zero. Not even

delta time is required, since all the computations are done using variables, and variables are updated instantaneously.

The function call is of the form

```
add4( A, B, carry )
```

*A* and *B* may be replaced with any expressions that evaluate to bit\_vectors with dimensions 3 **downto** 0, and *carry* may be replaced with any expression that evaluates to a bit. For example, the statement

```
Z <= add4(X, not Y, '1');
```

calls the function *add4*. Parameters *A*, *B*, and *carry* are set equal to the values of *X*, **not** *Y*, and '1', respectively. *X* and *Y* must be bit\_vectors dimensioned 3 **downto** 0. The function computes

$$\text{Sum} = A + B + \text{carry} = X + \text{not } Y + '1'$$

and returns this value. Since *Sum* is a variable, computation of *Sum* requires zero time. After delta time, *Z* is set equal to the returned value of *Sum*. Since **not** *Y* + '1' equals the 2's complement of *Y*, the computation is equivalent to subtracting by adding the 2's complement. If we ignore the carry stored in *Z*(4), the result is *Z*(3 **downto** 0) = *X* - *Y*.

Functions are frequently used to do type conversions. The function *vec2int(bitvec)* accepts a bit\_vector as input and returns the corresponding integer value. The function *int2vec(int,N)* accepts two positive integers as inputs and converts *int* to a bit\_vector of length *N*. We have written these functions in a general manner so that the bit\_vector can be of any length (see Chapter 8) and placed them in the *bit\_pack* package in the BITLIB library.

## 2.10 VHDL PROCEDURES

Procedures facilitate decomposition of VHDL code into modules. Unlike functions, which return only a single value through a return statement, procedures can return any number of values using output parameters. The form of a procedure declaration is

```
procedure procedure_name (formal-parameter-list) is
    [declarations]
begin
    sequential statements
end procedure-name;
```

The formal-parameter-list specifies the inputs and outputs to the procedure and their types. A procedure call is a sequential or concurrent statement of the form

```
procedure_name (actual-parameter-list);
```

As an example we will write a procedure *Addvec*, which will add two *N*-bit vectors and a carry, and return an *N*-bit sum and a carry. We will use a procedure call of the form

```
Addvec ( A, B, Cin, Sum, Cout, N);
```

where *A*, *B*, and *Sum* are *N*-bit vectors, *Cin* and *Cout* are bits, and *N* is an integer.

Figure 2-23 gives the procedure definition. *Add1*, *Add2*, and *Cin* are input parameters, and *Sum* and *Cout* are output parameters. *N* is a positive integer that specifies the number of bits in the bit\_vectors. The addition algorithm is essentially the same as the one used in the *add4* function. *C* must be a variable, since the new value of *C* is needed each time through the loop; however, *Sum* can be a signal since *Sum* is not used within the loop. After *N* times through the loop, all the values of the signal *Sum* have been computed, but *Sum* is not updated until delta time after exiting from the loop.

**Figure 2-23 Procedure for Adding Bit\_vectors**

```
-- This procedure adds two n-bit bit_vectors and a carry and
-- returns an n-bit sum and a carry. Add1 and Add2 are assumed
-- to be of the same length and dimensioned n-1 downto 0.
procedure Addvec
    (Add1,Add2: in bit_vector;
     Cin: in bit;
     signal Sum: out bit_vector;
     signal Cout: out bit;
     n:in positive) is
        variable C: bit;
begin
    C := Cin;
    for i in 0 to n-1 loop
        Sum(i) <= Add1(i) xor Add2(i) xor C;
        C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
    end loop;
    Cout <= C;
end Addvec;
```

Within the procedure declaration, the class, mode, and type of each parameter must be specified in the formal-parameter-list. The class of each parameter can be **signal**, **variable**, or **constant**. If the class is omitted, **constant** is used as the default. If the class is a **signal**, then the actual parameter in the procedure call must be a **signal** of the same type. Similarly, for a formal parameter of class **variable**, the actual parameter must be a **variable** of the same type. However, for a **constant** formal parameter, the actual parameter can be any expression that evaluates to a constant of the proper type. This constant value is used inside the procedure and cannot be changed; thus a **constant** formal parameter is always of mode **in**. Signals and variables can be of mode **in**, **out**, or **inout**. Parameters of mode **out** and **inout** can be changed in the procedure, so they are used to return values to the caller.

In procedure *Addvec*, parameters *Add1*, *Add2*, and *Cin* are, by default, of class constant. Therefore, in the procedure call, *Add1*, *Add2*, and *Cin* can be replaced with any expressions that evaluate to constants of the proper type and dimension. Since *Sum* and *Cout* change within the procedure and are used to return values, they have been declared as class signal. Thus, in the procedure call, *Sum* and *Cout* can be replaced only with signals of the proper type and dimension.

The formal-parameter-list in a **function** declaration is similar to that of a **procedure**, except parameters of class **variable** are not allowed. Furthermore, all parameters must be of mode **in**, which is the default mode. Parameters of mode **out** or **inout** are not allowed, since a function returns only a single value, and this value cannot be returned through a parameter. Table 2-1 summarizes the modes and classes that may be used for procedure and function parameters.

**Table 2-1 Parameters for Subprogram Calls**

		<b>Actual Parameter</b>	
Mode	Class	Procedure Call	Function Call
in <sup>1</sup>	constant <sup>2</sup>	expression	expression
	signal	signal	signal
	variable	variable	n/a
out/inout	signal	signal	n/a
	variable <sup>3</sup>	variable	n/a

<sup>1</sup> default mode for functions    <sup>2</sup> default for in mode    <sup>3</sup> default for out/inout mode

## 2.11 PACKAGES AND LIBRARIES

*Packages* and *libraries* provide a convenient way of referencing frequently used functions and components. A package consists of a package declaration and an optional package body. The package declaration contains a set of declarations, which may be shared by several design units. For example, it may contain type, signal, component, function, and procedure declarations. The package body usually contains the function and procedure bodies. The package and its associated compiled VHDL models may be placed in a library so they can be accessed as required by different VHDL designs. A package declaration has the form

```
package package-name is
    package declarations
end [package] [package-name];
```

A package body has the form

```
package body package-name is
    package body declarations
end [package body] [package name];
```

We have developed a package called *bit\_pack* that is used in a number of examples in this book. This package contains commonly used components and functions that use signals of type bit and bit\_vector. Appendix B contains a complete listing of this package and associated component models. Most of the components in this package have a default delay of 10 ns, but this delay can be changed by use of generics, as explained in Section 8.6. We have compiled this package and the component models and placed the result in a library called *BITLIB*.

One of the components in the library is a two-input NAND gate named *Nand2*, which has default delay of 10 ns. The package declaration for *bit\_pack* includes the component declaration

```
component Nand2
    generic (DELAY : time := 10 ns)
    port (A1, A2 : in bit;  Z : out bit);
end component;
```

The NAND gate is modeled using a concurrent statement. The entity-architecture pair for this component is

```
-- 2-input NAND gate
entity Nand2 is
    generic (DELAY : time)
    port (A1, A2 : in bit;  Z : out bit);
end Nand2;
architecture concur of Nand2 is
begin
    Z <= not(A1 and A2) after DELAY;
end;
```

To access components and functions within a package requires a **library** statement and a **use** statement. The statement

```
library BITLIB;
```

allows your design to access the *BITLIB*. The statement

```
use BITLIB.bit_pack.all;
```

allows your design to use the entire *bit\_pack* package. A statement of the form

```
use BITLIB.bit_pack.Nand2;
```

may be used if you want to use a specific component or function in the package.

## 2.12 VHDL MODEL FOR A 74163 COUNTER

The 74163 (Figure 2-24) is a 4-bit fully synchronous binary counter that is available in both TTL and CMOS logic families. In addition to performing the counting function, it can be cleared or loaded in parallel. The following table summarizes the counter operation:

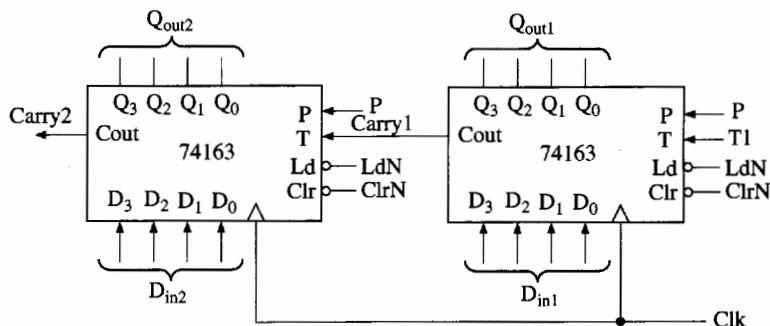
Control Signals			Next State			
ClrN	LdN	P.T	Q3 <sup>+</sup>	Q2 <sup>+</sup>	Q1 <sup>+</sup>	Q0 <sup>+</sup>
0	X	X	0	0	0	0
1	0	X	D3	D2	D1	D0
1	1	0	Q3	Q2	Q1	Q0
1	1	1	present state + 1			(increment count)

All state changes occur following the rising edge of the clock. The counter generates a carry ( $Cout$ ) in state 15 if  $T = 1$ , so

$$Cout = Q_3 Q_2 Q_1 Q_0 T$$

The VHDL description of the counter is shown in Figure 2-25. The carry output is computed whenever  $Q$  or  $T$  changes. The wait statement in the process waits until  $CK$  changes and then continues execution if  $CK$  has changed to '1'. Since clear overrides load and count,  $ClrN$  is tested first in the process. Since load overrides count,  $LdN$  is tested next. Finally, the counter is incremented if both  $P$  and  $T$  are 1. Since  $Q$  is a bit\_vector, we convert it to an integer, add 1, and then convert back to a 4-bit vector. We have named the model *c74163* and placed it in *BITLIB.bit\_pack*.

Figure 2-24 Two 74163 Counters Cascaded to Form an 8-bit Counter



To test the counter, we have cascaded two 74163s to form an 8-bit counter (Figure 2-24). When the counter on the right is in state 1111,  $Carry1 = 1$ . If  $P \cdot T = 1$ , then on the next clock the right counter is incremented to 0000 at the same time the left counter is incremented. Figure 2-26 shows the VHDL code for the 8-bit counter. In this code we have used the *c74163* model as a component and instantiated two copies of it. For convenience in reading the output, we have defined a signal *Count*, which is the integer equivalent of the 8-bit counter value. After testing the counter, we placed it in the *BITLIB.bit\_pack* for future use. When this is done, the component declaration can be omitted from the code in Figure 2-26.

**Figure 2-25 74163 Counter Model**

```
-- 74163 FULLY SYNCHRONOUS COUNTER

library BITLIB;                               -- contains int2vec and vec2int functions
use BITLIB.bit_pack.all;

entity c74163 is
    port(LdN, ClrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
          Cout: out bit; Q: inout bit_vector(3 downto 0) );
end c74163;

architecture b74163 of c74163 is
begin
    Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
    process
    begin
        wait until CK = '1';                  -- change state on rising edge
        if ClrN = '0' then Q <= "0000";
        elsif LdN = '0' then Q <= D;
        elsif (P and T) = '1' then
            Q <= int2vec(vec2int(Q)+1,4);
        end if;
    end process;
end b74163;
```

**Figure 2-26 VHDL for 8-bit Counter**

```
--Test module for 74163 counter

library BITLIB;
use BITLIB.bit_pack.all;

entity c74163test is
    port(ClrN,LdN,P,T1,Clk: in bit;
          Din1, Din2: in bit_vector(3 downto 0);
          Qout1, Qout2: inout bit_vector(3 downto 0);
          Carry2: out bit);
end c74163test;

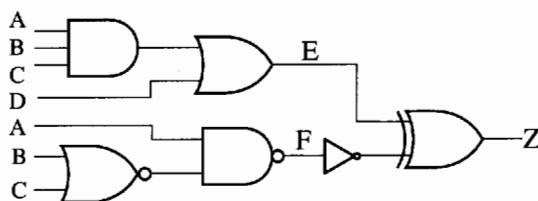
architecture tester of c74163test is
    component c74163
        port(LdN, ClrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
              Cout: out bit; Q: inout bit_vector(3 downto 0) );
    end component;
    signal Carry1: bit;
    signal Count: integer;
    signal temp: bit_vector(7 downto 0);
```

```
begin
  ct1: c74163 port map (LdN,ClrN,P,T1,Clk,Din1,Carry1,Qout1);
  ct2: c74163 port map (LdN,ClrN,P,Carry1,Clk,Din2,Carry2,Qout2);
  temp <= Qout2 & Qout1;
  Count <= vec2int(temp);
end tester;
```

In this chapter, we have covered the basics of VHDL. We have shown how to use VHDL to model combinational logic and sequential machines. Since VHDL is a hardware description language, it differs from an ordinary programming language in several ways. Most importantly, VHDL statements execute concurrently, since they must model real hardware in which the components are all in operation at the same time. Statements within a process execute sequentially, but the processes themselves operate concurrently. VHDL signals model actual signals in the hardware, but variables may be used for internal computation that is local to processes, procedures, and functions. After we have had opportunities to use VHDL in the design process, we cover more advanced features of VHDL in Chapter 8.

## Problems

- 2.1** Write a VHDL description of the following combinational network using concurrent statements. Each gate has a 5-ns delay, excluding the inverter, which has a 2-ns delay.



- 2.2**

- Write VHDL code for a full subtracter using logic equations.
- Write VHDL code for a 4-bit subtracter using the module defined in (a) as a component.

- 2.3** In the following VHDL process *A*, *B*, *C*, and *D* are all integers that have a value of 0 at time = 10 ns. If *E* changes from '0' to '1' at time = 20 ns, specify the time(s) at which each signal will change and the value to which it will change. List these changes in chronological order (20, 20 +  $\Delta$ , 20 + 2 $\Delta$ , etc.)

```
p1: process
begin
    wait on E;
    A <= 1 after 5 ns;
    B <= A + 1;
    C <= B after 10 ns;
    wait for 0 ns;
    D <= B after 3 ns;
    A <= A + 5 after 15 ns;
    B <= B + 7;
end process p1;
```

- 2.4** For the following VHDL code, assume that *D* changes to '1' at time 5 ns. Give the values of *A*, *B*, *C*, *D*, *E*, and *F* each time a change occurs. That is, give the values at time 5 ns, 5 +  $\Delta$ , 5 + 2 $\Delta$ , etc. Carry this out until either 20 steps have occurred, until no further change occurs, or until a repetitive pattern emerges.

```
entity prob4 is
    port (D: inout bit);
end prob4;
architecture q1 of prob4 is
    signal A, B, C, E, F: bit;
begin
    C <= A;
    A <= B or D;
    P1: process (A)
    begin
        B <= A;
    end process P1;
    P2: process
    begin
        wait until A <= '1';
        wait for 0 ns;
        E <= B;
        D <= '0';
        F <= E;
    end process P2;
end architecture q1;
```

**2.5** Write a VHDL description of an SR latch.

- (a) Use a conditional assignment statement.
- (b) Use the characteristic equation.
- (c) Use two logic gates.

**2.6** A gated D latch will hold its output value if  $G$  is 0, and the output follows  $D$  if  $G$  is 1. Write a VHDL description of a gated D latch using a process.

**2.7** A DD flip-flop is similar to a D flip-flop, except that the flip-flop can change state ( $Q^+ = D$ ) on both the rising edge and falling edge of the clock input. The flip-flop has a direct reset input  $R$ , and  $R = 0$  resets the flip-flop to  $Q = 0$  independent of the clock. Write a VHDL description of a DD flip-flop.

**2.8** An inhibited toggle flip-flop has inputs  $I0$ ,  $II$ ,  $T$ , and *Reset*, and outputs  $Q$  and  $QN$ . *Reset* is active high and overrides the action of the other inputs. The flip-flop works as follows. If  $I0 = 1$ , the flip-flop changes state on the rising edge of  $T$ ; if  $II = 1$ , the flip-flop changes state on the falling edge of  $T$ . If  $I0 = II = 0$ , no state change occurs (except on reset). Assume the propagation delay from  $T$  to output is 8 ns and from reset to output is 5 ns.

- (a) Write a complete VHDL description of this flip-flop.
- (b) Write a sequence of simulator commands that will test the flip-flop for the input sequence  $II = 1$ , toggle  $T$  twice,  $II = 0$ ,  $I0 = 1$ , toggle  $T$  twice.

### 2.9

(a) Write a behavioral VHDL description of the state machine that you designed in Problem 1.11. Assume that state changes occur on the falling edge of the clock pulse. Use a case statement together with if-then-else statements to represent the state table. Compile and simulate your code using the following test sequence:

$$X = 1011\ 0111\ 1000$$

$X$  should change 1/4 clock period after the falling edge of the clock.

- (b) Write a data flow VHDL description using the next state and output equations to describe the state machine. Indicate on your simulation output at which times  $Z$  should be read.
- (c) Write a structural model of the state machine in VHDL that contains the interconnection of the gates and J-K flip-flops. You may use the BITLIB library for this part.

### 2.10

(a) Write a behavioral VHDL description of the state machine you designed in Problem 1.12. Assume that state changes occur on the falling edge of the clock pulse. Instead of using if-then-else statements, represent the state table and output table by arrays. Compile and simulate your code using the following test sequence:

$$X = 1101\ 1110\ 1111$$

$X$  should change 1/4 clock period after the falling edge of the clock.

- (b) Write a data flow VHDL description using the next state and output equations to describe the state machine. Indicate on your simulation output at which times  $Z$  is to be read.
- (c) Write a structural model of the state machine in VHDL that contains the interconnection of the gates and J-K flip-flops. You may use the BITLIB library for this part.

- 2.11** A Moore sequential machine with two inputs ( $X1$  and  $X2$ ) and one output ( $Z$ ) has the following state table:

	00	01	10	11	Z
1	1	2	2	1	0
2	2	1	2	1	1

Write VHDL code that describes the machine at the behavioral level. Assume that state changes occur 10 ns after the falling edge of the clock, and output changes occur 10 ns after the state changes.

- 2.12** Write a VHDL function that will take the 2's complement of a  $n$ -bit vector. Use a call of the form  $comp2(bit\_vec, N)$  where  $N$  is the length of the vector. State any assumptions you make about the range of  $bit\_vec$ .

- 2.13**  $X$  and  $Y$  are bit-vectors of length  $N$  that represent signed binary numbers, with negative numbers represented in 2's complement. Write a VHDL procedure that will compute  $D = X - Y$ . This procedure should also return the borrow from the last bit position ( $B$ ) and an overflow flag ( $V$ ). Do not call any other functions or procedures in your code. The procedure call should be of the form

```
SUBVEC(X, Y, D, B, V, N);
```

- 2.14** Write a VHDL function that converts a 5-bit bit\_vector to an integer. Note that the integer value of the binary number  $a_4a_3a_2a_1a_0$  can be computed as

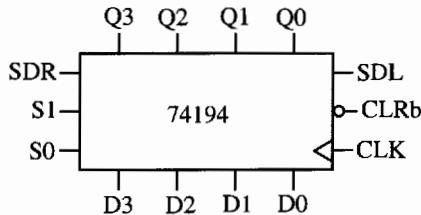
$$(((0 + a_4)*2 + a_3)*2 + a_2)*2 + a_1)*2 + a_0$$

How much simulated time will it take for your function to execute?

- 2.15** Write a VHDL module that describes a 16-bit serial-in, serial-out shift register with inputs  $SI$  (serial input),  $EN$  (enable), and  $CK$  (clock, shifts on rising edge) and a serial output ( $SO$ ).

- 2.16** A description of a 74194 4-bit bidirectional shift register follows:

The  $CLRb$  input is asynchronous and active low and overrides all the other control inputs. All other state changes occur following the rising edge of the clock. If the control inputs  $S1 = S0 = 1$ , the register is loaded in parallel. If  $S1 = 1$  and  $S0 = 0$ , the register is shifted right and  $SDR$  (serial data right) is shifted into  $Q3$ . If  $S1 = 0$  and  $S0 = 1$ , the register is shifted left and  $SDL$  is shifted into  $Q0$ . If  $S1 = S0 = 0$ , no action occurs.



- (a) Write a behavioral level VHDL model for the 74194.
- (b) Draw a block diagram and write a VHDL description of an 8-bit bidirectional shift register that uses two 74194s as components. The parallel inputs and outputs to the 8-bit register should be  $X(7 \text{ downto } 0)$  and  $Y(7 \text{ downto } 0)$ . The serial inputs should be  $RSD$  and  $LSD$ .

**2.17** A synchronous (4-bit) up/down decade counter with output  $Q$  works as follows: All state changes occur on the rising edge of the  $CLK$  input, except the asynchronous clear ( $CLR$ ). When  $CLR = 0$ , the counter is reset regardless of the values of the other inputs.

- If the  $LOAD$  input is 0, the data input  $D$  is loaded into the counter.
- If  $LOAD = ENT = ENP = UP = 1$ , the counter is incremented.
- If  $LOAD = ENT = ENP = 1$  and  $UP = 0$ , the counter is decremented.
- If  $ENT = UP = 1$ , the carry output ( $CO$ ) = 1 when the counter is in state 9.
- If  $ENT = 1$  and  $UP = 0$ , the carry output ( $CO$ ) = 1 when the counter is in state 0.

- (a) Write a VHDL description of the counter.
- (b) Draw a block diagram and write a VHDL description of a decimal counter that uses two of the above counters to form a two-decade decimal up/down counter that counts up from 00 to 99 or down from 99 to 00.

**2.18** Write a VHDL model for a 74HC192 synchronous 4-bit up/down counter. Ignore all timing data. Your code should contain a statement of the form **process** (DOWN, UP, CLR, LOADB).

#### 2.19

- (a) Write a VHDL module that describes the 74198 shift register. Use the following notation:

$Q$	8-bit output (labeled $Q_A$ through $Q_H$ in the data book)
$D$	8-bit input (labeled $A$ through $H$ in the data book)
$S0, S1$	mode control inputs
$LSI$	left serial input
$RSI$	right serial input

- (b) Two 74198 shift registers are connected to form a 16-bit cyclic shift register, which is controlled by signals  $L$  and  $R$ . If  $L = 1$  and  $R = 0$ , then the 16-bit register is cycled left. If  $L = 0$  and  $R = 1$ , the register is cycled right. If  $L = R = 1$ , the 16-bit register is loaded from  $X[15...0]$ . Write a VHDL description of the system using the module from part (a).

# CHAPTER 3

## DESIGNING WITH PROGRAMMABLE LOGIC DEVICES

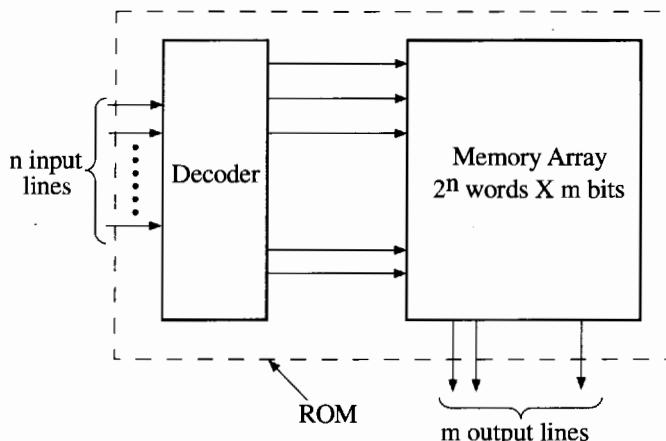
---

This chapter introduces the use of programmable logic devices (PLDs) in digital design. Read-only memories (ROMs), programmable logic arrays (PLAs), and programmable array logic devices (PALs) are discussed, and then more complex PLDs are introduced. Use of these devices allows us to implement complex logic functions, which require many gates and flip-flops, with a single IC chip. More complex PLDs and programmable gate arrays are described in Chapter 6.

This chapter contains several VHDL examples. Sequential networks using a ROM and using a PLA are described in VHDL. Two design examples, a traffic-light controller and a keypad scanner, also use VHDL. The latter example introduces the use of a testbench, which is written in VHDL, to test the VHDL code for the keypad scanner design.

### 3.1 READ-ONLY MEMORIES

A read-only memory (ROM) consists of an array of semiconductor devices interconnected to store an array of binary data. Once binary data is stored in the ROM, it can be read out whenever desired, but the data that is stored cannot be changed under normal operating conditions. A ROM that has  $n$  input lines and  $m$  output lines (Figure 3-1) contains an array of  $2^n$  words, and each word is  $m$  bits long. The input lines serve as an address to select one of the  $2^n$  words. Conceptually, a ROM consists of a decoder and a memory array. When a pattern of  $n$  0s and 1s is applied to the decoder inputs, exactly one of the  $2^n$  decoder outputs is 1. This decoder output line selects one of the words in the memory array, and the bit pattern stored in this word is transferred to the memory output lines. A  $2^n \times m$  ROM can realize  $m$  functions of  $n$  variables, since it can store a truth table with  $2^n$  rows and  $m$  columns.

**Figure 3-1 Basic ROM Structure**

Basic types of ROMs include mask-programmable ROMs and erasable programmable ROMs (usually called EPROMs). At time of manufacture, the data array is permanently stored in a mask-programmable ROM. This is accomplished by selectively including or omitting the switching elements at the row-column intersections of the memory array. This requires preparation of a special “mask,” which is used during fabrication of the integrated circuit. Preparation of this mask is expensive, so use of mask-programmable ROMs is economically feasible only if a large quantity (typically several thousand or more) are required with the same data array. If only a small quantity of ROMs are required with a given data array, EPROMs may be used.

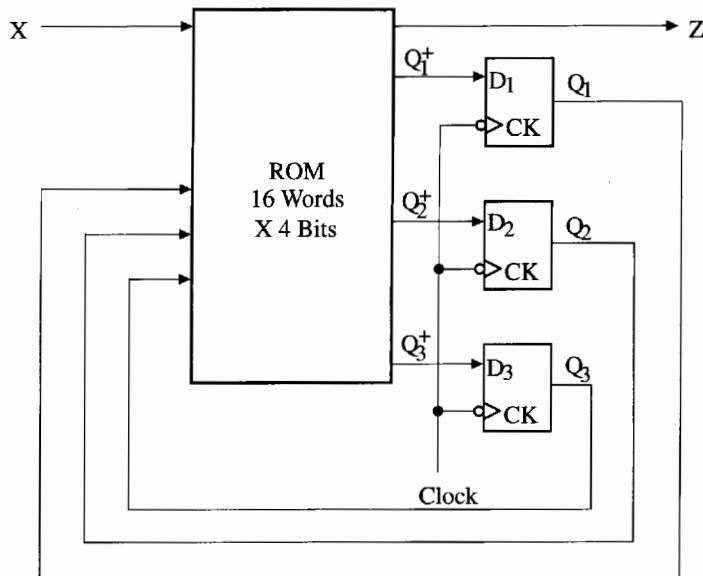
Modification of the data stored in a ROM is often necessary during the developmental phases of a digital system, so EPROMS are used instead of mask-programmable ROMs. EPROMs use a special charge-storage mechanism to enable or disable the switching elements in the memory array. A PROM programmer is used to provide appropriate voltage pulses to store electronic charges in the memory array locations. The data stored in this manner is generally permanent until erased using an ultraviolet light. After erasure, a new set of data can be stored in the EPROM. The electrically erasable PROM (or EEPROM) is a more recent development. It is similar to the EPROM, except that erasure is accomplished using electrical pulses instead of ultraviolet light. An EEPROM can be erased and reprogrammed only a limited number of times, typically 100 to 1000 times. Flash memories are similar to EEPROMs, except that they use a different charge-storage mechanism. They usually have built-in programming and erase capability so that data can be written to the flash memory while it is in place in a circuit without the need for a separate programmer.

A sequential network can easily be designed using a ROM and flip-flops. Referring to the general model of a Mealy sequential network given in Figure 1-16, the combinational part of the sequential network can be realized using a ROM. The ROM can be used to realize the output functions and the next-state functions. The state of the network can then be stored in a register of D flip-flops and fed back to the input of the ROM. Use of D flip-

flops is preferable to J-K flip-flops, since use of 2-input flip-flops would require increasing the number of outputs from the ROM. The fact that the D flip-flop input equations would generally require more gates than the J-K equations is of no consequence, since the size of the ROM depends only on the number of inputs and outputs and not on the complexity of the equations being realized. For this reason, the state assignment used is also of little importance, and generally a state assignment in straight binary order is as good as any.

We will realize the sequential machine of Figure 1-17 using a ROM and three D flip-flops (see Figure 3-2). Table 3-1 gives the truth table for the ROM, which implements the transition table of Figure 1-18(b) with the don't cares replaced by 0s. Since the ROM has four inputs, it contains  $2^4 = 16$  words. In general, a Mealy sequential network with  $i$  inputs,  $j$  outputs, and  $k$  state variables can be realized using  $k$  D flip-flops and a ROM with  $i + k$  inputs ( $2^{i+k}$  words) and  $j + k$  outputs.

**Figure 3-2 Realization of a Mealy Sequential Network with a ROM**



**Table 3-1 ROM Truth Table**

$Q_1$	$Q_2$	$Q_3$	X	$Q_1^+$	$Q_2^+$	$Q_3^+$	Z
0	0	0	0	1	0	0	1
0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	0	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	0	1	0	0
1	1	1	0	0	1	1	0
1	1	1	1	0	1	1	1

The VHDL code for the ROM realization (Figure 3-3) is similar to that of Figure 2-13, except the process that determines the next state and output has been replaced by code that reads the ROM. The state register is represented by  $Q$ , which is a 3-bit vector ( $Q_1, Q_2, Q_3$ ), and the next state of this register is  $Qplus$ . In VHDL, a ROM can be represented by a constant one-dimensional array of bit vectors. In this example, a type statement is used to declare type ROM as an array of 16 words of 4-bit vectors. A constant declaration specifies the contents of the ROM named *FSM\_ROM*. The input to the *FSM\_ROM* is  $Q$  concatenated with  $X$ . Since the index of an array must be an integer, the *vec2int* function is called to convert  $Q \& X$  to an integer. The variable *ROMValue* is set equal to the ROM output, and then *ROMValue* is split into  $Qplus$  and  $Z$ . The state register  $Q$  is updated after the rising edge of the clock.

**Figure 3-3 ROM Realization of Figure 1-17**

```
library BITLIB;
use BITLIB.bit_pack.all;

entity ROM1_2 is
  port(X,CLK: in bit;
       Z: out bit);

end ROM1_2;
architecture ROM1 of ROM1_2 is
  signal Q, Qplus: bit_vector(1 to 3) := "000";
  type ROM is array (0 to 15) of bit_vector(3 downto 0);
  constant FSM_ROM: ROM :=
    ("1001","1010","0000","0000",
     "0001","0000","0000","0001",
     "1111","1100","1100","1101",
     "0111","0100","0110","0111");
begin
  process(Q,X)           -- determines the next state and output
  variable ROMValue: bit_vector(3 downto 0);
  begin
    ROMValue := FSM_ROM(vec2int(Q & X));   -- read ROM output
    Qplus <= ROMValue(3 downto 1);
    Z <= ROMValue(0);
  end process;

  process(CLK)
  begin
    if CLK='1' then Q <= Qplus; end if;   -- update state register
  end process;
end ROM1;
```

## 3.2 PROGRAMMABLE LOGIC ARRAYS (PLAS)

A programmable logic array (PLA) performs the same basic function as a ROM. A PLA with  $n$  inputs and  $m$  outputs (Figure 3-4) can realize  $m$  functions of  $n$  variables. The internal organization of the PLA is different from that of the ROM. The decoder is replaced with an AND array that realizes selected product terms of the input variables. The OR array ORs together the product terms needed to form the output functions.

Figure 3-4 Programmable Logic Array Structure

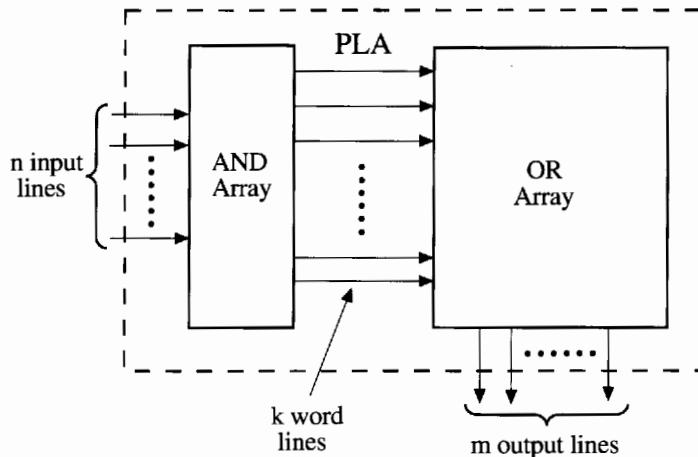


Figure 3-5 shows an nMOS PLA that realizes the following functions:

$$F_0 = \sum m(0, 1, 4, 6) = A'B' + AC' \quad (3-1)$$

$$F_1 = \sum m(2, 3, 4, 6, 7) = B + AC'$$

$$F_2 = \sum m(0, 1, 2, 6) = A'B' + BC'$$

$$F_3 = \sum m(2, 3, 5, 6, 7) = AC + B$$

Internally, the PLA uses NOR-NOR logic, but the added input and output inverting buffers make it equivalent to AND-OR logic. Logic gates are formed in the array by connecting nMOS switching transistors between the column lines and the row lines. Figure 3-6 shows the implementation of a two-input NOR gate. The transistors act as switches, so if the gate input is a logic 0, the transistor is off. If the gate input is a logic 1, the transistor provides a conducting path to ground. If  $X_1 = X_2 = 0$ , both transistors are off, and the pull-up resistor brings the  $Z$  output to a logic 1 level ( $+V$ ). If either  $X_1$  or  $X_2$  is 1, the corresponding transistor is turned on, and  $Z = 0$ . Thus,  $Z = (X_1 + X_2)' = X_1'X_2'$ , which corresponds to a NOR gate. The part of the PLA array that realizes  $F_0$  is equivalent to the NOR-NOR gate structure shown in Figure 3-7. After canceling the extra inversions, this reduces to an AND-OR structure. The AND-OR array shown in Figure 3-8 is thus equivalent to the nMOS PLA structure of Figure 3-5.

Figure 3-5 PLA with 3 Inputs, 5 Product Terms, and 4 Outputs

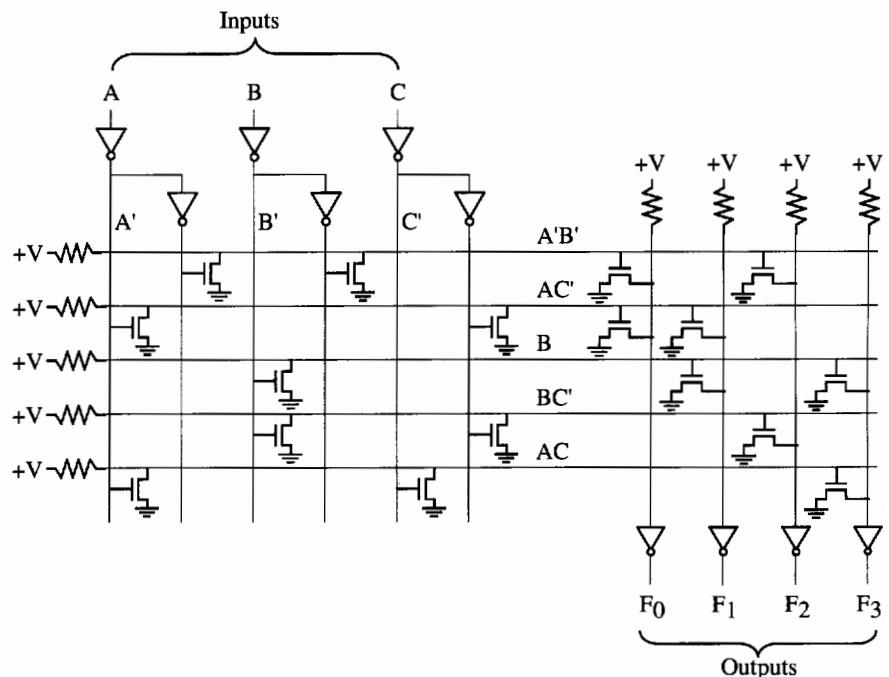


Figure 3-6 nMOS NOR Gate

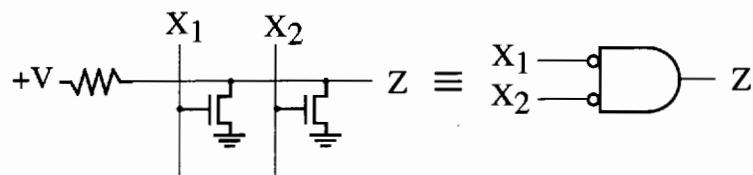


Figure 3-7 Conversion of NOR-NOR to AND-OR

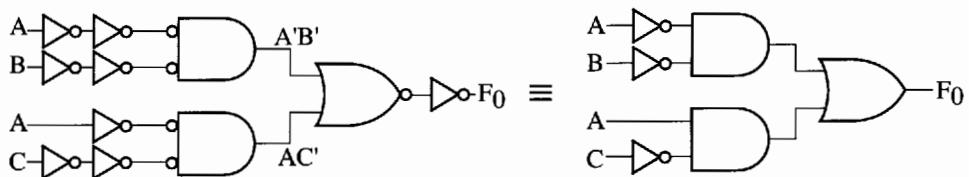
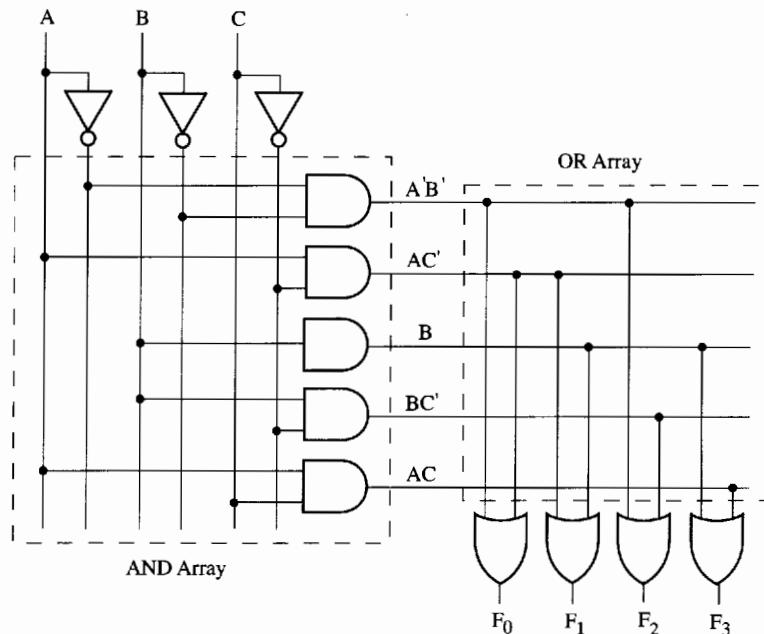


Figure 3-8 AND-OR Array Equivalent to Figure 3-5



The contents of a PLA can be specified by a modified truth table. Table 3-2 specifies the PLA in Figure 3-5. The input side of the table specifies the product terms. The symbols 0, 1, and – indicate whether a variable is complemented, not complemented, or not present in the corresponding product term. The output side of the table specifies which product terms appear in each output function. A 1 or 0 indicates whether a given product term is present or not present in the corresponding output function. Thus, the first row of Table 3-2 indicates that the term  $A'B'$  is present in output functions  $F_0$  and  $F_2$ , and the second row indicates that  $AC'$  is present in  $F_0$  and  $F_1$ .

Table 3-2 PLA Table for Figure 3-5

Product Term	Inputs			Outputs			
	A	B	C	$F_0$	$F_1$	$F_2$	$F_3$
$A'B'$	0	0	–	1	0	1	0
$AC'$	1	–	0	1	1	0	0
$B$	–	1	–	0	1	0	1
$BC'$	–	1	0	0	0	1	0
$AC$	1	–	1	0	0	0	1

Next we will realize the following functions using a PLA:

$$F_1 = \sum m(2, 3, 5, 7, 8, 9, 10, 11, 13, 15) \quad (3-2)$$

$$F_2 = \sum m(2, 3, 5, 6, 7, 10, 11, 14, 15)$$

$$F_3 = \sum m(6, 7, 8, 9, 13, 14, 15)$$

If we minimize each function separately, the result is

$$F_1 = bd + b'c + ab' \quad (3-3)$$

$$F_2 = c + a'bd$$

$$F_3 = bc + ab'c' + abd$$

If we implement these reduced equations in a PLA, a total of eight different product terms (including  $c$ ) are required.

Instead of minimizing each function separately, we want to minimize the total number of rows in the PLA table. In this case, the number of terms in each equation is not important, since the size of the PLA does not depend on the number of terms. Equations (3-3) are plotted on the Karnaugh maps shown in Figure 3-9. Since the term  $ab'c'$  is already needed for  $F_3$ , we can use it in  $F_1$  instead of  $ab'$ , since the other two 1s in  $ab'$  are covered by the  $b'c$  term. This eliminates the need to use a row of the PLA table for  $ab'$ . Since the terms  $a'bd$  and  $abd$  are needed in  $F_2$  and  $F_3$ , respectively, we can replace  $bd$  in  $F_1$  with  $a'bd + abd$ . This eliminates the need for a row to implement  $bd$ . Since  $b'c$  and  $bc$  are used in  $F_1$  and  $F_3$ , respectively, we can replace  $c$  in  $F_3$  with  $b'c + bc$ . The resulting equations (3-4) correspond to the reduced PLA table (Table 3-3). Instead of using Karnaugh maps to reduce the number of rows in the PLA, the Espresso algorithm can be used. This complex algorithm is described in *Logic Minimization Algorithms for VLSI Synthesis* by Brayton [10].

**Figure 3-9** Multiple-Output Karnaugh Maps

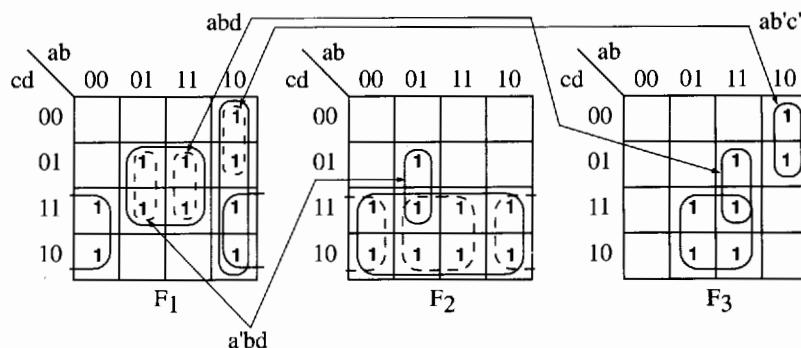


Table 3-3 Reduced PLA Table

$a$	$b$	$c$	$d$	$F_1$	$F_2$	$F_3$
0	1	-	1	1	1	0
1	1	-	1	1	0	1
1	0	0	-	1	0	1
-	0	1	-	1	1	0
-	1	1	-	0	1	1

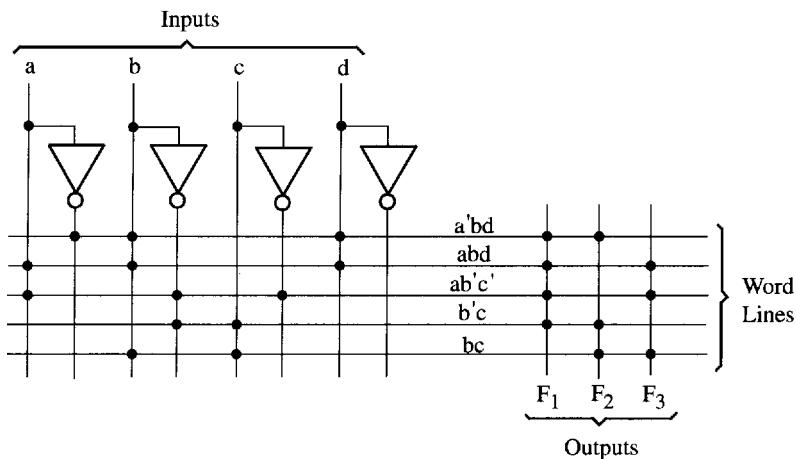
$$F_1 = a'bd + abd + ab'c' + b'c \quad (3-4)$$

$$F_2 = a'bd + b'c + bc$$

$$F_3 = abd + ab'c' + bc$$

Equations (3-4) have only 5 different product terms, so the PLA table has only five rows. This is a significant improvement over equations (3-3), which require 8 product terms. Figure 3-10 shows the corresponding PLA structure, which has 4 inputs, 5 product terms, and 3 outputs. A dot at the intersection of a word line and an input or output line indicates the presence of a switching element in the array.

Figure 3-10 PLA Realization of Equations (3-4)



A PLA table is significantly different than a truth table for a ROM. In a truth table each row represents a minterm; therefore, exactly one row will be selected by each combination of input values. The 0s and 1s of the output portion of the selected row determine the corresponding output values. On the other hand, each row in a PLA table represents a general product term. Therefore, zero, one, or more rows may be selected by each combination of input values. To determine the value of  $F$  for a given input combination, the values of  $F$  in the selected rows of the PLA table must be ORed together. The following examples refer to the PLA table of Table 3-3. If  $abcd = 0001$ , no rows are selected; and all  $F_i$ 's are 0. If  $abcd = 1001$ , only the third row is selected, and  $F_1 F_2 F_3 = 101$ . If  $abcd = 0111$ , the first and fifth rows are selected. Therefore,  $F_1 = 1 + 0 = 1$ ,  $F_2 = 1 + 1 = 1$ , and  $F_3 = 0 + 1 = 1$ .

Next we realize the sequential machine of Figure 1-17 using a PLA and three D flip-flops. The network structure is the same as Figure 3-2, except that the ROM is replaced by a PLA. The required PLA table, based on the equations given in Figure 1-19, is Table 3-4.

**Table 3-4 PLA Table**

Product Term	$Q_1$	$Q_2$	$Q_3$	$X$	$Q_1^+$	$Q_2^+$	$Q_3^+$	$Z$
$\bar{Q}_2'$	–	0	–	–	1	0	0	0
$Q_1$	1	–	–	–	0	1	0	0
$Q_1Q_2Q_3$	1	1	1	–	0	0	1	0
$Q_1Q_3X'$	1	–	0	0	0	0	1	0
$Q_1Q_2X$	0	0	–	1	0	0	1	0
$Q_3X'$	–	–	0	0	0	0	0	1
$Q_3X$	–	–	1	1	0	0	0	1

Reading the output of a PLA in VHDL is somewhat more difficult than reading the ROM output. Since the input to the PLA can match several rows, and the outputs from those rows must be ORed together, a function is required to sequentially scan the PLA array and determine the PLA output.

In Figure 3-11, we represent the PLA as a two-dimensional array of type *PLAtrx* and use a constant declaration to specify the contents of the *FSM\_PLA*. The function *PLAout* is called to determine the output of *FSM\_PLA* when the input is  $Q \& X$ , and this output is assigned to the variable *PLAValue*. After splitting *PLAValue* into *Qplus* and *Z*, *Q* is updated on the rising edge of the clock. The *PLAout* function is explained in Section 8.5.

Because of the –'s in the PLA table input section, multiple-valued logic is required for the PLA computation. We could define a VHDL type with elements 0, 1, and –; however, it is preferable to use a predefined type. We have used IEEE std\_logic type, which is defined in the IEEE library. The std\_logic type has nine values, including '0', '1', and 'X'. When we inserted the PLA table into the VHDL code, we used 'X' to represent a dash (–). We placed the type declaration for *PLAtrx* and the function *PLAout* in the *MVLLIB* (multivalued logic library) and also make use of the IEEE standard logic package. Details of using the multivalued logic and the standard logic package are discussed in Chapter 8.

Figure 3-11 PLA Realization of Figure 1-17

```

library ieee;
use ieee.std_logic_1164.all;          -- IEEE standard logic package
library MVLLIB;
use MVLLIB.mvl_pack.all;             -- includes PLAmtrx type and
                                      -- PLAout function

entity PLA1_2 is
  port(X,CLK: in std_logic;
       Z: out std_logic);
end PLA1_2;

architecture PLA of PLA1_2 is
signal Q, Qplus: std_logic_vector(1 to 3) := "000";
constant FSM_PLA: PLAmtrx(0 to 6, 7 downto 0) :=
  ("X0XX1000",
   "1XXX0100",
   "111X0010",
   "1X000010",
   "00X10010",
   "XX000001",
   "XX110001");

begin
  process(Q,X)
  variable PLAValue: std_logic_vector(3 downto 0);
  begin
    PLAValue := PLAout(FSM_PLA,Q & X);      -- read PLA output
    Qplus <= PLAValue(3 downto 1);
    Z <= PLAValue(0);
  end process;

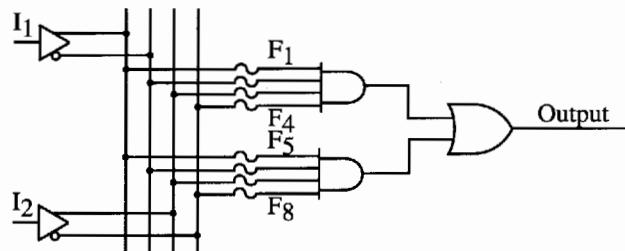
  process(CLK)
  begin
    if CLK='1' then Q <= Qplus; end if;      -- update state register
  end process;
end PLA;

```

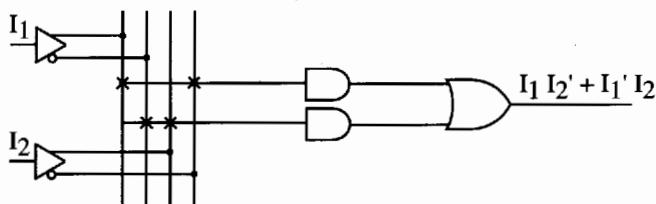
### 3.3 PROGRAMMABLE ARRAY LOGIC (PALS)

The PAL (programmable array logic) is a special case of the programmable logic array in which the AND array is programmable and the OR array is fixed. The basic structure of the PAL is the same as the PLA shown in Figure 3-4. Because only the AND array is programmable, the PAL is less expensive than the more general PLA, and the PAL is easier to program. For this reason, logic designers frequently use PALs to replace individual logic gates when several logic functions must be realized.

Figure 3-12 Combinational PAL Segment

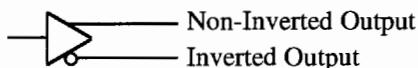


(a) Unprogrammed

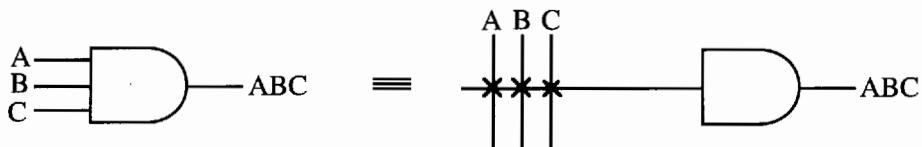


(b) Programmed

Figure 3-12(a) represents a segment of an unprogrammed PAL. The symbol



represents an input buffer with noninverted and inverted outputs. A buffer is used, since each PAL input must drive many AND gate inputs. When the PAL is programmed, the fusible links ( $F_1, F_2, \dots, F_8$ ) are selectively blown to leave the desired connections to the AND gate inputs. Connections to the AND gate inputs in a PAL are represented by xs, as shown here:



As an example, we will use the PAL segment of Figure 3-12(a) to realize the function  $I_1I_2 + I_1'I_2$ . The xs indicate that the  $I_1$  and  $I_2$  lines are connected to the first AND gate, and the  $I_1'$  and  $I_2'$  lines are connected to the other gate (see Figure 3-12(b)).

Typical combinational PALs have from 10 to 20 inputs and from 2 to 10 outputs, with 2 to 8 AND gates driving each OR gate. PALs are also available that contain D flip-flops with inputs driven from the programmable array logic. Such PALs provide a convenient way of realizing sequential networks. Figure 3-13 shows a segment of a sequential PAL. The D flip-flop is driven from an OR gate, which is fed by two AND gates. The flip-flop output is fed back to the programmable AND array through a buffer. Thus the AND gate inputs can be connected to  $A, A', B, B', Q$ , or  $Q'$ . The  $\times$ s on the diagram show the realization of the next-state equation

$$Q^+ = D = A'BQ' + AB'Q$$

The flip-flop output is connected to an inverting tristate buffer, which is enabled when  $EN = 1$ .

**Figure 3-13 Segment of a Sequential PAL**

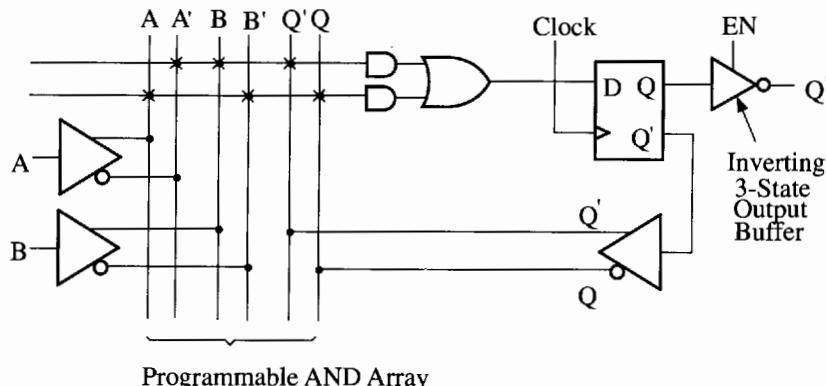
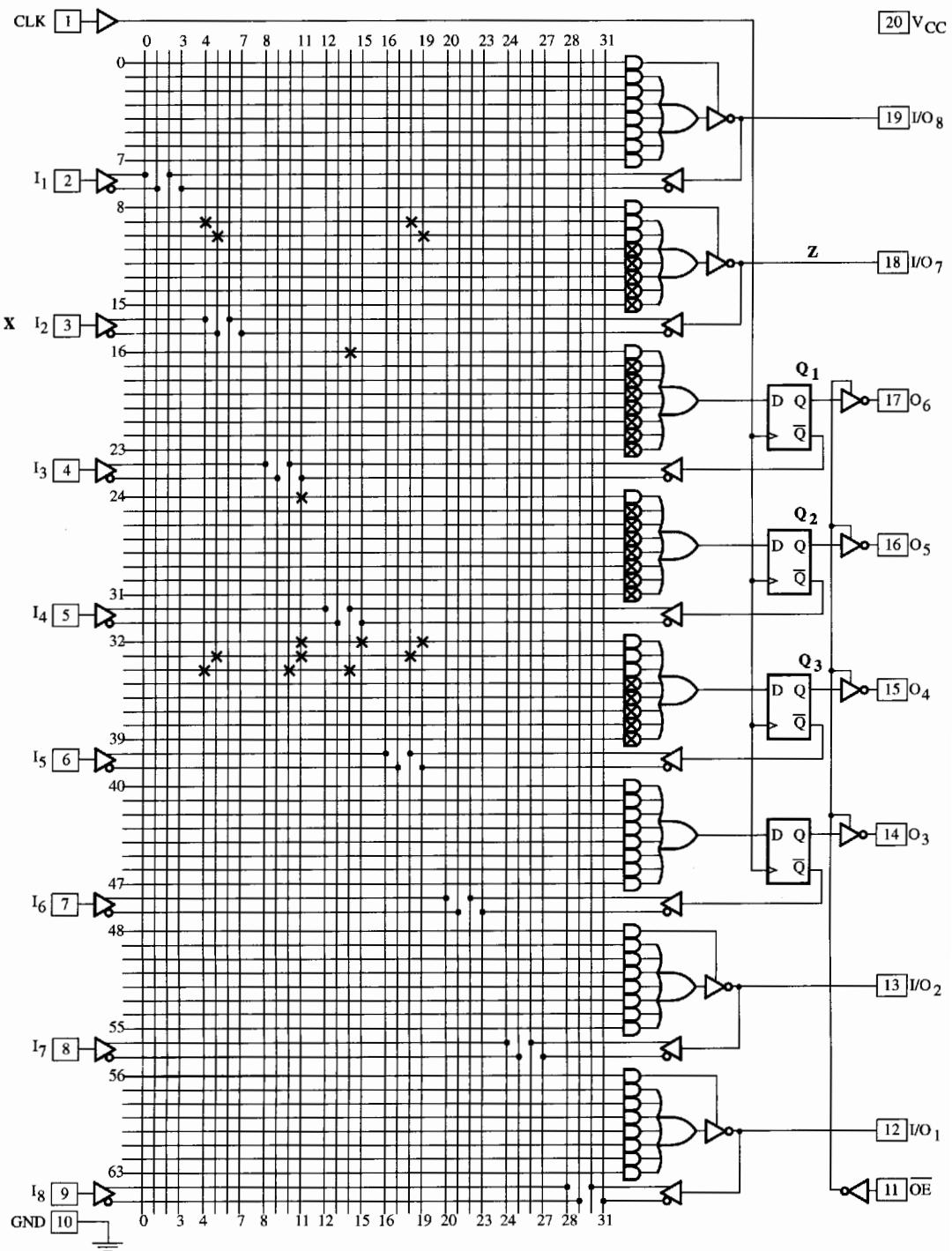


Figure 3-14 shows a logic diagram for a typical sequential PAL, the 16R4. This PAL has an AND gate array with 16 input variables, and it has 4 D flip-flops. Each flip-flop output goes through a tristate inverting buffer (output pins 14–17). One input (pin 11) is used to enable these buffers. The rising edge of a common clock (pin 1) causes the flip-flops to change state. Each D flip-flop input is driven from an OR gate, and each OR gate is fed from 8 AND gates. The AND gate inputs can come from the external PAL inputs (pins 2–9) or from the flip-flop outputs, which are fed back internally. In addition there are 4 input/output (I/O) terminals (pins 12, 13, 18, and 19), which can be used as either network outputs or as inputs to the AND gates. Thus, each AND gate can have a maximum of 16 inputs (8 external inputs, 4 inputs fed back from the flip-flop outputs, and 4 inputs from the I/O terminals). When used as an output, each I/O terminal is driven from an inverting tristate buffer. Each of these buffers is fed from an OR gate and each OR gate is fed from 7 AND gates. An eighth AND gate is used to enable the buffer.

When the 16R4 PAL is used to realize a sequential network, the I/O terminals are normally used for the Z outputs. Thus, a single 16R4 with no additional logic could realize a sequential network with up to 8 inputs, 4 outputs, and 16 states. Each next-state equation could contain up to 8 terms, and each output equation could contain up to 7 terms. As an

Figure 3-14 Logic Diagram for 16R4 PAL



example, we realize the code converter of Figure 1-20. Three flip-flops are used to store  $Q_1$ ,  $Q_2$ , and  $Q_3$ , and the array logic that drives these flip-flops is programmed to realize  $D_1$ ,  $D_2$ , and  $D_3$ , as shown in Figure 3-14. The  $\times$ s on the diagram indicate the connections to the AND-gate inputs. An  $\times$  inside an AND gate indicates that the gate is not used. For  $D_3$ , three AND gates are used, and the function realized is

$$D_3 = Q_1 Q_2 Q_3 + X' Q_1 Q_3' + X Q_1' Q_2'$$

The flip-flop outputs are not used externally, so the output buffers are disabled. Since the  $Z$  output comes through an inverting buffer, the array logic must realize

$$Z' = (X + Q_3)(X' + Q_3') = XQ_3' + X'Q_3$$

The  $Z$  output buffer is permanently enabled in this example, so there are no connections to the AND gate that drives the enable input, in which case the AND gate output is a logic 1.

When designing with PALs, we must simplify our logic equations and try to fit them into one (or more) of the available PALs. Unlike the more general PLA, the AND terms cannot be shared among two or more OR gates; therefore, each function to be realized can be simplified by itself without regard to common terms. For a given type of PAL, the number of AND terms that feed each output OR gate is fixed and limited. If the number of AND terms in a simplified function is too large, we may be forced to choose a PAL with more OR-gate inputs and fewer outputs.

Computer-aided design programs for PALs are widely available. Such programs accept logic equations, truth tables, state graphs, or state tables as inputs and automatically generate the required fuse patterns. These patterns can then be downloaded into a PLD programmer, which will blow the required fuses and verify the operation of the PAL. Many of the newer types of PLDs are erasable and reprogrammable in a manner similar to EPROMs and EEPROMs.

### 3.4 OTHER SEQUENTIAL PROGRAMMABLE LOGIC DEVICES (PLDS)

The 16R4 is an example of a simple sequential PLD. As integrated circuit technology has improved, a wide variety of other PLDs have become available. Some of these are based on extensions of the PAL concept, and others are based on gate arrays. Programmable gate arrays and other complex PLDs are discussed in Chapter 6.

The 22CEV10 (Figure 3-15) is a CMOS electrically erasable PLD that can be used to realize both combinational and sequential networks. It has 12 dedicated input pins and 10 pins that can be programmed as either inputs or outputs. It contains 10 D flip-flops and 10 OR gates. The number of AND gates that feed each OR gate ranges from 8 through 16. Each OR gate drives an *output logic macrocell*. Each macrocell contains one of the 10 D flip-flops. The flip-flops have a common clock, a common asynchronous reset (AR) input, and a common synchronous preset (SP) input. The name 22V10 indicates a versatile PAL with a total of 22 input and output pins, 10 of which are bidirectional I/O (input/output) pins.

**Figure 3-15 Block Diagram for 22V10**

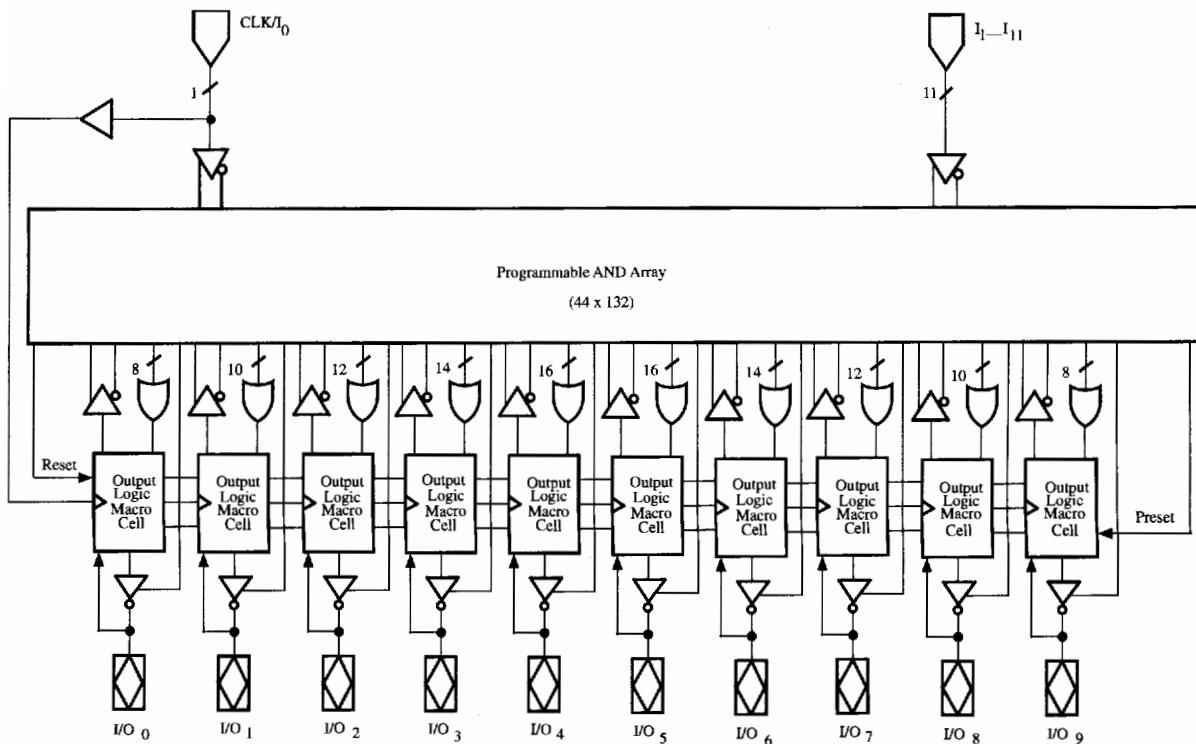
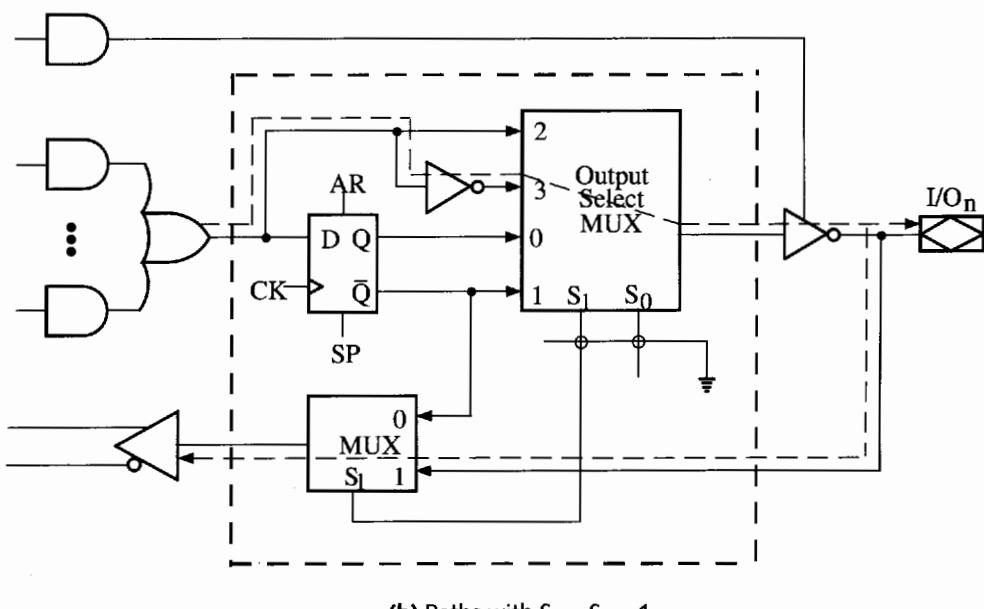
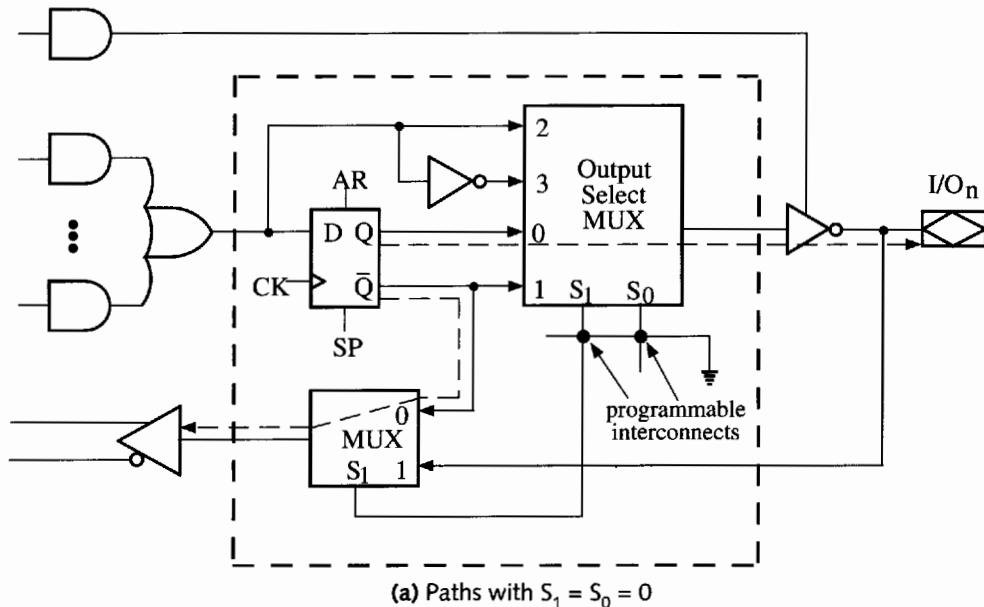


Figure 3-16 shows the details of a 22CEV10 output macrocell. The connections to the output pins are controlled by programming this macrocell. The output MUX control inputs  $S_1$  and  $S_0$  select one of the data inputs. For example,  $S_1S_0 = 10$  selects data input 2. Each macrocell has two programmable interconnect bits.  $S_1$  or  $S_0$  is connected to ground (logic 0) when the corresponding bit is programmed. Erasing a bit disconnects the control line ( $S_1$  or  $S_0$ ) from ground and allows it to float to Vcc (logic 1). When  $S_1 = 1$ , the flip-flop is bypassed, and the output is from the OR gate. The OR gate output is connected to the I/O pin through the multiplexer and the output buffer. The OR gate is also fed back so that it can be used as an input to the AND gate array. If  $S_1 = 0$ , then the flip-flop output is connected to the output pin, and it is also fed back so that it can be used for AND gate inputs. When  $S_0 = 1$ , the output is not inverted, so it is active high. When  $S_0 = 0$ , the output is inverted, so it is active low. The output pin is driven by a tristate inverting buffer. When the buffer output is in the high-impedance state, the OR gate and flip-flop are disconnected from the output pin, and the pin can be used as an input. The dashed lines on Figure 3-16(a) show the path through the output macrocell when both  $S_1$  and  $S_0$  are 0, and the dashed lines on Figure 3-16(b) show the path when both  $S_1$  and  $S_0$  are 1. Note that in the first case, the flip-flop  $Q$  output is inverted by the output buffer, and in the second case the OR gate output is inverted twice, so there is no net inversion.

Table 3-5 gives the characteristics of several PLDs that are similar to the 22V10. All these PLDs have output macrocells, and each macrocell has one D flip-flop. The I/O pins can be programmed so that they act as inputs or as combinational or flip-flop outputs. Some of the PLDs have a dedicated clock input (listed as *clk* in the table), and the others have a dual-purpose pin that can be used either as a clock or as an input. All of the PLDs have tristate buffers at the outputs, and some of them have a dedicated output enable ( $\overline{OE}$ ).

**Table 3-5 Characteristics of Simple CMOS PLDs**

Type No.	No. of Inputs	I/O	Macrocells = FFs	AND Gates per OR Gate
PALCE16V10	8 + $\overline{OE}$ + clk	8	8	8
PALCE20V8	14	8	8	8
PALCE22V10	12	10	10	8–16
PALCE24V10	14	10	10	8
PALCE29MA16	5 + clk	16	16	4–12
CY7C335	12 + $\overline{OE}$ + clk	12	12 in/12 out	9–19

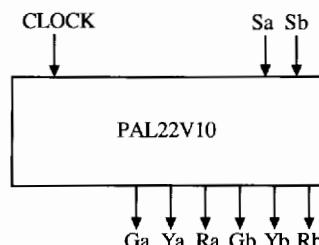
**Figure 3-16 Output Macrocell**

## Traffic-Light Controller

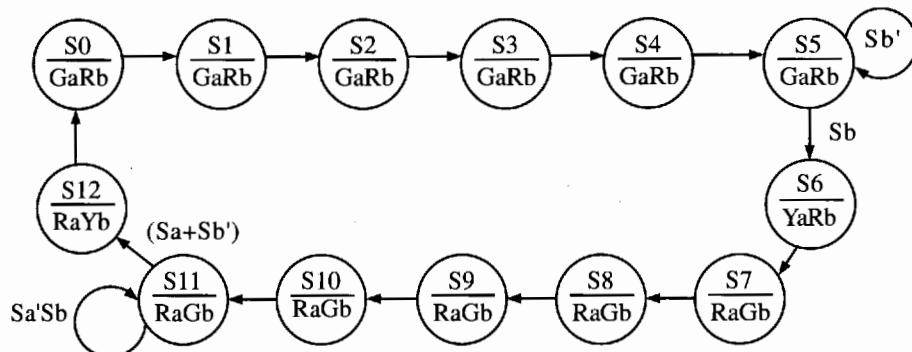
As an example of using the 22V10, we design a sequential traffic-light controller for the intersection of “A” street and “B” street. Each street has traffic sensors, which detect the presence of vehicles approaching or stopped at the intersection.  $S_a = 1$  means a vehicle is approaching on “A” street, and  $S_b = 1$  means a vehicle is approaching on “B” street. “A” street is a main street and has a green light until a car approaches on “B”. Then the light changes, and “B” has a green light. At the end of 50 seconds, the lights change back unless there is a car on “B” street and none on “A”, in which case the “B” cycle is extended 10 more seconds. When “A” is green, it remains green at least 60 seconds, and then the lights change only when a car approaches on “B”. Figure 3-17 shows the external connections to the controller. Three of the outputs ( $G_a$ ,  $Y_a$ , and  $R_a$ ) drive the green, yellow, and red lights on “A” street. The other three ( $G_b$ ,  $Y_b$ , and  $R_b$ ) drive the corresponding lights on “B” street.

Figure 3-18 shows a Moore state graph for the controller. For timing purposes, the sequential network is driven by a clock with a 10-second period. Thus, a state change can occur at most once every 10 seconds. The following notation is used:  $G_a R_b$  in a state means that  $G_a = R_b = 1$  and all the other output variables are 0.  $S_a' S_b$  on an arc implies that  $S_a = 0$  and  $S_b = 1$  will cause a transition along that arc. An arc without a label implies that a state transition will occur when the clock occurs, independent of the input variables. Thus, the green “A” light will stay on for 6 clock cycles (60 seconds) and then change to yellow if a car is waiting on “B” street.

**Figure 3-17** Block Diagram of Traffic-Light Controller



**Figure 3-18** State Graph for Traffic-Light Controller



The VHDL code for the traffic-light controller (Figure 3-19) represents the state machine by two processes. Whenever the state,  $S_a$ , or  $S_b$  changes, the first process updates the outputs and nextstate. When the rising edge of the clock occurs, the second process updates the state register. The case statement illustrates use of a when clause with a range. Since states S0 through S4 have the same outputs, and the next states are in numeric sequence, we use a when clause with a range instead of five separate when clauses:

```
when 0 to 4 => Ga <= '1'; Rb <= '1'; nextstate <= state + 1;
```

**Figure 3-19 VHDL Code for Traffic-Light Controller**

```
entity traffic_light is
  port (clk, Sa, Sb: in bit;
        Ra, Rb, Ga, Gb, Ya, Yb: out bit);
end traffic_light;

architecture behave of traffic_light is
  signal state, nextstate: integer range 0 to 12;
  type light is (R, Y, G);
  signal lightA, lightB: light; -- define signals for waveform output
begin
  process(state, Sa, Sb)
  begin
    Ra <= '0'; Rb <= '0'; Ga <= '0'; Gb <= '0'; Ya <= '0'; Yb <= '0';
    case state is
      when 0 to 4 => Ga <= '1'; Rb <= '1'; nextstate <= state+1;
      when 5 => Ga <= '1'; Rb <= '1';
        if Sb = '1' then nextstate <= 6; end if;
      when 6 => Ya <= '1'; Rb <= '1'; nextstate <= 7;
      when 7 to 10 => Ra <= '1'; Gb <= '1'; nextstate <= state+1;
      when 11 => Ra <= '1'; Gb <= '1';
        if (Sa='1' or Sb='0') then nextstate <= 12; end if;
      when 12 => Ra <= '1'; Yb <= '1'; nextstate <= 0;
    end case;
  end process;
  process(clk)
  begin
    if clk = '1' then
      state <= nextstate;
    end if;
  end process;
  lightA <= R when Ra='1' else Y when Ya='1' else G when Ga='1';
  lightB <= R when Rb='1' else Y when Yb='1' else G when Gb='1';
end behave;
```

For each state, only the signals that are '1' are listed within the case statement. Since in VHDL a signal will hold its value until it is changed, we should turn off each signal when the next state is reached. In state 6 we should set *Ga* to '0', in state 7 we should set *Ya* to '0', etc. This could be accomplished by inserting appropriate statements in the when clauses. For example, we could insert *Ga* <= '0' in the when 6 => clause. An easier way to turn off the outputs is to set them all to '0' before the case statement, as shown in Figure 3-19. At first, it seems that a glitch might occur in the output when we set a signal to '0' that should remain '1'. However, this is not a problem, because the sequential statements within a process execute instantaneously. For example, suppose that at time = 30 a state change from S2 to S3 occurs. *Ga* and *Rb* are '1', but as soon as the process starts executing, the first line of code is executed and *Ga* and *Rb* are scheduled to change to '0' at time  $20 + \Delta$ . The case statement then executes, and *Ga* and *Rb* are scheduled to change to '1' at time  $20 + \Delta$ . Since this is the same time as before, the new value ('1') preempts the previously scheduled value ('0'), and the signals never change to '0'.

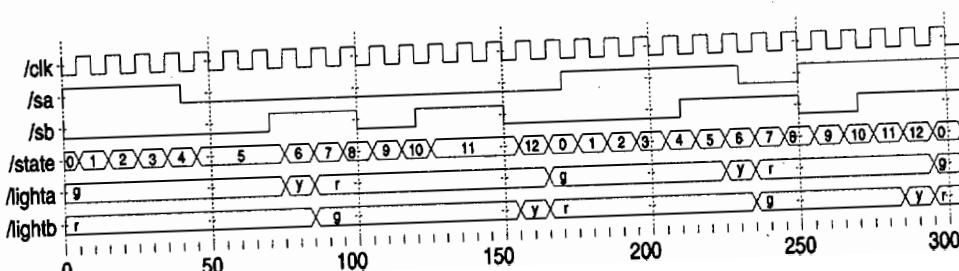
Before completing the design of the traffic controller, we will test the VHDL code to see that it meets specifications. As a minimum, our test sequence should cause all of the arcs on the state graph to be traversed at least once. We may want to perform additional tests to check the timing for various traffic conditions, such as heavy traffic on both "A" and "B", light traffic on both, heavy traffic on "A" only, heavy traffic on "B" only, and special cases such as a car fails to move when the light is green, a car goes through the intersection when the light is red, etc.

To make it easier to interpret the simulator output, we define a type named *light* with the values R, Y, and G and two signals, *lightA* and *lightB*, which can assume these values. Then we add code to set *lightA* to R when the light is red, to Y when the light is yellow, and to G when the light is green. The following simulator command file first tests the case where both self-loops on the graph are traversed and then the case where neither self-loop is traversed:

```
wave clk SA SB state lightA lightB
force clk 0 0, 1 5 sec -repeat 10 sec
force SA 1 0, 0 40, 1 170, 0 230, 1 250 sec
force SB 0 0, 1 70, 0 100, 1 120, 0 150, 1 210, 0 250, 1 270 sec
```

The test results in Figure 3-20 verify that the traffic lights change at the specified times.

Figure 3-20 Test Results for Traffic-Light Controller



**Table 3-6 State Table for Traffic-Light Controller**

	SaSb				Ga	Ya	Ra	Gb	Yb	Rb
	00	01	10	11						
S0	S1	S1	S1	S1	1	0	0	0	0	1 {Green A, Red B}
S1	S2	S2	S2	S2	1	0	0	0	0	1
S2	S3	S3	S3	S3	1	0	0	0	0	1
S3	S4	S4	S4	S4	1	0	0	0	0	1
S4	S5	S5	S5	S5	1	0	0	0	0	1
S5	S5	S6	S5	S6	1	0	0	0	0	1
S6	S7	S7	S7	S7	0	1	0	0	0	1 {Ya, Rb}
S7	S8	S8	S8	S8	0	0	1	1	0	0 {Ra, Gb}
S8	S9	S9	S9	S9	0	0	1	1	0	0
S9	S10	S10	S10	S10	0	0	1	1	0	0
S10	S11	S11	S11	S11	0	0	1	1	0	0
S11	S12	S11	S12	S12	0	0	1	1	0	0
S12	S0	S0	S0	S0	0	0	1	0	1	0 {Ra, Yb}

Table 3-6 shows the state table for the controller. We implement the table using four D flip-flops with inputs  $D_1, D_2, D_3, D_4$  and outputs  $Q_1, Q_2, Q_3, Q_4$ . Using a straight binary state assignment, the following equations were derived from the table with a logic design program:

$$D_1 = Q_1 Q_2' + Q_2 Q_3 Q_4$$

$$D_2 = Q_1' Q_2' Q_3 Q_4 + S_a Q_1 Q_3 Q_4 + S_b' Q_1 Q_3 Q_4 + Q_1' Q_2 Q_4 + Q_1' Q_2 Q_3'$$

$$D_3 = Q_3 Q_4' + S_b Q_3' Q_4 + Q_2' Q_3' Q_4 + S_a' S_b Q_1 Q_4$$

$$D_4 = S_a' S_b Q_1 Q_3 + Q_2' Q_4' + Q_1' Q_4' + S_a S_b' Q_2' Q_3' Q_4$$

$$Ga = Q_1' Q_3' + Q_1' Q_2' \quad Ya = Q_2 Q_3 Q_4' \quad Ra = Q_1 + Q_2 Q_3 Q_4$$

$$Gb = Q_1 Q_2' + Q_2 Q_3 Q_4 \quad Yb = Q_1 Q_2 \quad Rb = Q_1' Q_2' + Q_1' Q_4' + Q_1' Q_3'$$

Since all these equations have fewer than eight AND terms, they will easily fit in the 22V10. If some equations had so many terms that they would not fit in the 22V10, then it would be necessary to try different state assignments.

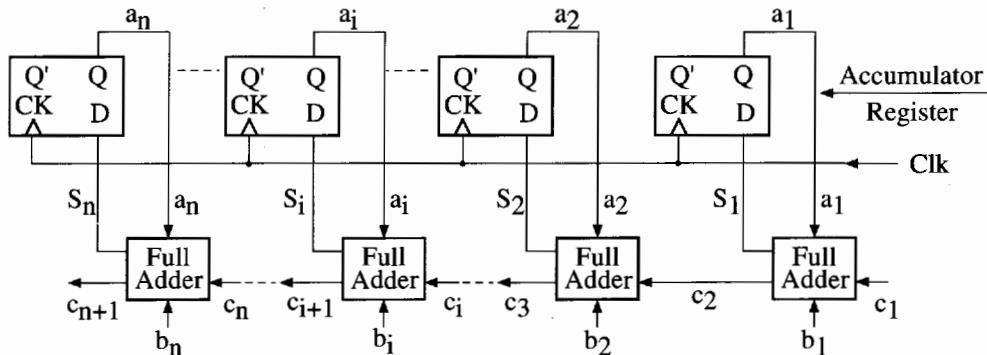
If the controller were implemented using J-K flip-flops and gates, 4 flip-flops and 34 gates would be required for the straight binary state assignment. These flip-flops and gates would require about 11 SSI (small-scale integration) integrated circuits, compared with only 1 IC for the 22V10 solution. Using the 22V10 leads to a simpler design that avoids the necessity of laying out a circuit board and wiring the ICs. The resulting circuit is smaller, uses less power, and is more reliable.

### Parallel Adder with Accumulator

Next we implement a parallel binary adder with accumulator using a 22V10. Figure 3-21 shows a block diagram for an adder that adds an  $n$ -bit number,  $B = b_n \dots b_2 b_1$ , to the accumulator,  $A = a_n \dots a_2 a_1$ , to give an  $n$ -bit sum and a carry. The full adder equations are given by Equations 1-2 and 1-3. First, the number  $A$  must be loaded into the accumulator, and then the number  $B$  is applied to the full adder inputs. After the carry has propagated through the adders, the sum appears at the adder outputs. Then a clock pulse ( $Clk$ ) transfers the adder outputs into the accumulator. One way to load  $A$  into the accumulator is to first

clear the accumulator using the clear inputs on the flip-flops and then put the  $A$  data on the  $B$  inputs and add.

**Figure 3-21** Parallel Adder with Accumulator



Next we will modify the design so that addition occurs only when an add signal ( $Ad$ ) is 1. One way to do this is to gate the clock so that the flip-flop clock inputs are clocked only when  $Ad = 1$  (see Figure 1-35(a)). A better way, which does not require gating the clock, is to modify the accumulator flip-flop input equations to include  $Ad$ :

$$a_i^+ = S_i = Ad(a_i'b_i'c_i + a_i'b_i'c'_i + a_i'b_i'c_i + a_i'b_i'c'_i) + Ad'a_i \quad (3-5)$$

so that  $a_i$  does not change when  $Ad = 0$ . No change is required for  $c_{i+1}$ .

How many bits of the adder and accumulator will fit into a 22V10? We must consider several limits. Let the number of flip-flops equal  $F$ , and the number of additional combinational functions equal  $C$ . Since the number of macrocells is 10,

$$F + C \leq 10 \quad (3-6)$$

Since there are 12 dedicated inputs and any of the unused macrocells can be used as inputs, the number of external inputs ( $I$ ) is

$$I \leq 12 + [10 - (F + C)] = 22 - F - C \quad (3-7)$$

In addition, we must make sure that the number of AND terms required for the D flip-flop input functions and the combinational functions does not exceed the available number of AND gates. Each bit of the adder requires one flip-flop, and the D input to this flip-flop is  $S_i$ . In addition, we must generate the carry for each bit, which uses up another macrocell. The  $c_{i+1}$  function must be fed back to the AND array through the macrocell, even if an external carry output is not required. For an  $N$ -bit adder,  $F = C = N$ ; thus, from Equation 3-6,  $2N \leq 10$ . The number of inputs is  $N$  plus one each for the clock, clear, carry in ( $c_1$ ), and  $Ad$  signals; thus from Equation 3-7,

$$I = N + 4 \leq 22 - 2N$$

Solving these inequalities gives  $N \leq 5$ . The operation of an adder implemented in this manner will be rather slow because of the time it takes for the carry to propagate through five AND-OR sections of the 22V10. One way to speed up operation of the adder and at the same time increase the number of bits that can be implemented in one 22V10 is to implement the adder in blocks of 2 bits at a time with no intermediate carry generated. The partial truth table and equations for such a 2-bit adder are

$A_2 A_1 B_2 B_1 C_1$	$C_3 S_2 S_1$
0 0 0 0 0	0 0 0
0 0 0 0 1	0 0 1
0 0 0 1 0	0 0 1
0 0 0 1 1	0 1 0
0 0 1 0 0	0 1 0
- - - - -	- - -
1 1 0 1 0	1 0 0
1 1 0 1 1	1 0 1
1 1 1 0 0	1 0 1
1 1 1 0 1	1 1 0
1 1 1 1 0	1 1 0
1 1 1 1 1	1 1 1

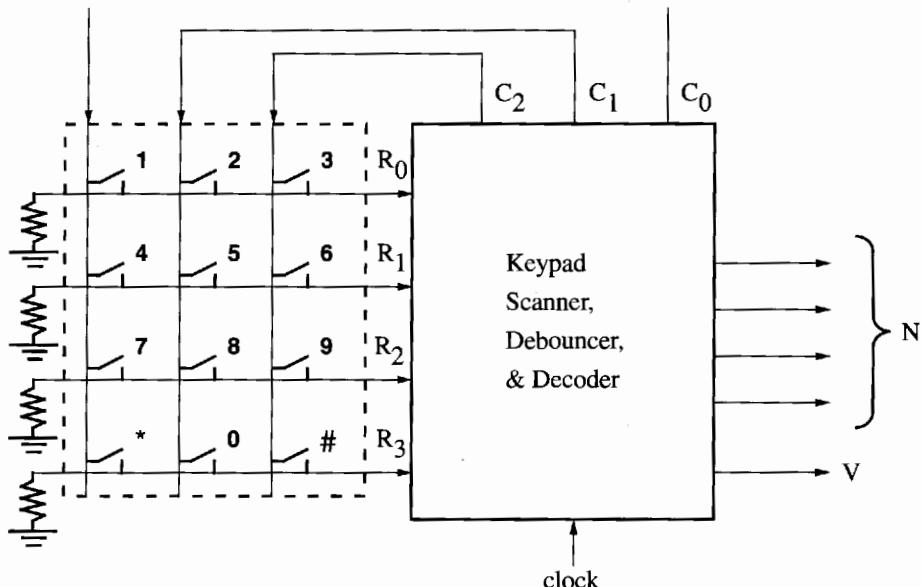
$C_3 = B_1 B_2 C_1 + A_1 B_2 C_1 + A_1 B_2 B_1 + A_2 B_1 C_1 + A_2 B_2 + A_1 A_2 C_1 + A_1 A_2 B_1$   
 $S_2 = (A'_2 B_1 B'_2 C_1 + A'_1 A'_2 B'_1 B_2 + A'_1 A'_2 B_2 C'_1 + A'_1 A'_2 B'_2 C_1 + A_1 A'_2 B'_1 B'_2 + A'_2 B'_1 B'_2 C'_1 + A'_1 A'_2 B'_1 B'_2 + A'_1 A'_2 B'_2 C'_1 + A_2 B_1 B'_2 C_1 + A_2 B'_1 B'_2 C'_1 + A_1 A_2 B'_2 C_1 + A_1 A_2 B'_2 C'_1) Ad + Ad' A_2$   
 $S_1 = (A'_1 B'_1 C_1 + A'_1 B_1 C'_1 + A_1 B'_1 C'_1 + A_1 B_1 C_1) Ad + Ad' A_1$

Since the longest equation requires 13 AND terms and the maximum number of AND terms for the 22V10 is 16, these equations will easily fit in a 22V10. We can fit three 2-bit adders in a 22V10 since  $F + C = 6 + 3 = 9 \leq 10$  and  $I = 6 + 4 \leq 22 - 9$ . With this implementation, the carry must propagate through only three AND-OR sections, so this 6-bit adder is faster than the 5-bit adder previously designed.

### 3.5 DESIGN OF A KEYPAD SCANNER

In this section, we design a scanner for a telephone keypad using a PLD. Figure 3-22 shows a block diagram for the system. The keypad is wired in matrix form with a switch at the intersection of each row and column. Pressing a key establishes a connection between a row and column. The purpose of the scanner is to determine which key has been pressed and output a binary number  $N = N_3 N_2 N_1 N_0$ , which corresponds to the key number. For example, pressing key 5 will output 0101, pressing the \* key will output 1010, and pressing the # key will output 1011. When a valid key has been detected, the scanner should output a signal  $V$  for one clock time. We will assume that only one key is pressed at time. Resistors to ground are connected to each row of the keyboard, so that  $R_1 = R_2 = R_3 = R_4 = 0$  when no key is pressed.

Figure 3-22 Block Diagram for Keypad Scanner



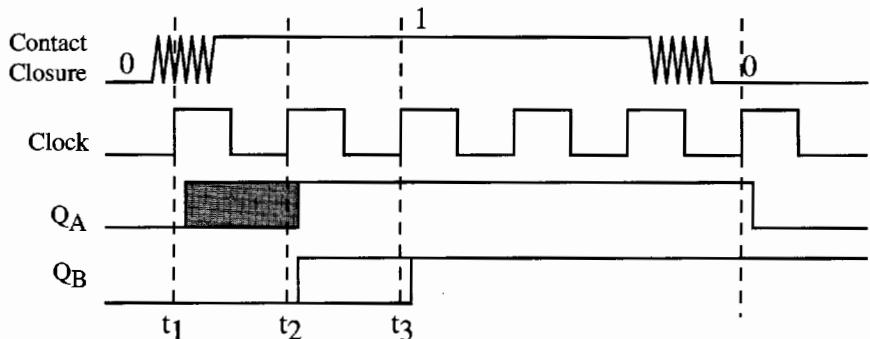
We will use the following procedure to scan the keyboard: First apply logic 1s to columns  $C_0$ ,  $C_1$ , and  $C_2$  and wait. If any key is pressed, a 1 will appear on  $R_0$ ,  $R_1$ ,  $R_2$ , or  $R_3$ . Then apply a 1 to column  $C_0$  only. If any of the  $R_i$ 's is 1, a valid key is detected, so set  $V = 1$  and output the corresponding  $N$ . If no key is detected in the first column, apply a 1 to  $C_1$  and repeat. If no key is detected in the second column, repeat for  $C_2$ . When a valid key is detected, apply 1s to  $C_0$ ,  $C_1$ , and  $C_2$  and wait until no key is pressed. This last step is necessary so that only one valid signal is generated each time a key is pressed.

In the process of scanning the keyboard to determine which key is pressed, the scanner must take contact bounce into account. When a mechanical switch is closed or opened, the switch contact will bounce, causing noise in the switch output, as shown in Figure 3-23(a). The contact may bounce for several milliseconds before it settles down to its final position. After a switch closure has been detected, we must wait for the bounce to settle before reading the key. The signal that indicates a key has been pressed also should be synchronized with the clock, since it will be used as an input to a synchronous sequential network.

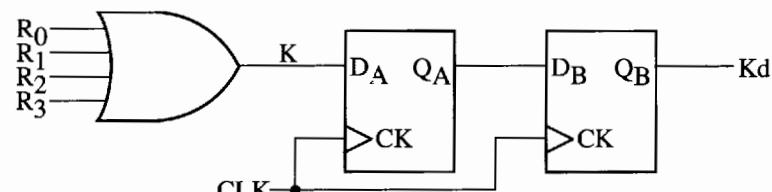
Figure 3-23(b) shows a proposed debouncing and synchronizing circuit. The clock period must be greater than the bounce time. If  $C_0 = C_1 = C_2 = 1$ , when any key is pressed,  $K$  will become 1 after the bounce is settled. If the rising edge of the clock occurs during the bounce, either a 0 or 1 will be clocked into the flip-flop at  $t_1$ . If a 0 was clocked in, a 1 will be clocked in at the next active clock edge ( $t_2$ ). So it appears that  $Q_A$  will be a debounced and synchronized version of  $K$ . However, a possibility of failure exists if  $K$  changes very close to the clock edge such that the setup or hold time is violated. In this case the flip-flop output  $Q_A$  may oscillate or otherwise malfunction. Although this situation will occur very infrequently, it is best to guard against it by adding a second flip-flop. We will choose the

clock period so that any oscillation at the output of  $Q_A$  will have died out before the next active edge of the clock so that the input  $D_B$  will always be stable at the active clock edge. The debounced signal,  $Kd$ , will always be clean and synchronized with the clock, although it may be delayed up to two clock cycles after the key is pressed.

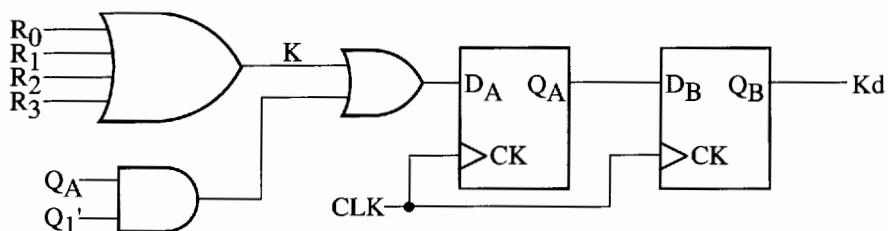
**Figure 3-23 Debouncing and Synchronizing Circuit**



(a)



(b)



(c)

We will divide the keypad scanner into three modules, as shown in Figure 3-24. The debounce module generates a signal  $K$  when a key has been pressed and a signal  $Kd$  after it has been debounced. The keyscan module generates the column signals to scan the keyboard. When a valid key is detected, the decoder determines the key number from the row and column numbers. Figure 3-25 shows the keyscan state graph. Keyscan waits in  $S_1$  with outputs  $C_1 = C_2 = C_3 = 1$  until a key is pressed. In  $S_2$ ,  $C_0 = 1$ , so if the key that was

pressed is in column 0,  $K = 1$ , and the network outputs a valid signal and goes to  $S_5$ . If no key press is found in column 0, column 1 is checked in  $S_3$ , and if necessary, column 2 is checked in  $S_4$ . In  $S_5$ , the network waits until all keys are released and  $Kd$  goes to 0 before resetting.

Figure 3-24 Scanner Modules

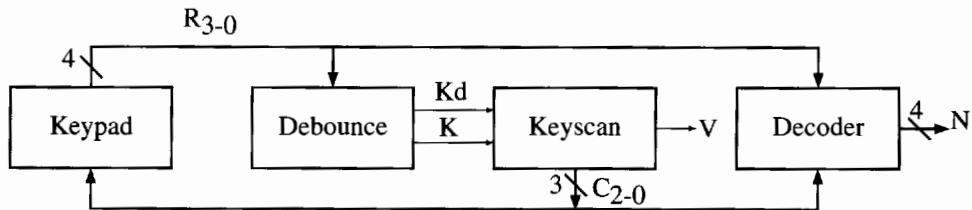
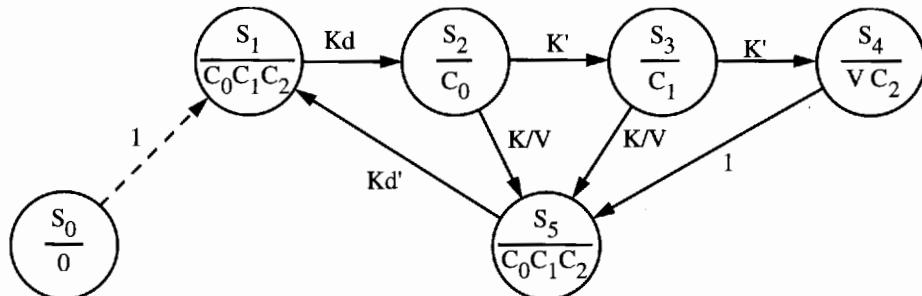


Figure 3-25 State Graph for Scanner



The decoder determines the key number from the row and column numbers using the truth table given in Table 3-7. The truth table has one row for each of the 12 keys. The remaining rows have don't care outputs since we have assumed that only one key is pressed at a time. Since the decoder is a combinational network, its output will change as the keypad is scanned. At the time a valid key is detected ( $K = 1$  and  $V = 1$ ), its output will have the correct value and this value can be saved in a register at the same time the network goes to  $S_5$ .

**Table 3-7 Truth Table for Decoder**

$R_3 R_2 R_1 R_0 C_0 C_1 C_2$	$N_3 N_2 N_1 N_0$
0 0 0 1 1 0 0	0 0 0 1
0 0 0 1 0 1 0	0 0 1 0
0 0 0 1 0 0 1	0 0 1 1
0 0 1 0 1 0 0	0 1 0 0
0 0 1 0 0 1 0	0 1 0 1
0 0 1 0 0 0 1	0 1 1 0
0 1 0 0 1 0 0	0 1 1 1
0 1 0 0 0 1 0	1 0 0 0
0 1 0 0 0 0 1	1 0 0 1
1 0 0 0 1 0 0	1 0 1 0 (*)
1 0 0 0 0 1 0	0 0 0 0
1 0 0 0 0 0 1	1 0 1 1 (#)

Logic equations for decoder:

$$N_3 = R_2 C_0' + R_3 C_1'$$

$$N_2 = R_1 + R_2 C_0$$

$$N_1 = R_0 C_0' + R_2' C_2 + R_1' R_0' C_0$$

$$N_0 = R_1 C_1 + R_1' C_2 + R_3' R_1' C_1'$$

We will try to implement the debounce, keyscan, and decoder modules with a single 22V10 with as little added hardware as possible. The 22V10 would require the following inputs:  $R_0$ ,  $R_1$ ,  $R_2$ ,  $R_3$ , clock, and reset. The outputs would be  $C_0$ ,  $C_1$ ,  $C_2$ ,  $N_3$ ,  $N_2$ ,  $N_1$ ,  $N_0$ , and  $V$ . This uses up 8 of the 10 macrocells. If the state graph was implemented using three flip-flops, 11 macrocells would be required, and it would not fit. One solution would be to use two PALs and put the decoder in a separate PAL. A better solution is to use four flip-flops to implement the state graph and encode the states so that the outputs  $C_0$ ,  $C_1$ , and  $C_2$  can be read directly from the flip-flops  $Q_2$ ,  $Q_3$ ,  $Q_4$ . The following state assignment can be used for  $Q_1 Q_2 Q_3 Q_4$ :  $S_1$ , 0111;  $S_2$ , 0100;  $S_3$ , 0010;  $S_4$ , 0001;  $S_5$ , 1111. The first three flip-flops produce the  $C$  outputs, and flip-flop  $Q_1$  distinguishes between states  $S_1$  and  $S_5$ . With this state encoding, a total of 9 macrocells are required to implement the keyscan and decoder modules. This leaves one flip-flop available for debouncing, so that only one external flip-flop is required for  $Kd$ . If the 22V10 is reset, the flip-flop states will be 0000, so we have added  $S_0$  to the state graph with a next state of  $S_1$ . The equations derived from the state graph using a CAD program (such as *LogicAid*) are as follows:

$$Q_1^+ = Q_1 Kd + Q_2 Q_3' K + Q_2' Q_3 K + Q_2' Q_4$$

$$Q_2^+ = Q_2' Q_3' + K + Q_4$$

$$Q_3^+ = Q_3' + Q_1 + Q_4 Kd' + Q_2' K$$

$$Q_4^+ = Q_2' + Q_1 + Q_3 Kd' + Q_3' K$$

$$V = KQ_2 Q_3' + KQ_2' Q_3 + Q_2' Q_4$$

To avoid generating  $K$ , which would use up a macrocell, we can substitute  $R_0 + R_1 + R_2 + R_3$  for  $K$  in the preceding equations. The resulting equation with the most terms is

$$Q_1^+ = Q_1 Kd + Q_2 Q_3 (R_0 + R_1 + R_2 + R_3) + Q_2' Q_3 (R_0 + R_1 + R_2 + R_3) + Q_2' Q_4$$

The maximum number of terms in any equation is 10, and all these equations, as well as the decoder equations, will easily fit in the 22V10. The final equations can be entered into a CAD program to generate the bit patterns for programming the PAL.

We tested the scanner design by using VHDL and discovered one flaw related to the debouncing circuit (Figure 3-23(b)). The original design works fine if the key pressed is in columns 0 or 1, but a problem occurs if the key pressed is in column 2. In this case,  $K$  goes to 0 when scanning columns 0 and 1, so  $Kd$  goes to 0 when  $S_5$  is reached, even if the key is still being pressed. To remedy this problem, we change the next state equation for  $Q_A$  to

$$Q_A^+ = K + Q_A Q_1'$$

The added term assures that once  $Q_A$  is set to 1, it will remain 1 until  $S_5$  is reached and  $Q_1$  becomes 1. The revised debounce circuit is shown in Figure 3-23(c).

The VHDL code used to test the design is shown in Figure 3-26. The decoder equations as well as the equations for  $K$  and  $V$  are implemented by concurrent statements. The process implements the next state equations for the keyscan and debounce flip-flops. This VHDL code would be very difficult to test by supplying waveforms for the inputs  $R_0$ ,  $R_1$ ,  $R_2$ , and  $R_3$ , since these inputs depend on the column outputs ( $C_0$ ,  $C_1$ ,  $C_2$ ). A much better way to test the scanner is write a test program, called scantest, in VHDL. Such a test program is often referred to as a test bench, by analogy with a hardware test bench. The scanner we are testing will be treated as a component and embedded in the test program. The signals generated within scantest are interfaced to the scanner as shown in Figure 3-27. Scantest simulates a key press by supplying the appropriate  $R$  signals in response to the  $C$  signals from the scanner. When scantest receives  $V = 1$  from the scanner, it checks to see if the value of  $N$  corresponds to the key that was pressed.

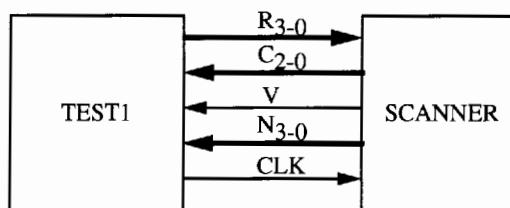
**Figure 3-26 VHDL Code for Scanner**

```

entity scanner is
  port (R0,R1,R2,R3,CLK: in bit;
        C0,C1,C2: inout bit;
        N0,N1,N2,N3,V: out bit);
end scanner;

architecture scan1 of scanner is
signal Q1,QA, K, Kd: bit;
alias Q2: bit is C0;                                -- column outputs will be the same
alias Q3: bit is C1;                                -- as the state variables because
alias Q4: bit is C2;                                -- of state assignment
begin
  K <= R0 or R1 or R2 or R3;                      -- this is the decoder section
  N3 <= (R2 and not C0) or (R3 and not C1);
  N2 <= R1 or (R2 and C0);
  N1 <= (R0 and not C0) or (not R2 and C2) or (not R1 and not R0 and C0);
  N0 <= (R1 and C1) or (not R1 and C2) or (not R3 and not R1 and not C1);
  V <= (Q2 and not Q3 and K) or (not Q2 and Q3 and K) or (not Q2 and Q4);
  process(CLK)                                     -- process to update flip-flops
begin
  if CLK = '1' then
    Q1 <= (Q1 and Kd) or (Q2 and not Q3 and K) or (not Q2 and q3 and K)
      or (not Q2 and Q4);
    Q2 <= (not Q2 and not Q3) or K or Q4;
    Q3 <= not Q3 or Q1 or (Q4 and not Kd) or (not Q2 and K);
    Q4 <= not Q2 or Q1 or (Q3 and not Kd) or (not Q3 and K);
    QA <= K or (QA and not Q1);                  -- first debounce flip-flop
    Kd <= QA;                                    -- second debounce flip-flop
  end if;
  end process;
end scan1;

```

**Figure 3-27 Interface for Scantest**

The VHDL code for scantest is shown in Figure 3-28. A copy of the scanner is instantiated within the Test1 architecture, and connections to the scanner are made by the port map. The sequence of key numbers used for testing is stored in the array *KARRAY*. The tester simulates the keypad operation using concurrent statements for  $R_0$ ,  $R_1$ ,  $R_2$ , and  $R_3$ . Whenever  $C_0$ ,  $C_1$ ,  $C_2$ , or the key number (*KN*) changes, new values for the *R*s are computed. For example if  $KN = 5$  (to simulate pressing key 5), then  $R_0R_1R_2R_3 = 0100$  is sent to the scanner when  $C_0C_1C_2 = 010$ . The test process is as follows:

1. Read a key number from the array to simulate pressing a key.
2. Wait until  $V = 1$  and the rising edge of the clock occurs.
3. Verify that the  $N$  output from the scanner matches the key number.
4. Set  $KN = 15$  to simulate no key pressed. (Since 15 is not a valid key number, all *R*s will go to 0.)
5. Wait until  $Kd = 0$  before selecting a new key.

Scantest will report an error if the scanner generates the wrong key number, and it will report "Testing complete" when all keys have been tested.

## Assert Statement

The scantest architecture uses an assert statement to check if the scanner output matches the key number. The assert statement checks to see if a certain condition is true, and if not causes an error message to be displayed. One form of the assert statement is

```
assert boolean-expression
    report string-expression
    severity severity-level;
```

If the boolean-expression is false, then the string-expression is displayed on the monitor along with the severity-level. If the boolean-expression is true, no message is displayed. The four possible severity-levels are note, warning, error, and failure. The action taken for these severity-levels depends on the simulator.

If the assert clause is omitted, then the report is always made. The severity-level is optional. Thus the statement

```
report "ALL IS WELL";
```

will display the message "ALL IS WELL" whenever the statement is executed.

**Figure 3-28 VHDL for Scantest**

```
library BITLIB;
use BITLIB.bit_pack.all;

entity scantest is
end scantest;
```

```

architecture test1 of scantest is
component scanner
  port (R0,R1,R2,R3,CLK: in bit;
        C0,C1,C2: inout bit;
        N0,N1,N2,N3,V: out bit);
end component;

type arr is array(0 to 11) of integer; -- array of keys to test
constant KARRAY:arr := (2,5,8,0,3,6,9,11,1,4,7,10);
signal C0,C1,C2,V,CLK,R0,R1,R2,R3: bit;-- interface signals
signal N: bit_vector(3 downto 0);
signal KN: integer;                      -- key number to test
begin
  CLK <= not CLK after 20 ns;           -- generate clock signal
                                         -- this section emulates the keypad
  R0 <= '1' when (C0='1' and KN=1) or (C1='1' and KN=2)
    or (C2='1' and KN=3)
  else '0';
  R1 <= '1' when (C0='1' and KN=4) or (C1='1' and KN=5)
    or (C2='1' and KN=6)
  else '0';
  R2 <= '1' when (C0='1' and KN=7) or (C1='1' and KN=8)
    or (C2='1' and KN=9)
  else '0';
  R3 <= '1' when (C0='1' and KN=10) or (C1='1' and KN=0)
    or (C2='1' and KN=11)
  else '0';

  process                                -- this section tests scanner
begin
  for i in 0 to 11 loop                  -- test every number in key array
    KN <= KARRAY(i);                   -- simulates keypress
    wait until (V='1' and rising_edge(CLK));
    assert (vec2int(N) = KN)           -- check if output matches
      report "Numbers don't match"
      severity error;
    KN <= 15;                         -- equivalent to no key pressed
    wait until rising_edge(CLK);       -- wait for scanner to reset
    wait until rising_edge(CLK);
    wait until rising_edge(CLK);
  end loop;
  report "Test complete.";
end process;
scanner1: scanner                      -- connect test1 to scanner
  port map(R0,R1,R2,R3,CLK,C0,C1,C2,N(0),N(1),N(2),N(3),V);
end test1;

```

In this chapter we have introduced several different types of PLDs and used them for designing sequential networks. Many other types of PLDs are available, together with software packages that facilitate their use. Some of these programs accept input only in the form of logic equations, whereas others have options for state table, state graph, or logic diagram input. These programs generally produce a data file used as input to a PLD programmer to program the PLD for a specific application. Programmable gate arrays (PGAs) and other complex PLDs are described in Chapter 6.

## Problems

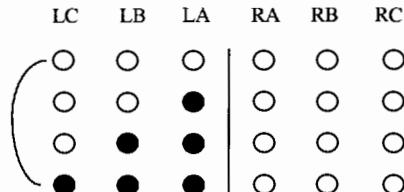
- 3.1** The following state table is implemented using a ROM and two D flip-flops (falling-edge triggered):

$Q_1 Q_2$	$Q_1^+ Q_2^+$		Z	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
00	01	10	0	1
01	10	00	1	1
10	00	01	1	0

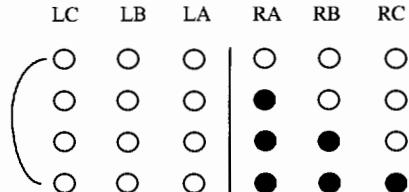
- (a) Draw the block diagram.
- (b) Write VHDL code that describes the system. Assume that the ROM has a delay of 10 ns, and each flip-flop has a propagation delay of 15 ns.

- 3.2** An older-model Thunderbird car has three left and three right tail lights, which flash in unique patterns to indicate left and right turns.

Left-turn pattern:



Right-turn pattern:



Design a Moore sequential network to control these lights. The network has three inputs, LEFT, RIGHT, and HAZ. LEFT and RIGHT come from driver's turn-signal switch and cannot be 1 at the same time. As indicated above, when LEFT = 1 the lights flash in a pattern LA on, LA and LB on, LA, LB and LC on, and all off; then the sequence repeats. When RIGHT = 1, the light sequence is similar. If a switch from LEFT to RIGHT (or vice versa) occurs in the middle of a flashing sequence, the network should immediately go to the IDLE (lights off) state and then start the new sequence. HAZ comes from the hazard switch, and when HAZ = 1, all six lights flash on and off in unison. HAZ takes precedence if LEFT or RIGHT is also on. Assume that a clock signal is available with a frequency equal to the desired flashing rate.

- (a) Draw the state graph (8 states).
- (b) Realize the network using six D flip-flops, and make a state assignment such that each flip-flop output drives one of the six lights directly.
- (c) Realize the network using three D flip-flops, using the guidelines to determine a suitable state assignment.
- (d) Note the trade-off between more flip-flops and more gates in (b) and (c). Suggest a suitable PAL or PLD for each case.
- (e) Write VHDL code for and simulate your solution to (b).

**3.3** Find a minimum-row PLA table to implement the following sets of functions.

(a)  $f_1(A, B, C, D) = \Sigma m(4, 5, 10, 11, 12),$

$$f_2(A, B, C, D) = \Sigma m(0, 1, 3, 4, 8, 11),$$

$$f_3(A, B, C, D) = \Sigma m(0, 4, 10, 12, 14)$$

(b)  $f_1(A, B, C, D) = \Sigma m(3, 4, 6, 9, 11),$

$$f_2(A, B, C, D) = \Sigma m(2, 4, 8, 10, 11, 12),$$

$$f_3(A, B, C, D) = \Sigma m(3, 6, 7, 10, 11)$$

**3.4** An  $N$ -bit bidirectional shift register has  $N$  parallel data inputs,  $N$  outputs, a left serial input (*LSI*), a right serial input (*RSI*), a clock input, and the following control signals:

*Load*: Load the parallel data into the register (load overrides shift).

*Rsh*: Shift the register right (*LSI* goes into the left end).

*Lsh*: Shift the register left (*RSI* goes into the right end).

(a) If the register is implemented using a 22V10, what is the maximum value of  $N$ ?

(b) Give equations for the rightmost two cells.

**3.5** An  $N$ -bit binary up-down counter is to be realized using a 22V10. The counter has control inputs  $U$  and  $D$  and a clock input. ( $U = 1$ , count up;  $D = 1$ , count down;  $U = D = 0$ , no count;  $U = D = 1$  is not allowed.) What is the maximum value of  $N$ ? Show how you arrived at your answer.

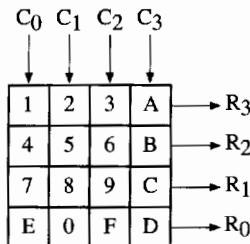
**3.6** Design an 6-bit up-down binary counter using a 22V10 and a minimum number of external gates. Give all of the flip-flop input equations. Write the VHDL code for the counter using a PLA. Simulate the code and verify that the counter works.

**3.7** A Mealy sequential network with four output variables is realized using a 22V10. What is the maximum number of input variables it can have? The maximum number of states? Can any Mealy network with these numbers of inputs and outputs be realized with a 22V10? Explain.

**3.8** A keypad has four rows and three columns as in Figure 3-22. Assume no more than two keys will be pressed at a time. Write the first 10 rows of the truth table for a keypad decoder similar to Table 3-7. If two keys are pressed in the same column, the  $N$  output should indicate the key in the first of the two rows.

**3.9** Write the VHDL behavioral model for the keypad scanner functionally equivalent to Figure 3-26 that includes the state machine in Figure 3-25 and the decoder in Table 3-7.

**3.10** Design a  $4 \times 4$  keypad scanner for the following keypad layout.



- (a) Assuming only one key can be pressed at a time, find the equations for a number decoder given  $R_{3,0}$  and  $C_{3,0}$ , whose output corresponds to the binary value of the key. For example, the F key will return  $N_{3,0} = 1111$  in binary, or 15.
- (b) Design a debouncing circuit that detects when a key has been pressed or depressed. Assume switch bounce will die out in one or two clock cycles. When a key has been pressed,  $K = 1$  and  $Kd$  is the debounced signal.
- (c) Design and draw an SM chart that performs the keyscan and issues a valid pulse when a valid key has been pressed using inputs from part (b).
- (d) Write a VHDL description of your keypad scanner and include the decoder, debouncing circuit, and the scanner.

### 3.11

- (a) Implement the traffic-light controller of Figure 3-18 using a 74163 counter with added logic. Use a ROM to generate the outputs.
- (b) Write a VHDL structural description of your answer to (a).
- (c) Write a testbench for part (b) and verify that your controller works correctly.

# CHAPTER 4

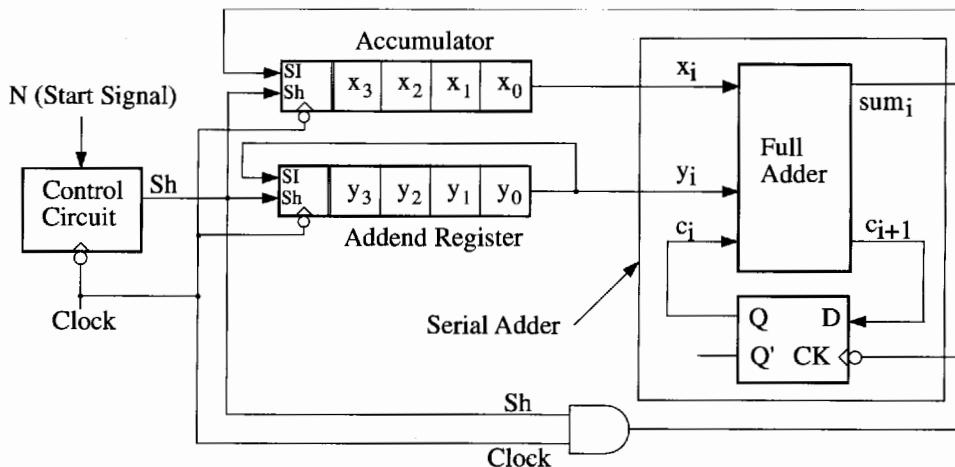
## DESIGN OF NETWORKS FOR ARITHMETIC OPERATIONS

This chapter uses binary multipliers and dividers as examples to illustrate the design of small digital systems. We introduce the concept of using a control circuit to control the sequence of operations in a digital system. We use VHDL to describe a digital system at the behavioral level so we can simulate the system to check out the algorithms used and to make sure that the sequence of operations is correct. We can then define the required control signals and the actions performed by these signals. Next, we write a VHDL description of the system in terms of the control signals and verify its correct operation by simulation. After completing the detailed design in terms of components and logic equations, we can again use VHDL to check out our design.

### 4.1 DESIGN OF A SERIAL ADDER WITH ACCUMULATOR

A control circuit for a digital system is a sequential network that outputs a sequence of control signals. These signals cause operations such as addition and shifting to take place at appropriate times. In this section, we illustrate the design of a control circuit for a serial adder with accumulator. Figure 4-1 shows the block diagram for the adder. Two shift registers are used to hold the 4-bit numbers to be added,  $X$  and  $Y$ . The box at the left end of each shift register shows the inputs:  $Sh$  (shift),  $SI$  (serial input), and  $Clock$ . When  $Sh = 1$  and the clock is pulsed,  $SI$  is entered into  $x_3$  (or  $y_3$ ) as the contents of the register are shifted right one position. The  $X$ -register serves as the accumulator, and after four shifts, the number  $X$  is replaced with the sum of  $X$  and  $Y$ . The addend register is connected as a cyclic shift register, so after four shifts it is back to its original state and the number  $Y$  is not lost. The serial adder consists of a full adder and a carry flip-flop. At each clock time, one pair of bits is added. When  $Sh = 1$ , the falling edge of the clock shifts the sum bit into the

Figure 4-1 Serial Adder with Accumulator



accumulator, stores the carry bit in the carry flip-flop, and causes the addend register to rotate right. Additional connections needed for initially loading the  $X$  and  $Y$  registers and clearing the carry flip-flop are not shown in the block diagram.

Table 4-1 illustrates the operation of the serial adder. In this table,  $t_0$  is the time before the first clock,  $t_1$  is the time after the first clock,  $t_2$  is the time after the second clock, etc. Initially, at time  $t_0$  the accumulator contains  $X$ , the addend register contains  $Y$ , and the carry flip-flop is clear. Since the full adder is a combinational network,  $x_0 = 1$ ,  $y_0 = 1$ , and  $c_0 = 0$  are added after a short propagation delay to give 10, so  $sum_0 = 0$  and carry  $c_1 = 1$ . When the first clock occurs,  $sum_0$  is shifted into the accumulator, and the remaining accumulator digits are shifted right one position. The same shift pulse stores  $c_1$  in the carry flip-flop and cycles the addend register right one position. The next pair of bits,  $x_1 = 0$  and  $y_1 = 1$ , are now at the full adder input, and the adder generates the sum and carry,  $sum_1 = 0$  and  $c_2 = 1$ . The second clock pulse shifts  $sum_1$  into the accumulator, stores  $c_2$  in the carry flip-flop and cycles the addend register right. Bits  $x_2$  and  $y_2$  are now at the adder input, and the process continues until all bit pairs have been added. After four clocks (time  $t_4$ ), the sum of  $X$  and  $Y$  is in the accumulator, and the addend register is back to its original state.

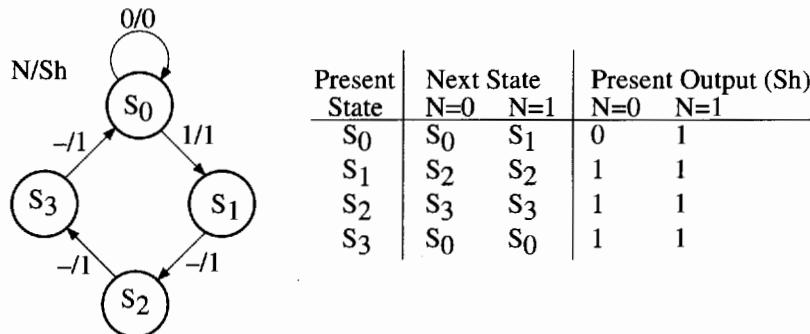
Table 4-1 Operation of Serial Adder

	$X$	$Y$	$c_i$	$sum_i$	$c_{i+1}$
$t_0$	0101	0111	0	0	1
$t_1$	0010	1011	1	0	1
$t_2$	0001	1101	1	1	1
$t_3$	1000	1110	1	1	0
$t_4$	1100	0111	0	(1)	(0)

The control circuit for the adder must now be designed so that after receiving a start signal, the control circuit will output  $Sh = 1$  for four clocks and then stop. Figure 4-2

shows the state graph and table for the control circuit. The network remains in  $S_0$  until a start signal ( $N$ ) is received, at which time the network outputs  $Sh = 1$  and goes to  $S_1$ . Then  $Sh = 1$  for three more clock times, and the network returns to  $S_0$ . It will be assumed that the start signal is terminated before the network returns to state  $S_0$ , so no further action occurs until another start signal is received. Dashes (don't cares) on the graph indicate that once  $S_1$  is reached, the network operation continues regardless of the value of  $N$ .

**Figure 4-2 Control State Graph and Table for Serial Adder**



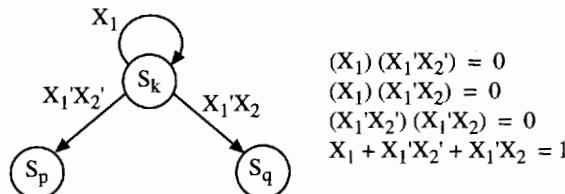
## 4.2 STATE GRAPHS FOR CONTROL NETWORKS

Before continuing with the next design example, we describe the notation we use on control state graphs, and then state the conditions that must be satisfied to have a proper state graph. We usually label control state graphs using variable names instead of 0s and 1s. This makes the graph easier to read, especially when the number of inputs and outputs is large. If we label an arc on a Mealy state graph  $X_i X_j / Z_p Z_q$ , this means if inputs  $X_i$  and  $X_j$  are 1 (we don't care what the other input values are), the outputs  $Z_p$  and  $Z_q$  are 1 (and the other outputs are 0), and we will traverse this arc to go to the next state. For example, for a network with four inputs ( $X_1, X_2, X_3, X_4$ ) and four outputs ( $Z_1, Z_2, Z_3, Z_4$ ), the label  $X_1 X_4' / Z_2 Z_3$  is equivalent to  $1 - 0/0110$ . In general, if we label an arc with an input expression,  $I$ , we will traverse the arc when  $I = 1$ . For example, if the input label is  $AB + C'$ , we will traverse the arc when  $AB + C' = 1$ .

In order to have a completely specified proper state graph in which the next state is always uniquely defined for every input combination, we must place the following constraints on the input labels for every state  $S_k$ :

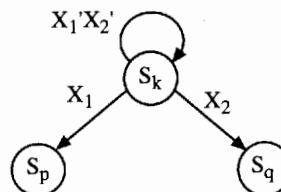
1. If  $I_i$  and  $I_j$  are any pair of input labels on arcs exiting state  $S_k$ , then  $I_i I_j = 0$  if  $i \neq j$ .
2. If  $n$  arcs exit state  $S_k$  and the  $n$  arcs have input labels  $I_1, I_2, \dots, I_n$ , respectively, then  $I_1 + I_2 + \dots + I_n = 1$ .

Condition 1 assures us that at most one input label can be 1 at any given time, and condition 2 assures us that at least one input label will be 1 at any given time. Therefore, exactly one label will be 1, and the next state will be uniquely defined for every input combination. For example, consider the following partial state graph where  $I_1 = X_1$ ,  $I_2 = X_1'X_2'$ , and  $I_3 = X_1'X_2$ :



Conditions 1 and 2 are satisfied for  $S_k$ .

An incompletely specified proper state graph must always satisfy condition 2, and it must satisfy condition 1 for all combinations of values of input variables that can occur for each state  $S_k$ . Thus, the following represents part of a proper state graph only if input combination  $X_1 = X_2 = 1$  cannot occur in state  $S_k$ :

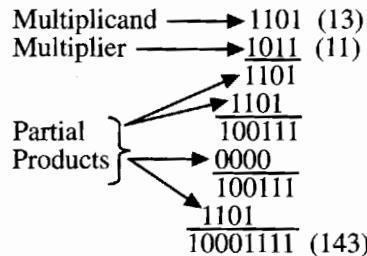


If there are three input variables ( $X_1, X_2, X_3$ ), the preceding partial state graph represents the following state table row:

	000	001	010	011	100	101	110	111
$S_k$	$S_k$	$S_k$	$S_q$	$S_q$	$S_p$	$S_p$	—	—

### 4.3 DESIGN OF A BINARY MULTIPLIER

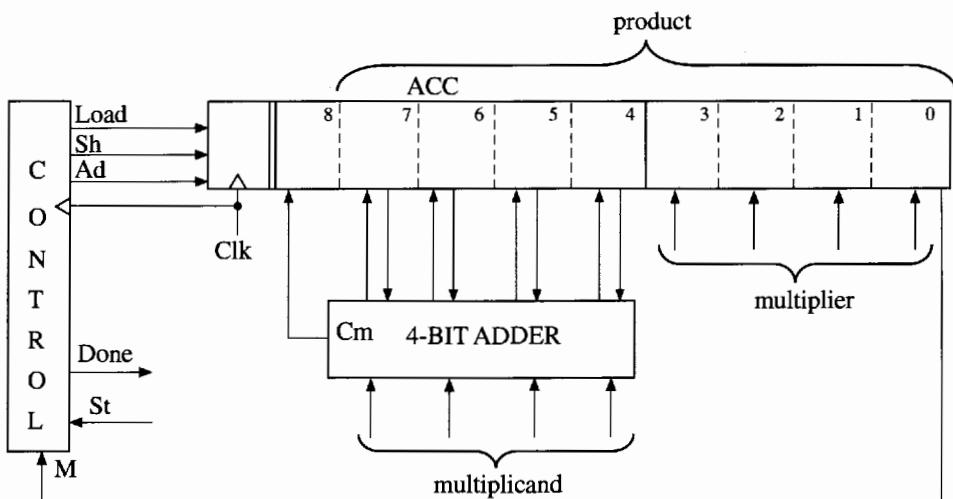
In this section, we will design a multiplier for unsigned binary numbers. When we form the product  $A \times B$ , the first operand ( $A$ ) is called the *multiplicand*, and the second operand ( $B$ ) is called the *multiplier*. As illustrated here, binary multiplication requires only shifting and adding. In the following example, we multiply  $13_{10}$  by  $11_{10}$  in binary:



Note that each partial product is either the multiplicand (1101) shifted over by the appropriate number of places or zero. Instead of forming all the partial products first and then adding, each new partial product is added in as soon as it is formed, which eliminates the need for adding more than two binary numbers at a time.

Multiplication of two 4-bit numbers requires a 4-bit multiplicand register, a 4-bit multiplier register, a 4-bit full adder, and an 8-bit register for the product. The product register serves as an accumulator to accumulate the sum of the partial products. If the multiplicand were shifted left each time before it was added to the accumulator, as was done in the previous example, an 8-bit adder would be needed. So it is better to shift the contents of the product register to the right each time, as shown in the block diagram of Figure 4-3. This type of multiplier is sometimes referred to as a serial-parallel multiplier, since the multiplier bits are processed serially, but the addition takes place in parallel. As indicated by the arrows on the diagram, 4 bits from the accumulator (ACC) and 4 bits from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator. When an add signal (*Ad*) occurs, the adder outputs are transferred to the accumulator by the next clock pulse, thus causing the multiplicand to be added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry that is generated when the multiplicand is added to the accumulator. When a shift signal (*Sh*) occurs, all 9 bits of ACC are shifted right by the next clock pulse.

**Figure 4-3 Block Diagram for Binary Multiplier**



Since the lower 4 bits of the product register are initially unused, we will store the multiplier in this location instead of in a separate register. As each multiplier bit is used, it is shifted out the right end of the register to make room for additional product bits. A shift signal (*Sh*) causes the contents of the product register (including the multiplier) to be shifted right one place when the next clock pulse occurs. The control circuit puts out the proper sequence of add and shift signals after a start signal (*St* = 1) has been received. If

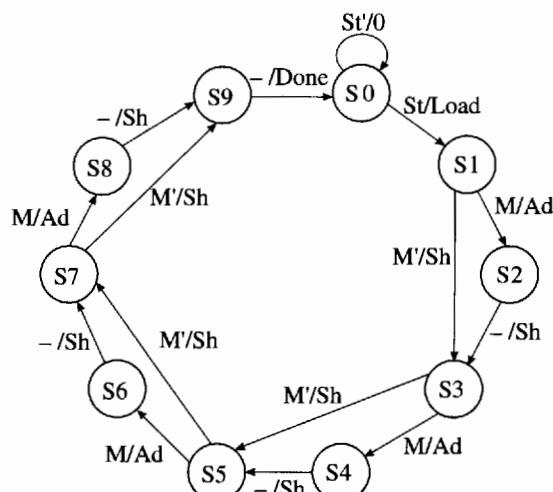
the current multiplier bit ( $M$ ) is 1, the multiplicand is added to the accumulator followed by a right shift; if the multiplier bit is 0, the addition is skipped, and only the right shift occurs. The multiplication example ( $13 \times 11$ ) is reworked below showing the location of the bits in the registers at each clock time.

initial contents of product register	$0\ 0\ 0\ 0\ 0 1\ 0\ 1\ 1$	$\leftarrow M(11)$
(add multiplicand since $M=1$ )	$\underline{1\ 1\ 0\ 1}$	(13)
after addition	$0\ 1\ 1\ 0\ 1 1\ 0\ 1\ 1$	
after shift	$0\ 0\ 1\ 1\ 0\ 1 1\ 0\ 1$	$\leftarrow M$
(add multiplicand since $M=1$ )	$\underline{1\ 1\ 0\ 1}$	
after addition	$1\ 0\ 0\ 1\ 1\ 1 1\ 0\ 1$	
after shift	$0\ 1\ 0\ 0\ 1\ 1 1\ 0$	$\leftarrow M$
(skip addition since $M=0$ )	$0\ 0\ 1\ 0\ 0\ 1\ 1 1$	
after shift	$0\ 0\ 1\ 0\ 0\ 1\ 1 1$	$\leftarrow M$
(add multiplicand since $M=1$ )	$\underline{1\ 1\ 0\ 1}$	
after addition	$1\ 0\ 0\ 0\ 1\ 1\ 1 1$	
after shift (final answer)	$0\ 1\ 0\ 0\ 0\ 1\ 1\ 1 1$	(143)

dividing line between product and multiplier

The control circuit must be designed to output the proper sequence of add and shift signals. Figure 4-4 shows a state graph for the control circuit. In Figure 4-4,  $S_0$  is the reset state, and the network stays in  $S_0$  until a start signal ( $St = 1$ ) is received. This generates a *Load* signal, which causes the multiplier to be loaded into the lower 4 bits of the accumulator (ACC) and the upper 5 bits of the accumulator to be cleared. In state  $S_1$ , the low-order bit of the multiplier ( $M$ ) is tested. If  $M = 1$ , an add signal is generated, and if  $M = 0$ , a shift

Figure 4-4 State Graph for Binary Multiplier Control



signal is generated. Similarly, in states  $S_3$ ,  $S_5$ , and  $S_7$ , the current multiplier bit ( $M$ ) is tested to determine whether to generate an add or shift signal. A shift signal is always generated at the next clock time following an add signal (states  $S_2$ ,  $S_4$ ,  $S_6$ , and  $S_8$ ). After four shifts have been generated, the control network goes to  $S_9$ , and a done signal is generated before returning to  $S_0$ .

The behavioral VHDL model (Figure 4-5) corresponds directly to the state graph. Since there are 10 states, we have declared an integer range 0 to 9 for the state signal. The signal  $ACC$  represents the 9-bit accumulator output. The statement

```
alias M: bit is ACC(0);
```

allows us to use the name  $M$  in place of  $ACC(0)$ . The notation **when** 1|3|5|7 => means when the state is 1 or 3 or 5 or 7, the action that follows occurs. All register operations and state changes take place on the rising edge of the clock. For example, in state 0, if  $St$  is 1, the multiplier is loaded into the accumulator at the same time the state changes to 1. The *add4* function computes the sum of two 4-bit vectors and a carry to give a 5-bit result. This represents the adder output, which is loaded into  $ACC$  at the same time the state counter is incremented. The right shift on  $ACC$  is accomplished by loading  $ACC$  with 0 concatenated with the upper 8 bits of  $ACC$ . The expression ' $0'&ACC(8 downto 1)$ ' could be replaced with  $ACC\ srl\ 1$ .

The done signal needs to be turned on only in state 9. If we had used the statement **when** 9 =>  $State \leq 0$ ;  $Done \leq '1'$ ,  $Done$  would be turned on at the same time the  $State$  changed to 0. This is too late, since we want  $Done$  to turn on when the  $State$  becomes 9. Therefore, we used a separate concurrent assignment statement. This statement is placed outside the process so that  $Done$  will be updated whenever  $State$  changes.

**Figure 4-5 Behavioral Model for  $4 \times 4$  Binary Multiplier**

```
-- This is a behavioral model of a multiplier for unsigned
-- binary numbers. It multiplies a 4-bit multiplicand
-- by a 4-bit multiplier to give an 8-bit product.

-- The maximum number of clock cycles needed for a
-- multiply is 10.

library BITLIB;
use BITLIB.bit_pack.all;

entity mult4X4 is
    port (Clk, St: in bit;
          Mplier, Mcand : in bit_vector(3 downto 0);
          Done: out bit);
end mult4X4;

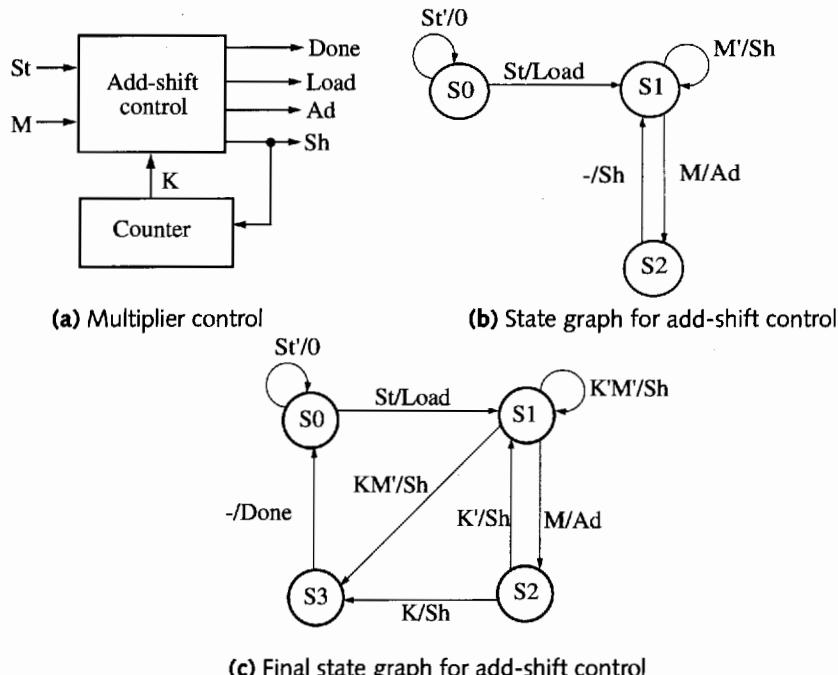
architecture behav of mult4X4 is
    signal State: integer range 0 to 9;
    signal ACC: bit_vector(8 downto 0);           --accumulator
    alias M: bit is ACC(0);                      --M is bit 0 of ACC
```

```

begin
process
begin
  wait until Clk = '1';           --executes on rising edge of
                                   clock
  case State is
    when 0=>                   --initial State
      if St='1' then
        ACC(8 downto 4) <= "00000"; --Begin cycle
        ACC(3 downto 0) <= Mplier;  --load the multiplier
        State <= 1;
      end if;
    when 1 | 3 | 5 | 7 =>         --"add/shift" State
      if M = '1' then             --Add multiplicand
        ACC(8 downto 4) <= add4(ACC(7 downto 4), Mcand, '0');
        State <= State+1;
      else
        ACC <= '0' & ACC(8 downto 1);--Shift accumulator right
        State <= State + 2;
      end if;
    when 2 | 4 | 6 | 8 =>         --"shift" State
      ACC <= '0' & ACC(8 downto 1); --Right shift
      State <= State + 1;
    when 9 =>                   --End of cycle
      State <= 0;
  end case;
end process;
Done <= '1' when State = 9 else '0';
end behav1;

```

As the state graph for the multiplier indicates, the control performs two functions—generating add or shift signals as needed and counting the number of shifts. If the number of bits is large, it is convenient to divide the control network into a counter and an add-shift control, as shown in Figure 4-6(a). First, we will derive a state graph for the add-shift control that tests  $S_t$  and  $M$  and outputs the proper sequence of add and shift signals (Figure 4-6(b)). Then we will add a completion signal ( $K$ ) from the counter that stops the multiplier after the proper number of shifts have been completed. Starting in  $S_0$  in Figure 4-6(b), when a start signal  $S_t = 1$  is received, a load signal is generated and the network goes to state  $S_1$ . Then if  $M = 1$ , an add signal is generated and the network goes to state  $S_2$ ; if  $M = 0$ , a shift signal is generated and the network stays in  $S_1$ . In  $S_2$ , a shift signal is generated since a shift always follows an add. The graph of Figure 4-6(b) will generate the proper sequence of add and shift signals, but it has no provision for stopping the multiplier.

**Figure 4-6 Multiplier Control with Counter**

In order to determine when the multiplication is completed, the counter is incremented each time a shift signal is generated. If the multiplier is  $n$  bits,  $n$  shifts are required. We will design the counter so that a completion signal ( $K$ ) is generated after  $n - 1$  shifts have occurred. When  $K = 1$ , the network should perform one more addition if necessary and then do the final shift. The control operation in Figure 4-6(c) is the same as Figure 4-6(b) as long as  $K = 0$ . In state  $S_1$ , if  $K = 1$ , we test  $M$  as usual. If  $M = 0$ , we output the final shift signal and go to the done state ( $S_3$ ); however, if  $M = 1$ , we add before shifting and go to state  $S_2$ . In state  $S_2$ , if  $K = 1$ , we output one more shift signal and then go to  $S_3$ . The last shift signal will increment the counter to 0 at the same time the add-shift control goes to the done state.

As an example, consider the multiplier of Figure 4-3, but replace the control network with Figure 4-6(a). Since  $n = 4$ , a 2-bit counter is needed to count the 4 shifts, and  $K = 1$  when the counter is in state 3 ( $11_2$ ). Table 4-2 shows the operation of the multiplier when 1101 is multiplied by 1011.  $S_0$ ,  $S_1$ ,  $S_2$ , and  $S_3$  represent states of the control circuit (Figure 4-6(c)). The contents of the product register at each step is the same as given on page 126.

Table 4-2 Operation of Multiplier Using a Counter

Time	State	Counter	Product Register	<i>St</i>	<i>M</i>	<i>K</i>	<i>Load</i>	<i>Ad</i>	<i>Sh</i>	<i>Done</i>
$t_0$	S0	00	000000000	0	0	0	0	0	0	0
$t_1$	S0	00	000000000	1	0	0	1	0	0	0
$t_2$	S1	00	000001011	0	1	0	0	1	0	0
$t_3$	S2	00	011011011	0	1	0	0	0	1	0
$t_4$	S1	01	001101101	0	1	0	0	1	0	0
$t_5$	S2	01	100111101	0	1	0	0	0	1	0
$t_6$	S1	10	010011110	0	0	0	0	0	1	0
$t_7$	S1	11	001001111	0	1	1	0	1	0	0
$t_8$	S2	11	100011111	0	1	1	0	0	1	0
$t_9$	S3	00	010001111	0	1	0	0	0	0	1

At time  $t_0$  the control is reset and waiting for a start signal. At time  $t_1$ , the start signal  $St = 1$ , and a *Load* signal is generated. At time  $t_2$ ,  $M = 1$ , so an *Ad* signal is generated. When the next clock occurs, the output of the adder is loaded into the accumulator and the control goes to S2. At  $t_3$ , an *Sh* signal is generated, so at the next clock shifting occurs and the counter is incremented. At  $t_4$ ,  $M = 1$  so *Ad* = 1, and the adder output is loaded into the accumulator at the next clock. At  $t_5$  and  $t_6$ , shifting and counting occur. At  $t_7$ , three shifts have occurred and the counter state is 11, so  $K = 1$ . Since  $M = 1$ , addition occurs and control goes to S2. At  $t_8$ , *Sh* =  $K = 1$ , so at the next clock the final shift occurs and the counter is incremented back to state 00. At  $t_9$ , a *Done* signal is generated.

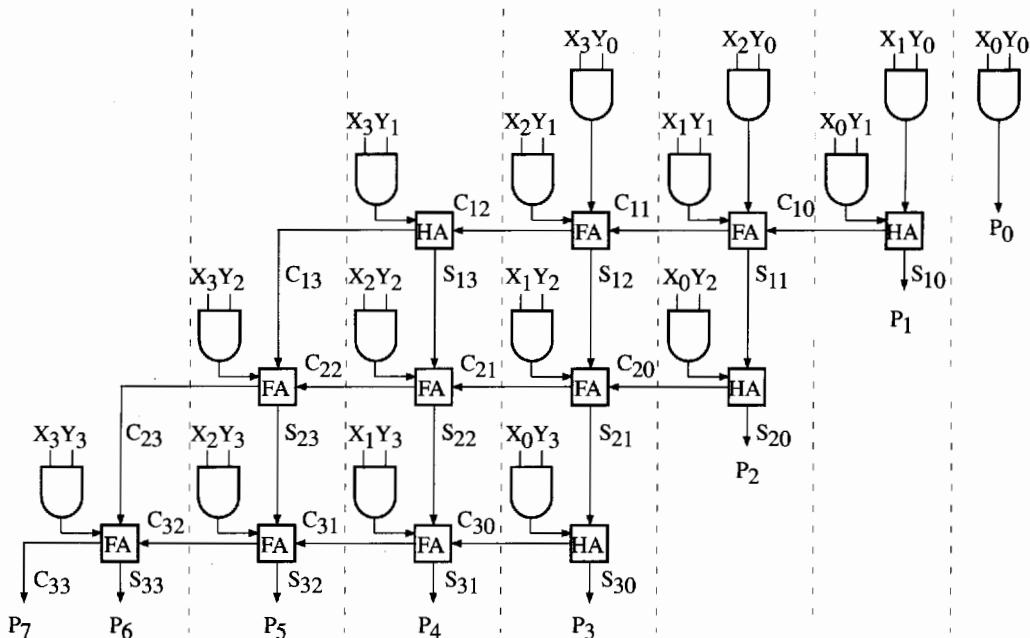
The multiplier design given here can easily be expanded to 8, 16, or more bits simply by increasing the register size and the number of bits in the counter. The add-shift control would remain unchanged.

Next, we design a multiplier that consists of an array of AND gates and adders. This multiplier will have an iterative structure with no sequential logic or registers required. Table 4-3 illustrates multiplication of two 4-bit unsigned numbers,  $X_3X_2X_1X_0$  times  $Y_3Y_2Y_1Y_0$ . Each of the  $X_iY_j$  product bits can be generated by an AND gate. Each partial product can be added to the previous sum of partial products using a row of adders. The sum output of the first row of adders, which adds the first two partial products, is  $S_{13}S_{12}S_{11}S_{10}$ , and the carry output is  $C_{13}C_{12}C_{11}C_{10}$ . Similar results occur for the other two rows of adders. (We have used the notation  $S_{ij}$  and  $C_{ij}$  to represent the sums and carries from the  $i$ th row of adders.) Figure 4-7 shows the corresponding array of AND gates and adders. If an adder has three inputs, a full adder (FA) is used, but if an adder has only two inputs, a half-adder (HA) is used. A half-adder is the same as a full adder with one of the inputs set to 0. This multiplier requires 16 AND gates, 8 full adders, and 4 half-adders. After the  $X$  and  $Y$  inputs have been applied, the carry must propagate along each row of cells, and the sum must propagate from row to row. The time required to complete the multiplication depends primarily on the propagation delay in the adders. The longest path from input to output goes through 8 adders. If  $t_{ad}$  is the worst-case (longest possible) delay through an adder, and  $t_g$  is the longest AND gate delay, then the worst-case time to complete the multiplication is  $8t_{ad} + t_g$ .

**Table 4-3 4-bit Multiplier Partial Products**

$X_3$ $Y_3$	$X_2$ $Y_2$	$X_1$ $Y_1$	$X_0$ $Y_0$	Multiplicand Multiplier
$X_3 Y_0$	$X_2 Y_0$	$X_1 Y_0$	$X_0 Y_0$	partial product 0
$X_3 Y_1$	$X_2 Y_1$	$X_1 Y_1$	$X_0 Y_1$	partial product 1
$C_{12}$	$C_{11}$	$C_{10}$		1st row carries
$S_{13}$	$S_{12}$	$S_{11}$	$S_{10}$	1st row sums
$X_3 Y_2$	$X_2 Y_2$	$X_1 Y_2$	$X_0 Y_2$	partial product 2
$C_{21}$	$C_{20}$			2nd row carries
$S_{23}$	$S_{22}$	$S_{21}$	$S_{20}$	2nd row sums
$X_3 Y_3$	$X_2 Y_3$	$X_1 Y_3$	$X_0 Y_3$	partial product 3
$C_{31}$	$C_{30}$			3rd row carries
$S_{33}$	$S_{32}$	$S_{31}$	$S_{30}$	3rd row sums
$P_7$	$P_6$	$P_5$	$P_4$	
			$P_3$	
			$P_2$	
			$P_1$	
			$P_0$	final product

In general, an  $n$ -bit-by- $n$ -bit array multiplier would require  $n^2$  AND gates,  $n(n - 2)$  full adders, and  $n$  half-adders. So the number of components required increases quadratically. For the serial-parallel multiplier previously designed, the amount of hardware required in addition to the control circuit increases linearly with  $n$ .

**Figure 4-7 Block Diagram of  $4 \times 4$  Array Multiplier**

For an  $n \times n$  array multiplier, the longest path from input to output goes through  $2n$  adders, and the corresponding worst-case multiply time is  $2nt_{ad} + t_g$ . The serial-parallel multiplier of the type previously designed requires  $2n$  clocks to complete the multiply in the worst case, although this can be reduced to  $n$  clocks using a technique discussed in the next section. The minimum clock period depends on the propagation delay through the  $n$ -bit adder as well as the propagation delay and setup time for the accumulator flip-flops.

## 4.4 MULTIPLICATION OF SIGNED BINARY NUMBERS

Several algorithms are available for multiplication of signed binary numbers. The following procedure is a straightforward way to carry out the multiplication:

1. Complement the multiplier if negative.
2. Complement the multiplicand if negative.
3. Multiply the two positive binary numbers.
4. Complement the product if it should be negative.

Although this method is conceptually simple, it requires more hardware and computation time than some of the other available methods.

The next method we describe requires only the ability to complement the multiplicand. Complementation of the multiplier or product is not necessary. Although the method works equally well with integers or fractions, we illustrate the method with fractions, since we will later use this multiplier as part of a multiplier for floating-point numbers. Using 2's complement for negative numbers, we will represent signed binary fractions in the following form:

$$0.101 \quad +5/8 \quad 1.011 \quad -5/8$$

The digit to the left of the binary point is the sign bit, which is 0 for positive fractions and 1 for negative fractions. In general, the 2's complement of a binary fraction  $F$  is  $F^* = 2 - F$ . Thus,  $-5/8$  is represented by  $10.000 - 0.101 = 1.011$ . (This method of defining 2's complement fractions is consistent with the integer case ( $N^* = 2^n - N$ ), since moving the binary point  $n - 1$  places to the left is equivalent to dividing by  $2^{n-1}$ .) The 2's complement of a fraction can be found by starting at the right end and complementing all the digits to the left of the first 1, the same as for the integer case. The 2's complement fraction  $1.000\ldots$  is a special case. It actually represents the number  $-1$ , since the sign bit is negative and the 2's complement of  $1.000\ldots$  is  $2 - 1 = 1$ . We cannot represent  $+1$  in the 2's complement fraction system, since  $0.111\ldots$  is the largest positive fraction.

When multiplying signed binary numbers, we must consider four cases:

Multiplicand	Multiplier
+	+
-	+
+	-
-	-

When both the multiplicand and the multiplier are positive, standard binary multiplication is used. For example,

$$\begin{array}{r}
 0.1\ 1\ 1 \\
 \times 0.1\ 0\ 1 \\
 \hline
 (0.00)0\ 1\ 1\ 1 \\
 (0.)0\ 1\ 1\ 1 \\
 \hline
 0.\ 1\ 0\ 0\ 0\ 1\ 1
 \end{array}
 \quad
 \begin{array}{l}
 (+7/8) \leftarrow \\
 (+5/8) \leftarrow \\
 (+7/64) \leftarrow \\
 (+7/16) \leftarrow \\
 (+35/64)
 \end{array}
 \quad
 \begin{array}{l}
 \text{Multiplicand} \\
 \text{Multiplier} \\
 \text{Note: The proper representation of the} \\
 \text{fractional partial products requires} \\
 \text{extension of the sign bit past the binary} \\
 \text{point, as indicated in parentheses. (Such} \\
 \text{extension is not necessary in the} \\
 \text{hardware.)}
 \end{array}$$

When the multiplicand is negative and the multiplier is positive, the procedure is the same as in the previous case, except that we must extend the sign bit of the multiplicand so that the partial products and final product will have the proper negative sign. For example,

$$\begin{array}{r}
 1.1\ 0\ 1 \\
 \times 0.1\ 0\ 1 \\
 \hline
 (1.11)1\ 1\ 0\ 1 \\
 (1.)1\ 1\ 0\ 1 \\
 \hline
 1.\ 1\ 1\ 0\ 0\ 0\ 1
 \end{array}
 \quad
 \begin{array}{l}
 (-3/8) \\
 (+5/8) \\
 (-3/64) \leftarrow \\
 (-3/16) \leftarrow \\
 (-15/64)
 \end{array}
 \quad
 \begin{array}{l}
 \text{Note: The extension of the sign bit} \\
 \text{provides proper representation of the} \\
 \text{negative products.}
 \end{array}$$

When the multiplier is negative and the multiplicand is positive, we must make a slight change in the multiplication procedure. A negative fraction of the form  $1.g$  has a numeric value  $-1 + 0.g$ ; for example,  $1.011 = -1 + 0.011 = -(1 - 0.011) = -0.101 = -5/8$ . Thus, when multiplying by a negative fraction of the form  $1.g$ , we treat the fraction part ( $g$ ) as a positive fraction, but the sign bit is treated as  $-1$ . Hence, multiplication proceeds in the normal way as we multiply by each bit of the fraction and accumulate the partial products. However, when we reach the negative sign bit, we must add in the 2's complement of the multiplicand instead of the multiplicand itself. The following example illustrates this:

$$\begin{array}{r}
 0.1\ 0\ 1 \\
 \times 1.1\ 0\ 1 \\
 \hline
 (0.00)0\ 1\ 0\ 1 \\
 (0.)0\ 1\ 0\ 1 \\
 (0.)0\ 1\ 1\ 0\ 0\ 1 \\
 \underline{1.\ 0\ 1\ 1} \\
 1.\ 1\ 1\ 0\ 0\ 0\ 1
 \end{array}
 \quad
 \begin{array}{l}
 (+5/8) \\
 (-3/8) \\
 (+5/64) \\
 (+5/16) \\
 (-5/8) \leftarrow \\
 (-15/64)
 \end{array}
 \quad
 \begin{array}{l}
 \text{Note: The 2's complement of the} \\
 \text{multiplicand is added at this point.}
 \end{array}$$

When both the multiplicand and multiplier are negative, the procedure is the same as before. At each step, we must be careful to extend the sign bit of the partial product to preserve the proper negative sign, and at the final step we must add in the 2's complement of the multiplicand, since the sign bit of the multiplier is negative. For example,

$$\begin{array}{r}
 & 1.1\ 0\ 1 \\
 \times & 1.1\ 0\ 1 \\
 \hline
 & (-3/8) \\
 (1. & 1\ 1) 1\ 1\ 0\ 1 \\
 (1. ) & 1\ 1\ 0\ 1 \\
 & (-3/64) \\
 1. & 1\ 1\ 0\ 0\ 0\ 1 \\
 0. & 0\ 1\ 1 \\
 \hline
 0. & 0\ 0\ 1\ 0\ 0\ 1 \\
 & (+3/8) \quad \leftarrow \quad \text{Add the 2's complement of the} \\
 & \text{multiplicand.} \\
 & (+9/64)
 \end{array}$$

In summary, the procedure for multiplying signed 2's complement binary fractions is the same as for multiplying positive binary fractions, except that we must be careful to preserve the sign of the partial product at each step, and if the sign of the multiplier is negative, we must complement the multiplicand before adding it in at the last step. The hardware is almost identical to that used for multiplication of positive numbers, except a completer must be added for the multiplicand.

Figure 4-8 shows the hardware required to multiply two 4-bit fractions (including the sign bit). A 5-bit adder is used so the sign of the sum is not lost due to a carry into the sign bit position. The *M* input to the control circuit is the currently active bit of the multiplier. Control signal *Sh* causes the accumulator to shift right one place with sign extension. *Ad* causes the ADDER output to be loaded into the left 5 bits of the accumulator. The carry

**Figure 4-8 Block Diagram for 2's Complement Multiplier**

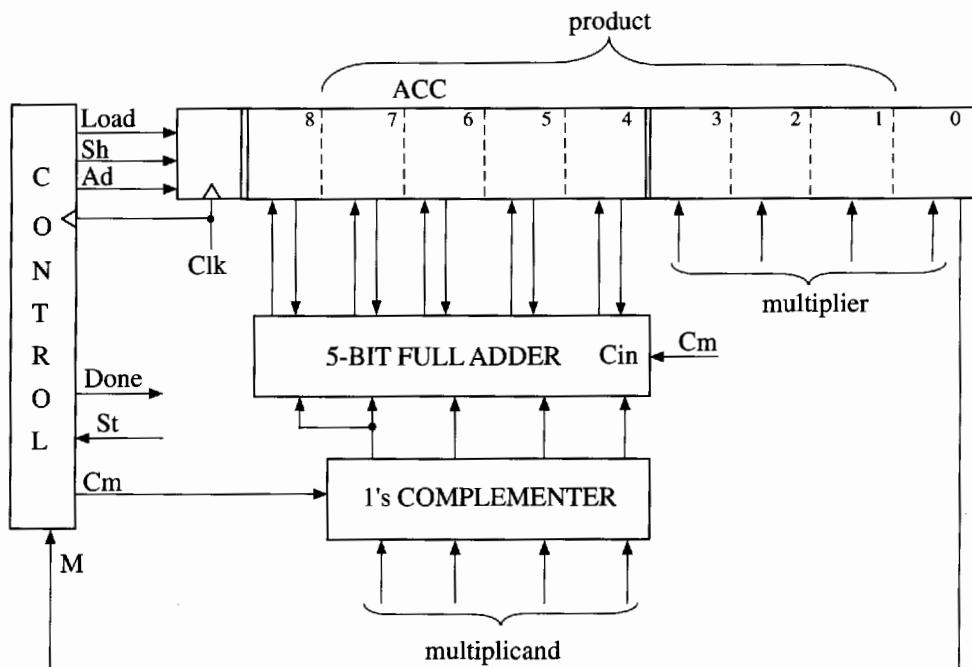
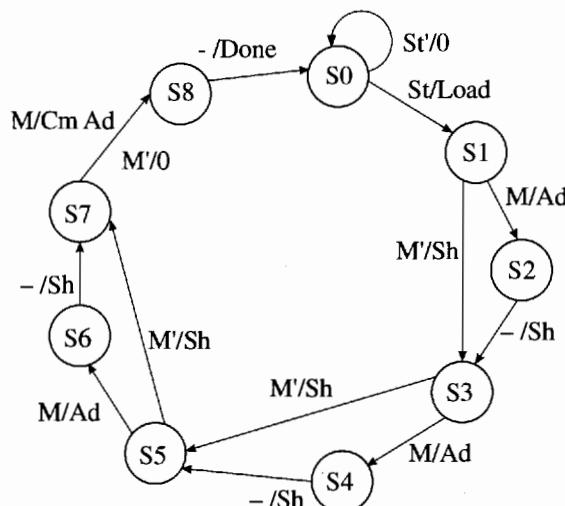


Figure 4-9 State Graph for 2's Complement Multiplier



out from the last bit of the adder is discarded, since we are doing 2's complement addition.  $C_m$  causes the multiplicand ( $M_{cand}$ ) to be complemented (1's complement) before it enters the adder inputs.  $C_m$  is also connected to the carry input of the adder so that when  $C_m = 1$ , the adder adds 1 plus the 1's complement of  $M_{cand}$  to the accumulator, which is equivalent to adding the 2's complement of  $M_{cand}$ . Figure 4-9 shows a state graph for the control circuit. Each multiplier bit ( $M$ ) is tested to determine whether to add and shift or whether to just shift. In state  $S_7$ ,  $M$  is the sign bit, and if  $M = 1$ , the complement of the multiplicand is added to the accumulator.

When the hardware in Figure 4-8 is used, the add and shift operations must be done at two separate clock times. We can speed up operation of the multiplier by moving the wires from the adder output one position to the right (Figure 4-10) so that the adder output is already shifted over one position when it is loaded into the accumulator. With this arrangement, the add and shift operations can occur at the same clock time, which leads to the control state graph of Figure 4-11. When the multiplication is complete, the product (6 bits plus sign) is in the lower 3 bits of A followed by B. The binary point then is in the middle of the A register. If we wanted it between the left two bits, we would have to shift A and B left one place.

Figure 4-10 Block Diagram for Faster Multiplier

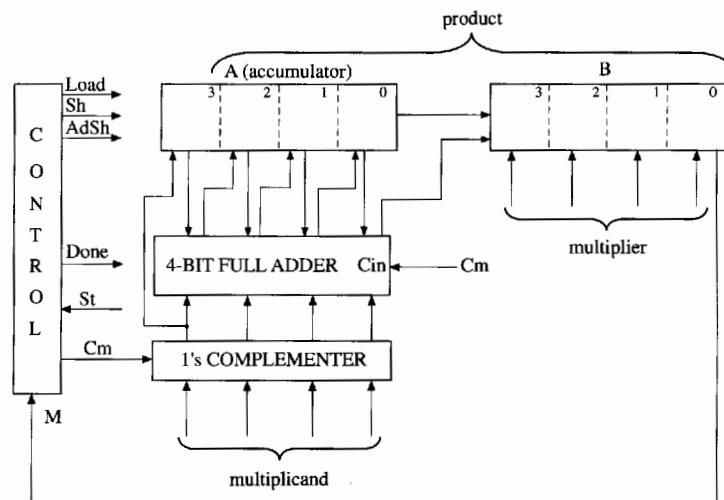


Figure 4-11 State Graph for Faster Multiplier

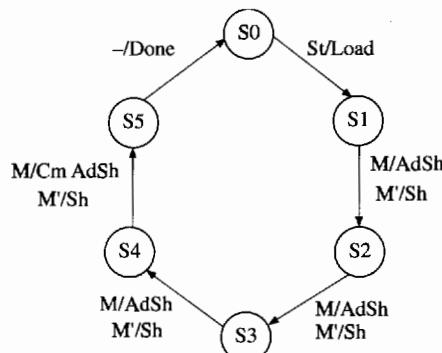


Figure 4-12 Behavioral Model for 2's Complement Multiplier

```

library BITLIB;
use BITLIB.bit_pack.all;

entity mult2C is
    port (CLK, St: in bit;
          Mplier, Mcand : in bit_vector(3 downto 0);
          Product: out bit_vector (6 downto 0);
          Done: out bit);
end mult2C;

architecture behav1 of mult2C is
    signal State : integer range 0 to 5;
    signal A, B: bit_vector(3 downto 0);
    alias M: bit is B(0);

```

```

begin
process
variable addout: bit_vector(4 downto 0);
begin
  wait until CLK = '1';
  case State is
    when 0=>                                --initial State
      if St='1' then
        A <= "0000";                         --Begin cycle
        B <= Mplier;                         --load the multiplier
        State <= 1;
      end if;
    when 1 | 2 | 3 =>                      --"addshift" State
      if M = '1' then
        addout := add4(A,Mcand,'0');          --Add multiplicand to A and
                                                shift
        A <= Mcand(3) & addout(3 downto 1);
        B <= addout(0) & B(3 downto 1);
      else
        A <= A(3) & A(3 downto 1);           --Arithmetic right shift
        B <= A(0) & B(3 downto 1);
      end if;
      State <= State + 1;
    when 4 =>                                --add complement if sign bit
                                                --of multiplier is 1
      if M = '1' then
        addout := add4(A, not Mcand,'1');
        A <= not Mcand(3) & addout(3 downto 1);
        B <= addout(0) & B(3 downto 1);
      else
        A <= A(3) & A(3 downto 1);           --Arithmetic right shift
        B <= A(0) & B(3 downto 1);
      end if;
      State <= 5;
      wait for 0 ns;
      Done <= '1';
      Product <= A(2 downto 0) & B;
    when 5 =>                                --output product
      State <= 0;
      Done <= '0';
  end case;
end process;
end behav1;

```

A behavioral VHDL model for this multiplier is shown in Figure 4-12. Shifting the *A* and *B* registers together is accomplished by the sequential statements

```

A <= A(3) & A(3 downto 1);
B <= A(0) & B(3 downto 1);

```

Although these statements are executed sequentially,  $A$  and  $B$  are both scheduled to be updated at the same delta time. Therefore, the old value of  $A(0)$  is used when computing the new value of  $B$ .

A variable *addout* has been defined to represent the 5-bit output of the adder. In states 1 through 4, if the current multiplier bit  $M$  is 1, then the sign bit of the multiplicand followed by three bits of *addout* are loaded into  $A$ . At the same time, the low-order bit of *addout* is loaded into  $B$  along with the high-order 3 bits of  $B$ . The done signal is turned on when control goes to state 5, and then the new value of the product is output. The wait for 0 ns is required so that  $A$  and  $B$  are updated to their final values before outputting the new product.

Before continuing with the design, we will test the behavioral level VHDL code to make sure that the algorithm is correct and consistent with the hardware block diagram. At early stages of testing, we will want a step-by-step printout to verify the internal operations of the multiplier and to aid in debugging if required. When we think that the multiplier is functioning properly, then we will only want to look at the final product output so that we can quickly test a large number of cases.

**Figure 4-13 Command File and Simulation Results for (+5/8 by -3/8)**

```
-- command file to test signed multiplier
list CLK St State A B Done Product
force st 1 2, 0 22
force clk 1 0, 0 10 - repeat 20
-- (5/8 * -3/8)
force Mcand 0101
force Mpplier 1101
run 120
```

ns	delta	CLK	St	State	A	B	Done	Product
0	+1	1	0		0 0000	0000	0	0000000
2	+0	1	1		0 0000	0000	0	0000000
10	+0	0	1		0 0000	0000	0	0000000
20	+1	1	1		1 0000	1101	0	0000000
22	+0	1	0		1 0000	1101	0	0000000
30	+0	0	0		1 0000	1101	0	0000000
40	+1	1	0		2 0010	1110	0	0000000
50	+0	0	0		2 0010	1110	0	0000000
60	+1	1	0		3 0001	0111	0	0000000
70	+0	0	0		3 0001	0111	0	0000000
80	+1	1	0		4 0011	0011	0	0000000
90	+0	0	0		4 0011	0011	0	0000000
100	+2	1	0		5 1111	0001	1	1110001
110	+0	0	0		5 1111	0001	1	1110001
120	+1	1	0		0 1111	0001	0	1110001

Figure 4-13 shows the command file and test results for multiplying  $+5/8$  by  $-3/8$ . A clock is defined with a 20-ns period. The *St* signal is turned on at 2 ns and turned off one clock period later. By inspection of the state graph, the multiplication requires six clocks,

so the run time is set at 120 ns. The simulator output corresponds to the example given on page 133.

To thoroughly test the multiplier, we need to test not only the four standard cases (+ +, + -, - +, and - -) but also special cases and limiting cases. Test values for the multiplicand and multiplier should include 0, largest positive fraction, most negative fraction, and all 1s. We will write a VHDL test bench to test the multiplier. This test bench (Figure 4-14) supplies a sequence of values for the multiplicand and multiplier. It also

**Figure 4-14** Test Bench for Signed Multiplier

```
library BITLIB;
use BITLIB.bit_pack.all;

entity testmult is
end testmult;

architecture test1 of testmult is
component mult2C
    port(CLK, St: in bit;
          Mplier,Mcand : in bit_vector(3 downto 0);
          Product: out bit_vector (6 downto 0);
          Done: out bit);
end component;

constant N: integer := 11;
type arr is array(1 to N) of bit_vector(3 downto 0);
constant Mcandarr: arr := ("0111", "1101", "0101", "1101", "0111",
                           "1000", "0111", "1000", "0000", "1111", "1011");
constant Mplierarr: arr := ("0101", "0101", "1101", "1101", "0111",
                           "0111", "1000", "1000", "1101", "1111", "0000");
signal CLK, St, Done: bit;
signal Mplier, Mcand: bit_vector(3 downto 0);
signal Product: bit_vector(6 downto 0);
begin
    CLK <= not CLK after 10 ns;
process
begin
    for i in 1 to N loop
        Mcand <= Mcandarr(i);
        Mplier <= Mplierarr(i);
        St <= '1';
        wait until rising_edge(CLK);
        St <= '0';
        wait until falling_edge(Done);
    end loop;
end process;
mult1: mult2c port map(Clk, St, Mplier, Mcand, Product, Done);
end test1;
```

generates the clock and start signal. The multiplicand and multiplier test values are placed in constant arrays. The for loop reads values from these arrays and then sets the start signal to '1'. After the next clock, the start signal is turned off. Since the done signal is turned off at the same time the multiplier control goes back to  $S_0$ , the process waits for the falling edge of done before looping back to supply new values of *M<sub>cand</sub>* and *M<sub>plier</sub>*.

Figure 4-15 shows the command file and simulator output. We have annotated the simulator output to interpret the test results. The *-NOtrigger* together with the *-Trigger* done in the list statement causes the output to be displayed only when the *Done* signal changes. Without the *-NOtrigger* and *-Trigger*, the output would be displayed every time any signal on the list changed. All the product outputs are correct, except for the special case of  $-1 \times -1$  ( $1.000 \times 1.000$ ), which gives  $1.000000$  ( $-1$ ) instead of  $+1$ . This occurs because no representation of  $+1$  is possible without adding another bit.

**Figure 4-15 Command File and Simulation of Signed Multiplier**

```
-- Command file to test results of signed multiplier
list -NOTrigger Mplier Mcand product -Trigger done
run 1320

ns    delta mplier mcand product done
  0      +1    0101   0111 0000000    0
  90     +2    0101   0111 0100011    1  5/8 * 7/8 = 35/64
 110     +2    0101   1101 0100011    0
 210     +2    0101   1101 1110001    1  5/8 * -3/8 = -15/64
 230     +2    1101   0101 1110001    0
 330     +2    1101   0101 1110001    1  -3/8 * 5/8 = -15/64
 350     +2    1101   1101 1110001    0
 450     +2    1101   1101 0001001    1  -3/8 * -3/8 = 9/64
 470     +2    0111   0111 0001001    0
 570     +2    0111   0111 0110001    1  7/8 * 7/8 = 49/64
 590     +2    0111   1000 0110001    0
 690     +2    0111   1000 1001000    1  7/8 * -1 = -7/8
 710     +2    1000   0111 1001000    0
 810     +2    1000   0111 1001000    1  -1 * 7/8 = -7/8
 830     +2    1000   1000 1001000    0
 930     +2    1000   1000 1000000    1  -1 * -1 = -1 (error)
 950     +2    1101   0000 1000000    0
1050     +2    1101   0000 0000000    1  -3/8 * 0 = 0
1070     +2    1111   1111 0000000    0
1170     +2    1111   1111 0000001    1  -1/8 * -1/8 = 1/64
1190     +2    0000   1011 0000001    0
1290     +2    0000   1011 0000000    1  0 * -3/8 = 0
1310     +2    0101   0111 0000000    0
```

Next we refine the VHDL model for the signed multiplier by explicitly defining the control signals and the actions that occur when each control signal is asserted. The VHDL code (Figure 4-16) is organized in a manner similar to the Mealy machine model of Figure 1-16. In the first part of the process, the *Nextstate* and output control signals are defined for each present *State*. After waiting for the rising edge of the clock, the appropriate

**Figure 4-16 Model for 2's Complement Multiplier with Control Signals**

```
-- This architecture of a 4-bit multiplier for 2's complement
-- numbers uses control signals.

architecture behave2 of mult2CS is
  signal State, Nextstate: integer range 0 to 5;
  signal A, B: bit_vector(3 downto 0);
  signal AdSh, Sh, Load, Cm, Done: bit;
  alias M: bit is B(0);
begin
  process
    variable addout: bit_vector(4 downto 0);
  begin
    Load <= '0'; AdSh <= '0'; Sh <= '0'; Cm <= '0'; Done <= '0';
    wait for 0 ns;
    case State is
      when 0=>                               --initial State
        if St='1' then Load <= '1'; Nextstate <= 1; end if;
      when 1 | 2 | 3 =>                      --"add/shift" State
        if M = '1' then AdSh <= '1';
        else Sh <= '1';
        end if;
      Nextstate <= State + 1;
      when 4 =>                            --add complement if sign
        if M = '1' then                      --bit of multiplier is 1
          Cm <= '1'; AdSh <= '1';
          else Sh <= '1';
          end if;
        nextstate <= 5;
      when 5 =>                           --output product
        done <= '1';
        nextstate <= 0;
    end case;
    wait until CLK = '1';                  --executes on rising edge
    if Cm = '0' then addout := add4(A,Mcand,'0');
    else addout := add4(A, not Mcand,'1');
    end if;
    if Load = '1' then                    --load the multiplier
      A <= "0000";
      B <= Mplier;
    end if;
    if AdSh = '1' then                   --Add multiplicand to A and shift
      A <= (Mcand(3) xor Cm) & addout(3 downto 1);
      B <= addout(0) & B(3 downto 1);
    end if;
    if Sh = '1' then
      A <= A(3) & A(3 downto 1);
      B <= A(0) & B(3 downto 1);
    end if;
  end process;
end architecture;
```

```

if Done = '1' then
    Product <= A(2 downto 0) & B;
end if;
State <= Nextstate;
end process;
end behave2;

```

registers are updated and the *State* is updated. We can test the VHDL code of Figure 4-16 using the same test file we used previously and verify that we get the same product outputs.

Since the control state graph (Figure 4-11) is a loop of states, it is natural to design the control network using a counter. We use a 74163 counter and associated logic, as shown in Figure 4-17. The counter output,  $Q_3Q_2Q_1Q_0$ , represents the state of the control network. We could have used only 3 bits of the counter to represent the six states. However, the logic is simpler if we use all 4 bits with the following state assignment:

$$S_0 \rightarrow 0000, S_1 \rightarrow 0100, S_2 \rightarrow 0101, S_3 \rightarrow 0110, S_4 \rightarrow 0111, S_5 \rightarrow 1000$$

With this assignment the counter should be cleared in  $S_5$ , loaded with  $Din = 0100$  in  $S_0$ , and incremented in the remaining states. By inspection

$CLR1$	$= Q'_3, Done = Q_3$	( $CLR1 = 0$ and $Done = 1$ in state 1000.)
$Load$	$= Q'_3Q'_2St$	(Load in state 0000 when $St = 1$ .)
$Ld1$	$= Load'$	(Load the counter in state 0000 when $St = 1$ .)
$P1$	$= Q_2$	(Increment in states 0100, 0101, 0110, 0111.)
$Sh$	$= M'Q_2$	(Shift in states 0100, 0101, 0110, 0111 if $M = 0$ .)
$AdSh$	$= MQ_2$	(Add/shift in states 0100, 0101, 0110, 0111 if $M = 1$ .)
$Cm$	$= MQ_1Q_0$	

To verify this design, we replace the behavioral description of the control network with these and update the states of the flip-flops after waiting for the clock edge (see Figure 4-18). The registers are updated in the same way as before. We have also explicitly defined the completer output as *comp*. We can then compute the completer output by XORing each input bit with *Cm*. In the statement

```
comp <= Mcand xor Cm&Cm&Cm&Cm
```

four copies of *Cm* are made using the concatenation operator. This corresponds to the actual hardware, since *Cm* will be connected to four XOR gate inputs. In the port map for the counter, the reserved word **open** is used to indicate no connection to the carry output. Again, we can use the same test files as before and verify that the product outputs are the same.

Figure 4-17 Realization of Multiplier Control Network

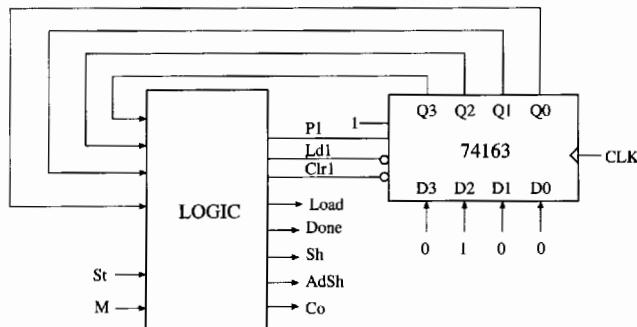


Figure 4-18 Model for 2's Complement Multiplier Using Control Equations

```
-- This model of a 4-bit multiplier for 2's complement numbers
-- implements the controller using a counter and logic equations.
library BITLIB;
use BITLIB.bit_pack.all;

entity mult2CEQ is
  port(CLK, St: in bit;
        Mplier, Mcand: in bit_vector(3 downto 0);
        Product: out bit_vector(6 downto 0));
end mult2CEQ;

architecture m2ceq of mult2CEQ is
  signal A, B, Q, Comp: bit_vector(3 downto 0);
  signal addout: bit_vector(4 downto 0);
  signal AdSh, Sh, Load, Cm, Done, Ld1, CLR1, P1: bit;
  Signal One: bit:='1';
  Signal Din: bit_vector(3 downto 0) := "0100";
begin
  Count1: C74163 port map (Ld1, CLR1, P1, One, CLK, Din, open, Q);
  P1 <= Q(2);
  CLR1 <= not Q(3);
  Done <= Q(3);
  Sh <= not M and Q(2);
  AdSh <= M and Q(2);
  Cm <= Q(1) and Q(0) and M;
  Load <= not Q(3) and not Q(2) and St;
  Ld1 <= not Load;
  Comp <= Mcand xor (Cm & Cm & Cm & Cm); --complement Mcand if Cm='1'
  addout <= add4(A,Comp,Cm); --add completer output to A
```

```

process
begin
  wait until CLK = '1';           --executes on rising edge
  if Load = '1' then             --load the multiplier
    A <= "0000";
    B <= Mplier;
  end if;
  if AdSh = '1' then            --Add multiplicand to A and shift
    A <= (Mcand(3) xor Cm) & addout(3 downto 1);
    B <= addout(0) & B(3 downto 1);
  end if;
  if Sh = '1' then              --Right shift with sign extend
    A <= A(3) & A(3 downto 1);
    B <= A(0) & B(3 downto 1);
  end if;
  if Done = '1' then
    Product <= A(2 downto 0) & B;
  end if;
end process;
end m2ceq;

```

## 4.5 DESIGN OF A BINARY DIVIDER

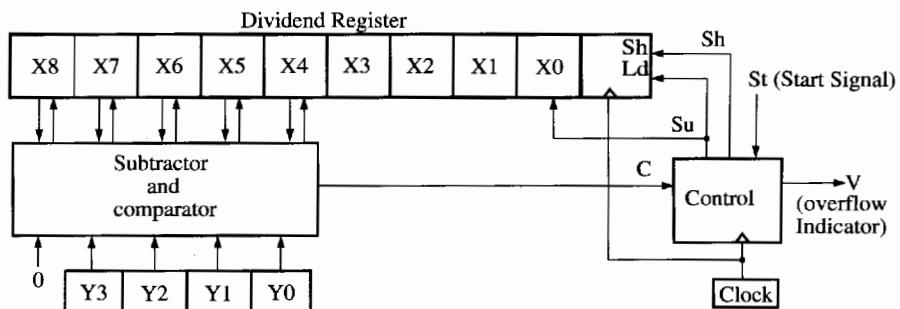
We will consider the design of a parallel divider for positive binary numbers. As an example, we will design a network to divide an 8-bit dividend by a 4-bit divisor to obtain a 4-bit quotient. The following example illustrates the division process:

$$\begin{array}{r}
 & \text{1010} & \text{quotient} \\
 \text{divisor} & \underline{\text{1101}} & \text{10000111} & \text{dividend} \\
 & \underline{\text{1101}} & & \\
 & \text{0111} & & \\
 & \underline{\text{0000}} & & \\
 & \text{1111} & & \\
 & \underline{\text{1101}} & & \\
 & \text{0101} & & \\
 & \underline{\text{0000}} & & \\
 & \text{0101} & & \text{remainder}
 \end{array}$$

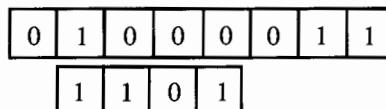
(135 ÷ 13 = 10 with a remainder of 5)

Just as binary multiplication can be carried out as a series of add and shift operations, division can be carried out by a series of subtract and shift operations. To construct the divider, we will use a 9-bit dividend register and a 4-bit divisor register, as shown in Figure 4-19. During the division process, instead of shifting the divisor right before each subtraction, we will shift the dividend to the left. Note that an extra bit is required on the left end of the dividend register so that a bit is not lost when the dividend is shifted left. Instead of using a separate register to store the quotient, we will enter the quotient bit-by-bit into the right end of the dividend register as the dividend is shifted left.

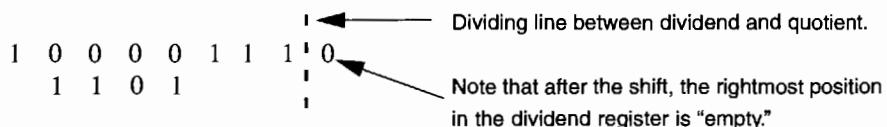
Figure 4-19 Block Diagram for Parallel Binary Divider



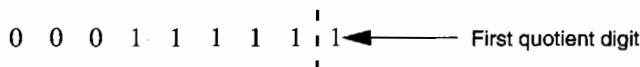
The preceding division example (135 divided by 13) is reworked next, showing the location of the bits in the registers at each clock time. Initially, the dividend and divisor are entered as follows:



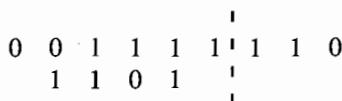
Subtraction cannot be carried out without a negative result, so we will shift before we subtract. Instead of shifting the divisor one place to the right, we will shift the dividend one place to the left:



Subtraction is now carried out and the first quotient digit of 1 is stored in the unused position of the dividend register:



Next we shift the dividend one place to the left:



Since subtraction would yield a negative result, we shift the dividend to the left again, and the second quotient bit remains zero:

$$\begin{array}{cccccc|cc} 0 & 1 & 1 & 1 & 1 & 1 & | & 1 & 0 & 0 \\ & 1 & 1 & 0 & 1 & | & & & & \end{array}$$

Subtraction is now carried out, and the third quotient digit of 1 is stored in the unused position of the dividend register:

$$\begin{array}{cccccc|cc} 0 & 0 & 0 & 1 & 0 & 1 & | & 1 & 0 & 1 \\ & & & & & & | & & & \leftarrow \text{Third quotient digit} \end{array}$$

A final shift is carried out and the fourth quotient bit is set to 0:

$$\begin{array}{cccccc|cc} 0 & 0 & 1 & 0 & 1 & 1 & | & 1 & 0 & 1 & 0 \\ \underbrace{0 \quad 0 \quad 1 \quad 0} & & | & \underbrace{1 \quad 1} & & & & & & & \\ \text{remainder} & & & \text{quotient} & & & & & & & \end{array}$$

The final result agrees with that obtained in the first example.

If, as a result of a division operation, the quotient contains more bits than are available for storing the quotient, we say that an *overflow* has occurred. For the divider of Figure 4-19, an overflow would occur if the quotient is greater than 15, since only 4 bits are provided to store the quotient. It is not actually necessary to carry out the division to determine if an overflow condition exists, since an initial comparison of the dividend and divisor will tell if the quotient will be too large. For example, if we attempt to divide 135 by 7, the initial contents of the registers are:

$$\begin{array}{ccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ & 0 & 1 & 1 & 1 & & & & \end{array}$$

Since subtraction can be carried out with a nonnegative result, we should subtract the divisor from the dividend and enter a quotient bit of 1 in the rightmost place in the dividend register. However, we cannot do this because the rightmost place contains the least significant bit of the dividend, and entering a quotient bit here would destroy that dividend bit. Therefore, the quotient would be too large to store in the 4 bits we have allocated for it, and we have detected an overflow condition. In general, for Figure 4-19, if initially  $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$  (i.e., if the left 5 bits of the dividend register exceed or equal the divisor), the quotient will be greater than 15 and an overflow occurs. Note that if  $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$ , the quotient is

$$\frac{X_8X_7X_6X_5X_4X_3X_2X_1X_0}{Y_3Y_2Y_1Y_0} \geq \frac{X_8X_7X_6X_5X_40000}{Y_3Y_2Y_1Y_0} = \frac{X_8X_7X_6X_5X_4 \times 16}{Y_3Y_2Y_1Y_0} \geq 16$$

The operation of the divider can be explained in terms of the block diagram of Figure 4-19. A shift signal ( $Sh$ ) will shift the dividend one place to the left. A subtract signal ( $Su$ ) will subtract the divisor from the 5 leftmost bits in the dividend register and set the quotient bit (the rightmost bit in the dividend register) to 1. If the divisor is greater than the 4 leftmost dividend bits, the comparator output is  $C = 0$ ; otherwise,  $C = 1$ . The control circuit generates the required sequence of shift and subtract signals. Whenever  $C = 0$ , subtraction cannot occur without a negative result, so a shift signal is generated. Whenever  $C = 1$ , a subtract signal is generated, and the quotient bit is set to 1.

Figure 4-20 shows the state diagram for the control circuit. When a start signal ( $St$ ) occurs, the 8-bit dividend and 4-bit divisor are loaded into the appropriate registers. If  $C$  is 1, the quotient would require five or more bits. Since space is only provided for a 4-bit quotient, this condition constitutes an overflow, so the divider is stopped and the overflow indicator is set by the  $V$  output. Normally, the initial value of  $C$  is 0, so a shift will occur first, and the control circuit will go to state  $S2$ . Then, if  $C = 1$ , subtraction occurs. After the subtraction is completed,  $C$  will always be 0, so the next clock pulse will produce a shift. This process continues until four shifts have occurred and the control is in state  $S5$ . Then a final subtraction occurs if necessary, and the control returns to the stop state. For this example, we will assume that when the start signal ( $St$ ) occurs, it will be 1 for one clock time, and then it will remain 0 until the control network is back in state  $S0$ . Therefore,  $St$  will always be 0 in states  $S1$  through  $S5$ .

**Figure 4-20 State Diagram for Divider Control Circuit**

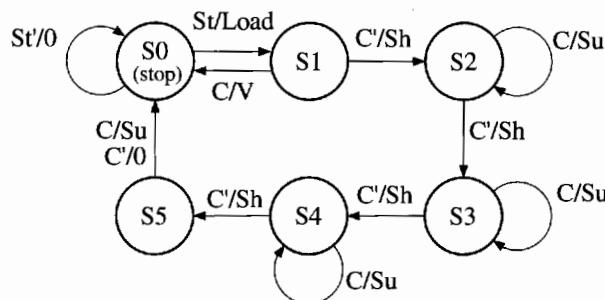


Table 4-4 gives the state table for the control circuit. Since we assumed that  $St = 0$  in states  $S1$ ,  $S2$ ,  $S3$ , and  $S4$ , the next states and outputs are don't cares for these states when  $St = 1$ . The entries in the output table indicate which outputs are 1. For example, the entry  $Sh$  means  $Sh = 1$  and the other outputs are 0.

**Table 4-4 State Table for Divider Control Circuit**

State	StC				StC			
	00	01	11	10	00	01	11	10
S <sub>0</sub>	S <sub>0</sub>	S <sub>0</sub>	S <sub>1</sub>	S <sub>1</sub>	0	0	Load	Load
S <sub>1</sub>	S <sub>2</sub>	S <sub>0</sub>	—	—	Sh	V	—	—
S <sub>2</sub>	S <sub>3</sub>	S <sub>2</sub>	—	—	Sh	Su	—	—
S <sub>3</sub>	S <sub>4</sub>	S <sub>3</sub>	—	—	Sh	Su	—	—
S <sub>4</sub>	S <sub>5</sub>	S <sub>4</sub>	—	—	Sh	Su	—	—
S <sub>5</sub>	S <sub>0</sub>	S <sub>0</sub>	—	—	0	Su	—	—

This example illustrates a general method for designing a divider for unsigned binary numbers, and the design can easily be extended to larger numbers such as 16 bits divided by 8 bits or 32 bits divided by 16 bits. We now design a divider for signed (2's complement) binary numbers that divides a 32-bit dividend by a 16-bit divisor to give a 16-bit quotient. Although algorithms exist to divide the signed numbers directly, such algorithms are rather complex. So we take the easy way out and complement the dividend and divisor if they are negative; when division is complete, we complement the quotient if it should be negative.

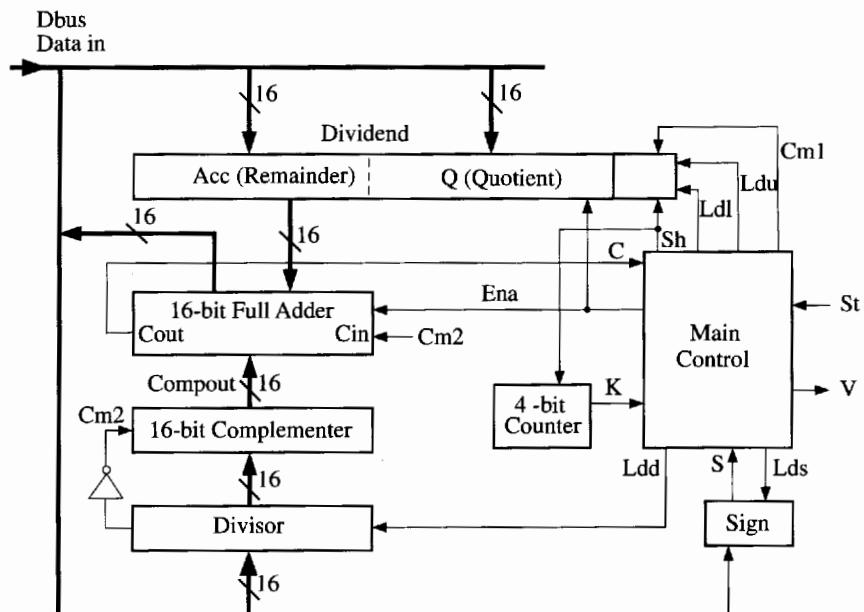
Figure 4-21 shows a block diagram for the divider. We use a 16-bit bus to load the registers. Since the dividend is 32 bits, two clocks are required to load the upper and lower halves of the dividend register, and one clock is needed to load the divisor. An extra sign flip-flop is used to store the sign of the dividend. We will use a dividend register with a built-in 2's completer. The subtracter consists of an adder and a completer, so subtraction can be accomplished by adding the 2's complement of the divisor to the dividend register. If the divisor is negative, using a separate step to complement it is unnecessary; we can simply disable the completer and add the negative divisor instead of subtracting its complement. The control network is divided into two parts—a main control, which determines the sequence of shifts and subtracts, and a counter, which counts the number of shifts. The counter outputs a signal  $K = 1$  when 15 shifts have occurred. Control signals are defined as follows:

- LdU* Load upper half of dividend from bus.
- LdL* Load lower half of dividend from bus.
- Lds* Load sign of dividend into sign flip-flop.
- S* Sign of dividend.
- Cm1* Complement dividend register (2's complement).
- Ldd* Load divisor from bus.
- Su* Enable adder output onto bus (*Ena*) and load upper half of dividend from bus.
- Cm2* Enable completer. (*Cm2* equals the complement of the sign bit of the divisor, so a positive divisor is complemented and a negative divisor is not.)
- Sh* Shift the dividend register left one place and increment the counter.
- C* Carry output from adder. (If  $C = 1$ , the divisor can be subtracted from the upper dividend.)
- St* Start.

$V$  Overflow.

$Q_{neg}$  Quotient will be negative. ( $Q_{neg} = 1$  when the sign of the dividend and divisor are different.)

Figure 4-21 Block Diagram for Signed Divider



The procedure for carrying out the signed division is as follows:

1. Load the upper half of the dividend from the bus, and copy the sign of the dividend into the sign flip-flop.
2. Load the lower half of the dividend from the bus.
3. Load the divisor from the bus.
4. Complement the dividend if it is negative.
5. If an overflow condition is present, go to the done state.
6. Else carry out the division by a series of shifts and subtracts.
7. When division is complete, complement the quotient if necessary, and go to the done state.

Testing for overflow is slightly more complicated than for the case of unsigned division. First, consider the case of all positive numbers. Since the divisor and quotient are each 15 bits plus sign, their maximum value is 7FFFh. Since the remainder must be less than the divisor, its maximum value is 7FFEh. Therefore, the maximum dividend for no overflow is

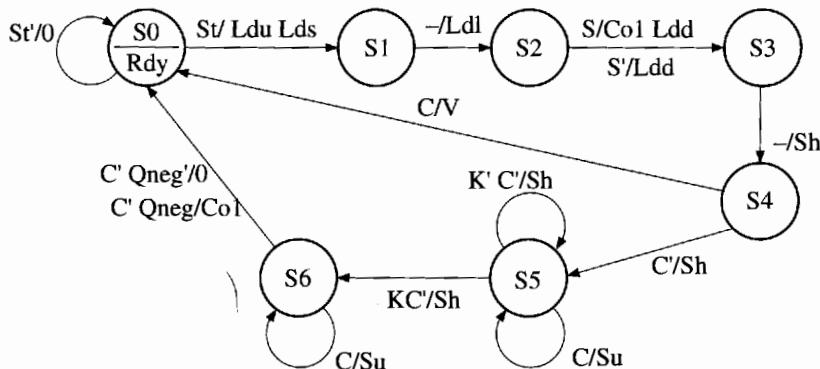
$$\text{divisor} \times \text{quotient} + \text{remainder} = 7FFFh \times 7FFFh + 7FFEh = 3FFF7FFFh$$

If the dividend is 1 larger (3FFF8000h), division by 7FFFh (or anything smaller) will give an overflow. We can test for the overflow condition by shifting the dividend left one place and then comparing the upper half of the dividend (divu) with the divisor. If  $\text{divu} \geq \text{divisor}$ , the quotient would be greater than the maximum value, which is an overflow condition. For the preceding example, shifting 3FFF8000h left once gives 7FFF0000h. Since 7FFFh equals the divisor, there is an overflow. On the other hand, shifting 3FFF7FFFh left gives 7FFEFFFh, and since 7FEh < 7FFFh, no overflow occurs when dividing by 7FFFh.

Another way of verifying that we must shift the dividend left before testing for overflow is as follows. If we shift the dividend left one place and then  $\text{divu} \geq \text{divisor}$ , we could subtract and generate a quotient bit of 1. However, this bit would have to go in the sign bit position of the quotient. This would make the quotient negative, which is incorrect. After testing for overflow, we must shift the dividend left again, which gives a place to store the first quotient bit after the sign bit. Since we work with the complement of a negative dividend or a negative divisor, this method for detecting overflow will work for negative numbers, except for the special case where the dividend is 80000000h (the largest negative value). Modifying the design to detect overflow in this case is left as an exercise.

Figure 4-22 shows the state graph for the control network. When  $S_t = 1$ , the registers are loaded. In S2, if the sign of the dividend ( $S$ ) is 1, the dividend is complemented. In S3, we shift the dividend left one place and then we test for overflow in S4. If  $C = 1$ , subtraction is possible, which implies an overflow, and the network goes to the done state. Otherwise, the dividend is shifted left. In S5,  $C$  is tested. If  $C = 1$ , then  $S_u = 1$ , which implies  $Ldu$  and  $E_{na}$ , so the adder output is enabled onto the bus and loaded into the upper dividend register to accomplish the subtraction. Otherwise,  $S_h = 1$  and the dividend register is shifted. This continues until  $K = 1$  at which time the last shift occurs if  $C = 0$ , and the network goes to S6. Then if the sign of the divisor and the saved sign of the dividend are different, the dividend register is complemented so that the quotient will have the correct sign.

**Figure 4-22 State Graph for Signed Divider Control Network**



The VHDL code for the signed divider is shown in Figure 4-23. Since the 1's complementer and adder are combinational networks, we have represented their operation by concurrent statements. *ADDVEC* is executed any time *ACC* or *compout* changes, so *Sum* and carry are immediately recomputed. All the signals that represent register outputs are updated on the rising edge of the clock, so these signals are updated in the process after waiting for *CLK* to change to '1'. For example, *ADDVEC* is called in states 2 and 6 to store the 2's complement of the dividend back into the dividend register. The counter is simulated by an integer signal, *count*. For convenience in listing the simulator output, we have added a ready signal (*Rdy*), which is turned on in S0 to indicate that the division is completed.

Figure 4-23 VHDL Model of 32-bit Signed Divider

```

library BITLIB;
use BITLIB.bit_pack.all;

entity sdiv is
    port(Clk,St: in bit;
          Dbus: in bit_vector(15 downto 0);
          Quotient: out bit_vector(15 downto 0);
          V, Rdy: out bit);
end sdiv;

architecture Signdiv of Sdiv is
    constant zero_vector: bit_vector(31 downto 0):=(others=>'0');
    signal State: integer range 0 to 6;
    signal Count : integer range 0 to 15;
    signal Sign,C,NC: bit;
    signal Divisor,Sum,Compout: bit_vector(15 downto 0);
    signal Dividend: bit_vector(31 downto 0);
    alias Q: bit_vector(15 downto 0) is Dividend(15 downto 0);
    alias Acc: bit_vector(15 downto 0) is Dividend(31 downto 16);
begin
    begin
        compout <= divisor when divisor(15) = '1'           -- concurrent statements
        else not divisor;                                     -- 1's complementer
        Addvec(Acc,compout,not divisor(15),Sum,C,16);      -- 16-bit adder
        Quotient <= Q;
        Rdy <= '1' when State=0 else '0';
    process
    begin
        wait until Clk = '1';                                -- wait for rising edge of clock
        case State is
            when 0=>
                if St = '1' then
                    Acc <= Dbus;                               -- load upper dividend
                    Sign <= Dbus(15);
                    State <= 1;
                    V <= '0';                                -- initialize overflow
                    Count <= 0;                             -- initialize counter
                end if;

```

```

when 1=>
  Q <= Dbus;                                -- load lower dividend
  State <= 2;
when 2=>
  Divisor <= Dbus;
  if Sign ='1' then      -- two's complement Dividend if necessary
    addvec(not Dividend,zero_vector,'1',Dividend,NC,32);
  end if;
  State <= 3;
when 3=>
  Dividend <= Dividend(30 downto 0) & '0';   -- left shift
  Count <= Count+1;
  State <= 4;
when 4 =>
  if C ='1' then                           -- C
    v <= '1';
    State <= 0;
  else                                     -- C'
    Dividend <= Dividend(30 downto 0) & '0'; -- left shift
    Count <= Count+1;
    State <= 5;
  end if;
when 5 =>
  if C = '1' then                         -- C
    ACC <= Sum;                          -- subtract
    Q(0)<= '1';
  else
    Dividend <= Dividend(30 downto 0) & '0'; -- left shift
    if Count = 15 then                   -- KC'
      State <= 6; Count <= 0;
    else Count <= Count+1;
    end if;
  end if;
when 6=>
  if C = '1' then                         -- C
    Acc <= Sum;                          -- subtract
    Q(0) <= '1';
  else if (Sign xor Divisor(15))='1' then -- C'Qneg
    addvec(not Dividend,zero_vector,'1',Dividend,NC,32);
  end if;                                 -- 2's complement Dividend
  state <= 0;
  end if;
end case;
end process;
end signdiv;

```

We are now ready to test the divider design by using the VHDL simulator. We will need a comprehensive set of test examples that will test all the different special cases that can arise in the division process. To start with, we need to test the basic operation of the divider for all the different combinations of signs for the divisor and dividend (+ +, + -, - +, and - -). We also need to test the overflow detection for these four cases. Limiting cases must also be tested, including largest quotient, zero quotient, etc. Use of a VHDL test bench is convenient because the test data must be supplied in sequence at certain times, and the length of time to complete the division is dependent on the test data. Figure 4-24 shows a test bench for the divisor. The test bench contains a dividend array and a divisor array for the test data. The notation X"07FF00BB" is the hexadecimal representation of a bit string. The process in testsdiv first puts the upper dividend on *Dbus* and supplies a start signal. After waiting for the clock, it puts the lower dividend on *Dbus*. After the next clock, it puts the divisor on *Dbus*. It then waits until the *Rdy* signal indicates that division is complete before continuing. *Count* is set equal to the loop-index, so that the change in *Count* can be used to trigger the listing output.

**Figure 4-24 Test Bench for Signed Divider**

```

library BITLIB;
use BITLIB.bit_pack.all;

entity testsdiv is
end testsdiv;

architecture test1 of testsdiv is
component sdiv
port(Clk,St: in bit;
      Dbus: in bit_vector(15 downto 0);
      Quotient: out bit_vector(15 downto 0);
      V, Rdy: out bit);
end component;

constant N: integer := 12;                                -- test sdiv N times
type arr1 is array(1 to N) of bit_vector(31 downto 0);
type arr2 is array(1 to N) of bit_vector(15 downto 0);
constant dividendarr: arr1 := (X"0000006F", X"07FF00BB", X"FFFFFE08",
                               X"FF80030A", X"3FFF8000", X"3FFF7FFF", X"C0008000", X"C0008000",
                               X"C0008001", X"00000000", X"FFFFFFFF", X"FFFFFFFF");
constant divisorarr: arr2 := (X"0007", X"E005", X"001E", X"EFFA", X"7FFF",
                               X"7FFF", X"7FFF", X"8000", X"7FFF", X"0001", X"7FFF", X"0000");
signal CLK, St, V, Rdy: bit;
signal Dbus, Quotient, divisor: bit_vector(15 downto 0);
signal Dividend: bit_vector(31 downto 0);
signal count: integer range 0 to N;

```

```

begin
  CLK <= not CLK after 10 ns;
process
begin
  for i in 1 to N loop
    St <= '1';
    Dbus <= dividendarr(i)(31 downto 16);
    wait until rising_edge(CLK);
    Dbus <= dividendarr(i)(15 downto 0);
    wait until rising_edge(CLK);
    Dbus <= divisorarr(i);
    St <= '0';
    dividend <= dividendarr(i)(31 downto 0);-- save dividend for
                                                listing
    divisor <= divisorarr(i);                  -- save divisor for
                                                listing
    wait until (Rdy = '1');                     -- save index for triggering
    count <= i;
  end loop;
end process;
sdiv1: sdiv port map(Clk, St, Dbus, Quotient, V, Rdy);
end test1;

```

Figure 4-25 shows the simulator command file and output. The `-NOtrigger`, together with the `-Trigger count` in the list statement, causes the output to be displayed only when the count signal changes. Examination of the simulator output shows that the divider operation is correct for all of the test cases, except for the following case:

$$C0008000h \div 7FFFh = -3FFF8000 \div 7FFFh = -8000h = 8000h$$

In this case, the overflow is turned on, and division never occurs. In general, the divider will indicate an overflow whenever the quotient should be `8000h` (the most negative value). This occurs because the divider basically divides positive numbers, and the largest positive quotient is `7FFFh`. If it is important to be able to generate the quotient `8000h`, the overflow detection can be modified so it does not generate an overflow in this special case.

**Figure 4-25 Simulation Test Results for Signed Divider**

```
-- Command file to test results of signed multiplier
list -hex -NOtrigger dividend divisor Quotient V -Trigger count
run 5300
```

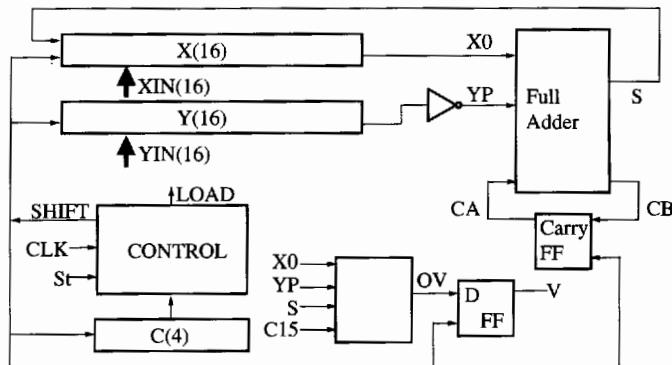
ns	delta	dividend	divisor	quotient	v	count
0	+0	00000000	0000	0000 0	0	0
470	+3	0000006F	0007	000F 0	1	
910	+3	07FF00BB	E005	BFFE 0	2	
1330	+3	FFFFFE08	001E	FFF0 0	3	
1910	+3	FF80030A	EFFA	07FC 0	4	
2010	+3	3FFF8000	7FFF	0000 1	5	
2710	+3	3FFF7FFF	7FFF	7FFF 0	6	
2810	+3	C0008000	7FFF	0000 1	7	
3510	+3	C0008000	8000	7FFF 0	8	
4210	+3	C0008001	7FFF	8001 0	9	
4610	+3	00000000	0001	0000 0	A	
5010	+3	FFFFFFF	7FFF	0000 0	B	
5110	+3	FFFFFFF	0000	0002 1	C	

In this chapter, we described algorithms for multiplication and division of unsigned and signed binary numbers. We designed digital systems to implement these algorithms. After developing a block diagram for such a system and defining the required control signals, we used a state graph to define a sequential machine that generates control signals in the proper sequence. We used VHDL to describe the systems at several different levels so that we can simulate and test for correct operation of the systems we have designed.

## Problems

**4.1** A block diagram for a 16-bit 2's complement serial subtracter is given here. When  $S_t = 1$ , the registers are loaded and then subtraction occurs. The shift counter,  $C$ , produces a signal  $C15 = 1$  after 15 shifts.  $V$  should be set to 1 if an overflow occurs. Set the carry flip-flop to 1 during load in order to form the 2's complement. Assume that  $S_t$  remains 1 for one clock time.

- (a) Draw a state diagram for the control (two states).
- (b) Write VHDL code for the system. Use two processes. The first process should determine the next state and control signals; the second process should update the registers on the rising edge of the clock.



**4.2** Initially a 3-digit BCD number is placed in the *A* register. When an *St* signal is received, conversion to binary takes place, and the resulting binary number is stored in the *B* register. At each step of the conversion, the entire BCD number (along with the binary number) is shifted one place to the right. If the result in a given decade is greater than or equal 1000, the correction network subtracts 0011 from that decade. (If the result is less than 1000, the correction network leaves the contents of the decade unchanged.) A shift counter is provided to count the number of shifts. When conversion is complete, the maximum value of *B* will be 999 (in binary). *Note:* *B* is 10 bits.

- (a) Draw the block diagram of the BCD-to-binary converter.
- (b) Draw a state diagram of the control network (3 states). Use the following control signals: *St*, start conversion; *Sh*, shift right; *Co*, subtract correction if necessary; and *C9*, counter is in state 9, or *C10*, counter is in state 10. (Use either *C9* or *C10* but not both.)
- (c) Write a VHDL description of the system.

**4.3** The block diagram for a multiplier for signed (2's complement) binary numbers is shown in Figure 4-10. Give the contents of the *A* and *B* registers after each clock pulse when multiplicand =  $-1/8$  and multiplier =  $-3/8$ .

#### 4.4

(a) Draw the block diagram for a 32-bit serial adder with accumulator. The control network uses a 5-bit counter, which outputs a signal *K* = 1 when it is in state 11111. When a start signal (*N*) is received, the registers should be loaded. Assume that *N* will remain 1 until the addition is complete. When the addition is complete, the control network should go to a stop state and remain there until *N* is changed back to 0. Draw a state diagram for the control network (excluding the counter).

- (b) Write the VHDL for the complete system, and verify its correct operation.

#### 4.5

(a) Draw the block diagram for a divider for unsigned binary numbers that divides an 8-bit dividend by a 3-bit divisor to give a 5-bit quotient.

(b) Draw a state graph for the control circuit. Assume that the start signal (*St*) is present for one clock period.

- (c) Write a high-level VHDL description of the divider.

**4.6** In Section 4.4 we developed an algorithm for multiplying signed binary fractions, with negative fractions represented in 2's complement.

- (a) Illustrate this algorithm by multiplying 1.0111 by 1.101.
- (b) Draw a block diagram of the hardware necessary to implement this algorithm for the case where the multiplier is 4 bits, including sign, and the multiplicand is 5 bits, including sign.

#### 4.7

(a) Write a VHDL module that describes one bit of a full adder with accumulator. The module should have two control inputs, *Ad* and *L*. If *Ad* = 1, the *Y* input (and carry input) are added to the accumulator. If *L* = 1, the *Y* input is loaded into the accumulator.

(b) Using the module defined in (a), write a VHDL description of a 4-bit subtracter with accumulator. Assume negative numbers are represented in 1's complement. The subtracter should have control inputs *Su* (subtract) and *Ld* (load).

**4.8**

- (a) Show a logic diagram for a 16-bit serial multiplier. Use three 16-bit shift registers, two 74163 counters, a full adder, a D flip-flop, and appropriate gates. *Note:* Be sure to account for the fact that the shift registers and 74163 change state on the rising edge of the clock.
- (b) Draw a state graph for the control network.
- (c) Realize the control network using a PLA and 2 D flip-flops.
- (d) Give the PLA table.
- (e) Write a VHDL description for the multiplier using a PLA model.

**4.9** Write a VHDL description of the serial multiplier in Problem 4.8. Define all the control signals explicitly. Do not use a case statement or IF clauses outside the modules. Specify the system in terms of the modules and their connections as well as statements describing the adder, PLA and flip-flop operations, etc.

**4.10** This problem concerns the design of a divider for unsigned binary numbers that will divide a 16-bit dividend by an 8-bit divisor to give an 8-bit quotient. Assume that the start signal ( $ST = 1$ ) is 1 for exactly one clock time. If the quotient would require more than 8 bits, the divider should stop immediately and output  $V = 1$  to indicate an overflow. Use a 17-bit dividend register and store the quotient in the lower 8 bits of this register. Use a 4-bit counter to count the number of shifts, together with a subtract-shift controller.

- (a) Draw a block diagram of the divider.
- (b) Draw a state graph for the subtract-shift controller (3 states).
- (c) Write a VHDL description of the divider.

**4.11** This problem concerns the design of a binary divider that will divide an 8-bit number by a 4-bit number to give a 4-bit quotient. All numbers are positive numbers without a sign bit. In order to speed up the operation of the divider, the circuit is configured so that shifting and subtracting can be completed in one clock time (instead of two clocks).

- (a) Draw a block diagram that includes an 8-bit register, a 5-bit subtracter, and other necessary components. Be careful to show which subtracter outputs connect to which register inputs. An overflow indication is *not* required.
- (b) Define the required control signals and draw a state diagram for the controller. Assume that the start signal is present for only one clock and the divider stops in a done state.

**4.12** This problem involves the design of a network that finds the square root of an 8-bit unsigned binary number  $N$  using the method of subtracting out odd integers. To find the square root of  $N$ , we subtract 1, then 3, then 5, etc., until we can no longer subtract without the result going negative. The number of times we subtract is equal to the square root of  $N$ . For example, to find  $\sqrt{27}$ :  $27 - 1 = 26$ ;  $26 - 3 = 23$ ;  $23 - 5 = 18$ ;  $18 - 7 = 11$ ;  $11 - 9 = 2$ ;  $2 - 11$  (can't subtract). Since we subtracted 5 times,  $\sqrt{27} = 5$ . Note that the final odd integer is  $11_{10} = 1011_2$ , and this consists of the square root ( $101_2 = 5_{10}$ ) followed by a 1.

(a) Draw a block diagram of the square rooter that includes a register to hold  $N$ , a subtracter, a register to hold the odd integers, and a control network. Indicate where to read the final square root. Define the control signals used on the diagram.

(b) Draw a state graph for the control network using a minimum number of states. The  $N$  register should be loaded when  $St = 1$ . When the square root is complete, the control network should output a done signal and wait until  $St = 0$  before resetting.

**4.13** Design a multiplier that will multiply two 16-bit signed binary integers to give a 32-bit product. Negative numbers should be represented in 2's complement form. Use the following method: First complement the multiplier and multiplicand if they are negative, multiply the positive numbers, and then complement the product if necessary. Design the multiplier so that after the registers are loaded, the multiplication can be completed in 16 clocks.

(a) Draw a block diagram of the multiplier. Use a 4-bit counter to count the number of shifts. (The counter will output a signal  $K = 1$  when it is in state 15.) Define all condition and control signals used on your diagram.

(b) Draw a state diagram for the multiplier control using a minimum number of states (3 states). When the multiplication is complete, the control network should output a done signal and then wait for  $ST = 0$  before returning to state  $S0$ .

(c) Write a VHDL behavioral description of the multiplier without using control signals (for example, see Figure 4-5) and test it.

(d) Write a VHDL behavioral description using control signals (for example, see Figure 4-12) and test it.

**4.14** Implement the block diagram for the multiplier of Figure 4-10 using 22V10 PALs and no other logic. Is it possible to partition the block diagram so that only two 22V10s are required? If not, use three 22V10s. Show the connections to the PALs and give the logic equations for the  $D$  input to each macrocell. Specify the required fuse pattern for typical macrocells.

**4.15** The objective of this problem is to use VHDL to describe and simulate a multiplier for signed binary numbers using Booth's algorithm. Negative numbers should be represented by their 2's complement. Booth's algorithm works as follows, assuming each number is  $n$  bits including sign: Use an  $(n + 1)$ -bit register for the accumulator ( $A$ ) so the sign bit will not be lost if an overflow occurs. Also, use an  $(n + 1)$ -bit register ( $B$ ) to hold the multiplier and an  $n$ -bit register ( $C$ ) to hold the multiplicand.

1. Clear  $A$  (the accumulator), load the multiplier into the upper  $n$  bits of  $B$ , clear  $B_0$ , and load the multiplicand into  $C$ .
2. Test the lower two bits of  $B$  ( $B_1B_0$ ).
  - If  $B_1B_0 = 01$ , then add  $C$  to  $A$  ( $C$  should be sign-extended to  $n + 1$  bits and added to  $A$  using an  $(n + 1)$ -bit adder).
  - If  $B_1B_0 = 10$ , then add the 2's complement of  $C$  to  $A$ .
  - If  $B_1B_0 = 00$  or  $11$ , skip this step.
3. Shift  $A$  and  $B$  together right one place with sign extended.
4. Repeat steps 2 and 3,  $n - 1$  more times.
5. The product will be in  $A$  and  $B$ , except ignore  $B_0$ .

Example for  $n = 5$ : Multiply  $-9$  by  $-13$ .

	<i>A</i>	<i>B</i>	$B_1B_0$	
1. Load registers.	000000	100110	10	$C = 10111$
2. Add 2's comp. of $C$ to $A$ .	<u>001001</u>	001001	100110	
3. Shift $A \& B$ .	000100	110011	11	
3. Shift $A \& B$ .	000010	011001	01	
2. Add $C$ to $A$ .	<u>110111</u>	111001	011001	
3. Shift $A \& B$ .	111100	101100	00	
3. Shift $A \& B$ .	111110	010110	10	
2. Add 2's comp. of $C$ to $A$ .	<u>001001</u>	000111	010110	
3. Shift $A \& B$ .	000011	101011		

Final result:  $0001110101 = +117$

- (a) Draw a block diagram of the system for  $n = 8$ . Use 9-bit registers for  $A$  and  $B$ , a 9-bit full adder, an 8-bit complementer, a 3-bit counter, and a control network. Use the counter to count the number of shifts.
- (b) Draw a state graph for the control network. When the counter is in state 111, return to the start state at the time the last shift occurs (3 states should be sufficient).
- (c) Write behavioral VHDL code for the multiplier (do not explicitly use output control signals in your VHDL code). Use the procedure *addvec* (in *BITLIB*), which will add two  $n$ -bit vectors and a carry to produce an  $n$ -bit sum and a carryout.
- (d) Simulate your VHDL design using the following test cases (in each pair, the second number is the multiplier):

$$01100110 \times 00110011$$

$$10100110 \times 01100110$$

$$01101011 \times 10001110$$

$$11001100 \times 10011001$$

Verify that your results are correct.

- 4.16** This problem involves the design of a parallel adder-subtractor for 8-bit numbers expressed in *sign and magnitude* notation. The inputs  $X$  and  $Y$  are in sign and magnitude, and the output  $Z$  must be in sign and magnitude. Internal computation may be done in either 2's complement or 1's complement (specify which you use), but no credit will be given if you assume the inputs  $X$  and  $Y$  are in 1's or 2's complement. If the input signal  $Sub = 1$ , then  $Z = X - Y$ , else  $Z = X + Y$ . Your network must work for all combinations of positive and negative inputs for both add and subtract. You may use only the following components: an 8-bit adder, a 1's complementer (for the input  $Y$ ), a second complementer (which may be either 1's complement or 2's complement—specify which you use), and a combinational logic network to generate control signals. Hint:  $-X + Y = -(X - Y)$ . Also generate an overflow signal that is 1 if the result cannot be represented in 8-bit sign and magnitude.

- (a) Draw the block diagram. No registers, multiplexers, or tristate busses are allowed.
  - (b) Give a truth table for the logic network that generates the necessary control signals. Inputs for the table should be *Sub*, *X<sub>s</sub>*, and *Y<sub>s</sub>* in that order, where *X<sub>s</sub>* is the sign of *X* and *Y<sub>s</sub>* is the sign of *Y*.
  - (c) Explain how you would determine the overflow and give an appropriate equation.
- 4.17** Using the full adder developed in Chapter 2, write a VHDL data flow description of the  $4 \times 4$  array multiplier in Figure 4-7.
- 4.18** Consider a  $2 \times 2$  array multiplier with 2-bit multiplicand and multiplier with a 4-bit result.
- (a) Draw the block diagram of the multiplier.
  - (b) How many AND gates, full adders, and half-adders did you use?
  - (c) If the maximum delay in a full adder is 15 ns, and the delay of an AND gate is 10 ns, what is the worst-case time to complete the multiplication?
  - (d) How fast does the clock of a 2-bit serial-parallel multiplier similar to Figure 4-3 have to be in order to be as fast as the answer in part (c)?

# CHAPTER 5

## DIGITAL DESIGN WITH SM CHARTS

A state machine is often used to control a digital system that carries out a step-by-step procedure or algorithm. The state graphs in Figures 4-2, 4-4, 4-6, 4-9, 4-11, 4-20, and 4-22 define state machines for controlling adders, multipliers, and dividers. As an alternative to using state graphs, a special type of flowchart, called a *state machine flowchart*, or *SM chart*, may be used to describe the behavior of a state machine. SM charts are often used to design control units for digital systems.

In this chapter we first describe the properties of SM charts and how they are used in the design of state machines. Then we show examples of SM charts for a multiplier and a dice game controller. We construct VHDL descriptions of these systems from the SM charts, and we simulate the VHDL code to verify correct operation. We then proceed with the design and show how PLA tables and logic equations can be derived from SM charts. Finally, we show how alternative designs can be obtained by transforming the SM charts.

### 5.1 STATE MACHINE CHARTS

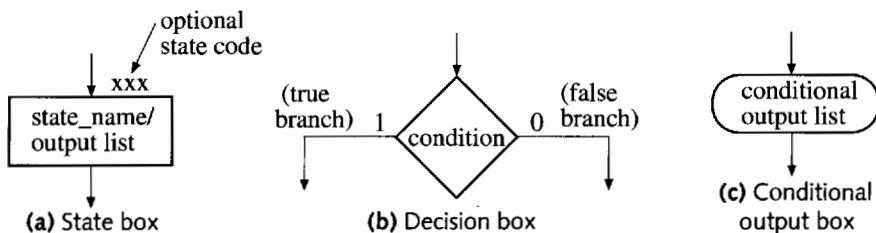
Just as flowcharts are useful in software design, flowcharts are useful in the hardware design of digital systems. In this section we introduce the SM chart, which is also called an *ASM (algorithmic state machine) chart*. We will see that the SM chart offers several advantages over state graphs. It is often easier to understand the operation of a digital system by inspection of the SM chart instead of the equivalent state graph. The conditions for a proper state graph (see Section 4.2) are automatically satisfied for an SM chart. A given SM chart can be converted into several equivalent forms, and each form leads directly to a hardware realization.

An SM chart differs from an ordinary flowchart in that certain specific rules must be followed in constructing the SM chart. When these rules are followed, the SM chart is equivalent to a state graph, and it leads directly to a hardware realization. Figure 5-1 shows the three principal components of an SM chart. The state of the system is represented by a *state box*. The state box contains a *state name*, followed by a slash (/) and an optional *output list*. After a state assignment has been made, a *state code* may be placed outside the box at the top. A *decision box* is represented by a diamond-shaped symbol with true and false branches. The *condition* placed in the box is a Boolean expression that is evaluated to determine which branch to take. The *conditional output box*, which has curved ends, contains

a *conditional output list*. The conditional outputs depend on both the state of the system and the inputs.

An SM chart is constructed from *SM blocks*. Each SM block (Figure 5-2) contains exactly one state box, together with the decision boxes and conditional output boxes associated with that state. An SM block has one *entrance path* and one or more *exit paths*. Each SM block describes the machine operation during the time that the machine is in one state. When a digital system enters the state associated with a given SM block, the outputs

**Figure 5-1 Components of an SM Chart**



on the output list in the state box become true. The conditions in the decision boxes are evaluated to determine which path (or paths) are followed through the SM block. When a conditional output box is encountered along such a path, the corresponding conditional outputs become true. If an output is not encountered along a path, that output is false by default. A path through an SM block from entrance to exit is referred to as a *link path*.

For the example of Figure 5-2, when state  $S_1$  is entered, outputs  $Z_1$  and  $Z_2$  become 1. If input  $X_1 = 0$ ,  $Z_3$  and  $Z_4$  also become 1. If  $X_1 = X_2 = 0$ , at the end of the state time the machine goes to the next state via exit path 1. On the other hand, if  $X_1 = 1$  and  $X_3 = 0$ , the output  $Z_5$  is 1, and exit to the next state will occur via exit path 3. Since  $Z_3$  and  $Z_4$  are not encountered along this link path,  $Z_3 = Z_4 = 0$  by default.

A given SM block can generally be drawn in several different forms. Figure 5-3 shows two equivalent SM blocks. In both (a) and (b), the output  $Z_2 = 1$  if  $X_1 = 0$ ; the next state is  $S_2$  if  $X_2 = 0$  and  $S_3$  if  $X_2 = 1$ . As illustrated in this example, the order in which the inputs are tested may affect the complexity of the SM chart.

The SM charts of Figure 5-4(a) and (b) each represent a combinational network, since there is only one state and no state change occurs. The output is  $Z_1 = 1$  if  $A + BC = 1$ ; otherwise  $Z_1 = 0$ . Figure 5-4(b) shows an equivalent SM chart in which the input variables are tested individually. The output is  $Z_1 = 1$  if  $A = 1$  or if  $A = 0, B = 1$ , and  $C = 1$ . Hence,

$$Z_1 = A + A'BC = A + BC$$

which is the same output function realized by the SM chart of Figure 5-4(a).

Certain rules must be followed when constructing an SM block. First, for every valid combination of input variables, there must be exactly one exit path defined. This is necessary since each allowable input combination must lead to a single next state. Second, no internal feedback within an SM block is allowed. Figure 5-5 shows incorrect and correct ways of drawing an SM block with feedback.

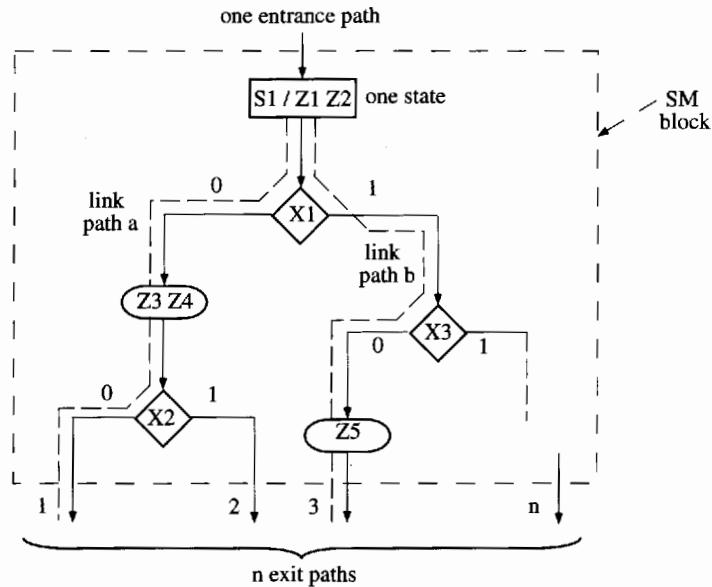
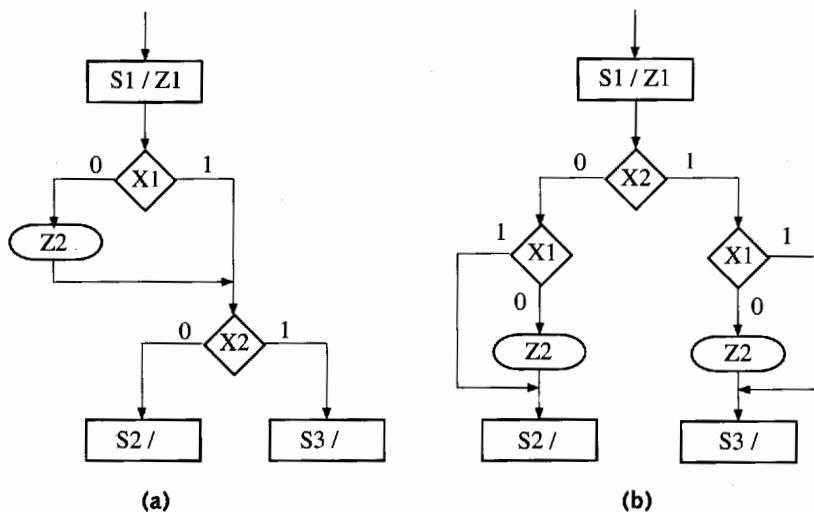
**Figure 5-2 Example of an SM Block****Figure 5-3 Equivalent SM Blocks**

Figure 5-4 Equivalent SM Charts for a Combinational Network

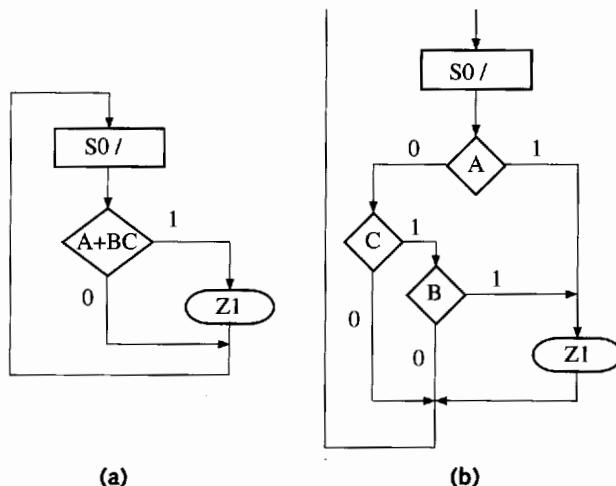
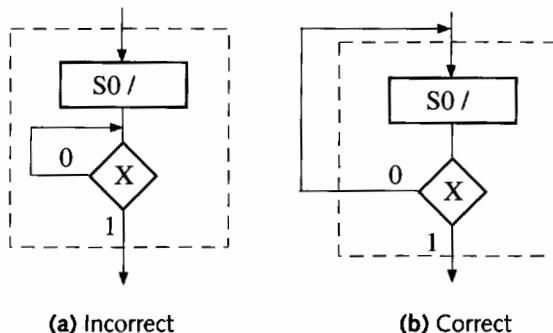
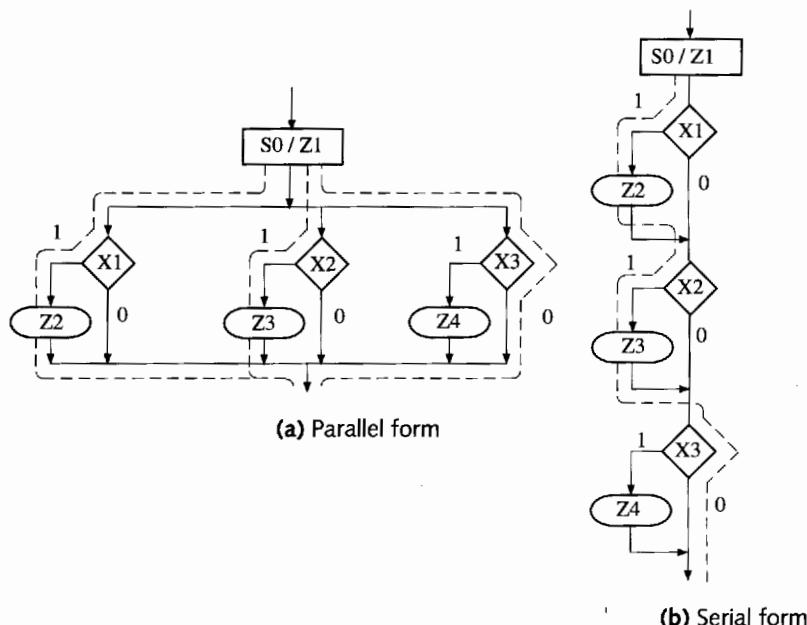


Figure 5-5 SM Block with Feedback



As shown in Figure 5-6(a), an SM block can have several parallel paths that lead to the same exit path, and more than one of these paths can be active at the same time. For example, if  $X_1 = X_2 = 1$  and  $X_3 = 0$ , the link paths marked with dashed lines are active, and the outputs  $Z_1$ ,  $Z_2$ , and  $Z_3$  are 1. Although Figure 5-6(a) would not be a valid flowchart for a program for a serial computer, it presents no problems for a state machine implementation. The state machine can have a multiple-output network that generates  $Z_1$ ,  $Z_2$ , and  $Z_3$  at the same time. Figure 5-6(b) shows a serial SM block, which is equivalent to Figure 5-6(a). In the serial block only one active link path between entrance and exit is possible. For any combination of input values, the outputs will be the same as in the equivalent parallel form. The link path for  $X_1 = X_2 = 1$  and  $X_3 = 0$  is shown with a dashed line, and the outputs encountered on this path are  $Z_1$ ,  $Z_2$ , and  $Z_3$ . Regardless of whether the SM block is drawn in serial or parallel form, all the tests take place within one clock time. In the rest of this text, we use only the serial form for SM charts.

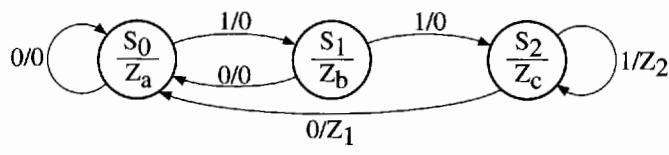
Figure 5-6 Equivalent SM Blocks



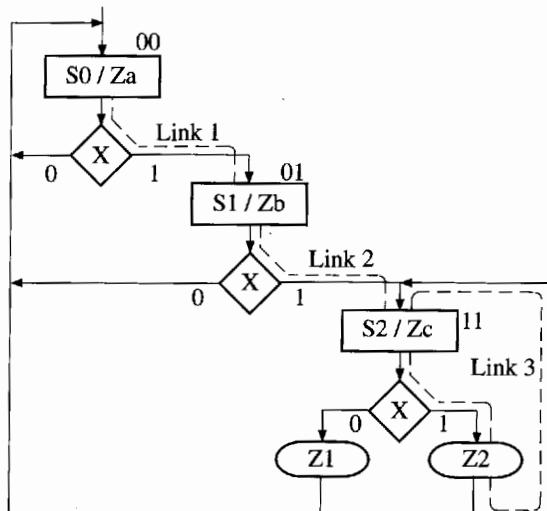
It is easy to convert a state graph for a sequential machine to an equivalent SM chart. The state graph of Figure 5-7(a) has both Moore and Mealy outputs. The equivalent SM chart has three blocks—one for each state. The Moore outputs ( $Z_a, Z_b, Z_c$ ) are placed in the state boxes, since they do not depend on the input. The Mealy outputs ( $Z_1, Z_2$ ) appear in conditional output boxes, since they depend on both the state and input. In this example, each SM block has only one decision box, since only one input variable must be tested. For both the state graph and SM chart,  $Z_c$  is always 1 in state  $S_2$ . If  $X = 0$  in state  $S_2$ ,  $Z_1 = 1$  and the next state is  $S_0$ . If  $X = 1$ ,  $Z_2 = 1$  and the next state is  $S_2$ . We have added a state assignment ( $S_0 = 00, S_1 = 01, S_2 = 10$ ) next to the state boxes.

Figure 5-8 shows a timing chart for the SM chart of Figure 5-7 with an input sequence  $X = 1, 1, 1, 0, 0, 0$ . In this example, all state changes occur immediately after the rising edge of the clock. Since the Moore outputs ( $Z_a, Z_b, Z_c$ ) depend on the state, they can change only immediately following a state change. The Mealy outputs ( $Z_1, Z_2$ ) can change immediately after a state change or an input change. In any case, all outputs will have their correct values at the time of the active clock edge.

Figure 5-7 Conversion of a State Graph to an SM Chart

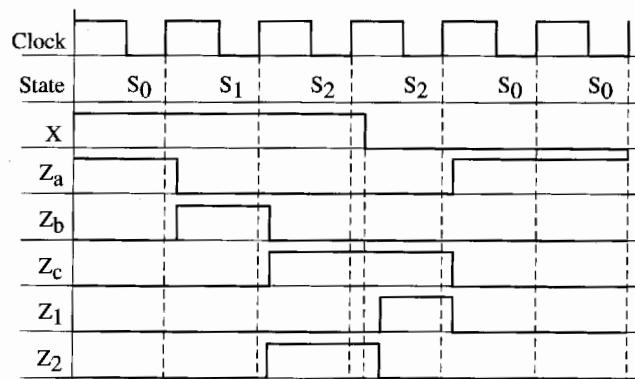


(a) State graph



(b) Equivalent SM chart

Figure 5-8 Timing Chart for Figure 5-7

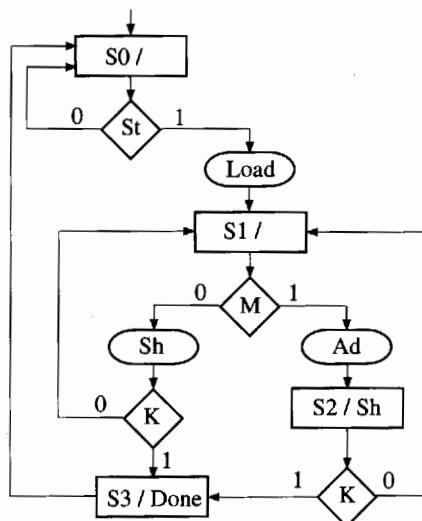


## 5.2 DERIVATION OF SM CHARTS

The method used to derive an SM chart for a sequential control network is similar to that used to derive the state graph. First, we should draw a block diagram of the system we are controlling. Next we should define the required input and output signals to the control network. Then we can construct an SM chart that tests the input signals and generates the proper sequence of output signals.

In this section we give two examples of SM charts. The first example is an SM chart for control of the binary multiplier shown in Figures 4-3 and 4-6(a). The add-shift control generates the required sequence of add and shift signals. The counter counts the number of shifts and outputs  $K = 1$  just before the last shift occurs. The SM chart for the multiplier control (Figure 5-9) corresponds closely to the state graph of Figure 4-6(c). In state  $S_0$ , when the start signal  $S_t$  is 1, the registers are loaded. In  $S_1$ , the multiplier bit  $M$  is tested. If  $M = 1$ , an add signal is generated and the next state is  $S_2$ . If  $M = 0$ , a shift signal is generated and  $K$  is tested. If  $K = 1$ , this will be the last shift and the next state is  $S_3$ . In  $S_2$ , a shift signal is generated, since a shift must always follow an add. If  $K = 1$ , the network goes to  $S_3$  at the time of the last shift; otherwise, the next state is  $S_1$ . In  $S_3$ , the done signal is turned on.

**Figure 5-9** SM Chart for Binary Multiplier



Conversion of an SM chart to a VHDL process is straightforward. A case statement can be used to specify what happens in each state. Each condition box corresponds directly to an if statement (or an elsif). Figure 5-10 shows the VHDL code for the SM chart in Figure 5-9. Two processes are used. The first process represents the combinational part of the network, and the second process updates the state register on the rising edge of the clock. The signals *Load*, *Sh*, and *Ad* are turned on in the appropriate states, and they must be turned off when the state changes. A convenient way to do this is to set them all to 0 at the start of the process.

**Figure 5-10** VHDL for SM Chart of Figure 5-9

```

entity Mult is
  port(CLK,St,K,M: in bit;
       Load,Sh,Ad,Done: out bit);
end Mult;

architecture SMbehave of Mult is
  signal State, Nextstate: integer range 0 to 3;
begin
  process(St, K, M, State)           -- start if state or inputs change
  begin
    Load <= '0'; Sh <= '0'; Ad <= '0';
    case State is
      when 0 => if St = '1' then          -- St (state 0)
                  Load <= '1';
                  Nextstate <= 1;
                else Nextstate <= 0;   -- St'
                  end if;
      when 1 => if M = '1' then          -- M (state 1)
                  Ad <= '1';
                  Nextstate <= 2;
                else                         -- M'
                  Sh <= '1';
                  if K = '1' then Nextstate <= 3;   -- K
                  else Nextstate <= 1;           -- K'
                  end if;
                end if;
      when 2 => Sh <= '1';             -- (state 2)
                  if K = '1' then Nextstate <= 3;   -- K
                  else Nextstate <= 1;           -- K'
                  end if;
      when 3 => Done <= '1';           -- (state 3)
                  Nextstate <= 0;
    end case;
  end process;
  process(CLK)
  begin
    if CLK = '1' then
      State <= Nextstate;           -- update state on rising edge
    end if;
  end process;
end SMbehave;

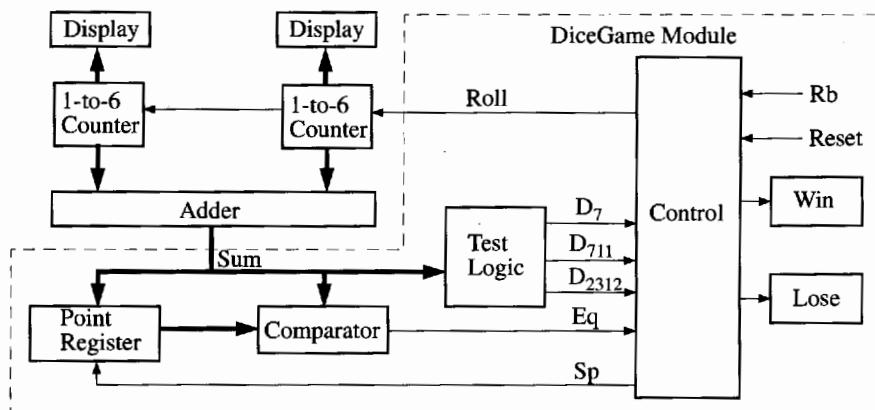
```

As a second example of SM chart construction, we will design an electronic dice game. Figure 5-11 shows the block diagram for the dice game. Two counters are used to simulate the roll of the dice. Each counter counts in the sequence 1, 2, 3, 4, 5, 6, 1, 2, ....

Thus, after the “roll” of the dice, the sum of the values in the two counters will be in the range 2 through 12. The rules of the game are as follows:

1. After the first roll of the dice, the player wins if the sum is 7 or 11. The player loses if the sum is 2, 3, or 12. Otherwise, the sum the player obtained on the first roll is referred to as a point, and he or she must roll the dice again.
2. On the second or subsequent roll of the dice, the player wins if the sum equals the point, and he or she loses if the sum is 7. Otherwise, the player must roll again until he or she finally wins or loses.

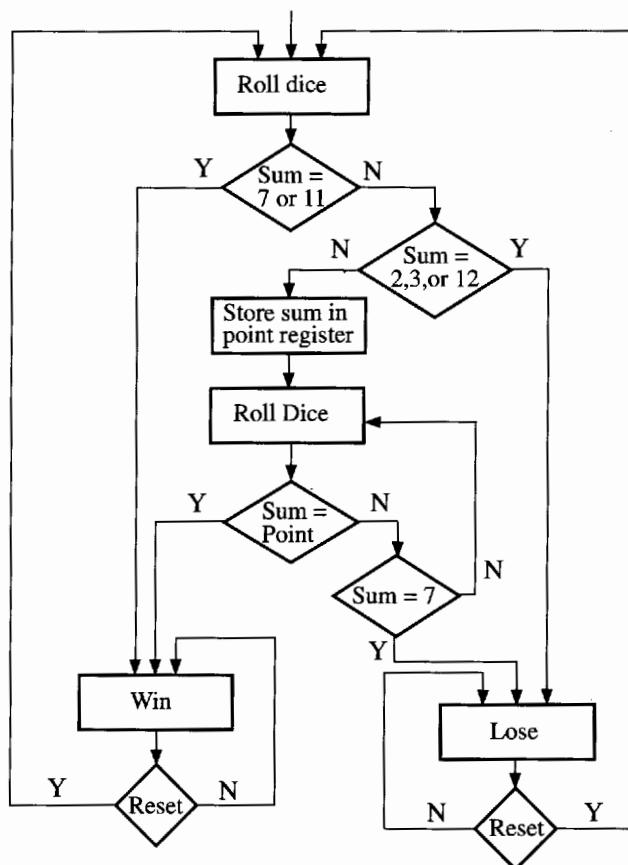
**Figure 5-11 Block Diagram for Dice Game**



The inputs to the dice game come from two push buttons, Rb (roll button) and Reset. Reset is used to initiate a new game. When the roll button is pushed, the dice counters count at a high speed, so the values cannot be read on the display. When the roll button is released, the values in the two counters are displayed, and the game can proceed. If the Win light or Lose light is not on, the player must push the roll button again.

Figure 5-12 shows a flowchart for the dice game. After rolling the dice, the sum is tested. If it is 7 or 11, the player wins; if it is 2, 3, or 12, he or she loses. Otherwise the sum is saved in the point register, and the player rolls again. If the new sum equals the point, the player wins; if it is 7, he or she loses. Otherwise, the player rolls again. After winning or losing, he or she must push Reset to begin a new game. We will assume at this point that the push buttons are properly debounced and that changes in Rb are properly synchronized with the clock. A method for debouncing and synchronization was discussed in Section 3.5.

Figure 5-12 Flowchart for Dice Game



The components for the dice game shown in the block diagram (Figure 5-11) include an adder, which adds the two counter outputs, a register to store the point, test logic to determine conditions for win or lose, and a control network. Input signals to the control network are defined as follows:

$$D_7 = 1 \text{ if the sum of the dice is } 7$$

$$D_{711} = 1 \text{ if the sum of the dice is } 7 \text{ or } 11$$

$$D_{2312} = 1 \text{ if the sum of the dice is } 2, 3, \text{ or } 12$$

$$Eq = 1 \text{ if the sum of the dice equals the number stored in the point register}$$

$$Rb = 1 \text{ when the roll button is pressed}$$

$$Reset = 1 \text{ when the reset button is pressed}$$

Outputs from the control network are defined as follows:

*Roll* = 1 enables the dice counters

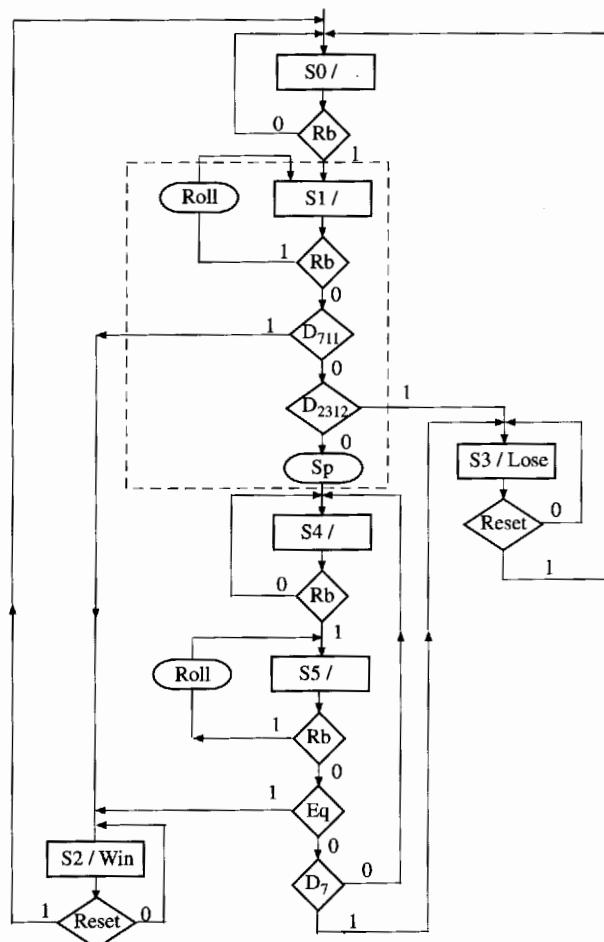
*Sp* = 1 causes the sum to be stored in the point register

*Win* = 1 turns on the win light

*Lose* = 1 turns on the lose light

We now convert the flowchart for the dice game to an SM chart for the control network using the control signals defined above. Figure 5-13 shows the resulting SM chart.

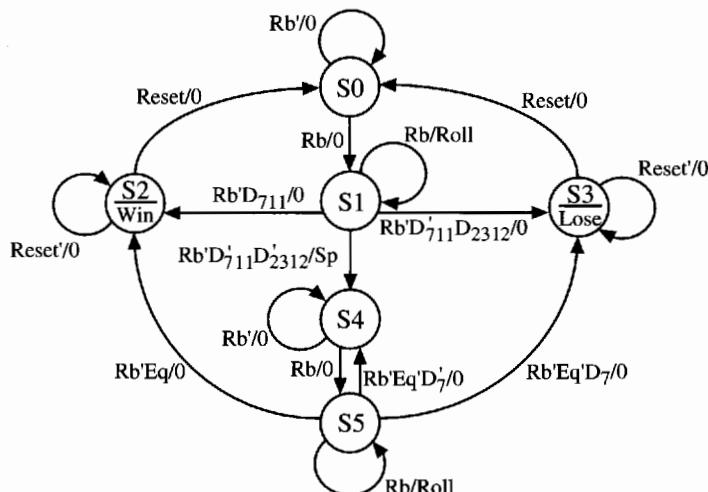
Figure 5-13 SM Chart for Dice Game



The control network waits in state  $S_0$  until the roll button is pressed ( $Rb = 1$ ). Then it goes to state  $S_1$ , and the roll counters are enabled as long as  $Rb = 1$ . As soon as the roll button is released ( $Rb = 0$ ),  $D_{711}$  is tested. If the sum is 7 or 11, the network goes to state  $S_2$  and turns on the Win light; otherwise,  $D_{2312}$  is tested. If the sum is 2, 3, or 12, the network goes to state  $S_3$  and turns on the Lose light; otherwise, the signal  $Sp$  becomes 1 and the sum is stored in the point register. It then enters  $S_4$  and waits for the player to “roll the dice” again. In  $S_5$ , after the roll button is released, if  $Eq = 1$ , the sum equals the point and state  $S_2$  is entered to indicate a win. If  $D_7 = 1$ , the sum is 7 and  $S_3$  is entered to indicate a loss. Otherwise, control returns to  $S_4$  so that the player can roll again. When in  $S_2$  or  $S_3$ , the game is reset to  $S_0$  when the Reset button is pressed.

Instead of using an SM chart, we could construct an equivalent state graph from the flowchart. Figure 5-14 shows a state graph for the dice game controller. The state graph has the same states, inputs, and outputs as the SM chart. The arcs have been labeled consistently with the rules for proper state graphs given in Section 4.2. Thus, the arcs leaving state  $S_1$  are labeled  $Rb$ ,  $Rb'D_{711}$ ,  $Rb'D_{711}D_{2312}$ , and  $Rb'D_{711}D_{2312}'$ .

**Figure 5-14 State Graph for Dice Game Controller**



Before proceeding with the design, it is important to verify that the SM chart (or state graph) is correct. We will write a behavioral VHDL description based on the SM chart and then write a test bench to simulate the roll of the dice. Initially we will write a dice game module that contains the control network, point register, and comparator (see Figure 5-11). Later, we will add the counters and adder so that we can simulate the complete dice game.

The VHDL code for the dice game in Figure 5-15 corresponds directly to the SM chart of Figure 5-13. The case statement in the first process tests the state, and in each state nested if-then-else (or elsif) statements are used to implement the conditional tests. In State 1 the *Roll* signal is turned on when *Rb* is 1. If all conditions test false, *Sp* is set to 1 and the next state is 4. In the second process, the state is updated after the rising edge of the clock, and if *Sp* is 1, the sum is stored in the point register.

**Figure 5-15 Behavioral Model for Dice Game**

```
entity DiceGame is
  port (Rb, Reset, CLK: in bit;
        Sum: in integer range 2 to 12;
        Roll, Win, Lose: out bit);
end DiceGame;

library BITLIB;
use BITLIB.bit_pack.all;

architecture DiceBehave of DiceGame is
  signal State, Nextstate: integer range 0 to 5;
  signal Point: integer range 2 to 12;
  signal Sp: bit;
begin
  process(Rb, Reset, Sum, State)
  begin
    Sp <= '0'; Roll <= '0'; Win <= '0'; Lose <= '0';
    case State is
      when 0 => if Rb = '1' then Nextstate <= 1; end if;
      when 1 =>
        if Rb = '1' then Roll <= '1';
        elsif Sum = 7 or Sum = 11 then Nextstate <= 2;
        elsif Sum = 2 or Sum = 3 or Sum = 12 then Nextstate <= 3;
        else Sp <= '1'; Nextstate <= 4;
        end if;
      when 2 => Win <= '1';
        if Reset = '1' then Nextstate <= 0; end if;
      when 3 => Lose <= '1';
        if Reset = '1' then Nextstate <= 0; end if;
      when 4 => if Rb = '1' then Nextstate <= 5; end if;
      when 5 =>
        if Rb = '1' then Roll <= '1';
        elsif Sum = Point then Nextstate <= 2;
        elsif Sum = 7 then Nextstate <= 3;
        else Nextstate <= 4;
        end if;
    end case;
  end process;
  process(CLK)
  begin
    if rising_edge(CLK) then
      State <= Nextstate;
      if Sp = '1' then Point <= Sum; end if;
    end if;
  end process;
end DiceBehave;
```

We are now ready to test the behavioral model of the dice game. It is not convenient to include the counters that generate random numbers in the initial test, since we want to specify a sequence of dice rolls that will test all paths on the SM chart. We could prepare a simulator command file that would generate a sequence of data for *Rb*, *Sum*, and *Reset*. This would require careful analysis of the timing to make sure that the input signals change at the proper time. A better approach for testing the dice game is to design a VHDL test bench module to monitor the output signals from the dice game module and supply a sequence of inputs in response.

Figure 5-16 shows the DiceGame connected to a module called GameTest. GameTest needs to perform the following functions:

1. Initially supply the *Rb* signal.
2. When the DiceGame responds with a *Roll* signal, supply a *Sum* signal, which represents the sum of the two dice.
3. If no *Win* or *Lose* signal is generated by the DiceGame, repeat steps 1 and 2 to roll again.
4. When a *Win* or *Lose* signal is detected, generate a *Reset* signal and start again.

**Figure 5-16** Dice Game with Test Bench

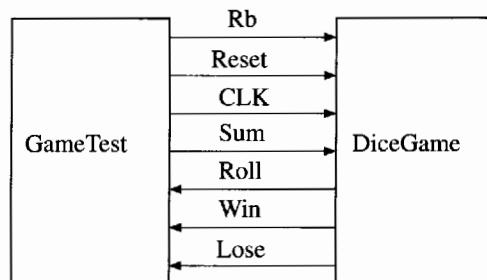
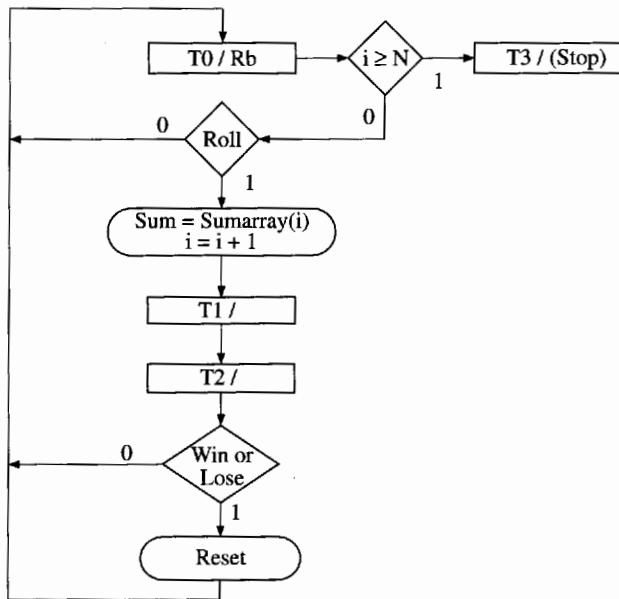


Figure 5-17 shows an SM chart for the GameTest module. *Rb* is generated in state T0. When DiceGame detects *Rb*, it goes to S1 and generates *Roll*. When GameTest detects *Roll*, the *Sum* that represents the next roll of the dice is read from *Sumarray(i)* and *i* is incremented. When the state goes to T1, *Rb* goes to 0. The DiceGame goes to S2, S3, or S4 and GameTest goes to T2. The *Win* and *Lose* outputs are tested in state T2. If *Win* or *Lose* is detected, a *Reset* signal is generated before the next roll of the dice. After *N* rolls of the dice, GameTest goes to state T3, and no further action occurs.

Figure 5-17 SM Chart for Dice Game Test



GameTest (Figure 5-18) implements the SM chart for the GameTest module. It contains an array of test data, a concurrent statement that generates the clock, and two processes. The first process generates *Rb*, *Reset*, and *Tnext* (the next state) whenever *Roll*, *Win*, *Lose*, or *Tstate* changes. The second process updates *Tstate* (the state of GameTest). When running the simulator, we want to display only one line of output for each roll of the dice. To facilitate this, we have added a signal *Trig1*, which changes everytime state **T2** is entered.

Figure 5-18 Dice Game Test Module

```

entity GameTest is
  port(Rb, Reset:  out bit;
       Sum: out integer range 2 to 12;
       CLK: inout bit;
       Roll, Win, Lose: in bit);
end GameTest;

library BITLIB;
use BITLIB.bit_pack.all;
architecture dicetest of GameTest is
  signal Tstate, Tnext: integer range 0 to 3;
  signal Trig1: bit;
  type arr is array(0 to 11) of integer;
  constant Sumarray:arr := (7,11,2,4,7,5,6,7,6,8,9,6);

```

```
begin
    CLK <= not CLK after 20 ns;
process(Roll, Win, Lose, Tstate)
    variable i: natural;                      -- i is initialized to 0
    begin
        case Tstate is
            when 0 => Rb <= '1';           -- wait for Roll
                Reset <='0';
            if i>=12 then Tnext <= 3;
            elsif Roll = '1' then
                Sum <= Sumarray(i);
                i:=i+1;
                Tnext <= 1;
            end if;
            when 1 => Rb <= '0'; Tnext <= 2;
            when 2 => Tnext <= 0;
                Trig1 <= not Trig1;          -- toggle Trig1
                if (Win or Lose) = '1' then
                    Reset <= '1';
                end if;
            when 3 => null;                 -- Stop state
        end case;
    end process;
process(CLK)
    begin
        if CLK = '1' then
            Tstate <= Tnext;
        end if;
    end process;
end dicetest;
```

Tester (Figure 5-19) connects the DiceGame and GameTest components so that the game can be tested. Figure 5-20 shows the simulator command file and output. The listing is triggered by *Trig1* once for every roll of the dice. The “run 2000” command runs for more than enough time to process all the test data.

**Figure 5-19 Tester for Dice Game**

```

entity tester is
end tester;
architecture test of tester is
component GameTest
port(Rb, Reset:  out bit;
      Sum: out integer range 2 to 12;
      CLK: inout bit;
      Roll, Win, Lose: in bit);
end component;
component DiceGame
port (Rb, Reset, CLK: in bit;
      Sum: in integer range 2 to 12 ;
      Roll, Win, Lose: out bit);
end component;
signal rbl, reset1, clk1, roll1, win1, lose1: bit;
signal sum1: integer range 2 to 12;
begin
  Dice: Dicegame port map(rbl,reset1,clk1,sum1,roll1,win1,lose1);
  Dicetest: GameTest port map(rbl,reset1,sum1,clk1,roll1,win1,lose1);
end test;

```

**Figure 5-20 Simulation and Command File for Dice Game Tester**

```

list /dicetest/trig1 -NOTtrigger sum1 win1 lose1 /dice/point
run 2000

```

ns	delta	trig1	sum1	win1	lose1	point
0	+0	0	2	0	0	2
100	+3	0	7	1	0	2
260	+3	0	11	1	0	2
420	+3	0	2	0	1	2
580	+2	1	4	0	0	4
740	+3	1	7	0	1	4
900	+2	0	5	0	0	5
1060	+2	1	6	0	0	5
1220	+3	1	7	0	1	5
1380	+2	0	6	0	0	6
1540	+2	1	8	0	0	6
1700	+2	0	9	0	0	6
1860	+3	0	6	1	0	6

### 5.3 REALIZATION OF SM CHARTS

Methods used to realize SM charts are similar to the methods used to realize state graphs. As with any sequential network, the realization will consist of a combinational subnetwork, together with flip-flops for storing the state of the network. In some cases, it may be possible to identify equivalent states in an SM chart and eliminate redundant states using the same method as was used for reducing state tables. However, an SM chart is usually incompletely specified in the sense that all inputs are not tested in every state, which makes the reduction procedure more difficult. Even if the number of states in an SM chart can be reduced, it is not always desirable to do so, since combining states may make the SM chart more difficult to interpret.

Before deriving next-state and output equations from an SM chart, a state assignment must be made. The best way of making the assignment depends on how the SM chart is realized. If gates and flip-flops (or the equivalent PLD realization) are used, the guidelines for state assignment given in Section 1.7 may be useful. If programmable gate arrays are used, a one-hot assignment may be best, as explained in Section 6.4.

As an example of realizing an SM chart, consider Figure 5-7(b). We have made the state assignment  $AB = 00$  for  $S_0$ ,  $AB = 01$  for  $S_1$ , and  $AB = 11$  for  $S_2$ . After a state assignment has been made, output and next-state equations can be read directly from the SM chart. Since the Moore output  $Z_a$  is 1 only in state 00,  $Z_a = A'B'$ . Similarly,  $Z_b = A'B$  and  $Z_c = AB$ . The conditional output  $Z_I = ABX'$ , since the only link path through  $Z_I$  starts with  $AB = 11$  and takes the  $X = 0$  branch. Similarly,  $Z_2 = ABX$ . There are three link paths (labeled link 1, link 2, and link 3 in Figure 5-7(b)), which terminate in a state that has  $B = 1$ . Link 1 starts with a present state  $AB = 00$ , takes the  $X = 1$  branch, and terminates on a state in which  $B = 1$ . Therefore, the next state of  $B$  ( $B^+$ ) equals 1 when  $A'B'X = 1$ . Link 2 starts in state 01, takes the  $X = 1$  branch, and ends in state 11, so  $B^+$  has a term  $A'BX$ . Similarly,  $B^+$  has a term  $ABX$  from link 3. The next-state equation for  $B$  thus has three terms corresponding to the three link paths:

$$B^+ = \underbrace{A'B'X}_{\text{link 1}} + \underbrace{A'BX}_{\text{link 2}} + \underbrace{ABX}_{\text{link 3}}$$

Similarly, two link paths terminate in a state with  $A = 1$ , so

$$A^+ = A'BX + ABX$$

These output and next-state equations can be simplified with Karnaugh maps using the unused state assignment ( $AB = 10$ ) as a don't care condition.

As illustrated above for flip-flops  $A$  and  $B$ , the next-state equation for a flip-flop  $Q$  can be derived from the SM chart as follows:

1. Identify all of the states in which  $Q = 1$ .
2. For each of these states, find all the link paths that lead *into* the state.
3. For each of these link paths, find a term that is 1 when the link path is followed. That is, for a link path from  $S_i$  to  $S_j$ , the term will be 1 if the machine is in state  $S_i$  and the conditions for exiting to  $S_j$  are satisfied.
4. The expression for  $Q^+$  (the next state of  $Q$ ) is formed by ORing together the terms found in step 3.

Next, consider the SM chart for the multiplier control (Figure 5-9). We can realize this SM chart with a PLA or ROM and two D flip-flops, using a network of the form shown in Figure 3-2. As indicated in Table 5-1, the PLA has five inputs and six outputs. Each row in the table corresponds to one of the link paths in the SM chart. Since  $S_0$  has two exit paths, the table has two rows for present state  $S_0$ . The first row corresponds to the  $St = 0$  exit path, so the next state and outputs are 0. In the second row,  $St = 1$ , so the next state is 01 and the other PLA outputs are 1000. Since  $St$  is not tested in states  $S_1$ ,  $S_2$ , and  $S_3$ ,  $St$  is a don't care in the corresponding rows. The outputs for each row can be filled in by tracing the corresponding link paths on the SM chart. For example, the link path from  $S_1$  to  $S_2$  passes through conditional output  $Ad$ , so  $Ad = 1$  in this row. Since  $S_2$  has a Moore output  $Sh$ ,  $Sh = 1$  in both of the rows for which  $AB = 10$ .

**Table 5-1 PLA Table for Multiplier Control**

	A	B	St	M	K	$A^+$	$B^+$	Load	$S_i$	Ad	Done
$S_0$	0	0	0	—	—	0	0	0	0	0	0
	0	0	1	—	—	0	1	1	0	0	0
$S_1$	0	1	—	0	0	0	1	0	1	0	0
	0	1	—	0	1	1	1	0	1	0	0
	0	1	—	1	—	1	0	0	0	1	0
$S_2$	1	0	—	—	0	0	1	0	1	0	0
	1	0	—	—	1	1	1	0	1	0	0
$S_3$	1	1	—	—	—	0	0	0	0	0	1

If a ROM is used instead of a PLA, the PLA table must be expanded to  $2^5 = 32$  rows since there are five inputs. To expand the table, the dashes in each row must be replaced with all possible combinations of 0s and 1s. If a row has  $n$  dashes, it must be replaced with  $2^n$  rows. For example, the fifth row in Table 5-1 would be replaced with the following four rows:

0	1	<b>0</b>	1	<b>0</b>	1	0	0	0	1	0
0	1	<b>0</b>	1	<b>1</b>	1	0	0	0	1	0
0	1	<b>1</b>	1	<b>0</b>	1	0	0	0	1	0
0	1	<b>1</b>	1	<b>1</b>	1	0	0	0	1	0

The added entries are printed in boldface.

By inspection of the PLA table, the logic equations for the multiplier control are

$$A^+ = A'BM'K + A'BM + AB'K = A'B(M + K) + AB'K$$

$$B^+ = A'B'St + A'BM'(K' + K) + AB'(K' + K) = A'B'St + A'BM' + AB'$$

$$Load = A'B'St$$

$$Sh = A'BM'(K' + K) + AB'(K' + K) = A'BM' + AB'$$

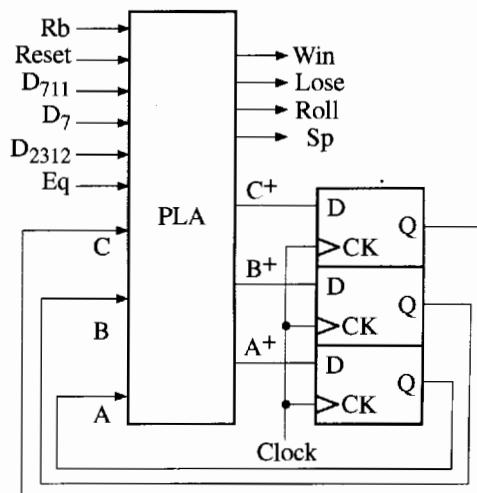
$$Ad = A'BM'$$

$$Done = AB$$

## 5.4 IMPLEMENTATION OF THE DICE GAME

We realize the SM chart for the dice game (Figure 5-13) using a PLA and three D flip-flops, as shown in Figure 5-21. We use a straight binary state assignment. The PLA has 9 inputs and 7 outputs, which are listed at the top of Table 5-2. The PLA table has one row for each link path on the SM chart. In state  $ABC = 000$ , the next state is  $A^+B^+C^+ = 000$  or 001, depending on the value of  $Rb$ . Since state 001 has four exit paths, the PLA table has four corresponding rows. When  $Rb = 1$ ,  $Roll$  is 1 and there is no state change. When  $Rb = 0$  and  $D_{711}$  is 1, the next state is 010. When  $Rb = 0$  and  $D_{2312} = 1$ , the next state is 011. For the link path from state 001 to 100,  $Rb$ ,  $D_{711}$ , and  $D_{2312}$  are all 0, and  $Sp$  is a conditional output. This path corresponds to row 4 of the PLA table, which has  $Sp = 1$  and  $A^+B^+C^+ = 100$ . In state 010, the  $Win$  signal is always on, and the next state is 010 or 000, depending on the value of  $Reset$ . Similarly,  $Lose$  is always on in state 011. In state 101,  $A^+B^+C^+ = 010$  if  $Eq = 1$ ; otherwise,  $A^+B^+C^+ = 011$  or 100, depending on the value of  $D_7$ . Since states 110 and 111 are not used, the next states and outputs are don't cares when  $ABC = 110$  or 111.

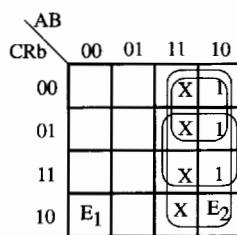
Figure 5-21 PLA Realization of Dice Game Controller



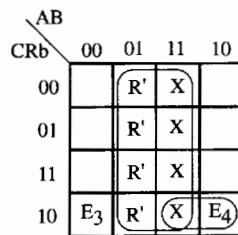
The dice game controller can also be realized using a PAL. The required PAL equations can be derived from Table 5-2 using the method of map-entered variables (see Section 1.3) or using a CAD program such as *LogicAid*. Figure 5-22 shows maps for  $A^+$ ,  $B^+$ , and  $Win$ , which were plotted directly from the PLA table. Since  $A$ ,  $B$ ,  $C$ , and  $Rb$  have assigned values in most of the rows of the table, these four variables are used on the map edges, and the remaining variables are entered within the map.  $E_1$ ,  $E_2$ ,  $E_3$ , and  $E_4$  on the maps represent the expressions given below the maps. From the  $A^+$  column in the PLA table,  $A^+$  is 1 in row 4, so we should enter  $D'_{711}D'_{2312}$  in the  $ABCRb = 0010$  square of the map. To save space, we define  $E_1 = D'_{711}D'_{2312}$  and place  $E_1$  in the square. Since  $A^+$  is 1 in rows 11, 12, and 16, 1s are placed on the map squares  $ABCRb = 1000$ , 1001, 1011. From row 13, we place  $E_2 = D_7Eq'$  in the 1010 square. In rows 7 and 8,  $Win$  is always 1 when  $ABC = 010$ , so 1s are plotted in the corresponding squares of the  $Win$  map.

**Table 5-2 PLA Table for Dice Game**

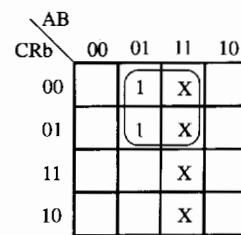
<i>ABC</i>	<i>Rb</i>	<i>Reset</i>	<i>D</i> <sub>7</sub>	<i>D</i> <sub>711</sub>	<i>D</i> <sub>2312</sub>	<i>Eq</i>	<i>A</i> <sup>+</sup>	<i>B</i> <sup>+</sup>	<i>C</i> <sup>+</sup>	<i>Win</i>	<i>Lose</i>	<i>Roll</i>	<i>Sp</i>
1	000	0	—	—	—	—	0	0	0	0	0	0	0
2	000	1	—	—	—	—	0	0	1	0	0	0	0
3	001	1	—	—	—	—	0	0	1	0	0	1	0
4	001	0	—	—	0	0	1	0	0	0	0	0	1
5	001	0	—	—	0	1	0	1	1	0	0	0	0
6	001	0	—	—	1	—	0	1	0	0	0	0	0
7	010	—	0	—	—	—	0	1	0	1	0	0	0
8	010	—	1	—	—	—	0	0	0	1	0	0	0
9	011	—	1	—	—	—	0	0	0	0	1	0	0
10	011	—	0	—	—	—	0	1	1	0	1	0	0
11	100	0	—	—	—	—	1	0	0	0	0	0	0
12	100	1	—	—	—	—	1	0	1	0	0	0	0
13	101	0	—	0	—	0	1	0	0	0	0	0	0
14	101	0	—	1	—	0	0	1	1	0	0	0	0
15	101	0	—	—	—	1	0	1	0	0	0	0	0
16	101	1	—	—	—	—	1	0	1	0	0	1	0
17	110	—	—	—	—	—	—	—	—	—	—	—	—
18	111	—	—	—	—	—	—	—	—	—	—	—	—

**Figure 5-22 Maps Derived from Table 5-2**

$$\begin{aligned} A^+ \\ E_1 &= D'_{711}D'_{2312} \\ E_2 &= D'_7E_q \end{aligned}$$



$$\begin{aligned} B^+ \\ R &= \text{Reset} \\ E_3 &= D_{711} + D'_{711}D_{2312} = D_{711} + D_{2312} \\ E_4 &= Eq + Eq'D_7 = Eq + D_7 \end{aligned}$$

*Win*

The resulting PAL equations are

$$A^+ = A'B'C Rb'D'_{711}D'_{2312} + AC' + ARb + AD'_7Eq' \quad (5-1)$$

$$B^+ = A'B'C Rb'(D_{711} + D_{2312}) + BReset' + AC Rb'(Eq + D_7)$$

$$C^+ = B'Rb + A'B'C D'_{711}D_{2312} + BC Reset' + AC D_7Eq'$$

$$Win = BC'$$

$$Lose = BC$$

$$Roll = B'CRb$$

$$Sp = A'B'C Rb'D'_{711}D'_{2312}$$

These equations can be implemented using a 16R4 PAL with no external components. Equations (5-1) can also be derived by tracing link paths on the SM chart and then simplifying the resulting equations using the don't care next states. The entire dice game, including the control network, can be implemented using a programmable gate array (see Section 6.2).

We now write a data flow VHDL model for the dice game controller based on the block diagram of Figure 5-11 and Equations (5-1). The corresponding VHDL architecture is shown in Figure 5-23. The process updates the flip-flop states and the point register when the rising edge of the clock occurs. Generation of the control signals and D flip-flop input equations is done using concurrent statements. In particular,  $D_7$ ,  $D_{711}$ ,  $D_{2312}$ , and  $Eq$  are implemented using conditional signal assignments. As an alternative, all the signals and D input equations could have been implemented in a process with a sensitivity list containing  $A$ ,  $B$ ,  $C$ ,  $Sum$ ,  $Point$ ,  $Rb$ ,  $D_7$ ,  $D_{711}$ ,  $D_{2312}$ ,  $Eq$ , and  $Reset$ . If the architecture of Figure 5-23 is used with the test bench of Figure 5-17, the results are identical to those obtained with the behavioral architecture.

To complete the VHDL implementation of the dice game, we add two modulo-six counters as shown in Figures 5-24 and 5-25. The counters are initialized to 1, so the sum of the two dice will always be in the range 2 through 12. When  $Cnt1$  is in state 6, the next clock sets it to state 1, and  $Cnt2$  is incremented (or  $Cnt2$  is set to 1 if it is in state 6).

**Figure 5-23 Data Flow Model for Dice Game**

```

architecture Dice_Eq of DiceGame is
  signal Sp, Eq, D7, D711, D2312: bit:='0';
  signal DA, DB, DC, A, B, C :bit:='0';
  signal Point: integer range 2 to 12;
begin
  process(Clk)

```

```

begin
  if rising_edge(Clk) then
    A <= DA;  B <= DB;  C <= DC;
    if Sp = '1' then Point <= Sum; end if;
  end if;
end process;
Win <= B and not C;
Lose <= B and C;
Roll <= not B and C and Rb;
Sp <= not A and not B and C and not Rb and
      not D711 and not D2312;
D7 <= '1' when Sum = 7 else '0';
D711 <= '1' when (Sum = 11) or (Sum = 7) else '0';
D2312 <= '1' when (Sum = 2) or (Sum = 3) or (Sum = 12) else '0';
Eq <= '1' when Point=Sum else '0';
DA <= (not A and not B and C and not Rb and not D711 and
       not D2312) or (A and not C) or (A and Rb)
       or (A and not D7 and not Eq);
DB <= ( (not A and not B and C and not Rb) and (D711 or D2312) )
       or (B and not Reset) or ( (A and C and not Rb)
       and (Eq or D7) );
DC <= (not B and Rb) or (not A and not B and C and not D711
      and D2312) or (B and C and not Reset) or
      (A and C and D7 and not Eq);
end Dice_Eq;

```

**Figure 5-24 Counter for Dice Game**

```

entity Counter is
  port(Clk, Roll: in bit;
        Sum: out integer range 2 to 12);
end Counter;

architecture Count of Counter is
signal Cnt1,Cnt2: integer range 1 to 6 := 1;
begin
process (Clk)
begin
  if Clk='1' then
    if Roll='1' then
      if Cnt1=6 then Cnt1 <= 1; else Cnt1 <= Cnt1+1; end if;
      if Cnt1=6 then
        if Cnt2=6 then Cnt2 <= 1; else Cnt2 <= Cnt2+1; end if;
      end if;
    end if;
  end if;
end process;
Sum <= Cnt1 + Cnt2;
end Count;

```

**Figure 5-25 Complete Dice Game**

```

entity Game is
  port (Rb, Reset, Clk: in bit;
        Win, Lose: out bit);
end Game;

architecture Play1 of Game is
  component Counter
    port(Clk, Roll: in bit;
         Sum: out integer range 2 to 12);
  end component;
  component DiceGame
    port (Rb, Reset, CLK: in bit;
          Sum: in integer range 2 to 12;
          Roll, Win, Lose: out bit);
  end component;
  signal roll1: bit;
  signal sum1: integer range 2 to 12;
begin
  Dice: Dicegame port map(Rb,Reset,Clk,sum1,roll1,Win,Lose);
  Count: Counter port map(Clk,roll1,sum1);
end Play1;

```

This section has illustrated one way of realizing an SM chart using a PLA, ROM, or PAL. Alternative procedures are available that make it possible to reduce the size of the PLA or ROM by adding some components to the network. These methods are generally based on transformation of the SM chart to different forms and encoding the inputs or outputs of the network.

## **5.5 ALTERNATIVE REALIZATIONS FOR SM CHARTS USING MICROPROGRAMMING**

In applications where the number of inputs to the control network is large, the number of inputs to the PLA, ROM, or PAL will be large. Several methods can be used to reduce the number of inputs required. These methods generally require more states in the SM chart and more output functions to be realized.

In Figure 5-26, the only inputs to the PLA come from the state register. The control network inputs go into a MUX instead of into the PLA. The PLA output has four fields: TEST, NSF, NST, and OUTPUT. TEST controls the input MUX, which selects one of the inputs to be tested in each state. If this input is 0 (false), then the second MUX selects the NSF field as the next state. If the input is 1 (true), it selects the NST field as the next state. The OUTPUT field is the same as for the standard realization. However, the SM chart must have only Moore outputs, since the outputs can be a function only of the state. (Figure 5-26 could be modified to allow Mealy outputs by replacing the OUTPUT field with OUTPUTF and OUTPUTT, and adding a MUX to select one of the two output fields.)

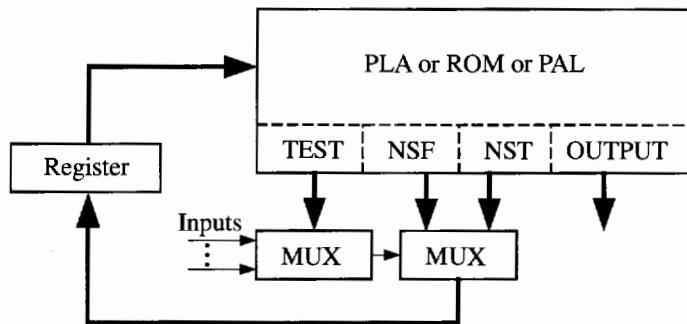
**Figure 5-26 Control Network Using an Input MUX to Select the Next State**

Figure 5-27 shows a modified version of the dice game SM chart. First, all the outputs have been converted to Moore outputs. Second, only one input variable is tested in each state. This corresponds directly to the block diagram of Figure 5-26, since the TEST field can select only one input to test in each state and the output depends only on the state.

Next we derive the PLA table (Table 5-3) using a straight binary state assignment. The variables  $Rb$ ,  $D_{711}$ ,  $D_{2312}$ ,  $Eq$ ,  $D_7$ , and  $Reset$  must be tested, so we will use an 8-to-1 MUX (Figure 5-28). When  $TEST = 001$ ,  $Rb$  is selected, etc. In state  $S_{13}$  the next state is always 0111, so  $NSF = NST = 0111$  and the TEST field is don't care. Each row in the PLA table corresponds to a link path on the SM chart. For example, in  $S_2$ , the test field 110 selects  $Reset$ . If  $Reset = 0$ ,  $NSF = 0100$  is selected, and if  $Reset = 1$ ,  $NST = 0000$  is selected. In  $S_2$ , the output  $Win = 1$  and the other outputs are 0.

**Table 5-3 PLA/ROM Table for Figure 5-27**

State	ABCD	TEST	NSF	NST	ROLL	Sp	Win	Lose
S0	0000	001	0000	0001	0	0	0	0
S1	0001	001	0010	0001	1	0	0	0
S11	0010	010	0011	0100	0	0	0	0
S12	0011	011	0101	0110	0	0	0	0
S2	0100	110	0100	0000	0	0	1	0
S13	0101	xxx	0111	0111	0	1	0	0
S3	0110	110	0110	0000	0	0	0	1
S4	0111	001	0111	1000	0	0	0	0
S5	1000	001	1001	1000	1	0	0	0
S51	1001	100	1010	0100	0	0	0	0
S52	1010	101	0111	0110	0	0	0	0

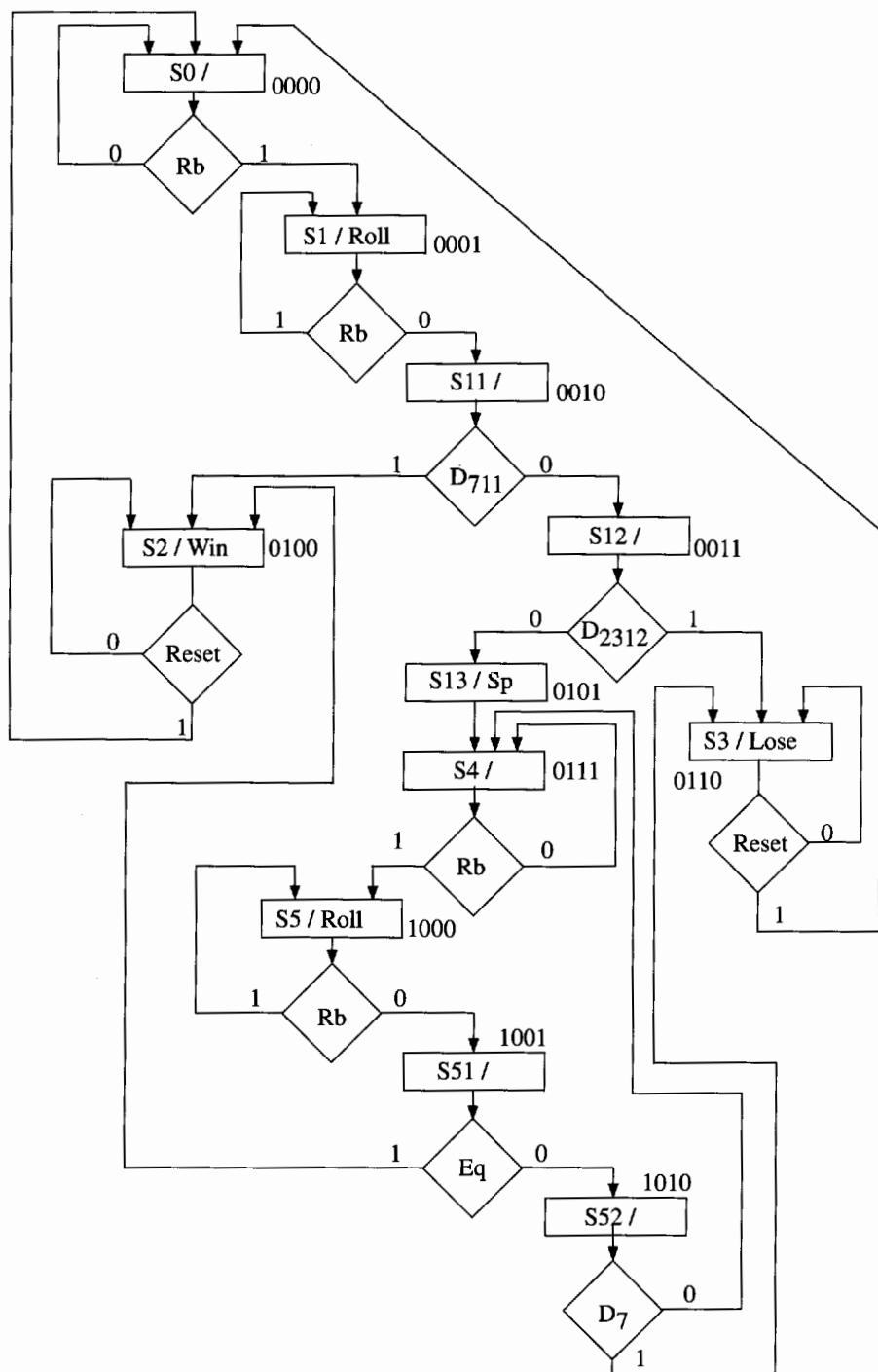
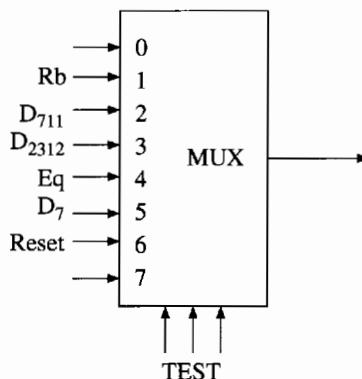
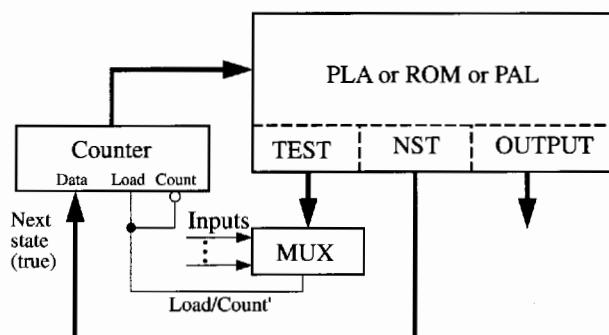
**Figure 5-27** SM Chart with Moore Outputs and One Test per State

Figure 5-28 MUX for SM Chart of Figure 5-27



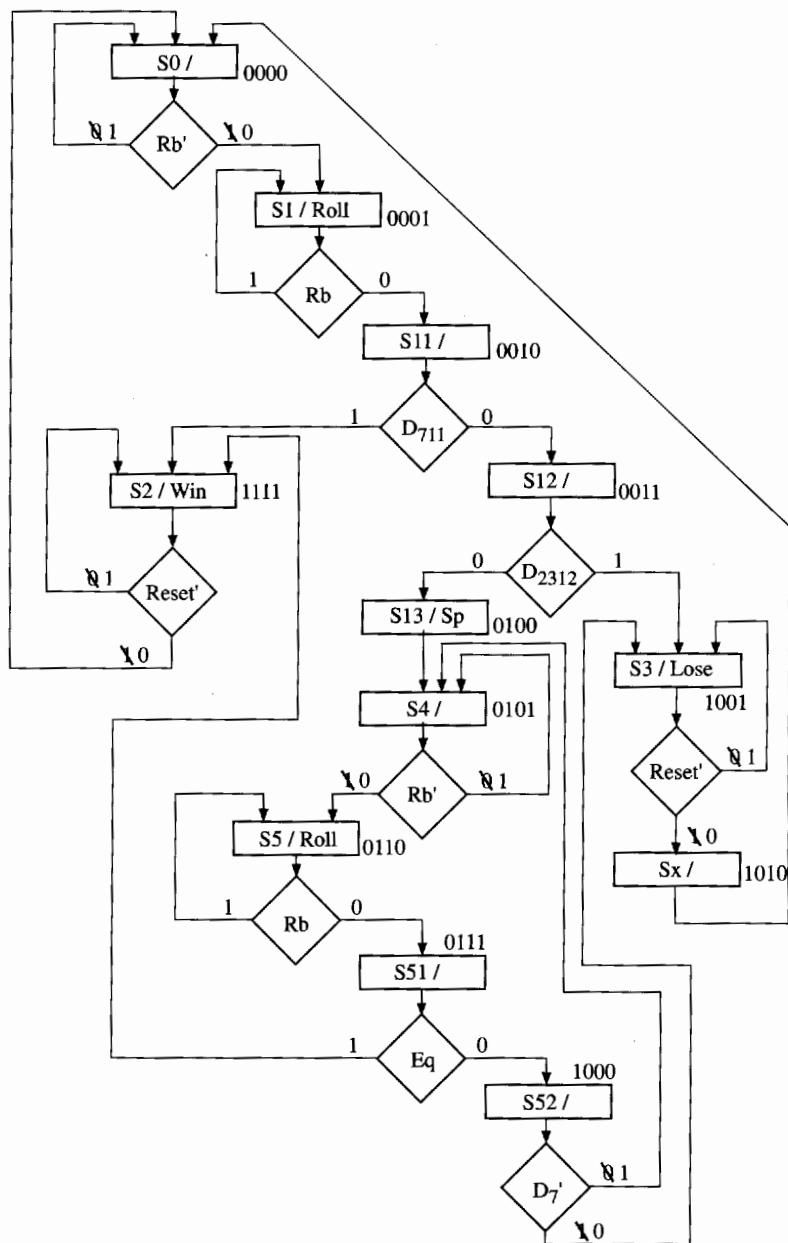
The block diagram of Figure 5-29 is similar to Figure 5-26, except the state register has been replaced with a counter. The NSF field has been eliminated from the PLA output. As before, the TEST field selects one of the inputs to be tested in each state. If the selected input is 1 (true), the NST field is loaded into the counter. If the selected input is 0, the counter is incremented. This requires that the SM chart be modified as shown in Figure 5-30 and that the state assignment be made in a special way. For each condition box, for the false branch, the next state is assigned in sequence if possible. If this is not possible, extra states (called X-states) must be added. The required number of X-states can be reduced by assigning long strings of states in sequence. To facilitate this, it may be necessary to complement some of the variables that are tested. In Figure 5-31, Rb and Reset have each been complemented in two places, and the 0 and 1 branches have been interchanged accordingly. With this change, states 0000, 0001, . . . , 1000 are in sequence. S<sub>3</sub> has been assigned 1001, and before adding an X-state, NSF was 0000 and NST was 1001, so neither next state was in sequence. Therefore, X-state S<sub>x</sub> was added with a sequential assignment 1010; the next state of S<sub>x</sub> is always 0000. If we assign 1011 to S<sub>2</sub>, the next states would be 1011 and 0000, and neither next state would be in sequence. We could solve the problem by adding an X-state. A better approach is to assign 1111 to S<sub>2</sub>, as shown. Since incrementing 1111 goes to 0000, one of the next states is in sequence, and no X-state is required.

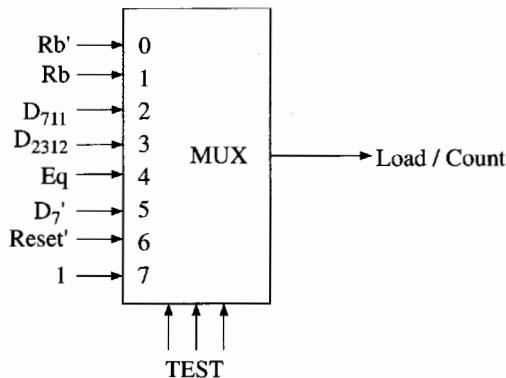
Figure 5-29 Control Network Using a Counter for the State Register



The inputs tested by the MUX in Figure 5-31 are similar to Figure 5-26 except  $D_7$  and  $Reset'$  have been complemented, and both  $Rb$  and  $Rb'$  are needed. Since NST is always 0101 in state  $S_{13}$ , a 1 input to multiplexer is needed. The corresponding PLA table is given in Table 5-4.

Figure 5-30 SM Chart with Serial State Assignment and Added X-State



**Figure 5-31 MUX for SM Chart of Figure 5-30****Table 5-4 PLA Table for Figure 5-31**

State	ABCD	Test	NST	Roll	Sp	Win	Lose
S0	0000	000	0000	0	0	0	0
S1	0001	001	0001	1	0	0	0
S11	0010	010	1111	0	0	0	0
S12	0011	011	1001	0	0	0	0
S13	0100	111	0101	0	1	0	0
S4	0101	000	0101	0	0	0	0
S5	0110	001	0110	1	0	0	0
S51	0111	100	1111	0	0	0	0
S52	1000	101	0101	0	0	0	0
S3	1001	110	1001	0	0	0	1
Sx	1010	111	0000	0	0	0	0
S2	1111	110	1111	0	0	1	0

The minimum logic equations for a PAL, derived from Table 5-4 with don't care outputs assigned to the unused states, are as follows:

$$\text{Test}(2) = BC'D' + BCD + A \quad (5-2)$$

$$\text{Test}(1) = B'C + BC'D' + AD$$

$$\text{Test}(0) = A'B'D + BD' + AD'$$

$$\text{NST}(3) = A'B'C + CD + AD$$

$$\text{NST}(2) = A'CD' + B + AC'D'$$

$$\text{NST}(1) = A'CD' + BC$$

$$\text{NST}(0) = D + A'B'C + BC' + AC'$$

$$\text{Roll} = A'B'C'D + BCD'$$

$$\text{SP} = BC'D'$$

$$\text{Win} = AB$$

$$\text{Lose} = AB'D$$

Note that each of these equations requires at most four variables, whereas some of Equations (5-1) require up to nine variables.

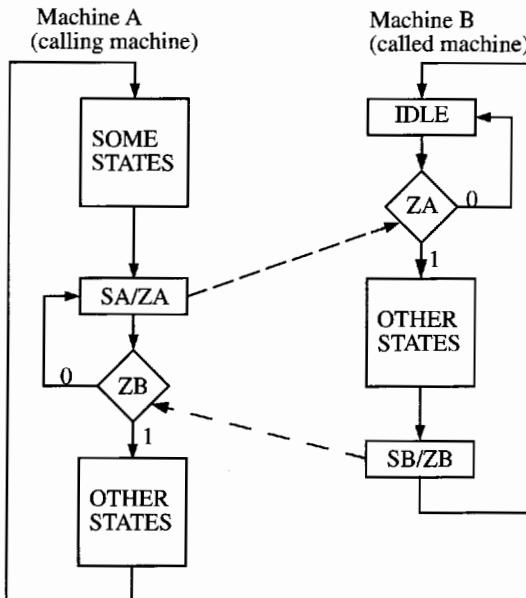
The methods we have just studied for implementing SM charts are examples of microprogramming techniques. The counter in Figure 5-28 is analogous to the program counter in a computer, which provides the address of the next instruction to be executed. The PLA or ROM output is a microinstruction, which is executed by the remaining hardware. Each microinstruction is like a conditional branch instruction that tests an input and branches to a different address if the test is true; otherwise, the next instruction in sequence is executed. The output field in the microinstruction has bits that control the operation of the hardware.

## 5.6 LINKED STATE MACHINES

When a sequential machine becomes large and complex, it is desirable to divide the machine up into several smaller machines that are linked together. Each of the smaller machines is easier to design and implement. Also, one of the submachines may be “called” in several different places by the main machine. This is analogous to dividing a large software program into procedures that are called by the main program.

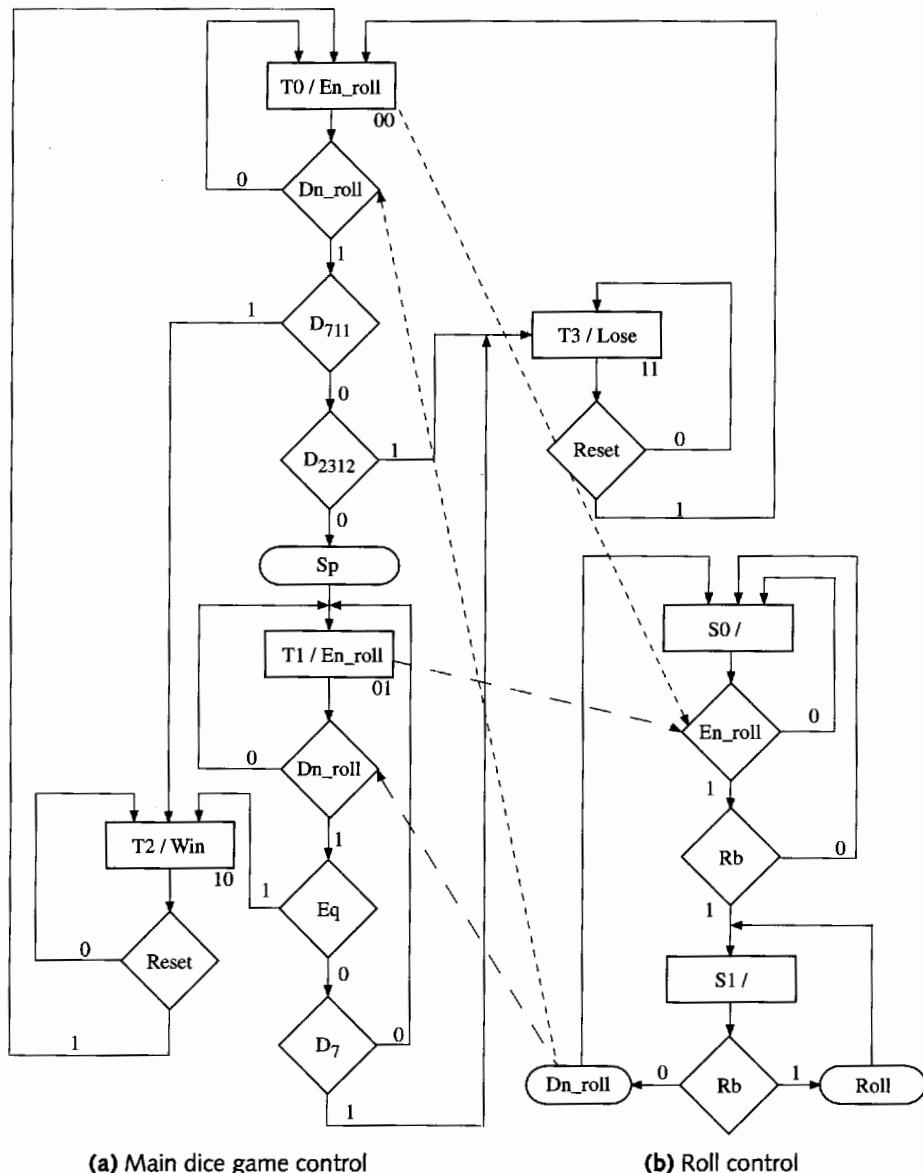
Figure 5-32 shows the SM charts for two serially linked state machines. The main machine (machine A) executes a sequence of “some states” until it is ready to call the submachine (machine B). When state SA is reached, the output signal ZA activates machine B. Machine B then leaves its idle state and executes a sequence of “other states.” When it is finished, it outputs ZB before returning to the idle state. When machine A receives ZB, it continues to execute “other states.” Figure 5-32 assumes that the two machines have a common clock.

**Figure 5-32 SM Charts for Serially Linked State Machines**



As an example of using linked state machines, we split the SM chart of Figure 5-13 into two linked SM charts. In Figure 5-13,  $Rb$  is used to control the roll of the dice in states  $S_0$  and  $S_1$  and in an identical way in states  $S_4$  and  $S_5$ . Since this function is repeated in two places, it is logical to use a separate machine for the roll control (Figure 5-33(b)). Use of the separate roll control allows the main dice control (Figure 5-33(a)) to be reduced from six states to four states. The main control generates an  $En\_roll$  (enable roll) signal in  $T_0$  and then waits for a  $Dn\_roll$  (done rolling) signal before continuing. Similar action occurs in  $T_1$ . The roll control machine waits in state  $S_0$  until it gets an  $En\_roll$  signal from the main dice game control. Then, when the roll button is pressed ( $Rb = 1$ ), the machine goes to  $S_1$  and generates a  $Roll$  signal. It remains in  $S_1$  until  $Rb = 0$ , in which case the  $Dn\_roll$  signal is generated and the machine goes back to state  $S_0$ .

Figure 5-33 Linked SM Charts for Dice Game



In this chapter we described a procedure for digital system design based on SM charts. An SM chart is equivalent to a state graph, but it is usually easier to understand the system operation by inspection of the SM chart. After we have drawn a block diagram for a digital system, we can represent the control unit by an SM chart. Next we can write a behavioral VHDL description of the system based on this chart. Using a test bench written in VHDL, we can simulate the VHDL code to verify that the system functions according to specifications. After making any necessary corrections to the VHDL code and SM chart, we can proceed with the detailed logic design of the system. Rewriting the VHDL architecture to describe the system operation in terms of control signals and logic equations allows us to verify that our design is correct.

PLA tables and logic equations can easily be derived by tracing link paths on an SM chart. When SM charts are realized using PLAs, the PLA size can be reduced by transforming the SM chart into a form in which only one input is tested in each state. However, this generally increases the number of states and slows down the operation of the system. For complex systems, we can split the control unit into several sections by using linked state machines.

## Problems

### 5.1

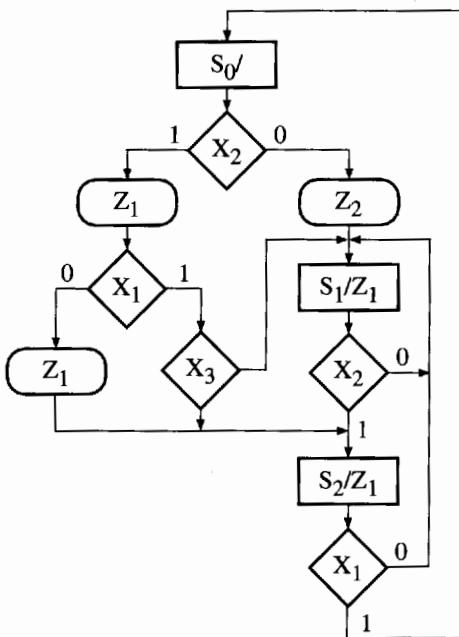
- (a) Construct an SM chart equivalent to the following state table. Test only one variable in each decision box. Try to minimize the number of decision boxes.
- (b) Write a VHDL description of the state machine based on the SM chart.

Present State	Next State				Output $Z_1Z_2$					
	$X_1X_2 =$	00	01	10	11	$X_1X_2 =$	00	01	10	11
S0	S3	S2	S1	S0			00	10	11	01
S1	S0	S1	S2	S3			10	10	11	11
S2	S3	S0	S1	S1			00	10	11	01
S3	S2	S2	S1	S0			00	00	01	01

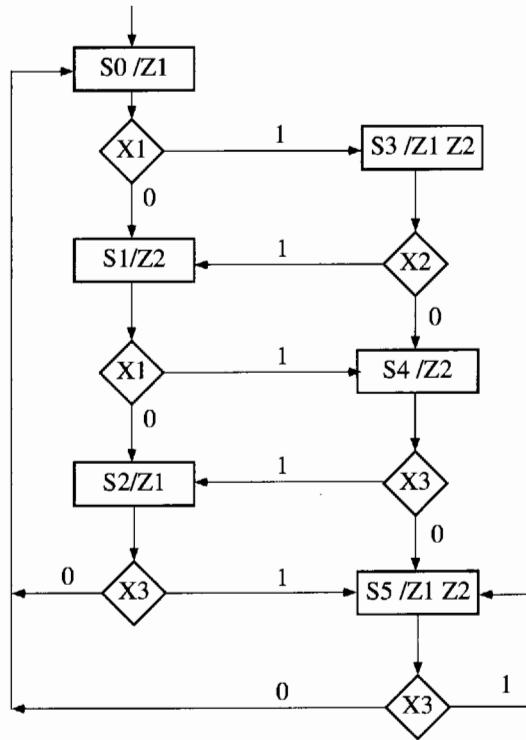
### 5.2

- (a) Draw the block diagram for a divider that divides an 8-bit dividend by a 5-bit divisor to give a 3-bit quotient. The dividend register should be loaded when St = 1.
- (b) Draw an SM chart for the control circuit.
- (c) Write a VHDL description of the divider based on your SM chart. Your VHDL should explicitly generate the control signals.
- (d) Give a sequence of simulator commands that would test the divider for the case 93 divided by 17.

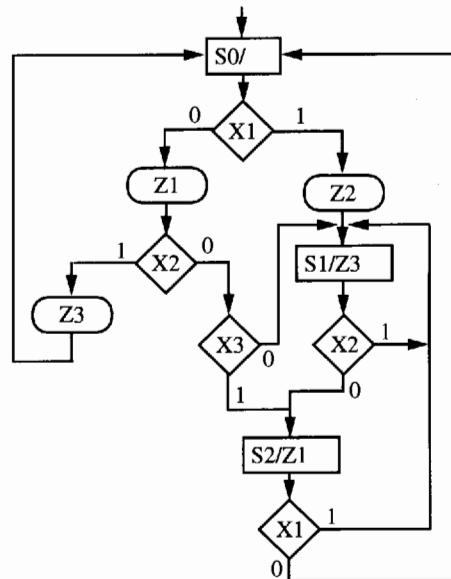
**5.3** For the following SM chart:



- (a) Draw a timing chart that shows the clock, the state ( $S_0$ ,  $S_1$ , or  $S_2$ ), the inputs ( $X_1$ ,  $X_2$ , and  $X_3$ ), and the outputs. The input sequence is  $X_1\ X_2\ X_3 = 011, 101, 111, 010, 110, 101, 001$ . Assume that all state changes occur on the rising edge of the clock, and the inputs change between clock pulses.
  - (b) Use the state assignment  $S_0: AB = 00$ ;  $S_1: AB = 01$ ;  $S_2: AB = 10$ . Derive the next state and output equations by tracing link paths. Simplify these equations using the don't care state ( $AB = 11$ ).
  - (c) Realize the chart using a PLA and D flip-flops. Give the PLA table.
  - (d) If a ROM is used instead of a PLA, what size ROM is required? Give the first five rows of the ROM table.
- 5.4** The following SM chart is to be realized using a PLA, a 4-to-1 MUX, and a 3-bit binary counter (similar to a 74163).
- (a) Draw a block diagram of the system.
  - (b) Make a suitable state assignment. Indicate any necessary changes on the SM chart.
  - (c) Give the PLA table.

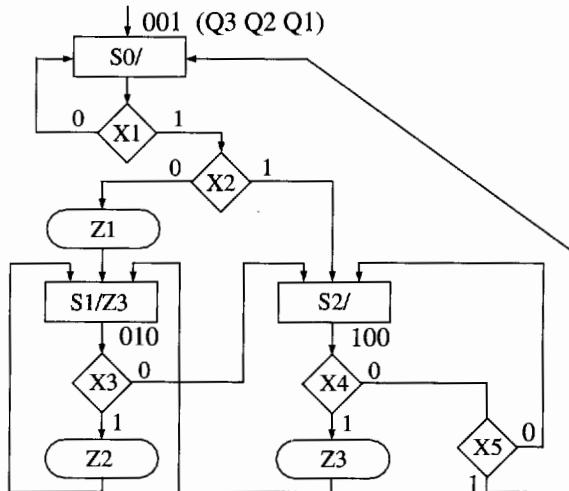


**5.5** For the following SM chart:

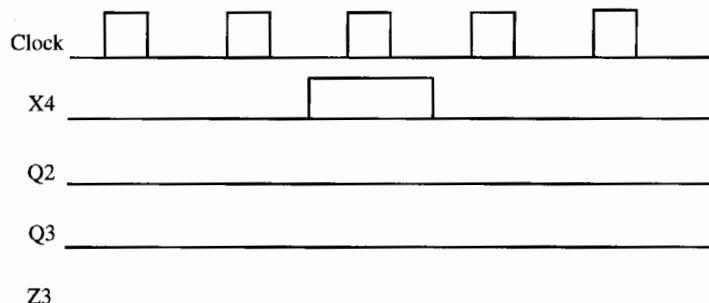


- (a) Convert the chart to the proper form for use with the hardware shown in Figure 5-26.
- (b) The left MUX data inputs are  $X_1$ ,  $X_2$ ,  $X_3$ , and 1, selected by 00, 01, 10 and 11, respectively. Assign  $S_0 = 000$ . Give the complete ROM table in binary.
- (c) Convert your answer to (a) for use with the hardware of Figure 5-29. Use the same four selector inputs as in (b). (Don't add any complemented variables as inputs to the selector.) Add X-states where needed and make a suitable state assignment (assign  $S_0 = 0000$ ).
- (d) Give the ROM table for (c) in binary.

### 5.6 For the given SM chart:



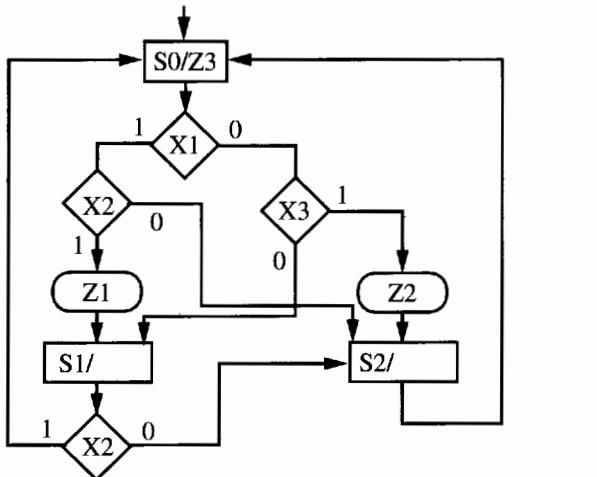
- (a) Complete the following timing diagram (assume that  $X_1 = 1$ ,  $X_2 = 0$ ,  $X_3 = 0$ ,  $X_5 = 1$ , and  $X_4$  is as shown). Flip-flops change state on falling edge of clock.



- (b) Using the given one-hot state assignment, derive the minimum next state and output equations by inspection of the SM chart.
- (c) Write a VHDL description of the digital system.
- (d) If a digital system is built using 22V10 PLDs, under what conditions, if any, would it be desirable to use a one-hot state assignment rather than using a minimum number of flip-flops?

**5.7** Realize the following SM chart using a PLA with a minimum number of inputs, a multiplexer, and a loadable counter (like the 74163). The PLA should generate both NST and NSF. The multiplexer inputs are selected as shown in the table beside the SM chart.

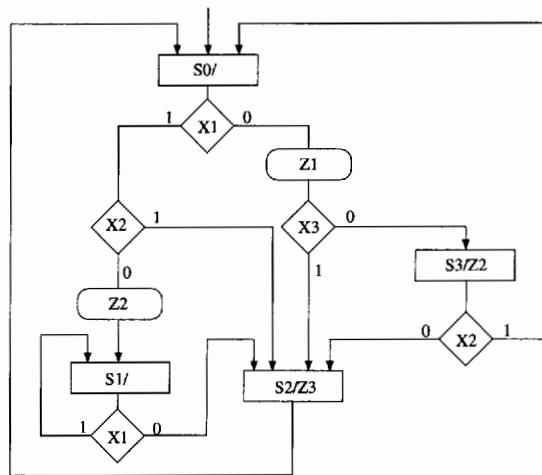
- (a) Draw the block diagram.
- (b) Convert the SM chart to the proper format. Add a minimum number of extra states.
- (c) Make a suitable state assignment and give the first five rows of the PLA table.
- (d) Write a VHDL description of the system using a PLA.



T1	T2	
00	1	
01	X1	
10	X2	
11	X3	

**5.8** The following SM chart is to be realized using the structure shown in Figure 5-26.

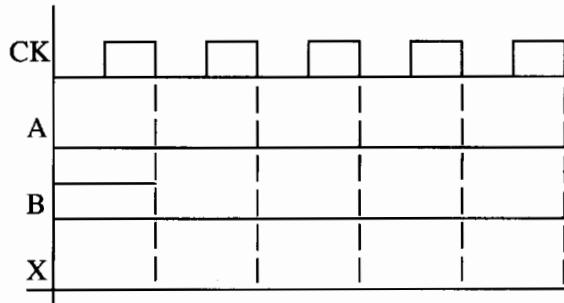
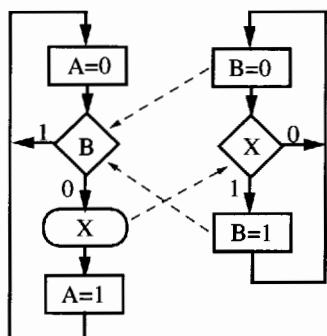
- (a) Convert the SM chart to the proper form by adding a minimum number of states to the given diagram. Make a suitable state assignment.
- (b) Complete the given ROM table.
- (c) Draw a block diagram showing how the SM chart can be realized using a ROM, multiplexers, and flip-flops.



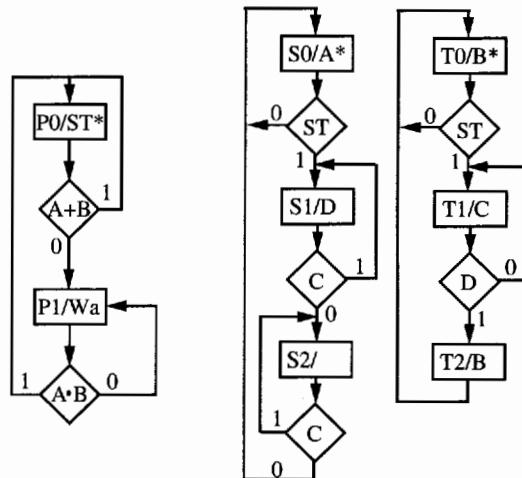
$Q_c$	$Q_b$	$Q_a$	T1	T0	CF	BF	AF	CT	BT	AT	Z1	Z2	Z3
0	0	0											

5.9 The SM charts for two linked state machines are given below. Assume that all state changes occur 10 ns after the falling edge of the common clock. Also assume that the combinational network that generates  $X$  has a propagation delay of 5 ns. The clock period is 40 ns.

- (a) Complete the timing charts. Initially  $A = 0$  and  $B = 1$ .
- (b) Realize the two SM charts using a PLA and D flip-flops. Draw a block diagram and derive the PLA table by inspection of the SM chart. Note: Only one PLA is needed.
- (c) Write VHDL code that describes the operation of the linked state machines (including the appropriate delays).

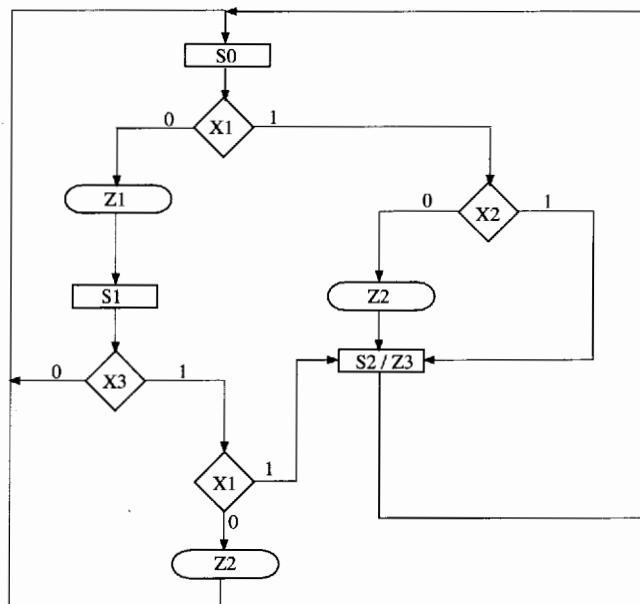


**5.10** The SM charts for three linked machines are given below. All state changes occur during the falling edge of a common clock. Complete a timing chart including  $ST$ ,  $Wa$ ,  $A$ ,  $B$ ,  $C$ , and  $D$ . All state machines start in the state with an asterisk (\*).

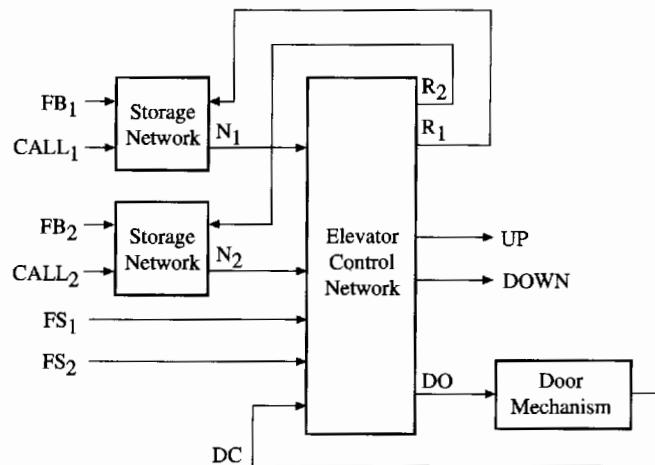


**5.11** Realize the SM chart given here using a PLA, counter, and a 4-to-1 multiplexer.

- Draw a block diagram. Show the MUX inputs.
- Change the SM chart to the proper form. Mark required changes on the following chart.
- Make a suitable state assignment. Give the first six rows of the PLA table.



**5.12** The block diagram for an elevator controller for a building with two floors is shown below. The inputs  $FB1$  and  $FB2$  are floor buttons in the elevator. The inputs  $CALL1$  and  $CALL2$  are call buttons in the hall. The inputs  $FS1$  and  $FS2$  are floor switches that output a 1 when the elevator is at the first or second floor landing. Outputs  $UP$  and  $DOWN$  control the motor, and the elevator is stopped when  $UP = DOWN = 0$ .  $N1$  and  $N2$  are flip-flops that indicate when the elevator is needed on the first or second floor.  $R1$  and  $R2$  are signals that reset these flip-flops.  $DO = 1$  causes the door to open, and  $DC = 1$  indicates that the door is closed. Draw an SM chart for the elevator controller (four states).



# CHAPTER 6

## DESIGNING WITH PROGRAMMABLE GATE ARRAYS AND COMPLEX PROGRAMMABLE LOGIC DEVICES

The PLDs we discussed in Chapter 3 are capable of implementing a sequential network but not a complete digital system. Programmable gate arrays (PGAs) and complex programmable logic devices (CPLDs) are more flexible and more versatile and can be used to implement a complete digital system on a single IC chip. Some of the larger devices can implement a small microprocessor. A typical PGA is an IC that contains an array of identical logic cells with programmable interconnections. The user can program the functions realized by each logic cell and the connections between the cells. Such PGAs are often called FPGAs since they are field-programmable.

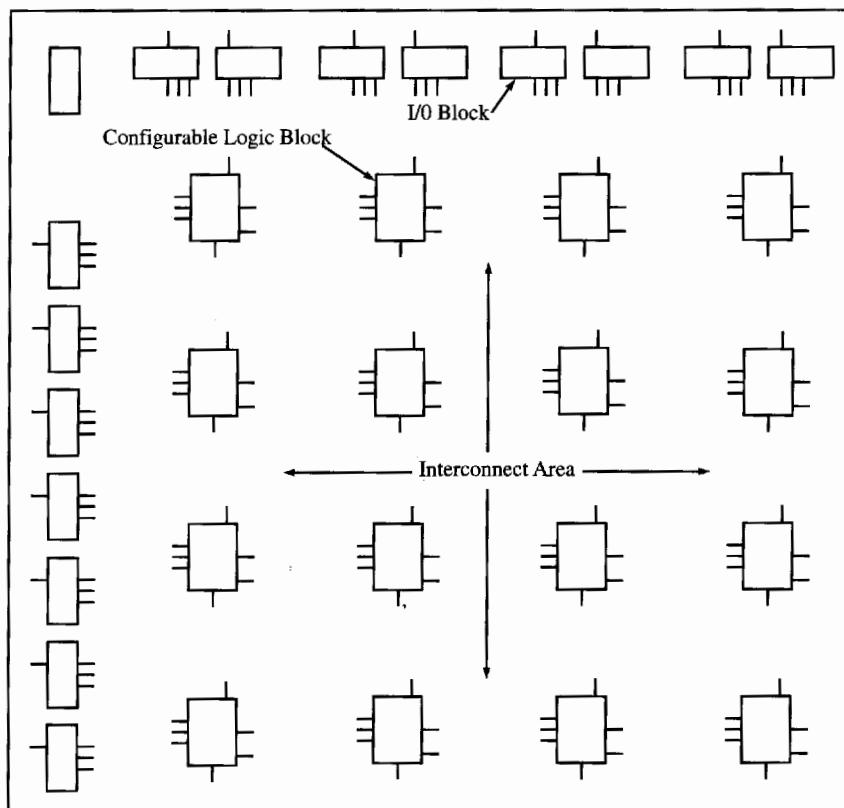
This chapter describes the internal structure of some typical FPGAs made by Xilinx. Methods for programming these FPGAs to implement digital logic are then discussed. The basic steps for designing with FPGAs are outlined, and the dice game from Chapter 5 is implemented using an FPGA. The chapter concludes with a discussion of Altera CPLDs. The reader should refer to the manufacturer's data books and web pages for more detailed specifications for the devices described in this chapter.

### 6.1 XILINX 3000 SERIES FPGAS

As an example of a FPGA, we will describe the Xilinx XC3020 Logic Cell Array (LCA). Figure 6-1 shows a part of the basic structure, which consists of an interior array of 64 configurable logic blocks (CLBs) surrounded by a ring of 64 input-output interface blocks. The interconnections between these blocks can be programmed by storing data in internal configuration memory cells. Each configurable logic block contains some combinational logic and two D flip-flops and can be programmed to perform a variety of logic functions.

The configuration memory cells are programmed after power has been applied to the LCA, and the programmed logic functions and interconnections are retained until the power is turned off. During configuration, each memory cell (see Figure 6-2) is selected in turn. When a *WRITE* signal is applied to the pass transistor, *DATA* is stored in the cell. Each connection point in the LCA has an associated memory cell, and the data stored in that cell determines whether the connection is made or not.

**Figure 6-1 Layout of Part of a Programmable Logic Cell Array**



**Figure 6-2 Configuration Memory Cell**

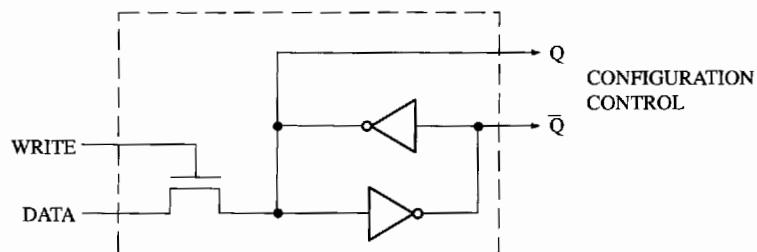
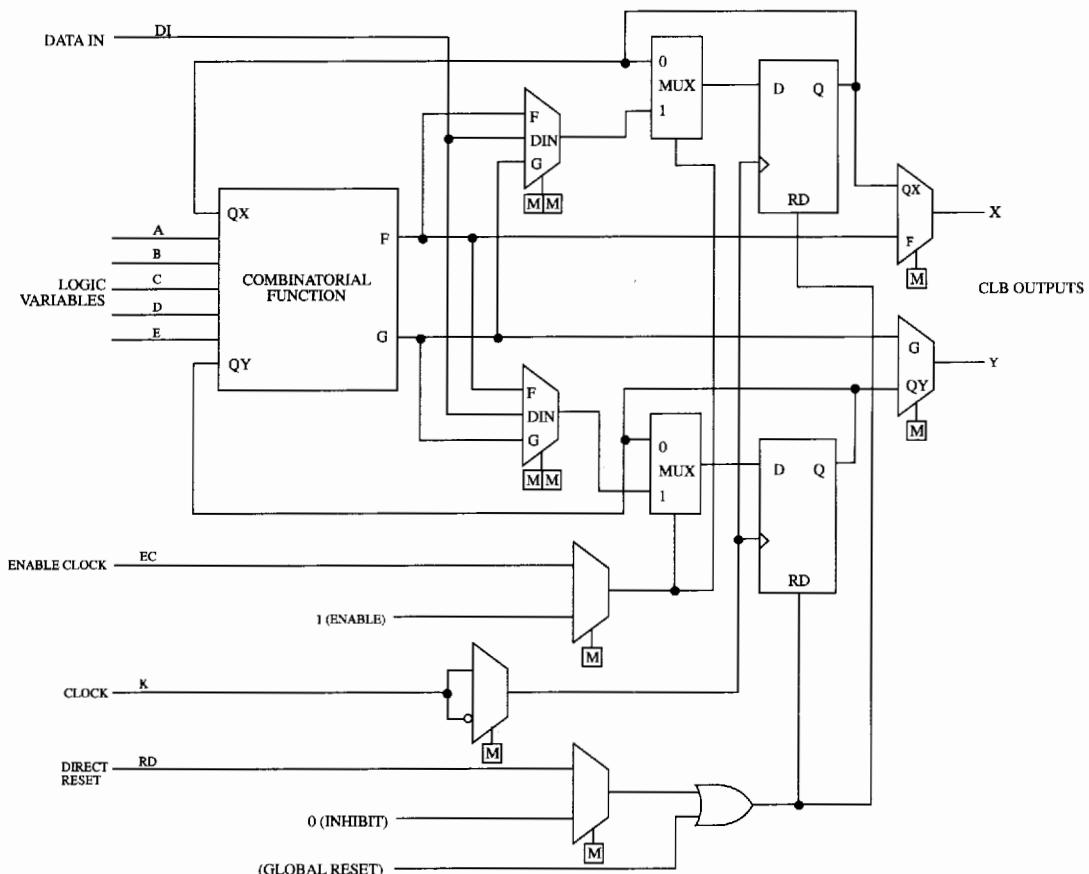


Figure 6-3 shows a configurable logic block. The block has five logic inputs ( $A, B, C, D, E$ ), a data input ( $DI$ ), a clock input ( $K$ ), a clock enable ( $EC$ ), a direct reset ( $RD$ ), and two outputs ( $X$  and  $Y$ ). The trapezoidal blocks on the diagram represent multiplexers, which can be programmed to select one of the inputs. For example, the  $X$  output can either come from the upper flip-flop ( $Q_X$ ) or from the  $F$  output of the "Combinatorial Function" block. Similarly, the  $Y$  output can come either from the lower flip-flop ( $Q_Y$ ) or from  $G$ . Each  $\blacksquare$  represents a configuration memory cell, and the data in the cell determines which MUX input is selected.

Figure 6-3 Xilinx 3000 Series Logic Cell



The Combinatorial Function block contains RAM memory cells and can be programmed to realize any function of five variables or any two functions of four variables. The functions are stored in truth table form, so the number of gates required to realize the functions is not important. Figure 6-4 shows the three different modes of operation for this block. As before, each trapezoidal block represents a multiplexer, which can be programmed to select one of its inputs. The FG mode generates two functions of four variables each. One variable (A) must be common to both functions. The next two variables can be chosen from B, C, QX, and QY. The remaining variable can be either D or E. For example, we could generate  $F = AB' + QXE$  and  $G = A'C + QYD$ . If QX and QY are not used, then the two four-variable functions must have A, B, and C in common, and the fourth variable can be D or E.

The *F* mode can generate one function of five variables (*A*, *D*, *E*, and two variables chosen from *B*, *C*, *QX*, and *QY*). We can realize functions ranging in complexity from a simple AND gate,

$$F = G = ABCDE$$

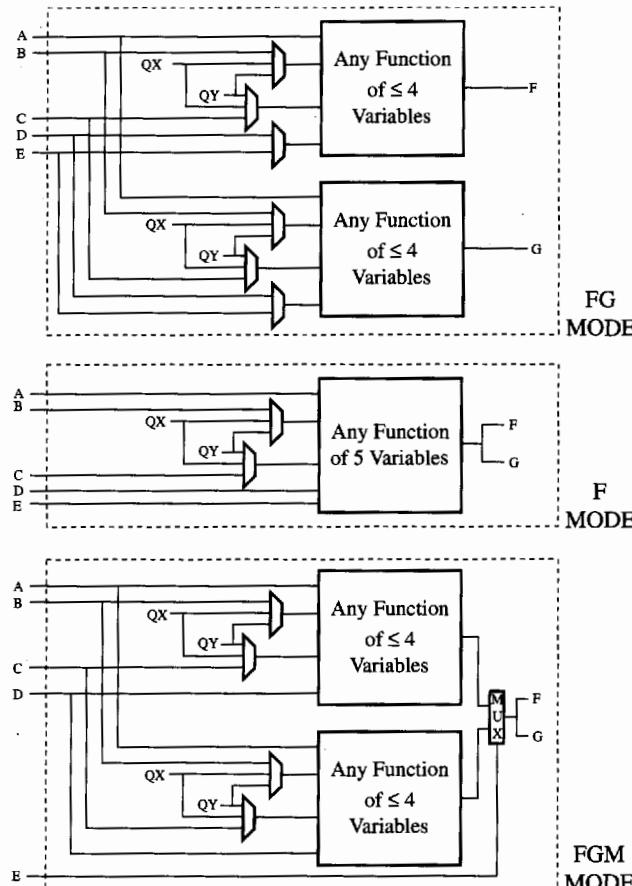
to a parity function,

$$F = G = A \oplus B \oplus C \oplus D \oplus E$$

which has 16 terms when expanded to a sum of products. The FGM mode uses a multiplexer with *E* as a control input to select one of two four-variable functions. Each function uses inputs *A*, *D*, and two of the inputs *B*, *C*, *QX*, and *QY*. The FGM mode can realize some functions of six or seven variables. For example, this mode could realize the seven-variable function

$$F = G = E(AB' + QXD) + E'(A'C + QYD)$$

**Figure 6-4 Combinatorial Logic Options**

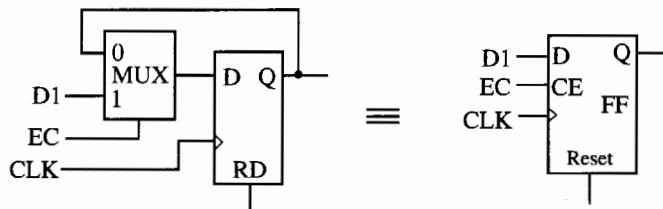


The  $D$  input on each flip-flop can be programmed to come from  $F$ ,  $G$ , or the  $D1$  data input. The two flip-flops have a common clock. The MUX connected to the CLOCK input ( $K$ ) can be programmed to select either  $K$  or  $K'$ , so the D flip-flops will change state either on the rising or falling edge of the clock. The clock is either always enabled, or it is controlled by the Enable Clock ( $EC$ ) input. The MUX connected to the  $D$  input of each flip-flop is used to effectively disable the clock. If  $EC = 0$ , the  $Q$  output is fed back to the  $D$  input so that  $Q^+ = Q$ , and the flip-flop never changes states even though the clock is changing. If  $EC = 1$ , the  $D$  input is connected to  $F$ ,  $G$ , or  $DIN$ , and state changes occur in response to the clock. The D flip-flop and MUX combination is equivalent to a D flip-flop with an enable clock ( $EC$ ) input as shown in Figure 6-5. Since  $Q$  can change only when  $EC = 1$ , the following characteristic equation describes the flip-flop behavior:

$$Q^+ = EC D + EC' Q$$

Using this type of flip-flop makes it unnecessary to gate the clock with a control signal, as was done in Section 1.12. Since the clock can go directly to each flip-flop input, achieving proper synchronous operation is much easier. The flip-flops have an active high asynchronous reset ( $RD$ ). The direct reset input (if it is not inhibited) will clear both flip-flops when it is set to 1. The global reset will clear the flip-flops in all of the cells in the array.

**Figure 6-5** Flip-flops with Clock Enable



As an example, we will implement a parallel adder-subtractor with an accumulator using an XC3020. The overall structure is similar to Figure 3-21, except control signals are needed for both add and subtract. If  $Ad = 1$ , the  $B$  input will be added to the accumulator. If  $Su = 1$ , the  $B$  input will be subtracted from the accumulator. Subtraction will be accomplished by adding the 2's complement of  $B$  to the accumulator. If  $Ad = Su = 0$ , the accumulator should remain unchanged. We will form the 2's complement by taking the 1's complement of  $B$  and setting the carry into the first full adder to 1.

Since each logic cell has two flip-flops, it might be possible to implement two bits of the accumulator in one cell. However, if two bits of the adder-subtractor were implemented in one cell, two outputs from the accumulator flip-flops plus a carry output to the next cell would be required. Since each cell has only two outputs, this scheme would not work. Therefore, we can implement only one bit per cell.

Figure 6-6 Parallel Adder-Subtracter Logic Cell

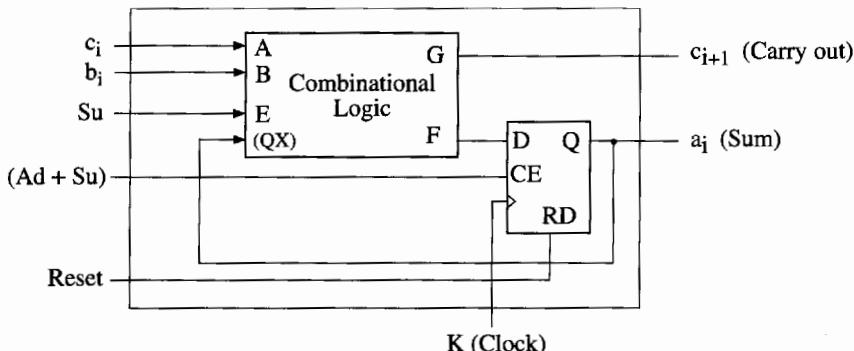


Figure 6-6 shows a typical cell of the parallel adder-subtracter. The logic inputs are  $b_i$ ,  $c_i$  (carry from previous cell), and  $Su$ . The accumulator flip-flop output ( $a_i$ ) is fed back internally within the cell. The combinatorial function block implements the following equations:

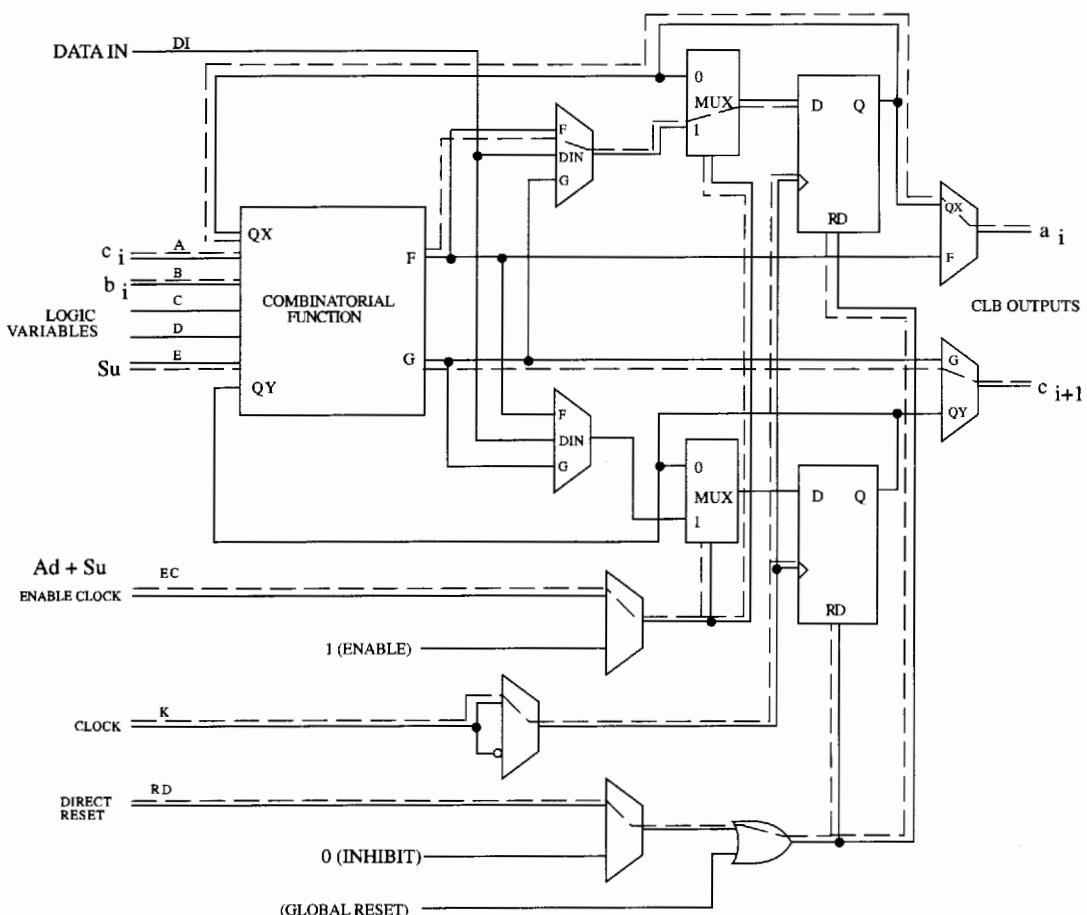
$$F = \text{sum} = a_i^+ = a_i \oplus (b_i \oplus Su) \oplus c_i$$

$$G = c_{i+1} = \text{carry out} = a_i c_i + (a_i + c_i)(b_i \oplus Su)$$

If  $Su = 0$ , these equations reduce to the standard equations for a full adder (equations (1-31) and (1-22), with  $x_i = a_i$  and  $y_i = b_i$ ). If  $Su = 1$ ,  $b_i$  is complemented by the exclusive-OR. If the carry-in to the least significant bit is also connected to  $Su$ , when  $Su = 1$  the 2's complement of  $B$  is added to  $A$ , so that subtraction will occur. Since both  $F$  and  $G$  are 4-variable functions of the same variables, we know that they can be implemented by the combinatorial function block using the FG mode in Figure 6-4. In Figure 6-6,  $c_i$  and  $b_i$  are connected to the  $A$  and  $B$  block inputs, so the internal feedback from the  $a_i$  flip-flop ( $QX$ ) must be routed to the third block input. Then the remaining input,  $Su$ , can be connected to block input  $D$  or  $E$ . Since the accumulator should only change when  $Ad = 1$  or  $Su = 1$ , we connect the clock enable ( $EC$ ) to the signal  $Ad + Su$ . An OR gate in another logic cell generates this signal, which is used by all of the adder-subtracter cells.

The dashed lines in Figure 6-7 indicate the relevant signal paths that are present within the logic cell after it has been programmed. The  $F$  function is connected to the  $D$  input of the accumulator flip-flop ( $a_i$ ), and the  $G$  function is connected to the carry out ( $c_{i+1}$ ).

**Figure 6-7 Signal Paths Within Adder-Subtracter Logic Cell**



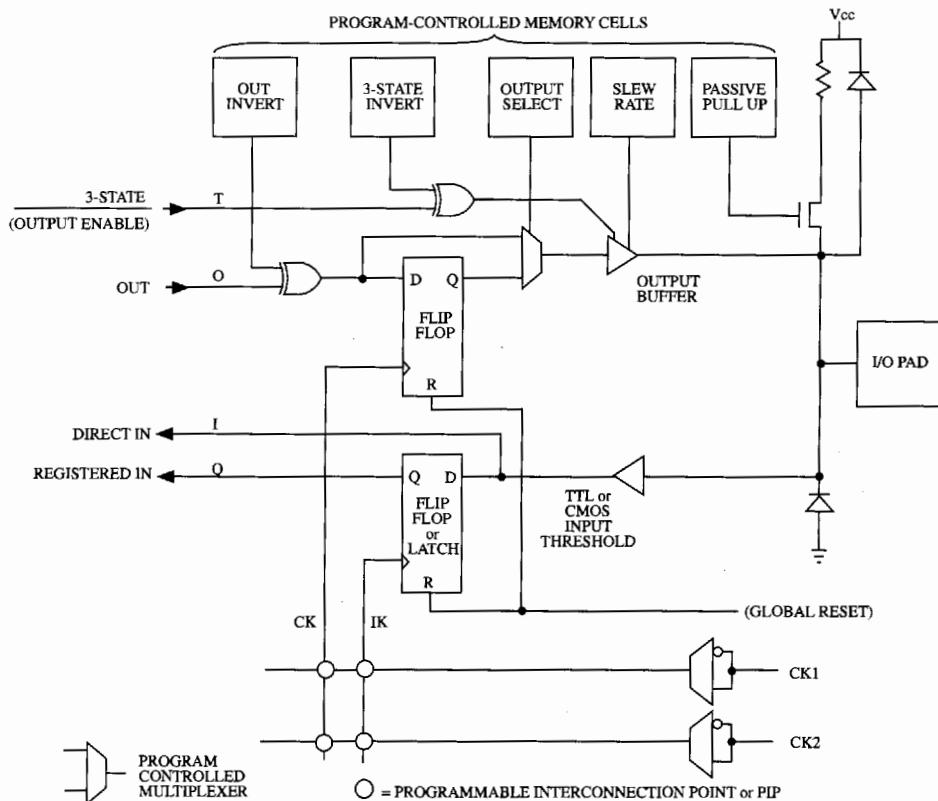
## Input-Output Blocks

Figure 6-8 shows a configurable input-output block (IOB). The I/O pad connects to one of the pins on the IC package so that external signals can be input to or output from the array of logic cells. To use the cell as an input, the 3-state control must be set to place the tristate buffer, which drives the output pin, in the high-impedance state. To use the cell as an output, the tristate buffer must be enabled. Flip-flops are provided so that input and output values can be stored within the IOB. The flip-flops are bypassed when direct input or output is desired. Two clock lines (*CK1* and *CK2*) can be programmed to connect to either flip-flop. The input flip-flop can be programmed to act as an edge-triggered D flip-flop or as a transparent latch. Even if the I/O pin is not used, the I/O flip-flops can still be used to store data.

An *OUT* signal coming from the logic array first goes through an exclusive-OR gate, where it is either complemented or not, depending on how the *OUT-INVERT* bit is

programmed. The *OUT* signal can be stored in the flip-flop if desired. Depending on how the *OUTPUT-SELECT* bit is programmed, either the *OUT* signal or the flip-flop output goes to the output buffer. If the *3-STATE* signal is 1 and the *3-STATE INVERT* bit is 0 (or

Figure 6-8 Xilinx 3000 Series I/O Block



if the *3-STATE* signal is 0 and the *3-STATE INVERT* bit is 1), the output buffer has a high-impedance output. Otherwise, the buffer drives the output signal to the I/O pad. When the I/O pad is used as an input, the output buffer must be in the high-impedance state. An external signal coming into the I/O pad goes through a buffer and then to the input of a D flip-flop. The buffer output provides a *DIRECT IN* signal to the logic array. Alternatively, the input signal can be stored in the D flip-flop, which provides the *REGISTERED IN* signal to the logic array.

Each IOB has a number of I/O options, which can be selected by configuration memory cells. The input threshold can be programmed to respond to either TTL or CMOS signal levels. The *SLEW RATE* bit controls the rate at which the output signal can change. When the output drives an external device, reduction of the slew rate is desirable to reduce the induced noise that can occur when the output changes rapidly. When the *PASSIVE PULL-UP* bit is set, a pull-up resistor is connected to the I/O pad. This internal pull-up resistor can be used to avoid floating inputs.

## Programmable Interconnects

The programmable interconnections between the configurable logic blocks and I/O blocks can be made in several ways—general-purpose interconnects, direct interconnects, and long lines. Figure 6-9 illustrates the general-purpose interconnect system. Signals between CLBs or between CLBs and IOBs can be routed through switch matrices as they travel along the horizontal and vertical interconnect lines. Direct interconnection of adjacent CLBs is possible, as shown in Figure 6-10. Long lines are provided to connect CLBs that are far apart. All the interconnections are programmed by storing bits in internal configuration memory cells within the LCA.

Figure 6-9 General-purpose Interconnects

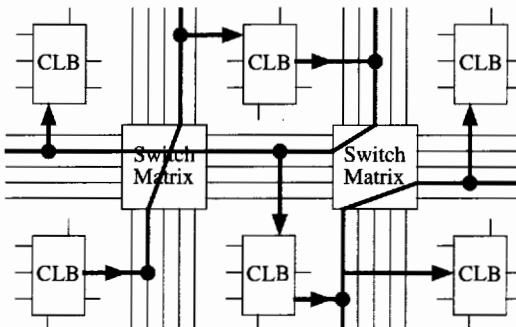
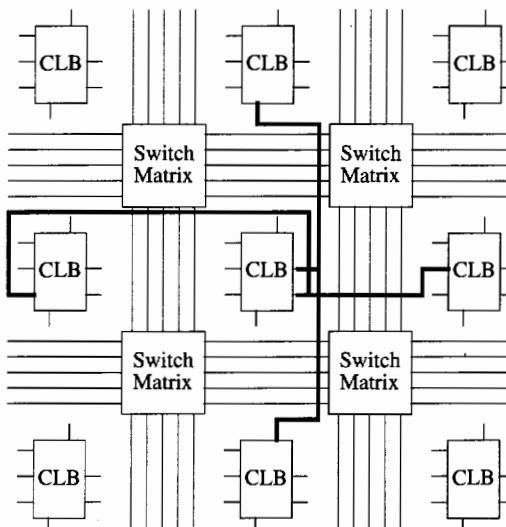
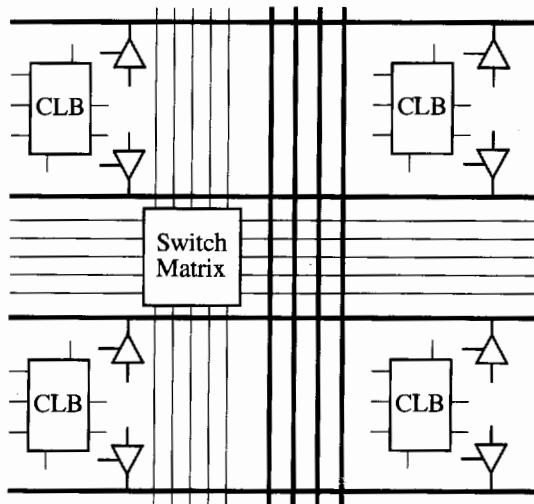


Figure 6-10 Direct Interconnects Between Adjacent CLBs



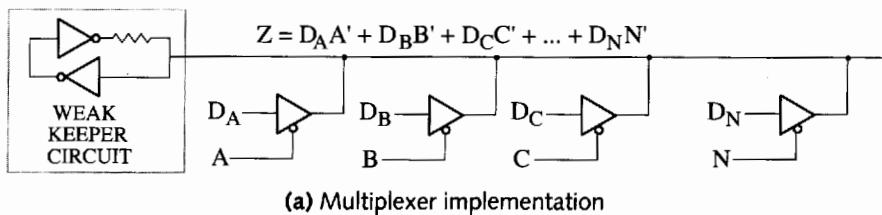
Long lines provide for high fan-out, low-skew distribution of signals that must travel a relatively long distance. As shown in Figure 6-11, there are four vertical long lines between each pair of adjacent columns of CLBs, and two of these can be used for clocks. There are two horizontal long lines between each pair of adjacent rows of CLBs. The long lines span the entire length or width of the interconnection area.

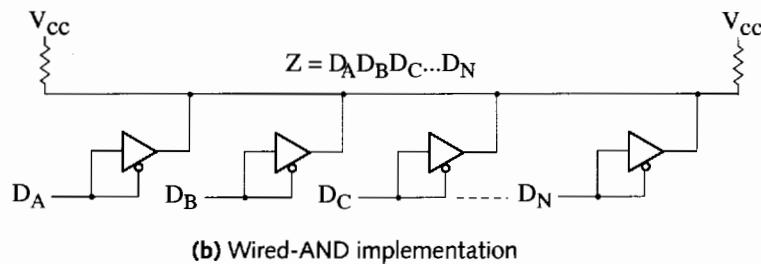
**Figure 6-11 Vertical and Horizontal Long Lines**



Each logic cell has two adjacent tristate buffers that connect to horizontal long lines. Designers can use these long lines and buffers to implement tristate busses. Figure 6-12(a) shows how tristate buffers can be used to multiplex signals onto a horizontal long line. These buffers have an active-low output enable, so when  $A = 0$ ,  $D_A$  is driven onto the line. A *weak keeper* circuit at the end of the line remembers the last value driven onto the line, so it is never left floating. Care must be taken to avoid bus contention, which would occur if both a 0 and 1 were driven onto the bus at the same time. The tristate buffers can also be used to implement a wired-AND function, as shown in Figure 6-12(b). When one or more of the  $D$  inputs is 0, the line is driven to 0. When all the  $D$  inputs are 1, all the buffer outputs are high-Z, and the pull-up resistor pulls the line up to a 1.

**Figure 6-12 Uses of Tristate Buffers**

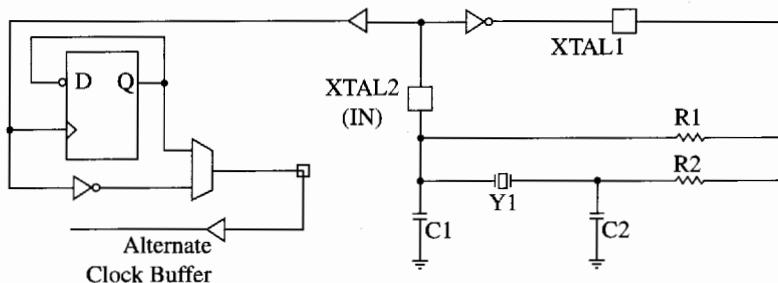




A crystal oscillator may be implemented using an internal high-speed inverting buffer together with an external crystal ( $Y_1$ ), resistors, and capacitors, as shown in Figure 6-13. The external components connect to two of the IOB pins, and the oscillator output connects to the *alternate clock buffer*. The alternate clock buffer drives a horizontal long line, which in turn can be used to drive vertical long lines and the  $K$  (clock) inputs to the logic blocks. If an external clock is used, it can be connected to the *global clock buffer*. This buffer drives a global network, which provides a high fan-out, synchronized clock to all the IOBs and logic blocks. If a symmetric clock is required, the oscillator output can be routed through a flip-flop that divides the frequency by 2.

The XC3020 FPGA, which we have just described, has 64 CLBs ( $8 \times 8$ ), 64 user I/Os, 256 flip-flops (128 in the CLBs and 128 in the IOBs), 16 horizontal long lines, and 14,779 configuration data bits. Other members of the XC3000 family have up to 484 CLBs ( $22 \times 22$ ), 176 user I/Os, 1320 flip-flops, 44 horizontal long lines, and 94,984 configuration data bits.

**Figure 6-13 Crystal Oscillator**



## 6.2 DESIGNING WITH FPGAS

Sophisticated CAD tools are available to assist with the design of systems using programmable gate arrays. One method of designing a digital system with a FPGA uses the following steps:

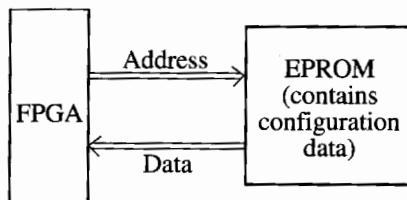
1. Draw a block diagram of the digital system. Define condition and control signals and construct SM charts or state graphs that describe the required sequence of operations.
2. Write a VHDL description of the system. Simulate and debug the VHDL code, and make any necessary corrections to the design that was developed in step 1.

3. Work out the detailed logic design of the system using gates, flip-flops, registers, counters, adders, etc.
4. Enter a logic diagram of the system into the computer using a schematic capture program. Simulate and debug the logic diagram, and make any necessary corrections to the design of step 3.
5. Run a partitioning program. This program will break the logic diagram into pieces that will fit into the configurable logic blocks.
6. Run an automatic place and route program. This will place the logic blocks in appropriate places in the FPGA and then route the interconnections between the logic blocks.
7. Run a program that will generate the bit pattern necessary to program the FPGA.
8. Download the bit pattern into the internal configuration memory cells in the FPGA and test the operation of the FPGA.

Automatic synthesis tools are available that will take a VHDL description of the system as an input and generate an interconnection of gates and flip-flops to realize the system. Using such tools effectively automates steps 3 and 4. However, certain restrictions must be placed on the VHDL code to make sure that it is synthesizable. These are discussed in Chapter 8.

When the final system is built, the bit pattern for programming the FPGA is normally stored in an EPROM and automatically loaded into the FPGA when the power is turned on. The EPROM is connected to the FPGA, as shown in Figure 6-14. The FPGA resets itself after the power has been applied. Then it reads the configuration data from the EPROM by supplying a sequence of addresses to the EPROM inputs and storing the EPROM output data in the FPGA internal configuration memory cells.

**Figure 6-14 EPROM Connections for LCA Initialization**



The following example shows how the dice game (Figure 5-11) can be implemented using an XC3020 LCA. Figure 6-15 shows the dice game block diagram after it has been entered using *ViewDraw* CAD software. Each of the modules will be expanded later. The heavy lines indicate busses, which connect some of the modules. The IPADs and OPADs represent the input and output pins on the XC3020. The output buffers (OBUFs) drive external LEDs to indicate the state of each counter and *WIN* or *LOSE*. IPADs P12 and P14 are connected to an external RC network, which together with the GOSC module forms an RC clock. This oscillator drives all the *CLK* inputs on the LCA through the ACLK buffer. External push buttons connected to P11 and P16 are used for the *GAME\_RESET* and the roll button, respectively. The roll button is connected to two D flip-flops, which serve to debounce the signal from the push button and synchronize it with the clock.

We design the dice\_controller module based on the SM charts of Figure 5-32. The main control has four states and requires two flip-flops. Using the state assignment T0:  $AB = 00$ , T1:  $AB = 01$ , T2:  $AB = 10$ , T3:  $AB = 11$ , the simplified logic equations derived from the SM chart are

$$\begin{aligned}
 A^+ &= A'B' Dn\_roll D7II + A'B' Dn\_roll D23I2 + A'B Dn\_roll Eq \\
 &\quad + A'B Dn\_roll D7 + A Reset' \\
 B^+ &= A'B' Dn\_roll D7II' + A'B Dn\_roll' + A'B Eq' + A B Reset' \\
 Win &= A B' \quad Lose = A B \quad En\_roll = A' \\
 Sp &= A'B' Dn\_roll D7II' D23I2' \tag{6-1}
 \end{aligned}$$

The roll control requires one flip-flop, and by inspection of its SM chart,

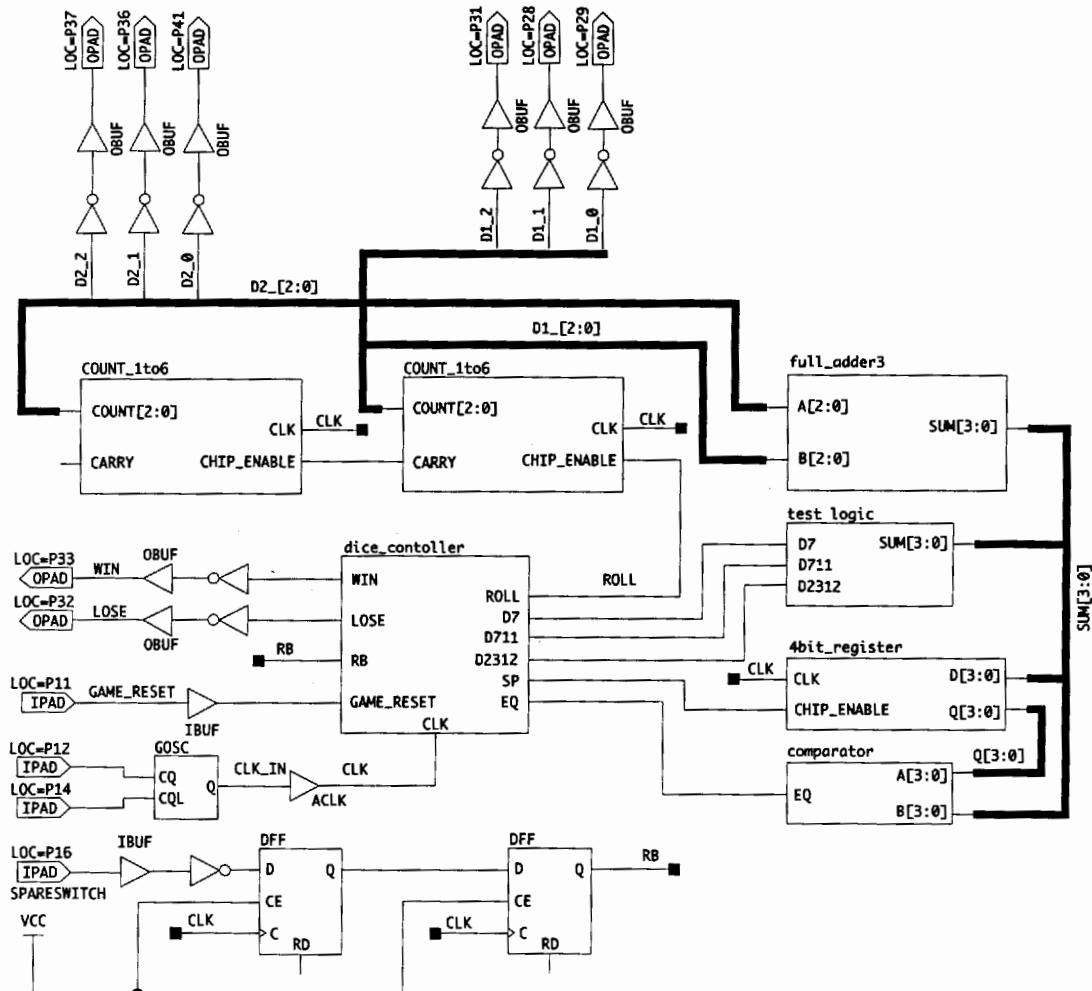
$$Q^+ = Q' En\_roll Rb + Q Rb \quad Roll = Q Rb \quad Dn\_roll = Q Rb' \tag{6-2}$$

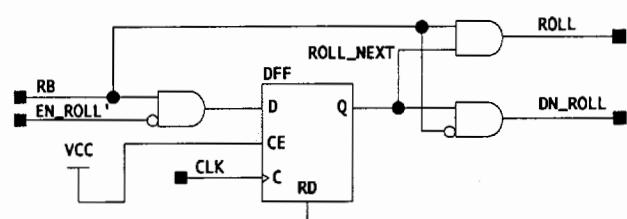
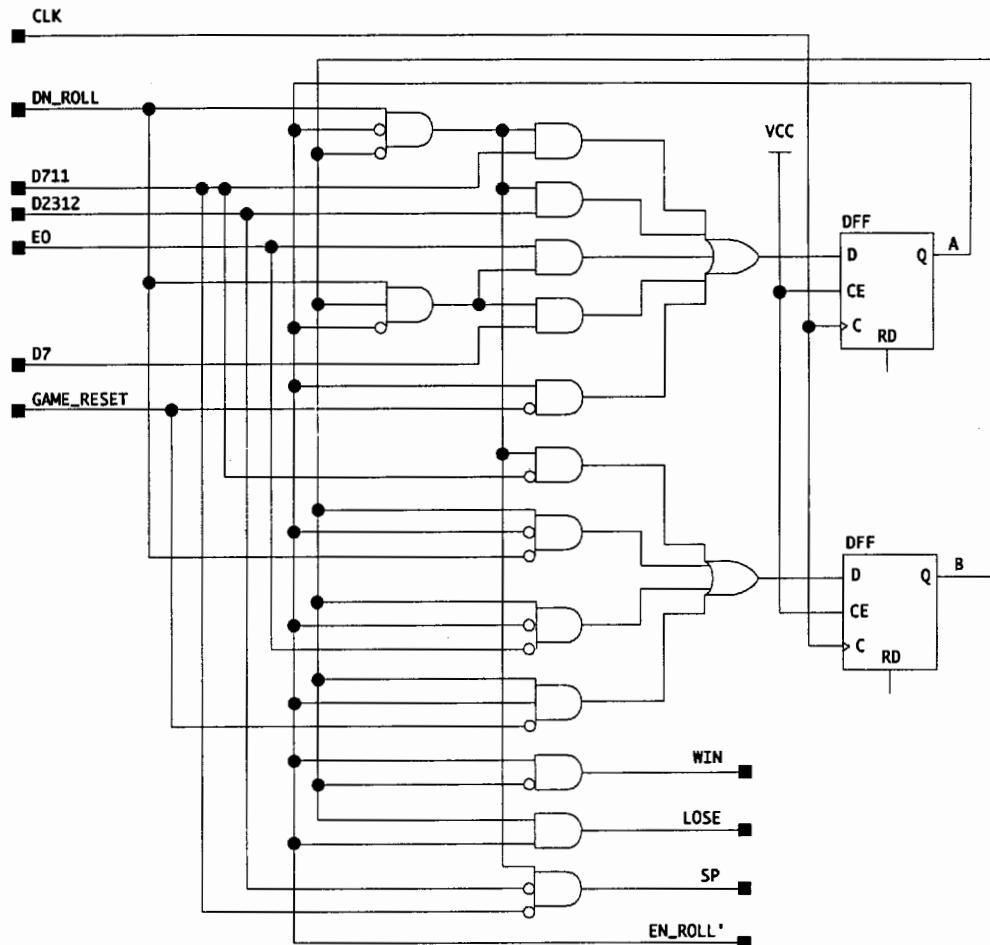
Since *En\_roll* is always '1' in S0, we can rewrite the equation for  $Q^+$  as

$$Q^+ = Q' En\_roll Rb + Q En\_roll Rb = En\_roll Rb$$

Figure 6-16 shows a *ViewDraw* schematic that implements equations (6-1) and (6-2).

Figure 6-15 Dice Game Block Diagram



**Figure 6-16 Dice Game Controller Module**

The logic equations for the modulo-6 counter are

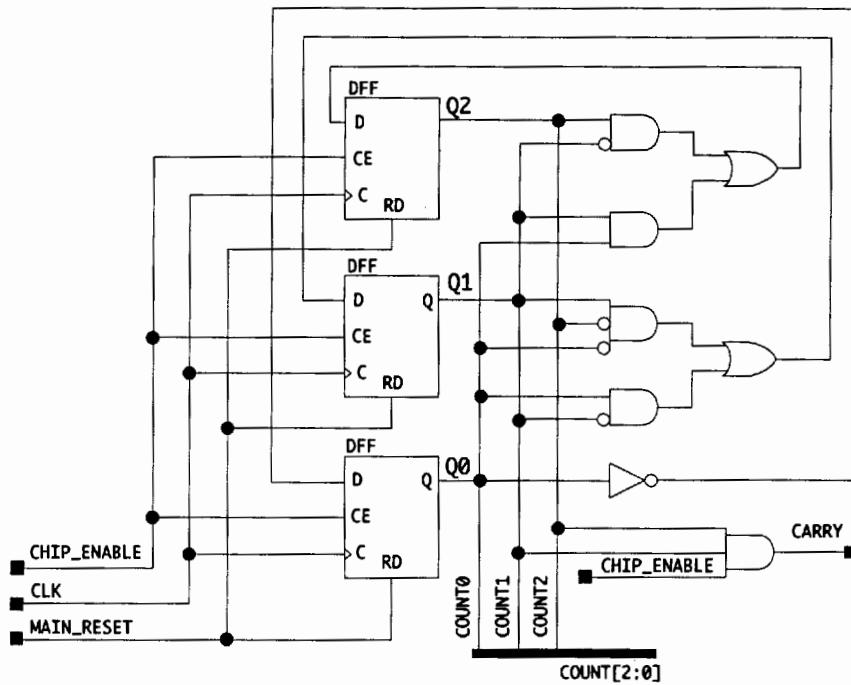
$$Q_2^+ = Q_1 Q_0 + Q_2 Q_1'$$

$$Q_1^+ = Q_2 Q_0' + Q_1 Q_0$$

$$Q_0^+ = Q_0'$$

This 3-bit counter module is implemented as shown in Figure 6-17. The *CHIP\_ENABLE* connects to *CE* on the flip-flops, so the counter increments only when *CHIP\_ENABLE* = 1.

**Figure 6-17 Modulo-6 Counter**



After the final design has been partitioned into logic cells, the logic cells placed and the connections routed, 29 out of the 64 logic cells on the 3020 are used to implement the dice game. Figure 6-18 shows the final routing of the interconnections for the dice game.

### Realizing Functions with Six or More Variables

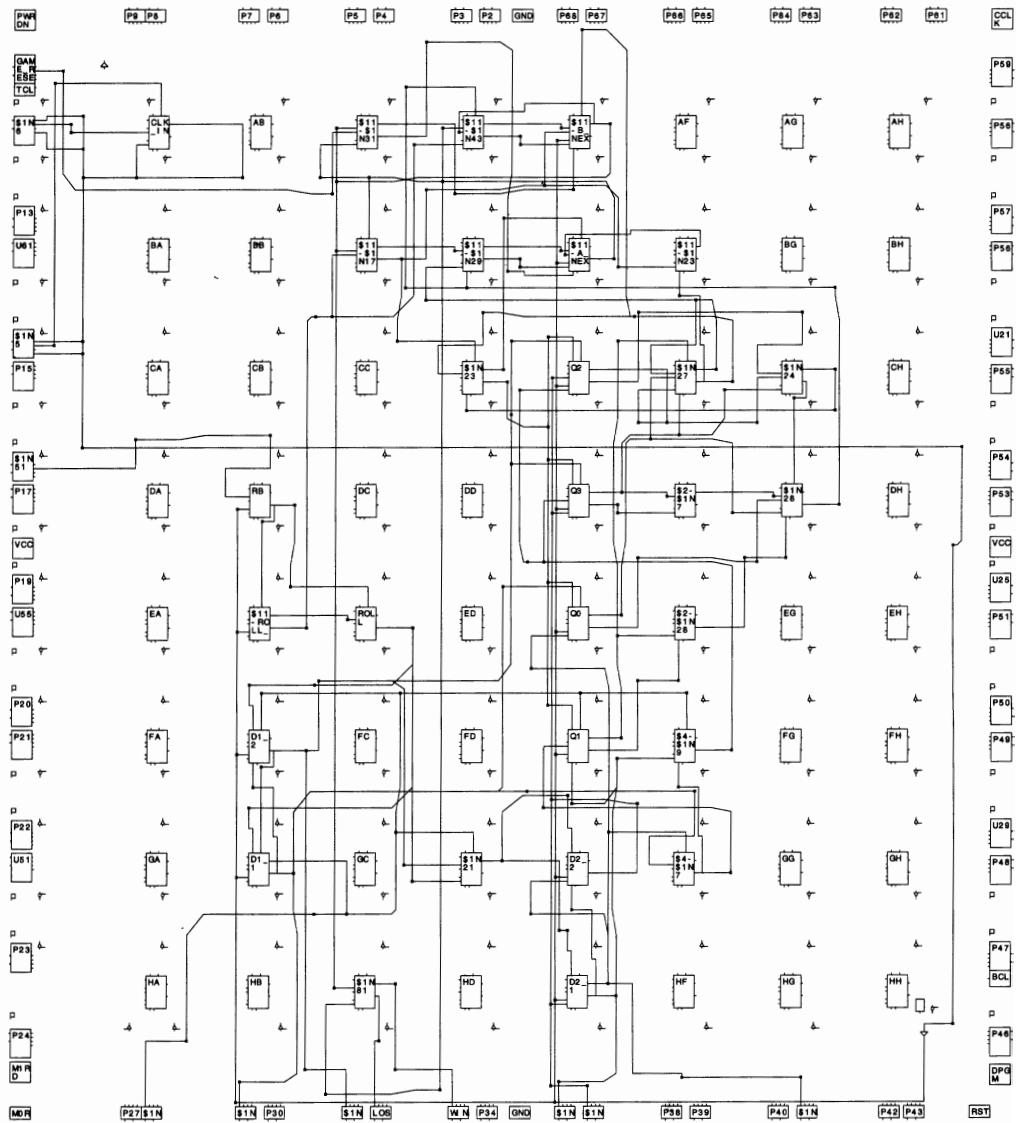
Although some 6-variable logic functions can be realized with one or two logic cells, a general 6-variable function may require three cells. We now describe a general method for realizing any 6-variable function. First, expand the function as follows:

$$Z(a,b,c,d,e,f) = a'Z(0,b,c,d,e,f) + a Z(1,b,c,d,e,f) = a'Z_0 + aZ_1 \quad (6-3)$$

This is an example of Shannon's expansion theorem. You can verify that equation (6-3) is correct by first setting  $a$  to 0 on both sides and then setting  $a$  to 1 on both sides. Since the equation is true for both  $a = 0$  and  $a = 1$ , it is always true. Equation (6-3) leads directly to the network of Figure 6-19(a), which uses two cells to realize  $Z_0$  and  $Z_1$ . Half of a third cell is used to realize the 3-variable function,  $Z = a'Z_0 + aZ_1$ . As an example, consider the following function:

$$Z = abcd'ef' + a'b'c'def' + b'cde'f$$

**Figure 6-18 Layout and Routing for Dice Game for XC3020**



Setting  $a = 0$  gives  $Z_0 = 0 \cdot bcd'ef' + 1 \cdot b'c'def' + b'cde'f = b'c'def' + b'cde'f$  and setting  $a = 1$  gives  $Z_1 = 1 \cdot bcd'ef' + 0 \cdot b'c'def' + b'cde'f = bcd'ef' + b'cde'f$ . Since  $Z_0$  and  $Z_1$  are 5-variable functions, each of them can be realized by a single cell.

Any 7-variable function can be realized with 6 or fewer logic cells. The expansion for a general 7-variable function is

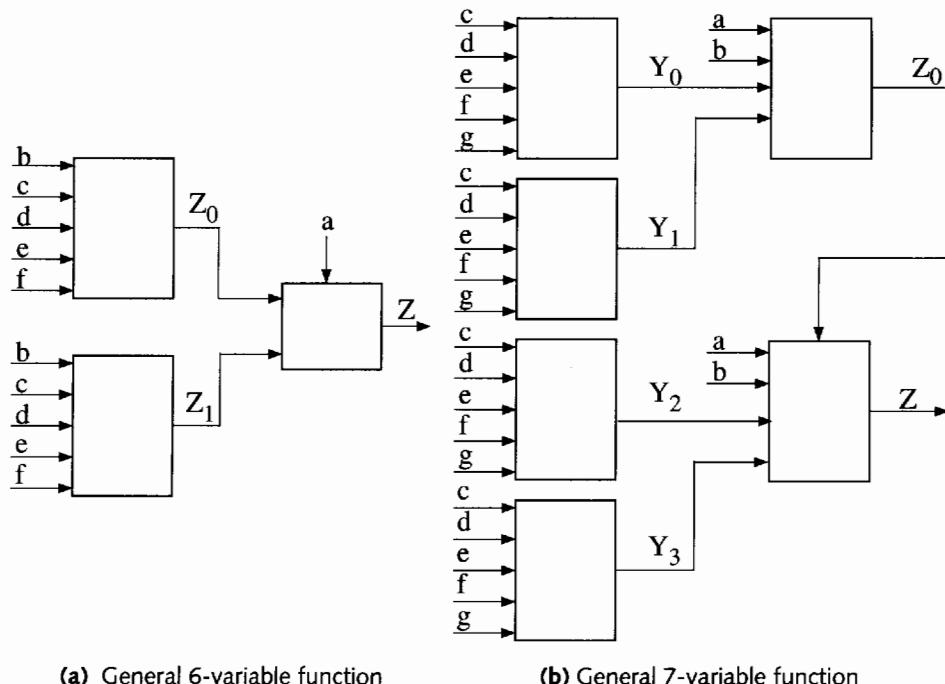
$$\begin{aligned} Z(a,b,c,d,e,f,g) &= a'b'Z(0,0,c,d,e,f,g) + a'bZ(0,1,c,d,e,f,g) \\ &\quad + ab'Z(1,0,c,d,e,f,g) + abZ(1,1,c,d,e,f,g) \\ &= a'b'Y_0 + a'bY_1 + ab'Y_2 + abY_3 \end{aligned} \quad (6-4)$$

Equation (6-4) can be obtained by applying the expansion theorem twice, first expanding about  $a$  and then expanding about  $b$ . As an example, consider the 7-variable function

$$Z = c'de'fg + bcd'e'fg' + a'c'def'g + a'b'd'ef'g' + ab'defg'$$

- Substituting  $a = b = 0$  gives  $Y_0 = c'de'fg + c'def'g + d'ef'g'$ .
- Substituting  $a = 0, b = 1$  gives  $Y_1 = c'de'fg + cd'e'fg' + c'def'g$ .
- Substituting  $a = 1, b = 0$  gives  $Y_2 = c'de'fg + defg'$ .
- Substituting  $a = b = 1$  gives  $Y_3 = c'de'fg + cd'e'fg'$ .

**Figure 6-19 Realization of 6- and 7-variable Functions**



The network of Figure 6-19(b) implements equation (6-4). Four cells implement the 5-variable functions,  $Y_0$ ,  $Y_1$ ,  $Y_2$ , and  $Y_3$ . A fifth cell implements the 4-variable function,  $Z_0 = a'b'Y_0 + a'bY_1$ , and the remaining cell implements a 5-variable function,  $Z = Z_0 + ab'Y_2 + abY_3$ . As the number of variables ( $n$ ) increases, the maximum number of logic cells required to realize an  $n$ -variable function increases rapidly. For this reason, PLDs may be a better solution than LCAs when  $n$  is large.

### 6.3 XILINX 4000 SERIES FPGAS

The Xilinx 4000 series FPGAs are similar to the 3000 series, but more inputs and outputs and many other features have been added. Figure 6-20 shows a simplified block diagram of an XC4000 configurable logic block. It has nine logic inputs ( $F1, F2, F3, F4, G1, G2, G3, G4$ , and  $H1$ ). It can generate two independent functions of four variables:

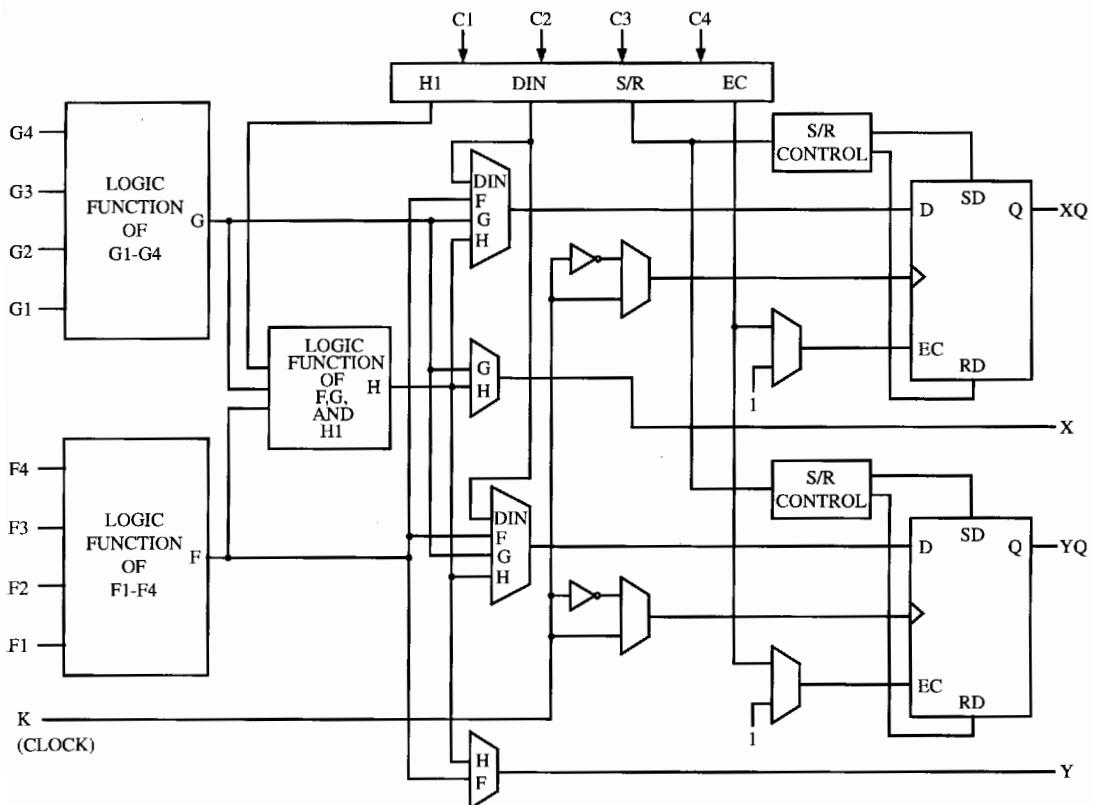
$$G(G1, G2, G3, G4) \text{ and } F(F1, F2, F3, F4)$$

This is in contrast to the 3000 series, where the inputs to the 4-variable function generators must overlap. The 4000 series CLB can also generate a function  $H$ , which depends on  $F$ ,  $G$ , and  $H1$ . By setting  $F1 = G1$ ,  $F2 = G2$ ,  $F3 = G3$ , and  $F4 = G4$ , it can generate any function of five variables in the form  $H = F(F1, F2, F3, F4) H1' + G(F1, F2, F3, F4) H1$ . It can also generate some functions of 6, 7, 8, and 9 variables.

The CLB has two D flip-flops with enable clock ( $EC$ ) inputs. The CLB has four outputs, two from the flip-flops and two from the combinational logic function generators. Unlike the 3000 CLB, the 4000 CLB has no internal feedback, so flip-flop outputs must be routed externally back to the logic inputs when feedback is required. The CLB has an *S/R* (set/reset) input that can be independently configured to connect to the *SD* or *RD* input on each flip-flop. Thus one flip-flop could be set to 1 and the other set to 0 using the same *S/R* signal. The clock input to each flip-flop can be configured to trigger on either the rising edge or falling edge of the *K* (clock) input. The *EC* input on each flip-flop can either be always enabled or it can be enabled by the *EC* input to the CLB. The *D* input to each flip-flop can be connected to *DIN*, *F*, *G*, or *H*.

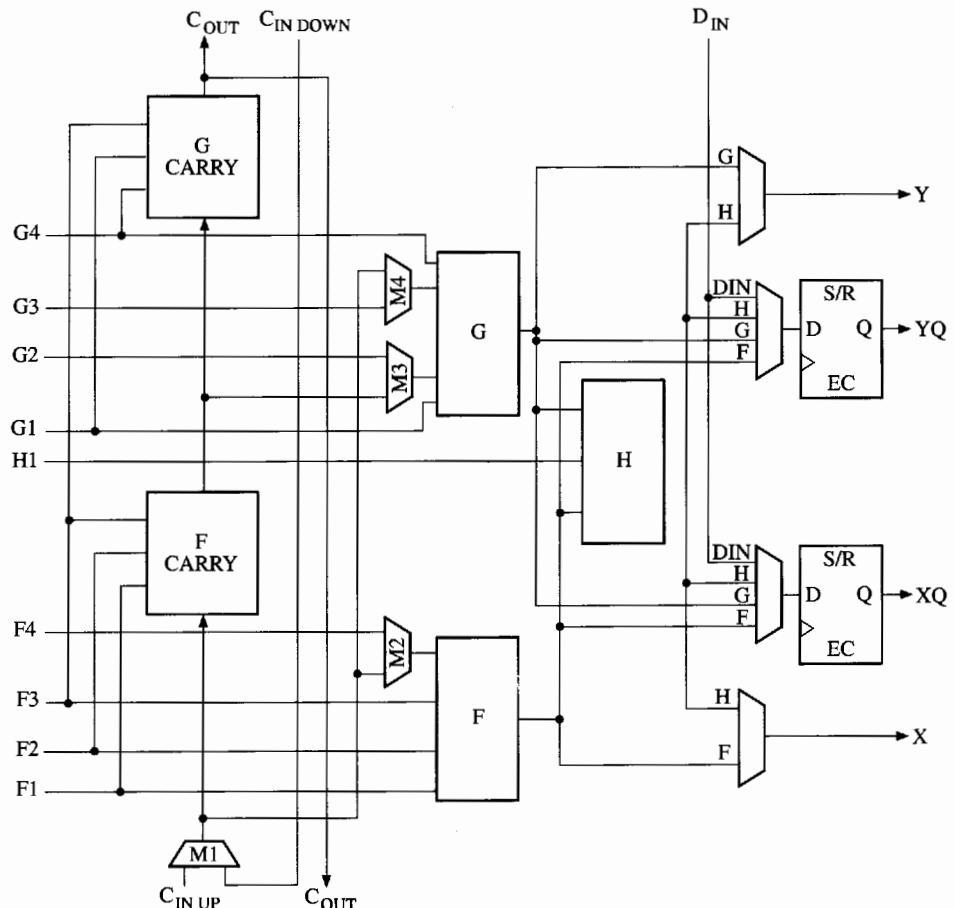
The 4000 logic cell contains dedicated carry logic, as shown in Figure 6-21. Each cell contains carry logic for two bits, and the *F* and *G* function generators can be used to generate the sum bits. Thus, each cell can be programmed to implement both the carry and sum for two bits of a full adder. The carry lines between cells are hardwired (not programmable) to provide for fast propagation of carries. The carries can propagate up or down between cells, and a programmable multiplexer (labeled *M1* in Figure 6-21) selects the direction of propagation. Multiplexers *M2*, *M3*, and *M4* allow some of the *F* and *G* function generator inputs to come from carries instead of from the normal *F* and *G* cell inputs. In addition to adders, the carry logic can be programmed to implement subtracters, adder/subtracters, incrementer/decrementers, 2's complementers, and counters.

Figure 6-20 Simplified Block Diagram for 4000 Series CLB



When a cell is programmed to implement two bits of a full adder, the logic is equivalent to that shown in Figure 6-22. In this configuration,  $A_i$  and  $B_i$  come from cell inputs  $F1$  and  $F2$ .  $A_{i+1}$  and  $B_{i+1}$  come from  $G1$  and  $G4$ . The  $C_i$  input and the  $C_{i+2}$  output connect to the hardwired carry propagation lines. Figure 6-23 shows the connections for a 4-bit adder. The middle two CLBs perform the four-bit addition. If a carry into the least significant bit is needed,  $C_0$  must be routed through an additional cell, since no direct connection to the hardwired carry lines is allowed. If an overflow indication and a carry out from the most significant bit is needed, a fourth cell is required. In this example,  $C_3$  (instead of  $C_4$ ) is routed to the fourth cell, and the cell inputs  $A_3$  and  $B_3$  are duplicated.  $C_4$  is computed using the  $F$  function generator to provide a carry output from the cell.  $C_4$  is also recomputed using the carry logic in the cell, and then the overflow is computed by the  $G$  function generator as  $V = C_3 \oplus C_4$ . This 4-bit adder can easily be expanded to 8 or 16 bits by adding two or six additional cells. These adder modules are available in the Xilinx library.

Figure 6-21 XC4000 Dedicated Carry Logic



When a cell is programmed to implement two bits of an adder/subtractor, an *Add/Sub* signal must be connected to *F3* and *G3*. When *Add/Sub* = 0, the  $B_i$  and  $B_{i+1}$  inputs are inverted inside the carry logic and the function generators so subtraction can be accomplished by adding the 2's complement.

Figure 6-22 Conceptual Diagram of a Typical Addition (2 Bits/CLB)

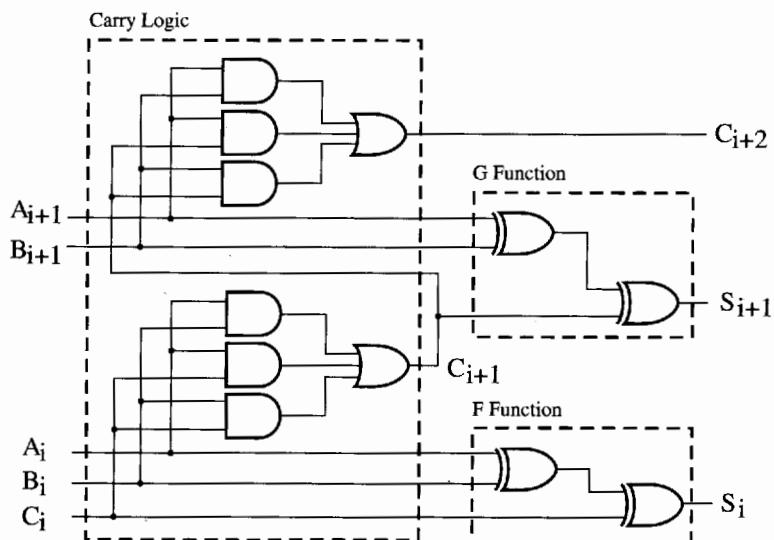
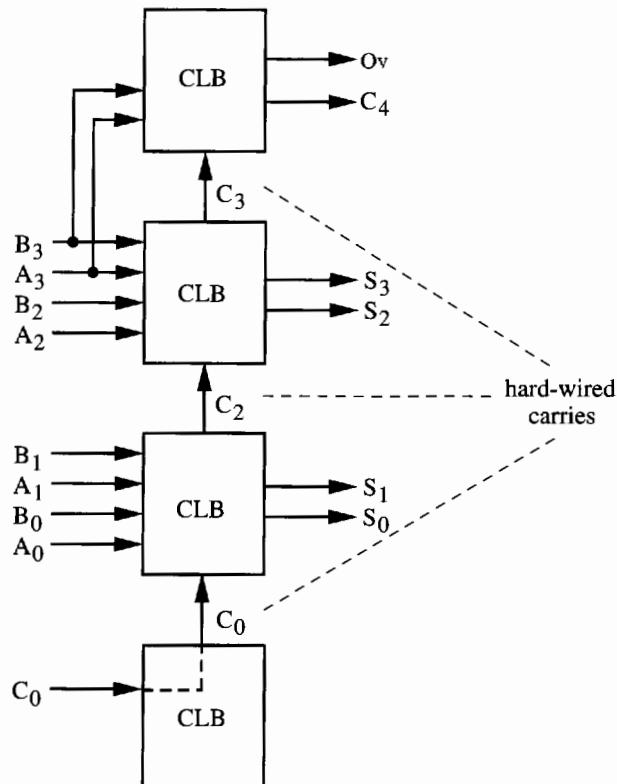


Figure 6-23 Connections for a 4-bit Adder



The *F* and *G* function generators can be programmed to serve as RAM memories (see Figure 6-24). Each cell can be configured as a 16-word-by-2-bit RAM. The *F* and *G* function generator inputs ( $F1 = G1$ ,  $F2 = G2$ ,  $F3 = G3$ , and  $F4 = G4$ ) provide a 4-bit address,  $C1$  is used as a write-enable line (*WE*), and  $C2$  and  $C3$  serve as data inputs ( $D1$  and  $D0$ ). The contents of the memory cells being addressed are available at the *F* and *G* function generator outputs. Alternatively, the cell can be configured as a  $32 \times 1$ -bit RAM, with  $C2$  used as the fifth address bit and  $C3$  as the data input. The configuration bits labeled *WRITE G* and *WRITE F* must be set to allow writing to the *G* and *F* function generators, respectively.

**Figure 6-24 CLB as a Read/Write Memory Cell**

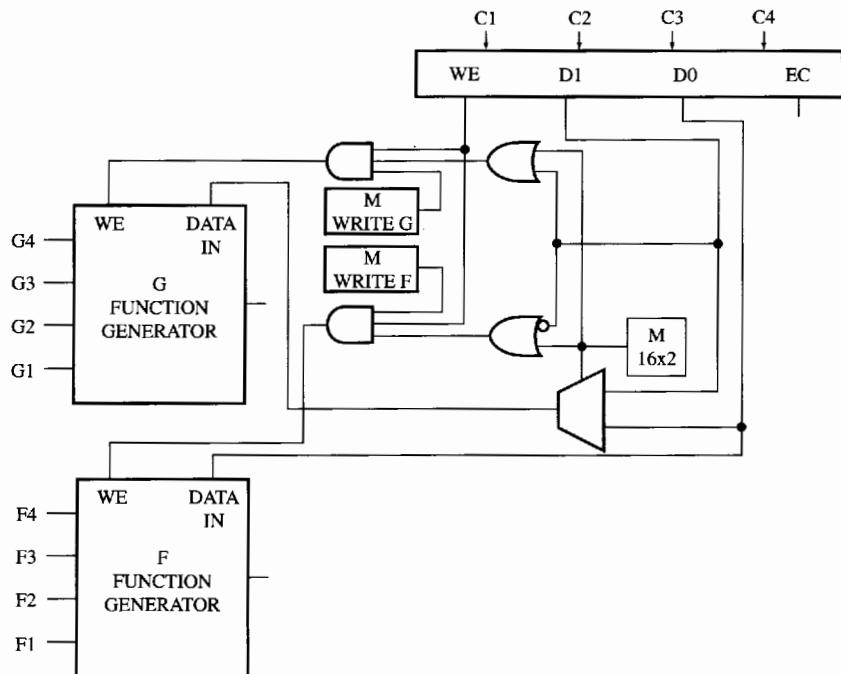
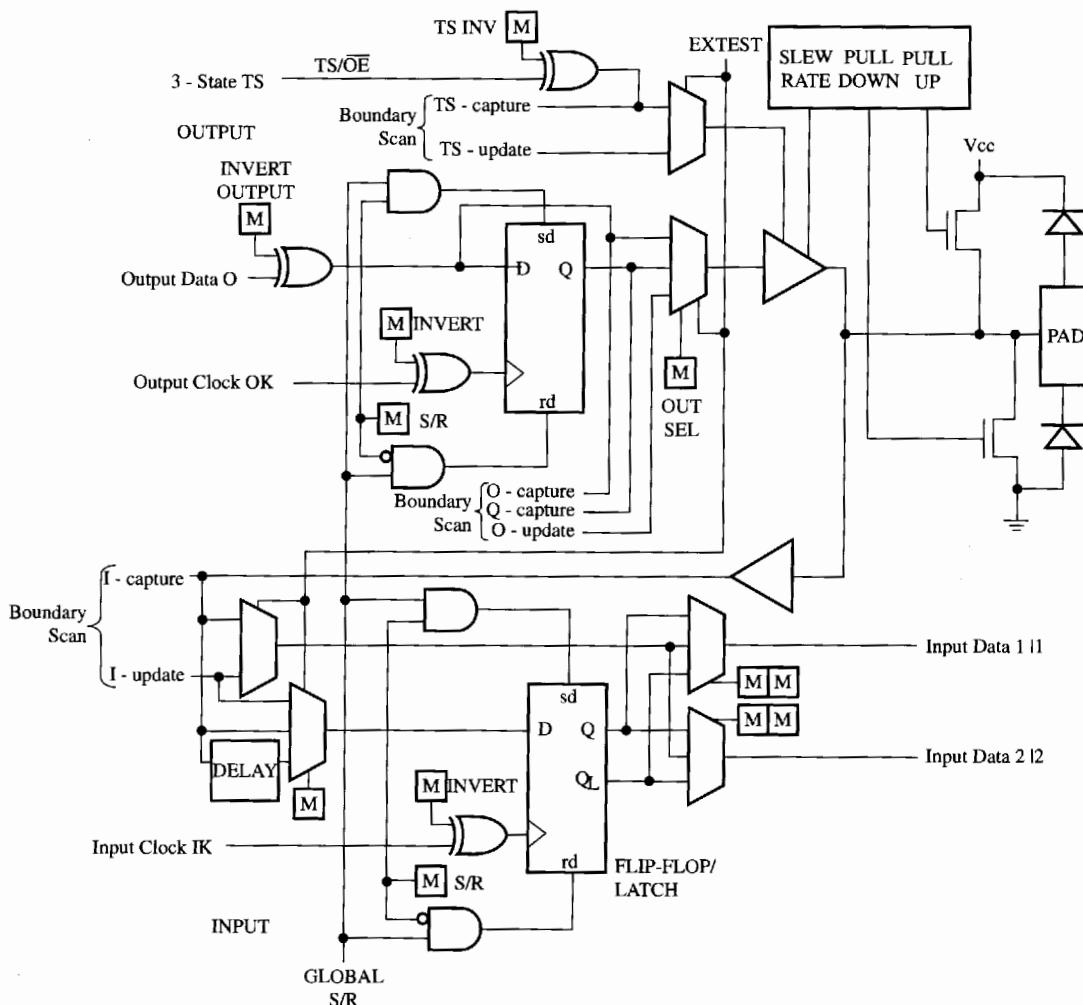


Figure 6-25 shows a 4000 series I/O block, which is similar in capability to the 3000 series I/O block. The Ms on the diagram represent configuration memory bits. As is the case for the 3000 IOB, the 4000 IOB can be programmed to invert the input, output, tristate buffer control, and clock input. In addition, the global S/R line can be programmed to set or reset each flip-flop. The 4000 IOB has both an optional pull-up and pull-down resistor connected to the I/O pad. The 4000 IOB also contains added logic for boundary scan testing, which is discussed in Chapter 10.

The XC4003 FPGA, which we have just described, has 100 CLBs ( $10 \times 10$ ), approximately 3000 gates, 80 user I/Os, 360 flip-flops (200 in the CLBs and 160 in the IOBs), and 45,636 configuration data bits. Other members of the XC4000 families have up to 2304 CLBs ( $48 \times 48$ ), 384 user I/Os, 5376 flip-flops, and more than one million configuration data bits.

Figure 6-25 4000 Series I/O Block



### VHDL Model for a 4000 Series CLB

We now develop a VHDL model for the basic 4000 series CLB, including the function generators and flip-flops but excluding the carry logic and RAM memory function. We then use this model to simulate a control state machine. In the port declaration (see Figure 6-26), the bit-vector *MEM\_BITS* represents the configuration memory bits, which determine the functions generated by the cell and the connections within the cell. *G\_IN* represents the inputs *G4*, *G3*, *G2*, and *G1*; *F\_IN* represents *F4*, *F3*, *F2*, and *F1*; *C\_IN* represents *C4*, *C3*, *C2*, and *C1*. The outputs are *Y*, *X*, and the two-bit vector *Q*, which represents the two flip-flop outputs (*YQ* and *XQ*).

**Figure 6-26 Behavioral Model for XC4000 CLB**

```
-- Behavioral description of the XC4000 CLB

library BITLIB;
use BITLIB.BIT_PACK.ALL;

entity XC4000CLB is
    port (MEM_BITS : in bit_vector(0 to 51);
          G_IN, F_IN, C_IN : in bit_vector(4 downto 1);
          K : in bit;
          Y,X : out bit;
          Q : out bit_vector (1 downto 0));
end XC4000CLB;

architecture behavior of XC4000CLB is

alias G_FUNC : bit_vector(0 to 15) is MEM_BITS(0 to 15);
alias F_FUNC : bit_vector(0 to 15) is MEM_BITS(16 to 31);
alias H_FUNC : bit_vector(0 to 7) is MEM_BITS(32 to 39);
type bv2D is array (1 downto 0) of bit_vector(1 downto 0);
constant FF_SEL : bv2D := (MEM_BITS(40 to 41),MEM_BITS(42 to 43));
alias Y_SEL : bit is MEM_BITS(44);
alias X_SEL : bit is MEM_BITS(45);
alias EDGE_SEL: bit_vector(1 downto 0) is MEM_BITS(46 to 47);
alias EC_SEL : bit_vector(1 downto 0) is MEM_BITS(48 to 49);
alias SR_SEL : bit_vector(1 downto 0) is MEM_BITS(50 to 51);

alias H1 : bit is C_IN(1);
alias DIN : bit is C_IN(2);
alias SR : bit is C_IN(3);
alias EC : bit is C_IN(4);

-- Timing spec for XC4000, Speed Grade -4
constant Tiho : TIME := 6 ns; -- F/G inputs to X/Y outputs via H
constant Tilo : TIME := 4 ns; -- F/G inputs to X/Y outputs
constant Tcko : TIME := 3 ns; -- Clock K to Q outputs
constant Trio : TIME := 7 ns; -- S/R to Q outputs

signal G,F,H : bit;

begin

    G <= G_FUNC (vec2int(G_IN));
    F <= F_FUNC (vec2int(F_IN));
    H <= H_FUNC (vec2int(H1&G&F)) after (Tiho-Tilo);
    X <= (X_SEL and H) or (not X_SEL and G) after Tilo;
    Y <= (Y_SEL and H) or (not Y_SEL and F) after Tilo;
```

```

process (K, SR)                                -- update FF outputs
variable DFF_EC,D : bit_vector(1 downto 0);
begin
  for i in 0 to 1 loop
    DFF_EC(i) := EC or EC_SEL(i);
    case FF_SEL(i) is
      when "00" => D(i) := DIN;
      when "01" => D(i) := F;
      when "10" => D(i) := G;
      when "11" => D(i) := H;
    end case;
    if (SR='1') then -- If SR set, then set or reset ff
      Q(i)<=SR_SEL(i) after Trio;
    else
      if (DFF_EC(i)='1') then -- If clock enabled then
        -- If correct triggering edge then update ff value
        if ((EDGE_SEL(i)='1' and rising_edge(K)) or
            (EDGE_SEL(i)='0' and falling_edge(K))) then
          Q(i)<=D(i) after Tcko;
        end if;
      end if;
    end if;
  end loop;
end process;
end behavior;

```

In order to make the VHDL code more readable, we use aliases to split up *MEM\_BITS* into its component parts. Within the cell, the functions generated by the function generators are stored in truth table form. *G\_FUNC* specifies the output column of the truth table for the function  $G(G4,G3,G2,G1)$ . Since *G* is a four-variable function, 16 bits are required. *F\_FUNC* is similar to *G\_FUNC*, but *H\_FUNC* requires only 8 bits, since *H* is a function of the three variables—*F*, *G*, and *H1*. *Y\_SEL* selects the *Y* output (1 selects *H* and 0 selects *G*). *X\_SEL* selects the *X* output (1 selects *H* and 0 selects *F*). *EDGE\_SEL*, *EC\_SEL*, and *SR\_SEL* select the clock edge, the clock enable, and the set or reset, respectively, for the two flip-flops. Since aliasing a bit-vector array to a bit-vector is not allowed, we have used a constant declaration to define *FF\_SEL*. *FF\_SEL(1)* selects the *D* input for flip-flop *YQ*, and *FF\_SEL(0)* selects the *D* input for flip-flop *XQ*.

We have included timing information in the model. The constant *Tilo* is the maximum propagation delay between a change in the *F/G* inputs and the *X/Y* outputs. When the output is via the *H* function generator, the maximum propagation delay is *Tih0*. The delay from clock *K* to outputs *Q* is *Tcko*, and the delay from *S/R* (*C3*) going high to *Q* is *Trio*. We have not included specifications for setup and hold times; VHDL code for checking these specifications will be discussed in Chapter 8.

The architecture body begins with concurrent statements that update *G*, *F*, *H*, *X*, and *Y*. The specification for individual function generator delays is not given in the data sheet, but these delays are included in *Tilo* when *X* or *Y* is updated. If the *H* function generator is used, the additional delay is (*Tih0* – *Tilo*).

The for loop within the process updates the flip-flop outputs whenever  $K$  or  $SR$  changes. The D flip-flop clock is enabled if  $EC\_SEL(i) = '1'$  or if the  $EC$  input ( $C4$ ) is '1'. The case statement represents the multiplexer, which selects the  $D(i)$  flip-flop input as  $DIN, F, G$ , or  $H$ , depending on the value of  $FF\_SEL(i)$ . When  $SR$  changes to '1',  $Q(i)$  is set or reset after the delay  $Trio$ . Otherwise,  $Q(i)$  is updated after  $Tcko$  if the clock is enabled and the selected clock edge has occurred.

To illustrate the use of the VHDL CLB model, we implement the multiplier control state graph of Figure 4-6(c) using XC4000 CLBs. Using a straight binary assignment, the logic equations are

$$QI^+ = K QI'Q0 + M QI'Q0 + K QI Q0' \quad (6-5a)$$

$$Q0^+ = St Q0' + M'QI'Q0 + QI Q0' \quad (6-5b)$$

$$Done = QI Q0 \quad (6-5c)$$

$$Load = St QI'Q0' \quad (6-5d)$$

$$Ad = M QI'Q0 \quad (6-5e)$$

$$Sh = M'QI'Q0 + QI Q0' \quad (6-5f)$$

Since each equation requires four variables or less, the equations can be implemented using three CLBs.

Figure 6-27 shows the VHDL code that instantiates three copies of the XC4000 CLB component and connects them to realize equations (6-5). Delays in the interconnections between the cells have not been included, since these delays are not known until after the cells have been placed and the interconnections routed. The constants  $MEM1$ ,  $MEM2$ , and  $MEM3$  specify the configuration memory bits for the CLBs. These bits are specified as constants, since they are loaded into the CLBs after power-up, and then they are not changed. The  $G$  inputs to CLB1 ( $G\_INI$ ) are  $G4 = K$ ,  $G3 = M$ ,  $G2 = QI$ , and  $G1 = Q0$ . The truth table for the  $G$  function generator, which implements  $QI^+$  (equation (6-5a)) is in Table 6-1. Based on this table, the first 16 bits of  $MEM1$  are "0000010001100110". The  $F$  function generator implements equation (6-5b), and the next 16 bits of  $MEM1$  are the truth table for  $F = Q0^+$  (see Table 6-1). Since the  $H$  function is not used, the next 8 bits are 0. The remaining bits of  $MEM1$  specify the cell configuration.

**Table 6-1 Truth Tables for G and F Function Generators**

(a)					(b)				
G4 K	G3 M	G2 Q1	G1 Q0	G Q1 <sup>+</sup>	F4 St	F3 M	F2 Q1	F1 Q0	F Q0 <sup>+</sup>
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	0	1	0	1
0	0	1	1	0	0	0	1	1	0
0	1	0	0	0	0	1	0	0	0
0	1	0	1	1	0	1	0	1	0
0	1	1	0	0	0	1	1	0	1
0	1	1	1	0	0	1	1	1	0
1	0	0	0	0	1	0	0	0	1
1	0	0	1	1	1	0	0	1	1
1	0	1	0	1	1	0	1	0	1
1	0	1	1	0	1	0	1	1	0
1	1	0	0	0	1	1	0	0	1
1	1	0	1	1	1	1	0	1	0
1	1	1	0	1	1	1	1	0	1
1	1	1	1	0	1	1	1	1	0

**Figure 6-27 XC4000 Implementation of Multiplier Control**

```

begin
  G_IN1<=K&M&Q; F_IN1<=St&M&Q;
  G_IN2<="00"&Q; F_IN2<=St&'0'&Q;
  G_IN3<=M&'0'&Q; F_IN3<=M&'0'&Q;

  CLB1: XC4000CLB port map (MEM1,G_IN1,F_IN1,"1000",CLK,open,open,Q);
  CLB2: XC4000CLB port map (MEM2,G_IN2,F_IN2,"1000",CLK,Done,Load,open);
  CLB3: XC4000CLB port map (MEM3,G_IN3,F_IN3,"1000",CLK,Ad,Sh,open);
end CLBs;

```

## 6.4 USING A ONE-HOT STATE ASSIGNMENT

When designing with PGAs, we should keep in mind that each logic cell contains two flip-flops. This means that it may not be important to minimize the number of flip-flops used in the design. Instead, we should try to reduce the total number of logic cells used and try to reduce the interconnections between cells. In order to design faster logic, we should try to reduce the number of cells required to realize each equation. Using a *one-hot state assignment* will often help to accomplish this.

The one-hot assignment uses one flip-flop for each state, so a state machine with  $N$  states requires  $N$  flip-flops. Exactly one flip-flop is set to 1 in each state. For example, a system with four states ( $T_0$ ,  $T_1$ ,  $T_2$ , and  $T_3$ ) could use four flip-flops ( $Q_0$ ,  $Q_1$ ,  $Q_2$ , and  $Q_3$ ) with the following state assignment:

$$T_0: Q_0\ Q_1\ Q_2\ Q_3 = 1000, \quad T_1: 0100, \quad T_2: 0010, \quad T_3: 0001 \quad (6-6)$$

The other 12 combinations are not used.

We can write next-state and output equations by inspection of the state graph or by tracing link paths on an SM chart. Consider the partial state graph given in Figure 6-28. The next-state equation for flip-flop  $Q_3$  could be written as

$$\begin{aligned} Q_3^+ &= X_1\ Q_0\ Q_1'\ Q_2'\ Q_3 + X_2\ Q_0'\ Q_1\ Q_2'\ Q_3' \\ &\quad + X_3\ Q_0'\ Q_1'\ Q_2\ Q_3 + X_4\ Q_0'\ Q_1'\ Q_2'\ Q_3 \end{aligned}$$

However, since  $Q_0 = 1$  implies  $Q_1 = Q_2 = Q_3 = 0$ , the  $Q_1'Q_2'Q_3'$  term is redundant and can be eliminated. Similarly, all the primed state variables can be eliminated from the other terms, so the next-state equation reduces to

$$Q_3^+ = X_1\ Q_0 + X_2\ Q_1 + X_3\ Q_2 + X_4\ Q_3$$

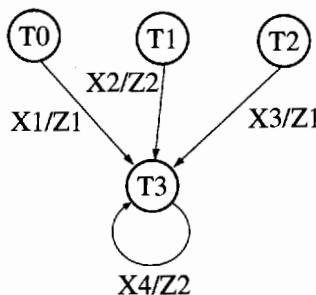
Note that each term contains exactly one state variable. Similarly, each term in each output equation contains exactly one state variable:

$$Z_1 = X_1\ Q_0 + X_3\ Q_2, \quad Z_2 = X_2\ Q_1 + X_4\ Q_3$$

When a one-hot assignment is used, the next-state equation for each flip-flop will contain one term for each arc leading into the corresponding state (or for each link path leading into the state). In general, each term in every next-state equation and in every output equation

will contain exactly one state variable. The one-hot state assignment for asynchronous networks is similar to that described above, but a “holding term” is required for each next-state equation (see *Fundamentals of Logic Design*, p. 644).

**Figure 6-28 Partial State Graph**



When a one-hot assignment is used, resetting the system requires that one flip-flop be set to 1 instead of resetting all flip-flops to 0. If the flip-flops used do not have a preset input (as is the case for the Xilinx 3000 series), then we can modify the one-hot assignment by replacing  $Q_0$  with  $Q_0'$  throughout. For the preceding assignment, the modification is

$$T0: Q0\bar{Q}1\bar{Q}2\bar{Q}3 = 0000, \quad T1: 1100, \quad T2: 1010, \quad T3: 1001 \quad (6-7)$$

and the modified equations are:

$$Q3^+ = X1\bar{Q}0' + X2\bar{Q}1 + X3\bar{Q}2 + X4\bar{Q}3$$

$$Z1 = X1\bar{Q}0' + X3\bar{Q}2, \quad Z2 = X2\bar{Q}1 + X4\bar{Q}3$$

Another way to solve the reset problem without modifying the one-hot assignment is to add an extra term to the equation for the flip-flop, which should be 1 in the starting state. As an example, we use the one-hot assignment given in (6-6) for the main dice game control of Figure 5-32(a). The next state equation for  $Q_0$  is

$$Q0^+ = Q0\bar{D}n\_roll' + Q2\bar{R}eset + Q3\bar{R}eset$$

If the system is reset to state 0000 after power-up, we can add the term  $Q0'Q1'Q2'Q3'$  to the equation for  $Q0^+$ . Then, after the first clock the state will change from 0000 to 1000 (T0), which is the correct starting state.

In general, both an assignment with a minimum number of state variables and a one-hot assignment should be tried to see which one leads to a design with the smallest number of logic cells. Alternatively, if speed of operation is important, the design that leads to the fastest logic should be chosen. When a one-hot assignment is used, more next-state equations are required, but in general both the next-state and output equations will contain fewer variables. An equation with fewer variables generally requires fewer logic cells to realize. Equations with five or fewer variables require a single cell. As seen in Figure 6-19, an equation with six variables may require cascading two cells, an equation with seven variables may require cascading three cells, etc. The more cells cascaded, the longer the propagation delay, and the slower the operation.

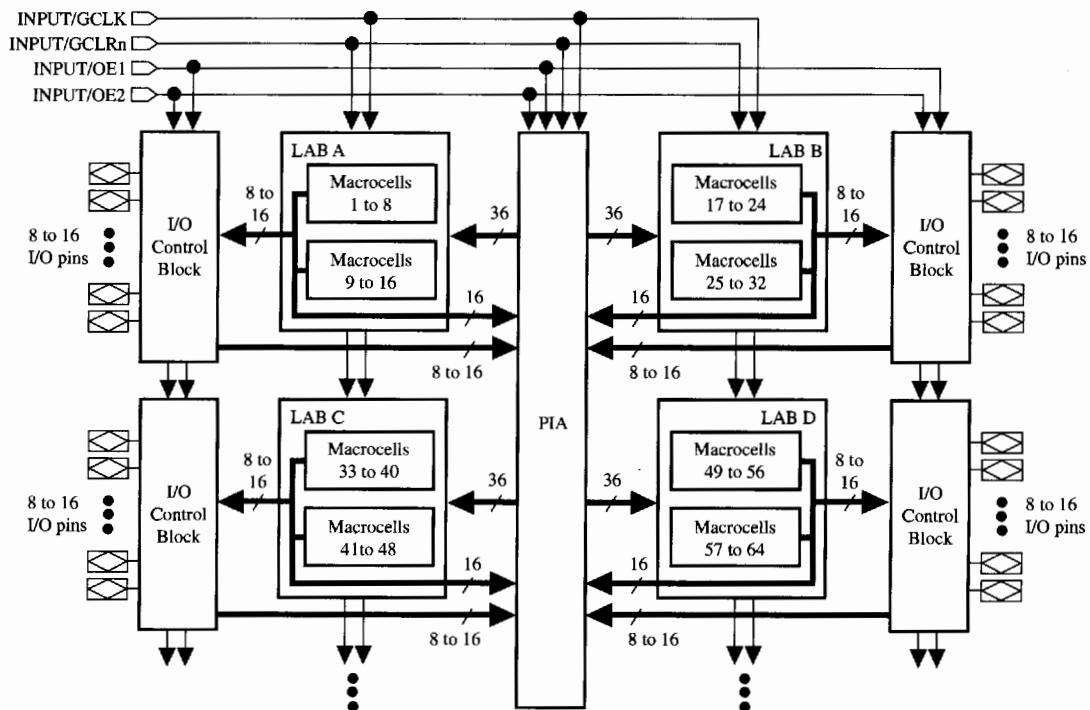
## 6.5 ALTERA COMPLEX PROGRAMMABLE LOGIC DEVICES (CPLDS)

CPLDs are an extension of the PAL concept. In general, a CPLD is an IC that consists of a number of PAL-like logic blocks together with a programmable interconnect matrix. Each PAL-like logic block has a programmable AND array that feeds macrocells, and the outputs of these macrocells can be routed to the inputs of other logic blocks within the same IC. Many CPLDs are electrically erasable and reprogrammable and, as such, are sometimes referred to as EPLDs (erasable PLDs).

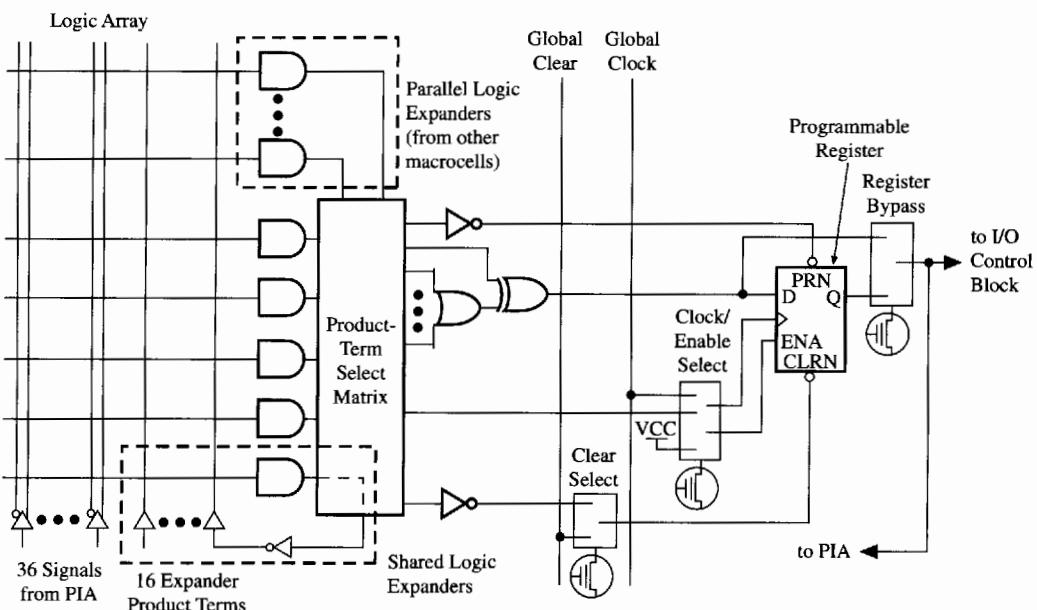
The Altera MAX 7000 series is a family of high-performance CMOS CPLDs. In contrast to the Xilinx FPGAs, the Altera 7000 series uses EEPROM-based configuration memory cells, so that once the configuration is programmed, it is retained until it is erased. Figure 6-29 shows the basic 7000 series architecture, which consists of a number of Logic Array Blocks (LABs), I/O Control Blocks, and a Programmable Interconnect Matrix (PIA). Each LAB contains 16 macrocells, each of which contains combinational logic and a flip-flop. Each LAB has 36 inputs from the PIA and 16 outputs to the PIA. From 8 to 16 outputs from each LAB can be routed to the I/O pins through the I/O control block. From 8 to 16 inputs from the I/O pins can be routed through the I/O control block to the PIA. The global clock input (*GCLK*) and global clear input (*GCLRn*) connect to all macrocells. Two output enable signals (*OE1n* and *OE2n*) connect to all I/O control blocks.

Each macrocell (Figure 6-30) includes a logic array, a product-term select matrix that feeds an OR gate, and a programmable flip-flop. The vertical lines in the logic array, which are common to all of the macrocells in a LAB, are driven with the programmable interconnect signals from the PIA and from shared logic expanders. Product terms are formed in the logic array just as they are in a PAL. Five product terms are provided in each macrocell, and these product terms are allocated by the product term select matrix. A product term may be used as an OR gate input, an XOR gate input, a logic expander, or as a flip-flop preset, clear, clock, or enable input.

**Figure 6-29** Altera 7000 Series Architecture for EPM7032, 7064, and 7096 Devices



**Figure 6-30** Macrocell for EPM7032, 7064, and 7096 Devices



The flip-flop in each macrocell is a D flip-flop with clock enable and asynchronous preset and clear. The clock input can be driven by the global clock or from a product term. The clock enable can be driven from a product term or V<sub>cc</sub> (always enabled). The clear can be driven from the global clear or from a product term. The preset can be driven from a product term. The *D* input always comes from the XOR gate output. Either the flip-flop *Q* output or the XOR gate output can be selected by the Register Bypass multiplexer. The selected output goes to the PIA and to the I/O control block. The D flip-flop in a cell can be converted to a T flip-flop using the XOR gate. Since the characteristic equation for a T flip-flop is  $Q^+ = Q \oplus T$ , we can connect one XOR gate input to *Q* and use the other input for *T*. Using T flip-flops to implement counters and adders often requires fewer gates than using D flip-flops.

Although only five product terms are available in each macrocell, more complex functions can be implemented by utilizing unused product terms from other macrocells. Two types of expander product terms are available—shareable expanders and parallel logic expanders. One of the product terms in each macrocell may be used as a shareable expander (Figure 6-31). The selected product term is fed back to the logic array through an inverter, and hence the inverted product term can be used as an input to any macrocell AND gate. When shareable expanders are used, the realization is equivalent to a three-level NAND-AND-OR network. An AND-OR logic expression with more than five terms can often be factored to utilize shareable expanders from other macrocells. For example,

$$\begin{aligned}P &= AB + B'C + C'D + E'F + E'G + E'H + F'I + F'J \\&= AB + B'C + C'D + E'(F'G'H')' + F'(I'J')'\end{aligned}$$

can use shareable expanders to generate  $(F'G'H')'$  and  $(I'J')'$ . The XOR gate in a cell can be used to complement a function, since  $F = F' \oplus 1$ . Sometimes the complement of a function ( $F'$ ) requires fewer terms than the original function ( $F$ ), in which case it is more economical to implement  $F'$  and complement it using the XOR gate.

Parallel expanders (Figure 6-32) allow unused product terms from a macrocell to be used in a neighboring macrocell. The parallel expander product terms can be chained from one macrocell to the next within two groups—macrocells 8 down to 1 and 16 down to 9. When parallel expanders are used without shareable expanders, the maximum number of product terms in any logic function is 20, five from the macrocell itself, and three additional groups of five chained from neighboring macrocells.

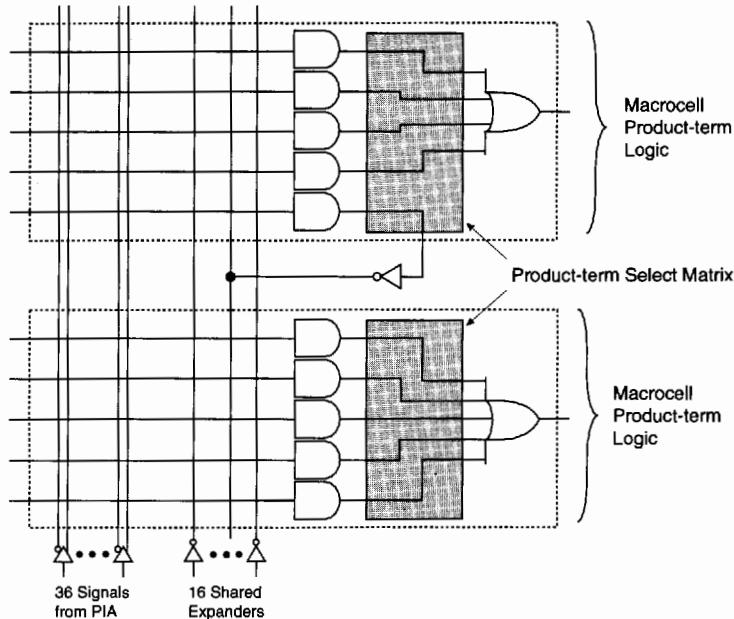
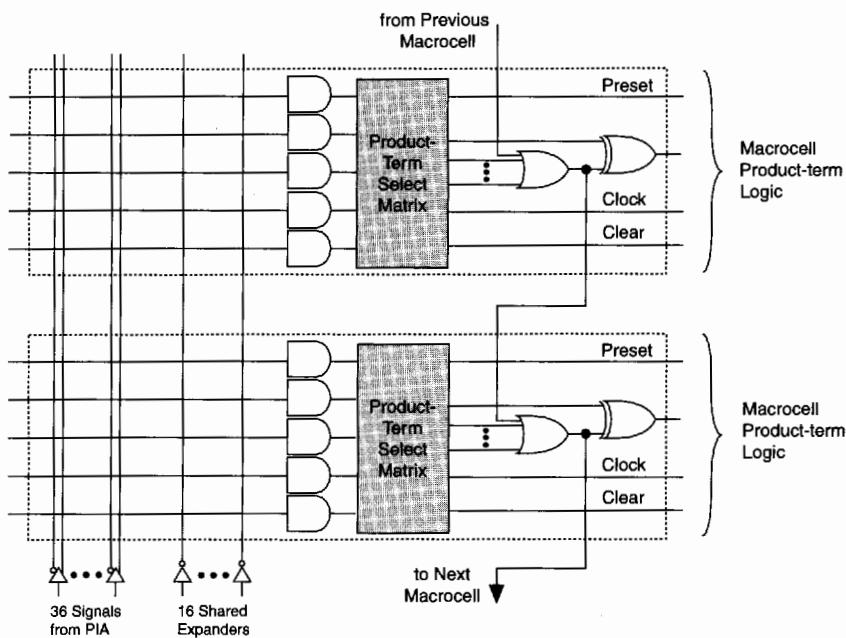
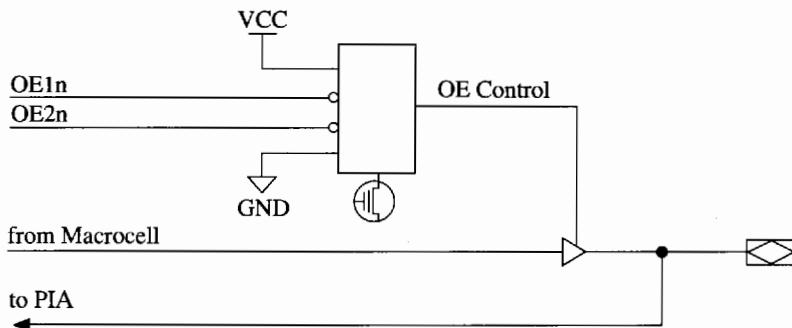
**Figure 6-31 Shareable Expanders****Figure 6-32 Parallel Expanders**

Figure 6-33 shows an I/O control block for an I/O pin. This block allows each I/O pin to be configured as an input, output, or bidirectional pin. A tristate buffer drives the I/O pin. The OE control mux is programmed to select either Vcc, Gnd, or one of the global output enable signals. If Vcc is selected, the macrocell output is enabled to the I/O pin. If Gnd is selected, the buffer is disabled and the I/O pin can be used as an input. Otherwise, the buffer is controlled by OE1n or OE2n.

**Figure 6-33 I/O Control Block for EPM 7032, 7064, and 7096**



Software provided by Altera can be used to optimize and partition a design to fit it into logic cells and route the connections between the cells. For example, if we use the Altera software to implement two bits of the full adder of Figure 3-21 using the equations given on page 109, the software first determines that T flip-flops will require fewer gates than D flip-flops and then it factors the equations to utilize shareable expanders. The resulting equations are

$$\begin{aligned}
 C3 &= A1 \ A2 \ X01 + B1 \ C1 \ X02 + A1 \ B2 \ X01 + A2 \ B2 \\
 \text{where } X01 &= B1+C1 \text{ and } X02 = A2+B2 \text{ are shareable} \\
 &\text{expander outputs} \\
 T2 &= Ad \ A1 \ B2' \ C1 + Ad \ B1 \ B2' \ X03 + Ad \ B1' \ B2 \ X04 + Ad \ A1' \ B2 \ C1' \\
 \text{where } X03 &= A1+C1 \text{ and } X04 = A1'+C1' \text{ are shareable} \\
 &\text{expander outputs} \\
 T1 &= Ad \ B1 \ C1' + Ad \ B1' \ C1
 \end{aligned}$$

Each logic equation has less than or equal to five terms, so it fits in a logic cell. Implementing these equations requires three logic cells and four shareable expanders.

Using the Altera software, we entered the schematics for the dice game (Figures 6-15 through 6-17) and compiled the design. The resulting equations require 23 logic cells and 8 shareable expanders, and they fit into an EPM7032 CPLD. More examples of using Altera CPLDs are given in Chapter 8.

Altera manufactures several other series of CPLDs. The MAX 7000S series is similar to the MAX 7000 series, except that it is in-circuit programmable rather than requiring a programmer. The MAX 9000 series, which is an enhanced version of the MAX 7000S series, has higher density and additional routing resources. The FLEX 8000 and FLEX 10K series use RAM-based configuration memory cells instead of EEPROM-based cells.

## 6.6 ALTERA FLEX 10K SERIES CPLDS

The Altera FLEX 10K embedded programmable logic family provides high-density logic along with RAM memory in each device. The logic and interconnections are programmed using configuration RAM cells in a manner similar to the Xilinx FPGAs. Figure 6-34 shows the block diagram for a FLEX 10K device. Each row of the logic array contains several logic array blocks (LABs) and an embedded array block (EAB). Each LAB contains eight logic elements and a local interconnect channel. The EAB contains 2048 bits of RAM memory. The LABs and EABs can be interconnected through fast row and column interconnect channels, referred to as FastTrack Interconnect. Each input-output element (IOE) can be used as an input, output, or bidirectional pin. Each IOE contains a bidirectional buffer and a flip-flop that can be used to store either input or output data. A single FLEX 10K device provides from 72 to 624 LABs, 3 to 12 EABs, and up to 406 IOEs. It can utilize from 10,000 to 100,000 equivalent gates in a typical application.

**Figure 6-34** FLEX 10K Device Block Diagram

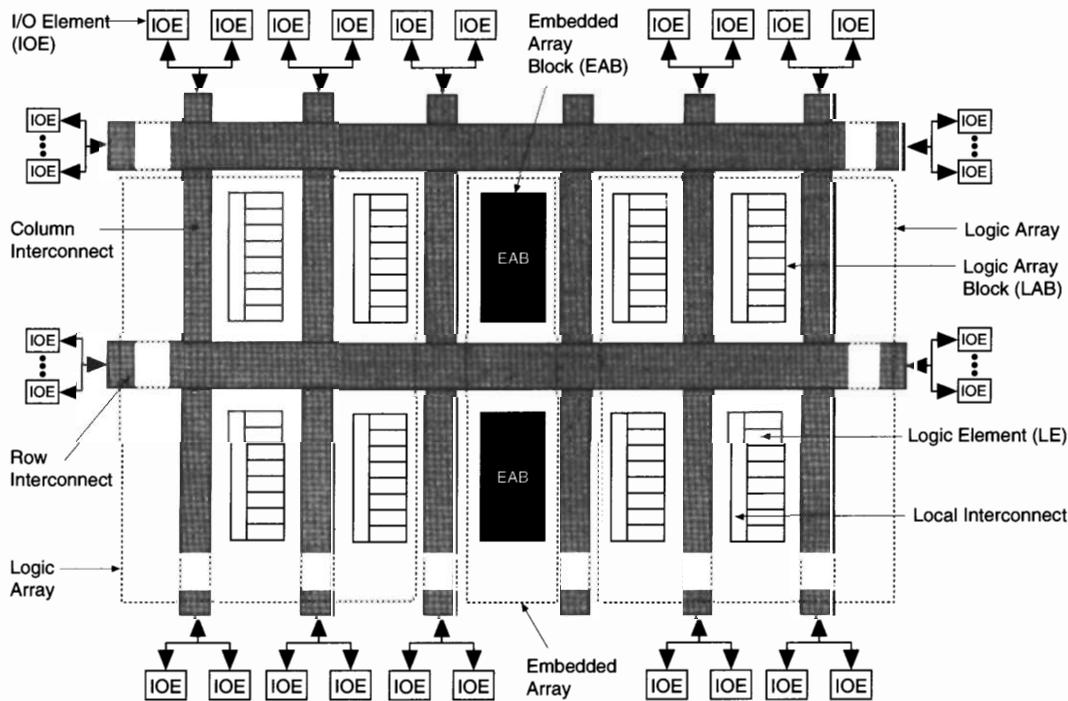
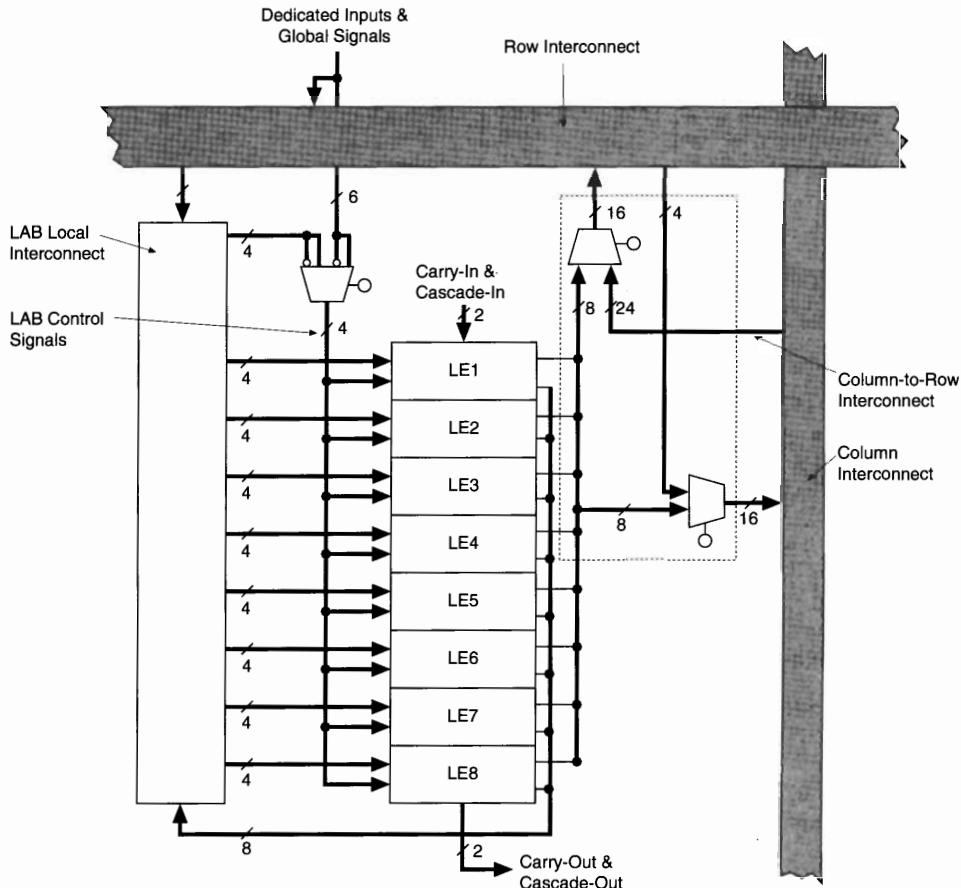


Figure 6-35 shows the block diagram for a FLEX 10K LAB that contains 8 logic elements (LEs). The local interconnect channel has 22 or more inputs from the row interconnect and 8 inputs fed back from the LE outputs. Each LE has four data inputs from the local interconnect channel as well as additional control inputs. The LE outputs can be routed to the row or column interconnects, and connections can also be made between the row and column interconnects.

Each logic element (Figure 6-36) contains a function generator that can implement any function of four variables using a lookup table (LUT). A cascade chain provides connections to adjacent LEs so functions of more than four variables can be implemented. The cascade chain can be used in an AND or in an OR configuration as illustrated in Figure 6-37.

**Figure 6-35 FLEX 10K Logic Array Block**



When used in the arithmetic mode, an LE can implement the sum and carry for one bit of a full adder. The carry chain provides for propagation of carries between adjacent cells. Each LE contains one D flip-flop with a clock enable and asynchronous clear and preset inputs. The LE output can come from the flip-flop or directly from the combinational logic.

Functions of more than four variables require multiple LEs for implementation. For example, a six-variable function,  $Z(a, b, c, d, e, f)$  can be implemented using six LEs. Applying the expansion theorem,

$$Z(a, b, c, d, e, f) = a'b'Z_0(c, d, e, f) + a'bZ_1(c, d, e, f) + ab'Z_2(c, d, e, f) + abZ_3(c, d, e, f)$$

$Z_0, Z_1, Z_2$ , and  $Z_3$  can each be implemented with an LE. The outputs of these LEs can be connected to inputs of other LEs via the local interconnect. The 4-variable functions  $Y_0 = a'b'Z_0 + a'bZ_1$  and  $Y_1 = ab'Z_2 + abZ_3$  each require another LE.  $Y_0$  can be ORed with  $Y_1$  using the cascade chain, so no additional LE is required.

Figure 6-36 FLEX 10K Logic Element

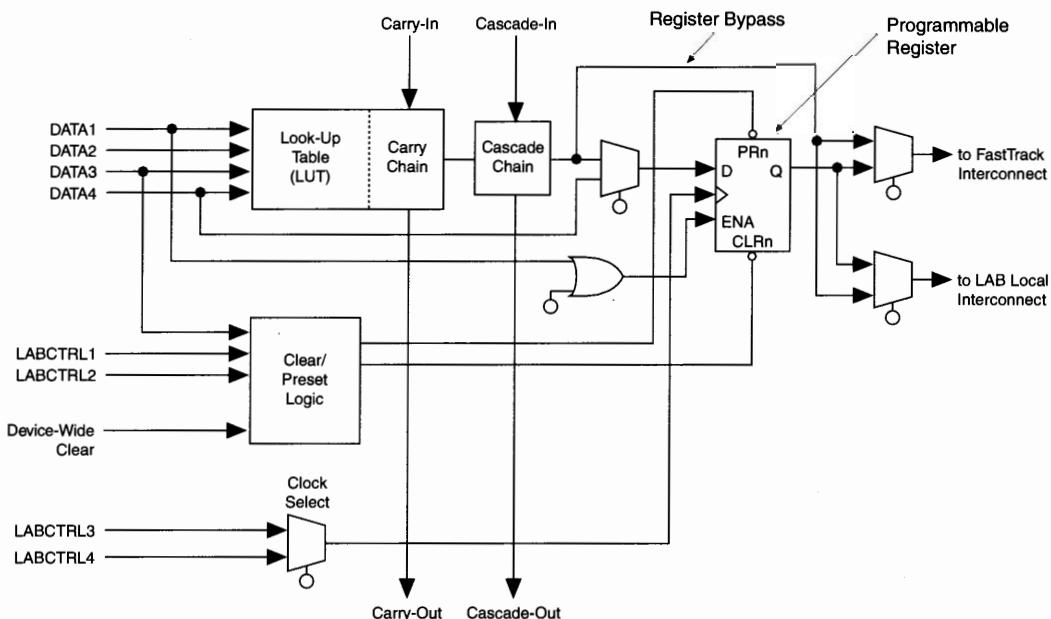
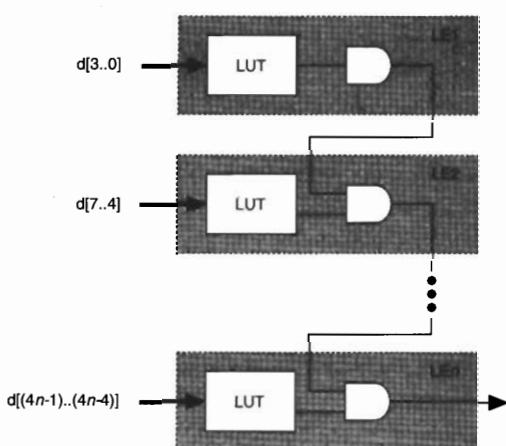


Figure 6-37 Cascade Chain Operation

AND Cascade Chain



OR Cascade Chain

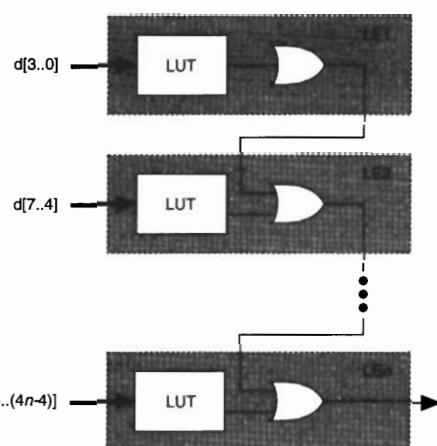
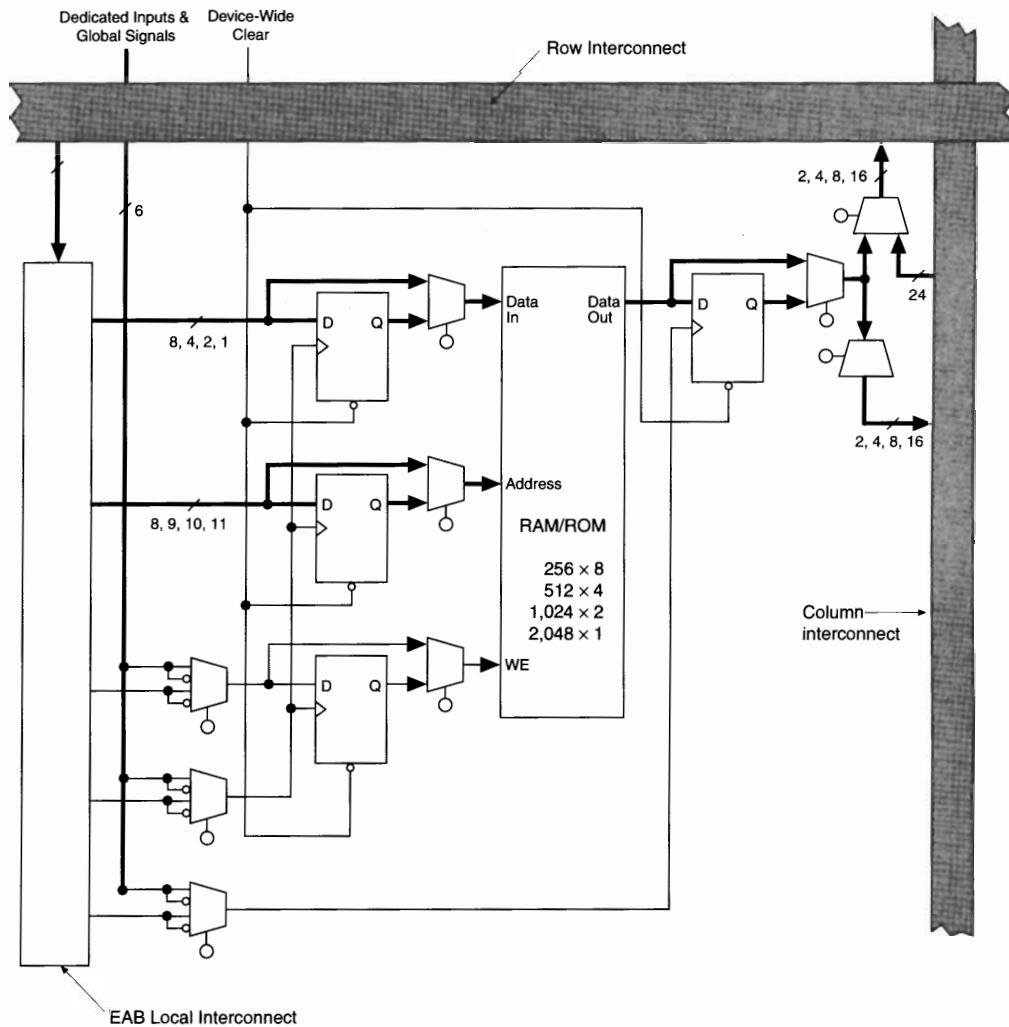


Figure 6-38 shows an embedded array block. The inputs from the row interconnect go through the EAB local interconnect and can be utilized as data inputs or address inputs to the EAB. The internal memory array can be used as a RAM or ROM of size  $256 \times 8$ ,  $512 \times 4$ ,  $1024 \times 2$ , or  $2048 \times 1$ . Several EABs can be used together to form a larger memory. The memory data outputs can be routed to either the row or column interconnects. All memory inputs and outputs are connected to registers so the memory can be operated in a synchronous mode. Alternatively, the registers can be bypassed and the memory operated asynchronously.

Use of CPLDs such as the FLEX 10K series allows us to implement a complex digital system using a single IC. An example of implementing a microcontroller with a FLEX 10K20 device is given in Chapter 11.

**Figure 6-38 FLEX 10K Embedded Array Block**



In this chapter we described several types of FPGAs and CPLDs and procedures for designing with these devices. Use of appropriate CAD software facilitates partitioning a design into logic blocks, placing these logic blocks within a logic array, and routing the connections between the blocks. In Chapter 8, we will introduce the use of synthesis tools, which allow us to start with a VHDL description of a digital system and synthesize digital logic to fit into a target FPGA or CPLD.

## Problems

**6.1** An 8-bit right shift register with parallel load is to be implemented using Xilinx 3000 logic cells. The flip-flops are labeled  $X_7X_6X_5X_4X_3X_2X_1X_0$ . The control signals  $N$  and  $S$  operate as follows:  $N = 0$ , do nothing;  $NS = 11$ , right shift;  $NS = 10$ , load. The serial input for right shift is  $SI$ .

- (a) How many logic cells are required?
- (b) Show the required connections for the rightmost cell on a copy of Figure 6-3.
- (c) Give the  $F$  and  $G$  equations for this cell.

**6.2** Is it possible to implement two J-K flip-flops with a single 3000 series logic cell? If not, explain why not. If yes, indicate the connections required on a copy of Figure 6-3, and give equations for  $F$  and  $G$ . Label cell inputs  $J1\ K1\ J2\ K2$ , outputs  $Q1\ Q2$ , etc. Draw heavy lines for all internal connections. *Hint:* The next-state equation for a J-K flip-flop is

$$Q^+ = JQ' + K'Q$$

**6.3** Implement a 2-bit binary counter using one 3000 series logic cell.  $Qx$  is the least significant bit, and  $Qy$  is the most significant bit of the counter. The counter has an asynchronous reset ( $AR$ ) and a synchronous load ( $Ld$ ). The counter operates as follows:

$En = 0$                   No change.

$En = 1, Ld = 1$  Load  $Qx$  and  $Qy$  with external inputs  $U$  and  $V$  on rising edge of clock.

$En = 1, Ld = 0$  Increment counter on rising edge of clock.

- (a) Give the next-state equations for  $Qx$  and  $Qy$ .
- (b) Show all required inputs and connections on a copy of Figure 6-3. Show the connection paths with heavy lines. Label the inputs on the FG mode diagram in Figure 6-4 and show the connection paths.

**6.4** Design a 4-bit right-shift register using a 3020 FPGA. When the register is clocked, the register loads if  $Ld = 1$  and  $En = 1$ , it shifts right when  $Ld = 0$  and  $En = 1$ , and nothing happens when  $En = 0$ .  $Si$  and  $So$  are the shift input and output of the register.  $D_{3,0}$  and  $Q_{3,0}$  are the parallel inputs and outputs, respectively. The next-state equation for the leftmost flip-flop is  $Q_3^+ = En'Q_3 + En(Ld D_3 + Ld' Si)$ .

- (a) Give the next-state equations for the other three flip-flops.
- (b) Determine the minimum number of 3000 series logic cells required to implement the shift register.
- (c) For the left cell, give the input connections and the internal paths on a copy of Figure 6-3. Also, give the  $F$  and  $G$  functions.

**6.5** Show how to realize the following combinational function using two 3000 series logic cells: Show the cell connections on a copy of Figure 6-3 and give the functions  $F$  and  $G$  for both cells.

$$F = X_1'X_2X_3'X_6 + X_2'X_3'X_4X_6' + X_2X_3'X_4' + X_2X_3X_4'X_6 + X_3'X_4X_5X_6' + X_7$$

**6.6** Realize the following next-state equation using a 3020 FPGA. Try to minimize the number of logic cells required. Draw a diagram that shows the connections to the logic cells (Figure 6-3) and give the functions  $F$  and  $G$  for each cell. (The equation is already in minimum form.) It is not necessary to show the individual gates.

$$Q^+ = UQV'W + U'Q'VX'Y' + UQX'Y + U'Q'V'Y + U'Q'XY + UQVW' + U'Q'V'X$$

### 6.7

(a) Make a one-hot state assignment for the state graph of Figure 4-22. By inspection, derive the next-state and output equations.

(b) If a 3000 series FPGA is used, estimate the number of logic cells required to implement the state graph. Repeat for a 4000 series FPGA.

**6.8** How many 3000 series I/O blocks with unused IO pads would be required to construct an 8-bit serial in-parallel out register? When  $IN$  is 1, data is shifted into the register on a clock edge, and the parallel outputs are always enabled. Draw the connections for the first I/O block.

**6.9** What is the minimum number of 4000 series logic cells required to realize the following function?

$$X = X_1'X_2'X_3'X_4'X_5 + X_1X_2X_3X_4X_5 + X_5'X_6X_7X_8'X_9 + X_5'X_6'X_7X_8X_9'$$

If your answer is 1, show the required input connections on a copy of Figure 6-20, and also mark the internal connection paths with heavy lines. If your answer is greater than 1, draw a block diagram showing the cell inputs and interconnections between cells. In any case, give the functions to be realized by each F, G, and H block.

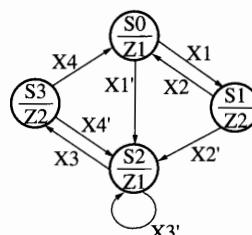
**6.10** For the given state graph:

(a) Derive the simplified next-state and output equations by inspection. Use the following one-hot state assignment for flip-flops  $Q_0Q_1Q_2Q_3$ : S0, 1000; S1, 0100; S2, 0010; S3, 0001.

(b) How many 4000 series CLBs are required to implement these equations?

(c) Refer to the behavioral VHDL model for the XC4000 CLB for this part in Figure 6-26. For the CLB that implements  $Q_0$  and  $Q_1$ , specify the following:  $G\_FUNC$ ,  $FF\_SEL$ ,  $SR\_SEL$ .

(d) Assuming a maximum interconnect delay of 2 ns between CLBs, what is the maximum clock rate for this implementation? Setup time for the flip-flop  $D$  inputs measured from the function generator inputs is 6 ns, and propagation delay from clock input to  $Q$  output is 5ns.



**6.11** Consider a 4-to-1 multiplexer. Can it be implemented using only one 4000 logic cell? If it can be implemented, show the configuration of the 4000 CLB (Figure 6-20) and give the equations for  $F$ ,  $G$ , and  $H$ . If not, show or explain why it cannot be done. What about a 3-to-1 multiplexer?

**6.12** Given  $Z(T, U, V, W, X, Y) = VW'X + U'V'WY + TV'WY'$ ,

(a) Show how  $Z$  can be realized using a single 4000 series logic cell. Show the cell inputs on a copy of Figure 6-20, indicate the internal connections in the cell, and specify the functions  $F$ ,  $G$ , and  $H$ . Use  $Y$  for the  $H1$  input.

(b) Show how  $Z$  can be realized using two 3000 series logic cells. Draw a diagram showing the inputs to each cell, the interconnections between cells, and the  $F$  and  $G$  functions for each cell.

**6.13** Complete the following VHDL model for a XC3000 logic cell (Figure 6-3). Assume that a procedure is available,  $FG\_GEN(A,B,C,D,E,QX,QY,F,G)$ , that returns the values of  $F$  and  $G$  when it is called. Ignore propagation delays. For each programmable 2-to-1 MUX, the top input is selected when  $M = 0$ . For each programmable 3-to-1 MUX,  $F$  is selected when  $MM = 00$ ,  $DIN$  is selected when  $MM = 01$ , and  $G$  is selected when  $MM = 10$ .  $M$  is the bit array used to program the logic cell. Label the control inputs on the top F-DIN-G MUX as  $M(0)$  and  $M(1)$ . Label the other  $M$  bits in a similar manner.

```
entity XC3000 is
    port (M: in bit_vector(0 to 8);
          A, B, C, D, E, DI, EC, K, DR, GR: in bit;
          X, Y: out bit);
end XC3000;
```

**6.14** A  $4 \times 4$  array multiplier (Figure 4-7) is to be implemented using an XC4003 FPGA.

(a) Without using the built-in carry logic, partition the logic so that it fits in a minimum number of logic cells. Draw loops around each set of components that will fit in a single logic cell. Determine the total number of  $F$  and  $G$  function generators required.

(b) Repeat part (a), except use the built-in carry logic.

### 6.15

(a) Implement an 8-to-1 multiplexer using an Altera 7000 series device. Give the logic equations and determine the number of macrocells required if parallel expanders are used.

(b) Repeat (a), except use shareable expanders and no parallel expanders.

(c) Implement the MUX using a FLEX 10K device. How many logic elements are required?

### 6.16

(a) Implement a 6-bit counter using an Altera 7000 series device. Give the logic equations and determine the number of macrocells required. Use the XOR gates to create T flip-flops.

(b) Implement the counter using a FLEX 10K device. Use the carry chain so that each logic element can implement a sum and carry.

# CHAPTER 7

## FLOATING-POINT ARITHMETIC

Floating-point numbers are frequently used for numerical calculations in computing systems. Arithmetic units for floating-point numbers are considerably more complex than those for fixed-point numbers. This chapter first describes a simple representation for floating-point numbers. Next an algorithm for floating-point multiplication is developed and tested using VHDL. Then the design of the floating-point multiplier is completed and implemented using an FPGA. Floating-point addition, subtraction, and division are also briefly described.

### 7.1 REPRESENTATION OF FLOATING-POINT NUMBERS

A simple representation of a floating-point (or real) number ( $N$ ) uses a binary fraction ( $F$ ) and exponent ( $E$ ), where  $N = F \times 2^E$ . We will represent negative fractions and exponents in 2's complement form. (Refer to Section 4.4 for a discussion of 2's complement fractions.) In a typical floating-point number system,  $F$  is 16 to 64 bits long and  $E$  is 8 to 15 bits. In order to keep the examples in this chapter simple and easy to follow, we will use a 4-bit fraction and a 4-bit exponent, but the concepts presented here are easily extended to more bits. Examples of floating-point numbers using a 4-bit fraction and 4-bit exponent are

$$F = 0.101$$

$$E = 0101$$

$$N = 5/8 \times 2^5$$

$$F = 1.011$$

$$E = 1011$$

$$N = -5/8 \times 2^{-5}$$

$$F = 1.000$$

$$E = 1000$$

$$N = -1 \times 2^{-8}$$

In order to utilize all the bits in  $F$  and have the maximum number of significant figures,  $F$  should be normalized so that its magnitude is as large as possible. If  $F$  is not normalized, we can normalize  $F$  by shifting it left until the sign bit and the next bit are different. Shifting  $F$  left is equivalent to multiplying by 2, so every time we shift we must decrement  $E$  by 1 to keep  $N$  the same. After normalization, the magnitude of  $F$  will be as large as possible, since any further shifting would change the sign bit. In the following examples,  $F$  is unnormalized to start with and then it is normalized by shifting left.

Unnormalized:	$F = 0.0101$	$E = 0011$	$N = 5/16 \times 2^3 = 5/2$
Normalized:	$F = 0.101$	$E = 0010$	$N = 5/8 \times 2^2 = 5/2$
Unnormalized:	$F = 1.11011$	$E = 1100$	$N = -5/32 \times 2^{-4} = -5 \times 2^{-9}$
(shift $F$ left)	$F = 1.1011$	$E = 1011$	$N = -5/16 \times 2^{-5} = -5 \times 2^{-9}$
Normalized:	$F = 1.011$	$E = 1010$	$N = -5/8 \times 2^{-6} = -5 \times 2^{-9}$

Zero cannot be normalized, so  $F = 0.000$  when  $N = 0$ . Any exponent could then be used; however, it is best to have a uniform representation of 0. We will associate the negative exponent with the largest magnitude with the fraction 0. In a 4-bit 2's complement integer number system, the most negative number is 1000, which represents  $-8$ . Thus when  $F$  and  $E$  are 4 bits, 0 is represented by

$$F = 0.000, \quad E = 1000, \quad \text{or } 0.000 \times 2^{-8}$$

This is logical, since the smallest nonzero positive number that could be represented is  $0.001 \times 2^{-8}$ . Some floating-point systems use a biased exponent, so  $E = 0$  is associated with  $F = 0$ .

IEEE has established a standard for floating-point numbers that provides a uniform way of storing floating-point numbers in computer systems. However, most floating-point arithmetic units convert the IEEE notation to 2's complement and then use the 2's complement internally for carrying out the floating-point operations. Then the final result is converted back to IEEE standard notation.

## 7.2 FLOATING-POINT MULTIPLICATION

In this section we design a multiplier for floating-point numbers. We use 4-bit fractions and 4-bit exponents, with negative numbers represented in 2's complement. Given two floating-point numbers, the product is

$$(F_1 \times 2^{E_1}) \times (F_2 \times 2^{E_2}) = (F_1 \times F_2) \times 2^{(E_1+E_2)} = F \times 2^E$$

The fraction part of the product is the product of the fractions, and the exponent part of the product is the sum of the exponents. We assume that  $F_1$  and  $F_2$  are properly normalized to start with, and we want the final result to be normalized.

Basically, all we have to do is multiply the fractions and add the exponents. However, several special cases must be considered. First, if  $F$  is 0, we must set the exponent  $E$  to the largest negative value (1000). Second, if we multiply  $-1$  by  $-1$  ( $1.000 \times 1.000$ ), the result should be  $+1$ . Since we cannot represent  $+1$  as a 2's complement fraction, we call this special case a *fraction overflow*. To correct this situation, we set  $F = 1/2$  (0.100) and add 1 to  $E$ . This is justified, since  $1 \times 2^E = 1/2 \times 2^{E+1}$ .

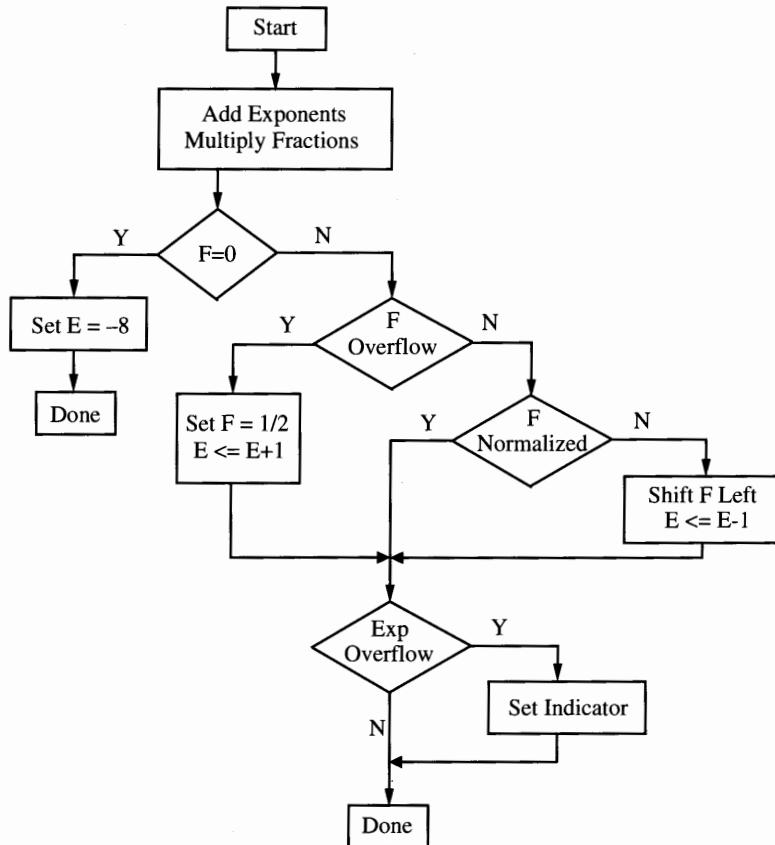
When we multiply the fractions, the result could be unnormalized. For example,

$$(0.1 \times 2^{E_1}) \times (0.1 \times 2^{E_2}) = 0.01 \times 2^{E_1+E_2} = 0.1 \times 2^{E_1+E_2-1}$$

In this example, we normalize the result by shifting the fraction left one place and subtracting 1 from the exponent to compensate. Finally, if the resulting exponent is too large in magnitude to represent in our number system, we have an exponent overflow. (Sometimes, an overflow in the negative direction is referred to as an *underflow*.) Since we are using 4-bit exponents, if the exponent is not in the range 1000 to 0111 (-8 to +7), an overflow has occurred. Since an exponent overflow cannot be corrected, an overflow indicator should be turned on.

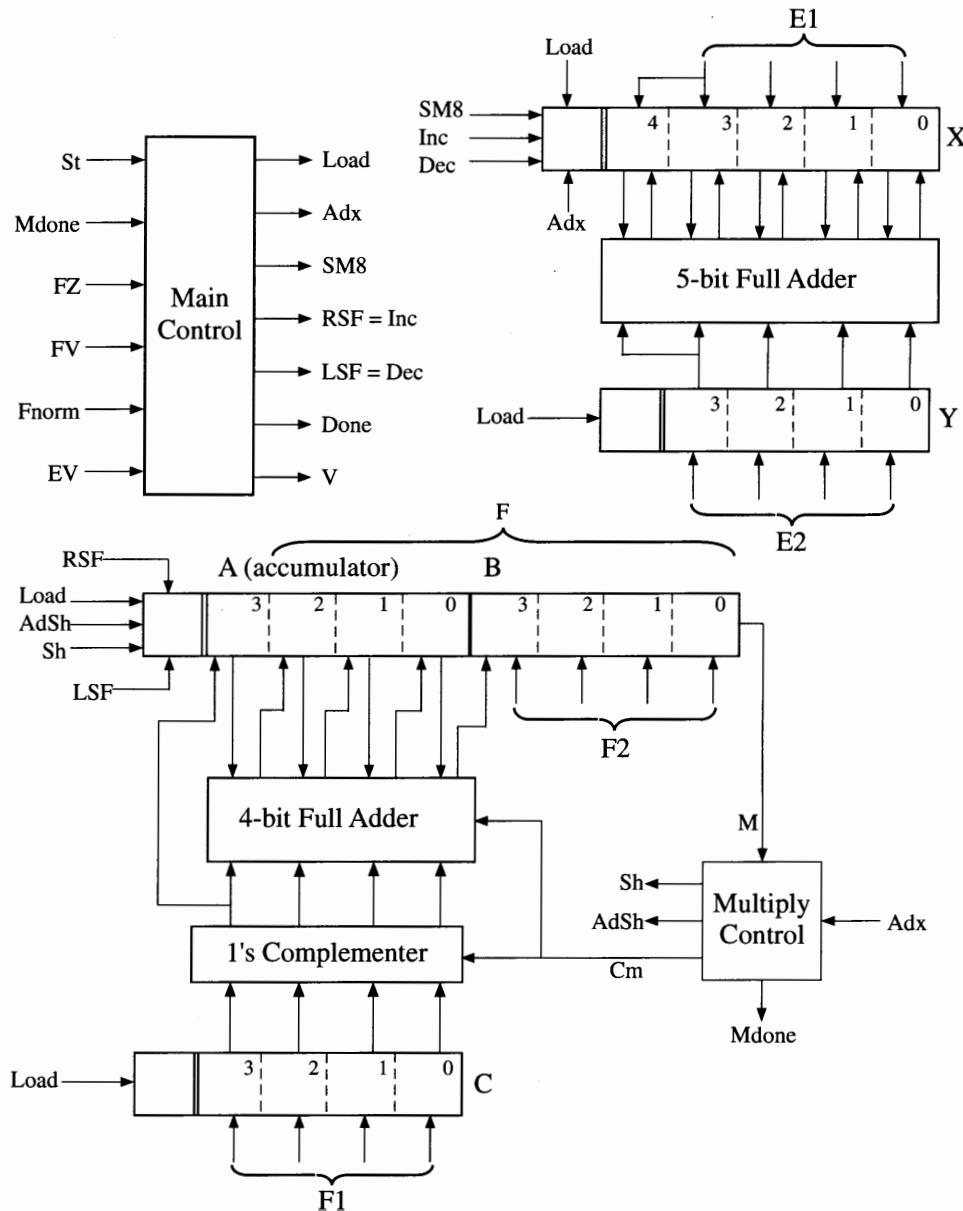
A flowchart for the floating-point multiplier is shown in Figure 7-1. After the fraction multiply is completed, all the special cases must be tested for. Since  $F_1$  and  $F_2$  are normalized, the smallest possible magnitude for the product is 0.01, as indicated in the preceding example. Therefore, only one left shift is required to normalize  $F$ .

**Figure 7-1** Flowchart for Floating-Point Multiplication



The hardware required to implement the multiplier (Figure 7-2) consists of an exponent adder and a fraction multiplier. The latter is similar to the 2's complement multiplier of Figure 4-10. Since we are multiplying 3 bits plus sign by 3 bits plus sign, the result will be 6 bits plus sign. After the fraction multiply, the 7-bit result ( $F$ ) will be the lower 3 bits of  $A$  concatenated with  $B$ .

**Figure 7-2 Exponent Adder and Fraction Multiplier**



When the exponents are added, an overflow can occur. If  $E_1$  and  $E_2$  are positive and the sum ( $E$ ) is negative, or if  $E_1$  and  $E_2$  are negative and the sum is positive, the result is a 2's complement overflow. However, this overflow might be corrected when 1 is added to or subtracted from  $E$  during normalization or correction of fraction overflow. To allow for this case, we have made the  $X$  register 5 bits long. When  $E_1$  is loaded into  $X$ , the sign bit must be extended so that we have a correct 2's complement representation. Since there are two sign bits, if the addition of  $E_1$  and  $E_2$  produces an overflow, the lower sign bit will get changed, but the high-order sign bit will be unchanged. Each of the following examples has an overflow, since the lower sign bit has the wrong value:

$$7 + 6 = 00111 + 00110 = 01101 = 13 \quad (\text{maximum allowable value is } 7)$$

$$-7 + (-6) = 11001 + 11010 = 10011 = -13 \quad (\text{most negative allowable value is } -8)$$

The following example illustrates the special case where an initial fraction overflow and exponent overflow occurs, but the exponent overflow is corrected when the fraction overflow is corrected:

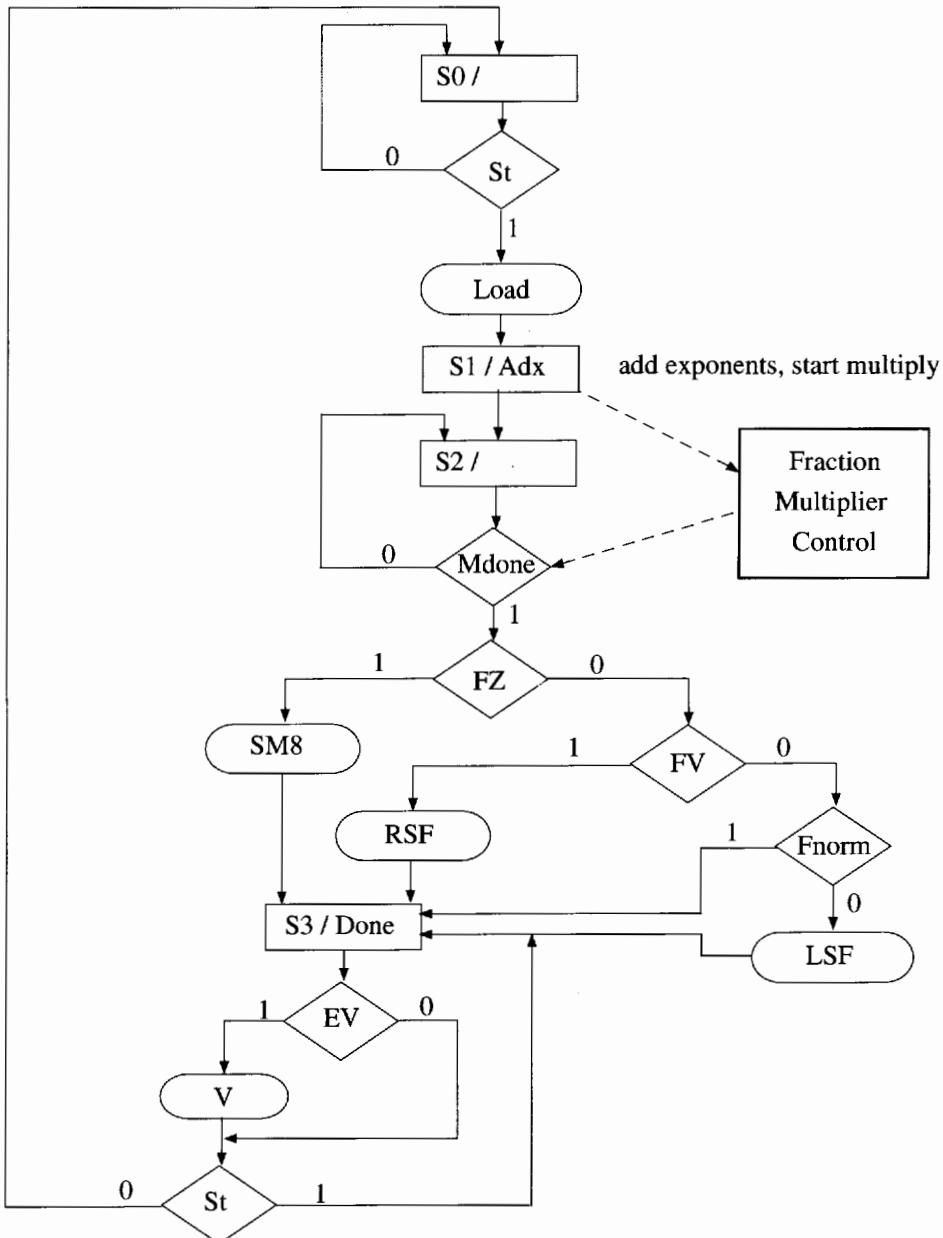
$$(1.000 \times 2^{-3}) \times (1.000 \times 2^{-6}) = 01.000000 \times 2^{-9} = 00.100000 \times 2^{-8}$$

The SM chart for the main controller (Figure 7-3) of the floating-point multiplier is based on the flowchart. The controller for the multiplier is a separate state machine, which is linked into the main controller. The SM chart uses the following inputs and control signals:

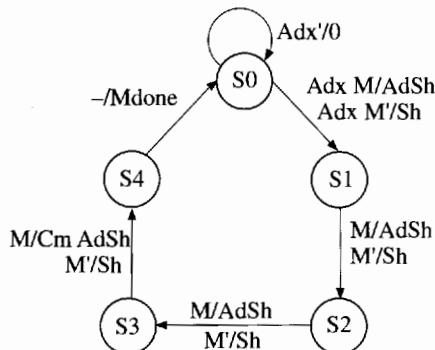
St	Start the floating-point multiplication.
Mdone	Fraction multiply is done.
FZ	Fraction is zero.
FV	Fraction overflow.
Fnorm	$F$ is normalized.
EV	Exponent overflow.
Load	Load $F_1, E_1, F_2, E_2$ into the appropriate registers (also clear $A$ in preparation for multiplication).
Adx	Add exponents; this signal also starts the fraction multiplier.
SM8	Set exponent to minus 8.
RSF	Shift fraction right; also increment $E$ .
LSF	Shift fraction left; also decrement $E$ .
V	Overflow indicator.
Done	Floating-point multiplication is complete.

The SM chart for the main controller has four states. In S0, the registers are loaded when the start signal is 1. In S1, the exponents are added, and fraction multiply is started. In S2, we wait until the fraction multiply is done and then test for special cases and take appropriate action. It may seem surprising that the tests on FZ, FV, and Fnorm can all be done in the same state, since they are done in sequence on the flowchart. However, FZ, FV, and Fnorm are generated by combinational circuits that operate in parallel and hence can be tested in the same state. However, we must wait until the exponent has been incremented or decremented at the next clock before we can check for exponent overflow in S3. In S3, the Done signal is turned on and the controller waits for St = 0 before returning to S0.

Figure 7-3 SM Chart for Floating-Point Multiplication



The state graph for the multiplier control (Figure 7-4) is similar to Figure 4-11, except the load state is not needed, because the registers are loaded by the main controller. When  $Adx = 1$ , the multiplier is started, and  $Mdone$  is turned on when the multiplication is completed.

**Figure 7-4 State Graph for Multiplier Control**

The VHDL behavioral description (Figure 7-5) uses three processes. The main process generates control signals based on the SM chart. A second process generates the control signals for the fraction multiplier. The third process tests the control signals and updates the appropriate registers on the rising edge of the clock. In state S2 of the main process,  $A = "0000"$  implies that  $F = 0$  ( $FZ = 1$  on SM chart). If we multiply  $1.000 \times 1.000$ , the result is  $A \& B = "01000000"$ , and a fraction overflow has occurred ( $FV = 1$ ). If  $A(2) = A(1)$ , the sign bit of  $F$  and the following bit are the same and  $F$  is unnormalized ( $F_{norm} = 0$ ). In state S3, if the two high-order bits of  $X$  are different, an exponent overflow has occurred ( $EV = 1$ ).

The registers are updated in the third process. The variable *addout* represents the output of the 4-bit full adder, which is part of the fraction multiplier. This adder adds the 2's complement of *C* to *A* when *Cm* = 1. When *Load* = 1, the sign-extended exponents are loaded into *X* and *Y*. When *Adx* = 1, the procedure *Addvec* (Figure 2-23) is called to add the 5-bit vectors *X* and *Y*. The constant '0' indicates that *Cin* is 0, and *NC* is a dummy output signal that is set equal to the carry-out but not used. When *SM8* = 1, -8 is loaded into *X*. *Addvec* is also called to increment or decrement *X* by adding +1 or -1. When *AdSh* = 1, *A* is loaded with the sign bit of *C* (or the complement of the sign bit if *Cm* = 1) concatenated with bits 3 downto 1 of the adder output, and the remaining bit of *addout* is shifted into the *B* register.

Testing the VHDL code for the floating-point multiplier must be done carefully to account for all the special cases in combination with positive and negative fractions, as well as positive and negative exponents. Figure 7-6 shows a command file and some test results. This is not a complete test.

Figure 7-5 VHDL Code for Floating-Point Multiplier

```

library BITLIB;
use BITLIB.bit_pack.all;

entity FMUL is
    port (CLK, St: in bit;
          F1,E1,F2,E2: in bit_vector(3 downto 0);
          F: out bit_vector(6 downto 0);
          V, done:out bit);
end FMUL;

architecture FMULB of FMUL is
signal A, B, C: bit_vector(3 downto 0);           --fraction registers
signal X, Y: bit_vector(4 downto 0);               --exponent registers
signal Load, Adx, Mdone, SM8, RSF, LSF, NC: bit;
signal AdSh, Sh, Cm: bit;
signal PS1, NS1: integer range 0 to 3;            -- present and next state
signal State, Nextstate: integer range 0 to 4; -- multiplier control state
alias M: bit is B(0);
constant one:bit_vector(4 downto 0):="00001";
constant neg_one:bit_vector(4 downto 0):="11111";
begin
main_control: process
begin
    Load <= '0'; Adx <= '0';                      --clear control signals
    SM8 <= '0'; RSF <= '0'; LSF <= '0';
    case PS1 is
        when 0 => done<='0'; V<='0';           --clear outputs
        if St = '1' then Load <= '1'; NS1 <= 1; F <= "0000000";
        end if;
        when 1 => Adx <= '1'; NS1 <= 2;
        when 2 =>
            if Mdone = '1' then                  --wait for multiply
                if A = "0000" then             --zero fraction
                    SM8 <= '1';
                elsif A = "0100" and B = "0000" then --fraction overflow
                    RSF <= '1';                 --shift AB right
                elsif A(2) = A(1) then        --test for unnormalized
                    LSF <= '1';                 --shift AB left
                end if;
                NS1 <= 3;
            end if;
        when 3 =>                                --test for exp overflow
            if X(4) /= X(3) then V <= '1'; else V <= '0'; end if;
            done <= '1';
            F <= A(2 downto 0) & B;           --output fraction
            if ST = '0' then NS1<=0; end if;
    end case;
    wait until rising_edge(CLK);
    PS1 <= NS1;
    wait for 0 ns;                            --wait for state change
end process main_control;

```

```

mul2c: process                               --2's complement
multiply
begin
  AdSh <= '0'; Sh <= '0'; Cm <= '0';           --clear control signals
  case State is
    when 0=> Mdone <= '0';                   --start multiply
    if Adx='1' then
      if M = '1' then AdSh <= '1'; else Sh <= '1'; end if;
      Nextstate <= 1;
    end if;
    when 1 | 2 =>                           --add/shift state
      if M = '1' then AdSh <= '1'; else Sh <= '1'; end if;
      Nextstate <= State + 1;
    when 3 =>
      if M = '1' then Cm <= '1'; AdSh <= '1'; else Sh <='1'; end if;
      Nextstate <= 4;
    when 4 =>
      Mdone <= '1'; Nextstate <= 0;
  end case;
  wait until rising_edge(CLK);
  State <= Nextstate;                      --wait for state change
  wait for 0 ns;
end process mul2c;

update: process                             --update registers
variable addout: bit_vector(4 downto 0);
begin
  wait until rising_edge(CLK);
  if Cm = '0' then addout := add4(A,C,'0');
  else addout := add4(A, not C,'1'); end if; --add 2's comp. of C
  if Load = '1' then X <= E1(3)&E1; Y <= E2(3)&E2;
    A <= "0000"; B <= F2; C <= F1; end if;
  if ADX = '1' then addvec(X,Y,'0',X,NC,5); end if;
  if SM8 = '1' then X <= "11000"; end if;
  if RSF = '1' then A <= '0'&A(3 downto 1);
    B <= A(0)&B(3 downto 1);
    addvec(X,one,'0',X,NC,5); end if;          -- increment X
  if LSF = '1' then
    A <= A(2 downto 0)&B(3); B <= B(2 downto 0)&'0';
    addvec(X,neg_one,'0',X,NC,5); end if;       -- decrement X
  if AdSh = '1' then
    A <= (C(3) xor Cm) & addout(3 downto 1);   -- load shifted adder
    B <= addout(0) & B(3 downto 1); end if;      -- output into A & B
  if Sh = '1' then
    A <= A(3) & A(3 downto 1);                  -- right shift A & B
    B <= A(0) & B(3 downto 1);                  -- with sign extend
  end if;
end process update;
end FMULB;

```

**Figure 7-6 Test Data and Simulation Results for Floating-Point Multiply**

```

list f x f1 e1 f2 e2 v done
force f1 0111 0, 1001 200, 1000 400, 0000 600, 0111 800
force e1 0001 0, 1001 200, 0111 400, 1000 600, 0111 800
force f2 0111 0, 1001 200, 1000 400, 0000 600, 1001 800
force e2 1000 0, 0001 200, 1001 400, 1000 600, 0001 800
force st 1 0, 0 20, 1 200, 0 220, 1 400, 0 420, 1 600, 0 620, 1 800, 0 820
force clk 0 0, 1 10 -repeat 20
run 1000

ns delta f      x      f1    e1    f2    e2    v done
 0   +0 0000000 00000 0000 0000 0000 0000 0 0
 0   +1 0000000 00000 0111 0001 0111 1000 0 0 (0.111 x 21) x (0.111 x 2-8)
10   +1 0000000 00001 0111 0001 0111 1000 0 0
30   +1 0000000 11001 0111 0001 0111 1000 0 0
150  +2 0110001 11001 0111 0001 0111 1000 0 1 = 0.110001 x 2-7
170  +2 0000000 11001 0111 0001 0111 1000 0 0
200  +0 0000000 11001 1001 1001 1001 0001 0 0 (1.001 x 2-7) x (1.001 x 21)
250  +1 0000000 11010 1001 1001 1001 0001 0 0
370  +2 0110001 11010 1001 1001 1001 0001 0 1 = 0.110001 x 2-6
390  +2 0000000 11010 1001 1001 1001 0001 0 0
400  +0 0000000 11010 1000 0111 1000 1001 0 0 (1.000 x 27) x (1.000 x 2-7)
430  +1 0000000 00111 1000 0111 1000 1001 0 0
450  +1 0000000 00000 1000 0111 1000 1001 0 0
570  +1 0000000 00001 1000 0111 1000 1001 0 0
570  +2 0100000 00001 1000 0111 1000 1001 0 1 = 0.100000 x 21
590  +2 0000000 00001 1000 0111 1000 1001 0 0
600  +0 0000000 00001 0000 1000 0000 1000 0 0 (0.000 x 2-8) x (0.000 x 2-8)
630  +1 0000000 11000 0000 1000 0000 1000 0 0
650  +1 0000000 10000 0000 1000 0000 1000 0 0
770  +1 0000000 11000 0000 1000 0000 1000 0 0
770  +2 0000000 11000 0000 1000 0000 1000 0 1 = 0.0000000 x 2-8
790  +2 0000000 11000 0000 1000 0000 1000 0 0
800  +0 0000000 11000 0111 0111 1001 0001 0 0 (0.111 x 27) x (1.001 x 21)
830  +1 0000000 00111 0111 0111 1001 0001 0 0
850  +1 0000000 01000 0111 0111 1001 0001 0 0
970  +2 1001111 01000 0111 0111 1001 0001 1 1 = 1.001111 x 28 (overflow)
990  +2 0000000 01000 0111 0111 1001 0001 0 0

```

After the VHDL code has been thoroughly tested, we complete the logic design and then implement the multiplier using a programmable gate array. To facilitate loading the registers, we use an input data bus, as shown in Figure 7-7. Instead of loading all the registers at the same time, we load *F1*, *F2*, *E1*, and *E2* at successive clock times using the input data bus. When we add the exponents, we must load *E1* from the exponent adder output, and we use the same bus for this purpose. To avoid bus conflict, we use tristate buffers for the input data and for the exponent adder output. During loading, the input tristate buffers are enabled, and during exponent addition, the other tristate buffers are enabled. Alternatively, we could use a quad 2-to-1 multiplexer at the *E1* register input and eliminate the tristate buffers. Use of the multiplexer would require two additional logic

cells and would increase the propagation delay; therefore, use of the tristate buffers is probably the best choice if sufficient tristate buffers and associated horizontal long lines are available. The VHDL code of Figure 7-5 was written in terms of conditional register transfers and did not use either multiplexers or tristate buffers. The tristate bus could be explicitly included in the VHDL model by using techniques described in Section 8.4.

**Figure 7-7 Bus Structure for Floating-Point Multiplier**

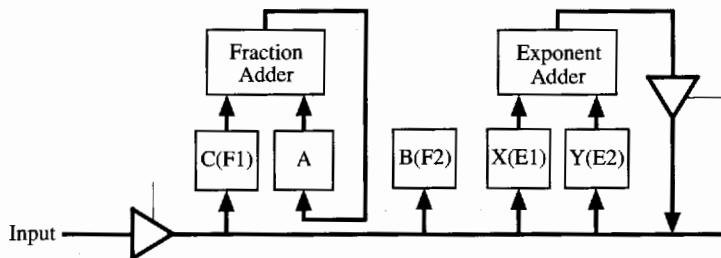


Figure 7-8 shows a high-level schematic diagram for the multiplier. This diagram includes the registers and adders that implement the fraction multiplier and exponent adder of Figure 7-2 together with the multiplier control and main control modules. This design is set up so the multiplier can be tested using a limited number of input switches. This requires a modification of the SM chart—the load signal is eliminated and instead an external load button (*LD*) is used to load each register. The four data inputs (*D*<sub>3</sub>, *D*<sub>2</sub>, *D*<sub>1</sub>, *D*<sub>0</sub>) are connected to the data bus (*DB*<sub>3</sub>, *DB*<sub>2</sub>, *DB*<sub>1</sub>, *DB*<sub>0</sub>) through tristate buffers. Switches *SW*<sub>0</sub> and *SW*<sub>1</sub> and the multiplexer (*D*<sub>2</sub> – *4E*) generate the signals *L*<sub>1</sub>, *L*<sub>2</sub>, *L*<sub>3</sub>, and *L*<sub>4</sub>, which select the register to be loaded from the data inputs when the *LD* button is pressed.

We will use the following one-hot assignment for the main controller:

$$\text{S0: } Q_0Q_1Q_2Q_3 = 1000, \quad \text{S1: } 0100, \quad \text{S2: } 0010, \quad \text{S3: } 0001$$

The following simplified logic equations can be obtained by inspection of the SM chart:

$$Q_0^+ = Q_0St' + Q_3St' \text{ or } Q_0^+ = Q_1'Q_2'St'$$

$$Q_1^+ = Q_0St$$

$$Q_2^+ = Q_2Mdone' + Q_1$$

$$Q_3^+ = Q_3St + Q_2Mdone$$

$$CLR = Q_0 \quad (\text{clear A})$$

$$Adx = Q_1$$

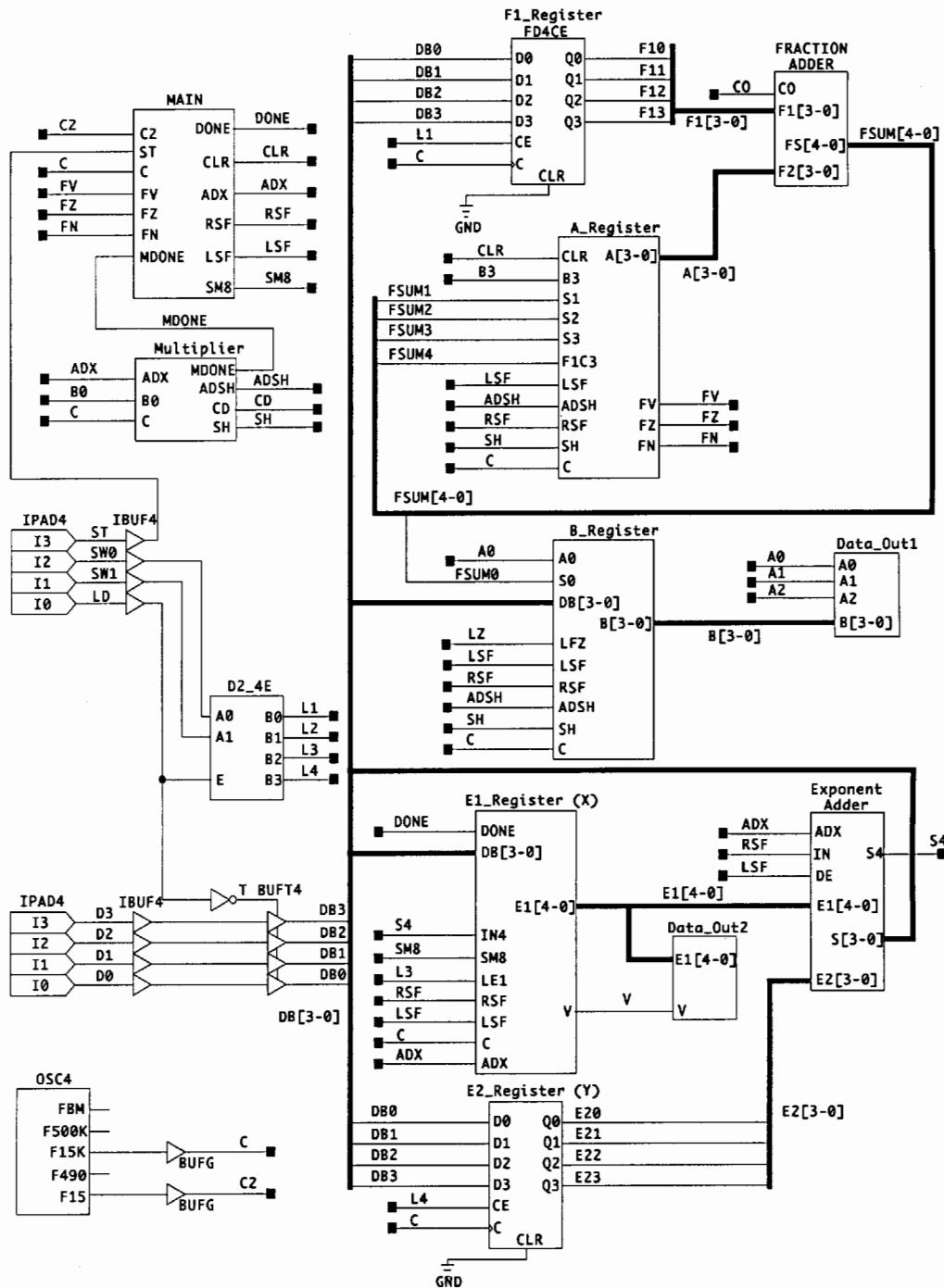
$$RSF = Q_2Mdone FZ'FV$$

$$LSF = Q_2Mdone FV'FZ'Fnorm'$$

$$SM8 = Q_2MdoneFZ$$

$$Done = Q_3$$

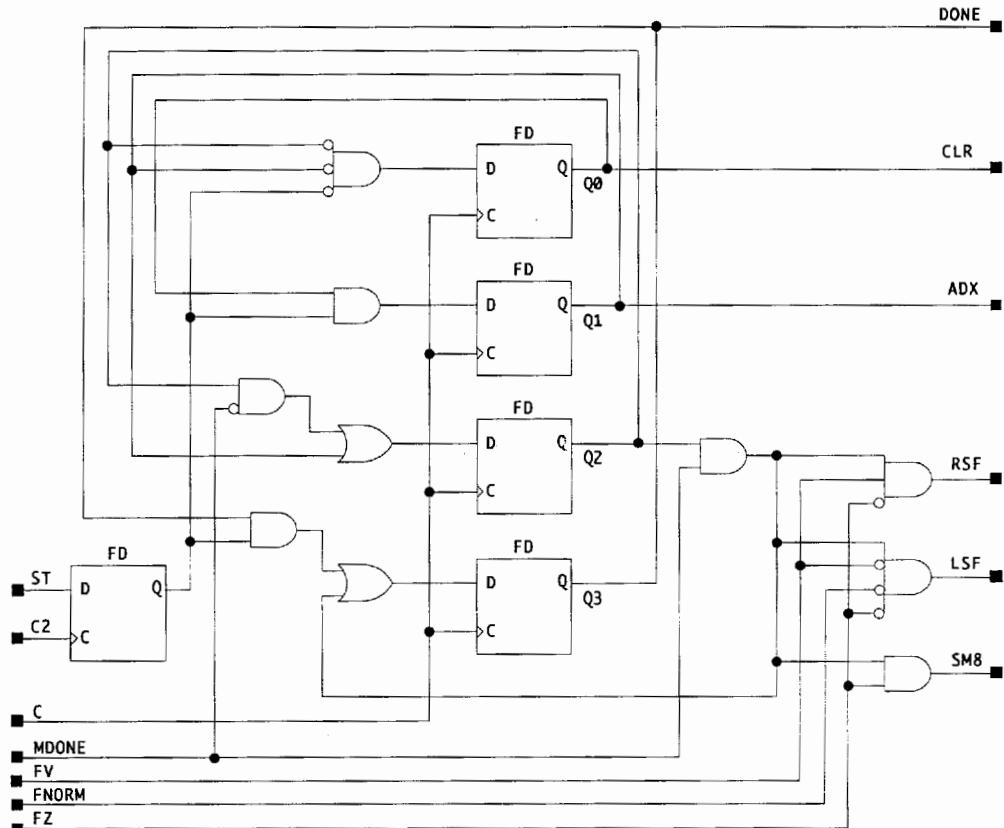
Figure 7-8 Top-level Schematic for Floating-Point Multiplier



The second equation for  $Q_0^+$  was obtained from a Karnaugh map by noting that if  $St = 0$ , then  $Q_0^+ = 1$  in states 1000 and 0001,  $Q_0^+ = 0$  in states 0100 and 0010, and  $Q_0^+$  is a don't care in the remaining unused states.

Using the second equation for  $Q_0^+$ , these equations are implemented in the main control module (see Figure 7-9). When the network is reset, it will actually start in state 0000 instead of 1000, but after the first clock  $Q_0$  will become 1. The extra flip-flop on the left side of the control module synchronizes the start signal with the clock.

**Figure 7-9 Main Control for Floating-Point Multiplier**



We will implement the multiplier control of Figure 7-4 using the following modified one-hot assignment :

$$S0: Q_1 Q_2 Q_3 Q_4 = 0000, \quad S1: 1000, \quad S2: 0100, \quad S3: 0010, \quad S4: 0001$$

The resulting logic equations are

$$Q_1^+ = AdxQ_1'Q_2'Q_3'Q_4' \text{ (since } Q_1 = Q_2 = Q_3 = Q_4 = 0 \text{ only in S0)}$$

$$Q_2^+ = Q_1$$

$$Q_3^+ = Q_2$$

$$Q_4^+ = Q_3$$

$$AdSh = M(Adx + Q_1 + Q_2 + Q_3)$$

$$Sh = M'(Adx + Q_1 + Q_2 + Q_3)$$

$$Cm = MQ_3$$

$$Mdone = Q_4$$

Since  $Q_1^+ = Q_2^+ = Q_3^+ = Q_4^+ = 0$  in S4, the next state of S4 is S0. The preceding equations are implemented in the multiplier control module (see Figure 7-10.)

**Figure 7-10 Multiplier Control**

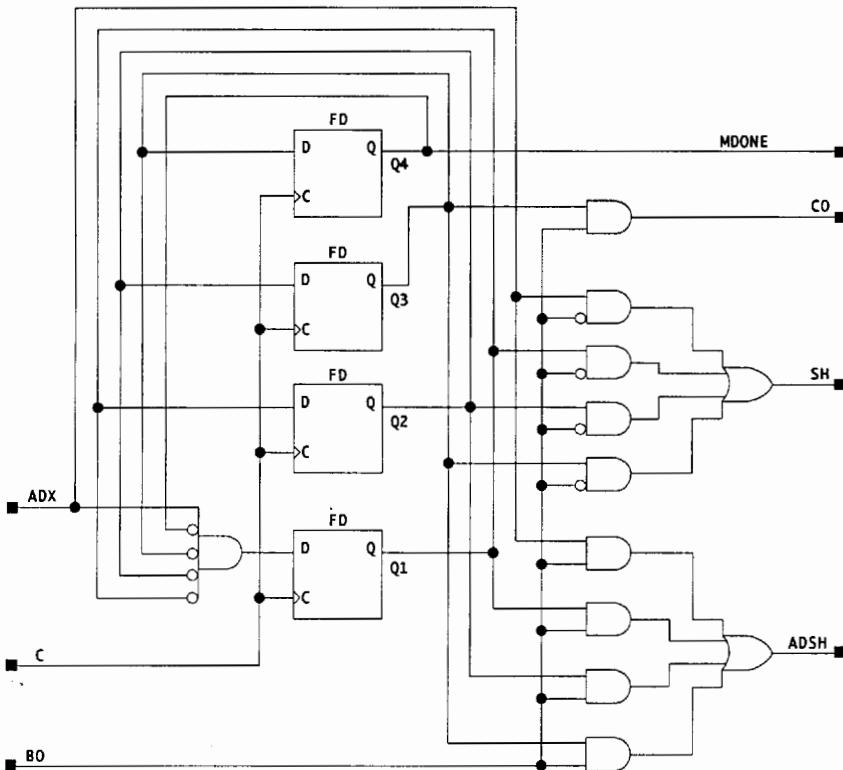


Figure 7-11 shows the implementation of the A register module. The X74-194 is a bidirectional shift register with control inputs  $S_0$  (left shift) and  $S_1$  (right shift). If both  $S_0$  and  $S_1$  are 1, a parallel load occurs. Left shift occurs when  $LSF = 1$ , and right shift occurs when  $RSF = 1$  or  $Sh = 1$ . When shifting right, the serial input ( $LSI$ ) is 0 when  $RSF = 1$ , else it is  $A_3$  (sign extend). When  $AdSh = 1$ ,  $S_0$  and  $S_1$  are both 1, so the adder output (offset by one place) is loaded into the register. This module also generates the condition signals  $FV$ ,  $FZ$ , and  $FN$  (fraction normalized). When fraction overflow occurs,  $A = 0100$ , so  $FV = A_3'A_2A_1'A_0'$ . Since we assume that  $F_1$  and  $F_2$  are normalized if nonzero, if product bits  $A_2 = A_1 = A_0 = 0$ , this implies that the product is 0, so  $FZ = A_2'A_1'A_0'$ . If the sign bit ( $A_2$ ) and following bit ( $A_1$ ) are complements,  $F$  is normalized, so  $FN = A_2 \oplus A_1$ .

**Figure 7-11 A Register**

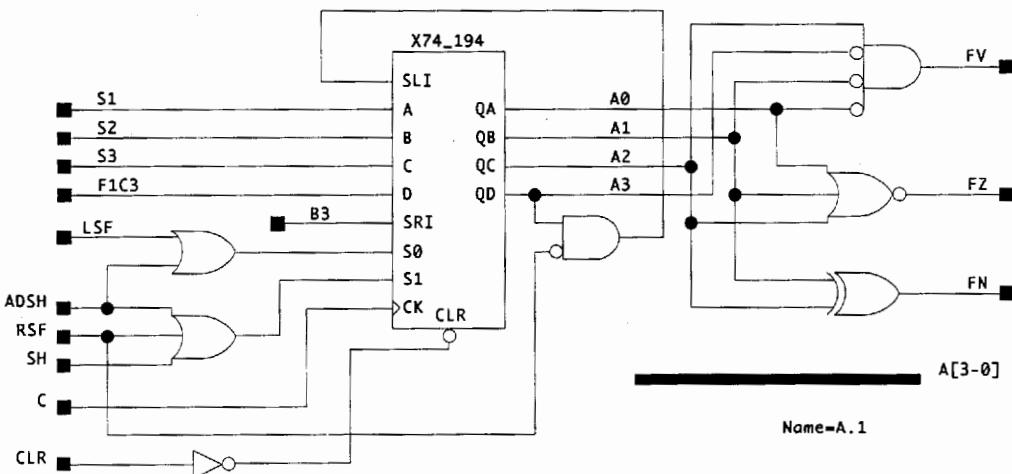


Figure 7-12 shows the implementation of the  $E1$  register module. The RD5CR is a 5-bit register with a clock enable. This register was implemented by modifying a standard 4-bit register from the library. When  $SM8 = 0$ , the register is loaded from the data bus ( $DB$ ).  $D4$  is loaded from the MUX output, so  $D4$  is either  $IN4$  or a copy of the sign bit ( $DB_3$ ). When  $SM8 = 1$ , the AND and OR gates driving the  $D$  inputs force 11000 to be loaded into the register. The register is loaded when  $SM8$ ,  $Ld1$ ,  $Adx$ ,  $Dec$ , or  $Inc$  is 1.

The exponent adder (Figure 7-13) adds  $E1$  and  $E2$  when  $Adx = 1$ . When  $Inc = 1$ , 0001 is added to  $E1$  (increment), and when  $Dec = 1$ , 1111 (-1) is added (decrement). A 4-bit adder plus an XOR gate is used to generate the 5-bit sum. Four bits from the  $E1$  bus go into the ADD4 module, and the fifth bit ( $E14$ ) goes into the XOR gate. The adder output is enabled onto the data bus when  $Adx$ ,  $Dec$ , or  $Inc$  is 1.

The design was placed and routed using the XACT software package from Xilinx using the 4003 FPGA as the target. The following results indicate the utilization of the device:

Occupied CLBs	50 out of 100
<i>F</i> and <i>G</i> function generators	67 out of 200
<i>H</i> function generators	17 out of 100
CLB flip-flops	30 out of 200
Three-state buffers	8 out of 240
CLB fast carry logic	7 out of 100

Since less than half of the available resources were used, it should be possible to design a floating-point multiplier with an 8-bit fraction and 8-bit exponent using the same part.

Now that the basic design has been completed, we need to determine how fast the floating-point multiplier will operate and determine the maximum clock frequency. Most CAD tools provide a way of simulating the final circuit taking into account both the delays within the logic blocks and the interconnection delays. If this timing analysis indicates that the design does not operate fast enough to meet specifications, several options are possible. Most FPGAs come in several different speed grades, so one option is to select a faster part. Another approach is to determine the longest delay path in the circuit and attempt to reroute the connections or redesign that part of the circuit to reduce the delays.

Figure 7-12 *E1* Register

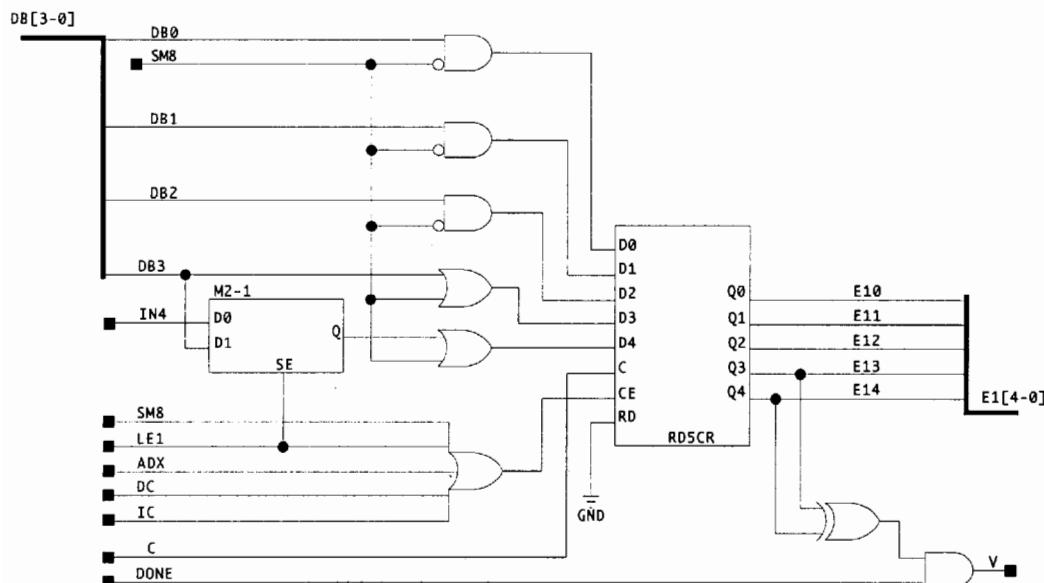
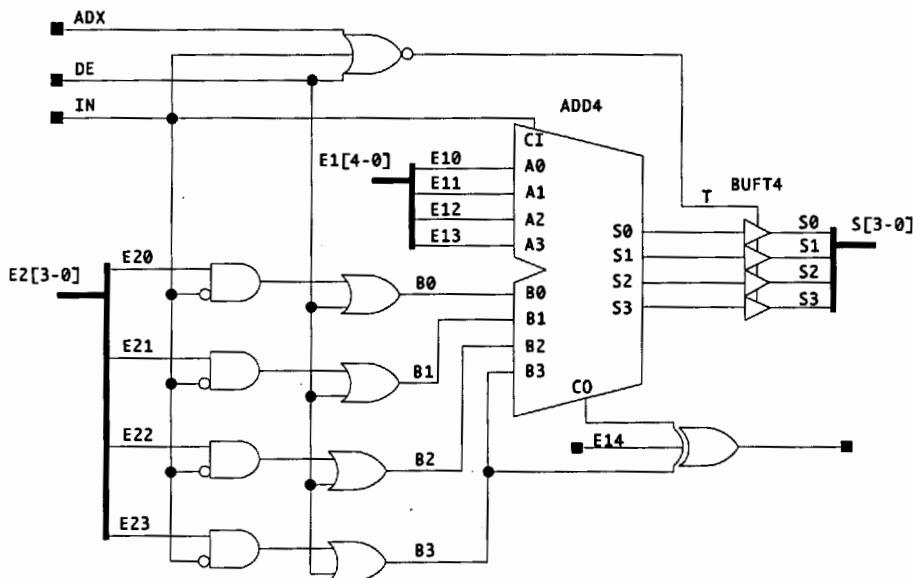


Figure 7-13 Exponent Adder



### 7.3 OTHER FLOATING-POINT OPERATIONS

Next, we consider the design of an adder for floating-point numbers. Two floating-point numbers will be added to form a floating-point sum:

$$(F_1 \times 2^{E_1}) + (F_2 \times 2^{E_2}) = F \times 2^E$$

Again, we will assume that the numbers to be added are properly normalized and that the answer should be put in normalized form. In order to add two fractions, the associated exponents must be equal. Thus, if the exponents  $E_1$  and  $E_2$  are different, we must unnormalize one of the fractions and adjust the exponent accordingly. To illustrate the process, we add

$$F_1 \times 2^{E_1} = 0.111 \times 2^5 \text{ and } F_2 \times 2^{E_2} = 0.101 \times 2^3$$

Since  $E_2 \neq E_1$ , we unnormalize  $F_2$  by shifting right two times and adding 2 to the exponent:

$$0.101 \times 2^3 = 0.0101 \times 2^4 = 0.00101 \times 2^5$$

Note that shifting right one place is equivalent to dividing by 2, so each time we shift we must add 1 to the exponent to compensate. When the exponents are equal, we add the fractions:

$$(0.111 \times 2^5) + (0.00101 \times 2^5) = 01.00001 \times 2^5$$

This addition caused an overflow into the sign bit position, so we shift right and add 1 to the exponent to correct the fraction overflow. The final result is

$$F \times 2^E = 0.100001 \times 2^6$$

When one of the fractions is negative, the result of adding fractions may be unnormalized, as illustrated in the following example:

$$\begin{aligned}
 & (1.100 \times 2^{-2}) + (0.100 \times 2^{-1}) \\
 &= (1.110 \times 2^{-1}) + (0.100 \times 2^{-1}) \text{ (after shifting } F_1\text{)} \\
 &= 0.010 \times 2^{-1} \quad \text{(result of adding fractions is unnormalized)} \\
 &= 0.100 \times 2^{-2} \quad \text{(normalized by shifting left and subtracting 1 from exponent)}
 \end{aligned}$$

In summary, the steps required to carry out floating-point addition are as follows:

1. If the exponents are not equal, shift the fraction with the smallest exponent right and add 1 to its exponent; repeat until the exponents are equal.
2. Add the fractions.
3. (a) If fraction overflow occurs, shift right and add 1 to the exponent to correct the overflow.  
 (b) If the fraction is unnormalized, shift left and subtract 1 from the exponent until the fraction is normalized.  
 (c) If the fraction is 0, set the exponent to the appropriate value.
4. Check for exponent overflow.

Step 4 is necessary, since step 3a or 3b may produce an exponent overflow. If  $E_1 \gg E_2$  and  $F_2$  is positive,  $F_2$  will become all 0s as we right-shift  $F_2$  to equalize the exponents. In this case, the result is  $F = F_1$  and  $E = E_1$ , so it is a waste of time to do the shifting. If  $E_1 \gg E_2$  and  $F_2$  is negative,  $F_2$  will become all 1s (instead of all 0s) as we right-shift  $F_2$  to equalize the exponents. When we add the fractions, we will get the wrong answer. To avoid this problem, we can skip the shifting when  $E_1 \gg E_2$  and set  $F = F_1$  and  $E = E_1$ . Similarly, if  $E_2 \gg E_1$ , we can skip the shifting and set  $F = F_2$  and  $E = E_2$ . For the 4-bit fractions in our example, if  $|E_1 - E_2| > 3$ , we can skip the shifting.

Floating-point subtraction is the same as floating-point addition, except in step 2 we must subtract the fractions instead of adding them.

The quotient of two floating-point numbers is

$$(F_1 \times 2^{E_1}) \div (F_2 \times 2^{E_2}) = (F_1/F_2) \times 2^{(E_1-E_2)} = F \times 2^E$$

Thus, the basic rule for floating-point division is divide the fractions and subtract the exponents. In addition to considering the same special cases as for multiplication, we must test for divide by 0 before dividing. If  $F_1$  and  $F_2$  are normalized, then the largest positive quotient ( $F$ ) will be

$$0.1111\ldots / 0.1000\ldots = 01.111\ldots$$

which is less than  $10_2$ , so the fraction overflow is easily corrected. For example,

$$(0.110101 \times 2^2) \div (0.101 \times 2^{-3}) = 01.010 \times 2^5 = 0.101 \times 2^6$$

Alternatively, if  $F_1 \geq F_2$ , we can shift  $F_1$  right before dividing and avoid fraction overflow in the first place.

The main thrust of this chapter was the design of a floating-point multiplier. In the process of designing the multiplier we used the following steps:

1. Develop an algorithm for floating-point multiplication, taking all of the special cases into account.
2. Draw a block diagram of the system and define the necessary control signals.
3. Construct an SM chart (or state graph) for the control state machine using a separate linked state machine for controlling the fraction multiplier.
4. Write VHDL code to implement the algorithm using control signals, with one process for each state machine and one process for updating the registers on the rising edge of the clock.
5. Test the VHDL code to verify that the high-level design of the multiplier is correct.
6. Draw a top-level schematic for the multiplier and complete the logic design for each module.
7. Enter the schematic diagrams into the computer.
8. Simulate and debug the design using the same test examples as in step 5.
9. Use the CAD software to implement the multiplier using an FPGA.

In the next chapter, we show how CAD tools can be used to synthesize the floating-point multiplier directly from the VHDL code.

## Problems

- 7.1** Add the following floating-point numbers (show each step). Assume that each fraction is 5 bits (including sign) and each exponent is 5 bits (including sign) with negative numbers in 2's complement.

$$\begin{array}{ll} F1 = 0.1011 & E1 = 11111 \\ F2 = 1.0100 & E2 = 11101 \end{array}$$

### 7.2

- (a) Draw a block diagram for a floating-point subtracter. Assume that the inputs to the subtracter are properly normalized, and the answer should be properly normalized. The fractions are 8 bits including sign, and the exponents are 5 bits including sign. Negative numbers are represented in 2's complement.

- (b) Draw an SM chart for the control network for the floating-point subtracter. Define the control signals used, and give an equation for each control signal used as an input to the control network.

- (c) Write the VHDL description of the floating-point subtracter.

- 7.3** This problem concerns the design of a digital system that converts an 8-bit signed integer (negative numbers are represented in 2's complement) to a floating-point number. Use a floating-point format similar to the one used in class, except the fraction should be 8 bits and the exponent 4 bits. The fraction should be properly normalized.

- (a) Draw a block diagram of the system and develop an algorithm for doing the conversion. Assume that the integer is already loaded into an 8-bit register, and when the conversion is complete the fraction should be in the same register. Illustrate your algorithm by converting -27 to floating point.
- (b) Draw a state diagram for the controller. Assume that the start signal is present for only one clock time. (Two states are sufficient.)
- (c) Write a VHDL description of the system.

**7.4** This problem concerns the design of a divider for floating point numbers:

$$(F_1 \times 2^{E_1}) / (F_2 \times 2^{E_2}) = F \times 2^E$$

Assume that  $F_1$  and  $F_2$  are properly normalized fractions (or 0), with negative fractions expressed in 2's complement. The exponents are integers with negative numbers expressed in 2's complement. The result should be properly normalized if it is not zero.

- (a) Draw a flowchart for the floating-point divider. Assume that a divider is available that will divide two binary fractions to give a fraction as a result. Do not show the individual steps in the division of the fractions on your flowchart, just say "divide." The divider requires that  $|F_2| > |F_1|$  before division is carried out.

- (b) Illustrate your procedure by computing

$$0.111 \times 2^3 / 1.011 \times 2^{-2}$$

When you divide  $F_1$  by  $F_2$ , you don't need to show the individual steps, just the result of the division.

- (c) Write a VHDL description for the system.

### 7.5

- (a) State the steps necessary to carry out floating-point subtraction, including special cases. Assume that the numbers are initially in normalized form, and the final result should be in normalized form.
- (b) Subtract the following (fractions are in 2's complement):

$$(1.0111 \times 2^{-3}) - (1.0101 \times 2^{-5})$$

- (c) Write a VHDL description of the system.

**7.6** Two floating-point numbers are added to form a floating-point sum:

$$(F_1 \times 2^{E_1}) + (F_2 \times 2^{E_2}) = F \times 2^E$$

Assume that  $F_1$  and  $F_2$  are normalized, and the result should be normalized.

- (a) List the steps required to carry out floating-point addition, including all special cases.
- (b) Illustrate these steps for  $F_1 = 1.0101$ ,  $E_1 = 1001$ ,  $F_2 = 0.1010$ ,  $E_2 = 1000$ . Note that the fractions are 5 bits, including sign, and the exponents are 4 bits, including sign.
- (c) Write a VHDL description of the system.

**7.7** A floating-point number system uses a 4-bit fraction and a 4-bit exponent with negative numbers expressed in 2's complement. Design an *efficient* system that will multiply the number by  $-4$  (minus four). Take all special cases into account, and give a properly normalized result. Assume that the initial fraction is properly normalized or zero. *Note:* This system multiplies *only* by  $-4$ .

- (a) Give examples of the normal and special cases that can occur (for multiplication by  $-4$ ).
- (b) Draw a block diagram of the system.
- (c) Draw an SM chart for the control unit. Define all signals used.

**7.8** Two exponents are added: 11011 ( $-5$ ) and 00111 ( $7$ ). What is the result, and is there an overflow? Explain your answer.

**7.9** Redesign the floating-point multiplier in Figure 7-2 using a common 5-bit full adder connected to a bus instead of two separate adders for the exponents and fractions.

- (a) Redraw the block diagram and be sure to include the connections to the bus and include all control signals.
- (b) Draw a new SM chart for the new control.
- (c) Write the VHDL description for the multiplier or specify what changes need to be made to an existing description.

**7.10** This problem concerns the design of a network to find the square of a floating-point number,  $F \times 2^E$ .  $F$  is a normalized 5-bit fraction, and  $E$  is a 5-bit integer; negative numbers are represented in 2's complement. The result should be properly normalized. Take advantage of the fact that  $(-F)^2 = F^2$ .

- (a) Draw a block diagram of the network. (Use only one adder and one completer.)
- (b) State your procedure, taking all special cases into account. Illustrate your procedure for

$$F = 1.0110, \quad E = 00100$$

- (c) Draw an SM chart for the main controller. You may assume that multiplication is carried out using a separate control network, which outputs  $M_{done} = 1$  when multiplication is complete.
- (d) Write a VHDL description of the system.

# CHAPTER 8

## ADDITIONAL TOPICS IN VHDL

Up to this point, we have described the basic features of VHDL and how it can be used in the digital system design process. In this chapter, we describe additional features of VHDL that illustrate its power and flexibility. Then we will show how VHDL code can be used as an input to synthesis tools that automatically design digital logic for implementation with PGAs, CPLDs, or ASICs (application-specific integrated circuits).

### 8.1 ATTRIBUTES

An important feature of the VHDL language is attributes. Table 8-1 gives some examples of attributes that can be associated with signals. In this table,  $S$  represents a signal name, and  $S$  is separated from an attribute name by a tick mark (single quote). In VHDL, an event on a signal means a change in the signal. Thus,  $S'EVENT$  (read as “ $S$  tick EVENT”) returns a value of TRUE if a change in  $S$  has just occurred. If  $S$  changes at time  $T$ , then  $S'EVENT$  is true at time  $T$  but false at time  $T + \Delta$ . A transaction occurs on a signal every time it is evaluated, regardless of whether the signal changes or not. Consider the concurrent VHDL statement  $A <= B$  and  $C$ . If  $B = 0$ , then a transaction occurs on  $A$  every time  $C$  changes, since  $A$  is recomputed every time  $C$  changes. If  $B = 1$ , then an event and a transaction occur on  $A$  every time  $C$  changes.  $S'ACTIVE$  returns true if  $S$  has just been re-evaluated, even if  $S$  does not change.

**Table 8-1 Signal Attributes That Return a Value**

Attribute	Returns
$S'EVENT$	True if an event occurred during the current delta, else false
$S'ACTIVE$	True if a transaction occurred during the current delta, else false
$S'LAST_EVENT$	Time elapsed since the previous event on $S$
$S'LAST_VALUE$	Value of $S$ before the previous event on $S$
$S'LAST_ACTIVE$	Time elapsed since previous transaction on $S$

Table 8-2 gives signal attributes that create a signal. The brackets around (time) indicate that (time) is optional. If (time) is omitted, then one delta is used. The attribute  $S'delayed(time)$  creates a signal identical to  $S$ , except it is shifted by the amount of time specified. The example in Figure 8-1 illustrates use of the attributes listed in Table 8-2.

The signal *C\_delayed5* is the same as *C* shifted right by 5 ns. The signal *A\_trans* toggles every time *B* or *C* changes, since *A* has a transaction whenever *B* or *C* changes. The initial computation of *A <= B* and *C* produces a transaction on *A* at time =  $\Delta$ , so *A\_trans* changes to '1' at that time. The signal *A'stable(time)* is true if *A* has not changed during the preceding interval of length (time). Thus, *A\_stable5* is false for 5 ns after *A* changes, and it is true otherwise. The signal *A'quiet(time)* is true if *A* has had no transactions during the preceding interval of length (time). Thus, *A\_quiet5* is false for 5 ns after *A* has had a transaction. S'EVENT and not S'STABLE both return true if an event has occurred during the current delta; however, they cannot always be used interchangeably, since the former just returns a value and the latter returns a signal.

**Table 8-2 Signal Attributes That Create a Signal**

Attribute	Creates
S'DELAYED [(time)]*	Signal same as <i>S</i> delayed by specified time
S'STABLE [(time)]*	Boolean signal that is true if <i>S</i> had no events for the specified time
S'QUIET [(time)]*	Boolean signal that is true if <i>S</i> had no transactions for the specified time
S'TRANSACTION	Signal of type BIT that changes for every transaction on <i>S</i>

\* Delta is used if no time is specified.

**Figure 8-1 Examples of Signal Attributes**

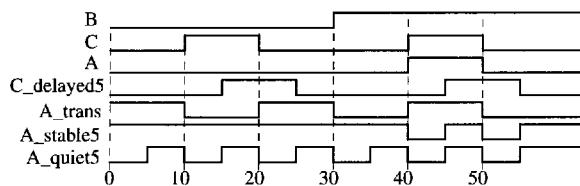
```

entity attr_ex is
  port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
  signal A, C_delayed5, A_trans : bit;
  signal A_stable5, A_quiet5 : boolean;
begin
  A <= B and C;
  C_delayed5 <= C'delayed(5 ns);
  A_trans <= A'transaction;
  A_stable5 <= A'stable(5 ns);
  A_quiet5 <= A'quiet(5 ns);
end test;

```

(a) VHDL code for attribute test



(b) Waveforms for attribute test

Table 8-3 gives array attributes. In this table, *A* can either be an array name or an array type. In the examples, *ROM1* is a two-dimensional array for which the first index range is 0 **to** 15, and the second index range is 7 **downto** 0. *ROM1'LEFT(2)* is 7, since the left bound of the second index range is 7. Although *ROM1* is declared as a signal, the array attributes also work with array constants and array variables. In the examples, the results are the same if *ROM1* is replaced with its type, *ROM*. For a vector (a one-dimensional array), *N* is 1 and can be omitted. If *A* is a bit\_vector dimensioned 2 **to** 9, then *A'LEFT* is 2 and *A'LENGTH* is 8.

**Table 8-3 Array Attributes**

```
Type ROM is array (0 to 15, 7 downto 0) of bit;
Signal ROM1 : ROM;
```

Attribute	Returns	Examples
<i>A'LEFT(N)</i>	left bound of Nth index range	<i>ROM1'LEFT(1) = 0</i> <i>ROM1'LEFT(2) = 7</i>
<i>A'RIGHT(N)</i>	right bound of Nth index range	<i>ROM1'RIGHT(1) = 15</i> <i>ROM1'RIGHT(2) = 0</i>
<i>A'HIGH(N)</i>	largest bound of Nth index range	<i>ROM1'HIGH(1) = 15</i> <i>ROM1'HIGH(2) = 7</i>
<i>A'LOW(N)</i>	smallest bound of Nth index range	<i>ROM1'LOW(1) = 0</i> <i>ROM1'LOW(2) = 0</i>
<i>A'RANGE(N)</i>	Nth index range reversed	<i>ROM1'RANGE(1) = 0 to 15</i> <i>ROM1'RANGE(2) = 7 downto 0</i>
<i>A'REVERSE_RANGE(N)</i>	Nth index range	<i>ROM1'REVERSE_RANGE(1) = 15 downto 0</i> <i>ROM1'REVERSE_RANGE(2) = 0 to 7</i>
<i>A'LENGTH(N)</i>	size of Nth index range	<i>ROM1'LENGTH(1) = 16</i> <i>ROM1'LENGTH(2) = 8</i>

Attributes are often used together with assert statements (see p. 116) for error checking. The assert statement checks to see if a certain condition is true and, if not, causes an error message to be displayed. As an example of using the assert statement together with an attribute, consider the following process, which checks to see if the setup and hold times are satisfied for a D flip-flop:

```
check: process
begin
  wait until rising_edge(Clk);
  assert (D'stable(setup_time))
    report ("Setup time violation")
    severity error;
  wait for hold_time;
  assert (D'stable(hold_time))
    report ("Hold time violation")
    severity error;
end process check;
```

In the *check* process, after the active edge of the clock occurs, the *D* input is checked to see if has been stable for the specified *setup\_time*. If not, a setup-time violation is reported as an error. Then, after waiting for the hold time, *D* is checked to see if it has been stable during the hold-time period. If not, a hold-time violation is reported as an error.

The procedure *Addvec* given in Figure 2-23 requires that the two bit-vectors to be added both be dimensioned *N – 1 downto 0* and that *N* be included in the procedure call. By using attributes, we can write a similar procedure that places no restrictions on the range of the vectors other than the lengths must be the same. When procedure *Addvec2* (Figure 8-2) is executed, it creates a temporary variable, *C*, for the internal carry and initializes it to the input carry, *Cin*. Then it creates aliases *n1*, *n2*, and *S*, which have the same length as *Add1*, *Add2*, and *Sum*, respectively. These aliases are dimensioned from their length minus 1 *downto 0*. Even though the ranges of *Add1*, *Add2*, and *Sum* might be **downto** or **to** and might not include 0, the ranges for the aliases are defined in a uniform manner to facilitate further computation. If the input vectors and *Sum* are not the same length, an error message is reported. The sum and carry are computed bit by bit in a loop, as in Figure 2-23. Since this loop must start with *i = 0*, the range of *i* is the reverse of the range for *S*. Finally, the carry output, *Cout*, is set equal to the corresponding temporary variable, *C*.

**Figure 8-2 Procedure for Adding Bit-Vectors**

```
-- This procedure adds two bit_vectors and a carry and returns a sum
-- and a carry. Both bit_vectors should be of the same length.

procedure Addvec2
  (Add1,Add2: in bit_vector;
  Cin: in bit;
  signal Sum: out bit_vector;
  signal Cout: out bit) is

  variable C: bit := Cin;
  alias n1 : bit_vector(Add1'length-1 downto 0) is Add1;
  alias n2 : bit_vector(Add2'length-1 downto 0) is Add2;
  alias S : bit_vector(Sum'length-1 downto 0) is Sum;

begin
  assert ((n1'length = n2'length) and (n1'length = S'length))
    report "Vector lengths must be equal!"
    severity error;

  for i in s'reverse_range loop
    S(i) <= n1(i) xor n2(i) xor C;
    C := (n1(i) and n2(i)) or (n1(i) and C) or (n2(i) and C);
  end loop;
  Cout <= C;
end Addvec2;
```

## 8.2 TRANSPORT AND INERTIAL DELAYS

VHDL provides for two types of delays—transport delays and inertial delays. The transport delay, which is intended to model the delay introduced by wiring, simply delays an input signal by the specified delay time. The inertial delay, which is the default delay type for VHDL, is intended to model gates and other devices that do not propagate short pulses from the input to the output. If a gate has an ideal inertial delay  $T$ , the input signal is delayed by time  $T$ , but any pulse with a width less than  $T$  is rejected. For example, if a gate has an inertial delay of 10 ns, a pulse of width 10 ns would pass through, but a pulse of width 9.999 ns would be rejected. Real devices do not behave in this way. In general, the maximum width of a pulse rejected by a device will be less than the delay time. VHDL can model such devices by adding a reject clause to the assignment statement. A statement of the form

```
signal_name <= reject pulse-width expression after delay-time
```

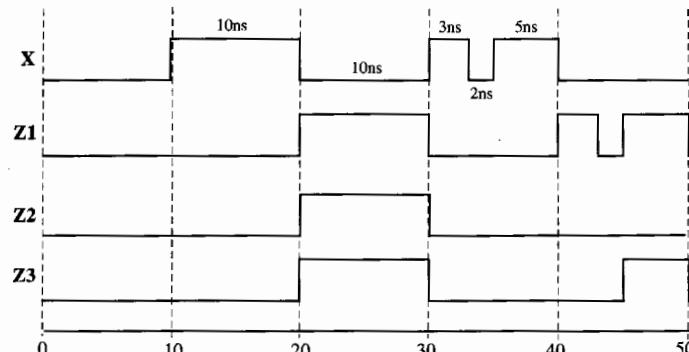
evaluates the expression, rejects any pulses whose width is less than pulse width, and then sets the signal equal to the result after a delay of delay-time. In statements of this type, the rejection pulse width must be less than the delay time.

Figure 8-3 illustrates the difference between transport and inertial delays. Consider the following VHDL statements:

```
Z1 <= transport X after 10 ns;      -- transport delay
Z2 <= X after 10 ns;            -- inertial delay
Z3 <= reject 4 ns X after 10 ns; -- delay with specified
                                  -- rejection pulse width
```

$Z_1$  is the same as  $X$ , except that it is shifted 10 ns in time.  $Z_2$  is similar to  $Z_1$ , except the pulses in  $X$  shorter than 10 ns are filtered out and do not appear in  $Z_2$ .  $Z_3$  is the same as  $Z_2$ , except that only the pulses of width less than 4 ns have been rejected. In general, using **reject** is equivalent to using a combination of an inertial delay and a transport delay. The statement for  $Z_3$  given here could be replaced with the concurrent statements

**Figure 8-3 Transport and Inertial Delays**



```
Zm <= X after 4 ns; -- inertial delay rejects short pulses
Z3 <= transport Zm after 6 ns; -- total delay is 10 ns
```

Suppose that the following sequential statements are executed at time  $T$ :

```
A <= transport B after 1 ns;
A <= transport C after 2 ns;
```

The result is that  $A$  is scheduled to receive the value of  $B$  at time  $T + 1$  ns, and  $A$  is also scheduled to receive the value of  $C$  at time  $T + 2$  ns.

Now suppose that the following sequential statements are executed at time  $T$ :

```
A <= B after 1 ns; -- inertial delay implied
A <= C after 2 ns; -- inertial delay implied
```

First,  $A$  is scheduled to receive the value of  $B$  at time  $T + 1$  ns. Then,  $A$  is scheduled to receive the value of  $C$  at  $T + 2$  ns. Since this implies an inertial delay of 2 ns, the first scheduled change is preempted. That is, the change to  $B$  is removed from the queue, and only the change to  $C$  occurs.

If a signal is scheduled to change at a given time and then a second change is scheduled to occur at an earlier time, the first change is deleted from the queue. For example, suppose that the following sequential statements are executed at time  $T$ :

```
A <= transport B after 2 ns;
A <= transport C after 1 ns;
```

First  $A$  is scheduled to change to  $B$  at time  $T + 2$  ns. Then  $A$  is scheduled to change to  $C$  at time  $T + 1$  ns, and the previous change to  $B$  is removed from the queue.

### 8.3 OPERATOR OVERLOADING

The VHDL arithmetic operators,  $+$  and  $-$ , are defined to operate on integers, but not on bit-vectors. In previous examples, we called a function or a procedure when we needed to add two bit-vectors. By using operator overloading, we can extend the definition of " $+$ " so that using the " $+$ " operator will implicitly call an appropriate addition function, which eliminates the need for an explicit function or procedure call. When the compiler encounters a function declaration in which the function name is an operator enclosed in double quotes, the compiler treats this function as an operator overloading function. The package shown in Figure 8-4 defines two " $+$ " functions. The first one adds two bit-vectors and returns a bit-vector. This function uses aliases as in Figure 8-2 so that it is independent of the ranges of the bit-vectors, but it assumes that the lengths of the vectors are the same. It uses a for loop to do the bit-by-bit addition. The second function in the package adds an integer to a bit-vector and returns a bit-vector.

When a " $+$ " operator is encountered, the compiler automatically checks the types of the operands and calls the appropriate functions. Consider the statement

```
A <= B + C + 3;
```

and assume that the *bit\_overload* package of Figure 8-4 is being used. If *A*, *B*, and *C* are of type integer, integer arithmetic is used. If *A*, *B*, and *C* are of type bit\_vector, the first function in the package is called to add *B* and *C*, then the second function is called to add 3 to the sum of *B* and *C*. The statement

```
A <= 3 + B + C
```

would cause a compile-time error, since we have not defined an overloading function for "+" when the first operand is an integer and the second operand is a bit-vector.

**Figure 8-4 VHDL Package with Overloaded Operators for Bit-Vectors**

```
-- This package provides two overloaded functions for the plus operator
package bit_overload is
    function "+" (Add1, Add2: bit_vector) return bit_vector;
    function "+" (Add1: bit_vector; Add2: integer) return bit_vector;
end bit_overload;

library BITLIB;
use BITLIB.bit_pack.all;

package body bit_overload is

    -- This function returns a bit_vector sum of two bit_vector operands
    -- The add is performed bit by bit with an internal carry
    function "+" (Add1, Add2: bit_vector) return bit_vector is
        variable sum: bit_vector(Add1'length-1 downto 0);
        variable c: bit := '0';                                -- no carry in
        alias n1: bit_vector(Add1'length-1 downto 0) is Add1;
        alias n2: bit_vector(Add2'length-1 downto 0) is Add2;
    begin
        for i in sum'reverse_range loop
            sum(i) := n1(i) xor n2(i) xor c;
            c := (n1(i) and n2(i)) or (n1(i) and c) or (n2(i) and c);
        end loop;
        return (sum);
    end "+";

    -- This function returns a bit_vector sum of a bit_vector and an integer
    -- using the previous function after the integer is converted.
    function "+" (Add1: bit_vector; Add2: integer) return bit_vector is
    begin
        return (Add1 + int2vec(Add2, Add1'length));
    end "+";
end bit_overload;
```

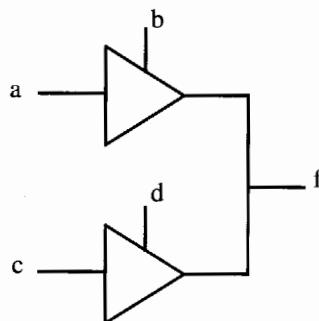
Overloading can also be applied to procedures and functions. Several procedures can have the same name, and the type of the actual parameters in the procedure call determines which version of the procedure is called.

## 8.4 MULTIVALUED LOGIC AND SIGNAL RESOLUTION

In previous chapters, we have used two-valued bit logic in our VHDL code. In order to represent tristate buffers and busses, we need to introduce a third value, 'Z', which represents the high-impedance state. We also introduce a fourth value, 'X', which represents an unknown state. This unknown state may occur if the initial value of a signal is unknown, or if a signal is simultaneously driven to two conflicting values, such as '0' and '1'. If the input to a gate is 'Z', the gate output may assume an unknown value, 'X'.

Figure 8-5 shows two tristate buffers with their outputs tied together, and Figure 8-6 shows the VHDL representation. All the signals in this example are of type X01Z and can assume the four values: 'X', '0', '1', and 'Z'. The tristate buffers have an active-high output enable, so that when  $b = 1$  and  $d = 0$ ,  $f = a$ ; when  $b = 0$  and  $d = 1$ ,  $f = c$ ; and when  $b = d = 0$ , the  $f$  output assumes the high-Z state. If  $b = d = 1$ , an output conflict can occur. Two VHDL architectures are shown. The first one uses two concurrent statements, and the second one uses two processes. In either case,  $f$  is driven from two different sources, and VHDL uses a *resolution function* to determine the actual output. For example, if  $a = c = d = 1$  and  $b = 0$ ,  $f$  is driven to 'Z' by one concurrent statement or process, and  $f$  is driven to '1' by the other concurrent statement or process. The resolution function is automatically called to determine that the proper value of  $f$  is '1'. The resolution function will supply a value of 'X' (unknown) if  $f$  is driven to both '0' and '1' at the same time.

**Figure 8-5 Tristate Buffers with Active-High Output Enable**



**Figure 8-6 VHDL Code for Figure 8-5**

```

use WORK.fourpack.all;
entity t_buff_exmpl is
    port (a,b,c,d : in X01Z;          -- signals are four-valued
          f: out X01Z);
end t_buff_exmpl;

architecture t_buff_conc of t_buff_exmpl is
begin
    f <= a when b = '1' else 'Z';
    f <= c when d = '1' else 'Z';
end t_buff_conc;

architecture t_buff_bhv of t_buff_exmpl is
begin
    buff1: process (a,b)
    begin
        if (b='1') then
            f<=a;
        else
            f<='Z';  --"drive" the output high Z when not enabled
        end if;
    end process buff1;

    buff2: process (c,d)
    begin
        if (d='1') then
            f<=c;
        else
            f<='Z';  --"drive" the output high Z when not enabled
        end if;
    end process buff2;
end t_buff_bhv;

```

VHDL signals may either be resolved or unresolved. Resolved signals have an associated resolution function, and unresolved signals do not. We have previously used signals of type bit, which are unresolved. If we drive a bit signal *B* to two different values in two concurrent statements (or in two processes), the compiler will flag an error, since there is no way to determine the proper value of *B*.

Consider the following three concurrent statements, where *R* is a resolved signal of type X01Z:

```

R <= transport '0' after 2 ns, 'Z' after 6 ns;
R <= transport '1' after 4 ns;
R <= transport '1' after 8 ns, '0' after 10 ns;

```

Assuming that *R* is initialized to 'Z', three drivers would be created for *R*, as shown in Figure 8-7. Each time one of the unresolved signals *s*(0), *s*(1), or *s*(2) changes, the resolution function is automatically called to determine the value of the resolved signal, *R*.

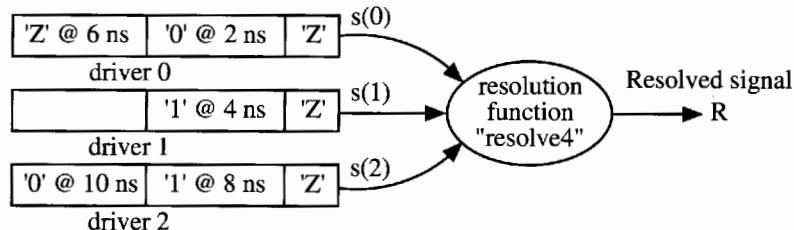
**Figure 8-7 Resolution of Signal Drivers**

Figure 8-8 shows how the resolution function for X01Z logic is defined in a package called *fourpack*. First, an unresolved logic type *u\_X01Z* is defined, along with the corresponding unconstrained array type, *u\_X01Z\_vector*. Then a resolution function, named *resolve4*, is declared. Resolved X01Z logic is defined as a subtype of *u\_X01Z*. The subtype declaration contains the function name *resolve4*. This implies that whenever a signal of type X01Z is computed, function *resolve4* is called to compute the correct value.

The resolution function, which is based on the operation of a tristate bus, is specified by the following table:

	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'X'	'X'
'0'	'X'	'0'	'X'	'0'
'1'	'X'	'X'	'1'	'1'
'Z'	'X'	'0'	'1'	'Z'

This table gives the resolved value of a signal for each pair of input values: Z resolved with any value returns that value, X resolved with any value returns X, and 0 resolved with 1 returns X. The function *resolve4* has an argument, *s*, which represents a vector of one or more signal values to be resolved. If the vector is of length 1, then the first (and only) element of the vector is returned. Otherwise, the return value (the resolved signal) is computed iteratively by starting with *result* = 'Z' and recomputing *result* by a table lookup using each element of the *s* vector in turn. In the example of Figure 8-7, the *s* vector has three elements, and *resolve4* would be called at 0, 2, 4, 6, 8, and 10 ns to compute *R*. The following table shows the result:

Time	s(0)	s(1)	s(2)	R
0	'Z'	'Z'	'Z'	'Z'
2	'0'	'Z'	'Z'	'0'
4	'0'	'1'	'Z'	'X'
6	'Z'	'1'	'Z'	'1'
8	'Z'	'1'	'1'	'1'
10	'Z'	'1'	'0'	'X'

**Figure 8-8 Resolution Function for X01Z Logic**

```

package fourpack is
    type u_x01z is ('X','0','1','Z');           -- u_x01z is unresolved
    type u_x01z_vector is array (natural range <>) of u_x01z;
    function resolve4 (s:u_x01z_vector) return u_x01z;
    subtype x01z is resolve4 u_x01z;
    -- x01z is a resolved subtype which uses the resolution function resolve4
    type x01z_vector is array (natural range <>) of x01z;
end fourpack;

package body fourpack is
    type x01z_table is array (u_x01z,u_x01z) of u_x01z;
    constant resolution_table : x01z_table := (
        ('X','X','X','X'),
        ('X','0','X','0'),
        ('X','X','1','1'),
        ('X','0','1','Z'));
    function resolve4 (s:u_x01z_vector) return u_x01z is
        variable result : u_x01z := 'Z';
begin
    if (s'length = 1) then
        return s(s'low);
    else
        for i in s'range loop
            result := resolution_table(result,s(i));
        end loop;
    end if;
    return result;
end resolve4;
end fourpack;

```

In order to write VHDL code using X01Z logic, we need to define the required operations for this type of logic. For example, AND and OR may be defined using the following tables:

AND	'X'	'0'	'1'	'Z'
'X'	'X'	'0'	'X'	'X'
'0'	'0'	'0'	'0'	'0'
'1'	'X'	'0'	'1'	'X'
'Z'	'X'	'0'	'X'	'X'

OR	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'1'	'X'
'0'	'X'	'0'	'1'	'X'
'1'	'1'	'1'	'1'	'1'
'Z'	'X'	'X'	'1'	'X'

The first table corresponds to the way an AND gate with 4-valued inputs would work. If one of the AND gate inputs is 0, the output is always 0. If both inputs are 1, the output is 1. In all other cases, the output is unknown (X), since a high-Z gate input may act like either a 0 or 1. For an OR gate, if one of the inputs is 1, the output is always 1. If both inputs are 0, the output is 0. In all other cases, the output is X. AND and OR functions based on these tables can be included in the package *fourpack* to overload the AND and OR operators.

## 8.5 IEEE-1164 STANDARD LOGIC

The IEEE-1164 Standard specifies a 9-valued logic system for use with VHDL. The nine logic values defined in this standard are

'U'	Uninitialized
'X'	Forcing unknown
'0'	Forcing 0
'1'	Forcing 1
'Z'	High impedance
'W'	Weak unknown
'L'	Weak 0
'H'	Weak 1
'-'	Don't care

The unknown, 0, and 1 values come in two strengths—forcing and weak. If a forcing signal and a weak signal are tied together, the forcing signal dominates. For example if '0' and 'H' are tied together, the result is '0'. The output of a pull-up resistor could be represented by a value of 'H'. The nine-valued logic is useful in modeling the internal operation of certain types of ICs. In this text, we will normally use only a subset of the IEEE values—'X', '0', '1', and 'Z'.

The IEEE-1164 standard defines the AND, OR, NOT, XOR, and other functions for 9-valued logic. It also specifies a number of subtypes of the 9-valued logic, such as the X01Z subtype, which we have already been using. Table 8-4 shows the resolution function table for the IEEE 9-valued logic. The row index values have been listed as comments to the right of the table. The resolution function table for X01Z logic is a subset of this table, as indicated by the black rectangle.

**Table 8-4 Resolution Function Table for IEEE 9-valued Logic**

```
CONSTANT resolution_table : stdlogic_table := (
-- -----
-- | U X 0 1 Z W L H - | | |
-- -----
( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
);

```

The table for the standard logic **and** operation is shown in Table 8-5. The **and** functions given in Figure 8-9 use this table. These functions provide for operator overloading. This means that if we write an expression that uses the **and** operator, the compiler will automatically call the appropriate **and** function to evaluate the **and** operation depending on the type of the operands. If **and** is used with bit variables, the ordinary **and** function is used, but if **and** is used with **std\_logic** variables, the **std\_logic and** function is called. Operator overloading also automatically applies the appropriate **and** function to vectors. When **and** is used with bit-vectors, the ordinary bit-by-bit **and** is performed, but when **and** is applied to **std\_logic** vectors, the **std\_logic and** is applied on a bit-by-bit basis. The first **and** function in Figure 8-9 computes the **and** of the left (*l*) and right (*r*) operands by doing a table lookup. The second **and** function works with **std\_logic** vectors. Aliases are used to make sure the index range is the same direction for both operands. If the vectors are not the same length, the **assert** false always causes the message to be displayed. Otherwise, each bit in the result vector is computed by table lookup.

**Table 8-5 And Table for IEEE 9-valued Logic**

```
CONSTANT and_table : stdlogic_table := (
-- -----
-- | U X 0 1 Z W L H - | | |
-- -----
( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ) -- | - |
);

```

Figure 8-9 And Function for std\_logic\_vectors

```

function "and"  ( l : std_ulogic; r : std_ulogic ) return UX01 is
begin
    return (and_table(l, r));
end "and";

function "and"  ( l,r : std_logic_vector ) return std_logic_vector is
    alias lv : std_logic_vector ( 1 to l'LENGTH ) is l;
    alias rv : std_logic_vector ( 1 to r'LENGTH ) is r;
    variable result : std_logic_vector ( 1 to l'LENGTH );
begin
    if ( l'LENGTH /= r'LENGTH ) then
        assert FALSE
        report "arguments of overloaded 'and' operator are not of the same
length"
        severity FAILURE;
    else
        for i in result'RANGE loop
            result(i) := and_table (lv(i), rv(i));
        end loop;
    end if;
    return result;
end "and";

```

The *rising\_edge* function for standard logic is as follows:

```

function rising_edge (signal s : std_ulogic) return BOOLEAN is
begin
    return (s'EVENT and (To_X01(s) = '1') and
            (To_X01(s'LAST_VALUE) = '0'));
end;

```

The *To\_X01* converts *s* to the three values, 'X', '0', and '1'. Only a change from '0' to '1' is recognized as a rising edge.

In Figure 3-11, we used the function *PLAout* to determine the output of a PLA from its input and the PLA table. The function call does not explicitly pass the dimensions of the input and output vectors as parameters. Instead, these dimensions are determined within the function body using signal attributes. The *PLAout* function is given in Figure 8-10. Input'length is the length of the *Input* vector. PLA'length(2) gives the width of the PLA table. The high index of the *Output* vector is then (PLA'length(2) – Input'length – 1). The outer loop (LP1) processes the PLA table row by row. Since the *Input* column range is from left to right, we must make sure that we also step through the PLA table columns from left to right. Since the PLA column range can either be high **downto** low or low **to** high, we define a step that is -1 or 1, respectively. Loop LP2 copies the current row of the PLA table into the variable *PLArow* in such a way that the index range of *PLArow* is always high **downto** 0. *PLAcol* is initially set to the left column index of the PLA table and incremented or decremented depending on the value of step. After exiting LP2, the input part of the *PLA* row is copied to *PLAinp*. Loop LP3 checks the input part of each row to

see if it matches the PLA inputs. If we get an input match, then we do a bit-by-bit **or** between the PLA output vector (*Output*) and the output part of the PLA table.

**Figure 8-10 Function to Determine a PLA Output**

```
-- The following function produces the PLA output according to the PLA
-- specification (PLAmtrx) and the PLA inputs.
-- The steps in this function are as follows:
-- 1) Start at the first row of the PLAmtrx
-- 2) Determine whether the given inputs match the PLA inputs in the
--    current row
-- 3) If the inputs match current PLA row, then OR the current outputs
--    with the PLA outputs
-- 4) Repeat steps (2) and (3) for all rows in the PLA matrix

type PLAmtrx is array (integer range <>, integer range <>) of std_logic;
function PLAout (PLA: PLAmtrx; Input: std_logic_vector)
  return std_logic_vector is
  alias In1: std_logic_vector(Input'length-1 downto 0) is Input;
  variable match: std_logic;
  variable PLAcoll, step: integer;
  variable PLArow: std_logic_vector(PLA'length(2)-1 downto 0);
  variable PLAinp: std_logic_vector(Input'length-1 downto 0);
  variable Output:
    std_logic_vector((PLA'length(2)-Input'length-1) downto 0);
begin
  Output := (others=>'0');           -- Initialize output to all zeros
  if PLA'left(2) > PLA'reight(2) then step := -1; else step := 1;
  end if;
  LP1: for row in PLA'range loop   --Scan each row of PLA
    match := '1';                  -- Assume match for now
    PLAcoll := PLA'left(2);

    LP2: for col in PLArow'range loop -- Copy row of PLA table
      PLArow(col) := PLA(row,PLAcoll);
      PLAcoll := PLAcoll + step;
    end loop LP2;
    PLAinp := PLArow(PLArow'high downto PLArow'high-Input'length+1);

    LP3: for col in In1'range loop -- Scan each input column
      if IN1(col) /= PLAinp(col) and PLAinp(col) /= 'X' then
        match := '0'; exit;          -- mismatched row
      end if;
    end loop LP3;

    if (match = '1') then
      Output := Output or PLArow(PLArow'range);
    end if;
  end loop LP1;
  return Output;
end PLAout;
```

We have placed the PLAmtrx type definition and the *PLAout* function in a multivalued logic library called *MVLLIB*. The VHDL code in Figure 8-11 tests the operation of *PLAout* using the PLA table of Table 3-2. When *PLAout* is called with *ABC* = "110", the computation proceeds as follows:

```

IN1 = "110", Output = "0000", 6 > 0 so step = -1
LP1: row = 0, PLAcol = 6
    LP2: PLArow = "00X1010"; PLAinp = "00X"
    LP3: col = 2, IN1(2) = '1', PLAinp(2) = '0', match = '0'
(LP1) row = 1, match = '1', PLAcol = 6
    LP2: PLArow = "1X01100"; PLAinp = "1X0"
    LP3: col = 2, IN1(2) = '1', PALinp(2) = '1'
        col = 1, IN1(1) = '1', PALinp(1) = 'X'
        col = 0, IN1(0) = '0', PALinp(0) = '0' -- row matches
Output = "0000" or "1100" = "1100"
(LP1) row = 2, etc.

```

**Figure 8-11 Test of PLAout Function**

```

library ieee;
use ieee.std_logic_1164.all;

library MVLLIB;
use MVLLIB.mvl_pack.all;
entity PLATest is
    port(ABC: in std_logic_vector(2 downto 0);
         F: out std_logic_vector(3 downto 0));
end PLATest;

architecture PLA1 of PLATest is
constant PLA3_2: PLAmtrx(0 to 4, 6 downto 0) :=
  ("00X1010", "1X01100", "X1X0101", "X100010", "1X10001");
begin
    F <= PLAout(PLA3_2, ABC);
end PLA1;

```

## 8.6 GENERICS

*Generics* are commonly used to specify parameters for a component in such a way that the parameter values may be specified when the component is instantiated. For example, the rise and fall times for a gate could be specified as generics, and different numeric values for these generics could be assigned for each instance of the gate. The example of Figure 8-12 describes a two-input nand gate whose rise and fall delay times depend on the number of loads on the gate. In the entity declaration, *Trise*, *Tfall*, and *load* are generics that specify the no-load rise time, the no-load fall time, and the number of loads. In the architecture, an internal *nand\_value* is computed whenever *a* or *b* changes. If *nand\_value* has just changed

to a '1', a rising output has occurred, and the gate delay time is computed as

$$Trise + 3 \text{ ns} * load$$

where 3 ns is the added delay for each load. Otherwise, a falling output has just occurred and the gate delay is computed as

$$Tfall + 2 \text{ ns} * load$$

where 2 ns is the added delay for each load.

**Figure 8-12** Rise/Fall Time Modeling Using Generic Statement

```

entity NAND2 is
  generic (Trise, Tfall: time; load: natural);
  port (a,b : in bit;
        c: out bit);
end NAND2;

architecture behavior of NAND2 is
  signal nand_value : bit;
begin
  nand_value <= a nand b;
  c <= nand_value after (Trise + 3 ns * load) when nand_value = '1'
    else nand_value after (Tfall + 2 ns * load);
end behavior;

entity NAND2_test is
  port (in1, in2, in3, in4 : in bit;
        out1, out2 : out bit);
end NAND2_test;

architecture behavior of NAND2_test is
  component NAND2 is
    generic (Trise: time := 3 ns; Tfall: time := 2 ns;
             load: natural := 1);
    port (a,b : in bit;
          c: out bit);
  end component;
begin
  U1: NAND2 generic map (2 ns, 1 ns, 2) port map (in1, in2, out1);
  U2: NAND2 port map (in3, in4, out2);
end behavior;

```

The entity *NAND2\_test* tests the *NAND2* component. The component declaration in the architecture specifies default values for *Trise*, *Tfall*, and *load*. When *U1* is instantiated, the generic map specifies different values for *Trise*, *Tfall*, and *load*. When *U2* is instantiated, no generic map is included, so the default values are used.

## 8.7 GENERATE STATEMENTS

In the example of Figure 2-4 we instantiated four full-adder components and interconnected them to form the 4-bit adder of Figure 2-3. Specifying the port maps for each instance of the full adder would become very tedious if the adder had 8 or more bits. When an iterative array of identical components is required, the **generate** statement provides an easy way of instantiating these components. The example of Figure 8-13 shows how a generate statement can be used to instantiate the 4-bit adder of Figure 2-4. The notation is the same as in the figure, except a 5-bit vector represents the carries, with *Cin* the same as *C*(0) and *Cout* the same as *C*(4). The **for** loop generates four copies of the full adder, each with the appropriate **port map** to specify the interconnections between the adders.

**Figure 8-13 Adder4 Using Generate Statement**

```

entity Adder4 is
    port (A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
          S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;

architecture Structure of Adder4 is
component FullAdder
    port (X, Y, Cin: in bit;                                -- Inputs
          Cout, Sum: out bit);                            -- Outputs
end component;

signal C: bit_vector(4 downto 0);

begin
    C(0) <= Ci;
    -- generate four copies of the FullAdder
    FullAdd4: for i in 0 to 3 generate
    begin
        FAX: FullAdder port map (A(i), B(i), C(i), C(i+1), S(i));
    end generate FullAdd4;
    Co <= C(4);
end Structure;

```

In the preceding example, we used a generate statement of the form

```

generate_label: for identifier in range generate
    [begin]
        concurrent statement(s)
    end generate [generate_label];

```

At compile time, a set of concurrent statement(s) is generated for each value of the identifier in the given range. In Figure 8-13, one concurrent statement—a component instantiation statement—is used. A generate statement itself is defined to be a concurrent statement, so nested generate statements are allowed.

A generate statement with an if clause may be used to conditionally generate a set of concurrent statement(s). This type of generate statement has the form

```
generate_label: if condition generate
  [begin]
    concurrent statement(s)
  end generate [generate_label];
```

In this case, the concurrent statements(s) are generated at compile time only if the condition is true.

## 8.8 SYNTHESIS OF VHDL CODE

A number of CAD tools are now available that take a VHDL description of a digital system and automatically generate a circuit description that implements the digital system. The output from such synthesis tools may be a logic schematic, together with an associated wirelist, which implements the digital system as an interconnection of gates, flip-flops, registers, counters, multiplexers, adders, and other basic logic blocks. Some synthesis tools are capable of implementing the digital system described by the VHDL code using a PGA or CPLD.

Even if VHDL code compiles and simulates correctly, it will not necessarily synthesize correctly. And even if the VHDL code does synthesize correctly, the resulting implementation may be inefficient. In general, synthesis tools will accept only a subset of VHDL as input. Other changes must be made in the VHDL code so the synthesis tool “understands” the intent of the designer. Further changes in the VHDL code may be required in order to produce an efficient implementation.

In VHDL, a signal may represent the output of a flip-flop or register, or it may represent the output of a combinational logic block. The synthesis tool will attempt to determine which is intended from the context. For example, the concurrent statement

```
A <= B and C;
```

implies that *A* should be implemented using combinational logic. On the other hand, if the sequential statements

```
wait for clock'event and clock = '1';
A <= B and C;
```

appear in a process, this implies that *A* represents a register (or flip-flop) that changes state on the rising edge of the clock.

When integer signals are used, specifying the integer range is important. If no range is specified, the VHDL synthesizer may interpret an integer signal to represent a 32-bit register, since the maximum size of a VHDL integer is 32 bits. When the integer range is specified, most synthesizers will implement integer addition and subtraction using binary adders with the appropriate number of bits.

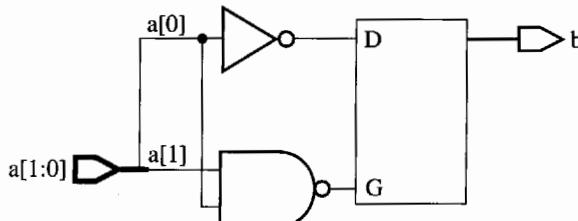
In general, when a VHDL signal is assigned a value, it will hold that value until it is assigned a new value. Because of this property, some VHDL synthesizers will infer a latch when none is intended by the designer. Figure 8-14 shows an example of a **case** statement that creates an unintended latch. Since the value of  $b$  is not specified if  $a$  is not equal to 0, 1, or 2, the synthesizer assumes that the value of  $b$  should be held in a latch if  $a = 3$ . When  $a$  equals 0, 1, or 2,  $b = a'_0$ , so  $D = a'_0$ . When  $a = 3$ , the previous value of  $B$  should be held in the latch, so  $G$  should be 0 when  $a = 3$ . Thus  $G = (a_1 a_0)'$ . The latch can be eliminated by replacing the word **null** in the VHDL code with  $b \leq '0'$ .

Figure 8-14 Example of Unintentional Latch Creation

```
entity latch_example is
  port(a: in integer range 0 to 3;
       b: out bit);
end latch_example;

architecture test1 of latch_example is
begin
  process(a)
  begin
    case a is
      when 0 => b <= '1';
      when 1 => b <= '0';
      when 2 => b <= '1';
      when others => null;
    end case;
  end process;
end test1;
```

(a) VHDL code that infers a latch



(b) Synthesized network

When **if** statements are used, care should be taken to specify a value for each branch. For example, if a designer writes

```
if A = '1' then Nextstate <= 3;
end if;
```

he or she may intend for *Nextstate* to retain its previous value if  $A \neq '1'$ , and the code will simulate correctly. However, the synthesizer might interpret this code to mean if  $A \neq '1'$ ,

then *Nextstate* is unknown ('X'), and the result of the synthesis may be incorrect. For this reason, it is always best to include an **else** clause in every **if** statement. For example,

```
if A = '1' then Nextstate <= 3;  
    else Nextstate <= 2;  
end if;
```

is unambiguous.

Most VHDL synthesizers do a line-by-line translation of the VHDL into gates, registers, multiplexers, and other general components with very little optimization up front. Then the resulting design is optimized and mapped into a specific implementation technology, such as PGAs or CPLDs. Some VHDL synthesizers allow the design to be optimized for speed, for chip area, or for some compromise between maximum speed and minimum area. When optimizing for speed, the number of components may be increased in order to reduce the length of the path that has the maximum propagation delay. For example, converting from a three-level gate network to a two-level network generally increases the number of gates, but it reduces the propagation delay. When optimizing for area, the number of components is generally reduced, which in turn reduces the required chip area. During the initial translation of the VHDL code and during the optimization phase, the synthesis tool will select components from those available in its library. Several different component libraries may be provided to allow implementation with different technologies.

The example of Figure 8-15 shows how the Synopsis Design Compiler implements a **case** statement using a multiplexer and gates. The integers *a* and *b* are each implemented with 2-bit binary numbers. The MUX symbol uses one control input for each data input. The three data inputs to the MUX are 01, 11, and 00. The three corresponding control inputs are formed by encoding the two bits of *a*. Figure 8-15(c) shows the resulting circuit after optimization. Because the MUX inputs are constants, elimination of the MUX and several gates was possible. The final equations are  $b_1 = a'_1 a_0$  and  $b_0 = a'_1 + a_0$ .

The example of Figure 8-16 shows how the Synopsis Design Compiler implements an **if-then-elsif-else** statement using a multiplexer and gates. Since the signals *C*, *D*, *E*, and *Z* are each 3 bits wide, a 3-bit-wide multiplexer is used. This multiplexer has three separate control inputs to select *C*, *D*, or *E*. *C* is selected if *A* = 1; *D* is selected if *A* = 0 and *B* = 0; and *E* is selected if *A* = 0 and *B* = 1.

## Standard VHDL Synthesis Package

Since standard VHDL does not provide for arithmetic operations on bit-vectors, we have used functions and procedures such as *Add4* and *Addvec* to perform addition on bit-vectors. Every VHDL synthesis tool provides its own package of functions for operations commonly used in hardware models. IEEE is developing a standard synthesis package, which includes functions for arithmetic operations on bit\_vectors and std\_logic vectors.

The *numeric\_bit* package defines arithmetic operations on bit\_vectors. The package defines two unconstrained array types to represent unsigned and signed binary numeric values:

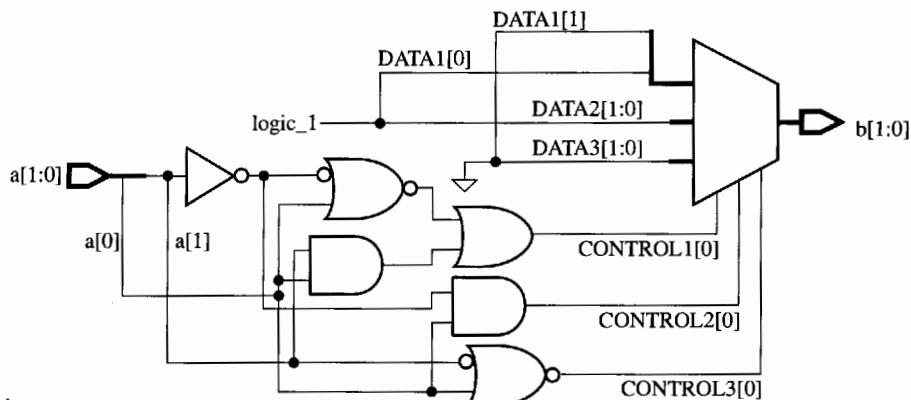
```
type unsigned is array (natural range <>) of bit;
type signed   is array (natural range <>) of bit;
```

Signed numbers are represented in 2's complement format. The package contains overloaded versions of arithmetic, relational, logical, and shifting VHDL operators as well as conversion functions. The *numeric\_std* package defines similar operations on std\_logic vectors. The types unsigned and signed are defined as arrays of std\_logic vectors instead of bit arrays.

**Figure 8-15 Synthesis of a Case Statement**

```
entity case_example is
  port(a: in integer range 0 to 3;
       b: out integer range 0 to 3);
end case_example;
architecture test1 of case_example is
begin
  process(a)
  begin
    case a is
      when 0 => b <= 1;
      when 1 => b <= 3;
      when 2 => b <= 0;
      when 3 => b <= 1;
    end case;
  end process;
end test1;
```

(a) VHDL code for case example



(b) Synthesized case statement before optimization

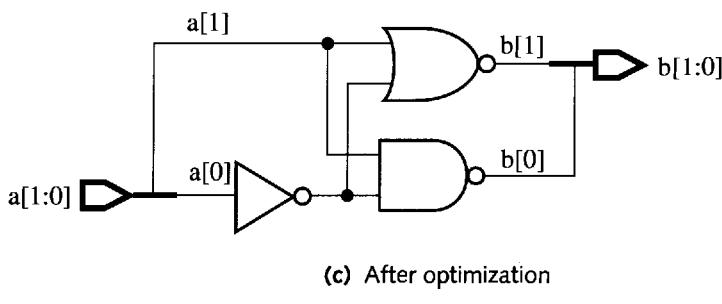


Figure 8-16 Synthesis of an if Statement

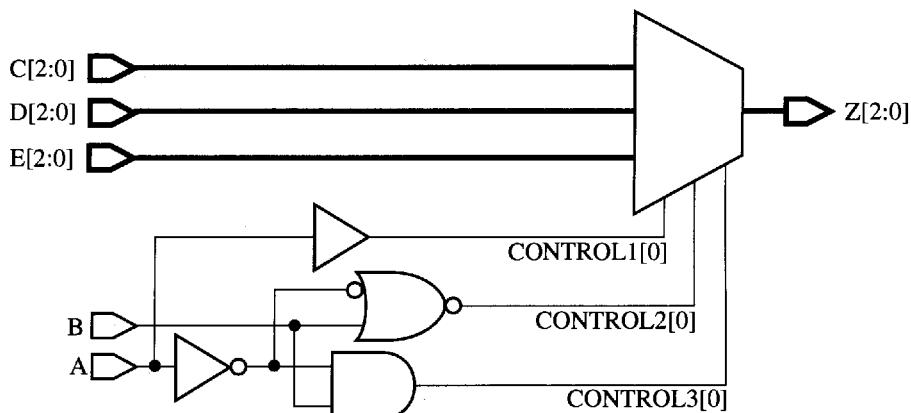
```

entity if_example is
  port(A,B: in bit;
       C,D,E: in bit_vector(2 downto 0);
       Z: out bit_vector(2 downto 0));
end if_example;

architecture test1 of if_example is
begin
  process(A,B)
  begin
    if A = '1' then Z <= C;
    elsif B = '0' then Z <= D;
    else Z <= E;
    end if;
  end process;
end test1;

```

(a) VHDL code for if example



(b) Synthesized VHDL code before optimization

The numeric\_bit and numeric\_std packages define the following overloaded operators:

- Unary: abs, -
- Arithmetic: +, -, \*, /, rem, mod
- Relational: >, <, >=, <=, =, /=
- Logical: not, and, or, nand, nor, xor, xnor
- Shifting: shift\_left, shift\_right, rotate\_left, rotate\_right, sll, srl, rol, ror

The unary operations require a single signed operand. The arithmetic, relational, and logical operators (except not) each require a left operand and a right operand. For arithmetic and relational operators, the following left and right operand pairs are acceptable: signed and signed, signed and integer, integer and signed, unsigned and unsigned, unsigned and natural, natural and unsigned. For logical operators, left and right operands must either both be signed or both unsigned.

If the left and right signed operands are of different lengths, the shortest operand will be sign-extended before performing an arithmetic operation. For unsigned operands, the shortest operand will be extended by filling in 0s on the left. For example,

```
signed: "01101" + "1011" becomes "01101" + "11011" = "01000"
unsigned: "01101" + "1011" becomes "01101" + "01011" = "11000"
```

When addition is performed on unsigned or signed operands, the final carry is discarded, and overflow is ignored. If a carry is needed, an extra bit can be added to one of the operands. For example,

```
constant A: unsigned(3 downto 0) := "1101";
constant B: signed(3 downto 0) := "1011";
variable Sumu: unsigned(4 downto 0);
variable Sums: signed(4 downto 0);
variable Overflow: boolean;
-----
Sumu := '0' & A + unsigned'("0101");
-- result is "10010" (sum = 2, carry = 1)
Sums := B(3)& B + signed'("1101");
-- result is "11000" (sum = -8, carry = 1)
Overflow := Sums(4) /= Sums(3); -- Overflow is false
```

In this example, the notation `unsigned' ("0101")` is a type qualification that assigns the type `unsigned` to the bit\_vector "0101".

The shifting operators require a signed or unsigned operand together with a shift count. A `shift_right` on an unsigned operand is with a 0 fill, and a right-shift on a signed operand is with a sign-extend. For example,

```
A = "1001"
unsigned:    shift_right(A, 2) = "0010"
signed:     shift_right(A, 2) = "1110"
```

The function *To\_Integer* converts Signed or Unsigned to Integer. The function *To\_Unsigned* converts Integer to Unsigned, and the function *To\_Signed* converts Integer to Signed.

CAD tools for synthesis have design libraries that include components to implement the operations defined in the numeric packages. The example of Figure 8-17 uses a standard logic arithmetic package. When this code is synthesized, the result includes library components that implement a 4-bit comparator, a 4-bit binary adder with a 4-bit accumulator register, and a 4-bit counter. Some synthesis tools will implement the counter with a 4-bit adder with an "0001" input and then optimize the result to eliminate unneeded gates.

**Figure 8-17 VHDL Code Example for Synthesis**

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity examples is
  port (signal clock: in bit;
        signal A, B: in signed(3 downto 0);
        signal ge: out boolean;
        signal acc: inout signed(3 downto 0) := "0000";
        signal count: inout unsigned(3 downto 0) := "0000");
end examples;

architecture x1 of examples is
begin
  ge <= (A >= B);                      -- 4-bit comparator
  process
  begin
    wait until clock'event and clock = '1';
    acc <= acc + B;                      -- 4-bit register and 4-bit adder
    count <= count + 1;                  -- 4-bit counter
  end process;
end;

```

## 8.9 SYNTHESIS EXAMPLES

In this section, we show the results of synthesizing some of the VHDL code we have previously written. We discuss what code changes were needed to achieve synthesis and compare the results of synthesis with our previous design. In testing the examples in this section, we used two different CAD systems: the Synopsis Design Compiler with the XACT libraries and the Altera Max-Plus II VHDL compiler.

The first example is the sequential machine of Figure 1-17, starting with the VHDL code of Figure 2-13. The corresponding synthesizable code is shown in Figure 8-18. We had to make a number of changes in the original code in order to complete the synthesis and obtain an economical result. First, we specified a range for the integer signals that represent *State* and *Nextstate*. In order to specify a state assignment, we replaced the integer

state names with constant names S0, S1, . . . , S6. Then we defined the constants to match the state assignment we used in Figure 1-18. We replaced each pair of if statements for X = '0' and X = '1' with a single if-then-else statement. Even though the VHDL code is correct for simulation, the synthesizer required that we replace `clock = '1'` with `clock = '1' and clock'event`.

Using the VHDL code in Figure 8-18, the circuit synthesized by the Synopsis Design Compiler using generic libraries requires 3 flip-flops and 13 gates, whereas the circuit designed in Chapter 1 requires only 7 gates. One reason for the discrepancy is that the optimizer apparently does not take don't care next states and outputs into account. We tried to improve the synthesis results by changing to IEEE standard logic and introducing don't cares, and we were able to reduce the number of gates to 12. Using the Xilinx libraries and choosing the XC4000 PGA as a target, two CLBs are required. The equivalent gate realization (Figure 8-19) requires 9 gates, one of which is an XNOR. Replacing the XNOR gate with 3 gates for comparison purposes, the result is 11 gates.

**Figure 8-18** VHDL Code for Synthesis of State Machine

```

entity SM1_2 is
  port(X, CLK: in bit;
       Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
  subtype s_type is integer range 0 to 7;
  signal State, Nextstate: s_type;
  constant S0: s_type := 0;                                -- state assignment
  constant S1: s_type := 4;
  constant S2: s_type := 5;
  constant S3: s_type := 7;
  constant S4: s_type := 6;
  constant S5: s_type := 3;
  constant S6: s_type := 2;
begin
  process(State,X)                                         --Combinational Network
  begin
    Z <= '0'; Nextstate <= S0;                            -- added to avoid latch
    case State is
      when S0 =>
        if X='0' then Z<='1'; Nextstate<=S1;
        else Z<='0'; Nextstate<=S2;  end if;
      when S1 =>
        if X='0' then Z<='1'; Nextstate<=S3;
        else Z<='0'; Nextstate<=S4;  end if;
      when S2 =>
        if X='0' then Z<='0'; Nextstate<=S4;
        else Z<='1'; Nextstate<=S5;  end if;
      when S3 =>
        if X='0' then Z<='0'; Nextstate<=S5;
        else Z<='1'; Nextstate<=S5;  end if;
    end case;
  end process;
end;

```

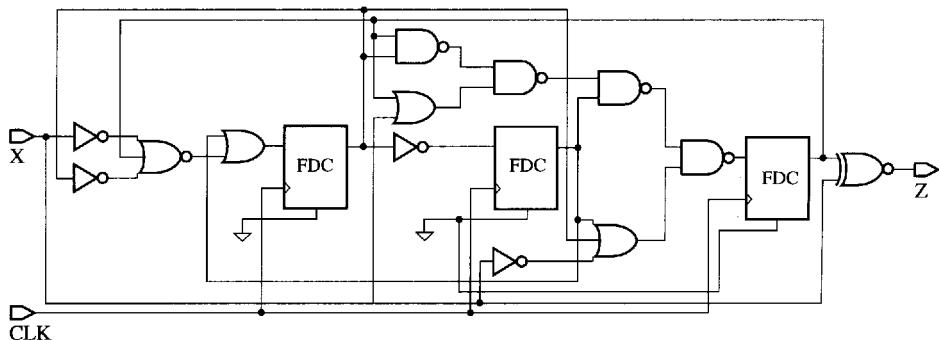
```

when S4 =>
  if X='0' then Z<='1'; Nextstate<=S5;
  else Z<='0'; Nextstate<=S6; end if;
when S5 =>
  if X='0' then Z<='0'; Nextstate<=S0;
  else Z<='1'; Nextstate<=S0; end if;
when S6 =>
  if X='0' then Z<='1'; Nextstate<=S0; end if;
when others => null;
end case;
end process;

process(CLK)                                -- State Register
begin
  if CLK='1' and CLK'event then      -- rising edge of clock
    State <= Nextstate;
  end if;
end process;
end Table;

```

Figure 8-19 State Machine Synthesized from Figure 8-18



We also synthesized the code of Figure 8-18 using the Altera Max-Plus II VHDL compiler with the MAX 7000 CPLD series as the target. The result was very inefficient, requiring 9 logic cells. We then changed the code by deleting the first line after `begin` in the first process and then replacing the code for `when S6` and `when others` with

```

when S6 =>
  if X = '0' then Z <= '0'; Nextstate <= S0;
  else Z <= '0'; Nextstate <= S0;
when others =>
  Z <= '0'; Nextstate <= S0;

```

The result of synthesis then required only four logic cells, which is the minimum number. As illustrated in this example, a small change in the code can result in a large change in the amount of resources used.

The second example is the dice game based on Figures 5-11 and 5-13. Using the Synopsis Design Compiler, synthesis of the counter from the VHDL code of Figure 5-24 was straightforward. We added **and** `Clk'event` after `Clk = '1'`. In order to synthesize the behavioral model of the dice game (Figure 5-15), we replaced `rising_edge(CLK)` with `CLK'event and CLK = '1'`, and we added

```
Nexstate <= '0' ;
```

to the third line of the first process to eliminate an unwanted latch. The synthesized version of the dice game including the counter requires 27.5 CLBs, which is the same as that derived from the schematic diagrams of Figures 6-15 through 6-17. If we substitute the data flow model of the dice-game controller for the behavioral model, the result requires only 25 CLBs, which is better than the one derived from schematic diagrams.

Next we synthesized the dice game using the Altera Max-Plus II compiler with the MAX 7000 series as a target. The synthesis results for the behavioral model were rather inefficient, requiring 39 logic cells and 34 shareable expanders. We obtained similar results for the data flow model, with the result requiring 38 logic cells and 25 shareable expanders. After examining the synthesizer output, we discovered the reason for the inefficient results. For each comparison in the VHDL code such as (`Sum = 11`), the synthesizer inserted a comparator. Even though one side of the comparator had constant inputs, no optimization was performed to remove unneeded gates. With this insight, we eliminated the comparisons from the VHDL code and used the following logic equations instead:

```
D7      <= Sum(2) and Sum(1) and Sum(0);
D711    <= (Sum(2) and Sum(1) and Sum(0))
          or (Sum(3) and Sum(1) and Sum(0));
D2312   <= (not Sum(3) and not Sum(2)) or (Sum(3) and Sum(2));
```

With this change, the synthesized circuit required only 31 logic cells and 13 shareable expanders, and it fit into a 7032.

The third synthesis example is based on the floating-point multiplier of Figure 7-2 and the VHDL code of Figure 7-5. Significant changes were required in order to produce good synthesis results. The revised VHDL code is shown in Figure 8-20. The original VHDL code used the `Addvec` procedure and the `Add4` function, which are not available in the synthesis libraries. For this reason, we changed the bit\_vectors to the UNSIGNED type so that we could use the overloaded "+" and "-" operators defined for this type. The UNSIGNED type is based on IEEE standard logic, and we also changed all bit logic to STD\_LOGIC type. Since the original VHDL could not be synthesized, the state registers were moved to the update process. This move eliminated the `wait for 0-ns` statements, which cannot be synthesized. As a result, combinational logic was left in two processes, so the sensitivity lists for these processes had to be modified to include all input signals. The output signals `V`, `Done`, and `F` were set to zero at the beginning of the process to eliminate unwanted latches.

When the VHDL code of Figure 8-20 was synthesized for the Xilinx XC4003 using the Synopsis Design Compiler, the result was 45 CLBs, which compares favorably with the results obtained from the schematics given in Section 7.2.

In order to synthesize the code of Figure 8-20 using the Altera software, further changes were required. In the *main\_control* process, we deleted the three lines after **begin**. For the case statement, we made appropriate changes within each when clause so that all control signals and outputs were assigned appropriate values. For example, the case when *PS1* equals 0 was changed to

```
when 0 => Result <= "00000000"; done <= '0'; V <= '0';
if St = '1' then Load <= '1'; NS1 = 1;
else NS1 <= 0; Load <= '0';
end if;
Adx <= '0'; SM8 <= '0'; RS0AB <= '0'; LSAB <= '0';
```

Similar changes were made in the *mul2c* process. Synthesis of the revised code using an EPM7096LC68 as a target and with the parallel expanders option turned on required 94 logic cells and 47 shareable expanders.

Figure 8-20 Revised VHDL Code for Floating-Point Multiplier

```
-- library BITLIB;
-- use BITLIB.bit_pack.all;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity FMUL is
  port (CLK, St: in std_logic;
        F1,E1,F2,E2: in UNSIGNED(3 downto 0);
        F: out UNSIGNED(6 downto 0);
        V, done:out std_logic);
end FMUL;

architecture FMULB of FMUL is
signal A, B, C: UNSIGNED(3 downto 0); --fraction registers
signal X, Y: UNSIGNED(4 downto 0); --exponent registers
signal Load, Adx, SM8, RSF, LSF: std_logic;
signal AdSh, Sh, Cm, Mdone: std_logic;
signal PS1, NS1: integer range 0 to 3; -- present and next state
signal State, Nextstate: integer range 0 to 4; -- multiplier control state

begin
main_control: process(PS1,St,Mdone,X,A,B)
  begin
    Load <= '0'; Adx <= '0'; NS1 <= 0; --clear control signals
    SM8 <= '0'; RSF <= '0'; LSF <= '0'; V <= '0'; F <= "0000000";
    done <= '0';
    case PS1 is
```

```

when 0 => F <= "0000000";                                --clear outputs
done<='0'; V <='0';
if St = '1' then Load <= '1'; NS1 <= 1; end if;
when 1 => Adx <= '1'; NS1 <= 2;
when 2 =>
  if Mdone = '1' then                                --wait for multiply
    if A = 0 then                                    --zero fraction
      SM8 <= '1';
    elsif A = 4 and B = 0 then
      RSF <= '1';                                 --shift AB right
    elsif A(2) = A(1) then                         --test for unnormalized
      LSF <= '1';                                 --shift AB left
    end if;
    NS1 <= 3;
  end if;
when 3 =>                                         --test for exp overflow
  if X(4) /= X(3) then V <= '1'; else V <= '0'; end if;
done <= '1';
F <= A(2 downto 0) & B;                            --output fraction
if ST = '0' then NS1<=0; end if;
end case;
end process main_control;

mul2c: process(State,Adx,B)                          --2's complement multiply
begin
AdSh <= '0'; Sh <= '0'; Cm <= '0'; Mdone <= '0';--clear control signals
Nextstate <= 0;
case State is
  when 0=>                                         --start multiply
    if Adx='1' then
      if B(0) = '1' then AdSh <= '1'; else Sh <= '1'; end if;
      Nextstate <= 1;
    end if;
  when 1 | 2  =>                                     --add/shift state
    if B(0) = '1' then AdSh <= '1'; else Sh <= '1'; end if;
    Nextstate <= State + 1;
  when 3  =>
    if B(0) = '1' then Cm <= '1'; AdSh <= '1'; else Sh <='1';
    end if;
    Nextstate <= 4;
  when 4  =>
    Mdone <= '1'; Nextstate <= 0;
end case;
end process mul2c;

```

```

update: process                                --update registers
variable addout: UNSIGNED(3 downto 0);
begin
wait until (CLK = '1' and CLK'event);
PS1 <= NS1;
State <= Nextstate;
if Cm = '0' then addout := A + C;
else addout := A - C; end if;                  --add 2's comp. of C
if Load = '1' then X <= E1(3)&E1; Y <= E2(3)&E2;
A <= "0000"; B <= F1; C <= F2; end if;
if ADX = '1' then X <= X + Y; end if;
if SM8 = '1' then X <= "11000"; end if;
if RSF = '1' then A <= '0'&A(3 downto 1);
B <= A(0)&B(3 downto 1);
X <= X + 1; end if;                           -- increment X
if LSF = '1' then
A <= A(2 downto 0)&B(3); B <= B(2 downto 0)&'0';
X <= X + 31; end if;                          -- decrement X
if AdSh = '1' then
A <= (C(3) xor Cm) & addout(3 downto 1);    -- load shifted adder
B <= addout(0) & B(3 downto 1); end if;       -- output into A & B
if Sh = '1' then
A <= A(3) & A(3 downto 1);                   -- right shift A & B
B <= A(0) & B(3 downto 1);                   -- with sign extend
end if;
end process update;
end FMULB;

```

## 8.10 FILES AND TEXTIO

This section introduces file input and output in VHDL. Files are frequently used with test benches to provide a source of test data and to provide storage for test results. VHDL provides a standard *TEXTIO* package that can be used to read or write lines of text from or to a file.

Before a file is used, it must be declared using a declaration of the form

```
file file-name: file-type [open mode] is "file-pathname";
```

For example,

```
file test_data: text open read_mode is "c:\test1\test.dat"
```

declares a file named *test\_data* of type *text* that is opened in the *read\_mode*. The physical location of the file is in the *test1* directory on the *c:* drive.

A file can be opened in *read\_mode*, *write\_mode*, or *append\_mode*. In *read\_mode*, successive elements in the file can be read using the *read* procedure. When a file is opened in *write\_mode*, a new empty file is created by the host computer's file system, and successive

data elements can be written to the file using the write procedure. To write to an existing file, the file should be opened in the append\_mode.

A file can contain only one type of object, such as integers, bit-vectors, or text strings, as specified by the file type. For example, the declaration

```
type bv_file is file of bit_vector;
```

defines bv\_file to be a file type that can contain only bit\_vectors. Each file type has an associated implicit endfile function. A call of the form

```
endfile(file_name)
```

returns TRUE if the file pointer is at the end of the file.

The standard *TEXTIO* package that comes with VHDL contains declarations and procedures for working with files composed of lines of text. The package specification for *TEXTIO* (see Appendix C) defines a file type named text:

```
type text is file of string;
```

The *TEXTIO* package contains procedures for reading lines of text from a file of type text and for writing lines of text to a file.

Procedure readline reads a line of text and places it in a buffer with an associated pointer. The pointer to the buffer must be of type line, which is declared in the *TEXTIO* package as

```
type line is access string;
```

When a variable of type line is declared, it creates a pointer to a string. The code

```
variable buff: line;  
...  
readline (buff, test_data);
```

reads a line of text from test\_data and places it in a buffer that is pointed to by *buff*. After reading a line into the buffer, we must call a version of the read procedure one or more times to extract data from the line buffer. The *TEXTIO* package provides overloaded read procedures to read data of types bit, bit\_vector, boolean, character, integer, real, string, and time from the buffer. For example, if *bv4* is a bit\_vector of length four, the call

```
read(buff, bv4);
```

extracts a 4-bit vector from the buffer, sets *bv4* equal to this vector, and adjusts the pointer *buff* to point to the next character in the buffer. Another call to read then extracts the next data object from the line buffer.

A call to read may be of one of two forms:

```
read (pointer, value);
read (pointer, value, good);
```

where *pointer* is of type line and *value* is the variable into which we want to read the data. In the second form, *good* is a boolean that returns TRUE if the read is successful and FALSE if it is not. The size and type of *value* determines which of the read procedures in the *TEXTIO* package is called. For example, if *value* is a string of length five, then a call to read reads the next five characters from the line buffer. If *value* is an integer, a call to read skips over any spaces and then reads decimal digits until a space or other nonnumeric character is encountered. The resulting string is then converted to an integer. Characters, strings, and bit-vectors within files of type text are not delimited by quotes.

To write lines of text to a file, we must call a version of the write procedure one or more times to write data to a line buffer and then call writeline to write the line of data to a file. The *TEXTIO* package provides overloaded write procedures to write data of types bit, bit\_vector, boolean, character, integer, real, string, and time to the buffer. For example, the code

```
variable buffw: line;
variable int1: integer;
variable bv8: bit_vector(7 downto 0);
...
write (buffw, int1, right, 6);
write (buffw, bv8, right, 10);
writeln (buffw, output_file);
```

converts *int1* to a text string, writes this string to the line buffer pointed to by *buffw*, and adjusts the pointer. The text will be right-justified in a field six characters wide. The second call to write puts the bit\_vector *bv8* in a line buffer, and adjusts the pointer. The 8-bit vector will be right-justified in a field ten characters wide. Then writeln writes the buffer to the *output\_file*. Each call to write has four parameters: (1) a buffer pointer of type line, (2) a value of any acceptable type, (3) justification (left or right), which specifies the location of the text within the output field, and (4) field\_width, an integer that specifies the number of characters in the field.

As an example, we write a procedure to read data from a file and store the data in a memory array. This procedure will later be used to load instruction codes into a memory module for a computer system. The computer system can then be tested by simulating the execution of the instructions stored in memory. The data in the file will be of the following format:

address *N* comments

byte1 byte2 byte3 ... byte*N* comments

The address consists of four hexadecimal digits, and *N* is an integer that indicates the number of bytes of code that will be on the next line. Each byte of code consists of two

hexadecimal digits. Each byte is separated by one space, and the last byte must be followed by a space. Anything following the last space will not be read and will be treated as a comment. The first byte should be stored in the memory array at the given address, the second byte at the next address, and so forth. For example, consider the following file:

```
12AC 7 (7 hex bytes follow)
AE 03 B6 91 C7 00 0C (LDX imm, LDA dir, STA ext)
005B 2 (2 hex bytes follow)
01 FC<space>
```

When the *fill\_memory* procedure is called using this file as an input, AE is stored in 12AC, 03 in 12AD, B6 in 12AE, 91 in 12AF, etc.

Figure 8-21 gives VHDL code that calls the procedure *fill\_memory* to read data from a file and store it in an array named *mem*. Since *TEXTIO* does not include a read procedure for hex numbers, the procedure *fill\_memory* reads each hex value as a string of characters and then converts the string to an integer. Conversion of a single hex digit to an integer value is accomplished by table lookup. The constant named *lookup* is an array of integers indexed by characters in the range '0' to 'F'. This range includes the 23 ASCII characters: '0', '1', '2', . . . , '9', ':', ';', '<', '=', '>', '?', '@', 'A', 'B', 'C', 'D', 'E', 'F'. The corresponding array values are 0, 1, 2, . . . , 9, -1, -1, -1, -1, -1, -1, 10, 11, 12, 13, 14, 15. The -1 could be replaced with any integer value, since the seven special characters in the index range should never occur in practice. Thus, *lookup*('2') is the integer value 2, *lookup*('C') is 12, and so forth.

Procedure *fill\_memory* calls *readline* to read a line of text that contains a hex address and an integer. The first call to read reads the address string from the line buffer, and the second call to read reads an integer, which is the byte count for the next line. The integer *addr1* is computed using the lookup table for each character in the address string. The next line of text is read into the buffer, and a loop is used to read each byte. Since *data\_s* is three characters long, each call to read reads two hex characters and a space. The hex characters are converted to an integer and then to a *std\_logic\_vector*, which is stored in the memory array. The address is incremented before reading and storing the next byte. The procedure exits when the end of file is reached. Another example of using *TEXTIO* is given in Figure 9-28.

**Figure 8-21 VHDL Code to Fill a Memory Array from a File**

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;      -- CONV_STD_LOGIC_VECTOR(int, size)
use std.textio.all;

entity testfill is
end test;

architecture fillmem of testfill is
type RAMtype is array (0 to 2047) of std_logic_vector(7 downto 0);
signal mem: RAMtype:= (others=>(others=> '0'));

procedure fill_memory(signal mem: inout RAMType) is
type HexTable is array(character range <>) of integer;
-- valid hex chars: 0, 1, ... A, B, C, D, E, F (upper-case only)
constant lookup : HexTable('0' to 'F'):=
  (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -1, -1, -1,
   -1, -1, -1, -1, 10, 11, 12, 13, 14, 15);
file infile: text is open read_mode "mem1.txt";      -- open file for
reading
-- file infile: text is in "mem1.txt"; -- VHDL '87 version
variable buff: line;
variable addr_s: string(4 downto 1);
variable data_s : string(3 downto 1); -- data_s(1) has a space
variable addr, byte_cnt: integer;
variable data: integer range 255 downto 0;
begin
  while (not endfile(infile)) loop
    readline (infile, buff);
    read (buff, addr_s);          -- read addr hexnum
    read(buff, byte_cnt);        -- read number of bytes to read
    addr1 := lookup(addr_s(4))*4096 + lookup(addr_s(3))*256
           + lookup(addr_s(2))*16 + lookup(addr_s(1));
    readline (infile, buff);
    for i in 1 to byte_cnt loop
      read (buff, data_s); -- read 2 digit hex data and a space
      data:= lookup(data_s(3))*16 + lookup(data_s(2));
      mem(addr1) <= CONV_STD_LOGIC_VECTOR(data, 8);
      addr:= addr1 + 1;
    end loop;
  end loop;
end fill_memory;

begin
  testbench: process begin
    fill_memory(mem);
    -- insert code which uses memory data
  end process;
end fillmem;

```

This chapter has introduced several important features of VHDL. Attributes associated with signals allow checking of setup and hold times and other timing specifications. Attributes associated with arrays allow us to write procedures that do not depend on the manner in which the arrays are indexed. Proper use of inertial and transport delays enables us to more accurately model the types of delays that occur in real systems. Operator overloading can be used to extend the definition of VHDL operators so that they can be used with different types of operands. Multivalued logic and the associated resolution functions allow us to model tristate busses and other systems where a signal is driven from more than one source. The IEEE Standard 1164 defines a system of 9-valued logic that is widely used with VHDL. Generics enable us to specify parameter values for a component when the component is instantiated. Generate statements provide an efficient way to describe systems that have an iterative structure. The *TEXTIO* package provides a convenient way of doing file input and output. Many of the VHDL features described in this chapter are used in the memory and bus models developed in Chapter 9.

This chapter has introduced the synthesis of digital systems directly from the VHDL code. In general, VHDL code must be written in specific ways in order for the code to be synthesizable, and the way in which the code is written can have a major effect on the amount of logic produced by the synthesis tool. The use of standard packages for signed and unsigned arithmetic facilitates the writing of synthesizable code. Two major examples of synthesis of digital systems from VHDL code are given in Chapter 11.

---

## Problems

- 8.1** Write a VHDL function that will take two integer vectors, *A* and *B*, and find the dot product  $C = \sum a_i * b_i$ . The function call should be of the form `DOT(A, B)`, where *A* and *B* are integer vector signals. Use attributes inside the function to determine the length and ranges of the vectors. Make no assumptions about the high and low values of the ranges. For example:

$$A(3 \text{ downto } 1) = (1, 2, 3), \quad B(3 \text{ downto } 1) = (4, 5, 6), \quad C = 3 * 4 + 2 * 5 + 1 * 6 = 32$$

Output a warning if the ranges are not the same.

- 8.2** Write a VHDL description of an address decoder. One input to the address decoder is an 8-bit address, which can have any range with a length of 8, for example: `bit_vector addr(8 to 15)`. The second input is `check : x01z_vector(5 downto 0)`. The address decoder will output `Sel = '1'` if the upper 6 bits of the 8-bit address match the 6-bit check vector. For example, if `addr = "10001010"` and `check = "1000XX"` then `Sel = '1'`. Only the 6 leftmost bits of `addr` will be compared; the remaining bits are ignored. An 'X' in the check vector is treated as a don't care.

- 8.3** A VHDL entity has inputs *A* and *B*, and outputs *C* and *D*. *A* and *B* are initially high. Whenever *A* goes low, *C* will go high 5 ns later, and if *A* changes again, *C* will change 5 ns later. *D* will change if *A* has not had any transactions for 5 ns.

- Write the VHDL architecture with a process that determines the outputs *C* and *D*.
- Write another process to check that *B* is stable 2 ns before and 1 ns after *A* goes high. The process should also report an error if *B* goes low for a time interval less than 10 ns.

- 8.4** Write an overloading function for the "`<`" operator for bit vectors. Return a boolean TRUE if *A* is less than *B*, otherwise FALSE. Report an error if the bit vectors are of different lengths.

**8.5** An open-collector bus with pull-up resistors can be modeled using IEEE standard logic. A pull-up resistor acts like a weak '1'. Write a VHDL model for a buffer that accepts IEEE std\_logic as an input and outputs a std\_logic signal that can drive an open-collector bus.

**8.6** Consider the following concurrent statements in VHDL:

```
A <= transport '1' after 5 ns, '0' after 10 ns, 'Z' after
      15 ns;
B <= transport 'X' after 8 ns, '0' after 4 ns, '1' after 12
      ns, 'Z' after 10 ns;
C <= A after 6 ns;
C <= transport A after 5 ns;
C <= reject 3 ns B after 4 ns;
```

- (a) Draw drivers (see Figure 2.12) for signals *A* and *B*.
- (b) Draw the three drivers *s*0, *s*1 and *s*2 for *C* (similar to Figure 8-7), and draw a timing chart for each.
- (c) List the value for *C* each time it is resolved by the drivers, and draw a timing chart for *C*.

**8.7** Write a VHDL model for one flip-flop in a 74HC374 (octal D-type flip-flop with 3-state outputs). Use the IEEE-standard 9-valued logic package. Assume that all logic values are 'x', '0', '1' or 'z'. Check setup, hold, and pulse width specs using assert statements. Unless the output is 'z', the output should be 'x' if *CLK* or *OC* is 'x', or if an 'x' has been stored in the flip-flop.

**8.8** Write a VHDL function to compare two IEEE std\_logic\_vectors to see if they are equal. Report an error if any bit in either vector is not '0', '1', or '-' (don't care), or if the lengths of the vectors are not the same. The function call should pass only the vectors. The function should return TRUE if the vectors are equal, else FALSE. When comparing the vectors, consider that '0' = '-', and '1' = '-'. Make no assumptions about the index range of the two vectors (for example, one could be 1 to 7 and the other 8 downto 0).

**8.9** Write a VHDL model for an *N*-bit comparator using an iterative network. In the entity, use the generic parameter *N* to define the length of the input bit-vectors *A* and *B*. The comparator outputs should be *EQ* = '1' if *A* = *B*, and *GT* = '1' if *A* > *B*. Use a for loop to do the comparison on a bit-by-bit basis, starting with the high-order bits. Even though the comparison is done on a bit-by-bit basis, the final values of *EQ* and *GT* apply to *A* and *B* as a whole.

**8.10** Write a VHDL model for an *N*-bit bidirectional shift register using a generate statement. *N* is a generic parameter that defines the length of the register (default length = 8). Define a component that represents one bit of the shift register. The component port should be

```
port (L, R, CLR, CLK, Pin, Lin, Rin: in bit; Q: out bit);
```

*L* and *R* are control inputs that operate as follows:

*LR* = 00, do nothing; *LR* = 01, shift right; *LR* = 10, shift left; *LR* = 11, parallel load

*CLR* is a direct clear input, *Pin* is the parallel input, *Lin* is the serial input for left shift, and *Rin* is the serial input for right shift.

**8.11**

- (a) Write a model for a D flip-flop with a direct clear input. Use the following generic timing parameters:  $t_{plh}$ ,  $t_{phl}$ ,  $t_{sh}$ ,  $t_h$ , and  $t_{cmin}$ . The minimum allowable clock period is  $t_{cmin}$ . Report appropriate errors if timing violations occur.
- (b) Write a test bench to test your model. Include tests for every error condition.

**8.12**

- (a) Make any necessary changes in the VHDL code for the traffic light controller (Figure 3-19) so that it can be synthesized using whatever synthesis tool you have available. Synthesize the code using a suitable FPGA or CPLD as a target.
- (b) Using the same synthesis tool and target device as in (a), try to find a more efficient implementation of traffic light controller that uses fewer logic cells in the target device. For example, try different state assignments, or try writing the VHDL code in terms of logic equations.

- 8.13** Figure 4-23 gives the VHDL code for a 32-bit signed divider. Make necessary changes in the code so it can be synthesized using whatever synthesis tool you have available. Instead of using the procedure *Addvec*, use an appropriate arithmetic package. Synthesize the code using a suitable FPGA or CPLD as a target. Try different options on your synthesis tool, such as optimize for speed and optimize for area, and compare the results.

# CHAPTER 9

## VHDL MODELS FOR MEMORIES AND BUSSES

---

In this chapter we first describe the operation of a static RAM memory and develop VHDL models that represent the operation and timing characteristics of that memory. Then we describe the operation of a microprocessor bus interface and develop a VHDL timing model for it. Finally, we design an interface between memory and the microprocessor bus. We will use the VHDL memory and bus models to verify that the timing specifications for the memory and bus interface have been satisfied.

### 9.1 STATIC RAM MEMORY

Read-only memories were discussed in Chapter 3, and now we discuss read-write memories. Such memories are usually referred to as RAMs. RAM stands for random-access memory, which means that any word in memory can be accessed in the same amount of time as any other word. Strictly speaking, ROM memories are also random-access, but the term RAM is normally applied only to read-write memories. Figure 9-1 shows the block diagram of a static RAM with  $n$  address lines,  $m$  data lines, and three control lines. This memory can store  $2^n$  words, each  $m$  bits wide. The data lines are bidirectional in order to reduce the required number of pins and the package size of the memory chip. When reading from the RAM, the data lines are outputs; when writing to the RAM, the data lines serve as inputs. The three control lines function as follows:

$\overline{\text{CS}}$

When asserted low, chip select selects the memory chip so that memory read and write operations are possible.

$\overline{\text{OE}}$

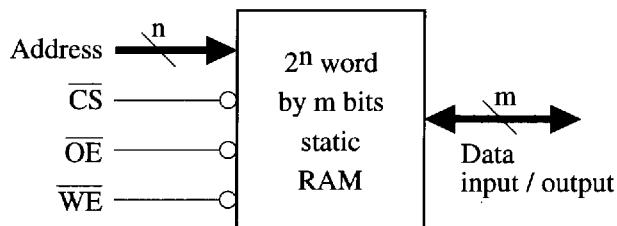
When asserted low, output enable enables the memory output onto an external bus.

$\overline{\text{WE}}$

When asserted low, write enable allows data to be written to the RAM.

(We say that a signal is asserted when it is in its active state. An active-low signal is asserted when it is low, and an active-high signal is asserted when it is high.)

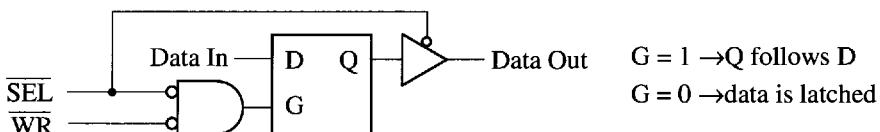
Figure 9-1 Block Diagram of Static RAM



The term static RAM means that once data is stored in the RAM, the data remains there until the power is turned off. This is in contrast with a dynamic RAM, which requires that the memory be refreshed on a periodic basis to prevent data loss. A detailed discussion of dynamic RAMs is beyond the scope of this book.

The RAM contains address decoders and a memory array. The address inputs to the RAM are decoded to select cells within the RAM. Figure 9-2 shows the functional equivalent of a static RAM cell that stores one bit of data. The cell contains a transparent D latch, which stores the data. When  $\overline{SEL}$  is asserted low and  $\overline{WR}$  is high,  $G = 0$ , the cell is in the read mode, and *Data Out* = *Q*. When  $\overline{SEL}$  and  $\overline{WR}$  are both low,  $G = 1$  and data can enter the transparent latch. When either  $SEL$  and  $WR$  goes high, the data is stored in the latch. When  $\overline{SEL}$  is high, *Data Out* is high-Z.

Figure 9-2 Functional Equivalent of a Static RAM Cell



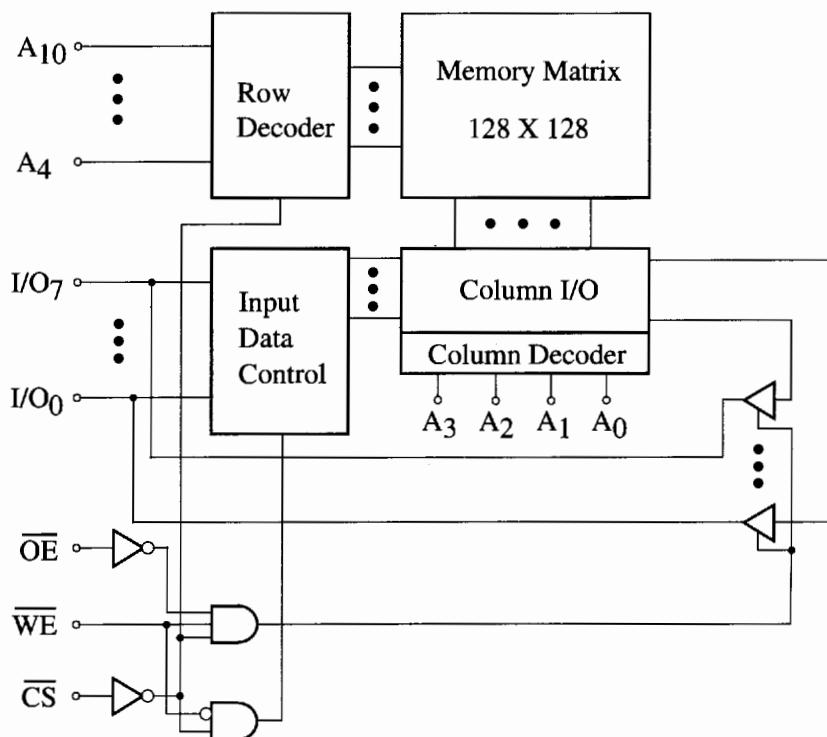
Static RAMs are available that can store up to several million bytes of data. For purposes of illustration, we describe a 6116 static CMOS RAM that can store 2K bytes of data, but the principles illustrated here also apply to large static RAMs. Figure 9-3 shows the block diagram of a 6116 static RAM, which can store 2048 8-bit words of data. This memory has 16,384 cells, arranged in a  $128 \times 128$  memory matrix. The 11 address lines, which are needed to address the  $2^{11}$  bytes of data, are divided into two groups. Lines A10 through A4 select one of the 128 rows in the matrix. Lines A3 through A0 select 8 columns in the matrix at a time, since there are 8 data lines. The data outputs from the matrix go through tristate buffers before connecting to the data I/O lines. These buffers are disabled except when reading from the memory.

The truth table for the RAM (Table 9-1) describes its basic operation. High-Z in the I/O column means that the output buffers have high-Z outputs, and the data inputs are not used. In the read mode, the address lines are decoded to select eight of the memory cells, and the data comes out on the I/O lines after the memory access time has elapsed. In the write mode, input data is routed to the latch inputs in the selected memory cells when  $\overline{WE}$  is low, but writing to the latches in the memory cells is not completed until either  $\overline{WE}$  goes high or the chip is deselected. The truth table does not take memory timing into account.

Table 9-1 Truth Table for Static RAM

$\overline{CS}$	$\overline{OE}$	$\overline{WE}$	Mode	I/O pins
H	X	X	not selected	high-Z
L	H	H	output disabled	high-Z
L	L	H	read	data out
L	X	L	write	data in

Figure 9-3 Block Diagram of 6116 Static RAM

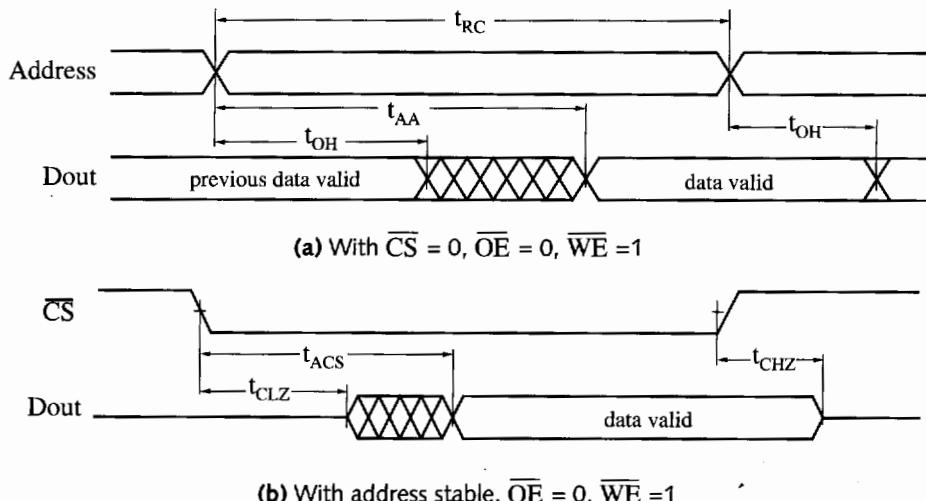


Memory timing diagrams and specifications must be considered when designing an interface to the memory.

Figure 9-4(a) shows the read cycle timing for the case where  $\overline{CS}$  and  $\overline{OE}$  are both low before the address changes. In this case, after the address changes, the old data remains at the memory output for a time  $t_{OH}$ ; then there is a transition period during which the data may change (as indicated by the cross-hatching). The new data is stable at the memory output after the address access time,  $t_{AA}$ . The address must be stable for the read cycle time,  $t_{RC}$ .

Figure 9-4(b) shows the timing for the case where the  $\overline{OE}$  is low and the address is stable before  $\overline{CS}$  goes low. When  $\overline{CS}$  is high,  $D_{out}$  is in the high-Z state, as indicated by a line halfway between '0' and '1'. When  $\overline{CS}$  goes low,  $D_{out}$  leaves high-Z after time  $t_{CLZ}$ ; there is a transition period during which the data may change, and the new data is stable at time  $t_{ACS}$  after  $\overline{CS}$  changes.  $D_{out}$  returns to high-Z at time  $t_{CHZ}$  after  $\overline{CS}$  goes high.

Figure 9-4 Read Cycle Timing



The timing parameters for CMOS static RAMs are defined in Table 9-2 for both read and write cycles. Specifications are given for the 6116-2 RAM, which has a 120-ns access time, and also for the 43258A-25 RAM, which has an access time of 25 ns. A dash in the table indicates that either the specification was not relevant or that the manufacturer did not provide the specification.

Table 9-2 Timing Specifications for Two Static CMOS RAMs

Parameter	Symbol	6116-2		43258A-25	
		min	max	min	max
Read cycle time	$t_{RC}$	120	—	25	—
Address access time	$t_{AA}$	—	120	—	25
Chip select access time	$t_{ACS}$	—	120	—	25
Chip selection to output in low-Z	$t_{CLZ}$	10	—	3	—
Output enable to output valid	$t_{OE}$	—	80	—	12
Output enable to output in low-Z	$t_{OLZ}$	10	—	0	—
Chip deselection to output in high-Z	$t_{CHZ}$	10*	40	3*	10
Chip disable to output in high-Z	$t_{OHZ}$	10*	40	3*	10
Output hold from address change	$t_{OH}$	10	—	3	—
Write cycle time	$t_{WC}$	120	—	25	—
Chip selection to end of write	$t_{CW}$	70	—	15	—
Address valid to end of write	$t_{AW}$	105	—	15	—
Address setup time	$t_{AS}$	0	—	0	—
Write pulse width	$t_{WP}$	70	—	15	—
Write recovery time	$t_{WR}$	0	—	0	—
Write enable to output in high-Z	$t_{WHZ}$	10*	35	3*	10
Data valid to end of write	$t_{DW}$	35	—	12	—
Data hold from end of write	$t_{DH}$	0	—	0	—
Output active from end of write	$t_{OW}$	10	—	0	—

\*Estimated value, not specified by manufacturer.

Figure 9-5 shows the write cycle timing for the case where  $\overline{OE}$  is low during the entire cycle and where writing to memory is controlled by  $\overline{WE}$ . In this case, it is assumed that  $\overline{CS}$  goes low before or at the same time as  $\overline{WE}$  goes low, and  $\overline{WE}$  goes high before or at the same time as  $\overline{CS}$  does. The cross-hatching on  $\overline{CS}$  indicates the interval in which it can go from high to low (or from low to high). The address must be stable for the address setup time,  $t_{AS}$ , before  $\overline{WE}$  goes low. After time  $t_{WHZ}$ , the data out from the tristate buffers go to the high-Z state and input data may be placed on the I/O lines. The data into the memory must be stable for the setup time  $t_{DW}$  before  $\overline{WE}$  goes high, and then it must be kept stable for the hold time  $t_{DH}$ . The address must be stable for  $t_{WR}$  after  $\overline{WE}$  goes high. When  $\overline{WE}$  goes high, the memory switches back to the read mode. After  $t_{OW}$  (min) and during region (a),  $Dout$  goes through a transition period and then becomes the same as the data just stored in the memory. Further change in  $Dout$  may occur if the address changes or if  $\overline{CS}$  goes high. To avoid bus conflicts during region (a),  $Din$  should either be high-Z or the same as  $Dout$ .

Figure 9-5  $\overline{WE}$ -controlled Write Cycle Timing ( $\overline{OE} = 0$ )

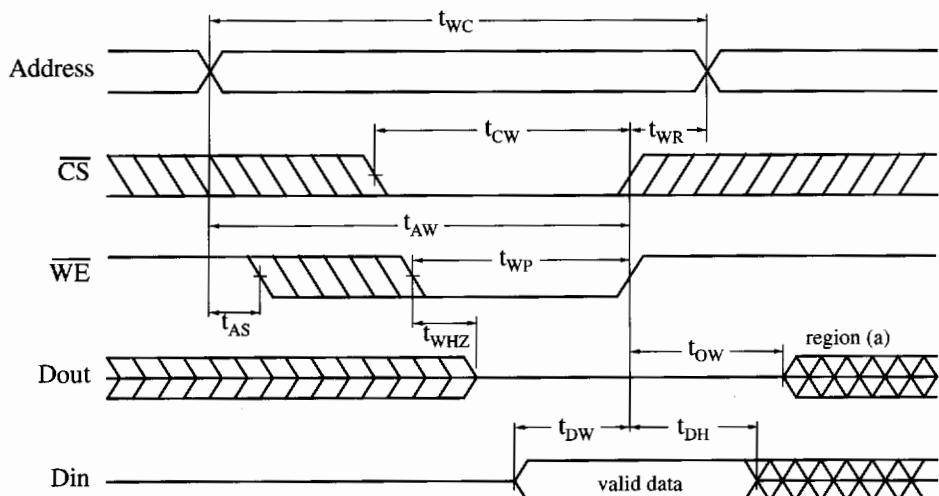
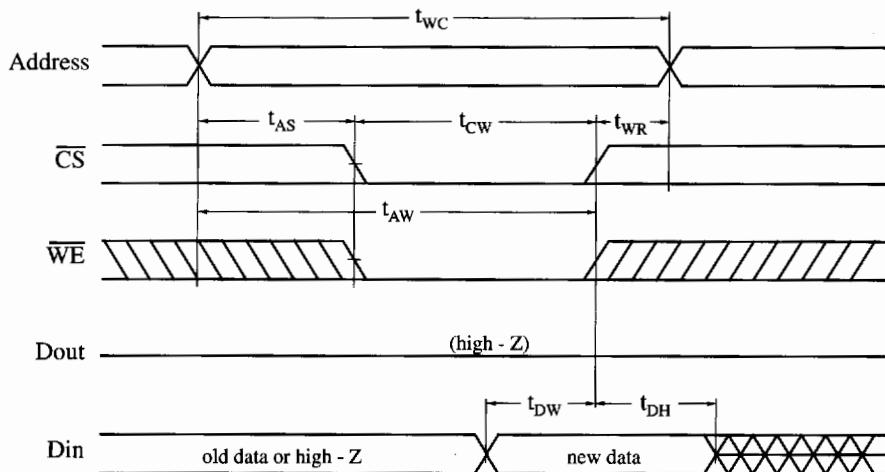


Figure 9-6 shows the write cycle timing for the case where  $\overline{OE}$  is low during the entire cycle and where writing to memory is controlled by  $\overline{CS}$ . In this case, it is assumed that  $\overline{WE}$  goes low before or at the same time as  $\overline{CS}$  goes low, and  $\overline{CS}$  goes high before or at the same time as  $\overline{WE}$  does. The address must be stable for the address setup time,  $t_{AS}$ , before  $\overline{CS}$  goes low. The data into the memory must be stable for the setup time  $t_{DW}$  before  $\overline{CS}$  goes high, and then it must be kept stable for the hold time  $t_{DH}$ . The address must be stable for  $t_{WR}$  after  $\overline{CS}$  goes high. Note that this write cycle is very similar to the  $\overline{WE}$ -controlled cycle. In both cases, writing to memory occurs when both  $\overline{CS}$  and  $\overline{WE}$  are low, and writing is completed when either one goes high.

Figure 9-6  $\overline{CS}$ -controlled Write Cycle Timing ( $\overline{OE} = 0$ )



We now write a simple VHDL model for the memory that does not take timing considerations into account. Then we add timing information to obtain a more accurate model, and we write a process to check some of the more important timing specifications. We assume that  $\overline{OE}$  is permanently tied low, so it will not appear in the models. We also assume that timing is such that writing to memory is controlled by  $\overline{WE}$ . To further simplify the model, we have reduced the number of address lines to 8 and the size of the memory to 256 bytes. We will model only the external behavior of the memory and make no attempt to model the internal behavior. In the VHDL code we use  $WE\_b$  to represent  $\overline{WE}$  ( $WE$ -bar).

In Figure 9-7, the RAM memory array is represented by an array of standard logic vectors ( $RAM1$ ). Since *Address* is typed as a bit-vector, it must be converted to an integer in order to index the memory array. The RAM process sets the I/O lines to high-Z if the chip is not selected. Otherwise, the data on the I/O lines is stored in  $RAM1$  on the rising edge of  $We\_b$ . If *Address* and  $We\_b$  change simultaneously, the old value of *Address* should be used. *Address'delayed* is used as the array index to delay *Address* by one delta to make sure that the old address is used. The wait for 0 ns is needed so that the data will be stored in the RAM before it is read back out. If  $We\_b = '1'$ , the RAM is in the read mode, and I/O is the data read from the memory array. If  $We\_b = '0'$ , the memory is in the write mode, and the I/O lines are driven to high-Z so external data can be supplied to the RAM.

**Figure 9-7 Simple Memory Model**

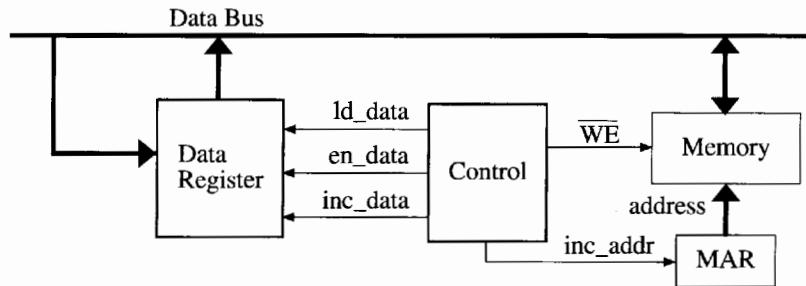
```
-- Simple memory model
library IEEE;
use IEEE.std_logic_1164.all;
library BITLIB;
use BITLIB.bit_pack.all;

entity RAM6116 is
    port(Cs_b, We_b: in bit;
          Address: in bit_vector(7 downto 0);
          IO: inout std_logic_vector(7 downto 0));
end RAM6116;

architecture simple_ram of RAM6116 is
    type RAMtype is array(0 to 255) of std_logic_vector(7 downto 0);
    signal RAM1: RAMtype:=(others=>(others=>'0'));
                                -- Initialize all bits to '0'
begin
    process
    begin
        if Cs_b = '1' then IO <= "ZZZZZZZZ";      -- chip not selected
        else
            if We_b'event and We_b = '1' then -- rising-edge of We_b
                RAM1(vec2int(Address'delayed)) <= IO; -- write
                wait for 0 ns;                      -- wait for RAM update
            end if;
            if We_b = '1' then
                IO <= RAM1(vec2int(Address));       --read
            else IO <= "ZZZZZZZZ";                  --drive high-Z
            end if;
        end if;
        wait on We_b, Cs_b, Address;
    end process;
end simple_ram;
```

To test the RAM model, we implement the system shown in Figure 9-8. This system has a memory address register (MAR) that holds the memory address and a data register to store data read from the memory. The system reads a word from the RAM, loads it into the data register, increments the data register, stores the result back in the RAM, and then increments the memory address. This process continues until the memory address equals 8. Required control signals are *ld\_data* (load data register from Data Bus), *en\_data* (enable data register output onto Data Bus), *inc\_data* (increment Data Register), and *inc\_addr* (increment MAR). Figure 9-9 shows the SM chart for the system. The memory data is loaded in Data Register during the transition to S1. Data Register is incremented during the transition to S2.  $\overline{WE}$  is an active-low signal, which is asserted low only in S2, so that  $\overline{WE}$  is high in the other states. Thus, writing to the RAM is initiated in S2 and completed on the rising edge of  $\overline{WE}$ , which occurs during the transition from S2 to S3.

**Figure 9-8 Block Diagram of RAM System**



**Figure 9-9 SM Chart for RAM System**

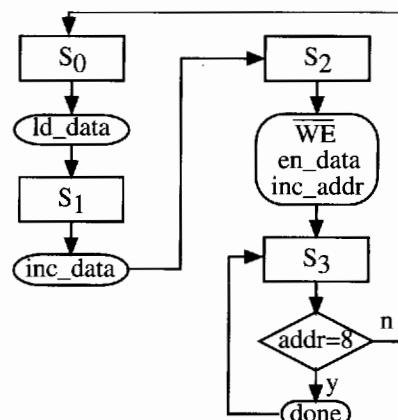


Figure 9-10 shows the VHDL code for the RAM system. The first process represents the SM chart, and the second process is used to update the registers on the rising edge of the clock. A short delay is added when the address is incremented to make sure the write to memory is completed before the address changes. A concurrent statement is used to simulate the tristate buffer, which enables the data register output onto the I/O lines.

**Figure 9-10 Tester for Simple Memory Model**

```

-- Tester for simple ram model
library ieee;
use ieee.std_logic_1164.all;
library bitlib;
use bitlib.bit_pack.all;

entity RAM6116_system is
end RAM6116_system;

```

```
architecture RAMtest of RAM6116_system is
component RAM6116 is
    port(Cs_b, We_b: in bit;
          Address: in bit_vector(7 downto 0);
          IO: inout std_logic_vector(7 downto 0));
end component RAM6116;
signal state, next_state: integer range 0 to 3;
signal inc_adrs, inc_data, ld_data, en_data, Cs_b, clk, done: bit;
signal We_b: bit := '1';                                -- initialize to read mode
signal Data: bit_vector(7 downto 0);                  -- data register
signal Address: bit_vector(7 downto 0);                -- address register
signal IO: std_logic_vector(7 downto 0);              -- I/O bus

begin
    RAM1: RAM6116 port map(Cs_b, We_b, Address, IO);
control: process(state, Address)
begin
    --initialize all control signals (RAM always selected)
    ld_data<='0'; inc_data<='0'; inc_adrs<='0'; en_data <='0';
    done <= '0'; We_b <='1'; Cs_b <= '0';
    --start SM chart here
    case (state) is
        when 0 =>    ld_data <= '1'; next_state <= 1;
        when 1 =>    inc_data <= '1'; next_state <= 2;
        when 2 =>    We_b <= '0'; en_data <= '1'; inc_adrs <= '1';
        when 3 =>    if (Address = "00001000") then done <= '1';
                        else next_state <= 0;
                        end if;
    end case;
end process control;

--The following process is executed on the rising edge of a clock.
register_update: process
begin
    wait until clk = '1';
    state <= next_state;
    if (inc_data = '1') then data <= int2vec(vec2int(data)+1,8); end if;
    if (ld_data = '1') then data <= To_bitvector(IO); end if;
    if (inc_adrs = '1') then
        Address <= int2vec(vec2int(Address)+1,8) after 1 ns;
        -- delay added to allow completion of memory write
    end if;
end process register_update;

-- Concurrent statements
clk <= not clk after 100 ns;
IO <= To_StdLogicVector(data) when en_data = '1'
    else "ZZZZZZZZ";
end RAMtest;
```

Next, we revise the RAM model to include timing information based on the read and write cycles shown in Figures 9-4, 9-5, and 9-6. We still assume that  $\overline{OE} = 0$ . The VHDL RAM timing model in Figure 9-11 uses a generic declaration to define default values for the important timing parameters. Transport delays are used throughout to avoid cancellation problems, which can occur with inertial delays. The RAM process waits for a change in  $CS\_b$ ,  $WE\_b$ , or the address. If a rising edge of  $WE\_b$  occurs when  $CS\_b$  is '0', or a rising edge of  $CS\_b$  occurs when  $WE\_b$  is '0', this indicates the end of write, so the data is stored in the RAM, and then the data is read back out after  $t_{OW}$ . If a falling edge of  $WE\_b$  occurs when  $CS\_b = '0'$ , the RAM switches to write mode and the data output goes to high-Z.

If a rising edge of  $CS\_b$  has occurred, the RAM is deselected, and the data output goes to high-Z after the specified delay. Otherwise, if a falling edge of  $CS\_b$  has occurred and  $WE\_b$  is 1, the RAM is in the read mode. The data bus can leave the high-Z state after time  $t_{CLZ}$  (min), but it is not guaranteed to have valid data out until time  $t_{ACS}$  (max). The region in between is a transitional region where the bus state is unknown, so we model this region by outputting 'X' on the I/O lines. If an address change has just occurred and the RAM is in the read mode (Figure 9-4(a)), the old data holds its value for time  $t_{OH}$ . Then the output is in an unknown transitional state until valid data has been read from the RAM after time  $t_{AA}$ .

The check process, which runs concurrently with the RAM process, tests to see if some of the memory timing specifications are satisfied. *NOW* is a predefined variable that equals the current time. To avoid false error messages, checking is not done when  $NOW = 0$  or when the chip is not selected. When the address changes, the process checks to see if the address has been stable for the write cycle time ( $t_{WC}$ ) and outputs a warning message if it is not. Since an address event has just occurred when this test is made, *Address'stable*( $t_{WC}$ ) would always return FALSE. Therefore, *Address'delayed* must be used instead of *Address* so that *Address* is delayed one delta and the stability test is made just before *Address* changes. Next the timing specifications for write are checked. First, we verify that the address has been stable for  $t_{AW}$ . Then we check to see that  $WE\_b$  has been low for  $t_{WP}$ . Finally, we check the setup and hold times for the data.

**Figure 9-11 VHDL Timing Model for 6116 Static CMOS RAM**

```
-- memory model with timing (OE_b=0)

library ieee;
use ieee.std_logic_1164.all;
library bitlib;
use bitlib.bit_pack.all;

entity static_RAM is
generic (constant tAA: time := 120 ns; -- 6116 static CMOS RAM
         constant tACS:time := 120 ns;
         constant tCLZ:time := 10 ns;
         constant tCHZ:time := 10 ns;
         constant tOH:time := 10 ns;
```

```
constant tWC:time := 120 ns;
constant tAW:time := 105 ns;
constant tWP:time := 70 ns;
constant tWHZ:time := 35 ns;
constant tDW:time := 35 ns;
constant tDH:time := 0 ns;
constant tOW:time := 10 ns);

port (CS_b, WE_b, OE_b: in bit;
      Address: in bit_vector(7 downto 0);
      Data: inout std_logic_vector(7 downto 0) := (others => 'Z'));
end Static_RAM;

architecture SRAM of Static_RAM is
type RAMtype is array(0 to 255) of bit_vector(7 downto 0);
signal RAM1: RAMtype := (others => (others => '0'));

begin
  RAM: process
  begin
    if (rising_edge(WE_b) and CS_b'delayed = '0')
      or (rising_edge(CS_b) and WE_b'delayed = '0') then
      RAM1(vec2int(Address'delayed)) <= to_bitvector(Data'delayed); --write
      Data <= transport Data'delayed after tOW; -- read back after write
      -- Data'delayed is the value of Data just before the rising edge
    end if;
    if falling_edge(WE_b) and CS_b = '0' then -- enter write mode
      Data <= transport "ZZZZZZZZ" after tWHZ;
    end if;
    if CS_b'event and OE_b = '0' then
      if CS_b = '1' then -- RAM is deselected
        Data <= transport "ZZZZZZZZ" after tCHZ;
      elsif WE_b = '1' then --read
        Data <= "XXXXXXXX" after tCLZ;
        Data <= transport to_stdlogicvector(RAM1(vec2int(Address)))
          after tACS;
      end if;
    end if;
    if Address'event and CS_b ='0' and OE_b ='0' and WE_b ='1' then --read
      Data <= "XXXXXXXX" after tOH;
      Data <= transport to_stdlogicvector(RAM1(vec2int(Address)))
        after tAA;
    end if;
    wait on CS_b, WE_b, Address;
  end process RAM;
```

```

check: process
begin
  if CS_b'delayed = '0' and NOW /= 0 ns then
    if address'event then
      assert (address'delayed'stable(tWC)) -- tRC = tWC assumed
        report "Address cycle time too short"
        severity WARNING;
    end if;
    if rising_edge(WE_b) then
      assert (address'delayed'stable(tAW))
        report "Address not valid long enough to end of write"
        severity WARNING;
      assert (WE_b'delayed'stable(tWP))
        report "Write pulse too short"
        severity WARNING;
      assert (Data'delayed'stable(tDW))
        report "Data setup time too short"
        severity WARNING;
      wait for tDH;
      assert (Data'last_event >= tDH)
        report "Data hold time too short"
        severity WARNING;
    end if;
  end if;
  wait on WE_b, address, CS_b;
end process check;
end SRAM;

```

VHDL code for a partial test of the RAM timing model is shown in Figure 9-12. This code runs a write cycle followed by two read cycles. The RAM is deselected between cycles. Figure 9-13 shows the test results. We also tested the model for cases where simultaneous input changes occur and cases where timing specifications are violated, but these test results are not included here.

**Figure 9-12 VHDL Code for Testing the RAM Timing Model**

```

library IEEE;
use IEEE.std_logic_1164.all;
library BITLIB;
use BITLIB.bit_pack.all;

entity RAM_timing_tester is
end RAM_timing_tester;

architecture test1 of RAM_timing_tester is

component static_RAM is
port (CS_b, WE_b, OE_b: in std_logic;
      Address: in bit_vector(7 downto 0);
      Data: inout std_logic_vector(7 downto 0));
end component Static_RAM;

```

```

signal Cs_b, We_b: std_logic := '1';           -- active low signals
signal Data: std_logic_vector(7 downto 0) := "ZZZZZZZZ";
signal Address: bit_vector(7 downto 0);

begin
  SRAM1: Static_RAM port map(Cs_b, We_b, '0', Address, Data);
  process
  begin
    wait for 100 ns;

    Address <= "00001000";                      -- write(2) with CS pulse
    Cs_b <= '0';
    We_b <= transport '0' after 20 ns;
    Data <= transport "11100011" after 140 ns;
    Cs_b <= transport '1' after 200 ns;
    We_b <= transport '1' after 180 ns;
    Data <= transport "ZZZZZZZZ" after 220 ns;
    wait for 200 ns;

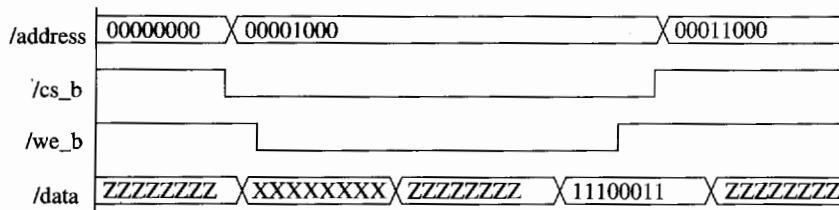
    Address <= "00011000";                      -- RAM deselected
    wait for 200 ns;

    Address <= "00001000";                      -- Read cycles
    Cs_b <= '0';
    wait for 200 ns;
    Address <= "00010000";
    Cs_b <= '1' after 200 ns;
    wait for 200 ns;

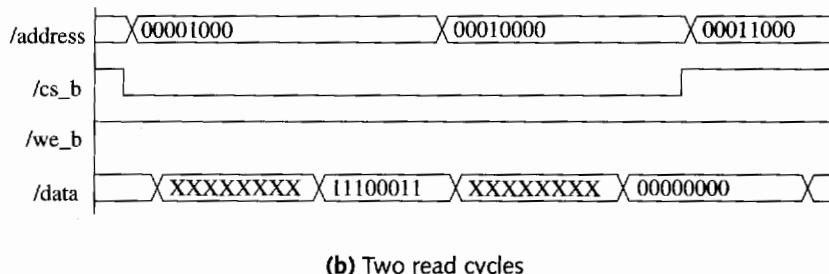
    Address <= "00011000";                      -- RAM deselected
    wait for 200 ns;
  end process;
end test1;

```

**Figure 9-13 Test Results for RAM Timing Model**



(a) Write cycle

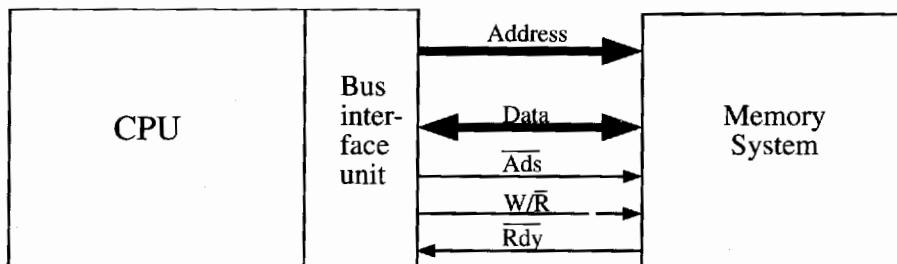


## 9.2 A SIMPLIFIED 486 BUS MODEL

Memories and input-output devices are usually interfaced to a microprocessor by means of a tristate bus. To assure proper transfer of data on this bus, the timing characteristics of both the microprocessor bus interface and the memory must be carefully considered. We have already developed a VHDL timing model for a RAM memory, and next we will develop a timing model for a microprocessor bus interface. Then we will simulate the operation of a system containing a microprocessor and memory to determine whether the timing specifications are satisfied.

Figure 9-14 shows a typical bus interface to a memory system. The normal sequence of events for writing to memory is: (1) The microprocessor outputs an address on the address bus and asserts  $\overline{Ads}$  (address strobe) to indicate a valid address on the bus; (2) the processor places data on the data bus and asserts  $\overline{W/R}$  (write/read) to initiate writing the data to memory. The memory system asserts  $\overline{Rdy}$  (ready) to indicate that the data transfer is complete. For reading from memory, step (1) is the same, but in step (2) the memory places data on the data bus and these data are stored inside the processor when the memory system asserts  $\overline{Rdy}$ .

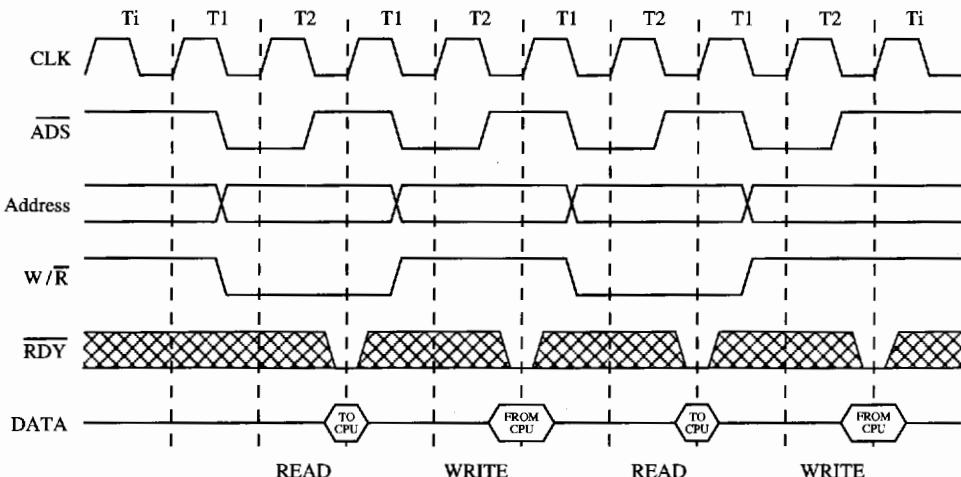
**Figure 9-14** Microprocessor Bus Interface



As an example, we develop a simplified model for a 486 microprocessor bus interface. The actual 486 bus interface is very complex and supports many different types of bus cycles. Figures 9-15 and 9-16 illustrate two of these bus cycles. In Figure 9-15, one word of data is transferred between the CPU and the bus every two clock cycles. These clock cycles are labeled T1 and T2, and they correspond to states of the internal bus controller. In addition, the bus has an idle state, Ti. During Ti and between data transfers on the bus,

the data bus is in a high-impedance state (indicated on the diagram by *DATA* being halfway between 0 and 1). The bus remains in the idle state until the bus interface receives a bus request from the CPU. In T1, the interface outputs a new address on the bus and asserts *Ads* low. For a read cycle, the read-write signal (*W/R*) is also asserted low during T1 and T2. During T2 of the read cycle, the memory responds to the new address and places data on the data bus (labeled “to CPU” on the diagram). The memory system also asserts *Rdy* low to indicate that valid data is on the bus. At the rising edge of the clock that ends T2, the bus interface senses that *Rdy* is low and the data is stored inside the CPU.

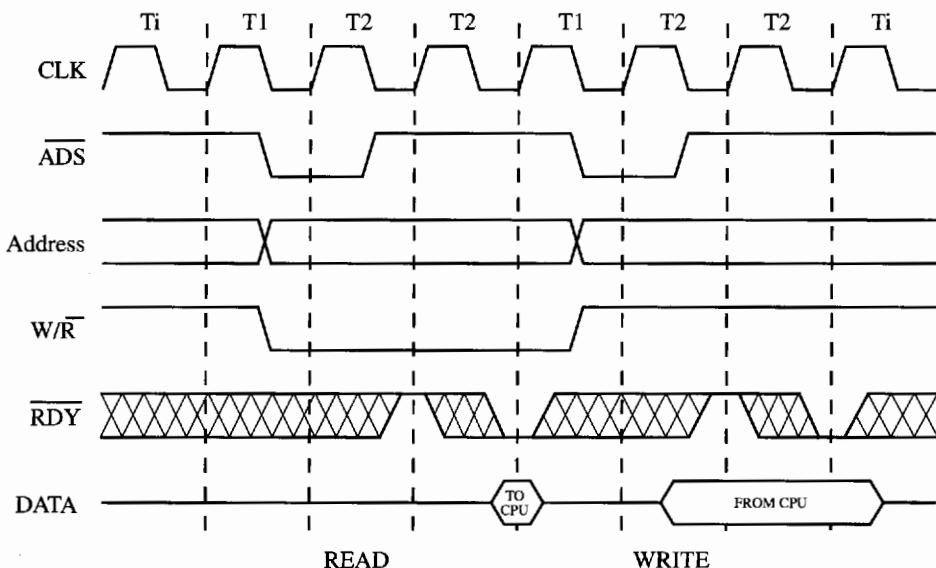
**Figure 9-15 Intel 486 Basic 2-2 Bus Cycle**



The next bus cycle in Figure 9-15 is a write cycle. As before, the new address is output during T1 and *Ads* goes low, but *W/R* remains high. During T2, the CPU places data on the bus. Near the end of T2, the memory system asserts *Rdy* low to indicate completion of the write cycle, and the data is stored in the memory at the end of T2 (rising edge of the clock). This is followed by another read and another write cycle.

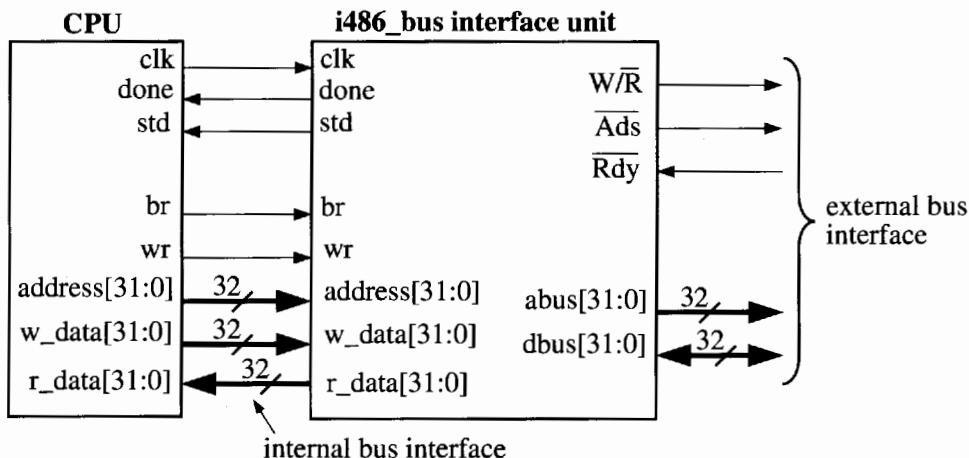
Figure 9-16 shows 486 read and write bus cycles for the case where the memory is slow and reading one word from memory or writing one word to memory requires three clock cycles. The read operation is similar to that in Figure 9-15, except at the end of the first T2 cycle, the bus interface senses that *Rdy* is high, and another T2 cycle is inserted. At the end of the second T2 cycle, *Rdy* is low and the read is complete. The write operation is also similar to that in Figure 9-15, except at the end of the first T2 cycle, *Rdy* is high and a second T2 cycle is inserted. At the end of the second T2, *Rdy* is low and writing to memory is complete. The added T2 states are often referred to as wait states, since the processor is waiting on the memory.

Figure 9-16 Intel 486 Basic 3-3 Bus Cycle



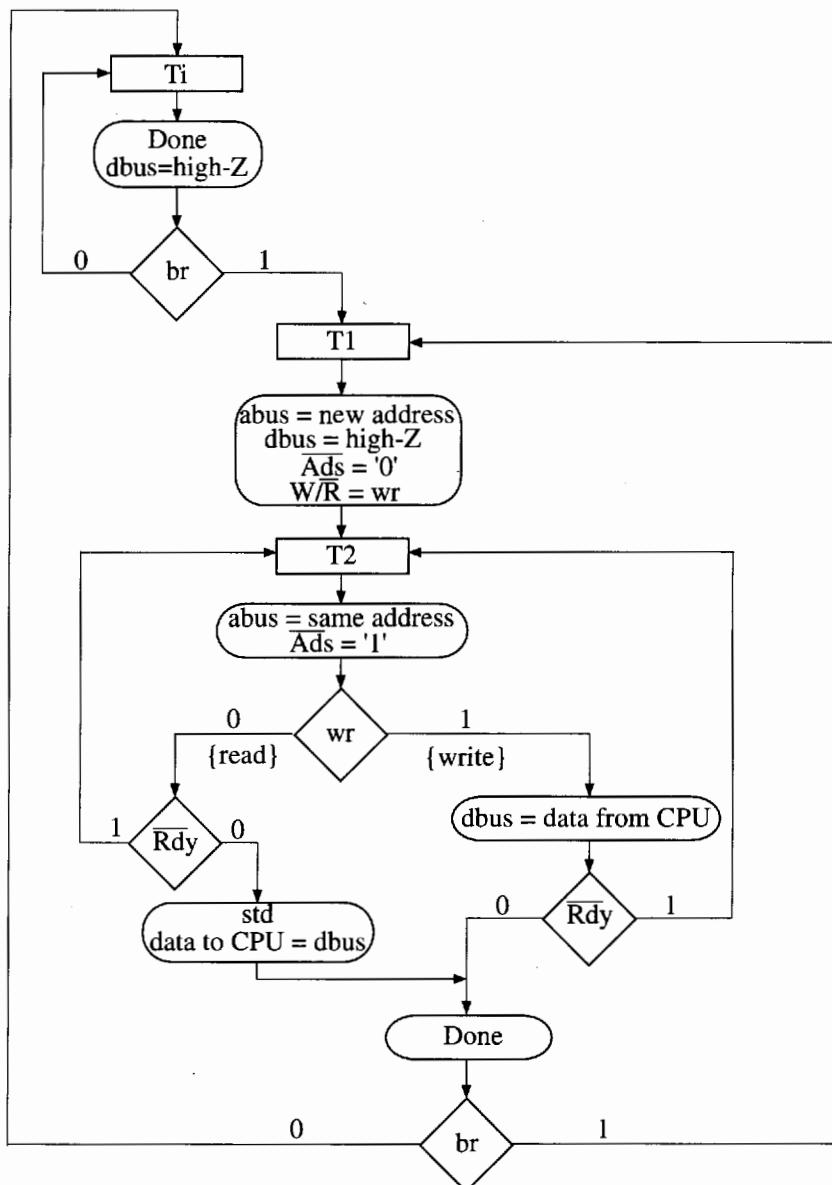
We now develop a model for a simplified 486 bus interface unit. This unit is the part of the 486 microprocessor that provides an interface between the external 486 bus and the rest of the processor. Our bus model is based on the 486 data sheet, and it is intended to represent the external behavior of the 486 bus interface. The block diagram in Figure 9-17 shows the interface signals we include in our model. The actual 486 bus interface is much more complex. It has external interface signals for transferring data 8, 16, or 32 bits at a time, for running burst memory cycles, which allow transferring blocks of data at a faster rate, for allowing other devices to take control of the external bus, etc. In our simplified model, we include only those signals needed to run the basic read and write bus cycles, as illustrated in Figures 9-15 and 9-16. We do not attempt to develop an accurate model for the 486 CPU. The internal bus interface in Figure 9-17 shows only those signals needed for transferring data between the bus interface unit and the CPU. If the CPU needs to write data to a memory attached to the external bus interface, it requests a write cycle by setting *br* (bus request) to 1 and *wr* (write) to 1. If the CPU needs to read data, it requests a read cycle by setting *br* = 1 and *wr* = 0. When the write or read cycle is complete, the bus interface unit returns *done* = 1 to the CPU.

Figure 9-17 Simplified 486 Bus Interface Unit

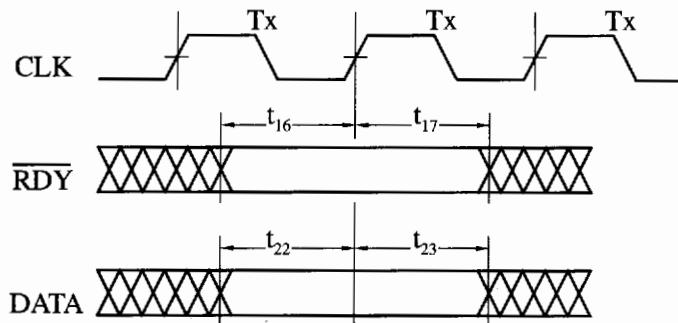


The 486 bus interface unit contains a state machine to control the bus operation. Figure 9-18 is a modified SM chart that represents a simplified version of this state machine. In state  $T_i$ , the bus interface is idle, and the data bus is driven to high-Z. When a bus request ( $br$ ) is received from the CPU, the controller goes to state  $T_1$ . In  $T_1$ , the new address is driven onto the address bus, and  $\overline{Ads}$  is set to 0 to indicate a valid address on the bus. The write-read signal ( $W/R$ ) is set low for a read cycle or high for a write cycle, and the controller goes to state  $T_2$ . In  $T_2$ ,  $\overline{Ads}$  returns to 1. For a read cycle,  $wr = 0$  and the controller waits for  $\overline{Rdy} = 0$ , which indicates valid data is available from the memory, and then  $std$  (store data) is asserted to indicate that the data should be stored in the CPU. For a write cycle,  $wr = 1$  and data from the CPU is placed on the data bus. The controller then waits for  $\overline{Rdy} = 0$  to indicate that the data has been stored in memory. For both read and write, the  $done$  signal is turned on when  $\overline{Rdy} = 0$  to indicate completion of the bus cycle. After read or write is complete, the controller goes to  $T_i$  if no bus request is pending, otherwise it goes to state  $T_1$  to initiate another read or write cycle. The  $done$  signal remains on in  $T_i$ .

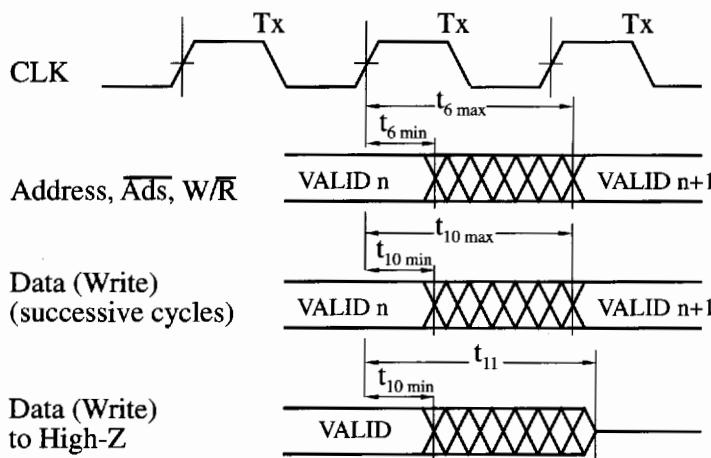
Figure 9-18 SM Chart for Simplified 486 Bus Interface



The 486 processor bus is synchronous, so all timing specifications are measured with respect to the rising edge of the clock. When the bus interface senses  $\overline{Rdy}$ ,  $Rdy$  must be stable for the setup time before the rising edge of the clock and for the hold time after the rising edge. These setup and hold times are designated as  $t_{16}$  and  $t_{17}$  in Figure 9-19. During a read cycle, the data to the CPU must also satisfy setup and hold time requirements. The data must be stable  $t_{22}$  before and  $t_{23}$  after the active clock edge.

**Figure 9-19 486 Setup and Hold Time Specifications**

When the output signals from the bus interface change, a delay occurs between the active clock edge and the time at which the signals are stable. When the address changes from one valid address to the next, after the clock edge, there is a minimum time ( $t_{6 \text{ min}}$ ) and a maximum time ( $t_{6 \text{ max}}$ ) during which the address can change, as shown in Figure 9-20. The crosshatched region indicates the region in which the address can change. Similarly, when the data is changing from one value to the next, the change can occur any time between  $t_{10 \text{ min}}$  and  $t_{10 \text{ max}}$ . When the data bus is changing from valid data to high-Z, the data remains valid for at least  $t_{10 \text{ min}}$ , and the change to high-Z occurs some time between  $t_{10 \text{ min}}$  and  $t_{11}$  after the clock edge. All the clock cycles in the figure are labeled Tx, but the address and data transitions can only occur between certain clock cycles. Address changes are always initiated by the rising clock edge at the transition between Ti and T1 or between T2 and T1, whereas write data transitions are initiated at the end of a T1 cycle. The transition from valid data out to high-Z is initiated at the end of the write cycle at the transition from T2 to T1 or Ti.

**Figure 9-20 486 Bus Timing Specifications for Address and Data Changes**

The VHDL model for the 486 bus interface, based on the SM chart of Figure 9-18 and the block diagram of Figure 9-17, is given in Figure 9-21. The 486 comes in several different speeds, and the default values of the generic timing specifications are for the 50-MHz version of the 486. The bus state machine is modeled by two processes, and a third process is used to check some of the critical timing parameters. The bus state is initialized to Ti and *dbus* (data bus) is driven to high-Z after time  $t_{10\text{ min}}$ . When the bus interface state machine enters state T1, *ads\_b* ( $\overline{AdS}$ ) is driven low, *w\_rb* ( $W/R$ ) is driven to the appropriate value, and the address is driven onto the address bus (*abus*). In each case, the appropriate delay time is included. In state T2, for a write cycle (*wr* = 1) the write data from the CPU (*w\_data*) is driven onto *dbus*. For a read cycle (*wr* = 0) the data from *dbus* is sent to the CPU as *r\_data* (read data), and *std* is asserted to indicate that the data should be stored in the CPU on the next rising edge of the clock.

**Figure 9-21** VHDL Model for 486 Bus Interface Unit

```

LIBRARY ieee;
use ieee.std_logic_1164.all;

entity i486_bus is
    generic (-- These specs are for the i486DX 50
        constant t6_max:time:=12 ns;
        constant t10_min:time:=3 ns;
        constant t10_max:time:=12 ns;
        constant t11_max:time:=18 ns;
        constant t16_min:time:=5 ns;
        constant t17_min:time:=3 ns;
        constant t22_min:time:=5 ns;
        constant t23_min:time:=3 ns);
    port (--external interface
        abus: out bit_vector(31 downto 0);
        dbus: inout std_logic_vector(31 downto 0) := (others => 'Z');
        w_rb, ads_b: out bit := '1';
        rdy_b, clk: in bit;
    --internal interface
        address, w_data: in bit_vector(31 downto 0);
        r_data: out bit_vector(31 downto 0);
        wr, br: in bit;
        std, done:out bit);
end i486_bus;
--*****
architecture simple_486_bus of i486_bus is
type state_t is (Ti, T1, T2);
signal state, next_state:state_t:=Ti;
--*****
begin
-- The following process outputs the control signals and address of
-- the processor during a read/write operation. The process also drives
-- or tristates the data bus depending on the operation type.
-- During the execution of a read/write operation, the done signal
-- is low. When the bus is ready to accept a new request, done is high.

```



```
--The following process checks that all setup and hold times are
-- met for all incoming control signals.
-- Setup and hold times are checked for the data bus during a read only
wave_check: process (clk, dbus, rdy_b)
  variable clk_last_rise:time:= 0 ns;
begin
  if (now /= 0 ns) then
    if clk'event and clk = '1' then                                --check setup times
      --The following assert assumes that the setup for RDY
      -- is equal to or greater than that for data
      assert (rdy_b /= '0') OR (wr /= '0') OR
        (now - dbus'last_event >= t22_min)
      report "i486 bus:Data setup too short"
      severity WARNING;
      assert (rdy_b'last_event >= t16_min)
      report "i486 bus:RDY setup too short"
      severity WARNING;
      clk_last_rise := NOW;
    end if;
    if (dbus'event) then                                         --check hold times
      --The following assert assumes that the hold for RDY
      -- is equal to or greater than that for data
      assert (rdy_b /= '0') OR (wr /= '0') OR
        (now - clk_last_rise >= t23_min)
      report "i486 bus:Data hold too short"
      severity WARNING;
    end if;
    if (rdy_b'event) then
      assert (now - clk_last_rise >= t17_min)
      report "i486 bus: RDY signal hold too short"
      severity WARNING;
    end if;
  end if;
end process wave_check;
end simple_486_bus;
```

The check process checks to see that the setup times for reading data and for *rdy\_b* are satisfied whenever the rising edge of the clock occurs. To avoid false error messages, checking is done only when the chip is selected and *now*  $\neq$  0. The first assert statement checks the data setup time if *rdy\_b* = '0' and *wr* = '0' and reports an error if

$$(now - dbus'event) < \text{minimum setup time}$$

where *now* is the time at which the clock changed and *dbus'event* is the time at which the data changed. The process also checks to see if the read data hold time is satisfied whenever the data changes and reports an error if

$$(now - clock_last_rise) < \text{minimum hold time}$$

where *now* is the time when the data changed, and *clock\_last\_rise* is the time at which the last rising edge of *clk* occurred. The process also checks to see if the *rdy\_b* hold time is satisfied whenever *rdy\_b* changes.

## 9.3 INTERFACING MEMORY TO A MICROPROCESSOR BUS

In this section we discuss the timing aspects of interfacing memory to a microprocessor bus. In order to design the interface, the timing specifications for both the memory and microprocessor must be satisfied. When writing to memory, the setup and hold time specifications for the memory must be satisfied, and when reading from memory, the setup and hold time specifications for the microprocessor bus must be satisfied. If the memory is slow, it may be necessary to insert wait states in the bus cycle.

We design the interface between a 486 bus and a small static RAM memory, and then we write VHDL code to test the interface timing. We use the static RAM and 486 bus interface models that we have already developed. Figure 9-22 shows how the bus interface is connected to the static RAM memory. The memory consists of four static RAM chips, and each RAM chip contains  $2^{15} = 32,768$  8-bit words. The four chips operate in parallel to give a memory that is 32 bits wide. Data bits 31–24 are connected to the first chip, bits 23–16 to the second chip, etc. The lower 15 bits of the address bus are connected in parallel to all four chips. The system includes a memory controller that generates  $\overline{WE}$  and  $\overline{CS}$  signals for the memory and returns a  $\overline{Rdy}$  signal to the bus interface to indicate completion of a read or write bus cycle. We will assign the address range 0 through 32,767 to the static RAM. In general, a complete 486 system would have a large dynamic RAM memory and I/O interface cards connected to the address and data busses. To allow for expansion of the system, we use an address decoder, so the static RAM is selected only for the specified address range. When an address in the range 0 to 32,767 is detected, the decoder outputs  $CS1 = 1$ , which activates the memory controller.

**Figure 9-22 486 Bus Interface to Static RAM System**

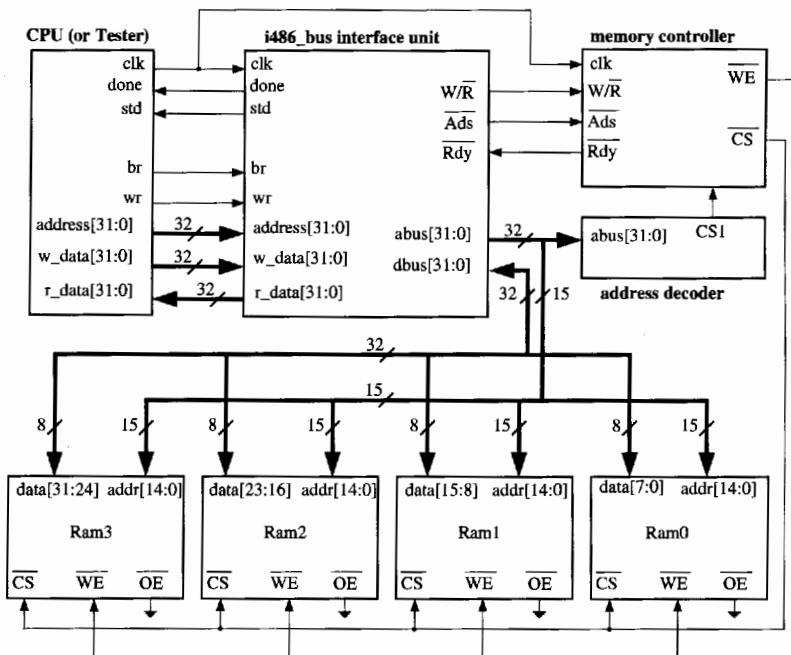
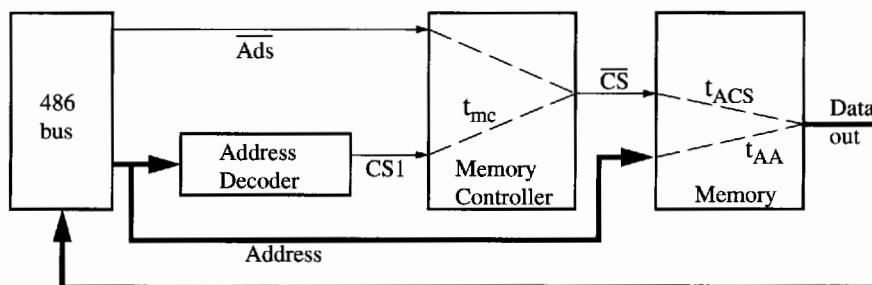


Table 9-2 gives the timing characteristics of the 43258A-25, which is a fast static CMOS RAM, with a 25-ns access time. We use this RAM in our design so we can illustrate the insertion of wait states in the bus cycle. (If we use a faster RAM, we can eliminate the need for wait states and run 2-2 bus cycles.) In general, a detailed timing analysis is necessary in order to determine how many wait states (if any) must be inserted so that both the 486 bus and the RAM timing specifications are met. Figure 9-23 shows the signal paths for reading data from the RAM. We want to do a worst-case analysis, so we must find the path with the longest delay. Since  $t_{AA} = t_{ACS}$ , the total propagation delay along the  $\overline{CS}$  path will be longer than for the address path. Also, since  $Ad_s$  and the address become stable at the same time, the longest path includes the address decoder. Using the start of T1 as a reference point, the delays along the longest path are

- Time from clock high to address valid =  $t_{6max} = 12$  ns
- Propagation delay through address decoder =  $t_{decode} = 5$  ns
- Propagation delay through memory controller =  $t_{mc} = 5$  ns
- Memory access time =  $t_{ACS} = 25$  ns
- Data setup time for 486 bus =  $t_{22} = 5$  ns
- Total delay = 52 ns

Propagation delays of 5 ns were assumed for both the address decoder and memory controller. These values may have to be changed when the design of these components is completed. If the 486 is operated at its maximum clock rate of 50 MHz, the clock period is 20 ns, and three clock periods are required to complete a read cycle.

**Figure 9-23 Signal Paths for Memory Read**

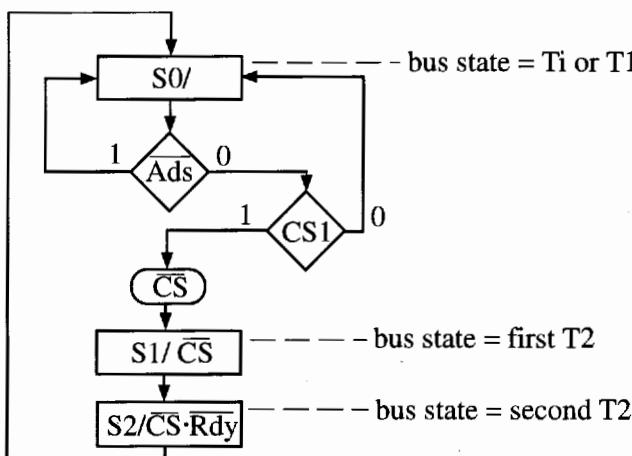


The SM chart in Figure 9-24 shows one possible design of the memory controller for read cycles. The controller waits in state S0 until  $Ad_s$  is asserted to indicate a valid address in bus state T1 and until  $CS1$  is asserted by the address decoder to indicate that the address is in the range for the static RAM. Then  $\overline{CS}$  is asserted to select the RAM, and the controller goes to S1. In S2, the controller also asserts  $\overline{Rdy}$  to indicate that this is the last T2. The timing requirements will still be satisfied if  $\overline{CS}$  is not asserted until S1. In this case, the propagation delays measured with respect to the end of T1 are:

- Propagation delay through memory controller =  $t_{mc} = 5$  ns
- Memory access time =  $t_{ACS} = 25$  ns
- Data setup time for 486 bus =  $t_{22} = 5$  ns
- Total delay = 35 ns

Since the clock period is 20 ns, valid data is available at  $2 \times 20\text{ ns} - 5\text{ ns} = 35\text{ ns}$ , which is 5 ns before the end of the second T2 state.

**Figure 9-24** Memory Controller SM Chart for Read Cycles



Next, we check the data setup time for writing to RAM. Using the start of T2 as a reference point, the worst-case delays along the data path are

- Time from clock high to data valid (486) =  $t_{10\max} = 12 \text{ ns}$
  - Data valid to end of write (RAM) =  $t_{DW} = 12 \text{ ns}$
  - Total time = 24 ns

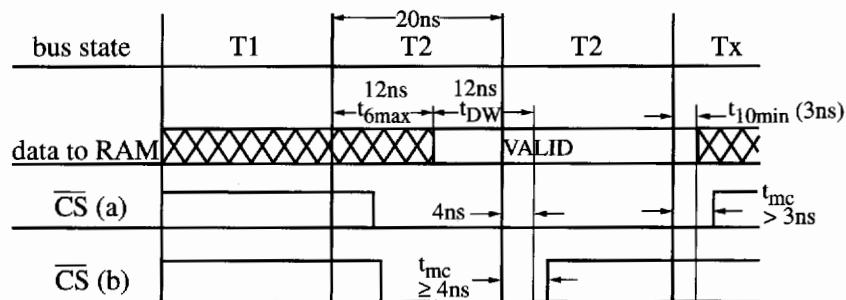
Since the total time from the start of T2 to completion of write must be at least 24 ns, one T2 cycle (20 ns) is not sufficient, and a second T2 cycle is required. This means that three clocks (T1 and two T2s) are required to complete the write cycle.

The data hold time for the 486 read cycle is  $t_{23} = 3$  ns minimum. After  $\overline{CS}$  goes high, the data out of the RAM is valid for at least  $t_{CHZ}$  minimum, which is 0 ns. Therefore,  $t_{23}$  will be satisfied if  $t_{mc}$  is at least 3 ns, so  $\overline{CS}$  goes high and the RAM is deselected at least 3 ns after the rising clock edge.

The write to RAM can be controlled by either  $\overline{CS}$  or  $\overline{WE}$ . Both  $\overline{CS}$  and  $\overline{WE}$  must go low, and the first one to go high completes the write operation. We will set  $\overline{WE} = (W/R)$  and use  $\overline{CS}$  to control the write cycle. Since  $t_{CW} = 15$  ns,  $\overline{CS}$  must be low for at least 15 ns. This requirement is satisfied if  $CS$  goes low for one clock period.

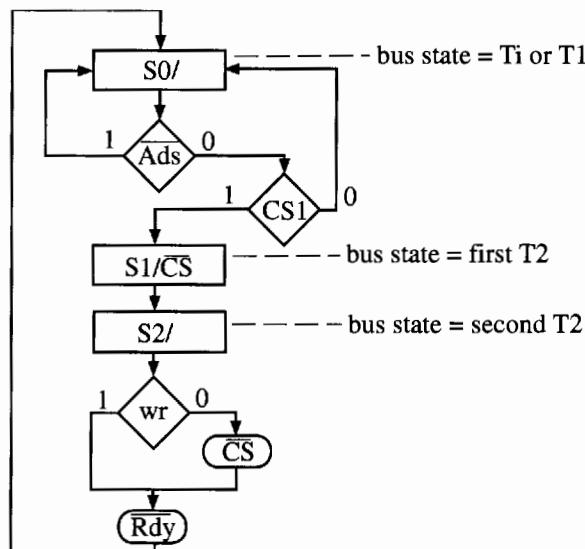
Since the memory data hold time is  $t_{DH} = 0$  ns, we must make sure that the data remains valid until  $\overline{CS}$  goes high. The 486 bus spec indicates that the write data may go invalid as soon as  $t_{10\min} = 3$  ns after the end of the last T2. This presents a problem if  $\overline{CS}$  goes high at the end of the second T2, and the propagation delay in the memory controller ( $t_{mc}$ ) is greater than 3 ns. In this case, as illustrated by  $\overline{CS}$  waveform (a) in Figure 9-25,  $\overline{CS}$  goes high after the data to the RAM becomes invalid. One solution is to use faster logic in the memory controller. Another solution is to have  $\overline{CS}$  go high at the end of the first T2, as shown by  $\overline{CS}$  waveform (b). As long as  $t_{mc} \geq 4$  ns, the data setup time  $t_{DW}$  will be satisfied.

Figure 9-25 Chip Select Timing for Write to RAM



The SM chart for a memory controller that meets the timing requirements for both read and write bus cycles is shown in Figure 9-26. The write timing is based on Figure 9-25,  $\overline{\text{CS}}$  (b). The VHDL code for the memory controller (Figure 9-27) is based on the SM chart and follows the standard pattern for a state machine, except that delays have been added to the output signals,  $\overline{\text{CS}}$ ,  $\overline{WE}$ , and  $\overline{Rdy}$ .

Figure 9-26 SM Chart for Memory Controller



**Figure 9-27 VHDL Code for Memory Controller**

```
-- Memory Controller for fast CMOS SRAM w/ one wait state

entity memory_control is
  port(      clk, w_rb, ads_b, cs1: in bit;
             rdy_b, we_b, cs_b: out bit := '1');
end memory_control;

architecture behave1 of memory_control is
  constant delay: time := 5 ns;
  signal state, nextstate: integer range 0 to 2;
  signal new_we_b, new_cs_b, new_rdy_b: bit := '1';
begin
  process(state,ads_b,w_rb,cs1)
  begin
    new_cs_b <= '1'; new_rdy_b <= '1'; new_we_b <= '1';
    case state is
      when 0 => if ads_b = '0' and cs1 = '1' then nextstate <= 1;
                   else nextstate <= 0;
                   end if;
      when 1 => new_cs_b <= '0';
                   nextstate <= 2;
      when 2 => if w_rb = '1' then new_cs_b <= '1';
                   else new_cs_b <= '0';
                   end if;
                   new_rdy_b <= '0';
                   nextstate <= 0;
    end case;
  end process;

  process(clk)
  begin
    if clk = '1' then state <= nextstate; end if;
  end process;

  we_b <= not w_rb after delay;
  cs_b <= new_cs_b after delay;
  rdy_b <= new_rdy_b after delay;
end behave1;
```

In order to test the 486 bus interface to the static RAM system, we write a tester module to take the place of the CPU in Figure 9-22. The tester needs to verify that the memory controller works correctly in conjunction with the 486 bus interface and the RAM memory. Since both the bus model and memory model have built-in timing checks, the tester needs to verify only that the system is fully functional, that data written to RAM is read back correctly, and that no bus conflicts occur. Each time a bus cycle is completed, the tester must supply values of *br*, *wr*, *address*, and *data* for the next bus cycle. To make it easy to change the test data, we place the test data in a text file and have the tester read the test data from the file. Each line of the test file contains values for the following:

*br* (bit)   *wr* (bit)   *address* (integer)   *data* (integer)

with each field separated by one or more spaces. We have chosen integer format for the address and data, since integer fields are compact and easy to read using the standard *TEXTIO* routines (see Section 8.10).

The tester (Figure 9-28) interacts with the bus interface state machine (Figure 9-18). Since the bus interface senses *Rdy* just before the end of the T2 bus state, the tester should check *Done* just before the rising clock edge that ends T2. To facilitate this, we have defined a test clock (*testclk*) within the tester. The bus clock (*Clk*) is the same as *testclk*, except that it is delayed by a short time. In this way, we can check the value of *Done* just after the rising edge of *testclk*, which occurs just before the rising edge of *Clk*. The process *read\_test\_file* waits for the rising edge of *testclk* and then tests *Done*. If *Done* is '1' and a read cycle has just been completed, data has been read from memory and *std* = '1'. In this case, the tester verifies that the data read from memory (*r\_data*) is the same as the data previously read from the test file (*dataint*). Then the tester reads a line from the test file and reads the values of *br*, *wr*, *data*, and *address* from the read buffer. These values define the next bus cycle. If the next cycle is a write, the data from the test file is output as *w\_data*. The value of *br* is checked by the bus state machine to determine if the next bus state should be *Ti* or *T1*.

**Figure 9-28** VHDL Code for 486 Bus System Test Module

```
-- Tester for Bus model
library BITLIB;
use BITLIB.bit_pack.all;
use std.textio.all;

entity tester is
    port ( address, w_data: out bit_vector(31 downto 0);
           r_data: in bit_vector(31 downto 0);
           clk, wr, br: out bit;
           std, done: in bit := '0');
end tester;

architecture test1 of tester is
    constant half_period: time := 10 ns;           -- 20 ns clock period
    signal testclk: bit := '1';
begin
    testclk <= not testclk after half_period;
    clk <= testclk after 1 ns;                      -- Delay bus clock
    read_test_file: process(testclk)
        file test_file: text open read_mode is "test2.dat";
        variable buff: line;
        variable dataint, addrint: integer;
        variable new_wr, new_br: bit;
    begin
        if testclk = '1' and done = '1' then
            if std = '1' then
                assert dataint = vec2int(r_data)
                    report "Read data doesn't match data file!"
                    severity error;
            end if;
        end if;
    end process;
    process(clk)
        variable wrdata: bit_vector(31 downto 0);
    begin
        if rising_edge(clk) then
            if wr = '1' then
                wrdata := w_data;
            else
                r_data <= wrdata;
            end if;
        end if;
    end process;
end architecture;
```

```

if not endfile(test_file) then
    readline(test_file, buff);
    read(buff, new_br);
    read(buff, new_wr);
    read(buff, addrint);
    read(buff, dataint);
    br <= new_br;
    wr <= new_wr;
    address <= int2vec(addrint,32);
    if new_wr = '1' and new_br = '1' then
        w_data <= int2vec(dataint,32);
    else w_data <= (others => '0');
    end if;
end if;
end if;
end process read_test_file;
end test1;

```

The VHDL for the complete 486 bus system with static RAM is shown in Figure 9-29. This model uses the tester, 486 bus interface unit, memory controller, and static RAM as components and instantiates these components within the architecture. A generate statement is used to instantiate four copies of the static RAM. In addition to the port map for the RAM, a generic map is used to specify the timing parameters for the 43258A-25 CMOS static RAM. Since our RAM model uses only 8 address lines, we have reduced the number of address lines from 15 to 8. The address decoder is implemented by a single concurrent statement.

**Figure 9-29 VHDL Code for Complete 486 Bus System with Static RAM**

```

library IEEE;
use IEEE.std_logic_1164.all;
entity i486_bus_sys is
end i486_bus_sys;

architecture bus_sys_bhv of i486_bus_sys is
--*****COMPONENTS*****
--*****COMPONENTS*****
component i486_bus
port (
    abus: out bit_vector(31 downto 0);
    dbus: inout std_logic_vector(31 downto 0);
    w_rb, ads_b: out bit;
    rdy_b, clk: in bit;
    address, w_data: in bit_vector(31 downto 0);
    r_data: out bit_vector(31 downto 0);
    wr, br: in bit;
    std, done:out bit);
end component;

```

```

component static_RAM
generic (constant tAA,tACS,tCLZ,tCHZ,tOH,tWC,tAW,tWP,tWHZ,tDW,tDH,tOW:
          time);
port (    CS_b, WE_b, OE_b: in bit;
          Address: in bit_vector(7 downto 0);
          Data: inout std_logic_vector(7 downto 0));
end component;
component memory_control
port(      clk, w_rb, ads_b, cs1: in bit;
          rdy_b, we_b, cs_b: out bit);
end component;
component tester
port (    address, w_data: out bit_vector(31 downto 0);
          r_data: in bit_vector(31 downto 0);
          clk, wr, br: out bit;
          std, done: in bit);
end component;
-- SIGNALS
--*****
constant decode_delay: time := 5 ns;
constant addr_decode: bit_vector(31 downto 8) := (others => '0');
signal cs1: bit;
--signals between tester and bus interface unit
signal address, w_data, r_data: bit_vector(31 downto 0);
signal clk, wr, br, std, done: bit;
--external 486 bus signals
signal w_rb, ads_b, rdy_b: bit;
signal abus: bit_vector(31 downto 0);
signal dbus: std_logic_vector(31 downto 0);
--signals to RAM
signal cs_b, we_b: bit;
--*****
begin
bus1: i486_bus port map (abus, dbus, w_rb, ads_b, rdy_b, clk, address,
                           w_data, r_data, wr, br, std, done);
control1: memory_control port map (clk, w_rb, ads_b, cs1, rdy_b, we_b,
                                    cs_b);
RAM32: for i in 3 downto 0 generate
  ram: static_RAM
  generic map(25 ns,25 ns,3 ns,3 ns,3 ns,25 ns,15 ns,15 ns
              10 ns,12 ns, 0 ns,0 ns)
  port map(cs_b, we_b, '0', abus(7 downto 0), dbus(8*i+7 downto 8*i));
end generate RAM32;
test: tester port map(address, w_data, r_data, clk, wr, br, std, done);
--***** Address decoder signal sent to memory controller
cs1 <= '1' after decode_delay when (abus(31 downto 8) = addr_decode)
  else '0' after decode_delay;
--*****
end bus_sys_bhv;

```

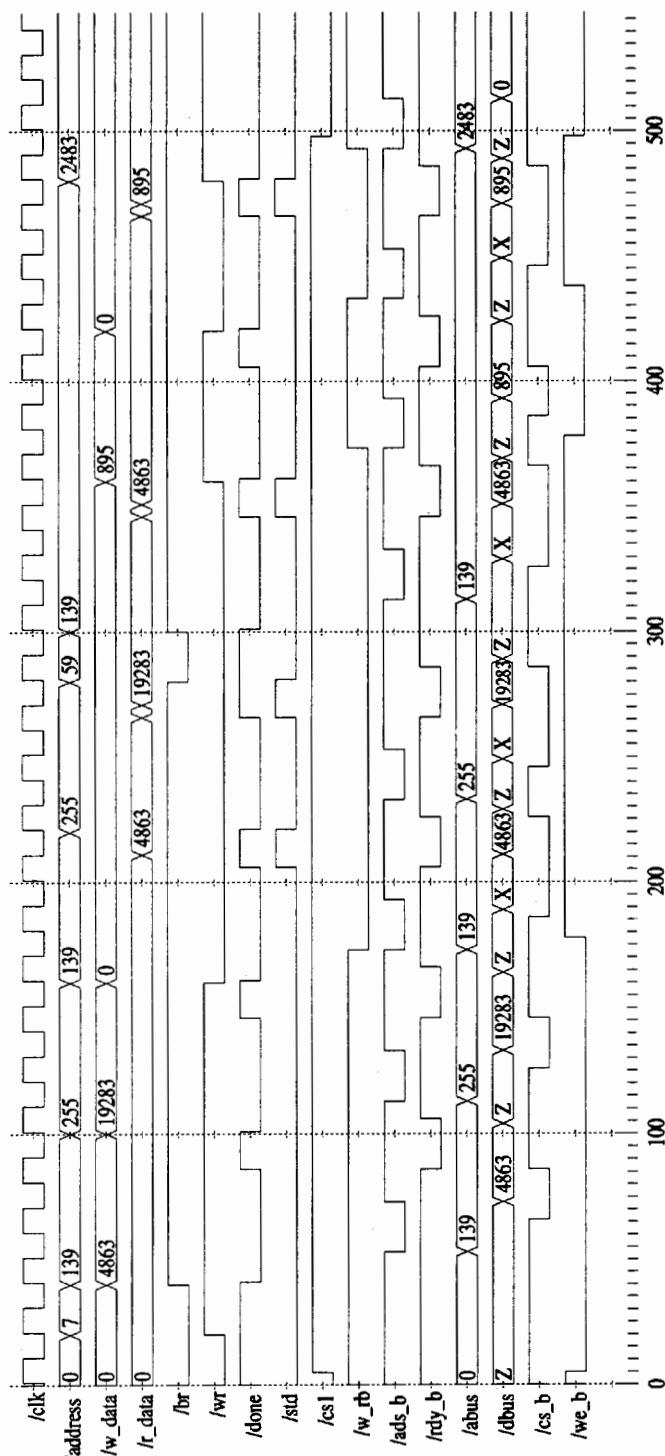
Table 9-3 shows a data file that was used to test various sequences of bus cycles, including idle followed by read or write, two successive writes, two successive reads, read followed by write, and write followed by read. The last line of test data contains an address that is outside of the RAM address range. When this bus cycle is executed, the memory controller should remain inactive, no  $\overline{Rdy}$  should be generated, and the 486 bus interface should continue to insert wait states until the simulation is terminated.

**Table 9-3 Test Data for 486 Bus System**

<i>br</i>	<i>wr</i>	<i>addr</i>	Data	Bus action
0	1	7	23	Idle
1	1	139	4863	Write
1	1	255	19283	Write
1	0	139	4863	Read
1	0	255	19283	Read
0	0	59	743	Idle
1	0	139	4863	Read
1	1	139	895	Write
1	0	139	895	Read
1	1	2483	0	Bus hang

The simulator output is shown in Figure 9-30. The performance of the memory, 486 bus, and memory controller are as expected. Since *r\_data* is represented by an integer signal, *r\_data* is 0 when *Dbus* is in the high-Z state. When interpreting the simulation results, we should keep in mind that the results are only as good as the models we have used. Although both the memory model and the 486 model are adequate for simulating these components, the models are not complete. Not all the timing specifications have been checked in the VHDL code. In many cases, only the maximum or minimum delay was selected to correspond to a worst-case condition. Under different conditions, the other limit on the delay may become significant. When simulating the memory controller, we used only a nominal delay. Before completing the memory controller design, we should go back and determine the maximum and minimum controller delays that are acceptable and make sure that the design conforms to this requirement.

In this chapter, we developed a simple VHDL model for a static RAM memory. Then we developed a more complex model, which included timing parameters and built-in checks to verify that setup and hold times and other timing specifications are met. Next we developed a timing model for a microprocessor bus interface, including checks to verify that timing specifications are met. Then we interfaced the bus to static RAMs and designed a memory controller to meet the timing requirements. We simulated the entire system to verify that the timing specs were satisfied for both the memory and the bus interface. In this example, we demonstrated the principles for designing an interface to meet worst-case timing specifications, and we demonstrated how to use VHDL to verify that the design is correct.

**Figure 9-30** Test Results for 486 Bus System

## Problems

**9.1** Answer the following questions for the 6116-2 and 43258A-25 static CMOS RAMs. Refer to the timing specifications in Table 9-2.

- (a) What is the maximum clock frequency that can be used?
- (b) What is the minimum time after a change in address or  $\overline{CS}$  at which valid data can be read?
- (c) For a  $\overline{WE}$ -controlled write cycle, what is the earliest and latest time new data can be driven after  $WE$  goes low?
- (d) For a  $\overline{CS}$ -controlled write cycle, what is the earliest and latest time new data can be driven after an address change?

**9.2** This problem concerns a simplified memory model for a 6116 CMOS RAM. Assume that both  $\overline{CS}$  and  $\overline{OE}$  are always low, so memory operation depends only on the address and  $\overline{WE}$ .

- (a) Write a simple VHDL model for the memory that ignores all timing information. (Your model should *not* contain  $\overline{CS}$  or  $\overline{OE}$ .)
- (b) Add the following timing specs to your model:  $t_{AA}$ ,  $t_{OH}$ ,  $t_{WHZ}$ , and  $t_{OW}$ . For read,  $Dout$  should go to "XXXXXXXX" (unknown) after  $t_{OH}$  and then to valid data out after  $t_{AA}$ . For write,  $Dout$  should go to high-Z after  $t_{WHZ}$ , and it should go to unknown after  $t_{OW}$ .
- (c) Add another process that gives appropriate error messages if any of the following specs are not satisfied:  $t_{WP}$ ,  $t_{DW}$ , and  $t_{DH}$ .

**9.3** VHDL code that describes the operation of the 6116 memory is given in Figure 9-11.

- (a) Add code that will report a warning if the data setup time for writing to memory is not met, if the data hold time for writing to memory is not met, or if the minimum pulse width spec for  $WE_b$  is not met.
- (b) Indicate the changes and additions to the original VHDL code that are necessary if  $OE\_b$  ( $\overline{OE}$ ) is taken into account. Note that for read, if  $OE\_b$  goes low after  $CS\_b$  goes low, the  $t_{OE}$  access time must be considered. Also note that when  $OE\_b$  goes high, the data bus will go high-Z after time  $t_{OHZ}$ .

**9.4** What modifications must be made in the check process in the VHDL 6116 RAM timing model (Figure 9-11) in order to verify the address setup time ( $t_{AS}$ ) and the write recovery time ( $t_{WR}$ ) specifications?

**9.5** Consider the  $CS$ -controlled write cycle for a static CMOS RAM (Figure 9-5.) What VHDL code is needed in the check process in the timing model (Figure 9-11) to verify the correct operation of a  $CS$ -controlled write? You must check timing specifications such as  $t_{CW}$ ,  $t_{DW}$ , and  $t_{DH}$ .

**9.6** Design a memory-test system to test the first 256 bytes of a 6116-2 static RAM memory. The system consists of simple controller, an 8-bit counter, a comparator, and a memory as shown below. The counter is connected to both the address and data (IO) bus so that 0 will be written to address 0, 1 to address 1, 2 to address 2, . . . , and 255 to address 255. Then the data will be read back from address 0, address 1, . . . , address 255 and compared with the address. If the data does not match, the controller goes to the fail state as soon as a mismatch is detected; otherwise, it goes to a pass state after all 256 locations have been matched. Assume that  $OE\_b = 0$  and  $CS\_b = 0$ .

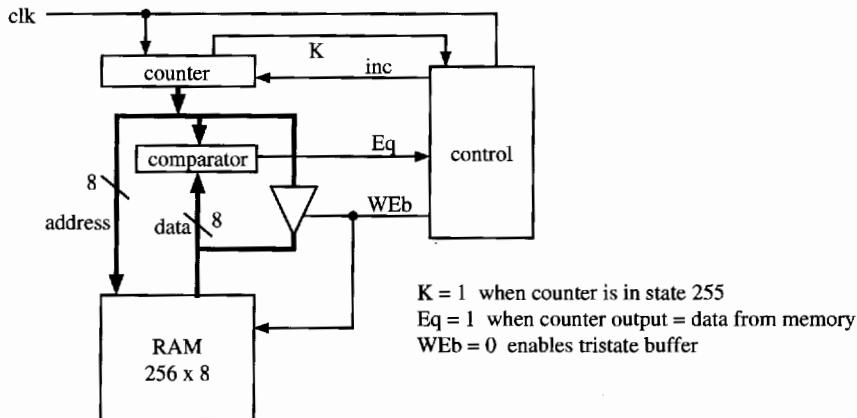
(a) Draw an SM chart or a state graph for the controller (5 states). Assume that the clock period is long enough so that one word can be read every clock period.

(b) For this system, determine the minimum clock period for satisfactory read operation. Assume the following delays:

clock rising edge to counter output stable – 15 ns

clock rising edge to control signal from state machine stable – 20 ns

comparator delay – 15 ns



**9.7** Design a memory tester that verifies the correct operation of a 6116 static CMOS RAM. The tester should store a checkerboard pattern (alternating 0s and 1s in the even addresses, and alternating 1s and 0s in the odd addresses) in all memory locations and then read it back. The tester should then repeat the test using the reverse pattern.

(a) Draw a block diagram of the memory tester. Show and explain all control signals.

(b) Draw an SM chart or state graph for the control unit. Assume you are using a simple RAM model and disregard timing.

(c) Write VHDL code for the tester and use a test bench to verify its operation.

### 9.8

(a) Consider the simplified VHDL model for the 486 bus (Figure 9-21). As indicated by the cross-hatching in Figure 9-20, following the rising edge of the clock, the address can change from the current value (VALID  $n$ ) to the next value (VALID  $n+1$ ) at any time between  $t_{6\min}$  and  $t_{6\max}$ . Thus, the value of the address can be considered as unknown ('X') during this time interval. A more accurate model of the bus should include this unknown region. Indicate on the VHDL code listing for the bus model the change(s) needed for the bus to be driven to 'X' in this region.

(b) A similar situation occurs with respect to the data bus. During a write cycle, there is a time interval between  $t_{10\min}$  and  $t_{10\max}$  where the data assumes an unknown ('X') value before the data assumes its final value. Also, when the T2 bus state is terminated, there is a time interval between  $t_{10\min}$  and  $t_{11\max}$  where the data assumes an unknown value ('X') before the bus enters the high-Z state. Indicate on the VHDL code listing for the bus model the necessary changes to take these 'X' regions into account.

(c) The clock period must be  $\geq t_{1\text{min}}$  and  $\leq t_{1\text{max}}$ . Add VHDL code to the *wave\_check* process, which will report "clock pulse width error" if the clock pulse is not within bounds.

**9.9** Determine an upper and lower bound for the delay used in the memory controller of Figure 9-26. Use a simulator to verify that a timing violation occurs on either side of the boundary.

**9.10** Redesign the memory controller of Figure 9-26 using a slower clock so that no wait state is required, allowing a 2-2 bus cycle to be used. Assume state T1 follows state T2 after a read or write cycle, and make sure the data and control signals meet the timing specs.

(a) What is the fastest clock speed that can be used with the 43258A-25 static CMOS RAM with no wait states?

(b) Draw the SM chart for the new memory controller.

(c) Write VHDL code for the memory controller and verify its operation with the bus model.

**9.11** Redesign the memory controller from Figure 9-26 using a 43258A-15 static RAM with a 2-2 bus cycle and no wait states. The timing specifications for the 43258A-15 are as follows:  $t_{RC} = t_{AA} = t_{ACS} = t_{WC} = 15$  ns,  $t_{CLZ} = t_{OH} = t_{CHZ} = t_{WHZ} = 3$  ns (min),  $t_{CHZ} = 10$  ns (max),  $t_{WHZ} = 8$  ns (max),  $t_{CW} = t_{AW} = t_{WP} = 12$  ns,  $t_{DW} = 9$ .

(a) What is the longest propagation delay allowed for the memory controller?

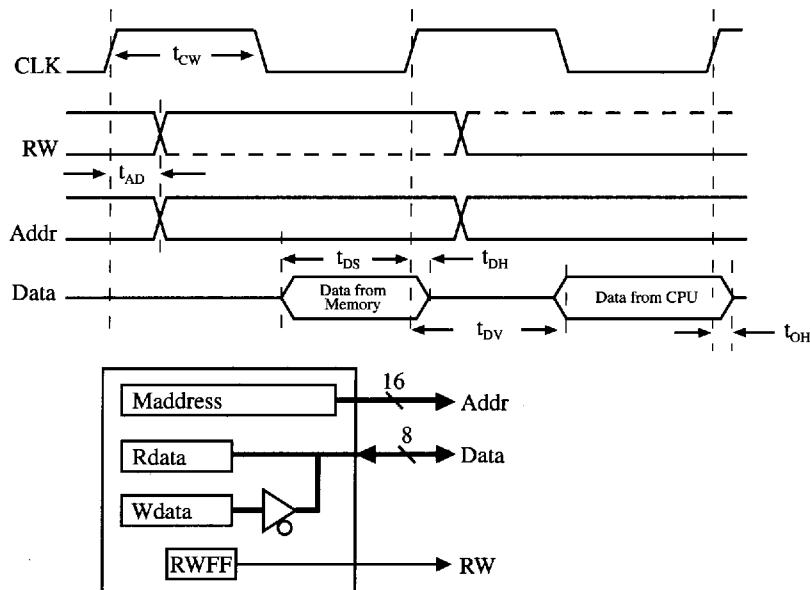
(b) Draw the SM chart for the new memory controller.

(c) Write VHDL code for the memory controller and verify it by simulation with the bus model.

**9.12** A simple microprocessor transfers one byte of data during every clock cycle, as shown in the following timing diagram. The internal registers in the microprocessor include a 16-bit *Maddress* register, an 8-bit *Rdata* register (which stores input data read from memory), an 8-bit *Wdata* register (which contains data to be written to memory) and a RWFF (read-write flip-flop), which is set to 1 during a read cycle and to 0 during a write cycle. These registers change state in response to the rising edge of the clock.

(a) Write VHDL code that describes the operation of the microprocessor bus interface. Assume there already exists a process that updates the internal registers on the rising edge of the clock, so you do not need to write this process. Your VHDL code should output the bus signals at the proper time.

(b) Write a process that verifies the setup and hold times are satisfied during a read cycle.



# CHAPTER 10

## HARDWARE TESTING AND DESIGN FOR TESTABILITY

This chapter introduces digital system testing and design methods that make the systems easier to test. We have already discussed the use of testing during the design process. We have written VHDL test benches to verify that the overall design and algorithms used are correct. We have used simulation at the logic level to verify that a design is logically correct and that it meets specifications. After the design of an IC is completed, additional testing can be done by simulating it at the circuit level to verify that the design has been correctly implemented and that the timing is correct.

When a digital system is manufactured, further testing is required to verify that it functions correctly. When multiple copies of an IC are manufactured, each copy must be tested to verify that it is free from manufacturing defects. This testing process can become very expensive and time consuming. With today's complex ICs, the cost of testing is a major component of the manufacturing cost. Therefore, it is very important to develop efficient methods of testing digital systems and to design the systems so that they are easy to test.

In this chapter, we first discuss methods of testing combinational logic for the basic types of faults that can occur. Then we describe methods for determining test sequences for sequential logic. One of the problems encountered is that normally we have access only to the inputs and outputs of the circuit being tested and not to the internal state. To remedy this problem, internal test points may be brought out to additional pins on the IC. To reduce the number of test pins required, we introduce the concept of scan design, in which the state of the system can be stored in a shift register and shifted out serially. Finally, we discuss the concept of built-in self-test. By adding more components to the IC, we can generate test sequences and verify the response to these sequences internally without the need for expensive external testing.

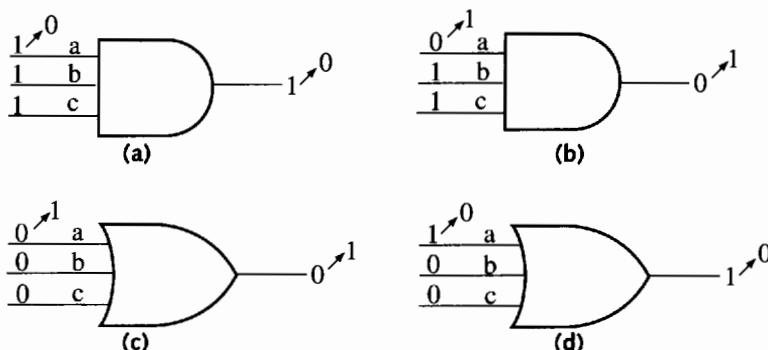
### 10.1 TESTING COMBINATIONAL LOGIC

Two common types of faults are short circuits and open circuits. If the input to a gate is shorted to ground, the input acts as if it is stuck at a logic 0. If the input to a gate is shorted to a positive power supply voltage, the gate input acts as if it is stuck at a logic 1. If the input to a gate is an open circuit, the input may act as if it is stuck at 0 or stuck at 1, depending on the type of logic being used. Thus, it is common practice to model faults in logic circuits as stuck-at-1 (s-a-1) or stuck-at-0 (s-a-0) faults. To test a gate input for s-a-0,

the gate input must be 1 so a change to 0 can be detected. Similarly, to test a gate input for s-a-1, the normal gate input must be 0 so a change to 1 can be detected.

We can test an AND gate for s-a-0 faults by applying 1s to all inputs, as shown in Figure 10-1(a). The normal gate output is then 1, but if any input is s-a-0, the output becomes 0. The notation  $1 \rightarrow 0$  on the gate input  $a$  means that the normal value of  $a$  is 1, but the value has changed to 0 because of the s-a-0 fault. The notation  $1 \rightarrow 0$  at the gate output indicates that this change has propagated to the gate output. We can test an AND gate input for s-a-1 by applying 0 to the input being tested and 1s to the other inputs, as shown in Figure 10-1(b). The normal gate output then is 0, but if the input being tested is s-a-1, the output becomes 1. To test OR gate inputs for s-a-1, we apply 0s to all inputs, and if any input is s-a-1, the output will change to 1 (Figure 10-1(c)). To test an OR gate input for s-a-0, we apply a 1 to the input under test and 0s to the other inputs. If the input under test is s-a-0, the output will change to 0 (Figure 10-1(d)). In the process of testing the inputs to a gate for s-a-0 and s-a-1, we also can detect s-a-0 and s-a-1 faults at the gate output.

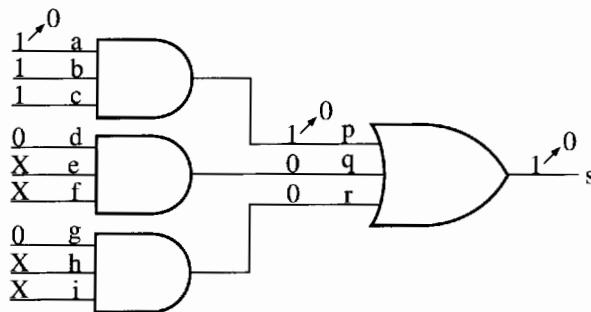
**Figure 10-1 Testing AND and OR Gates for Stuck-at Faults**



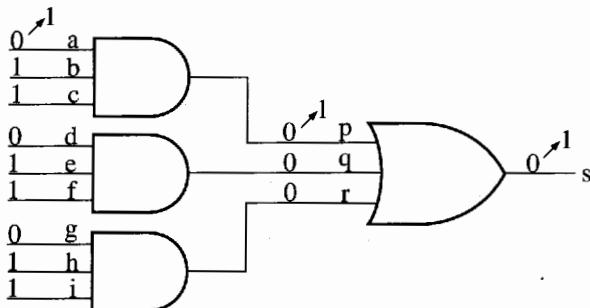
The two-level AND-OR network of Figure 10-2 has nine inputs and one output. We assume that the OR gate inputs ( $p$ ,  $q$ , and  $r$ ) are not accessible, so the gates cannot be tested individually. One approach to testing the network would be to apply all  $2^9 = 512$  different input combinations and observe the output. A more efficient approach is based on testing for all s-a-0 and s-a-1 faults, as shown in Table 10-1. To test the  $abc$  AND gate inputs for s-a-0, we must apply 1s to  $a$ ,  $b$ , and  $c$ , as shown in Figure 10-2(a). Then, if any gate input is s-a-0, the gate output ( $p$ ) will become 0. In order to transmit the change to the OR gate output, the other OR gate inputs must be 0. To achieve this, we can set  $d = 0$  and  $g = 0$  ( $e$ ,  $f$ ,  $h$ , and  $i$  are then don't cares). This test vector will detect  $p0$  ( $p$  stuck-at-0) as well as  $a0$ ,  $b0$ , and  $c0$ . In a similar manner, we can test for  $d0$ ,  $e0$ ,  $f0$ , and  $q0$  by setting  $d = e = f = 1$  and  $a = g = 0$ . A third test with  $g = h = i = 1$  and  $a = d = 0$  will test the remaining s-a-0 faults. To test  $a$  for s-a-1 ( $a1$ ), we must set  $a = 0$  and  $b = c = 1$ , as shown in Figure 10-2(b). Then, if  $a$  is s-a-1,  $p$  will become 1. In order to transmit this change to the output, we must have  $q = r = 0$ , as before. However, if we set  $d = g = 0$  and  $e = f = h = i = 1$ , we can test for  $d1$  and  $g1$  at the same time as  $a1$ . This same test vector also tests for  $p1$ ,  $q1$ , and  $r1$ . As shown in the table, we can test for  $b1$ ,  $e1$ , and  $h1$  with a single test vector and test similarly for  $c1$ ,  $f1$ , and  $i1$ . Thus we can test all s-a-0 and s-a-1 faults with only six tests, whereas the brute-

force approach would require 512 tests. When we apply the six tests, we can determine whether or not a fault is present, but we cannot determine the exact location of the fault. In the preceding analysis, we have assumed that only one fault occurs at a time. In many cases the presence of multiple faults will also be detected.

**Figure 10-2 Testing an AND-OR Network**



(a) Stuck-at-0 test



(b) Stuck-at-1 test

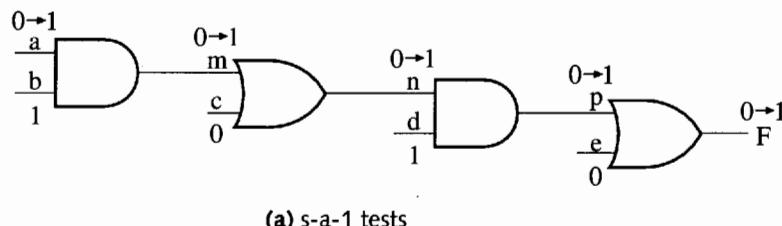
**Table 10-1 Test Vectors for Figure 10-2**

a	b	c	d	e	f	g	h	i	Faults Tested
1	1	1	0	X	X	0	X	X	a0, b0, c0, p0
0	X	X	1	1	1	0	X	X	d0, e0, f0, q0
0	X	X	0	X	X	1	1	1	g0, h0, i0, r0
0	1	1	0	1	1	0	1	1	a1, d1, g1, p1, q1, r1
1	0	1	1	0	1	1	0	1	b1, e1, h1, p1, q1, r1
1	1	0	1	1	0	1	1	0	c1, f1, i1, p1, q1, r1

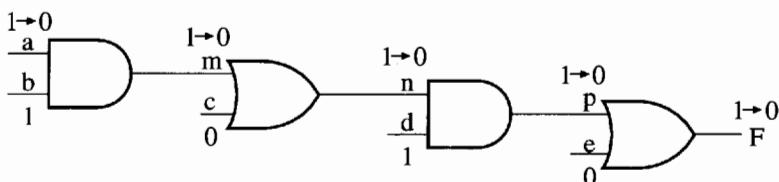
Testing multilevel networks is considerably more complex than testing two-level networks. In order to test for an internal fault in a network, we must choose a set of inputs that will excite that fault and then propagate the effect of that fault to the network output. In Figure 10-3, *a*, *b*, *c*, *d*, and *e* are network inputs. If we want to test for gate input *n*

$s-a-1$ ,  $n$  must be 0. This can be achieved if we make  $c = 0$ ,  $a = 0$ , and  $b = 1$ , as shown. In order to propagate the fault  $n$   $s-a-1$  to the output  $F$ , we must make  $d=1$  and  $e=0$ . With this set of inputs, if  $a, m, n$ , or  $p$  is  $s-a-1$ , the output  $F$  will have the incorrect value and the fault can be detected. Furthermore, if we change  $a$  to 1 and gate input  $a, m, n$ , or  $p$  is  $s-a-0$ , the output  $F$  will change from 1 to 0. We say that the path through  $a, m, n$ , and  $p$  has been sensitized, since any fault along that path can be detected. The method of path sensitization allows us to test for a number of different stuck-at faults using one set of network inputs.

Figure 10-3 Fault Detection Using Path Sensitization



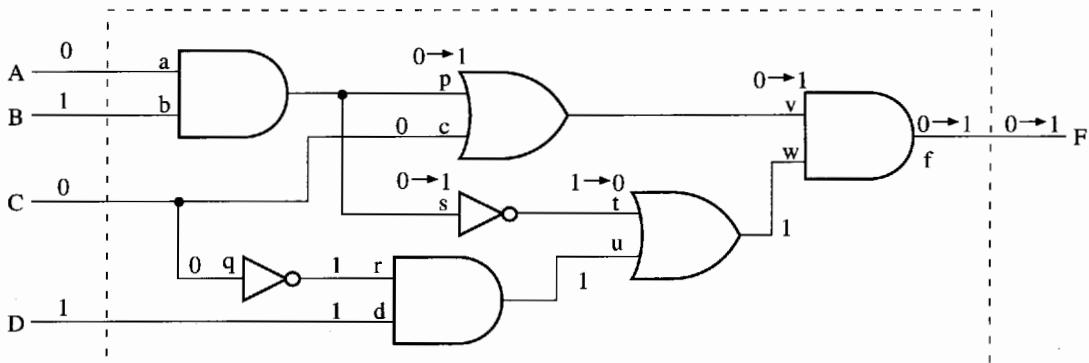
(a)  $s-a-1$  tests



(b)  $s-a-0$  tests

Next, we try to determine a minimum set of test vectors to test the network of Figure 10-4 for all single stuck-at-1 and stuck-at-0 faults. We assume that we can apply inputs to  $A, B, C$ , and  $D$  and observe the output  $F$  and that the internal gate inputs and outputs cannot be accessed. Suppose that we want to test input  $p$  for  $s-a-1$ . In order to do this, we must choose inputs  $A, B, C$ , and  $D$  such that  $p = 0$ , and if  $p$  is  $s-a-1$ , we must propagate this fault to the output  $F$  so it can be observed. In order to propagate the fault, we must make  $c = 0$  and  $w = 1$ . We can make  $w = 1$  by making  $t = 1$  or  $u = 1$ . To make  $u = 1$ , we must have both  $D$  and  $r = 1$ . Fortunately, our choice of  $C = 0$  makes  $r = 1$ . To make  $p = 0$ , we choose  $A = 0$ . By choosing  $B = 1$ , we can sensitize the path  $A-a-p-v-f-F$  so that the set of inputs  $ABCD = 0101$  will test for faults  $a1, p1, v1$ , and  $f1$ . This set of inputs also tests for  $c$   $s-a-1$ . We assume that  $c$   $s-a-1$  is a fault internal to the gate, so it is still possible to have  $q = 0$  and  $r = 1$  if  $c$   $s-a-1$  occurs.

Figure 10-4 Example Network for Stuck-at Fault Testing



To test for  $s-a-0$  inputs along the path  $A-a-p-v-f-F$ , we can use the inputs  $ABCD = 1101$ . In addition to testing for faults  $a0$ ,  $p0$ ,  $v0$ , and  $f0$ , this input vector also tests the following faults:  $b0$ ,  $w0$ ,  $u0$ ,  $r0$ ,  $q1$ , and  $d0$ . To determine tests for the remaining stuck-at faults, we can select an untested fault, determine the required  $ABCD$  inputs, and then determine the additional faults that are tested. Then we can repeat this procedure until tests are found for all of the faults. Table 10-2 lists a set five test vectors that will test for all single stuck-at faults in Figure 10-4.

Table 10-2 Tests for Stuck-at Faults in Figure 10-4

Normal Gate Inputs								Faults Tested								
A	B	C	D	a	b	p	c	q	r	d	s	t	u	v	w	Faults Tested
0	1	0	1	0	1	0	0	0	1	1	0	1	1	0	1	a1 p1 c1 v1 f1
1	1	0	1	1	1	1	0	0	1	1	1	1	0	1	1	a0 b0 p0 q1 r0 d0 u0 v0 w0 f0
1	0	1	1	1	0	0	1	1	0	1	0	1	0	1	1	b1 c0 s1 t0 v0 w0 f0
1	1	0	0	1	1	1	0	0	1	0	1	0	0	1	0	a0 b0 d1 s0 t1 u1 w1 f1
1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	0	a0 b0 q0 r1 s0 t1 u1 w1 f1

In addition to stuck-at faults, other types of faults, such as bridging faults, may occur. A bridging fault occurs when two unconnected signal lines are shorted together. For a large combinational network, finding a minimum set of test vectors that will test for all possible faults is very difficult and time consuming. For networks that contain redundant gates, testing for some of the faults may be impossible. Even if a comprehensive set of test vectors can be found, applying all of the vectors may take too much time and cost too much. For these reasons, it is common practice to use a relatively small set of test vectors that will test most of the faults. In general, determining such a set of vectors is a difficult and computationally intensive problem. Many algorithms and corresponding computer programs have been developed to generate such sets of test vectors. Computer programs have also been developed to simulate faulty networks. Such programs allow the user to determine what percentage of possible faults are tested by a given set of input vectors.

## 10.2 TESTING SEQUENTIAL LOGIC

Testing sequential logic is generally much more difficult than testing combinational logic, because we must use sequences of inputs for testing. If we can observe only the input and output sequences and not the state of the flip-flops in a sequential network, a very large number of test sequences may be required. Basically, the problem is to determine if the network under test is equivalent to a correctly functioning network. We will assume that the sequential network being tested has a reset input so we can reset it to a known initial state. If we attempted to test the network using the brute-force approach, we would reset the network to the initial state, apply a test sequence, and observe the output sequence. If the output sequence was correct, then we would repeat the test for another sequence. In theory, according to the definition of state equivalence, we would have to try all possible input sequences, which is an infinite number, to verify that the initial state of the network under test is equivalent to the initial state of the correct network. In practice, if we know that the network has  $N$  or fewer states, then we have to apply only input sequences of length less than or equal to  $2N - 1$ .

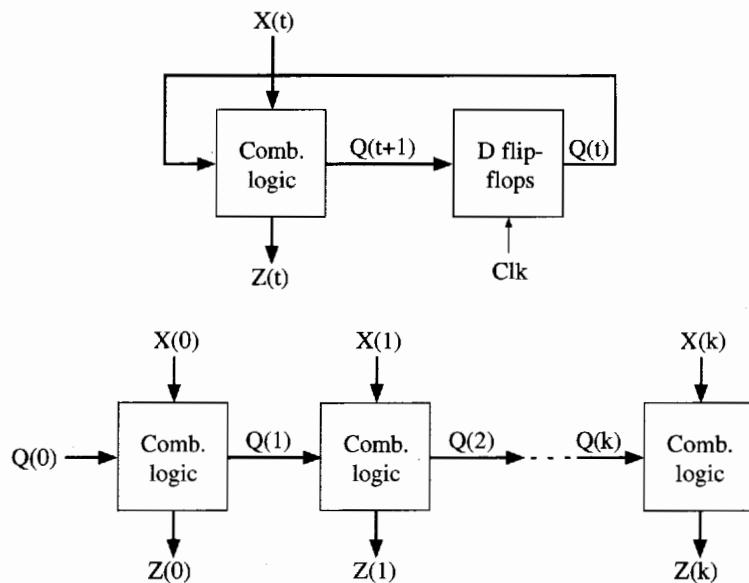
As an example, consider a sequential network that has five inputs, one output, and four states. If we used the brute-force approach, we would have to apply all input sequences of length seven or less. At time = 1, we could apply any one of the  $2^5$  input combinations, and similarly at times 2 through 7. Thus, the total number of test sequences required is

$$(2^5)(2^5)(2^5)(2^5)(2^5)(2^5) = 2^{35}$$

Even in this simple example, the number of test sequences would be prohibitive.

Since the brute-force approach is totally impractical, the question then arises, Can we derive a relatively small set of test sequences that will adequately test the network? One way to derive test sequences for a sequential network is to convert it to an iterative network. Since the iterative network is a combinational network, we could then derive test vectors for the iterative network using one of the standard methods for combinational networks.

Figure 10-5 shows a standard Mealy sequential network and the corresponding iterative network. In these figures,  $X$ ,  $Z$ , and  $Q$  can either be single variables or vectors. The iterative network has  $k$  identical copies of the Comb. network used in the sequential network, where  $k$  is the length of the sequence used to test the sequential network. For the sequential network,  $X(t)$  represents a sequence of inputs in time. In the iterative network,  $X(0) X(1) \dots X(k)$  represents the same sequence in space. Each cell of the iterative network computes  $Z(t)$  and  $Q(t+1)$  in terms of  $Q(t)$  and  $X(t)$ . The leftmost cell computes the values for  $t=0$ , the next cell for  $t=1$ , etc. After the test vectors have been derived for the iterative network, these vectors then become the input sequences used to test the original sequential network. The number of cells in the iterative network depends on the length of the sequences required to test the sequential network.

**Figure 10-5 Sequential and Iterative Networks**

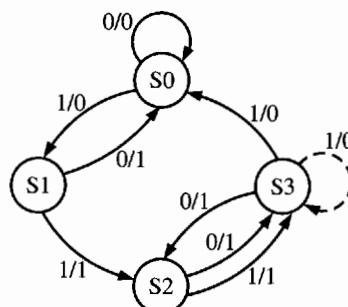
Derivation of a small set of test sequences that will adequately test a sequential network is generally difficult to do. Consider the state graph shown in Figure 10-6 and the corresponding state table (Table 10-3). We assume that we can reset the network to state S0. It is certainly necessary that the test sequence cause the network to go through all possible state transitions, but this is not an adequate test. For example, the input sequence

$$X = 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1$$

traverses all the arcs connecting the states and produces the output sequence

$$Z = 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0$$

If we replace the arc from S3 to S0 with a self-loop, as shown by the dashed line, the output sequence will be the same, but the new sequential machine is not equivalent to the old one.

**Figure 10-6 State Graph for Test Example**

**Table 10-3 State Table for Figure 10-6**

$Q_1 Q_2$	State	Next State		Output	
		$X = 0$	$1$	$X = 0$	$1$
00	S0	S0	S1	0	0
10	S1	S0	S2	1	1
01	S2	S3	S3	1	1
11	S3	S2	S0	1	0

A state graph in which every state can be reached from every other state is referred to as *strongly connected*. A general test strategy for a sequential network with a strongly connected state graph and no equivalent states is to first find an input sequence that will distinguish each state from the other states. Such an input sequence is referred to as a distinguishing sequence. Given a distinguishing sequence, each entry in the state table can be verified. For the example of Figure 10-6, one distinguishing sequence is 11. If we start in S0, the input sequence 11 gives the output sequence 01; for S1 the output is 11; for S2, 10; and for S3, 00. Thus, we can distinguish the four states by using the input sequence 11. We can then verify every entry in the state table using the following sequences, where R means reset to state S0:

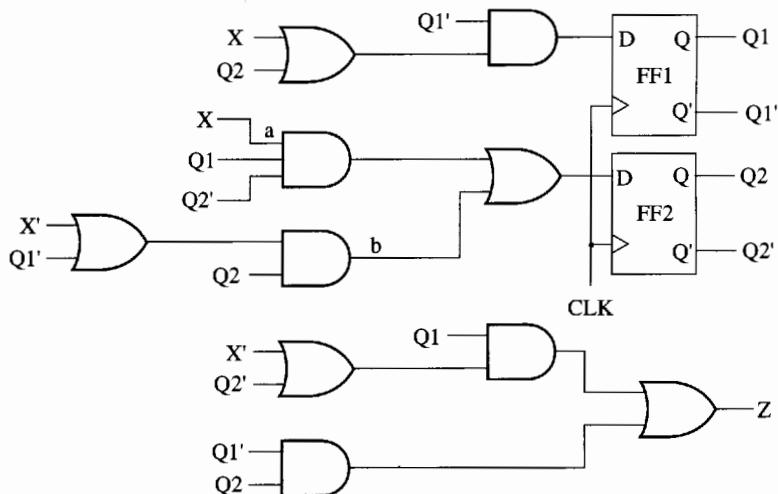
Input	Output	Transition Verified
R 0 1 1	0 0 1	(S0 to S0)
R 1 1 1	0 1 1	(S0 to S1)
R 1 0 1 1	0 1 0 1	(S1 to S0)
R 1 1 1 1	0 1 1 0	(S1 to S2)
R 1 1 0 1 1	0 1 1 0 0	(S2 to S3)
R 1 1 1 1 1	0 1 1 0 0	(S2 to S3)
R 1 1 0 0 1 1	0 1 1 1 1 0	(S3 to S2)
R 1 1 0 1 1 1	0 1 1 0 1 0	(S3 to S0)

Another approach to deriving test sequences is based on testing for stuck-at faults. Figure 10-7 shows the realization of Figure 10-6 using the following state assignment: S0, 00; S1, 10; S2, 01; S3, 11. If we want to test for a s-a-1, we must first excite the fault by going to state S3, in which  $Q_1 Q_2 = 10$  and then setting  $X = 0$ . In normal operation, the next state will be S0. However, if a is s-a-1, then next state is  $Q_1 Q_2 = 01$ , which is S2. This test sequence can then be constructed as follows:

- To go to S1: reset followed by  $X = 1$ .
- To test a s-a-1:  $X = 0$ .
- To distinguish the state that is reached:  $X = 11$ .

The final sequence is R1011. The normal output is 0101, and the faulty output is 0110.

Figure 10-7 Realization of Figure 10-6



We have shown some simple examples that illustrate some of the methods used to derive test sequences for sequential networks. As the number of inputs and states in the network increases, the number and length of the required test sequence increases rapidly, and the derivation of these test sequences becomes much more difficult. This, in turn, means that the time and expense required to test the networks increases rapidly with the number of inputs and states.

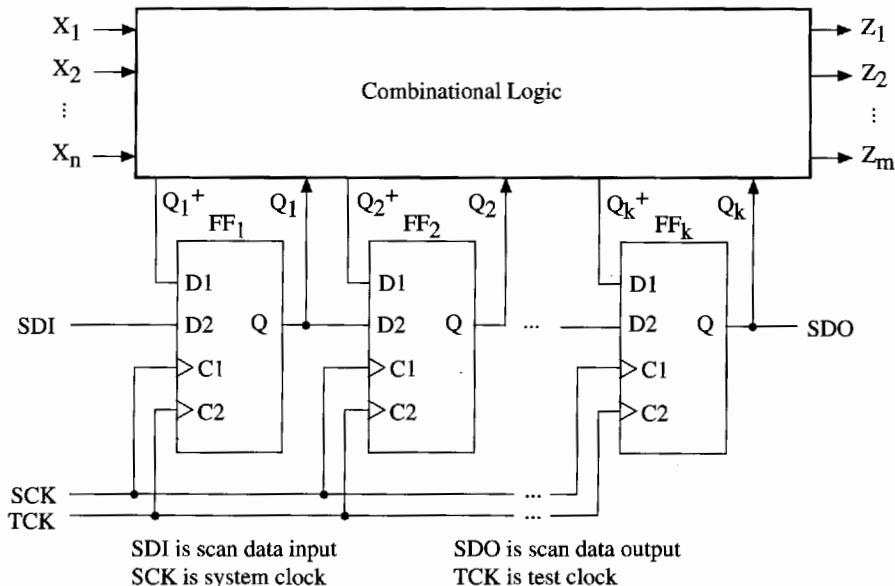
### 10.3 SCAN TESTING

The problem of testing a sequential network is greatly simplified if we can observe the state of all the flip-flops instead of just observing the network outputs. For each state of the flip-flops and for each input combination, we need to verify that the network outputs are correct and that the network goes to the correct next state. One approach would be to connect the output of each flip-flop within the IC being tested to one of the IC pins. Since the number of pins on the IC is very limited, this approach is not very practical. So the question arises, how can we observe the state of all the flip-flops without using up a large number of pins on the IC? If the flip-flops were arranged to form a shift register, then we could shift out the state of the flip-flops bit by bit using a single serial output pin on the IC. This leads to the concept of *scan path testing*.

Figure 10-8 shows a method of scan path testing based on two-port flip-flops. In the usual way, the sequential network is separated into a combinational logic part and a state register composed of flip-flops. Each of the flip-flops has two D inputs and two clock inputs. When  $C1$  is pulsed, the  $D1$  input is stored in the flip-flop. When  $C2$  is pulsed,  $D2$  is stored in the flip-flop. The  $Q$  output of each flip-flop is connected to the  $D2$  input of the next flip-flop to form a shift register. The next state ( $Q_1^+ Q_2^+ \dots Q_k^+$ ) generated by the combinational logic is loaded into the flip-flops when  $C1$  is pulsed, and the new state ( $Q_1 Q_2 \dots Q_k$ ) feeds back into the combinational logic. When the network is not being

tested, the system clock ( $SCK = CI$ ) is used. A set of inputs ( $X_1 X_2 \dots X_n$ ) is applied, the outputs ( $Z_1 Z_2 \dots Z_m$ ) are generated,  $SCK$  is pulsed, and the network goes to the next state.

Figure 10-8 Scan Path Test Circuit Using Two-port Flip-flops



When the network is being tested, the flip-flops are set to a specified state by shifting the state code into the register using the scan data input ( $SDI$ ) and the test clock ( $TCK$ ). A test input vector ( $X_1 X_2 \dots X_n$ ) is applied, the outputs ( $Z_1 Z_2 \dots Z_m$ ) are verified, and  $SCK$  is pulsed to take the network to the next state. The next state is then verified by pulsing  $TCK$  to shift the state code out of the scan data register via the scan data output ( $SDO$ ). This method reduces the problem of testing a sequential network to that of testing a combinational network. Any of the standard methods can be used to generate a set of test vectors for the combinational logic. Each test vector contains  $(n + k)$  bits, since there are  $n$   $X$  inputs and  $K$  state inputs to the combinational logic. The  $X$  part of the test vector is applied directly, and the  $Q$  part is shifted in via the  $SDI$ . In summary, the test procedure is as follows:

1. Scan in the test vector  $Q_i$  values via  $SDI$  using the test clock  $TCK$ .
2. Apply the corresponding test values to the  $X_i$  inputs.
3. After sufficient time for the signals to propagate through the combinational network, verify the output  $Z_i$  values.
4. Apply one clock pulse to the system clock  $SCK$  to store the new values of  $Q_i^+$  into the corresponding flip-flops.
5. Scan out and verify the  $Q_i$  values by pulsing the test clock  $TCK$ .
6. Repeat steps 1 through 5 for each test vector.

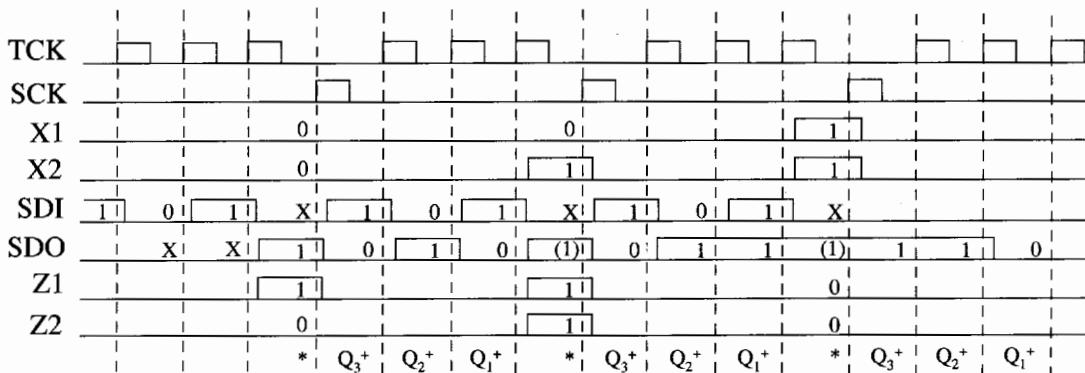
Steps 5 and 1 can overlap, since it is possible to scan in one test vector while scanning out the previous test result.

We will apply this method to test a sequential network with two inputs, three flip-flops, and two outputs. The network is configured as in Figure 10-8 with inputs  $X_1X_2$ , flip-flops  $Q_1Q_2Q_3$ , and outputs  $Z_1Z_2$ . One row of the state transition table is as follows:

$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$	$Z_1Z_2$
101	$X_1X_2 = 00 \quad 01 \quad 11 \quad 10$	$00 \quad 01 \quad 11 \quad 10$
	010 110 011 111	10 11 00 01

Figure 10-9 shows the timing diagram for testing this row of the transition table. First, 101 is shifted in using  $TCK$ , least significant bit ( $Q_3$ ) first. The input  $X_1X_2 = 00$  is applied, and  $Z_1Z_2 = 10$  is then read.  $SCK$  is pulsed and the network goes to state 010. As 010 is shifted out using  $TCK$ , 101 is shifted in for the next test. This process continues until the test is completed.

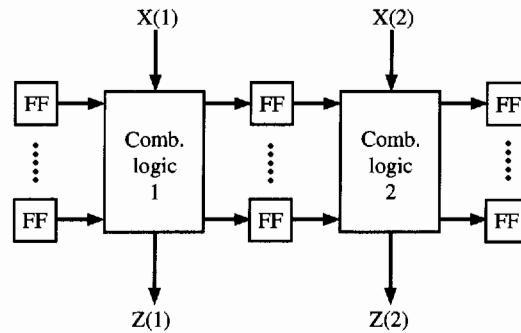
Figure 10-9 Timing Chart for Scan Test



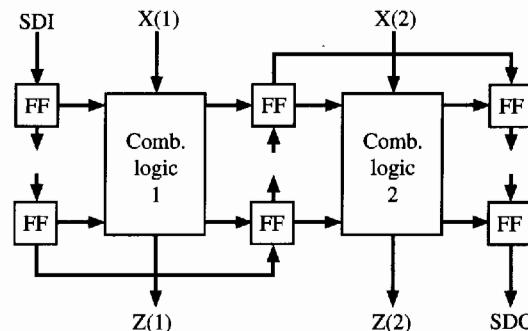
\* Read output (output at other times not shown)

In general, a digital system implemented by an IC consists of flip-flop registers separated by blocks of combinational logic, as shown in Figure 10-10(a). In order to apply scan test to the IC, we need to replace the flip-flops with two-port flip-flops (or other types of scannable flip-flops) and link all the flip-flops into a scan chain, as shown in Figure 10-10(b). Then we can scan test data into all the registers, apply the test clock, and then scan out the results.

**Figure 10-10 System with Flip-flop Registers and Combinational Logic Blocks**



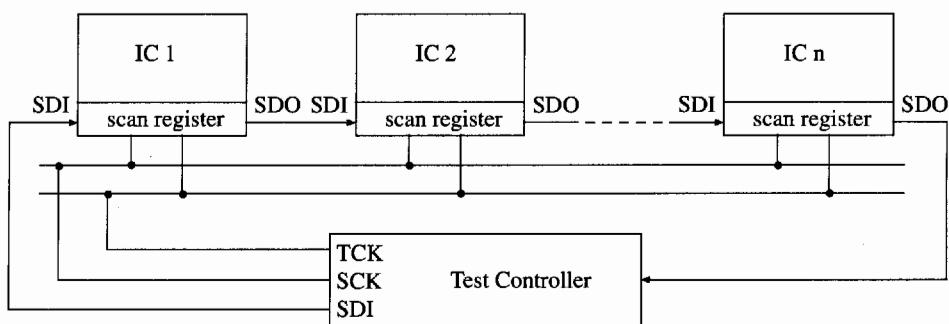
(a) Without scan chain



(b) With scan chain added

When multiple ICs are mounted on a PC board, it is possible to chain together the scan registers in each IC so that the entire board can be tested using a single serial access port (Figure 10-11).

**Figure 10-11 Scan Test Configuration with Multiple ICs**



## 10.4 BOUNDARY SCAN

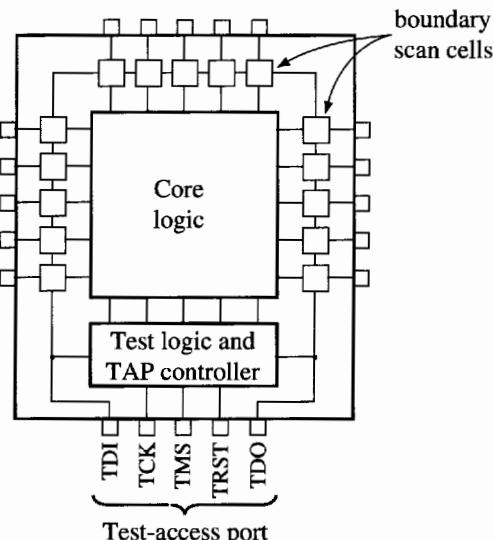
As ICs have become more complex, with more and more pins, printed circuit boards have become denser, with multiple layers and very fine traces. Testing these PC boards after they have been loaded with complex ICs has become very difficult. Testing a board by means of its edge connector does not provide adequate testing and may require very long test sequences. When PC boards were less dense with wider traces, testing was often done using a bed-of-nails test fixture. This method used sharp probes to contact the traces on the board so test data could be applied to and read from various ICs on the board. Bed-of-nails testing is not practical for high-density PC boards with fine traces and complex ICs.

Boundary scan test methodology was introduced to facilitate the testing of complex PC boards. A standard for boundary scan testing was developed by the Joint Test Action Group (JTAG), and this standard has been adopted as ANSI/IEEE Standard 1149.1, "Standard Test Access Port and Boundary-Scan Architecture." Many IC manufacturers make ICs that conform to this standard. Such ICs can be linked together on a PC board so that they can be tested using only a few pins on the PC board edge connector.

Figure 10-12 shows an IC with added boundary scan logic. One cell of the boundary scan register (BSR) is placed between each input or output pin and the internal core logic. Four or five pins of the IC are devoted to the test-access port, or TAP. The TAP controller and additional test logic are also added to the core logic on the IC. The functions of the TAP pins are as follows:

- TDI* Test data input (this data is shifted serially into the BSR)
- TCK* Test clock
- TMS* Test mode select
- TDO* Test data output (serial output from the BSR)
- TRST* Test reset (resets the TAP controller and test logic; optional pin)

Figure 10-12 IC with Boundary Scan Register and Test-access Port



A PC board with several boundary scan ICs is shown in Figure 10-13. The boundary scan registers in the ICs are linked together serially in a single chain with input *TDI* and output *TDO*. *TCK*, *TMS*, and *TRST* (if used) are connected in parallel to all of the ICs. Using these signals, test instructions and test data can be clocked into every IC on the board.

Figure 10-13 PC Board with Boundary Scan ICs

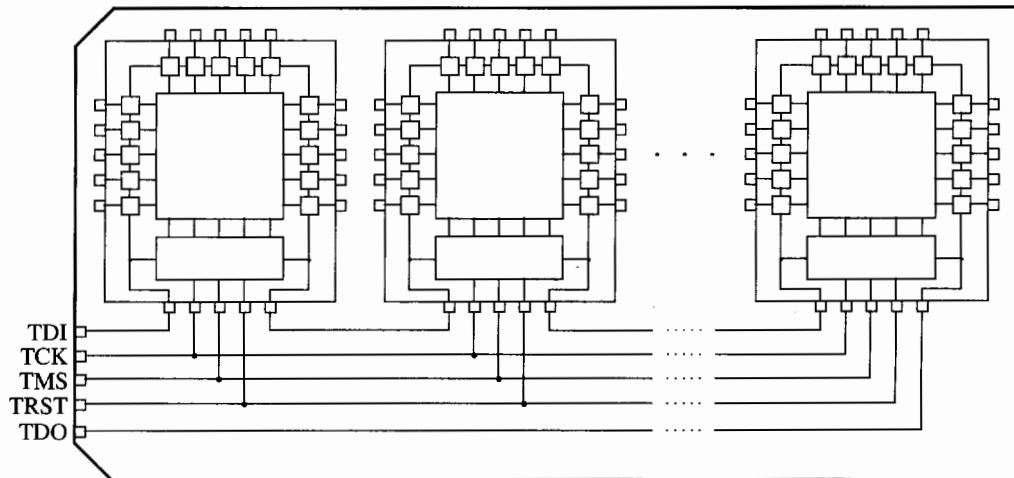


Figure 10-14 shows a typical boundary scan cell. When in the normal mode, data from the input pin is routed to the internal core logic in the IC, or data from the core logic is routed to the output pin. When in the shift mode, serial data from the previous cell is clocked into flip-flop *Q1* at the same time as the data stored in *Q1* is clocked into the next boundary scan cell. After *Q2* is updated, test data can be supplied to the internal logic or to the output pin.

Figure 10-14 Typical Boundary Scan Cell

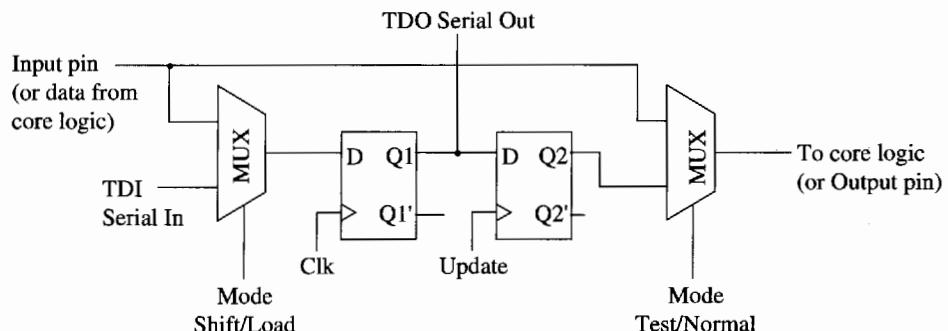
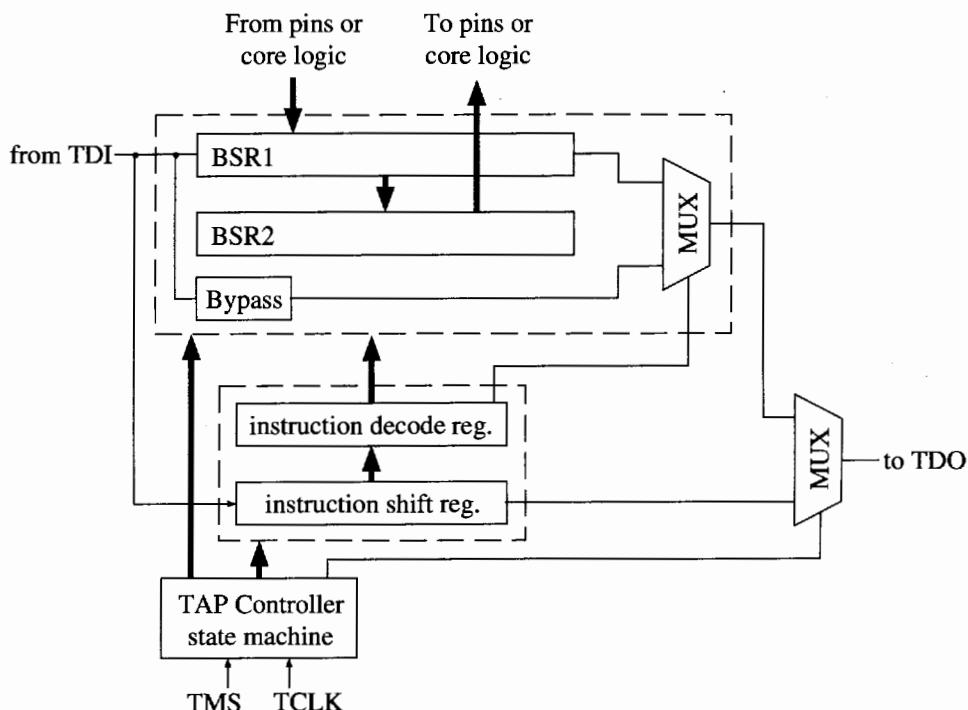
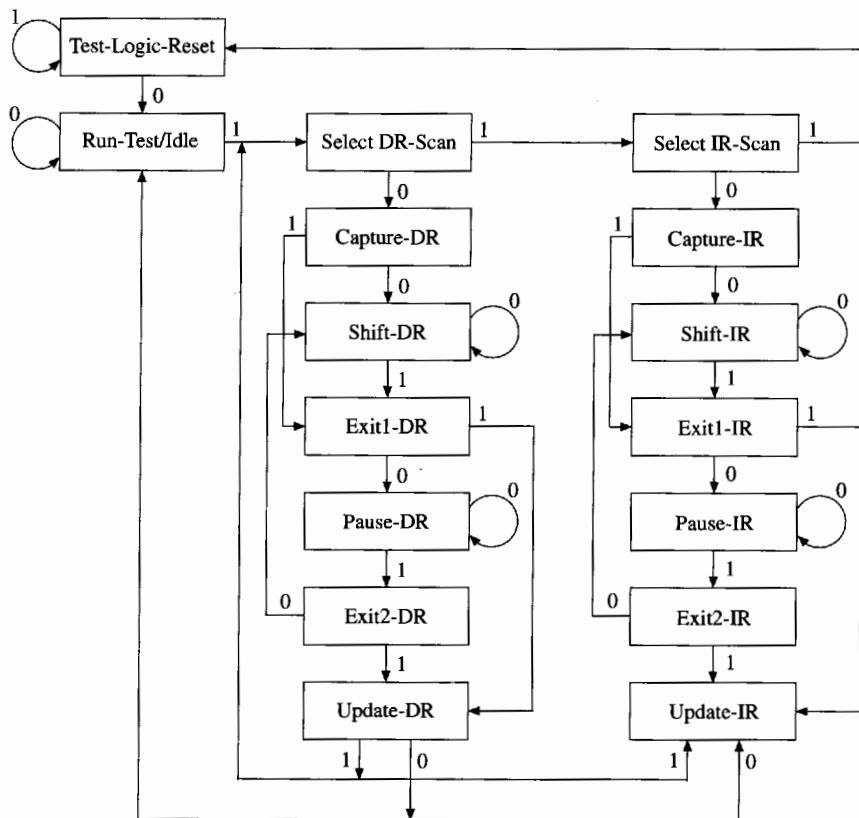


Figure 10-15 shows the basic boundary scan architecture that is implemented on each boundary scan IC. The boundary scan register is divided into two parts. BSR1 represents the shift register, which consists of the  $Q1$  flip-flops in the boundary scan cells. BSR2 represents the  $Q2$  flip-flops, which can be parallel-loaded from BSR1 when an update signal is received. The serial input data ( $TDI$ ) can be shifted into the boundary scan register (BSR1), through a bypass register, or into the instruction register. The TAP controller on each IC contains a state machine (Figure 10-16). The input to the state machine is  $TMS$ , and the sequence of 0s and 1s applied to  $TMS$  determines whether the  $TDI$  data is shifted into the instruction register or through the boundary scan cells. The TAP controller and the instruction register control the operation of the boundary scan cells.

Figure 10-15 Basic Boundary Scan Architecture



**Figure 10-16 State Machine for TAP Controller**



The following instructions are defined in the IEEE standard:

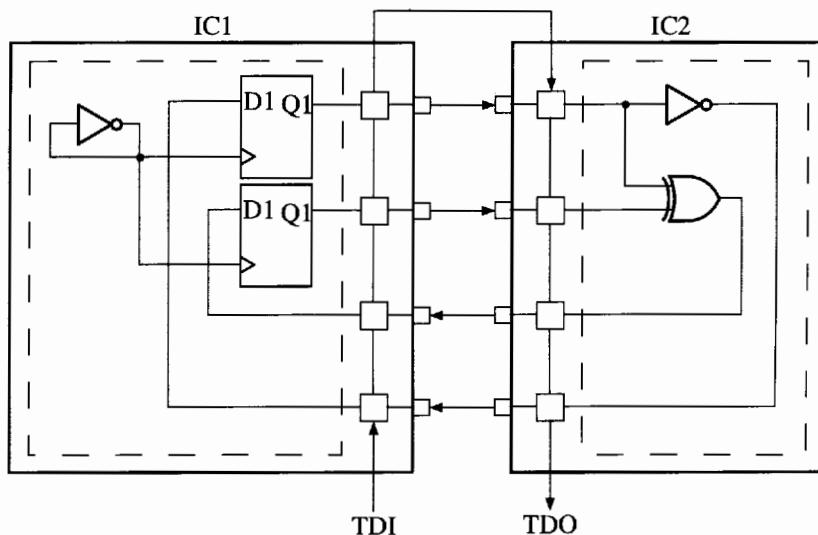
- **BYPASS:** This instruction allows the *TDI* serial data to go through a 1-bit bypass register on the IC instead of through the boundary scan register. In this way, one or more ICs on the PC board may be bypassed while other ICs are being tested.
- **SAMPLE/PRELOAD:** This instruction is used to scan the boundary scan register without interfering with the normal operation of the core logic. Data is transferred to or from the core logic from or to the IC pins without interference. Samples of this data can be taken and scanned out through the boundary scan register. Test data can be shifted into the BSR.
- **EXTEST:** This instruction allows board-level interconnect testing, and it also allows testing of clusters of components that do not incorporate the boundary scan test features. Test data is shifted into the BSR and then it goes to the output pins. Data from the input pins is captured by the BSR.
- **INTEST (optional):** This instruction allows testing of the core logic by shifting test data into the boundary scan register. Data shifted into the BSR takes the place of data from the input pins, and output data from the core logic is loaded into the BSR.

- RUNBIST (optional): This instruction causes special built-in self-test (BIST) logic within the IC to execute. (Section 10.5 explains how BIST logic can be used to generate test sequences and check the test results.)

Several other optional and user-defined instructions may also be included.

The following simplified example illustrates how the connections between two ICs can be tested using the SAMPLE/PRELOAD and EXTEST instructions. The test is intended to check for shorts and opens in the PC board traces. Both ICs have two input pins and two output pins, as shown in Figure 10-17. Test data is shifted into the BSRs via *TDI*. Then data from the input pins is parallel-loaded into the BSRs and shifted out via *TDO*. We assume that the instruction register on each IC is three bits long with EXTEST coded as 000 and SAMPLE/PRELOAD as 001. The core logic in IC1 is an inverter connected as a clock oscillator and two flip-flops. The core logic in IC2 is an inverter and XOR gate. The two ICs are interconnected to form a 2-bit counter.

**Figure 10-17 Interconnection Testing Using Boundary Scan**



The steps required to test the connections between the ICs are as follows:

1. Reset the TAP state machine to the Test-Logic-Reset state by inputting a sequence of five 1s on *TMS*. The TAP controller is designed so that a sequence of five 1s will always reset it, regardless of the present state. Alternatively, *TRST* can be asserted if it is available.
2. Scan in the SAMPLE/PRELOAD instruction to both ICs using the sequences for *TMS* and *TDI* given here. The state numbers refer to Figure 10-16.

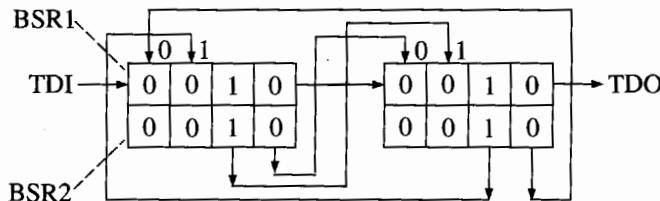
State:	0	1	2	9	10	11	11	11	11	11	11	12	15	2
TMS:	0	1	1	0	0	0	0	0	0	0	1	1	1	1
TDI:	-	-	-	-	-	1	0	0	1	0	0	-	-	-

The *TMS* sequence 01100 takes the TAP controller to the Shift-IR state. In this state, copies of the SAMPLE/PRELOAD instruction (code 001) are shifted into the instruction registers on both ICs. In the Update-IR state, the instructions are loaded into the instruction decode registers. Then the TAP controller goes back to the Select DR-scan state.

3. Preload the first set of test data into the ICs using the following sequences for *TMS* and *TDI*:

```
State: 2 3 4 4 4 4 4 4 4 4 5 8 2
TMS:   0 0 0 0 0 0 0 0 0 1 1 1
TDI:   - - 0 1 0 0 0 1 0 0 - -
```

Data is shifted into BSR1 in the Shift-DR state, and it is transferred to BSR2 in the Update-DR state. The result is as follows:



4. Scan in the EXTEST instruction to both ICs using the following sequences:

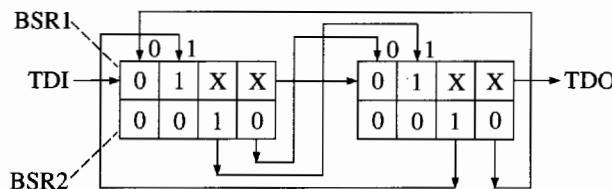
```
State: 2 9 10 11 11 11 11 11 11 11 12 15 2
TMS:   1 0 0 0 0 0 0 0 0 1 1 1
TDI:   - - - 0 0 0 0 0 0 - -
```

The EXTEST instruction (000) is scanned into the instruction register in state Shift-IR and loaded into the instruction decode register in state Update-IR. At this point, the preloaded test data goes to the output pins, and it is transmitted to the adjacent IC input pins via the printed circuit board traces.

5. Capture the test results from the IC inputs. Scan this data out to *TDO* and scan the second set of test data in using the following sequences:

```
State: 2 3 4 4 4 4 4 4 4 4 5 8 2
TMS:   0 0 0 0 0 0 0 0 0 1 1 1
TDI:   - - 1 0 0 0 1 0 0 0 - -
TDO:   - - x x 1 0 x x 1 0 - -
```

The data from the input pins is loaded into BSR1 in state Capture-DR. At this time, if no faults have been detected, the BSRs should be configured as shown below, where the Xs indicate captured data that is not relevant to the test.



The test results are then shifted out of BSR1 in state Shift-DR as the new test data is shifted in. The new data is loaded into BSR2 in the Update-IR state.

6. Capture the test results from the IC inputs. Scan this data out to *TDO* and scan all 0s in using the following sequences:

```
State: 2 3 4 4 4 4 4 4 4 4 5 8 2 9 0
TMS: 0 0 0 0 0 0 0 0 0 1 1 1 1 1
TDI: - - 0 0 0 0 0 0 0 0 - - - -
TDO: - - x x 0 1 x x 0 1 - - - -
```

The data from the input pins is loaded into BSR1 in state Capture-DR. Then it is shifted out in state Shift-DR as all 0s are shifted in. The 0s are loaded into BSR2 in the Update-IR state. The controller then returns to the Test-Logic-Reset state, and normal operation of the ICs can then occur. The interconnection test passes if the observed *TDO* sequences match the ones given above.

VHDL code for the basic boundary scan architecture of Figure 10-15 is given in Figure 10-18. Only the three mandatory instructions (EXTEST, SAMPLE/PRELOAD, and BYPASS) are implemented using a 3-bit instruction register. These instructions are coded as 000, 001, and 111, respectively. The number of cells in the *BSR* is a generic parameter. A second generic parameter, *CellType*, is a bit\_vector that specifies whether each cell is an input cell or output cell. The case statement implements the TAP controller state machine. The instruction code is scanned in and loaded into *IDR* in states Capture-IR, Shift-IR, and Update-IR. The instructions are executed in states Capture-DR, Shift-DR, and Update-DR. The actions taken in these states depend on the instruction being executed. The register updates and state changes all occur on the rising edge of *TCK*. The VHDL code implements most of the functions required by the IEEE boundary scan standard, but it does not fully comply with the standard.

**Figure 10-18 VHDL Code for Basic Boundary Scan Architecture**

```
-- VHDL for Boundary Scan Architecture of Figure 10-15

entity BS_arch is
  generic (NCELLS: natural range 2 to 120 := 2);
  -- number of boundary scan cells
  port (TCK, TMS, TDI: in bit;
        TDO: out bit;
        BSRin: in bit_vector(1 to NCELLS);
        BSRout: inout bit_vector(1 to NCELLS);
        CellType: bit_vector(1 to NCELLS));
        -- '0' for input cell, '1' for output cell
end BS_arch;
```

```

architecture behavior of BS_arch is
  signal IR, IDR: bit_vector(1 to 3);          -- instruction registers
  signal BSR1, BSR2: bit_vector(1 to NCELLS);   -- boundary scan cells
  signal BYPASS: bit;                          -- bypass bit
  type TAPstate is (TestLogicReset, RunTest_Idle,
    SelectDRScan, CaptureDR, ShiftDR, Exit1DR, PauseDR, Exit2DR, UpdatedDR,
    SelectIRScan, CaptureIR, ShiftIR, Exit1IR, PauseIR, Exit2IR, UpdateIR);
  signal St: TAPstate;                         -- TAP Controller State
begin
  process (TCK)
  begin
    if (TCK='1') then
      -- TAP Controller State Machine
      case St is
        when TestLogicReset =>
          if TMS='0' then St<=RunTest_Idle; else St<=TestLogicReset; end if;
        when RunTest_Idle =>
          if TMS='0' then St<=RunTest_Idle; else St<=SelectDRScan; end if;
        when SelectDRScan =>
          if TMS='0' then St<=CaptureDR;     else St<=SelectIRScan; end if;
        when CaptureDR =>
          if IDR = "111" then BYPASS <= '0';
          elsif IDR = "000" then -- EXTEST (input cells capture pin data)
            BSR1 <= (not CellType and BSRin) or (CellType and BSR1);
          elsif IDR = "001" then -- SAMPLE/PRELOAD
            BSR1 <= BSRin; end if; -- all cells capture cell input data
          if TMS='0' then St<=ShiftDR;     else St<=Exit1DR; end if;
        when ShiftDR =>
          if IDR = "111" then BYPASS <= TDI; -- shift data through bypass reg.
          else BSR1 <= TDI & BSR1(1 to NCELLS-1); end if;
          -- shift data into BSR
          if TMS='0' then St<=ShiftDR;     else St<=Exit1DR; end if;
        when Exit1DR =>
          if TMS='0' then St<=PauseDR;     else St<=UpdateDR; end if;
        when PauseDR =>
          if TMS='0' then St<=PauseDR;     else St<=Exit2DR; end if;
        when Exit2DR =>
          if TMS='0' then St<=ShiftDR;     else St<=UpdateDR; end if;
        when UpdateDR =>
          if IDR = "000" then -- EXTEST (update output reg. for output cells)
            BSR2 <= (CellType and BSR1) or (not CellType and BSR2);
          elsif IDR = "001" then -- SAMPLE/PRELOAD
            BSR2 <= BSR1; end if; -- update output reg. in all cells
          if TMS='0' then St<=RunTest_Idle; else St<=SelectDRScan; end if;
        when SelectIRScan =>
          if TMS='0' then St<=CaptureIR;   else St<=TestLogicReset; end if;
        when CaptureIR =>
          IR <= "001"; -- load 2 LSBs of IR with 01 as required by the
                        standard
          if TMS='0' then St<=ShiftIR;     else St<=Exit1IR; end if;
      end case;
    end if;
  end process;
end;

```

```

when ShiftIR =>
    IR <= TDI & IR(1 to 2); -- shift in instruction code
    if TMS='0' then St<=ShiftIR;      else St<=Exit1IR;      end if;
when Exit1IR =>
    if TMS='0' then St<=PauseIR;      else St<=UpdateIR;      end if;
when PauseIR =>
    if TMS='0' then St<=PauseIR;      else St<=Exit2IR;      end if;
when Exit2IR =>
    if TMS='0' then St<=ShiftIR;      else St<=UpdateIR;      end if;
when UpdateIR =>
    IDR <= IR; -- update instruction decode register
    if TMS='0' then St<=RunTest_Idle; else St<=SelectDRScan; end if;
end case;
end if;
end process;

TDO <= BYPASS when St = ShiftDR and IDR = "111" -- BYPASS
else BSR1(NCELLS) when St=ShiftDR -- EXTEST or SAMPLE/PRELOAD
else IR(3) when St=ShiftIR;

BSRout <= BSRin when (St = TestLogicReset or not (IDR = "000"))
else BSR2; -- define cell outputs
end behavior;

```

VHDL code that implements the interconnection test example of Figure 10-17 is given in Figure 10-19. The *TMS* and *TDI* test patterns are the concatenation of the test patterns used in steps 2 through 6. A copy of the basic boundary scan architecture is instantiated for IC1 and for IC2. The external connections and internal logic for each IC are then specified. The internal clock frequency was arbitrarily chosen to be different than the test clock frequency. The test process runs the internal logic, then runs the scan test, and then runs the internal logic again. The test results verify that the IC logic runs correctly and that the scan test produces the expected results.

**Figure 10-19 VHDL Code for Interconnection Test Example**

```

-- Boundary Scan Tester

entity system is
end system;

architecture IC_test of system is
component BS_arch is
    generic (NCELLS:natural range 2 to 120 := 4);
    port (TCK, TMS, TDI: in bit;
          TDO: out bit;
          BSRin: in bit_vector(1 to NCELLS);
          BSRout: inout bit_vector(1 to NCELLS);
          CellType in bit_vector(1 to NCELLS));
          -- '0' for input cell, '1' for output cell
end component;

```

```

signal TCK,TMS,TDI,TDO,TDO1: bit;
signal Q0, Q1, CLK1: bit;
signal BSR1in, BSR1out, BSR2in, BSR2out: bit_vector(1 to 4);
signal count: integer := 0;

constant TMSpattern: bit_vector(0 to 62) :=
  "01100000001110000000001110000000111000000000111000000000111111";
constant TDIPattern: bit_vector(0 to 62) :=
  "0000010010000001000100000000000000010001000000000000000000000000";
begin
  BS1: BS_arch port map(TCK, TMS, TDI, TDO1, BSR1in, BSR1out, "0011");
  BS2: BS_arch port map(TCK, TMS, TDO1, TDO, BSR2in, BSR2out, "0011");
  -- each BSR has two input cells and two output cells

  BSR1in(1) <= BSR2out(4);                                -- IC1 external connections
  BSR1in(2) <= BSR2out(3);
  BSR1in(3) <= Q1;                                       -- IC1 internal logic
  BSR1in(4) <= Q0;
  CLK1 <= not CLK1 after 7 ns;                         -- internal clock
process(CLK1)
begin
  if (CLK1='1') then                                     -- D flip-flops
    Q0 <= BSR1out(1);
    Q1 <= BSR1out(2);
  end if;
end process;

  BSR2in(1) <= BSR1out(4);                                -- IC2 external connections
  BSR2in(2) <= BSR1out(3);
  BSR2in(3) <= BSR2out(1) xor BSR2out(2); -- IC2 internal logic
  BSR2in(4) <= not BSR2out(1);

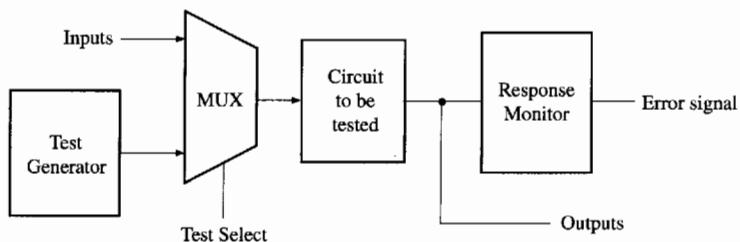
  TCK <= not TCK after 5 ns;                           -- test clock
process
begin
  TMS <= '1';
  wait for 70 ns;                                      -- run internal logic
  wait until TCK='1';
  for i in TMSpattern'range loop                      -- run scan test
    TMS <= TMSpattern(i);
    TDI <= TDIPattern(i);
    wait for 0 ns;
    count <= i+1;                                         -- count triggers listing output
    wait until TCK='1';
  end loop;
  wait for 70 ns;                                      -- run internal logic
  wait;                                               -- stop
end process;
end IC_test;

```

## 10.5 BUILT-IN SELF-TEST

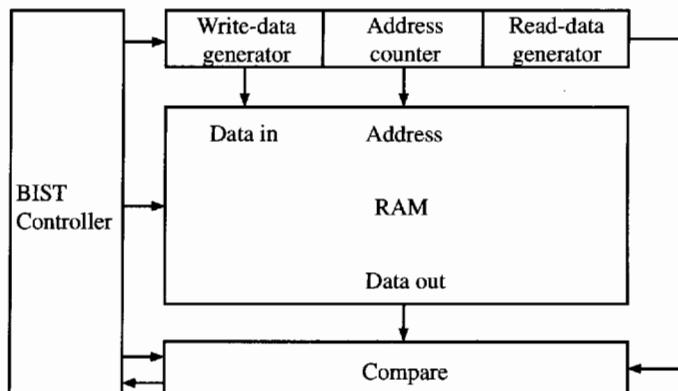
As digital systems become more and more complex, they become much harder and more expensive to test. One solution to this problem is to add logic to the IC so that it can test itself. This is referred to as Built-In Self-Test, or BIST. Figure 10-20 illustrates the general method for using BIST. When the test mode is selected by the test-select signal, an on-chip test generator applies test patterns to the circuit under test. The resulting output is observed by the response monitor, which produces an error signal if an incorrect output pattern is detected.

Figure 10-20 Generic BIST Scheme



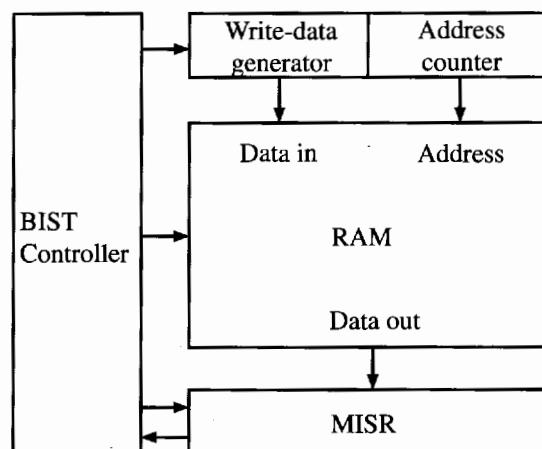
BIST is often used for testing memory. The regular structure of a memory chip makes it easy to generate test patterns. Figure 10-21 shows a block diagram of a self-test circuit for a RAM. The BIST controller enables the write-data generator and address counter so that data is written to each location in the RAM. Then the address counter and read-data generator are enabled, and the data read from each RAM location is compared with the output of the read-data generator to verify that it is correct.

Figure 10-21 Self-test Circuit for RAM



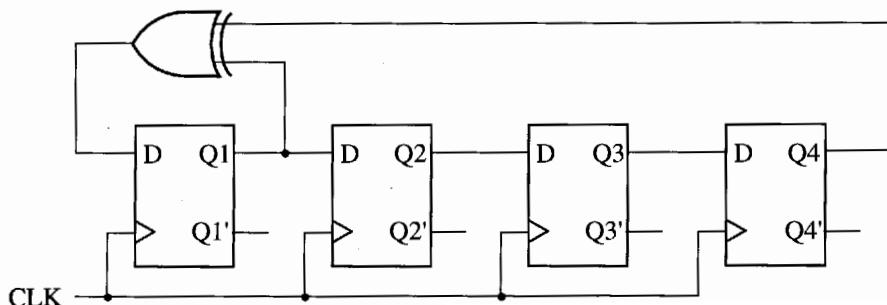
The test circuit can be simplified by using a signature register. The signature register compresses the output data into a short string of bits called a *signature*, and this signature is compared with the signature for a correctly functioning component. A multiple-input signature register (MISR) combines and compresses several output streams into a single signature. Figure 10-22 shows a simplified version of the RAM self-test circuit. The read-data generator and comparator have been eliminated and replaced with a MISR. One type of MISR simply forms a check sum by adding up all the data bytes stored in the RAM. When testing a ROM, Figure 10-22 can be simplified further, since no write-data generator is needed.

**Figure 10-22 Self-test Circuit for RAM with Signature Register**



Linear feedback shift registers (LFSRs) are often used to generate test patterns. Figure 10-23 shows an example of a LFSR. The outputs from the first and fourth flip-flops are XORed together and fed back into the *D* input of the first flip-flop. The general form of a LFSR is a shift register with two or more flip-flop outputs XORed together and fed back into the first flip-flop. The name *linear* comes from the fact that exclusive OR is equivalent to modulo-2 addition, and addition is a linear operation.

**Figure 10-23 4-bit Linear Feedback Shift Register (LFSR)**



By proper choice of the outputs that are fed back through the exclusive OR gate, it is possible to generate  $2^n - 1$  different bit patterns using an  $n$ -bit shift register. All possible patterns can be generated except for all 0s. The patterns generated by the LFSR of Figure 10-23 are:

1000, 1100, 1110, 1111, 0111, 1011, 0101, 1010, 1101, 0110, 0011,  
1001, 0100, 0010, 0001, 1000, ...

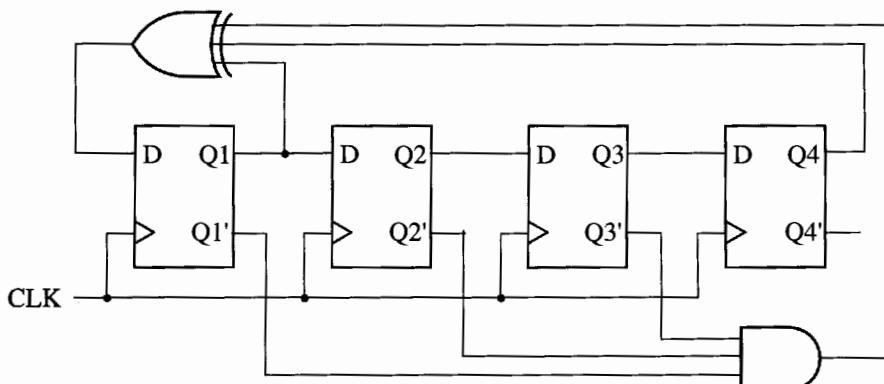
These patterns have no obvious order, and they have certain randomness properties. Such a LFSR is often referred to as a *pseudo-random pattern generator*, or PRPG. PRPGs are obviously very useful for BIST, since they can generate a large number of test patterns with a small amount of logic circuitry. Table 10-4 gives one feedback combination that will generate all  $2^n - 1$  bit patterns for some LFSRs with lengths in the range  $n = 4$  to 32.

**Table 10-4 Feedback for Maximum-length LFSR Sequence**

$n$	Feedback
4,6,7	$Q_1 \oplus Q_n$
5	$Q_1 \oplus Q_2 \oplus Q_5$
8	$Q_2 \oplus Q_3 \oplus Q_4 \oplus Q_8$
12	$Q_1 \oplus Q_4 \oplus Q_6 \oplus Q_{12}$
14,16	$Q_3 \oplus Q_4 \oplus Q_5 \oplus Q_n$
24	$Q_1 \oplus Q_2 \oplus Q_7 \oplus Q_{24}$
32	$Q_1 \oplus Q_2 \oplus Q_{22} \oplus Q_{32}$

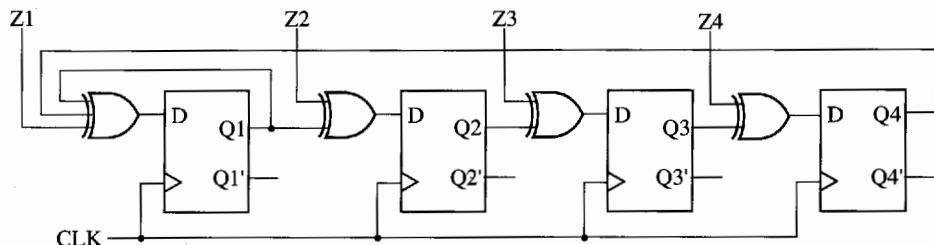
If the all-0s test pattern is required, an  $n$ -bit LFSR can be modified by adding an AND gate with  $n - 1$  inputs, as shown in Figure 10-24 for  $n = 4$ . When in state 0001, the next state is 0000; when in state 0000, the next state is 1000; otherwise, the sequence is the same.

**Figure 10-24 Modified LFSR with 0000 State**



A MISR can be constructed by modifying a LFSR by adding XOR gates, as shown in Figure 10-25. The test data ( $Z_1Z_2Z_3Z_4$ ) is XORed into the register with each clock, and the final result represents a signature that can be compared with the signature for a known correctly functioning component. This type of signature analysis will catch many, but not all, possible errors. An  $n$ -bit signature register maps all possible input streams into one of the  $2^n$  possible signatures. One of these is the correct signature, and the others indicate that errors have occurred. The probability that an incorrect input sequence will map to the correct signature is of the order of  $1/2^n$ .

**Figure 10-25 Multiple-input Signature Register (MISR)**



For the MISR of Figure 10-25, assume that the correct input sequence is 1010, 0001, 1110, 1111, 0100, 1011, 1001, 1000, 0101, 0110, 0011, 1101, 0111, 0010, 1100. This sequence maps to the signature 1010. Any sequence that differs in one bit will map to a different signature. For example, if 0001 in the sequence is changed to 1001, the resulting sequence maps to 1000. Most sequences with two errors will be detected, but if we change 0001 to 1001 and 0010 to 0110 in the original sequence, the result maps to 1010, which is the correct signature, so the errors would not be detected.

To adapt the scan test scheme of Figure 10-10(b) for BIST, the scan register is modified so each part of the register can serve as a state register, pattern generator, signature register, or shift register. When used as a shift register, the test data can be scanned in and out in the usual way. Then part of the scan register can be used as a pattern generator (PRPG) and part as a signature register (MISR) to test one of the combinational blocks. The roles can then be changed to test another combinational block. When the testing is finished, the scan register is placed in the state register mode for normal operation.

One such scheme is referred to as BILBO (built-in logic block observation). Figure 10-26 shows the placement of BILBO registers for testing a network with two combinational blocks. Comb. network 1 is tested when the first BILBO is used as a PRPG and the second as a MISR. The roles of the registers are reversed to test Comb. network 2. In the normal operating mode, both BILBOs serve as registers for the associated combinational logic. To scan data in and out, both BILBOs operate in the shift register mode.

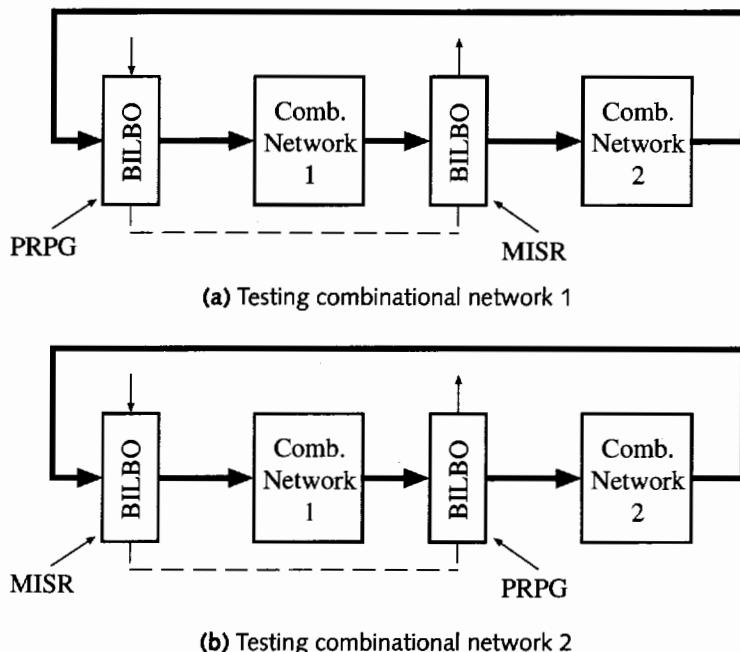
**Figure 10-26 BIST Using BILBO Registers**

Figure 10-27 shows the structure of one version of a 4-bit BILBO register. The control inputs  $B_1$  and  $B_2$  determine the operating mode.  $S_i$  and  $S_o$  are the serial input and output for the shift register mode. The  $Z$ s are inputs from the combinational logic. The equations for this BILBO register are

$$D_1 = Z_1 B_1 \oplus (S_i B_2' + FB B_2) (B_1' + B_2)$$

$$D_i = Z_i B_1 \oplus Q_{i-1} (B_1' + B_2) \quad (i > 1)$$

When  $B_1 = B_2 = 0$ , these equations reduce to

$$D_1 = S_i \text{ and } D_i = Q_{i-1} \quad (i > 1)$$

which corresponds to the shift register mode. When  $B_1 = 0$  and  $B_2 = 1$ , the equations reduce to

$$D_1 = FB, \quad D_i = Q_{i-1}$$

which corresponds to the PRPG mode, and the BILBO register is equivalent to Figure 10-23. When  $B_1 = 1$  and  $B_2 = 0$ , the equations reduce to

$$D_1 = Z_1, \quad D_i = Z_i$$

which corresponds to the normal operating mode. When  $B_1 = B_2 = 1$ , the equations reduce to

$$D_1 = Z_1 \oplus FB, \quad D_i = Z_i \oplus Q_{i-1}$$

which corresponds to the MISR mode, and the BILBO register is equivalent to Figure 10-25. In summary, the BILBO operating modes are as follows:

B1B2	Operating Mode
00	shift register
01	PRPG
10	normal
11	MISR

Figure 10-27 4-bit BILBO Register

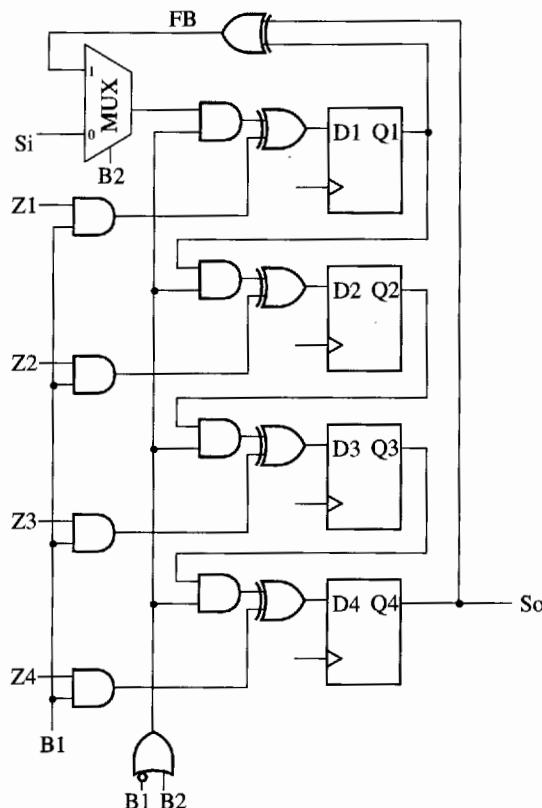


Figure 10-28 shows the VHDL description of an  $n$ -bit BILBO register. NBITS, which equals the number of bits, is a generic parameter in the range 4 through 8. The register is functionally equivalent to Figure 10-27, except that we have added a clock enable (CE). The feedback (FB) for the LFSR depends on the number of bits.

The system shown in Figure 10-29 illustrates the use of BILBO registers. In this system, registers A and B can be loaded from the Dbus using the LDA and LDB signals. Then the registers are added and the sum and carry are stored in register C. When  $B1B2 = 10$ , the registers are in the normal mode ( $Test = 0$ ), and loading of the registers is controlled by

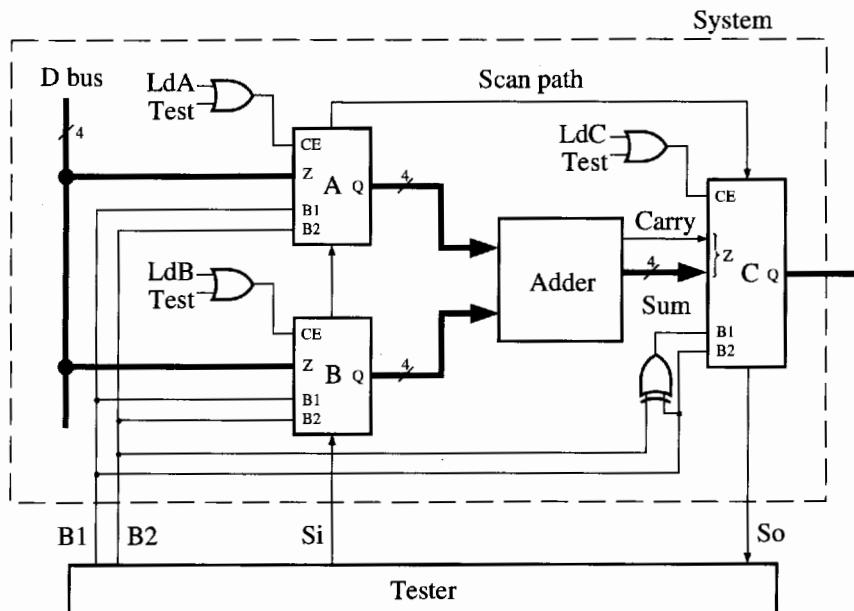
*LDA*, *LDB*, and *LDC*. To test the adder, we first set  $B1B2 = 00$  to place the registers in the shift register mode and scan in initial values for  $A$ ,  $B$ , and  $C$ . Then we set  $B1B2 = 10$ , which places registers  $A$  and  $B$  in PRPG mode and register  $C$  in MISR mode. After 15 clocks, the test is complete. Then we can set  $B1B2 = 00$  and scan out the signature.

Figure 10-28 VHDL Code for BILBO Register of Figure 10-27

```
entity BILBO is                                -- BILBO Register
  generic (NBITS: natural range 4 to 8 := 4);
  port (Clk, CE, B1, B2, Si: in bit;
        So: out bit;
        Z: in bit_vector(1 to NBITS);
        Q: inout bit_vector(1 to NBITS));
end BILBO;

architecture behavior of BILBO is
  signal FB: bit;
begin
  FB <= Q(2) xor Q(3) xor Q(4) xor Q(NBITS) when (NBITS=8)
    else Q(2) xor Q(NBITS) when (NBITS=5)
    else Q(1) xor Q(NBITS);
  process(Clk)
    variable mode: bit_vector(1 downto 0);
  begin
    if (Clk = '1' and CE = '1') then
      mode := B1 & B2;
      case mode is
        when "00" =>                      -- Shift register mode
          Q <= Si & Q(1 to NBITS-1);
        when "01" =>                      -- Pseudo Random Pattern Generator mode
          Q <= FB & Q(1 to NBITS-1);
        when "10" =>                      -- Normal Operating mode
          Q <= Z;
        when "11" =>                      -- Multiple Input Signature Register mode
          Q <= Z(1 to NBITS) xor (FB & Q(1 to NBITS-1));
      end case;
    end if;
  end process;
  So <= Q(NBITS);
end;
```

Figure 10-29 System with BILBO Registers and Tester



The VHDL code for the system is given in Figure 10-30, and a test bench is given in Figure 10-31. The system uses three BILBO registers and the 4-bit adder of Figure 8-12. The test bench scans in a test vector to initialize the BILBO registers, then it runs the test with registers A and B used as PRPGs and register C as a MISR. The resulting signature is shifted out and compared with the correct signature.

Figure 10-30 VHDL Code for System with BILBO Registers and Tester

```
entity BILBO_System is
  port (Clk, LdA, LdB, LdC, B1, B2, Si: in bit;
        So: out bit;
        DBus: in bit_vector(3 downto 0);
        Output: inout bit_vector(4 downto 0));
end BILBO_System;
```

```

architecture BSys1 of BILBO_System is
component Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;
        S: out bit_vector(3 downto 0); Co:out bit);
end component;
component BILBO is
  generic (NBITS: natural range 4 to 8 := 4);
  port (Clk, CE, B1, B2, Si : in bit;
        So: out bit;
        Z: in bit_vector(1 to NBITS);
        Q: inout bit_vector(1 to NBITS));
end component;

signal Aout, Bout: bit_vector(3 downto 0);
signal Cin: bit_vector(4 downto 0);
alias Carry: bit is Cin(4);
alias Sum: bit_vector is Cin(3 downto 0);
signal ACE, BCE, CCE, CB1, Test, S1, S2: bit;
begin
  Test <= not B1 or B2;
  ACE <= Test or LdA;
  BCE <= Test or LdB;
  CCE <= Test or LdC;
  CB1 <= B1 xor B2;
  RegA: BILBO generic map (4) port map(Clk, ACE, B1, B2, S1, S2, DBus,
                                         Aout);
  RegB: BILBO generic map (4) port map(Clk, BCE, B1, B2, Si, S1, DBus,
                                         Bout);
  RegC: BILBO generic map (5) port map(Clk, CCE, CB1, B2, S2, So, Cin,
                                         Output);
  Adder: Adder4 port map(Aout, Bout, '0', Sum, Carry);
end BSys1;

```

Figure 10-31 Test Bench for BILBO System

```

-- System with BILBO test bench

entity BILBO_test is
end BILBO_test;

architecture Btest of BILBO_test is
component BILBO_System is
  port (Clk, LdA, LdB, LdC, B1, B2, Si: in bit;
        So: out bit;
        DBus: in bit_vector(3 downto 0);
        Output: inout bit_vector(4 downto 0));
end component;

```

```
signal Clk: bit := '0';
signal LdA, LdB, LdC, B1, B2, Si, So: bit := '0';
signal DBus: bit_vector(3 downto 0);
signal Output: bit_vector(4 downto 0);
signal Sig: bit_vector(4 downto 0);

constant test_vector: bit_vector(12 downto 0) := "1000110000000";
constant test_result: bit_vector(4 downto 0) := "01011";
begin
  clk <= not clk after 10 ns;
  Sys: BILBO_System port map(Clk,Lda,LdB,LdC,B1,B2,Si,So,DBus,Output);
process
begin
  B1 <= '0'; B2 <= '0';           -- Shift in test vector
  for i in test_vector'right to test_vector'left loop
    Si <= test_vector(i);
    wait until (clk = '1');
  end loop;

  B1 <= '0'; B2 <= '1';           -- Use PRPG and MISR
  for i in 1 to 15 loop
    wait until (clk = '1');
  end loop;

  B1 <= '0'; B2 <= '0';           -- Shift signature out
  for i in 0 to 5 loop
    Sig <= So & Sig(4 downto 1);
    wait until (clk = '1');
  end loop;

  if (Sig = test_result) then      -- Compare signature
    report "System passed test.";
  else
    report "System did not pass test!";
  end if;

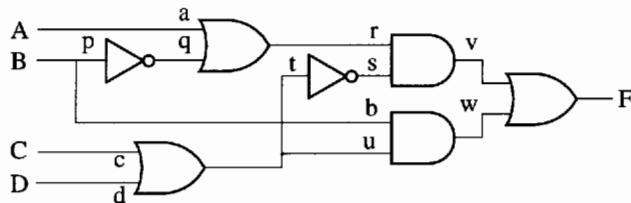
  wait;
end process;
end Btest;
```

In this chapter, we introduced the subject of testing hardware, including combinational networks, sequential networks, complex ICs, and PC boards. Use of scan techniques for testing and built-in self-test has become a necessity as digital systems have become more complex. It is very important that design for testability be considered early in the design process so that the final hardware can be tested efficiently and economically.

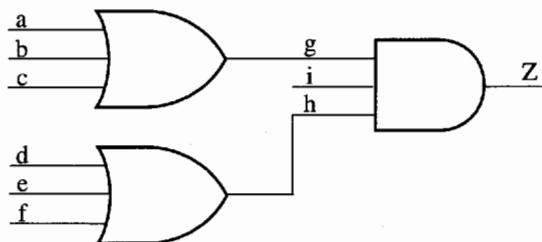
## Problems

### 10.1

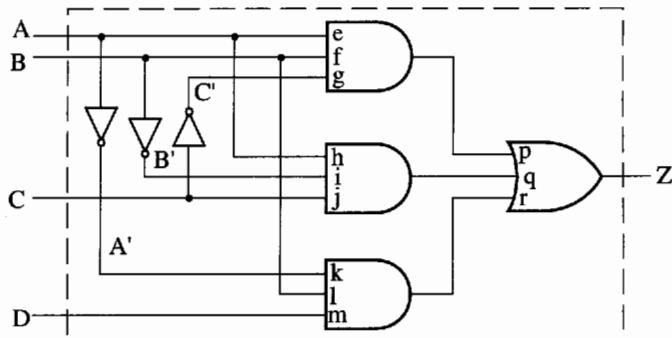
- (a) Determine the necessary inputs to the following network to test for  $u$  stuck-at-0.
- (b) For this set of inputs, determine which other stuck-at faults can be tested.
- (c) Repeat (a) and (b) for  $r$  stuck-at-1.



- 10.2** Find a minimum set of tests that will test all single stuck-at-0 and stuck-at-1 faults in the following network. For each test, specify which faults are tested for s-a-0 and for s-a-1.



- 10.3** For the following network, find a minimum number of test vectors that will test all s-a-0 and s-a-1 faults at the AND and OR gate inputs. For each test vector, specify the values of A, B, C and D, and the stuck-at faults that are tested.



- 10.4** Find a test sequence to test for  $b$  s-a-0 in the sequential network of Figure 10-7.

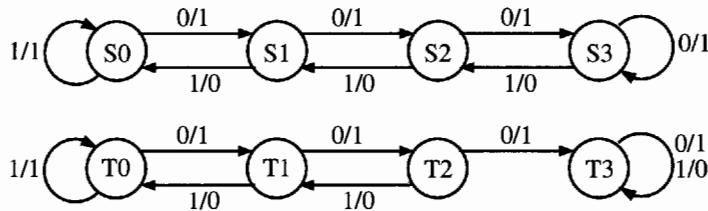
**10.5**

- (a) Redraw the code converter network of Figure 1-20 in the form of Figure 10-8 using dual-port flip-flops.
- (b) Determine a test sequence that will verify the first two rows of the transition table of Figure 1-18. Draw a timing diagram similar to Figure 10-9 for your test sequence.

**10.6**

- (a) Write VHDL code for a dual-port flip-flop.
- (b) Write VHDL code for your solution to Problem 10.5(a).
- (c) Write a test bench that applies the test sequence from Problem 10.5(b), and compare the resulting waveforms with your solution to Problem 10.5(b).

**10.7** State graphs for two sequential machines are given below. The first graph represents a correctly functioning machine, and the second represents the same machine with a malfunction. Assuming that the two machines can be reset to their starting states ( $S_0$  and  $T_0$ ), determine the shortest input sequence that will distinguish the two machines.



**10.8** Simulate the boundary scan tester of Figure 10-19 and verify that the results are as expected. Change the code to represent the case where the lower input to IC1 is shorted to ground, simulate again, and interpret the results.

**10.9** Write VHDL code for the boundary scan cell of Figure 10-14. Rewrite the VHDL code of Figure 10-18 to use this boundary scan cell as a component in place of some of the behavioral code for the BSR. Use a generate statement to instantiate NCELLS copies of this component. Test your new code using the boundary scan tester example of Figure 10-19.

**10.10**

- (a) Draw a schematic diagram for an LFSR with  $n = 5$  that generates a maximum length sequence.
- (b) Add logic so that 00000 is included in the state sequence.
- (c) Determine the actual state sequence.

**10.11**

- (a) Write VHDL for an 8-bit MISR that is similar to Figure 10-25.
- (b) Design a self-test circuit, similar to Figure 10-22, for a 6116 static RAM. The write-data generator should store data in the following sequence: 00000000, 10000000, 11000000, ..., 11111111, 01111111, 00111111, ..., 00000000.
- (c) Write VHDL code to test your design. Simulate the system for at least one example with no errors, 1 error, 2 errors, and 3 errors.

# CHAPTER 11

## DESIGN EXAMPLES

Up to this point, we have used simple design examples that illustrate how to use VHDL in the design process. In this chapter, we present some realistic design examples that show how VHDL, together with synthesis tools, can be used to design complex digital systems. We first design a receiver-transmitter for a serial data port, and then we design a simple microcontroller similar to the Motorola M68HC05.

### 11.1 UART DESIGN

Most computers and microcontrollers have one or more serial data ports used to communicate with serial input/output devices such as keyboards and serial printers. By using a modem (modulator-demodulator) connected to a serial port, serial data can be transmitted to and received from a remote location via telephone lines (see Figure 11-1). The serial communication interface, which receives and transmits serial data, is often called a UART (Universal Asynchronous Receiver-Transmitter). *RxD* is the received serial data signal and *TxD* is the transmitted data signal.

Figure 11-1 Serial Data Transmission

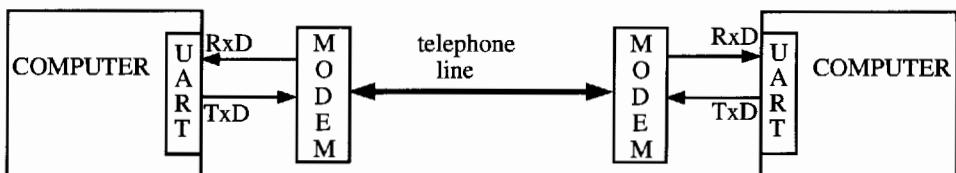
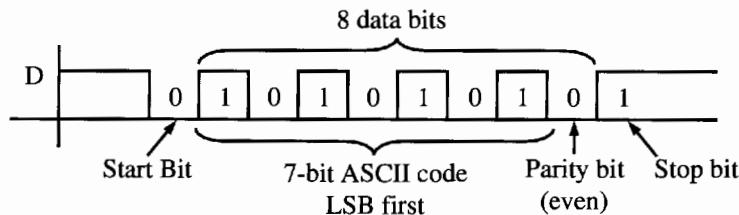


Figure 11-2 shows the standard format for serial data transmission. Since there is no clock line, the data (*D*) is transmitted asynchronously, one byte at a time. When no data is being transmitted, *D* remains high. To mark the start of transmission, *D* goes low for one bit time, which is referred to as the start bit. Then eight data bits are transmitted, least significant bit first. When text is being transmitted, ASCII code is usually used. In ASCII code, each alphanumeric character is represented by a 7-bit code. The eighth bit may be used as a parity check bit. In the example, the letter U, coded as 1010101, is transmitted followed by a 0 parity bit, so that the total number of 1s is even (even parity). After eight

bits are transmitted,  $D$  must go high for at least one bit time, which is referred to as the stop bit. Then transmission of another character can start at any time. The number of bits transmitted per second is frequently referred to as the *BAUD rate*.

**Figure 11-2 Standard Serial Data Format**

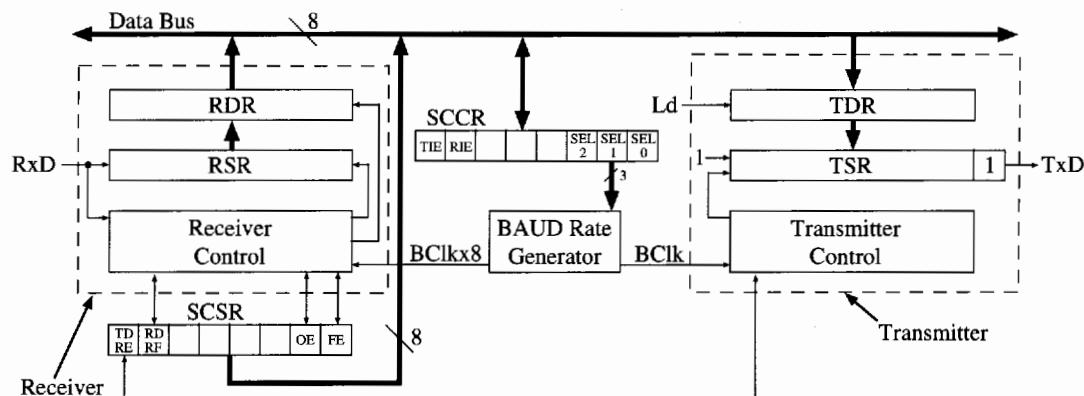


When transmitting, the UART takes eight bits of parallel data and converts the data to a serial bit stream that consists of a start bit (logic '0'), 8 data bits (least significant bit first), and one or more stop bits (logic '1'). When receiving, the UART detects the start bit, receives the 8 data bits, and converts the data to parallel form when it detects the stop bit. Since no clock is transmitted, the UART must synchronize the incoming bit stream with the local clock.

We now design a simplified version of a UART similar to the one used within the MC6805, MC6811, and other microcontrollers. Figure 11-3 shows the UART connected to the 8-bit data bus. The following six 8-bit registers are used:

<i>RSR</i>	Receive shift register
<i>RDR</i>	Receive data register
<i>TDR</i>	Transmit data register
<i>TSR</i>	Transmit shift register
<i>SCCR</i>	Serial communications control register
<i>SCSR</i>	Serial communications status register

The following discussion assumes that the UART is connected to a microcontroller data and address bus so that the CPU can read and write to the registers. *RDR*, *TDR*, *SCCR*, and *SCSR* are memory-mapped; that is, each register is assigned an address in the microcontroller memory space. *RDR*, *SCSR*, and *SCCR* can drive the data bus through tristate buffers; *TDR* and *SCCR* can be loaded from the data bus.

**Figure 11-3** UART Block Diagram

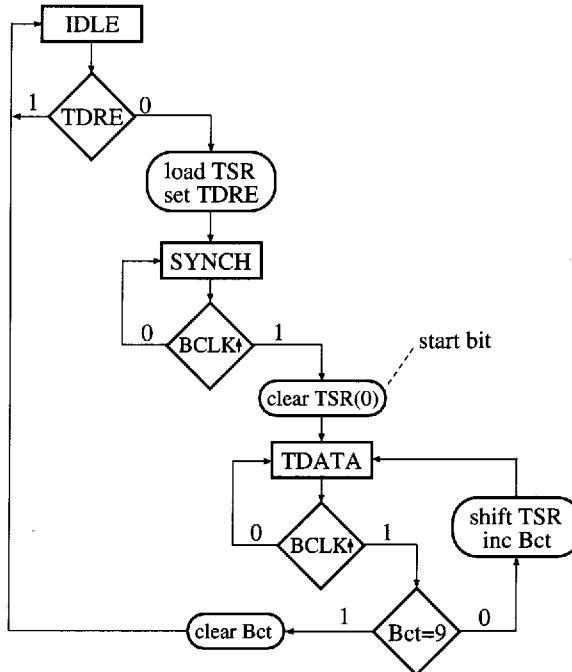
Beside the registers, the three main components of the UART are the BAUD rate generator, the receiver control, and the transmitter control. The BAUD rate generator divides down the system clock to provide the bit clock (*BClk*) with a period equal to one bit time and also *BClkX8*, which has a frequency eight times the *BClk* frequency.

The *TDRE* (transmit data register empty) bit in the *SCSR* is set when *TDR* is empty. When the microcontroller is ready to transmit data, the following occurs:

1. The microcontroller waits until *TDRE* = '1' and then loads a byte of data into *TDR* and clears *TDRE*.
2. The UART transfers data from *TDR* to *TSR* and sets *TDRE*.
3. The UART outputs a start bit ('0') for one bit time and then shifts *TSR* right to transmit the eight data bits followed by a stop bit ('1').

Figure 11-4 shows the SM chart for the transmitter. The corresponding sequential machine (SM) is clocked by the microcontroller system clock (*CLK*). In the IDLE state, the SM waits until *TDR* has been loaded and *TDRE* is cleared. In the SYNCH state, the SM waits for the rising edge of the bit clock (*Bclk*↑) and then clears the low-order bit of *TSR* to transmit a '0' for one bit time. In the TDATa state, each time *Bclk*↑ is detected, *TSR* is shifted right to transmit the next data bit and the bit counter (*Bct*) is incremented. When *Bct* = 9, 8 data bits and a stop bit have transmitted. *Bct* is then cleared and the SM goes back to IDLE.

Figure 11-4 SM Chart for UART Transmitter



The VHDL code for the UART transmitter (Figure 11-5) is based on the SM chart of Figure 11-4. The transmitter contains the *TDR* and *TSR* registers and the transmit control. It interfaces with *TDRE* and the data bus (*DBUS*). The first process represents the combinational network, which generates the nextstate and control signals. The second process updates the registers on the rising edge of the clock. The signal *Bclk\_rising* is '1' for one system clock time following the rising edge of *Bclk*. To generate *Bclk\_rising*, *Bclk* is stored in a flip-flop named *Bclk\_Dlayed*. Then *Bclk\_rising* is '1' if the current value of *Bclk* is '1' and the previous value (stored in *Bclk\_Dlayed*) is '0'. Thus,

```
Bclk_rising <= Bclk and not Bclk_Dlayed
```

Figure 11-5 VHDL Code for UART Transmitter

```

library ieee;
use ieee.std_logic_1164.all;

entity UART_Transmitter is
port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
      DBUS:in std_logic_vector(7 downto 0);
      setTDRE, TxD: out std_logic);
end UART_Transmitter;

architecture xmit of UART_Transmitter is

```

```

type stateType is (IDLE, SYNCH, TDATA);
signal state, nextstate : stateType;
signal TSR : std_logic_vector (7 downto 0); -- Transmit Shift Register
signal TDR : std_logic_vector(7 downto 0); -- Transmit Data Register
signal Bct: integer range 0 to 9; -- counts number of bits sent
signal inc, clr, loadTSR, shftTSR, start: std_logic;
signal Bclk_rising, Bclk_Dlayed: std_logic;

begin
TxD <= TSR(0);
setTDRE <= loadTSR;
Bclk_rising <= Bclk and (not Bclk_Dlayed);
-- indicates the rising edge of bit clock

Xmit_Control: process(state, TDRE, Bct, Bclk_rising)
begin
  inc <= '0'; clr <= '0'; loadTSR <= '0'; shftTSR <= '0'; start <= '0';
  -- reset control signals
  case state is
    when IDLE => if (TDRE = '0') then
      loadTSR <= '1'; nextstate <= SYNCH;
    else nextstate <= IDLE; end if;
    when SYNCH => -- synchronize with the bit clock
      if (Bclk_rising = '1') then
        start <= '1'; nextstate <= TDATA;
      else nextstate <= SYNCH; end if;
    when TDATA =>
      if (Bclk_rising = '0') then nextstate <= TDATA;
      elsif (Bct /= 9) then
        shftTSR <= '1'; inc <= '1'; nextstate <= TDATA;
      else clr <= '1'; nextstate <= IDLE; end if;
  end case;
end process;

Xmit_update: process (sysclk, rst_b)
begin
  if (rst_b = '0') then
    TSR <= "111111111"; state <= IDLE; Bct <= 0; Bclk_Dlayed <= '0';
  elsif (sysclk'event and sysclk = '1') then
    state <= nextstate;
    if (clr = '1') then Bct <= 0; elsif (inc = '1') then
      Bct <= Bct + 1; end if;
    if (loadTDR = '1') then TDR <= DBUS; end if;
    if (loadTSR = '1') then TSR <= TDR & '1'; end if;
    if (start = '1') then TSRout <= '0'; end if;
    if (shftTSR = '1') then TSR <= '1' & TSR(8 downto 1); end if;
    -- shift out one bit
    Bclk_Dlayed <= Bclk; -- Bclk delayed by 1 sysclk
  end if;
end process;
end xmit;

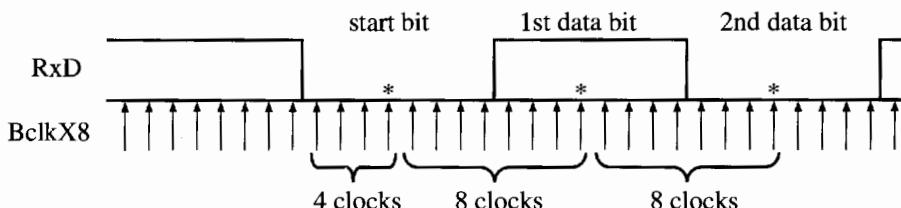
```

The operation of the UART receiver is as follows:

1. When the UART detects a start bit, it reads in the remaining bits serially and shifts them into the *RSR*.
2. When all the data bits and the stop bit have been received, the *RSR* is loaded into the *RDR*, and the Receive Data Register Full (*RDRF*) flag in the *SCSR* is set.
3. The microcontroller checks the *RDRF* flag, and if it is set, the *RDR* is read and the flag is cleared.

The bit stream coming in on *RxD* is not synchronized with the local bit clock (*Bclk*). If we attempted to read *RxD* at the rising edge of *Bclk* we would have a problem if *RxD* changed near the clock edge. We could have setup and hold time problems. If the bit rate of the incoming signal differed from *Bclk* by a small amount, we could end up reading some bits at the wrong time. To avoid these problems, we will sample *RxD* eight times during each bit time. (Some systems sample 16 times per bit.) We will sample on the rising edge of *BclkX8*. The arrows in Figure 11-6 indicate the rising edge of *BclkX8*. Ideally, we should read the bit value at the middle of each bit time for maximum reliability. When *RxD* first goes to 0, we will wait for four *BclkX8* periods, and we should be near the middle of the start bit. Then we will wait eight more *BclkX8* periods, which should take us near the middle of the first data bit. We continue reading once every eight *BclkX8* clocks until we have read the stop bit.

**Figure 11-6 Sampling RxD with BclkX8**



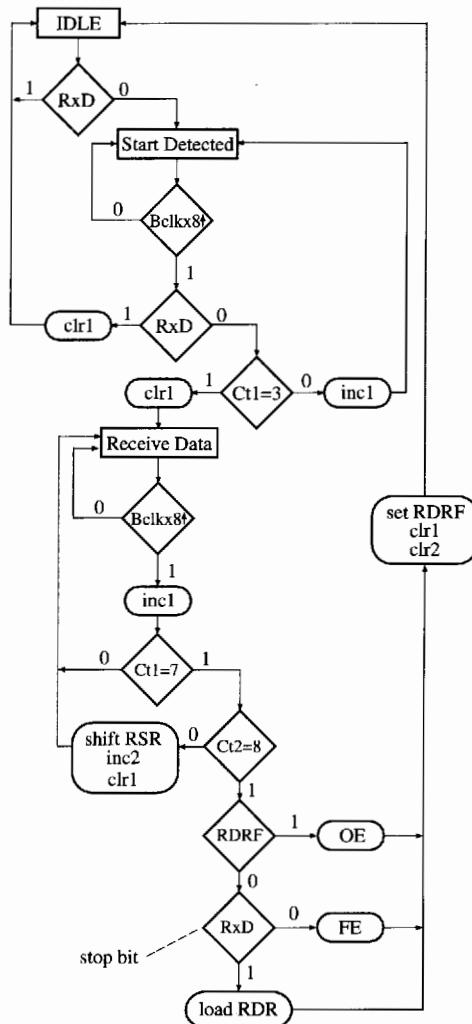
\* Read data at these points.

Figure 11-7 shows an SM chart for the UART receiver. Two counters are used. *Ct1* counts the number of *BclkX8* clocks. *Ct2* counts the number of bits received after the start bit. In the IDLE state, the SM waits for the start bit (*RxD* = '0') and then goes to the Start Detected state. The SM waits for the rising edge of *BclkX8* (*BclkX8* $\uparrow$ ) and then samples *RxD* again. Since the start bit should be '0' for eight *BclkX8* clocks, we should read '0'. *Ct1* is still 0, so *Ct1* is incremented and the SM waits for *BclkX8* $\uparrow$ . If *RxD* = '1', this is an error condition and the SM clears *Ct1* and resets to the IDLE state. Otherwise, the SM keeps looping. When *RxD* is '0' for the fourth time, *Ct1* = 3, so *Ct1* is cleared and the state goes to Receive Data. In this state, the SM increments *Ct1* after every rising edge of *BclkX8*. After the eighth clock, *Ct1* = 7 and *Ct2* is checked. If it is not 8, the current value of *RxD* is shifted into *RSR*, *Ct2* is incremented, and *Ct1* is cleared. If *Ct2* = 8, all 8 bits have been read and we should be in the middle of the stop bit. If *RDRF* = 1, the microcontroller has not yet read the previously received data byte, and an overrun error has occurred, in which

case the *OE* flag in the status register is set and the new data is ignored. If *RxD* = '0', the stop bit has not been detected properly, and the framing error (*FE*) flag in the status register is set. If no errors have occurred, *RDR* is loaded from *RSR*. In all cases, *RDRF* is set to indicate that the receive operation is completed and the counters are cleared.

The VHDL code for the UART receiver (Figure 11-8) is based on the SM chart of Figure 11-7. The receiver contains the *RDR* and *RSR* registers and the receive control. The control interfaces with *SCSR*, and *RDR* can drive data onto the data bus. The first process represents the combinational network, which generates the nextstate and control signals. The second process updates the registers on the rising edge of the clock. The signal *BclkX8\_rising* is '1' for one system clock time following the rising edge of *BclkX8*. *BclkX8\_rising* is generated the same manner as *Bclk\_rising*.

**Figure 11-7** SM Chart for UART Receiver



**Figure 11-8** VHDL Code for UART Receiver

```
library ieee;
use ieee.std_logic_1164.all;

entity UART_Receiver is
port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
      RDR: out std_logic_vector(7 downto 0);
      setRDRF, setOE, setFE: out std_logic);
end UART_Receiver;

architecture rcvr of UART_Receiver is

type stateType is (IDLE, START_DETECTED, RECV_DATA);
signal state, nextstate: stateType;

signal RSR: std_logic_vector (7 downto 0);      -- receive shift register
signal ct1 : integer range 0 to 7; -- indicates when to read the RxD input
signal ct2 : integer range 0 to 8;      -- counts number of bits read
signal inc1, inc2, clr1, clr2, shftRSR, loadRDR : std_logic;
signal BclkX8_Dlayed, BclkX8_rising : std_logic;

begin
BclkX8_rising <= BclkX8 and (not BclkX8_Dlayed);
-- indicates the rising edge of bitX8 clock
Rcvr_Control:    process(state, RxD, RDRF, ct1, ct2, BclkX8_rising)
begin
    -- reset control signals
    inc1 <= '0'; inc2 <= '0'; clr1 <= '0'; clr2 <= '0';
    shftRSR <= '0'; loadRDR <= '0'; setRDRF <= '0'; setOE <= '0';
    setFE <= '0';
    case state is
        when IDLE => if (RxD = '0') then nextstate <= START_DETECTED;
                      else nextstate <= IDLE; end if;
        when START_DETECTED =>
            if (BclkX8_rising = '0') then nextstate <= START_DETECTED;
            elsif (RxD = '1') then clr1 <= '1'; nextstate <= IDLE;
            elsif (ct1 = 3) then clr1 <= '1'; nextstate <= RECV_DATA;
            else inc1 <= '1'; nextstate <= START_DETECTED; end if;
        when RECV_DATA =>
            if (BclkX8_rising = '0') then nextstate <= RECV_DATA;
            else inc1 <= '1';
                if (ct1 /= 7) then nextstate <= RECV_DATA;
                -- wait for 8 clock cycles
            elsif (ct2 /= 8) then
                shftRSR <= '1'; inc2 <= '1'; clr1 <= '1'; -- read next data bit
                nextstate <= RECV_DATA;
```

```

    else
        nextstate <= IDLE;
        setRDRF <= '1'; clr1 <= '1'; clr2 <= '1';
        if (RDRF = '1') then setOE <= '1'; -- overrun error
        elsif (RxD = '0') then setFE <= '1'; -- framing error
        else loadRDR <= '1'; end if; -- load recv data register
    end if;
end if;
end case;
end process;

Rcvr_update: process (sysclk, rst_b)
begin
    if (rst_b = '0') then state <= IDLE; BclkX8_Delayed <= '0';
        ct1 <= 0; ct2 <= 0;
    elsif (sysclk'event and sysclk = '1') then
        state <= nextstate;
        if (clr1 = '1') then ct1 <= 0; elsif (inc1 = '1') then
            ct1 <= ct1 + 1; end if;
        if (clr2 = '1') then ct2 <= 0; elsif (inc2 = '1') then
            ct2 <= ct2 + 1; end if;
        if (shftRSR = '1') then RSR <= RxD & RSR(7 downto 1); end if;
            -- update shift reg.
        if (loadRDR = '1') then RDR <= RSR; end if;
        BclkX8_Delayed <= BclkX8; -- BclkX8 delayed by 1 sysclk
    end if;
end process;
end rcvr;

```

Figure 11-9 shows the result of synthesizing the UART receiver using the Xilinx 4000 series as a target. The resulting implementation requires 26 flip-flops and 18 function generators.

Next we will design a programmable BAUD rate generator. Three bits in the *SCCR* are used to select any one of eight BAUD rates. We will assume that the system clock is 8 MHz and we want BAUD rates 300, 600, 1200, 2400, 4800, 9600, 19200, and 38400. The maximum *BclkX8* frequency needed is  $38400 \times 8 = 307200$ . To get this frequency, we should divide 8 MHz by 26.04. Since we can divide only by an integer, we need to either accept a small error in the BAUD rate or adjust the system clock frequency downward to 7.9877 MHz to compensate.

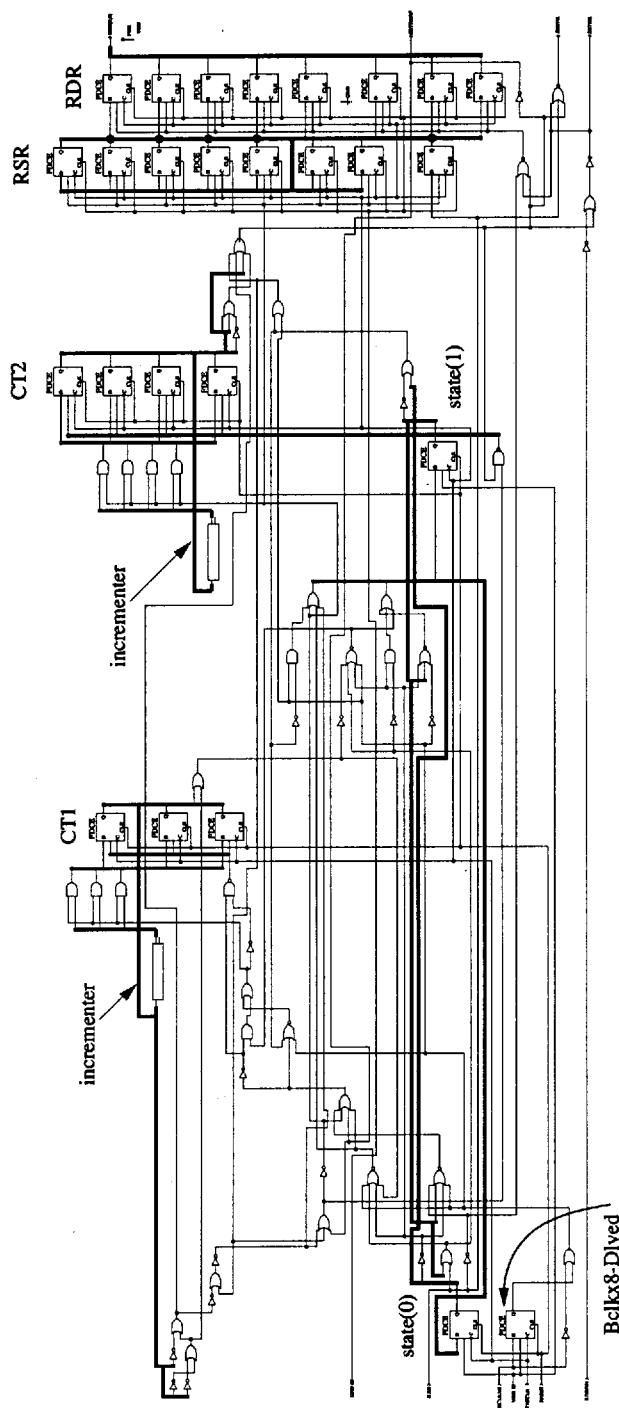
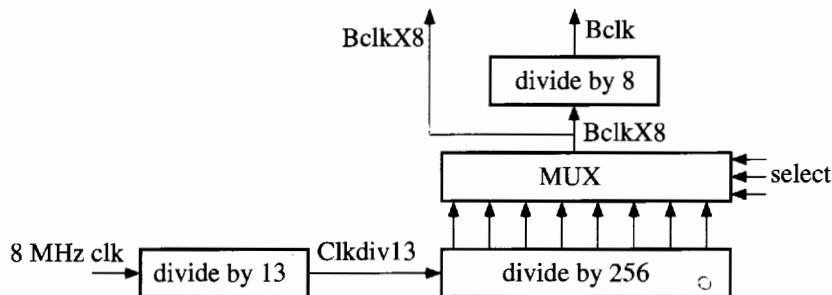


Figure 11-9 Synthesized UART Receiver

Figure 11-10 shows a block diagram for the BAUD rate generator. The 8-MHz system clock is first divided by 13 using a counter. This counter output goes to an 8-bit binary counter. The outputs of the flip-flops in this counter correspond to divide by 2, divide by 4, . . . , and divide by 256. One of these outputs is selected by a multiplexer. The MUX select inputs come from the lower 3 bits of the SCCR. The MUX output corresponds to  $BclkX8$ , which is further divided by 8 to give  $Bclk$ . Assuming an 8-MHz clock, the frequencies generated are given by the following table:

Select Bits	BAUD Rate ( $Bclk$ )
000	38462
001	19231
010	9615
011	4808
100	2404
101	1202
110	601
111	300.5

Figure 11-10 Baud Rate Generator



The VHDL code for the BAUD rate generator is given in Figure 11-11. The first process increments the divide-by-13 counter on the rising edge of the system clock. The second process increments the divide-by-256 counter on the rising edge of  $Clkdiv13$ . A concurrent statement generates the MUX output,  $BclkX8$ . The third process increments the divide-by-8 counter on the rising edge of  $BclkX8$  to generate  $Bclk$ .

**Figure 11-11 VHDL Code for BAUD Rate Generator**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; -- use '+' operator, CONVERT_INT func.

entity clk_divider is
port(Sysclk, rst_b: in std_logic;
      Sel: in std_logic_vector(2 downto 0);
      BclkX8: buffer std_logic;
      Bclk: out std_logic);
end clk_divider;

architecture baudgen of clk_divider is
signal ctr1: std_logic_vector (3 downto 0):= "0000"; -- divide by 13
                                                    counter
signal ctr2: std_logic_vector (7 downto 0):= "00000000"; -- div by 256 ctr
signal ctr3: std_logic_vector (2 downto 0):= "000"; -- divide by 8
                                                    counter
signal Clkdiv13: std_logic;

begin
process (Sysclk) -- first divide system clock by 13
begin
  if (Sysclk'event and Sysclk = '1') then
    if (ctr1 = "1100") then ctr1 <= "0000";
    else ctr1 <= ctr1 + 1; end if;
  end if;
end process;
Clkdiv13 <= ctr1(3); -- divide Sysclk by 13

process (Clkdiv13) -- clk_divdr is an 8-bit counter
begin
  if (rising_edge(Clkdiv13)) then
    ctr2 <= ctr2 + 1;
  end if;
end process;

BclkX8 <= ctr2(CONVERT_INTEGER(sel)); -- select baud rate
process (BclkX8)
begin
  if (rising_edge(BclkX8)) then
    ctr3 <= ctr3 + 1;
  end if;
end process;
Bclk <= ctr3(2); -- Bclk is BclkX8 divided by 8

end baudgen;
```

To complete the UART design, we need to interconnect the three components we have designed, connect them to the control and status registers, and add the interrupt generation logic and the bus interface. Figure 11-12 gives the VHDL code for the complete UART.

*SCI IRQ* is an interrupt signal that interrupts the CPU when the UART receiver or transmitter needs attention. When the *RIE* (receive interrupt enable) is set in *SCCR*, *SCI IRQ* is generated whenever *RDRE* or *OE* is '1'. When *TIE* (transmit interrupt enable) is set in *SCCR*, *SCI IRQ* is generated whenever *TDRE* is '1'.

The UART is interfaced to a microcontroller address and data busses so that the CPU can read and write to the UART registers when the UART is selected by *SCI sel* = '1'. The last two bits of the address (*ADDR2*), together with the *R\_W* signal, are used for register selection as follows:

<i>ADDR2</i>	<i>R_W</i>	Action
00	1	DBUS $\leftarrow$ RDR
00	0	TDR $\leftarrow$ DBUS
01	1	DBUS $\leftarrow$ SCSR
01	0	DBUS $\leftarrow$ hi-Z
1-	1	DBUS $\leftarrow$ SCCR
1-	0	SCCR $\leftarrow$ DBUS

When the UART is not selected for reading, the data bus is driven to high-Z.

Figure 11-12 VHDL Code for Complete UART

```

library ieee;
use ieee.std_logic_1164.all;

entity UART is
port(SCI_sel, R_W, clk, RXD : in std_logic;
      ADDR2: in std_logic_vector(1 downto 0);
      DBUS : inout std_logic_vector(7 downto 0);
      SCI_IRQ, TxD : out std_logic);
end UART;

architecture uart1 of UART is
component UART_Receiver
port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
      RDR: out std_logic_vector(7 downto 0);
      setRDRF, setOE, setFE: out std_logic);
end component;
component UART_Transmitter
port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
      DBUS: in std_logic_vector(7 downto 0);
      setTDRE, TxD: out std_logic);
end component;

```

```

component clk_divider
port(Sysclk, rst_b: in std_logic;
      Sel: in std_logic_vector(2 downto 0);
      BclkX8, Bclk: out std_logic);
end component;

signal RDR : std_logic_vector(7 downto 0);      -- Receive Data Register
signal SCSR : std_logic_vector(7 downto 0); -- Status Register
signal SCCR : std_logic_vector(7 downto 0); -- Control Register
signal TDRE, RDRF, OE, FE, TIE, RIE : std_logic;
signal BaudSel : std_logic_vector(2 downto 0);

signal setTDRE, setRDRF, setOE, setFE, loadTDR, loadSCCR : std_logic;
signal clrRDRF, Bclk, BclkX8, SCI_Read, SCI_Write : std_logic;

begin

RCVR: UART_Receiver port map(RxD, BclkX8, clk, rst_b, RDRF, RDR, "setRDRF,
                           setOE, setFE);
XMIT: UART_Transmitter port map(Bclk, clk, rst_b, TDRE, loadTDR, DBUS,
                           setTDRE, TxD);
CLKDIV: clk_divider port map(clk, rst_b, BaudSel, BclkX8, Bclk);

-- This process updates the control and status registers
process (clk, rst_b)
begin
  if (rst_b = '0') then
    TDRE <= '1'; RDRF <= '0'; OE<= '0'; FE <= '0';
    TIE <= '0'; RIE <= '0';
  elsif (rising_edge(clk)) then
    TDRE <= (setTDRE and not TDRE) or (not loadTDR and TDRE);
    RDRF <= (setRDRF and not RDRF) or (not clrRDRF and RDRF);
    OE <= (setOE and not OE) or (not clrRDRF and OE);
    FE <= (setFE and not FE) or (not clrRDRF and FE);
    if (loadSCCR = '1') then TIE <= DBUS(7); RIE <= DBUS(6);
      BaudSel <= DBUS(2 downto 0); end if;
  end if;
end process;

-- IRQ generation logic
SCI_IRQ <= '1' when ((RIE = '1' and (RDRF = '1' or OE = '1')) or (TIE = '1' and TDRE = '1'))
           else '0';
-- Bus Interface
SCSR <= TDRE & RDRF & "0000" & OE & FE;
SCCR <= TIE & RIE & "000" & BaudSel;
SCI_Read <= '1' when (SCI_sel = '1' and R_W = '0') else '0';
SCI_Write <= '1' when (SCI_sel = '1' and R_W = '1') else '0';

```

```
clrRDRF <= '1' when (SCI_Read = '1' and ADDR = "00") else '0';
loadTDR <= '1' when (SCI_Write = '1' and ADDR = "00") else '0';
loadSCCR <= '1' when (SCI_Write = '1' and ADDR = "10") else '0';
DBUS <= "ZZZZZZZZ" when (SCI_Read = '0')-- tristate bus when not reading
else RDR when (ADDR = "00") -- write appropriate register to the bus

else SCSR when (ADDR = "01")
else SCCR;           -- dbus = sccr, if addr is "10" or "11"
end uart1;
```

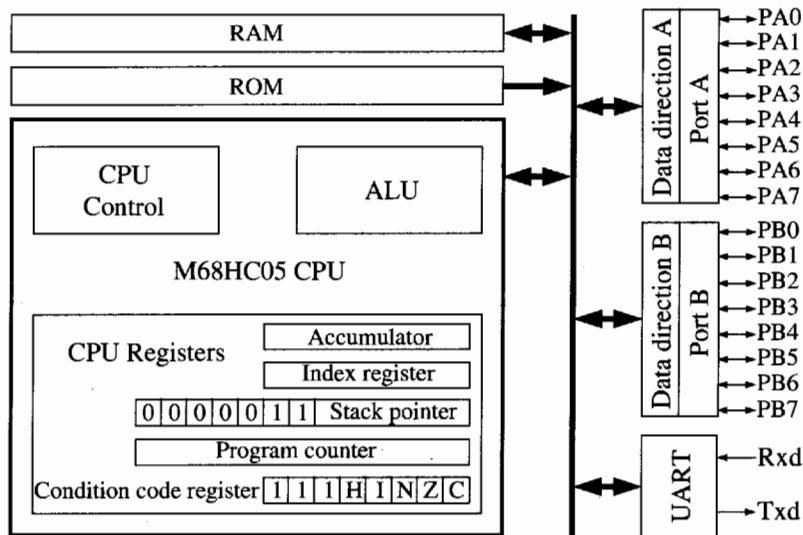
The VHDL code in Figure 11-12 was synthesized using the Xilinx 4000 series as a target. The resulting implementation required 90 FG function generators and 74 flip-flops and fits into an XC4003. When synthesized using the ALTERA 7000E series as a target, 120 logic cells and 74 flip-flops were required.

## 11.2 DESCRIPTION OF THE M68HC05 MICROCONTROLLER

A microcontroller typically contains a CPU, RAM, and ROM memory and various serial and parallel input-output interfaces, all on a single IC chip. As a final example, we will design a microcontroller similar to the Motorola MC68HC05. This type of microcontroller is widely used in simple control applications such as thermostats, appliance controllers, keyless entry systems, and so forth. These applications typically require much I/O capability and relatively little computational capability. Low cost is much more important than high speed. Motorola manufactures a whole family of 6805 microcontrollers, which differ mainly in the amount and type of memory and I/O capability.

A simplified version of the MC68HC05 microcontroller is shown in Figure 11-13. The block diagram shows the CPU core, RAM, ROM, two 8-bit parallel I/O ports (*Port A* and *Port B*), and a UART, which provides a serial communications interface. The actual 6805 has additional parallel and serial I/O ports and an on-chip timer. The chip has an internal address and data bus, which connects the CPU to the internal RAM and ROM and to the I/O interfaces. In the sections that follow, we design a CPU similar to the 6805 CPU and then integrate this CPU into the system shown in Figure 11-13.

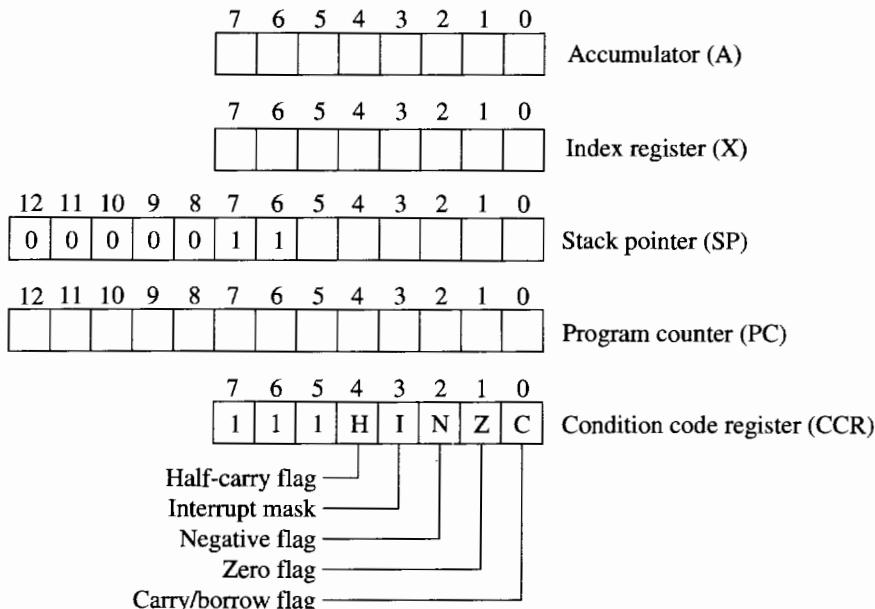
**Figure 11-13 Simplified Block Diagram for M68HC05 Microcontroller**



Next we describe the operation of the 6805 CPU from the programmer's point of view. We assume that the reader is familiar with assembly language programming for microprocessors, so we do not explain the terminology in detail. The 6805 data bus is 8 bits wide, and this version of the 6805 has a 13-bit address bus, so it is capable of addressing  $2^{13} = 8192$  bytes of memory.

Figure 11-14 shows the register structure for the MC68HC05C4 programming model. The accumulator (*A*), the index register (*X*), and the condition code register (*CCR*) are 8 bits long. The left 3 bits of the *CCR* are permanently set to 111 and the remaining bits are used as follows:

- *C* (carry flag) Stores the carry or borrow that results from an arithmetic operation.
- *H* (half-carry flag) Used for BCD arithmetic (not discussed in this text).
- *N* (negative flag) Set to 1 if the result of an operation is negative.
- *Z* (zero flag) Set to 1 if the result of an operation is 0.
- *I* (interrupt mask) When set to 1, prevents hardware interrupts from interrupting the processor.

**Figure 11-14 MC68HC05C4 Programming Model**

The program counter (*PC*), which is 13 bits long, addresses the instructions in memory as they are executed. In the 6805, a portion of the RAM memory is reserved for the stack. This stack is used for storing subroutine return addresses, and the *PC*, *A*, *X*, and *CCR* registers are pushed onto the stack when an interrupt is processed. The programmer has no direct access to the stack. The MC6805C4 stack always starts at address 000001111111 and grows downward in memory. The stack pointer (*SP*) is 6 bits wide, so the maximum stack size is 64 bytes. When addressing memory, *SP* is prefixed by 0000011 to give a 13-bit address. *SP* always points to the first empty location on the stack, and it is decremented after a byte is pushed onto the stack. Therefore, *SP* must be incremented before a byte is popped off the stack.

Each MC6805 instruction is from one to three bytes long. The first byte is always the opcode, which specifies the operation to be executed and the addressing mode. The next one or two bytes generally contain addressing information. Table 11-1 shows the mnemonics for the opcodes and corresponding hexadecimal codes. The four most significant bits of the opcode determine the addressing mode. The hex equivalents of these bits and the corresponding addressing modes are listed across the top of the table. The second four opcode bits, which are listed in hex on the side of the table, determine the operation to be performed. Thus opcode B4h (h indicates hexadecimal) specifies an AND operation with direct (dir) addressing, and FCh specifies a jump (JMP) instruction with indexed (ix) addressing.

Table 11-1 Opcodes for MC6805

		Bit Manipulation		Branch		Read/Modify/Write				Control		Register/Memory					
		BTB	BSC	REL	DIRM	INHA	INHX	IXIM	IXM	INHI	INH2	IMM	DIR	EXT	IX2	IX1	IX
Hi Lo	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRSET0*	BSET0*	BRA	NEG	NEG	NEG	NEG	NEG	RTI	SUB	SUB	SUB	SUB	SUB	SUB	SUB	
1	BRCLR0*	BCLR0*	BRN						RTS	CMP	CMP	CMP	CMP	CMP	CMP	CMP	
2	BRSET1*	BSET1*	BHI	MUL*						SBC	SBC	SBC	SBC	SBC	SBC	SBC	SBC
3	BRCLR1*	BCLR1*	BLS	COM	COM	COM	COM	COM	SWI	CPX	CPX	CPX	CPX	CPX	CPX	CPX	CPX
4	BRSET2*	BSET2*	BCC	LSR	LSR	LSR	LSR	LSR	LSR	AND	AND	AND	AND	AND	AND	AND	AND
5	BRCLR2*	BCLR2*	BCS							BIT	BIT	BIT	BIT	BIT	BIT	BIT	BIT
6	BRSET3*	BSET3*	BNE	ROR	ROR	ROR	ROR	ROR	ROR	LDA	LDA	LDA	LDA	LDA	LDA	LDA	LDA
7	BRCLR3*	BCLR3*	BEQ	ASR	ASR	ASR	ASR	ASR	TAX	STA	STA	STA	STA	STA	STA	STA	STA
8	BRSET4*	BSET4*	BHCC*	BSL	BSL	BSL	BSL	BSL	CLC	EOR	EOR	EOR	EOR	EOR	EOR	EOR	EOR
9	BRCLR4*	BCLR4*	BHCS*	ROL	ROL	ROL	ROL	ROL	SEC	ADC	ADC	ADC	ADC	ADC	ADC	ADC	ADC
A	BRSET5*	BSETS*	BPL	DEC	DEC	DEC	DEC	DEC	CLI	ORA	ORA	ORA	ORA	ORA	ORA	ORA	ORA
B	BRCLR5*	BCLR5*	BMI						SEI	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD
C	BRSET6*	BSET6*	BMC	INC	INC	INC	INC	INC	RSP	JMP	JMP	JMP	JMP	JMP	JMP	JMP	JMP
D	BRCLR6*	BCLR6*	BMS	TST	TST	TST	TST	TST	NOP	BSR*	JSR	JSR	JSR	JSR	JSR	JSR	JSR
E	BRSET7*	BSET7*	BIL*						STOP*	LDX	LDX	LDX	LDX	LDX	LDX	LDX	LDX
F	BRCLR7*	BCLR7*	BIH*	CLR	CLR	CLR	CLR	CLR	WAIT*	TXA	STX	STX	STX	STX	STX	STX	STX

\*not implemented in this text

Table 11-2 defines the operations performed by four groups of 6805 instructions. In this table,  $M$  represents data read from memory (or data to be written to memory), and  $R$  represents  $A$ ,  $X$ , or  $M$ .  $NZ$  in the last column indicates that the  $N$  and  $Z$  flags are updated, and the new value of  $C$  is given whenever  $C$  is updated. The effective address ( $EA$ ) is determined by the addressing mode. When the program counter is pushed onto the stack, it is divided into a high byte ( $PCH$ ) and a low byte ( $PCL$ ).

The branch instructions in Table 11-2(d) test a condition. If the condition is false, the next instruction in sequence is executed. If the condition is true, the next instruction is fetched from the branch address.

**Table 11-2 6805 Instructions****(a) Register-memory instructions**

Symbol	Instruction	Operation	Flags
ADD	add	$A \leftarrow A + M$	$NZ, C \leftarrow$ carry
ADC	add with carry	$A \leftarrow A + M + C$	$NZ, C \leftarrow$ carry
SUB	subtract	$A \leftarrow A - M$	$NZ, C \leftarrow$ borrow
SBC	subtract with borrow	$A \leftarrow A - M - C$	$NZ, C \leftarrow$ borrow
CMP	compare A	$A - M$	$NZ, C \leftarrow$ borrow
CPX	compare X	$X - M$	$NZ, C \leftarrow$ borrow
AND	and	$A \leftarrow A$ and $M$	$NZ$
BIT	bit test	$A$ and $M$	$NZ$
ORA	or	$A \leftarrow A$ or $M$	$NZ$
EOR	exclusive-or	$A \leftarrow A$ xor $M$	$NZ$
LDA	load A	$A \leftarrow M$	$NZ$
LDX	load X	$X \leftarrow M$	$NZ$
STA	store A	$M \leftarrow A$	$NZ$
STX	store X	$M \leftarrow X$	$NZ$
JMP	jump	jump to EA	
JSR	jump to subroutine	push PC on stack, jump to EA	

**(b) Read-modify-write instructions**

Symbol	Instruction	Operation	Flags
NEG	negate	$R \leftarrow 0 - R$	$NZ, C \leftarrow$ borrow
COM	complement	$R \leftarrow$ not $R$	$NZ, C \leftarrow 1$
TST	test	$R = 0$	$NZ$
CLR	clear	$R \leftarrow 0$	$NZ$
INC	increment	$R \leftarrow R + 1$	$NZ$
DEC	decrement	$R \leftarrow R - 1$	$NZ$
LSR	logical shift left	$R \leftarrow R(6 \text{ downto } 0) \& '0'$	$NZ, C \leftarrow R(7)$
ROL	rotate left	$R \leftarrow R(6 \text{ downto } 0) \& C$	$NZ, C \leftarrow R(7)$
ASR	arithmetic shift right	$R \leftarrow R(7) \& R(7 \text{ downto } 1)$	$NZ, C \leftarrow R(0)$
LSR	logical shift right	$R \leftarrow '0' \& R(7 \text{ downto } 1)$	$NZ, C \leftarrow R(0)$
ROR	rotate right	$R \leftarrow C \& R(7 \text{ downto } 1)$	$NZ, C \leftarrow R(0)$

## (c) Control instructions

Symbol	Instruction	Operation
TAX	transfer A to X	$X \leftarrow A$
TXA	transfer X to A	$A \leftarrow X$
CLC	clear carry	$C \leftarrow '0'$
SEC	set carry	$C \leftarrow '1'$
CLI	clear I	$I \leftarrow '0'$
SEI	set I	$I \leftarrow '1'$
RSP	reset SP	$SP \leftarrow "111111"$
NOP	no operation	
RTI	return from interrupt	pop CCR, A, X, PCH, PCL return to address in PC
RTS	return from subroutine	pop PCH, PCL return to address in PC
SWI	software interrupt	push PCL, PCH, X, A, CCR jump to address from interrupt vector table

## (d) Branch instructions

Symbol	Instruction	Branch if
BRA	branch always	always
BRN	branch never	never
BHI	branch if higher	$(C \text{ or } Z) = '0'$
BLS	branch if lower or same	$(C \text{ or } Z) = '1'$
BCC	branch if carry clear	$C = '0'$
BCS	branch if carry set	$C = '1'$
BNE	branch if not equal	$Z = '0'$
BEQ	branch if equal	$Z = '1'$
BPL	branch if plus	$N = '0'$
BMI	branch if minus	$N = '1'$
BMC	branch if int. mask clear	$I = '0'$
BMS	branch if int. mask set	$I = '1'$

Each 6805 instruction is one to three bytes long. The first byte is always the opcode, and any remaining bytes contain an address or an offset. The address of the data (or of the next instruction for jumps) is called the *effective address*, or *EA*. The 6805 has the following addressing modes for register-memory instructions:

- immediate (imm): data is in byte 2 of the instruction.
- direct (dir): *EA* is byte 2 of instruction.
- extended (ext): *EA* is bytes 2 and 3 of instruction (high byte first).
- indexed, no offset (ix): *EA* is in *X*.
- indexed, 1-byte offset (ix1): offset is in byte 2 of instructions;  $EA = X + \text{offset}$ .
- indexed, 2-byte offset (ix2): offset is bytes 2 and 3 of instructions;  
 $EA = X + \text{offset}$ .

Read-modify-write instructions (Table 11-1(b)) use direct (dirm), indexed with no offset (ixm), indexed with 1-byte offset (ix1m), and inherent (inha or inhx) addressing modes. For modes inha and inhx, the data is in the *A* and *X* registers, respectively. The dirm, ixm, and ix1m modes work the same way as dir, ix, and ix1 modes; however, we have given

them different names, since they apply to different groups of instructions. Control instructions use inherent addressing (inh1 or inh2), since the 1-byte opcode implies the address of the operands. Branch instructions use relative addressing. The opcode is at the original *PC* address, and the second byte of the instruction is a relative address, which is sign-extended and added to the original *PC* + 2 to get the branch address. Table 11-3 summarizes the 6805 addressing modes. The length of the instruction is given in the bytes column.

**Table 11-3 6805 Addressing Modes**

Mode	Name	Bytes	Examples	Effective Address
imm	immediate	2	ADD ii	data = byte 2 of instructions
dir, dirm	direct	2	ADD dd INC dd	EA = dd
ext	extended	3	ADD hh ll	EA = hh ll
ix ixm	indexed, no offset	1	ADD ,X INC ,X	EA = X
ix1, ix1m	indexed, 1-byte offset	2	ADD ff,X INC ff,X	EA = ff + X
ix2	indexed, 2-byte offset	3	ADD ee ff, X	EA = (ee ff) + X
rel	relative	2	BRA rr	EA = PC + 2 + rr*
inha, inhx	inherent	1	INCA INCX	data is in A data is in X
inh1, inh2	inherent	1	RTI TAX	opcode implies location of operands

\*rr is sign-extended before addition.

When a hardware interrupt, software interrupt, or reset occurs, the 6805 uses an interrupt vector table, which is stored at the high end of memory, to find the starting addresses of the corresponding interrupt subroutines. The starting addresses are stored at the following hex locations, with the high byte stored first:

reset 1FFEh, 1FFFh

software interrupt (SWI) 1FFCh, 1FFDh

ext. hardware int. (IRQ) 1FFAh, 1FFBh

serial comm. int. (SCI\_int) 1FF6h, 1FF7h

When an *IRQ* interrupt occurs, *PCL*, *PCH*, *X*, *A*, and *CCR* are pushed onto the stack. Then the *PC* is loaded with the address stored at 1FFAh and 1FFBh, which causes the processor to jump to the *IRQ* interrupt subroutine.

### 11.3 DESIGN OF A MICROCONTROLLER CPU

In this section, we design the CPU for a microcontroller that is similar to the 6805 CPU. We have omitted several of the 6805 instructions given in Table 11-1 to reduce the complexity of this example. For a few of the instructions, the timing in our design is different than that of the Motorola 6805. Starting with the specifications for the CPU given in Section 11-2, we determine the cycle-by-cycle operations necessary to implement different types of instructions and addressing modes. Then we write behavioral level VHDL code and verify that our design meets specifications. After constructing block diagrams for the CPU, we rewrite the VHDL code in terms of register transfers and control signals. After simulating the CPU, we synthesize it from the VHDL code to fit into an FPGA and a CPLD.

In addition to the registers used in programmer's model (Figure 11-14), three additional registers are needed to implement the CPU. An 8-bit register (*Opcode*) is needed to hold the opcode while an instruction is executing. A 13-bit memory address register (*MAR*) is needed to hold the effective address of the data that is to be read from or written to memory. An 8-bit memory data register (*Md*) is needed to hold the data after it is read from memory.

The next step in designing the CPU is to determine what actions should take place during each clock cycle. In a simple processor like the 6805, the internal clock period is the same as the memory cycle time. That is, during one clock cycle we can read or write a byte to memory, or we can complete an internal CPU operation such as addition. Each instruction takes from two to ten clock cycles to execute, depending on its complexity. The first cycle of every instruction is used to fetch the opcode from memory. At the start of the cycle, the program counter is pointing to the first byte of an instruction in memory, which is the opcode. The *PC* goes out on the address bus, and the memory returns the opcode on the data bus. At the end of the cycle, the opcode is loaded into the opcode register and the program counter is incremented. We designate these actions as follows:

$$\text{Opcode} \leftarrow \text{mem}(\text{PC});$$

$$\text{PC} \leftarrow \text{PC} + 1;$$

where *mem* is an array of bytes that represents the memory.

Table 11-4 shows the actions that should take place during each clock cycle for typical instructions. In this table, *MARH* is the high byte of *MAR*, and *MARL* is the low byte. The names in braces at the top of each box represent state names, which are discussed later. The ADD instruction is typical of all register-memory instructions. Thus, the SUB instruction is identical to the ADD instruction, except for the action that occurs during the final clock cycle. Similarly, the cycle-by-cycle timing for the INC instruction is the same for all other read-modify-write instructions. The first cycle of each instruction, which is always fetch the opcode, has been omitted from the table. During the second cycle, short instructions that require no further memory operation complete execution. For example, INCA adds 1 to the *A* register. Those instructions that require a second byte from memory read the byte and either load it into *Md* (memory data register) or *MAR* (memory address register). Any time a byte is read from memory, the *PC* is incremented so that it points to the next instruction byte. The second cycle for ADD imm is

$Md \leftarrow mem(PC); \quad \text{-- get immediate data from memory}$ 
 $PC \leftarrow PC + 1;$ 

During the last cycle of every ADD instruction, the data from memory is added to A. This addition is actually completed during the fetch cycle of the next instruction, so ADD imm requires only two clock cycles to execute. This overlap of the ALU operation and opcode fetch is possible because the old opcode is still in the register during the fetch cycle, and the new opcode is not loaded until the end of the cycle.

**Table 11-4 Cycle-by-Cycle Operations for 6805 Instructions**

1st cycle is {fetch} for all instructions

	2nd cycle	3rd cycle	4th cycle	5th cycle	6th cycle
ADD imm	{addr1} Md $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	(A $\leftarrow$ A + Md)*			
ADD dir	{addr1} MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{data} Md $\leftarrow$ mem(MAR)	(A $\leftarrow$ A + Md)*		
ADD ix	{addr1} MARL $\leftarrow$ X	{data} Md $\leftarrow$ mem(MAR)	(A $\leftarrow$ A + Md)*		
ADD ix1	{addr1} MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{addx} MARL $\leftarrow$ MAR + X	{data} Md $\leftarrow$ mem(MAR)	(A $\leftarrow$ A + Md)*	
ADD ext	{addr1} MARH $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{addr2} MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{data} Md $\leftarrow$ mem(MAR)	(A $\leftarrow$ A + Md)*	
ADD ix2	{addr1} MARH $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{addr2} MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{addx} MARL $\leftarrow$ MAR + X	{data} Md $\leftarrow$ mem(MAR)	(A $\leftarrow$ A + Md)*
STA ext	{addr1} MARH $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{addr2} MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{data} mem(MAR) $\leftarrow$ A		
INC A	{addr1} A $\leftarrow$ A + 1				
INC dir	{addr1} MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{data} Md $\leftarrow$ mem(MAR)	{rd_mod_wr} Md $\leftarrow$ Md + 1	{writeback} mem(MAR) $\leftarrow$ Md	
INC ix	{addr1} MARL $\leftarrow$ X	{data} Md $\leftarrow$ mem(MAR)	{rd_mod_wr} Md $\leftarrow$ Md + 1	{writeback} mem(MAR) $\leftarrow$ Md	
INC ix1	{addr1} MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	{addx} MARL $\leftarrow$ MAR + X	{data} Md $\leftarrow$ mem(MAR)	{rd_mod_wr} Md $\leftarrow$ Md + 1	{writeback} mem(MAR) $\leftarrow$ Md
JMP dir	{addr1} PCL $\leftarrow$ mem(PC)				
JMP ix	{addr1} PCL $\leftarrow$ X				

JMP ix1	<b>{addr1}</b> MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{addrx}</b> PC $\leftarrow$ MAR + X			
JMP ext	<b>{addr1}</b> MARH $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{addr2}</b> PCL $\leftarrow$ mem(PC) PCH $\leftarrow$ MARH			
JMP ix2	<b>{addr1}</b> MARH $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{addr2}</b> MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{addrx}</b> PC $\leftarrow$ MAR + X		
JSR dir	<b>{addr1}</b> MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{push1}</b> mem(SP) $\leftarrow$ PCL SP $\leftarrow$ SP - 1	<b>{push2}</b> mem(SP) $\leftarrow$ PCH SP $\leftarrow$ SP - 1 PC $\leftarrow$ MAR		
JSR ix	<b>{addr1}</b> MARL $\leftarrow$ X	<b>{push1}</b> mem(SP) $\leftarrow$ PCL SP $\leftarrow$ SP - 1	<b>{push2}</b> mem(SP) $\leftarrow$ PCH SP $\leftarrow$ SP - 1 PC $\leftarrow$ MAR		
JSR ix1	<b>{addr1}</b> MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{addrx}</b> MAR $\leftarrow$ MAR + X	<b>{push1}</b> mem(SP) $\leftarrow$ PCL SP $\leftarrow$ SP - 1	<b>{push2}</b> mem(SP) $\leftarrow$ PCH SP $\leftarrow$ SP - 1 PC $\leftarrow$ MAR	
JSR ext	<b>{addr1}</b> MARH $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{addr2}</b> MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{push1}</b> mem(SP) $\leftarrow$ PCL P $\leftarrow$ SP - 1	<b>{push2}</b> mem(SP) $\leftarrow$ PCH SP $\leftarrow$ SP - 1 PC $\leftarrow$ MAR	
JSR ix2	<b>{addr1}</b> MARH $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{addr2}</b> MARL $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{addrx}</b> MAR $\leftarrow$ MAR + X	<b>{push1}</b> mem(SP) $\leftarrow$ PCL SP $\leftarrow$ SP - 1	<b>{push2}</b> mem(SP) $\leftarrow$ PCH SP $\leftarrow$ SP - 1 PC $\leftarrow$ MAR
RTS	<b>{addr1}</b> SP $\leftarrow$ SP + 1	<b>{pop2}</b> PCH $\leftarrow$ mem(SP) SP $\leftarrow$ SP + 1	<b>{pop1}</b> PCL $\leftarrow$ mem(SP)		
BRA rel	<b>{addr1}</b> Md $\leftarrow$ mem(PC) PC $\leftarrow$ PC + 1	<b>{BRtest}</b> PC $\leftarrow$ PC + sign_ext&Md			
SWI	<b>{addr1}</b> no action	<b>{push1}</b> mem(SP) $\leftarrow$ PCL SP $\leftarrow$ SP - 1	<b>{push2}</b> mem(SP) $\leftarrow$ PCH SP $\leftarrow$ SP - 1	<b>{push3}</b> mem(SP) $\leftarrow$ X SP $\leftarrow$ SP - 1	<b>{push4}</b> mem(SP) $\leftarrow$ A SP $\leftarrow$ SP - 1
SWI continued	(7th cycle) <b>{push5}</b> mem(SP) $\leftarrow$ CCR SP $\leftarrow$ SP - 1	(8th cycle) <b>{cycle8}</b> MAR $\leftarrow$ vector addr. 1 $\leftarrow$ 1	(9th cycle) <b>{cycle9}</b> PCH $\leftarrow$ mem(MAR) MAR $\leftarrow$ MAR + 1	(10th cycle) <b>{cycle10}</b> PCL $\leftarrow$ mem(MAR)	
RTI	<b>{addr1}</b> SP $\leftarrow$ SP + 1	<b>{pop5}</b> CCR $\leftarrow$ mem(SP) SP $\leftarrow$ SP + 1	<b>{pop4}</b> A $\leftarrow$ mem(SP) SP $\leftarrow$ SP + 1	<b>{pop3}</b> X $\leftarrow$ mem(SP) SP $\leftarrow$ SP + 1	6th and 7th cycles same as 3rd and 4th cycles of RTS

\*Completion of ALU operation overlaps fetch of next instruction.

For instructions with direct addressing, such as ADD dir, the direct address is read from memory and loaded into *MAR* during the second cycle:

$$MARL \leftarrow mem(PC); \quad \text{Get direct address from memory.}$$

and *MARH* is also cleared. During the third cycle, the address in *MAR* goes out on the address bus, and the data from memory is loaded into *Md*:

$$Md \leftarrow mem(MAR);$$

For extended addressing (such as ADD ext), the high byte of the data address is read during cycle 2, the low byte is read during cycle 3, and the data is read during cycle 4. For indexed addressing with no offset (such as ADD ix), *X* is loaded into *MAR* at the end of cycle 2. Indexed addressing with a one-byte or two-byte offset is similar to direct or extended addressing with an extra cycle required to add *X* to *MAR*.

The STA and STX instructions are similar to the ADD instructions with the corresponding addressing modes, except during the last cycle data is stored in memory rather than being read from memory.

The read-modify-write-to-memory instructions, such as INC dirm, INC ixm, and INC ix1m, start out the same way as the corresponding ADD instructions. However, instead of doing the ALU operation during the next fetch cycle, two extra cycles must be added for the INC instruction. During the first added cycle, the INC operation is completed, and during the second added cycle, the result is written back to memory.

The JMP instructions are similar to the ADD instructions with the corresponding addressing modes. For JMP, during the cycle in which the address calculations are completed, the address is loaded into the *PC* instead of the *MAR*. For example, for JMP ext, the following action occurs during the 3rd cycle:

$$PC \leftarrow MARH \& mem(PC);$$

The low byte of the jump address is read from memory, and the high byte is already in *MARH*. Thus the two bytes of the jump address are loaded into the *PC*, and the next instruction is fetched from the jump address.

The JSR instructions are similar to JMP, except the return address must be pushed onto the stack. After the jump address has been determined and loaded into *MAR*, the low and high bytes of the *PC* are pushed onto the stack using two cycles:

$$mem(SP) \leftarrow PCL;$$

$$SP \leftarrow SP - 1;$$

$$mem(SP) \leftarrow PCH;$$

$$SP \leftarrow SP - 1;$$

Initially, *SP* is pointing to the first empty location on the stack, so *PCL* is written to that location. At the end of the cycle, the *SP* is decremented in preparation for the next push. At the end of the second push cycle, *MAR* (which contains the jump address) is loaded into *PC* to accomplish the jump.

For RTS,  $SP$  is incremented during the 2nd cycle. During the 3rd cycle, the high byte of the return address is read from memory and loaded into  $PCH$ . During the 4th cycle, the low byte of the return address is read and loaded into  $PCL$ . In the fetch cycle that follows, the opcode is read from the return address.

For software interrupt (SWI), no action occurs during the 2nd cycle, except to check the opcode and go to the proper next state to begin a sequence of stack operations.  $PCL$ ,  $PCH$ ,  $X$ ,  $A$ , and  $CCR$  are pushed onto the stack in the 3rd through 7th cycles. The interrupt vector address is loaded into  $MAR$  during the 8th cycle. The interrupt vector high and low bytes are loaded into the  $PC$  during the 9th and 10th cycles in preparation to jumping to the interrupt subroutine. The return from interrupt instruction (RTI) is similar to RTS except  $CCR$ ,  $A$ , and  $X$  are popped off the stack before the return address is popped. The sequence of actions for a hardware interrupt is very similar to SWI except that the fetch cycle and 2nd cycle are skipped.

The following example illustrates the sequence of operations for an ADD instruction with direct addressing followed by STA with direct addressing. Initially, assume the following is stored in memory starting at address 300h: BBh 43h B7h 57h 4Ch. Also assume that  $PC = 300h$ ,  $A = 12h$ , and memory location 0043h contains 36h. Table 11-5 shows the sequence of operations that occur in successive memory cycles, with all data in hex. The relevant registers are updated at the end of each cycle. For example, during the first cycle the opcode BBh is read from location 0300h, and at the end of the cycle  $PC$  is incremented and BBh is loaded into the opcode register.

**Table 11-5 Example of Cycle-by-Cycle Instruction Execution**

$PC$	$MAR$	Address Bus	Data Bus	Opcode Reg.	$A$	$Md$	
0300	xxxx	0300	BB	xx	12	xx	fetch opcode
0301	xxxx	0301	43	BB	12	xx	get direct address
0302	0043	0043	36	BB	12	xx	read memory data
0302	0043	0302	B7	BB	12	36	do addition, fetch next opcode
0303	0043	0303	42	B7	48	36	get direct address
0304	0057	0057	48	B7	48	36	store data in memory
0305	0057	0305	4C	B7	48	36	fetch next opcode

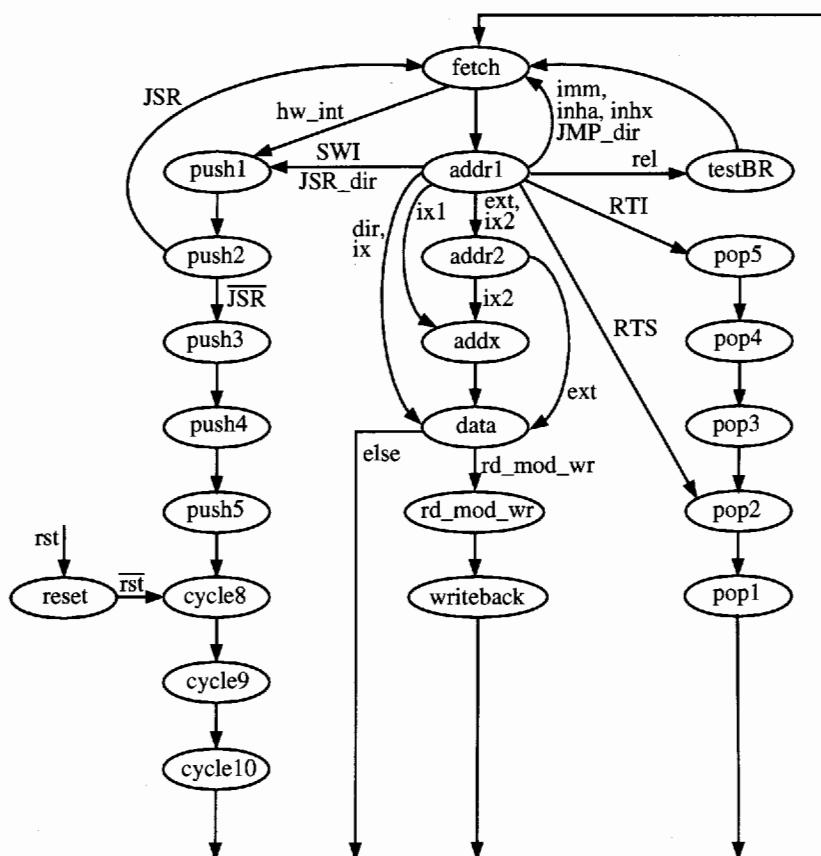
## Design of the CPU Controller

Once the cycle-by-cycle timing chart for executing instructions has been determined, we can design a state machine to control the CPU. We need to associate states with the various actions listed in Table 11-4. We have placed state names in braces at the top of each box in the table. We could have used one state for each of the clock cycles listed at the top of the chart. If we did that, we would have to test the opcode in each state to determine the required actions. A better approach is to define states so that specific actions are associated with each state. For example, we have used one state {addr1} to read the first byte of the address or offset, one state {addx} to add  $X$  to the  $MAR$ , one state {data} to read data from memory, one state {push1} to push  $PCL$  onto the stack, etc.

Figure 11-15 shows a partial state graph for the controller. In most cases, the sequence of states depends on the addressing mode. Paths for JMP and JSR instructions have been omitted from the graph except for JMP\_dir and JSR\_dir. The state graph includes a {reset} state, which is entered whenever the microcontroller is reset. After {reset}, the controller goes to states {cycle8}, {cycle9}, and {cycle10} to get the starting address of the user code from the interrupt vector table.

Execution of each instruction begins in the {fetch} state, where the opcode is read from memory. The opcode is loaded into the *Opcode* register and the *PC* is incremented at the same time the change to the next state occurs. When in the {fetch} state, if the previous instruction was a register-memory instruction such as ADD, the ALU operation is completed and the result is loaded into *A* or *X* at the same time the new opcode is loaded.

**Figure 11-15 Partial State Graph for CPU Controller**



Since the opcode is not available in the *Opcode* register until we reach the second state, the second state must be the same for all instructions. We have called this state {addr1}, since the first byte of the address is often read during this state. All the actions listed under “2nd cycle” in Table 11-4 are performed in state {addr1}.

The state following {addr1} depends on the addressing mode and type of instruction being executed. The state sequences for the different modes and instruction types can be read from the rows in Table 11-4. The next state following the last state in each row is always {fetch}.

At this point, we write behavioral level VHDL code for the 6805 CPU based on Table 11-4 and Figure 11-15. This code defines state sequences for the control state machine based on the addressing modes and types of instructions being executed. The code also specifies the actions to be taken in each state. We can use this code to simulate the CPU and verify that all of the instructions and addressing modes execute properly according to the specifications given in Section 11.2. For convenience in testing, a memory array (*mem*) is defined within the architecture even though it is not part of the CPU.

A complete listing of the behavioral code for the CPU is given in Appendix D. The main sections of the code are

1. Signal and control declarations;
2. Procedure *ALU\_OP*, which performs ALU operations for register-memory instructions, including all two-operand instructions;
3. Procedure *ALU\_I*, which performs ALU and shifting operations for read-modify-write instructions, which have a single operand;
4. Procedure *fill\_memory*, which fills the memory with instructions and data for test purposes;
5. Process *cpu\_cycles*, which specifies the state sequence for the CPU controller and the actions that occur in each state.

After declaring signals to represent the *PC*, *MAR*, *SP*, *Opcode*, and other registers, aliases are used to split *PC* and *MAR* into high and low bytes. The *Opcode* is split into the lower four bits (*OP*) and the upper four bits (*mode*). To make the code more readable, we have used constant declarations to associate the opcode and addressing mode mnemonics with the corresponding four-bit (hex) codes from Table 11-1. For example,

```
-- lower 4 bits of opcode (specifies operation)
subtype ot is std_logic_vector(3 downto 0);
constant SUB: ot := "0000"; constant CMP: ot := "0001";
...
-- upper 4 bits of opcode (specifies addressing mode)
constant REL: ot := "0010"; constant DIRM: ot := "0011";
...
```

Procedure *ALU\_OP* (see Figure 11-16) is called in the {fetch} state to complete the execution of register-memory instructions. The procedure defines the operation of the ALU based on Table 11-2(a). All arithmetic is carried out on unsigned bytes. The variable *res*, which is 9 bits long to allow for a carry, represents the result of the ALU operation. Depending on the operation, *A* or *X* is loaded with *res*(7 *downto* 0) and the carry (*C*) is loaded with *res*(8). The *N* and *Z* flags are updated based on the value of *res*.

Figure 11-16 ALU Operations for Register-Memory Instructions

```

procedure ALU_OP           -- perform ALU operation with 2 operands
  (Md : in std_logic_vector(7 downto 0);
   signal A, X : inout std_logic_vector(7 downto 0);
   signal N, Z, C : inout std_logic) is
variable res : std_logic_vector(8 downto 0); -- result of ALU operation
variable updateNZ : Boolean := TRUE;          -- update NZ flags by default
begin
  case OP is
    when LDA => res := '0'&Md; A <= res(7 downto 0);
    when LDX => res := '0'&Md; X <= res(7 downto 0);
    when ADD => res := ('0'&A) + ('0'&Md);
      C <= res(8); A <= res(7 downto 0);
    when ADC => res:= ('0'&A) + ('0'&Md) + C;
      C <= res(8); A <= res(7 downto 0);
    when SUB => res:= ('0'&A) - ('0'&Md);
      C <= res(8); A <= res(7 downto 0);
    when SBC => res:= ('0'&A) - ('0'&Md) - C;
      C <= res(8); A <= res(7 downto 0);
    when CMP => res:= ('0'&A) - ('0'&Md); C <= res(8);
    when CPX => res:= ('0'&X) - ('0'&Md); C <= res(8);
    when ANDa => res := '0'&(A and Md) ; A <= res(7 downto 0);
    when BITa => res := '0'&(A and Md);
    when ORa => res := '0'&(A or Md); A <= res(7 downto 0);
    when EOR => res := '0'&(A xor Md); A <= res(7 downto 0);
    when others => updateNZ := FALSE;
  end case;
  if updateNZ then N <= res(7);
    if res(7 downto 0) = "00000000" then Z <= '1'; else Z <= '0'; end if;
  end if;
end ALU_OP;

```

Procedure *ALUI* defines ALU and shifting operations for the read-modify-write instructions defined in Table 11-2(b). The single operand, *op1*, may be *A*, *X*, or *Md*, depending on the addressing mode. The 8-bit result of each operation is placed in the variable *res8*, with the exception of *NEG*, where a ninth bit is required for the carry.

For convenience in testing, a procedure called *fill\_memory*, which is not part of the CPU, has been added to load the memory array with test instructions and data. This procedure is called when the processor is reset, and it reads the instruction codes and data from a test file. Reading from memory is simulated by reading from the *mem* array, and writing to memory is simulated by storing data in the *mem* array. Since *mem* is defined with an integer index, the function *CONV\_INTEGER* is called to convert the address in *PC*, *MAR*, or *SP* to an integer whenever memory is referenced.

The process *cpu\_cycles* (Figure 11-17) represents the state machine (Figure 11-15), which controls the CPU and specifies the actions that occur in each state. State changes and register updates occur on the rising edge of the clock. If the reset signal *rst\_b* is '0', the process goes to the {reset} state; otherwise, the case statement tests ST to determine the appropriate actions for each state.

In the {fetch} state, *ALU\_OP* is called for register-memory instructions. If a hardware interrupt has occurred, control goes to state {push1} to initiate pushing registers onto the stack; otherwise, the next opcode is read and control goes to state {addr1}.

**Figure 11-17 Partial Listing of *cpu\_cycles* Process**

```
cpu_cycles: process
    variable reg_mem, hw_interrupt, BR: Boolean;
    variable sign_ext: std_logic_vector(4 downto 0);

begin
    reg_mem:= (mode = imm) or (mode = dir) or (mode = ext) or (mode = ix) or
        (mode = ix1) or (mode = ix2);
    hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');

    wait until rising_edge(CLK);
    if (rst_b = '0') then ST <= reset; fill_memory(mem);
    else
        case ST is
            when reset => SP <= "111111";
                if (rst_b = '1') then ST <= cycle8; end if;
            when fetch =>
                if reg_mem then ALU_OP(Md, A, X, N, Z, C); end if;
                -- complete previous operation
                if hw_interrupt then ST <= push1;
                    else Opcode <= mem(CONV_INTEGER(PC)); PC <= PC+1; -- fetch opcode
                    ST <= addr1; end if;

            when addr1 =>
                case mode is
                    when inha => ALU1(A, N, Z, C); ST <= fetch; -- do operation on A
                    when inhx => ALU1(X, N, Z, C); ST <= fetch; -- do operation on X
                    when imm => Md <= mem(CONV_INTEGER(PC)); -- get immediate data
                        PC <= PC+1; ST <= fetch;
                    when inh1 =>
                        if OP = SWI then ST <= push1;
                        elsif OP = RTS then ST<= pop2; SP <= SP+1;
                        elsif OP = RTI then ST <= pop5; SP <= SP+1;
                        end if;
                end case;
        end case;
    end if;
end process;
```

```

when inh2 =>
  case OP is
    when TAX => X <= A;
    when CLC => C <= '0';
    when SEC => C <= '1';
    when CLI => I <= '0';
    when SEI => I <= '1';
    when RSP => SP <= "111111";
    when TXA => A <= X;
    when others =>
      assert(false) report "illegal instruction, mode = inh2";
  end case;
  ST <= fetch;
when dir =>
  if OP = JMP then PC <= zero&mem(CONV_INTEGER(PC)); ST <= fetch;
  else MAR <= zero&mem(CONV_INTEGER(PC)); PC <= PC+1;
    -- get direct address
    if (OP=JSR) then ST <= push1; else ST <= data; end if;
  end if;
... (remainder of process omitted - see Appendix D)

```

In state {addr1}, the addressing mode is tested in a case statement. For inha and inhx modes, procedure *ALUI* is called to perform the required operation on A or X and control returns to {fetch}. For imm mode, *Md* is loaded with immediate data from memory. For inh2 mode, the control instructions defined in Table 11-2(c) are executed. For dir mode, the direct address is read from memory, prefixed with zero, and loaded into the *MAR* for most instructions. When JMP\_dir is being executed, the direct address is loaded into *PC* instead of *MAR*. For other addressing modes, the actions listed in Table 11-4 are carried out as shown in Appendix D.

Branch instructions defined in Table 11-2(d) are executed in state {testBR}. The boolean variable *BR* is set to TRUE or FALSE, depending on the instruction being executed and the flags that are set. If *BR* is true, then the relative address, which is in *Md*, is sign-extended and added to the current *PC*. Part of the VHDL code for {testBR} is as follows:

```

when testBR =>
  case OP is
    when BRA => BR := TRUE;           -- branch always
    when BRN => BR := FALSE;          -- branch never
    when BHI => BR := (C or Z) = '0'; -- branch if higher
    when BLS => BR := (C or Z) = '1'; -- branch if lower or same
    when BCC => BR := C = '0';        -- branch if carry clear
    when BCS => BR := C = '1';        -- branch if carry set
    ... (other branch instructions omitted here)
  end case;
  if Md(7) = '1' then sign_ext := "11111"; else sign_ext :=
    zero; end if;
  if BR then PC <= PC + (sign_ext&Md); end if;
  ST <= fetch;

```

Reading and writing memory data takes place in state {data}. For STA or STX, A or X is written into the memory location addressed by *MAR*, and the flags are updated. For other instructions, data from the memory location addressed by *MAR* is loaded into *Md*. The actions taken in the other states correspond to the actions given in Table 11-4.

When an interrupt occurs, *PCH*, *PCL*, *X*, *A*, and *CCR* are pushed onto the stack in states {push1} through {push5}. Then *MAR* is loaded with the interrupt vector address in {cycle8}, and the *PC* is loaded with the starting address of the interrupt subroutine in states {cycle9} and {cycle10}.

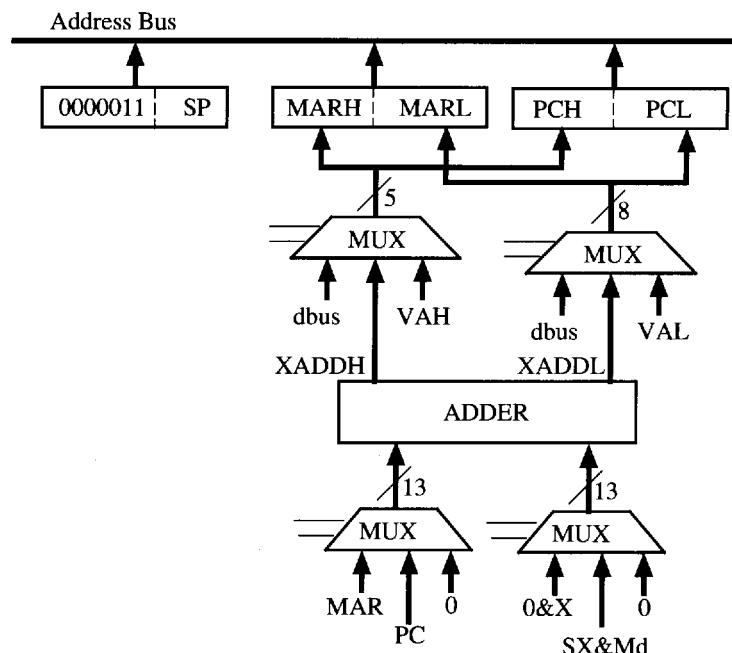
We verified that the behavioral VHDL code for the CPU correctly executes the instructions given in Table 11-2 with the addressing modes given in Table 11-3. We did this by loading the memory array from a file, simulating the instruction execution, and observing the resulting waveforms. We did not attempt to synthesize the CPU from the behavioral VHDL code for several reasons. First, the address and data busses and the interface to the memory are not defined in the code, and memory timing is not taken into account. In particular, reading from memory requires that the address go out on the bus early in the clock cycle so the data from memory can be loaded into a register at the end of the clock cycle. Second, when we wrote the VHDL code, we did not write it so it could be efficiently realized in hardware.

## Hardware Design and Synthesizable VHDL Code

Next, we work out some of the details of the hardware design and then attempt to write VHDL code that will result in an economical hardware realization when it is synthesized. We draw block diagrams for the hardware based on the required register transfers listed in Table 11-4 and on the corresponding behavioral VHDL code. Then we define the control signals that must be generated in each state and rewrite the VHDL code to explicitly generate these control signals.

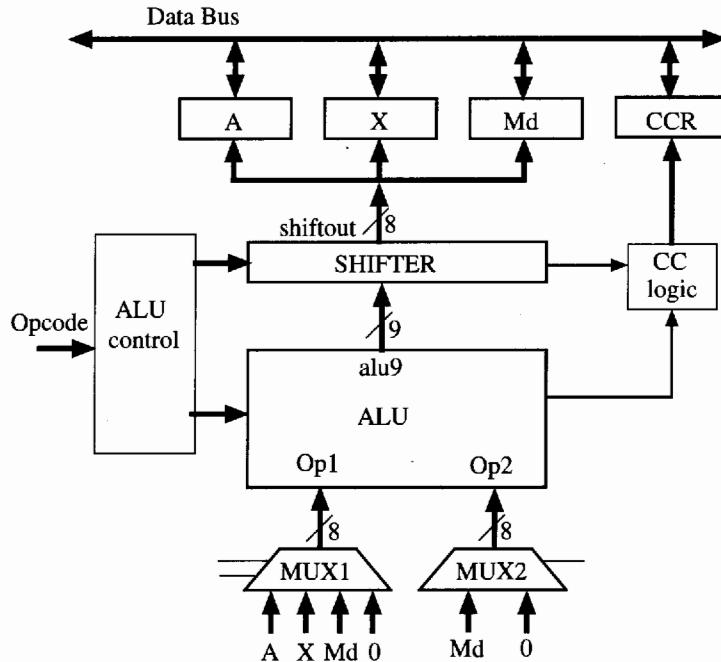
The hardware is naturally divided into three sections—a control unit, an addressing unit that computes the addresses that go out on the address bus, and a data unit that performs arithmetic, logic, and shifting operations. The functions performed by the addressing unit include adding *X* to *MAR* for indexed addressing, adding sign-extended *Md* (*SX&Md*) to *PC* for relative addressing, loading *PC* and *MAR* with the proper addresses, and incrementing *SP*, *PC*, and *MAR*. Figure 11-18 shows the block diagram for one possible configuration for the addressing unit. This design uses a single adder with multiplexers to select the adder inputs. *MARH*, *MARL*, *PCH*, and *PCL* can be loaded from the data bus, from the adder output, or from the vector address table (*VAH* and *VAL*). To avoid additional MUX inputs, *PC* can be loaded from *MAR* by selecting *MAR* and 0 as adder inputs. Also, *PC* or *MAR* can be loaded with *X* by selecting 0 and 0&*X* as adder inputs. Additional logic (not shown) is required for incrementing *PC*, *MAR*, and *SP* and for decrementing *SP*.

Figure 11-18 Addressing Unit



Next, we consider design of the data unit. We could use one ALU for the two-operand arithmetic operations such as ADD and SUB and a separate ALU for single-operand operations such as INC and COM. Alternatively, we could use a single ALU for both types of operations. We chose the latter approach because it reduces the total amount of logic required and simplifies setting the *N*, *Z*, and *C* flags. Figure 11-19 shows a block diagram for one possible configuration for the data unit. When two operands are used, MUX1 selects *A* or *X* and the MUX2 selects *Md*. When a single operand is used, MUX1 selects *A*, *X*, or *Md* and MUX2 selects 0. We have placed a shifter at the ALU output. For all operations except shifting, the ALU output passes straight through the shifter. For shifting operations, *Op1* (*A*, *X*, or *Md*) passes through the ALU and is shifted left or right by the shifter. With this configuration, the result of all data operations always comes from one place (shiftout), and this result can be loaded into *A*, *X*, or *Md*. Shiftout can also be tested to determine the proper settings for the *N* and *Z* flags.

Figure 11-19 Data Unit



The ALU input MUXes can be described in VHDL as follows:

```
-- define ALU input operand MUXes
op1 <= A when selA = '1'                      -- MUX for op1
      else X when selX = '1'
      else Md when selMd1 = '1'
      else "00000000";
op2 <= Md when selMd2 = '1'                      -- MUX for op2
      else "00000000";
```

The control signals *selA*, *selX*, *selMd1*, and *selMd2* depend on the opcode and are generated by the ALU control.

Figure 11-20 shows more details of the ALU. All the ALU operations, except for AND, OR, and XOR, can be implemented using an adder and complementers. Complementers are required at both adder inputs, since the COM operation on a single operand requires complementing *op1*, and subtraction requires complementing *op2*. When control signal *and2* = '1', the ALU output is (*op1 and op2*). When *or2* = '1', the output is (*op1 or op2*). When *xor2* = '1', the output is (*op1 xor op2*). Otherwise, the ALU output is the sum of *op1\_com*, *op2\_com*, and *Cin*. The VHDL code that describes the ALU is as follows:

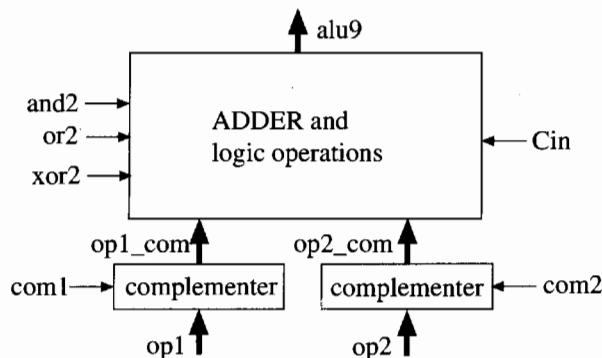
```

op1_com <= not op1 when com1='1' else op1; -- complementers
op2_com <= not op2 when com2='1' else op2;
alu9 <= '0'&(op1_com and op2_com) when and2 ='1' -- logic
operations
else '0'&(op1_com or op2_com) when or2 ='1'
else '0'&(op1_com xor op2_com) when xor2 ='1'
else ('0'&op1_com) + ('0'&op2_com) + Cin; -- adder

```

The ALU output (*alu9*), which is 9 bits wide, consists of the carry from the adder followed by the 8-bit sum. For logic operations, the carry is set to '0'. The signals *com1*, *com2*, *Cin*, *and2*, *or2*, and *xor2* are generated by the ALU control.

**Figure 11-20 ALU Detail**



The VHDL code for the SHIFTER is

```

shiftout <= shiftin&alu9(7 downto 1) when rsh='1' -- right
shift
else alu9(6 downto 0)&shiftin when lsh='1' -- left shift
else alu9(7 downto 0); -- pass through

```

If *rsh* = '1', the ALU output is shifted right and if *lsh* = '1', the ALU output is shifted left; else the ALU passes straight through. The variable *shiftin* is '0', *C*, or the sign bit of the adder output, depending on the type of shift being done.

Synthesizable VHDL code for the CPU is given in Appendix E. We describe further details of the CPU design as we discuss this code. The entity CPU6805 makes the address bus and data bus explicit so we can connect the CPU to a RAM memory and other system components. The architecture *CPU1* has the following sections:

1. Declarations of signals and constants
2. Concurrent statements for the bus interface, ALU input MUXes, ALU and shifter operations, address adder, and operation decoder
3. Process *ALU\_control*, which generates control signals for the ALU and for loading the *A*, *X*, and *Md* registers

4. Process *CPU\_control*, which implements the control state machine (Figure 11-15) and generates control signals for loading registers
5. Process *update\_reg*, which updates the registers on the rising edge of the clock

The signal declarations for the registers and states in the *CPU1* architecture are essentially the same as in the behavioral level VHDL code (Appendix D). In *CPU1* we use concurrent statements, which imply tristate buffers, to drive the address and data busses. Code for the data bus interface is

```
-- drive the data bus with tristate buffers
dbus <= A when A2db='1' else hi_Z;
dbus <= X when X2db='1' else hi_Z;
dbus <= Md when Md2db='1' else hi_Z;
dbus <= "000"&PCH when PCH2db='1' else hi_Z;
dbus <= PCL when PCL2db='1' else hi_Z;
dbus <= "1110"&CCR when CCR2db='1' else hi_Z;
```

Control signals such as *A2db* (*A* to data bus) and *PCL2db* (*PCL* to data bus) are generated in *CPU\_control*.

In the behavioral code, the operation (*OP*) was tested in several case statements, and many other places. Generally, the synthesizer instantiates a 4-bit comparator each time *OP* is tested. To avoid this, we rewrote the code so that a single 4-to-16-line decoder is implied by the VHDL code. The input to this decoder is  $OP = \text{Opcode}(3 \text{ downto } 0)$ , and the output is the signal *opd*, which is a 16-bit vector. We implemented the decoder by indexing into a constant array named *decode* using the following code:

```
type decode_type is array(0 to 15) of bit_vector(15 downto 0);
signal opd : bit_vector(15 downto 0);
constant decode: decode_type :=
(X"0001", X"0002", X"0004", X"0008", X"0010", X"0020",
 X"0040", X"0080",
 X"0100", X"0200", X"0400", X"0800", X"1000", X"2000",
 X"4000", X"8000");
...
opd <= decode(TO_INTEGER(Opcode(3 downto 0))); -- 4-to-16
decoder
```

If  $OP = "0000"$ , then  $opd(0) = '1'$ ; if  $OP = "0001"$ , then  $opd(1) = '1'$ ; etc. To make the code more readable, the opcode names are aliased to bits of *opd*. Thus SUB is *opd(0)*, CMP is *opd(1)*, etc. Then in the code, each time we test the opcode, we have to test only a single bit, which leads to more efficient logic. In contrast, we chose not to explicitly decode the addressing mode and left the tests for the mode the same as in the behavioral code.

The *ALU\_control* process generates signals necessary to execute most register-memory, read-modify-write, and inherent instructions. At the start of the process, control signals are initialized to their most common values. For example, *updateNZ* and *updateC* are set to '1', since the *N*, *Z*, and *C* flags are updated for most operations. Control signals *selA*, *selMd2*, and *ALU2A* are set to '1', since for many operations, the ALU inputs come

from  $A$  and  $Md$ , and the ALU output is stored in  $A$ . For *reg\_mem* instructions, additional signals are turned on or off as required. For example, for SBC (subtract with borrow), *com2* is set to '1' to complement *op2*, and *Cin*  $\leq$  **not** *C*, since the borrow is stored in *C*. For CPX,  $Md$  must be subtracted from  $X$ , so *selX*  $\leq$  '1', *selA*  $\leq$  '0' to select  $X$  as the MUX1 output. Also, *com2*  $\leq$  '1', and *Cin*  $\leq$  '1' to add the 2's complement of  $Md$  to  $X$ . Partial VHDL code for *ALU\_control* is

```
-- set default values of control signals
Cin <= '0'; com1 <= '0'; com2 <= '0'; updateNZ<='1';
updateC<='1';
ALU2A <= '1'; ALU2X <= '0'; ALU2Md <= '0';
selA <= '1'; selX <= '0'; selMd1 <= '0'; selMd2 <= '1';
...
if SUBC='1' then com2<='1'; Cin<= not C; end if;
if CPX = '1' then selX <= '1'; selA <= '0'; com2<='1';
Cin <= '1'; ALU2A<='0'; end if;
```

For *rd\_mod\_wr* instructions, *selMd2*  $\leq$  '0', since *op2* should always be zero. For inha addressing mode, the default selections (*selA* = '1' and *ALU2A* = '1') select *A* as *op1* and store the result in *A*. For *inxh*, *selX*  $\leq$  '1' and *ALU2X*  $\leq$  '1' so the operation is performed on *X*. For other *rd\_mod\_wr* addressing modes, *selMd1*  $\leq$  '1' and *ALU2Md*  $\leq$  '1', so the operation is performed on *Md*. Other control signals are turned on and off as needed. For example, for rotate right (ROR1), *rsh*  $\leq$  '1' and *shiftin*  $\leq$  *C*. For DEC, *updateC*  $\leq$  '0', since the *C* flag is not updated. Also, *com2*  $\leq$  '1' so "11111111" (-1) is added to *op1*. For CLR, *and2*  $\leq$  '1' so *op2* = 0 is ANDed with *op1* to give an ALU output of 0, which is loaded into the appropriate register.

The *CPU\_cycles* process in the behavioral code has been split into two processes—*CPU\_control* and *update\_reg*. The *CPU\_control* process implements the control state machine, and the sequence of states is identical to that for the *CPU\_cycles* process. However, instead of specifying the register transfers directly, *CPU\_control* is a combinational process that generates the control signals for loading the *Opcode*, *PC*, *MAR*, and *SP* registers, for doing address computations, and for controlling the address and data bus interfaces.

Figure 11-21 gives the VHDL code for part of the *CPU\_control* process. The code for each state is derived from the corresponding state in *CPU\_cycles*. For example, in state {fetch}, the code

```
Opcode <= mem(PC), PC <= PC+1; ST <= addr1;
```

is replaced with

```
PC2ab<='1'; db2opcode<='1'; incPC<='1'; nST <= addr1;
```

The signal *PC2ab* enables the *PC* output to the address bus (*ab*), and the signal *db2opcode* enables loading the opcode register from the data bus (*db*). Signals for updating *A*, *X*, *Md*, and the flags are also generated in the *ALU\_control* process. However, these registers should be updated only under certain conditions. To accomplish this, an update signal is

set to '1' whenever updating *A*, *X*, *Md*, and the flags is allowed. Thus update is set to '1' in {fetch} if a *reg\_mem* instruction other than JMP or JSR is being executed. This replaces the call to *ALU\_OP* in the *CPU\_cycles* process.

**Figure 11-21 Partial Listing of *CPU\_control* Process**

```
CPU_control: process (ST, rst_b, opd, mode, IRQ, SCint, CCR, MAR, X, PC, Md)
  variable reg_mem, hw_interrupt, BR : boolean;
begin
  nST <= reset; BR := FALSE; wr <= '0'; update <= '0';
  xadd1 <= (others => '0'); xadd2 <= (others => '0'); va <= "000";
  db2A <= '0'; db2X <= '0'; db2Md <= '0'; db2CCR <= '0'; db2opcode <= '0';
  ... (all control signals are set to '0' here)

  reg_mem:= (mode = imm) or (mode = dir) or (mode = ext) or (mode = ix) or
            (mode = ix1) or (mode = ix2);
  hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');

  if (rst_b = '0') then nST <= reset;
  else
    case ST is
      when reset =>
        setSP <= '1';
        if (rst_b = '1') then nST <= cycle8; end if;
      when fetch =>
        if (reg_mem and JMP = '0' and JSR = '0')
          then update <= '1'; end if; -- update registers if not JMP or JSR
        if hw_interrupt then nST <= push1;
        else PC2ab<='1'; db2opcode<='1'; incPC<='1'; -- read opcode
              nST <= addr1; end if;

      when addr1 =>
        case mode is
          when inha | inhx => update <= '1'; nST<= fetch;
          when imm => PC2ab<= '1'; db2Md<='1'; incPC<='1';
            nST <= fetch;
          when inh1 =>
            if SWI = '1' then nST <= push1;
            elsif RTS = '1' then nST <= pop2; incSP<='1';
            elsif RTI = '1' then nST <= pop5; incSP<='1';
            end if;
          when inh2 => update <= '1'; nST <= fetch;
          when dir => PC2ab<='1';
            if JMP='1' then db2PCL<='1'; clrPCH<='1'; nST<=fetch;
            else db2MARL<='1'; clrMARH<='1'; incPC <= '1';
              if JSR='1' then nST<=push1; else nST<=data; end if;
            end if;
          ... (remainder of process omitted -- see Appendix E)
```

The *update\_reg* process updates the registers when the rising edge of the clock occurs. The portion of the process that updates the *PC* is as follows:

```
wait until CLK'event and CLK='1';
if incPC = '1' then PC <= PC + 1; end if;
if xadd2PC = '1' then PC <= xadd; end if;
if db2PCH = '1' then PCH <= dbus(4 downto 0); end if;
if MARTH2PCH = '1' then PCH <= MARTH; end if;
if MARL2PCL = '1' then PCL <= MARL; end if;
if clrPCH = '1' then PCH <= "00000"; end if;
if db2PCL = '1' then PCL <= dbus; end if;
if X2PCL = '1' then PCL <= X; end if;
```

We have not explicitly defined any multiplexers in this code. Instead, we allow the synthesizer to infer multiplexers when a register must be loaded from several different sources. If the synthesizer has a good optimizer, this may lead to an efficient implementation. If it does not, we can probably improve the implementation by making the multiplexers explicit.

The code for updating *A*, *X*, *Md*, and the flags is as follows:

```
if (update = '1') then
    if (ALU2A = '1') then A <= Shiftout; end if;
    if (ALU2X = '1') then X <= Shiftout; end if;
    if (ALU2Md = '1') then Md <= Shiftout; end if;
    if updateNZ='1' then N <= Shiftout(7);
        if Shiftout = "00000000" then Z <= '1'; else Z <= '0';
        end if;
    end if;
    if updateC='1' then C <= newC; end if;
end if;
```

The update signal is generated in *CPU\_control*, and the other control signals are generated in the *ALU\_control*.

After completing the VHDL code given in Appendix E, we simulated and debugged it using a test bench. We are now ready to integrate the CPU with the other microcontroller components in preparation for synthesis of the complete system.

## 11.4 COMPLETION OF THE MICROCONTROLLER DESIGN

In this section, we complete the design of the microcontroller shown in Figure 11-13. In Section 11.1 we designed the UART, and in Section 11.3 we designed the CPU. After designing the parallel ports, we integrate the components to form the complete microcontroller.

Each parallel port has eight bidirectional I/O pins and two 8-bit registers, as shown in Figure 11-22. The contents of the data direction register (*DDRA*) determine which pins are inputs and which are outputs. If a bit in *DDRA* is '1', the corresponding pin is an output; otherwise, it is an input. The port registers are memory-mapped so that the CPU can load

them or read them. Writing to the *PORTA* register loads data into the register, and this data is transmitted to any I/O pins programmed as outputs. Reading from *PORTA* reads data from the I/O pins. If a pin is programmed as an input, the data applied to the pin is read. If programmed as an output, the data read is normally the same as the *PORTA* register.

**Figure 11-22 Parallel Port Block Diagram**

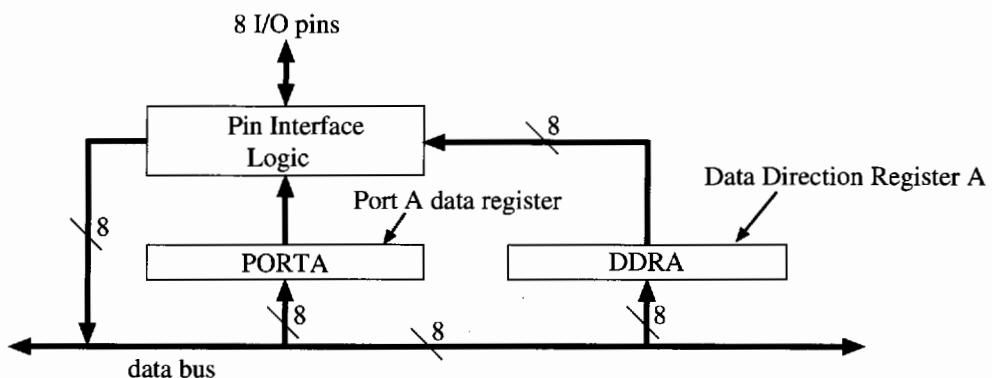


Figure 11-23 shows the VHDL code for the parallel port. Concurrent statements generate the control signals for reading and writing to the registers. *Port\_Sel* is '1' when the port is selected for reading or writing. A single address bit, *ADDR0*, selects either the *PORTA* or *DDRA* register. The generate statement labeled *Portbits* generates the logic associated with each bit in the port. The process updates the port registers on the rising edge of the clock. When the code was synthesized for the Xilinx 4000 series, 17 logic cells were required; when synthesized for the Altera FLEX 10K series, 38 logic elements were required.

**Figure 11-23 VHDL Code for Parallel Port**

```

library ieee;
use ieee.std_logic_1164.all;

entity PORT_A is
port(clk, rst_b, Port_Sel, ADDR0, R_W: in std_logic;
      DBUS: inout std_logic_vector(7 downto 0);
      PinA: inout std_logic_vector(7 downto 0));
end PORT_A;

architecture port1 of PORT_A is
signal DDRA, PORTA : std_logic_vector(7 downto 0);
signal loadDDRA, loadPORTA, ReadPORTA, ReadDDRA : std_logic;

```

```

begin
loadPORTA <= '1' when (Port_Sel='1' and ADDR0='0' and R_W='1') else '0';
loadDDRA <= '1' when (Port_Sel='1' and ADDR0='1' and R_W='1') else '0';
ReadPORTA <= '1' when (Port_Sel='1' and ADDR0='0' and R_W='0') else '0';
ReadDDRA <= '1' when (Port_Sel='1' and ADDR0='1' and R_W='0') else '0';

-- pin interface logic
Portbits: for i in 7 downto 0 generate
    PinA(i) <= PORTA(i) when DDRA(i) = '1' else 'Z'; -- set external pin
state
    DBUS(i) <= DDRA(i) when (ReadDDRA = '1') -- read data direction
register
        else PinA(i) when (ReadPORTA = '1')
        else 'Z';
end generate;

process (clk, rst_b)      -- this process writes to the port registers
begin
    if (rst_b = '0') then DDRA <= "00000000"; -- set all pins to inputs
    elsif (rising_edge(clk)) then
        if (loadDDRA = '1') then DDRA <= DBUS; end if;
        if (loadPORTA = '1') then PORTA <= DBUS; end if;
    end if;
end process;
end port1;

```

At this point we are ready to link all of the components together to form the complete microcontroller. The code shown in Figure 11-24 instantiates the cpu6805 component, two copies of the PORT\_A (parallel port), one UART, a low RAM, and a high RAM. For test purposes, we have scaled down the memory size so that we can synthesize the entire system using a Xilinx 4020E FPGA or an Altera FLEX 10K20 CPLD as a target. For the FPGA implementation, we initially used one  $32 \times 8$  RAM in low memory so that direct addressing could be tested, and a second  $32 \times 8$  RAM in high memory so that extended addressing could be tested. The ram32X8\_io component uses eight CLBs configured as read/write memory cells (see Figure 6-24). A tristate buffer is used with each CLB to provide a bidirectional I/O line.

The memory map for the components is as follows:

- PortA is selected for addresses 0000h–0001h.
- PortB is selected for addresses 0002h–0003h.
- SCI is selected for addresses 0004h–0007h.
- Low RAM is selected for addresses 0020h–003Fh.
- High RAM is selected for addresses 1FE0h–1FFFh.

The address decoder is implemented using conditional assignment statements.

The write signal from the CPU (*wr*) is asserted for several clock cycles in a row when pushing registers onto the stack. In order to write a byte to memory every clock cycle, we generate the write enable to RAM (*we*) by ANDing *wr* with the clock so that a write pulse is generated every cycle. The memory module we are using stores data on the falling edge of the *we* signal. To avoid writing spurious data to the RAM, the write enable

signal (*we*) and the memory-enable signal (*cs*) should be high during the second half of the clock cycle when the address and data lines are stable. Therefore,

```
cs1 <= SelLowRam and not clk;      -- select on 2nd half of
                                         clock cycle
we <= wr and not clk;           -- write enable on 2nd half
                                         of clock cycle
```

Figure 11-24 Top-level VHDL for 6805 Microcontroller

```
library ieee;
use ieee.std_logic_1164.all;

entity m68hc05 is
port(clk, rst_b, irq, RxD : in std_logic;
      PortA, PortB : inout std_logic_vector(7 downto 0);
      TxD : out std_logic);
end m68hc05;

architecture M6805_64 of m68hc05 is
component cpu6805
port(clk, rst_b, IRQ, SCint: in std_logic;
      dbus : inout std_logic_vector(7 downto 0);
      abus : out std_logic_vector(12 downto 0);
      wr: out std_logic);
end component;

component ram32X8_io
port (addr_bus: in std_logic_vector(4 downto 0);
      data_bus: inout std_logic_vector(7 downto 0);
      cpu_wr: in std_logic);
end component;

component PORT_A
port(clk, rst_b, Port_Sel, ADDR, R_W : in std_logic;
      DBUS : inout std_logic_vector(7 downto 0);
      PinA : inout std_logic_vector(7 downto 0));
end component;

component UART
port(SCI_sel, R_W, clk, rst_b, RxD : in std_logic;
      ADDR : in std_logic_vector(1 downto 0);
      DBUS : inout std_logic_vector(7 downto 0);
      SCI_IRQ, TxD : out std_logic);
end component;

signal SCint, wr, cs, we: std_logic;
signal SelLowRam, SelHiRAM, SelPA, SelPB, SelSC : std_logic;
signal addr_bus: std_logic_vector(12 downto 0) := (others => '0');
signal data_bus: std_logic_vector(7 downto 0) := (others => '0');
```

```

begin
CPU: cpu6805 port map (clk, rst_b, irq, SCint, data_bus, addr_bus, wr);
PA: PORT_A port map (clk, rst_b, SelPA, addr_bus(0), wr, data_bus, PortA);
PB: PORT_A port map (clk, rst_b, SelPB, addr_bus(0), wr, data_bus, PortB);
Uart1: UART port map (SelSC, wr, clk, rst_b, RxD, addr_bus(1 downto 0),
                      data_bus, SCint, TxD);
LowRAM: ram32X8_io port map (addr_bus(4 downto 0), data_bus, cs1, we);
HiRAM: ram32X8_io port map (addr_bus(4 downto 0), data_bus, cs2, we);

-- memory interface
cs1 <= SelLowRam and not clk; -- select ram on 2nd half of clock cycle
cs2 <= SelHiRam and not clk;
we <= wr and not clk; -- write enable on 2nd half of clock cycle

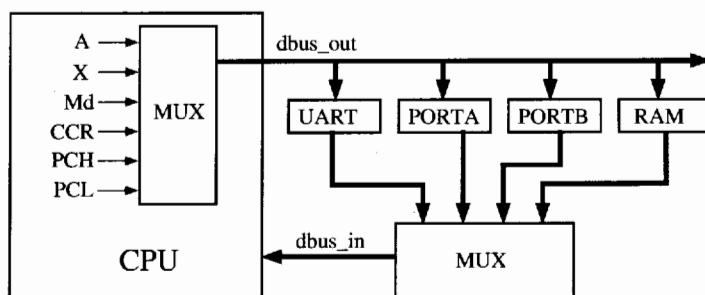
-- address decoder
SelPA <= '1' when addr_bus(12 downto 1) = "000000000000" else '0';
SelPB <= '1' when addr_bus(12 downto 1) = "000000000001" else '0';
SelSC <= '1' when addr_bus(12 downto 2) = "000000000001" else '0';
SelLowRam <= '1' when addr_bus(12 downto 5) = "00000001" else '0';
--      32 <= addr <= 63
SelHiRam <= '1' when addr_bus(12 downto 5) = "11111111" else '0';
--      addr >= 8160 (1FE0h)
end M6805_64;

```

When we synthesized our 6805 microcontroller VHDL code with the XC4020E as a target, the resulting implementation required 692 *F* and *G* function generators and 178 flip-flops, plus 16 CLBs for the RAM. We simulated the system to verify correct execution of the instructions, interrupts, and I/O interfaces. We then downloaded the bit file to a 4020 so that we could test the hardware. The exercises at the end of this chapter suggest ways in which the design can be improved and expanded.

In order to synthesize the 6805 microcontroller for the FLEX 10K20, a number of changes in the VHDL code were required. Since the 10K20 does not support internal tristate bidirectional busses, we changed the CPU data bus structure and used multiplexers to select the data going into and out of the CPU, as shown in Figure 11-25. We also changed the memory components and used four of the 10K20 EABs to implement a  $1024 \times 8$  RAM. Except for the EABs, our design requires less than 50% of the resources on the 10K20.

**Figure 11-25 Multiplexed Data Bus Structure**



In this chapter we designed and implemented a UART and a microcontroller using VHDL and synthesis tools. We used the following steps in designing the microcontroller CPU:

1. Define the register structure, instruction set, and addressing modes.
2. Construct a table that shows the register transfers that take place during each clock cycle.
3. Design the control state machine.
4. Write behavioral VHDL code based on (1), (2), and (3). Simulate execution of the instructions to verify that specifications are met.
5. Work out block diagrams for the major components of the CPU and determine the needed control signals.
6. Rewrite the VHDL based on (5). Again, simulate execution of the instructions.
7. Synthesize the CPU from the VHDL code. Make changes in the VHDL code as needed to improve the synthesis results.
8. Download the bit file to the actual hardware and verify the operation.

Once we have written and debugged our VHDL code, use of synthesis tools makes it easy to develop a hardware prototype. After we have evaluated the prototype, it is relatively easy to change the design at the VHDL level and then resynthesize it. Although we targeted our design for a specific PLD, retargeting the design for different components is usually straightforward, although changes in the VHDL code may be required.

## Problems

**11.1** Make necessary changes in the UART receiver VHDL code so that it uses a 16X bit clock instead of an 8X bit clock. Using a faster sampling clock can improve the noise immunity of the receiver.

**11.2**

(a) Write a VHDL test bench for the UART. Include cases to test overrun error, framing error, noise causing a false start, change of BAUD rate, etc. Simulate the VHDL code.

(b) If suitable hardware is available, write a simpler test bench to allow a loop-back test with  $TxD$  externally connected to  $RxD$ . Synthesize the test bench along with the UART, download to the target device, and verify correct operation of the hardware.

**11.3** Make necessary changes to the VHDL code to add a parity option to the UART described in Section 11.1. Add two bits ( $P_1P_0$ ) to the SCCR that select the parity mode as follows:

- |               |  |
|---------------|--|
| $P_1P_0 = 00$ | 8 data bits, no parity bit             |
| $P_1P_0 = 01$ | 7 data bits, 8th bit makes parity even |
| $P_1P_0 = 10$ | 7 data bits, 8th bit makes parity odd  |
| $P_1P_0 = 11$ | 7 data bits, 8th bit is always '0'     |

The transmitter should generate the even, odd, or '0' parity bit as specified. The receiver should check the parity bit to verify that it is correct. If not, it should set a PE (parity error) flag in the SCSR.

**11.4** The BSR instruction (branch to subroutine) works like JSR except that BSR uses relative addressing. Add BSR to Table 11-4 and to the VHDL code in Appendix D or E.

**11.5** The BSET  $n$  (bit set) instruction is two bytes long and always uses direct addressing. It sets bit  $n$  ( $0 \leq n \leq 7$ ) in the specified memory location. BCLR  $n$  (bit clear) is similar, except that it clears bit  $n$ . Add BSET and BCLR to Table 11-4 and to the VHDL code in Appendix D or E.

**11.6** The BRSET  $n$  (branch if bit set) is three bytes long. Byte 2 is the direct address, and byte 3 is the relative address (rel) for the branch. BRSET reads the data from memory and sets  $C$  equal to bit  $n$ . If bit  $n$  is set, a branch to the original  $PC + 3 + rel$  occurs. BRCLR  $n$  (branch if bit clear) is similar, except that the branch occurs if bit  $n$  is clear. Add BRCLR and BRSET to Table 11-4, to Figure 11-15, and to the VHDL code in Appendix D or E.

**11.7** Add a WAIT (wait for interrupt) instruction that clears  $I$  and then stops the processor until a hardware interrupt occurs. Add WAIT to Figure 11-15 and to the VHDL code in Appendix D or E.

**11.8** Synthesize the CPU VHDL code (Appendix E) using different optimization options (such as optimize for area or for speed) and compare the results.

**11.9** Rewrite the VHDL code for the CPU (Appendix E) to produce a realization that requires fewer logic elements when it is synthesized. Try the following:

(a) Write code to use a single decoder for the mode. Change all the tests for "mode = ..." to test a single bit from the decoder output. (Do this by changing the aliases for the addressing modes.)

(b) Explicitly generate control signals for the MUXes in the addressing unit (Figure 11-18), and write VHDL code to infer these MUXes.

(c) Derive logic equations for each bit of the "ADDER and logic operations" block of Figure 11-20, and use these equations in the VHDL code.

**11.10** Rewrite the VHDL code for the parallel port (Figure 11-23) using a process instead of concurrent statements for the decoding and pin interface logic and compare the synthesis results.

# APPENDIX A

## VHDL LANGUAGE SUMMARY

Reserved words are in boldface type. Square brackets enclose optional items. Curly brackets enclose items that are repeated zero or more times. A vertical bar (|) indicates or.

*Disclaimer:* This VHDL summary is not complete and contains some special cases. Only VHDL statements used in this text are listed. For a complete description of VHDL syntax, refer to references [7] and [17].

*signal assignment statement:* (sequential or concurrent statement)

signal <= [reject pulse-width | **transport**] expression [**after** delay\_time];

*Note:* If concurrent, signal value is recomputed every time a change occurs on the right-hand side. If after time-spec is omitted, signal is updated after delta time.

*variable assignment statement:* (sequential statement only)

variable := expression;

*Note:* This can be used only within a process, function, or procedure. The variable is always updated immediately.

*conditional assignment statement:* (concurrent statement only)

signal <= expression1 **when** condition1

**else** expression2 **when** condition2

...

[**else** expression];

*selected signal assignment statement:* (concurrent statement only)

**with** expression **select**

signal <= expression1 [**after** delay\_time] **when** choice1,

expression2 [**after** delay\_time] **when** choice2,

...

[expression [**after** delay\_time] **when others**];

*entity declaration:*

```
entity entity-name is
    [generic (list-of-generics-and-their-types)];
    [port (interface-signal-declaration)];
    [declarations]
end entity-name;
```

*interface-signal declaration:*

```
list-of-interface-signals: mode type [:= initial-value]
{; list-of-interface-signals: mode type [:= initial-value]}
```

*Note:* An interface signal can be of mode in, out, inout, or buffer.

*architecture declaration:*

```
architecture architecture-name of entity-name is
    [declarations] -- variable declarations not allowed
begin
    architecture-body
end architecture-name;
```

*Note:* The architecture body may contain component-instantiation statements, processes, blocks, assignment statements, procedure calls, etc.

*integer type declaration:*

```
type type_name is range integer_range;
```

*enumeration type declaration:*

```
type type_name is (list-of-names-or-characters);
```

*subtype declaration:*

```
subtype subtype_name is type_name [index-or-range-constraint];
```

*variable declaration:*

```
variable list-of-variable-names : type_name [ := initial_value ];
```

*signal declaration:*

```
signal list-of-signal-names : type_name [ := initial_value ];
```

*constant declaration:*

```
constant constant_name : type_name := constant_value;
```

*alias declaration:*

**alias** identifier [:identifier-type] **is** item-name;

*Note:* Item-name can be a constant, signal, variable, file, function name, type name, etc.

*array type and object declaration:*

```
type array_type_name is array index_range of element_type;
```

```
signal | variable | constant array_name: array_type_name [ := initial_values ];
```

*process statement (with sensitivity list):*

```
[process-label:] process (sensitivity-list)
    [declarations]           -- signal declarations not allowed
    begin
        sequential statements
    end process [process-label];
```

*Note:* This form of process is executed initially and thereafter only when an item on the sensitivity list changes value. The sensitivity list is a list of signals. No wait statements are allowed.

*process statement (without sensitivity list):*

```
[process-label:] process
    [declarations]           -- signal declarations not allowed
    begin
        sequential statements
    end process [process-label];
```

*Note:* This form of process must contain one or more wait statements. It starts execution immediately and continues until a wait statement is encountered.

*wait statements can be of the form:*

```
wait on sensitivity-list;
wait until boolean-expression;
wait for time-expression;
```

*if statement:* (sequential statement only)

```
if condition then
    sequential statements
{ elsif condition then
    sequential statements }   -- 0 or more elsif clauses may be included
[else sequential statements]
end if;
```

*case statement:* (sequential statement only)

```
case expression is
    when choice1 => sequential statements
    when choice2 => sequential statements
    ...
    [when others => sequential statements]
end case;
```

*for loop statement:* (sequential statement only)

```
[loop-label:] for identifier in range loop
    sequential statements
end loop [loop-label];
```

*Note:* You may use **exit** to exit the current loop.

*while loop statement:* (sequential statement only)  
[loop-label:] **while** boolean-expression **loop**  
    sequential statements  
**end loop** [loop-label];

*exit statement:* (sequential statement only)  
**exit** [loop-label] [**when** condition];

*assert statement:* (sequential or concurrent statement)  
**assert** boolean-expression  
    [**report** string-expression]  
    [**severity** severity-level];

*report statement:* (sequential statement only)  
**report** string-expression  
    [**severity** severity-level];

*procedure declaration:*  
**procedure** procedure-name (parameter list) **is**  
    [declarations]  
**begin**  
    sequential statements  
**end** procedure-name;

*Note:* Parameters may be signals, variables, or constants.

*procedure call:*  
procedure-name (actual-parameter-list);

*Note:* An expression may be used for an actual parameter of mode in; types of the actual parameters must match the types of the formal parameters; open cannot be used.

*function declaration:*  
**function** function-name (parameter-list) **return** return-type **is**  
    [declarations]  
**begin**  
    sequential statements -- must include **return** return-value;  
**end** function-name;

*Note:* Parameters may be signals or constants.

*function call:*  
function-name (actual-parameter list)

*Note:* A function call is used within (or in place of) an expression.

*library declaration:*  
**library** list-of-library-names;

*use statement:*

**use library\_name.package\_name.item;** (.item may be .all)

*package declaration:*

```
package package-name is  
    package declarations  
end [package][package-name];
```

*package body:*

```
package body package-name is
    package body declarations
end [package body][package name];
```

*component declaration:*

```

component component-name
    [generic (list-of-generics-and-their-types);]
    port (list-of-interface-signals-and-their-types);
end component;

```

*component instantiation:*

label: component-name  
[generic map (generic-association-list);]  
port map (list-of-actual-signals);

*Note:* Use **open** if a component output has no connection.

*generate statements:*

```
generate_label: for identifier in range generate
[begin]
    concurrent statement(s)
end generate [generate_label];
```

**generate\_label:** if condition generate  
[begin]  
                concurrent statement(s)  
**end generate** [generate\_label];

### *file type declaration:*

**type** file\_name is file of type name:

*file declaration:*

**file** file name; file type [**open mode**] is "file pathname";

*Note:* Mode may be read mode, write mode, or append mode.

# APPENDIX B

## BIT PACKAGE

```
-- Bit package for Digital Systems Design Using VHDL

package bit_pack is
    function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
        return bit_vector;
    function falling_edge(signal clock:bit)
        return Boolean ;
    function rising_edge(signal clock:bit)
        return Boolean ;
    function vec2int(vec1: bit_vector)
        return integer;
    function int2vec(int1,NBits: integer)
        return bit_vector;
    procedure Addvec
        (Add1,Add2: in bit_vector;
        Cin: in bit;
        signal Sum: out bit_vector;
        signal Cout: out bit;
        n: in natural);

    component jkff
        generic(DELAY:time := 10 ns);
        port(SN, RN, J,K,CLK: in bit; Q, QN: inout bit);
    end component;

    component dff
        generic(DELAY:time := 10 ns);
        port (D, CLK: in bit; Q: out bit; QN: out bit := '1');
    end component;

    component and2
        generic(DELAY:time := 10 ns);
        port(A1, A2: in bit; Z: out bit);
    end component;
```

```
component and3
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3: in bit; Z: out bit);
end component;

component and4
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3, A4: in bit; Z: out bit);
end component;

component or2
    generic(DELAY:time := 10 ns);
    port(A1, A2: in bit; Z: out bit);
end component;

component or3
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3: in bit; Z: out bit);
end component;

component or4
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3, A4: in bit; Z: out bit);
end component;

component nand2
    generic(DELAY:time := 10 ns);
    port(A1, A2: in bit; Z: out bit);
end component;

component nand3
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3: in bit; Z: out bit);
end component;

component nand4
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3, A4: in bit; Z: out bit);
end component;

component nor2
    generic(DELAY:time := 10 ns);
    port(A1, A2: in bit; Z: out bit);
end component;

component nor3
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3: in bit; Z: out bit);
end component;
```

```
component nor4
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3, A4: in bit; Z: out bit);
end component;

component inverter
    generic(DELAY:time := 10 ns);
    port(A : in bit; Z: out bit);
end component;

component xor2
    generic(DELAY:time := 10 ns);
    port(A1, A2: in bit; Z: out bit);
end component;

component c74163
    port(LdN, ClrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
          Cout: out bit; Q:  inout bit_vector(3 downto 0) );
end component;
end bit_pack;

package body bit_pack is

-- This function adds 2 4-bit numbers, returns a 5-bit sum
function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
    return bit_vector is
variable cout: bit:='0';
variable cin: bit:=carry;
variable retval: bit_vector(4 downto 0):="00000";
begin
lp1: for i in 0 to 3 loop
    cout :=(reg1(i) and reg2(i)) or ( reg1(i) and cin) or
              (reg2(i) and cin );
    retval(i) := reg1(i) xor reg2(i) xor cin;
    cin := cout;
end loop lp1;
retval(4):=cout;
return retval;
end add4;

-- Function for falling edge
function falling_edge(signal clock:bit)
    return Boolean is
begin
    return clock'event and clock = '0';
end falling_edge;
```

```
-- Function for rising edge
function rising_edge(signal clock:bit)
  return Boolean is
begin
  return clock'event and clock = '1';
end rising_edge;

-- Function vec2int (converts a bit vector to an integer)
function vec2int(vec1: bit_vector)
  return integer is
variable retval: integer:=0;
alias vec: bit_vector(vec1'length-1 downto 0) is vec1;
begin
  for i in vec'high downto 1 loop
    if (vec(i)='1') then
      retval:=(retval+1)*2;
    else
      retval:=retval*2;
    end if;
  end loop;
  if vec(0)='1' then
    retval:=retval+1;
  end if;
  return retval;
end vec2int;

-- Function int2vec (converts a positive integer to a bit vector)
function int2vec(int1,NBits: integer)
  return bit_vector is
variable N1: integer;
variable retval: bit_vector(NBits-1 downto 0);
begin
  assert int1 >= 0;
  report "Function int2vec: input integer cannot be negative"
  severity error;
  N1:=int1;
  for i in retval'Reverse_Range loop
    if (N1 mod 2)=1 then
      retval(i):='1';
    else
      retval(i):='0';
    end if;
    N1:=N1/2;
  end loop;
  return retval;
end int2vec;
```

```
-- This procedure adds two n-bit bit_vectors and a carry and
-- returns an n-bit sum and a carry. Add1 and Add2 are assumed
-- to be of the same length and dimensioned n-1 downto 0.
procedure Addvec
  (Add1,Add2: in bit_vector;
   Cin: in bit;
   signal Sum: out bit_vector;
   signal Cout: out bit;
   n:in natural) is
  variable C: bit;
begin
  C := Cin;
  for i in 0 to n-1 loop
    Sum(i) <= Add1(i) xor Add2(i) xor C;
    C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
  end loop;
  Cout <= C;
end Addvec;

end bit_pack;

-- 2 input AND gate
entity And2 is
  generic(DELAY:time);
  port (A1,A2: in bit;
        Z: out bit);
end And2;
architecture concur of And2 is
begin
  Z <= A1 and A2 after DELAY;
end;

-- 3 input AND gate
entity And3 is
  generic(DELAY:time);
  port (A1,A2, A3: in bit;
        Z: out bit);
end And3;
architecture concur of And3 is
begin
  Z <= A1 and A2 and A3 after DELAY;
end;
```

```
--4 input AND gate
entity And4 is
  generic(DELAY:time);
  port (A1,A2,A3,A4:  in bit;
        Z: out bit);
end And4;
architecture concur of And4 is
begin
  Z <= A1 and A2 and A3 and A4 after DELAY;
end;

--2 input OR gate
entity Or2 is
  generic(DELAY:time);
  port (A1,A2:  in bit;
        Z: out bit);
end Or2;
architecture concur of Or2 is
begin
  Z <= A1 or A2 after DELAY;
end;

--3 input OR gate
entity Or3 is
  generic(DELAY:time);
  port (A1,A2,A3:  in bit;
        Z: out bit);
end Or3;
architecture concur of Or3 is
begin
  Z <= A1 or A2 or A3 after DELAY;
end;

--4 input OR gate
entity Or4 is
  generic(DELAY:time);
  port (A1,A2,A3,A4:  in bit;
        Z: out bit);
end Or4;
architecture concur of Or4 is
begin
  Z <= A1 or A2 or A3 or A4 after DELAY;
end;
```

```
--2 input NAND gate
entity Nand2 is
  generic(DELAY:time);
  port (A1,A2:  in bit;
        Z: out bit);
end Nand2;
architecture concur of Nand2 is
begin
  Z <= not (A1 and A2) after DELAY;
end;

--3 input NAND gate
entity Nand3 is
  generic(DELAY:time);
  port (A1,A2, A3:  in bit;
        Z: out bit);
end Nand3;
architecture concur of Nand3 is
begin
  Z <= not (A1 and A2 and A3) after DELAY;
end;

--4 input NAND gate
entity Nand4 is
  generic(DELAY:time);
  port (A1,A2,A3,A4:  in bit;
        Z: out bit);
end Nand4;
architecture concur of Nand4 is
begin
  Z <= not (A1 and A2 and A3 and A4) after DELAY;
end;

--2 input NOR gate
entity Nor2 is
  generic(DELAY:time);
  port (A1,A2:  in bit;
        Z: out bit);
end Nor2;
architecture concur of Nor2 is
begin
  Z <= not (A1 or A2) after DELAY;
end;
```

```
--3 input NOR gate
entity Nor3 is
  generic(DELAY:time);
  port (A1,A2,A3:  in bit;
        Z: out bit);
end Nor3;
architecture concur of Nor3 is
begin
  Z <= not (A1 or A2 or A3) after DELAY;
end;

--4 input NOR gate
entity Nor4 is
  generic(DELAY:time);
  port (A1,A2,A3,A4:  in bit;
        Z: out bit);
end Nor4;
architecture concur of Nor4 is
begin
  Z <= not (A1 or A2 or A3 or A4) after DELAY;
end;

--An INVERTER
entity Inverter is
  generic(DELAY:time);
  port (A:  in bit;
        Z: out bit);
end Inverter;
architecture concur of Inverter is
begin
  Z <= not A after DELAY;
end;

--A 2 INPUT XOR2 GATE
entity XOR2 is
  generic(DELAY:time);
  port (A1,A2:  in bit;
        Z: out bit);
end XOR2;
architecture concur of XOR2 is
begin
  Z <= A1 xor A2 after DELAY;
end;
```

```
--JK Flip-flop
entity JKFF is
    generic(DELAY:time);
    port(SN, RN, J,K,CLK: in bit;
          Q, QN:  inout bit);
end JKFF;
use work.bit_pack.all;
architecture JKFF1 of JKFF is
begin
process(CLK, SN, RN)
begin
    if RN='0' then
        Q <= '0' after DELAY;
    elsif SN='0' then
        Q<='1' after DELAY;
    elsif falling_edge(CLK) then
        Q <= (J and not Q) or (not K and Q) after DELAY;
    end if;
end process;
QN <= not Q;
end JKFF1;

--D Flip-flop
entity DFF is
    generic(DELAY:time);
    port (D, CLK: in bit;
          Q: out bit; QN: out bit := '1');
    -- initialize QN to '1' since bit signals are initialized to '0' by
    default
end DFF;
architecture SIMPLE of DFF is
begin
process(CLK)
begin
    if CLK = '1' then --rising edge of clock
        Q <= D after DELAY;
        QN <= not D after DELAY;
    end if;
end process;
end SIMPLE;

--74163 COUNTER
entity c74163 is
    port(LdN, ClrN, P, T, CK: in bit;  D: in bit_vector(3 downto 0);
          Cout: out bit; Q:  inout bit_vector(3 downto 0) );
end c74163;

use work.bit_pack.all;
```

```
architecture b74163 of c74163 is
begin
    Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
process
begin
    wait until CK = '1';           -- change state on rising edge
    if ClrN = '0' then Q <= "0000";
    elsif LdN = '0' then Q <= D;
    elsif (P and T) = '1' then
        Q <= int2vec(vec2int(Q)+1,4);
    end if;
end process;
end b74163;
```

# APPENDIX C

## TEXTIO PACKAGE

```
package TEXTIO is
  -- Type Definitions for Text I/O
  type LINE is access STRING;      -- a LINE is a pointer to a STRING value
  type TEXT is file of STRING;     -- a file of variable-length ASCII records
  type SIDE is (RIGHT, LEFT);      -- for justifying output data w/in fields
  subtype WIDTH is NATURAL;        -- for specifying widths of output fields

  -- Standard Text Files
  file INPUT: TEXT open read_mode is "STD_INPUT";
  file OUTPUT: TEXT open write_mode is "STD_OUTPUT";

  -- Input Routines for Standard Types
  procedure READLINE (file F: TEXT; L: out LINE);
  procedure READ (L: inout LINE; VALUE: out BIT; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out BIT);
  procedure READ (L: inout LINE; VALUE: out BIT_VECTOR;
                  GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out BIT_VECTOR);
  procedure READ (L: inout LINE; VALUE: out BOOLEAN; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out CHARACTER;
                  GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out CHARACTER);
  procedure READ (L: inout LINE; VALUE: out INTEGER; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out INTEGER);
  procedure READ (L: inout LINE; VALUE: out REAL; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out REAL);
  procedure READ (L: inout LINE; VALUE: out STRING; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out STRING);
  procedure READ (L: inout LINE; VALUE: out TIME; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out TIME);
```

```
-- Output Routines for Standard Types
procedure WRITELINE (file F: TEXT; L: inout LINE);
procedure WRITE (L: inout LINE; VALUE: in BIT;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in BIT_VECTOR;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in BOOLEAN;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in CHARACTER;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in INTEGER;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in REAL;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                DIGITS: in NATURAL:= 0);
procedure WRITE (L: inout LINE; VALUE: in STRING;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in TIME;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                UNIT: in TIME:= ns);

end package TEXTIO;
```

## APPENDIX D

# BEHAVIORAL VHDL CODE FOR M6805 CPU

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.CONV_STD_LOGIC_VECTOR
use std.textio.all;

entity M6805 is
port(clk: in std_logic;
      rst_b: in std_logic;                      -- active low reset signal
      IRQ, SCint: in std_logic);                -- hardware interrupt signals
end M6805;

architecture behv of M6805 is
  type RAMtype is array (0 to 8191) of std_logic_vector(7 downto 0);
  signal mem: RAMtype:= (others=>(others=> '0'));
  signal memory : std_logic_vector(7 downto 0);
  signal Opcode,A,X,Md: std_logic_vector(7 downto 0) := (others => '0');
    alias OP: std_logic_vector(3 downto 0) is Opcode(3 downto 0);
    alias mode: std_logic_vector(3 downto 0) is Opcode(7 downto 4);
  type state_type is (reset, fetch, addr1, addr2, addX, data, rd_mod_wr,
    writeback, testBR, push1, push2, push3, push4, push5, cycle8,
    cycle9,cycle10, pop5, pop4, pop3, pop2, pop1);
  signal ST : state_type;
  signal CCR: std_logic_vector(3 downto 0);   -- H not implemented
    alias I: std_logic is CCR(3);  alias N : std_logic is CCR(2);
    alias Z : std_logic is CCR(1);  alias C : std_logic is CCR(0);
  signal PC, MAR: std_logic_vector (12 downto 0);
    alias PCH : std_logic_vector(4 downto 0) is PC(12 downto 8);
    alias PCL : std_logic_vector(7 downto 0) is PC(7 downto 0);
    alias MARH : std_logic_vector(4 downto 0) is MAR(12 downto 8);
    alias MARL : std_logic_vector(7 downto 0) is MAR(7 downto 0);
  signal SP: std_logic_vector(5 downto 0);
  constant zero: std_logic_vector(4 downto 0) := "00000";
```

```

-- lower 4 bytes of opcode
subtype ot is std_logic_vector(3 downto 0);
constant SUB: ot:="0000"; constant CMP: ot:="0001";
constant SBC: ot:="0010"; constant CPX: ot:="0011";
constant ANDa: ot:="0100"; constant BITa: ot:="0101";
constant LDA: ot:="0110"; constant STA: ot:="0111";
constant EOR: ot:="1000"; constant ADC: ot:="1001";
constant ORA: ot:="1010"; constant ADD: ot:="1011";
constant JMP: ot:="1100"; constant JSR: ot:="1101";
constant LDX: ot:="1110"; constant STX: ot:="1111";
constant RTI: ot:="0000"; constant RTS: ot:="0001";
{constant values for remaining opcodes should be inserted here}

-- upper 4 bytes of opcode
constant REL: ot:="0010"; constant DIRM: ot:="0011";
constant INHA: ot:="0100"; constant INHX: ot:="0101";
constant IX1M: ot:="0110"; constant IXM: ot:="0111";
constant INH1: ot:="1000"; constant INH2: ot:="1001";
constant IMM: ot:="1010"; constant DIR: ot:="1011";
constant EXT: ot:="1100"; constant IX2: ot:="1101";
constant IX1: ot:="1110"; constant IX: ot:="1111";

procedure ALU_OP          -- perform ALU operation
(Md : in std_logic_vector(7 downto 0);
 signal A, X : inout std_logic_vector(7 downto 0);
 signal N, Z, C : inout std_logic) is
variable res : std_logic_vector(8 downto 0); -- result of ALU operation
variable updateNZ : Boolean := TRUE;      -- update NZ flags by default
begin
  case OP is
    when LDA => res := '0'&Md; A <= res(7 downto 0);
    when LDX => res := '0'&Md; X <= res(7 downto 0);
    when ADD => res := ('0'&A) + ('0'&Md);
      C <= res(8); A <= res(7 downto 0);
    when ADC => res:= ('0'&A) + ('0'&Md) + C;
      C <= res(8); A <= res(7 downto 0);
    when SUB => res:= ('0'&A) - ('0'&Md);
      C <= res(8); A <= res(7 downto 0);
    when SBC => res:= ('0'&A) - ('0'&Md) - C;
      C <= res(8); A <= res(7 downto 0);
    when CMP => res:= ('0'&A) - ('0'&Md); C <= res(8);
    when CPX => res:= ('0'&X) - ('0'&Md); C <= res(8);
    when ANDa => res := '0'&(A and Md); A <= res(7 downto 0);
    when BITa => res := '0'&(A and Md);
    when ORA => res := '0'&(A or Md); A <= res(7 downto 0);
    when EOR => res := '0'&(A xor Md); A <= res(7 downto 0);
    when others => updateNZ := FALSE;
  end case;
end;

```

```

if updateNZ then N <= res(7);
  if res(7 downto 0) = "00000000" then Z <= '1'; else Z <= '0'; end if;
end if;
end ALU_OP;

Procedure ALU1                               -- perform operation on single operand
  (signal op1: inout std_logic_vector(7 downto 0);
   signal N, Z, C : inout std_logic) is
variable res9 : std_logic_vector(8 downto 0);
variable res8 : std_logic_vector(7 downto 0);
begin
case OP is
  when NEG => res9 := not('0'&op1) + 1;
    C <= res9(8);  res8 := res9(7 downto 0);
  when COM => res8 := not op1;  C <= '1';
    when LSR => res8 := '0'&op1(7 downto 1); C <= op1(0);
  when RORx => res8 := C&op1(7 downto 1);  C <= op1(0);
  when ASR => res8 := op1(7)&op1(7 downto 1); C <= op1(0);
  when LSL => res8:= op1(6 downto 0)&'0'; C <= op1(7);
  when ROLX => res8 := op1(6 downto 0)&C; C <= op1(7);
  when DEC => res8 := op1 - 1;
  when INC => res8 := op1 + 1;
  when CLR => res8 := "00000000";
  when TST => res8 := op1;  C <= '0';
  when others => assert (false) report "illegal opcode";
end case;
op1 <= res8;  N <= res8(7);
if (res8 = "00000000") then Z <= '1'; else Z <= '0'; end if;
end ALU1;

Procedure fill_memory(signal mem: inout RAMType) is
  {insert procedure to fill memory here -- see Section 8.10}
end fill_memory;

begin
cpu_cycles: process
  variable reg_mem, hw_interrupt, BR: Boolean;
  variable sign_ext: std_logic_vector(4 downto 0);

begin
reg_mem:= (mode = imm) or (mode = dir) or (mode = ext) or (mode = ix) or
          (mode = ix1) or (mode = ix2);
hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');

```

```

wait until rising_edge(CLK);
if (rst_b = '0') then ST <= reset; fill_memory(mem);
else
case ST is
when reset => SP <= "111111";
  if (rst_b = '1') then ST <= cycle8; end if;
when fetch =>
  if reg_mem then ALU_OP(Md, A, X, N, Z, C); end if;
  -- complete previous operation
  if hw_interrupt then ST <= push1;
    else Opcode <= mem(CONV_INTEGER(PC)); PC <= PC+1; -- fetch opcode
    ST <= addr1; end if;

when addr1 =>
  case mode is
    when inha => ALU1(A, N, Z, C); ST <= fetch; -- do operation on A
    when inhx => ALU1(X, N, Z, C); ST <= fetch; -- do operation on X
    when imm => Md <= mem(CONV_INTEGER(PC)); -- get immediate data
    PC <= PC+1; ST <= fetch;
    when inh1 =>
      if OP = SWI then ST <= push1;
      elsif OP = RTS then ST <= pop2; SP <= SP+1;
      elsif OP = RTI then ST <= pop5; SP <= SP+1;
      end if;
    when inh2 =>
      case OP is
        when TAX => X <= A;
        when CLC => C <= '0';
        when SEC => C <= '1';
        when CLI => I <= '0';
        when SEI => I <= '1';
        when RSP => SP <= "111111";
        when TXA => A <= X;
        when others => assert (false)
          report "illegal opcode, mode = inh2";
      end case;
      ST <= fetch;
    when dir =>
      if OP = JMP then PC <= zero&mem(CONV_INTEGER(PC)); ST <= fetch;
      else MAR <= zero&mem(CONV_INTEGER(PC)); PC <= PC+1;
      -- get direct address
      if (OP=JSR) then ST <= push1; else ST <= data; end if;
    end if;
    when dirM => MAR <= zero&mem(CONV_INTEGER(PC)); PC <= PC+1;
    ST <= data;
  when ix =>
    if OP = JMP then PC <= zero&X; ST <= fetch;
    else MAR <= zero&X;
      if (OP=JSR) then ST <= push1; else ST <= data; end if;
    end if;

```

```

when ixm => MAR <= zero&X; ST <= data;
when ext | ix2 => MARH <= mem(CONV_INTEGER(PC))(4 downto 0);
    -- get high byte
    PC <= PC+1; ST <= addr2;
when ix1 | ix1m => MAR <= zero&mem(CONV_INTEGER(PC)); -- get offset
    PC <= PC+1; ST <= addX;
when rel => Md <= mem(CONV_INTEGER(PC));                      -- get offset
    PC <= PC+1; ST <= testBR;
when others => ST <= fetch;
    assert(false) report "address mode not implemented";
end case;

when addr2 =>
if (mode = ix2) then MARL <= mem(CONV_INTEGER(PC)); PC <= PC+1;
    -- get low byte
    ST <= addX;
elsif OP=JMP then PC <= MARH&mem(CONV_INTEGER(PC)); ST <= fetch;
else MARL <= mem(CONV_INTEGER(PC)); PC <= PC+1; -- get low byte
    if OP=JSR then ST <= push1; else ST <= data; end if;
end if;

when addX => if OP=JMP then PC <= MAR + (zero&X); ST <= fetch;
else MAR <= MAR + (zero&X);
    if OP=JSR then ST <= push1; else ST <= data; end if;
end if;

when data =>
if OP = STA then mem(CONV_INTEGER(MAR)) <= A; N <= A(7);
    if (A = "00000000") then Z <= '1'; else Z <= '0'; end if;
elsif OP = STX then mem(CONV_INTEGER(MAR)) <= X; N <= X(7);
    if X = "00000000" then Z <= '1'; else Z <= '0'; end if;
else Md <= mem(CONV_INTEGER(MAR));
end if;
if ((mode = dirM) or (mode = ixm) or (mode = ix1m)) then
    ST <= rd_mod_wr; else ST <= fetch; end if;

when rd_mod_wr => ALU1(Md, N, Z, C); ST <= writeback;
when writeback => mem(CONV_INTEGER(MAR)) <= Md; ST <= fetch;

when testBR =>
    case OP is
        when BRA => BR := TRUE;
        when BRN => BR := FALSE;
        when BHI => BR := (C or Z) = '0';
        when BLS => BR := (C or Z) = '1';
        when BCC => BR := C = '0';
        when BCS => BR := C = '1';
        when BNE => BR := Z = '0';
        when BEQ => BR := Z = '1';
        when BPL => BR := N = '0';
    end case;

```

```

when BMI => BR := N = '1';
when BMC => BR := I = '0';
when BMS => BR := I = '1';
when others => assert(false) report "illegal branch instruction";
end case;
if Md(7) = '1' then sign_ext := "11111"; else sign_ext := zero; end if;
if BR then PC <= PC + (sign_ext&Md); end if;
ST <= fetch;

when push1 => mem(CONV_INTEGER("0000011"&SP)) <= PCL;      -- push LO byte
SP <= SP - 1; ST <= push2;
when push2 => mem(CONV_INTEGER("0000011"&SP)) <= "000"&PCH;-- push HI byte
SP <= SP - 1;
if (hw_interrupt or OP = SWI) then ST <= push3;
else PC <= MAR; ST <= fetch; end if;                      -- JSR
when push3 => mem(CONV_INTEGER("0000011"&SP)) <= X;        -- push X
SP <= SP - 1; ST <= push4;
when push4 => mem(CONV_INTEGER("0000011"&SP)) <= A;        -- push A
SP <= SP - 1; ST <= push5;
when push5 => mem(CONV_INTEGER("0000011"&SP)) <= "0000"&CCR;-- push CCR
SP <= SP - 1; ST <= cycle8;
when cycle8 => I <= '1'; ST <= cycle9;
if OP = SWI then MAR <= "1111111111100";-- get interrupt vector addr.
elsif IRQ = '1' then MAR <= "1111111111010";
elsif SCint = '1' then MAR <= "1111111110110";
else MAR <= "1111111111111";           -- reset vector addr.
end if;
when cycle9 => PCH <= mem(CONV_INTEGER(MAR))(4 downto 0);
          -- get high byte
MAR <= MAR + 1; ST <= cycle10;
when cycle10 => PCL <= mem(CONV_INTEGER(MAR)); ST <= fetch;
          -- get low byte
when pop5 => CCR <= mem(CONV_INTEGER("0000011"&SP))(3 downto 0);
          -- pop CCR
SP <= SP + 1; ST <= pop4;
when pop4 => A <= mem(CONV_INTEGER("0000011"&SP));       -- pop A
SP <= SP + 1; ST <= pop3;
when pop3 => X <= mem(CONV_INTEGER("0000011"&SP));       -- pop X
SP <= SP + 1; ST <= pop2;
when pop2 =>
      PCH <= mem(CONV_INTEGER("0000011"&SP))(4 downto 0); -- pop HI byte
      SP <= SP + 1; ST <= pop1;
when pop1 => PCL <= mem(CONV_INTEGER("0000011"&SP));       -- pop LO byte
      ST <= fetch;
when others => null;
end case;
end if; -- if (rst_b = '1')
end process;
end behv;

```

## APPENDIX E

# M6805 CPU VHDL CODE FOR SYNTHESIS

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_bit: TO_INTEGER;
use std.textio.all;

entity cpu6805 is
port(clk, rst_b, IRQ, SCint: in std_logic;
dbus : inout std_logic_vector(7 downto 0);
abus : out std_logic_vector(12 downto 0);
wr: out std_logic);
end cpu6805;

architecture cpul of cpu6805 is
-- define registers
signal Opcode : bit_vector(7 downto 0) := (others => '0');
signal A, X, Md : std_logic_vector(7 downto 0) := (others => '0');
alias mode : bit_vector(3 downto 0) is Opcode(7 downto 4);
signal CCR: std_logic_vector(3 downto 0); -- CCR = I N Z C
alias I: std_logic is CCR(3); alias N : std_logic is CCR(2);
alias Z : std_logic is CCR(1); alias C : std_logic is CCR(0);
signal PC, MAR, xadd, xadd1, xadd2: std_logic_vector (12 downto 0);
signal SP: std_logic_vector(5 downto 0);
alias PCH : std_logic_vector(4 downto 0) is PC(12 downto 8);
alias PCL : std_logic_vector(7 downto 0) is PC(7 downto 0);
alias MARTH : std_logic_vector(4 downto 0) is MAR(12 downto 8);
alias MARL : std_logic_vector(7 downto 0) is MAR(7 downto 0);
signal va : std_logic_vector(2 downto 0);

type state_type is (reset, fetch, addr1, addr2, addX, data,
rd_mod_wr, writeback, testBR, push1, push2, push3, push4, push5,
cycle8, cycle9, cycle10, pop5, pop4, pop3, pop2, pop1);
signal ST, nST : state_type;
```

```
type decode_type is array(0 to 15) of bit_vector(15 downto 0);
signal opd : bit_vector(15 downto 0);
constant decode: decode_type :=
(X"0001", X"0002", X"0004", X"0008", X"0010", X"0020", X"0040", X"0080",
 X"0100", X"0200", X"0400", X"0800", X"1000", X"2000", X"4000", X"8000");
alias SUB : bit is opd(0); alias CMP : bit is opd(1);
alias SUBC : bit is opd(2); alias CPX : bit is opd(3);
alias ANDa : bit is opd(4); alias BITa : bit is opd(5);
alias LDA : bit is opd(6); alias STA : bit is opd(7);
alias EOR : bit is opd(8); alias ADC : bit is opd(9);
alias ORA : bit is opd(10); alias ADD : bit is opd(11);
alias LDX : bit is opd(14); alias STX : bit is opd(15);
alias BRA : bit is opd(0); alias BRN : bit is opd(1);
alias BHI : bit is opd(2); alias BLS : bit is opd(3);
alias BCC : bit is opd(4); alias BCS : bit is opd(5);
alias BNE : bit is opd(6); alias BEQ : bit is opd(7);
alias BPL : bit is opd(10); alias BMI : bit is opd(11);
alias BMC : bit is opd(12); alias BMS : bit is opd(13);
alias BIL : bit is opd(14); alias BIH : bit is opd(15);
alias TAX : bit is opd(7); alias CLC : bit is opd(8);
alias SEC : bit is opd(9); alias CLI : bit is opd(10);
alias SEI : bit is opd(11); alias RSP : bit is opd(12);
alias NOP : bit is opd(13); alias TXA : bit is opd(15);
alias NEG : bit is opd(0); alias COM : bit is opd(3);
alias LSR : bit is opd(4); alias ROR1 : bit is opd(6);
alias ASR : bit is opd(7); alias LSL : bit is opd(8);
alias ROL1 : bit is opd(9); alias DEC : bit is opd(10);
alias INC : bit is opd(12); alias TST : bit is opd(13);
alias CLR : bit is opd(15); alias RTI : bit is opd(0);
alias RTS : bit is opd(1); alias SWI : bit is opd(3);

-- define addressing mode = upper 4 bits of opcode
subtype ot is bit_vector(3 downto 0);
constant REL: ot:="0010"; constant DIRM: ot:="0011";
constant INHA: ot:="0100"; constant INHX: ot:="0101";
constant IX1M: ot:="0110"; constant IXM: ot:="0111";
constant INH1: ot:="1000"; constant INH2: ot:="1001";
constant IMM: ot:="1010"; constant DIR: ot:="1011";
constant EXT: ot:="1100"; constant IX2: ot:="1101";
constant IX1: ot:="1110"; constant IX: ot:="1111";
```

```
signal shiftout, op1_com, op2_com, op1, op2 : std_logic_vector(7 downto 0);
signal alu9 : std_logic_vector(8 downto 0);
alias Cout : std_logic is alu9(8);
signal shiftin, Cin, newC : std_logic;
signal com2, and2, or2, xor2, rsh, lsh, clear : std_logic;
signal incPC, xadd2PC, db2PCH, MARH2PCH, MARL2PCL, clrPCH, db2PCL, X2PCL :
    std_logic;
signal db2opcode, incSP, decSP, setSP, setSP1, setI, setI1, clrI :
    std_logic;
signal xadd2MAR, va2MAR, db2MARH, clrMARH, db2MARL, incMAR, X2MARL:
    std_logic;
signal ALU2A, db2A, ALU2X, db2X, db2Md, db2CCR, updateNZ, updateC :
    std_logic;
signal ALU2Md, Md2db : std_logic;
signal A2db, X2db, PCH2db, PCL2db, CCR2db, PC2ab, MAR2ab, SP2ab :
    std_logic;
signal com1, selA, selX, selMd1, selMd2, update : std_logic;
signal setC, clrC : std_logic;

constant hi_Z : std_logic_vector(7 downto 0) := (others => 'Z');
constant hi_Z13 : std_logic_vector(12 downto 0) := (others => 'Z');
constant zero: std_logic_vector(4 downto 0) := "00000";

begin

    -- drive the data bus with tristate buffers
    dbus <= A when A2db='1' else hi_Z;
    dbus <= X when X2db='1' else hi_Z;
    dbus <= Md when Md2db='1' else hi_Z;
    dbus <= "000"&PCH when PCH2db='1' else hi_Z;
    dbus <= PCL when PCL2db='1' else hi_Z;
    dbus <= "1110"&CCR when CCR2db='1' else hi_Z;

    -- drive the address bus
    abus <= MAR when MAR2ab='1'
        else "0000011"&SP when SP2ab='1'
        else PC;

    -- define ALU input operand MUXes
    op1 <= A when selA = '1'                                -- MUX for op1
        else X when selX = '1'
        else Md when selMd1 = '1'
        else "00000000";
    op2 <= Md when selMd2 = '1'                                -- MUX for op2
        else "00000000";
```

```

-- ALU and shifter operations
op1_com <= not op1 when com1='1' else op1;           -- complementers
op2_com <= not op2 when com2='1' else op2;
alu9 <= '0'&(op1_com and op2_com) when and2='1' -- adder/logic operations
      else '0'&(op1_com or op2_com) when or2='1'
      else '0'&(op1_com xor op2_com) when xor2='1'
      else ('0'&op1_com) + ('0'&op2_com) + Cin;
shiftout <= shiftin&alu9(7 downto 1) when rsh='1' -- shifter
      else alu9(6 downto 0)&shiftin when lsh='1'
      else "00000000" when clear = '1'
      else alu9(7 downto 0);
newC <= alu9(0) when rsh='1'                         -- carry logic
      else alu9(7) when lsh='1'
      else '1' when setC = '1'
      else '0' when clrC = '1'
      else Cout xor (com1 or com2);

xadd <= xadd1 + xadd2;                                -- address adder
opd <= decode(CONV_INTEGER(TO_INTEGER(Opcode(3 downto 0)))); -- operation
                                                               decoder

ALU_control: process (opd, mode, C, alu9)
-- this process generates control signals for ALU operations
  variable reg_mem, rd_md_wr : boolean := FALSE;
begin
  Cin <= '0'; shiftin<='0'; and2 <= '0'; or2 <= '0'; xor2 <= '0'; rsh<='0';
  lsh<='0'; com1 <= '0'; com2 <= '0'; updateNZ<='1'; updateC<='1'; clrI <= '0';
  setC <= '0'; clrC <= '0'; clear <= '0'; setI1<='0'; setSP1 <= '0';
  ALU2A <= '1'; ALU2X <= '0'; ALU2Md <= '0';
  selA <= '1'; selX <= '0'; selMd1 <= '0'; selMd2 <= '1';

  reg_mem:= (mode = imm) or (mode = dir) or (mode = ext) or (mode = ix) or
            (mode = ix1) or (mode = ix2);
  rd_md_wr := (mode = dirM) or (mode = inha) or (mode = inhx) or
            (mode = ix1m) or (mode = ixm);

  if reg_mem then          -- control signals for reg-mem ALU operations
    if ADD='1' then null; end if;      -- use defaults
    if ADC='1' then Cin<=C; end if;
    if SUB='1' then com2<='1'; Cin<='1'; end if;
    if SUBC='1' then com2<='1'; Cin<= not C; end if;
    if CMP='1' then com2 <= '1'; Cin <= '1'; ALU2A<='0'; end if;
    if CPX = '1' then selX <= '1'; selA <= '0'; com2 <= '1';
                      Cin <= '1'; ALU2A<='0'; end if;
    if ANDA='1' then and2<= '1'; updateC<='0'; end if;
    if BITA='1' then and2<= '1'; ALU2A <='0'; updateC<='0'; end if;
    if ORA='1' then or2 <= '1'; updateC<='0'; end if;
    if EOR='1' then xor2<= '1'; updateC<='0'; end if;
    if LDA ='1' then updateC<='0'; selA <= '0'; end if;
    if LDX ='1' then selA <= '0';
                      updateC<='0'; ALU2A<='0'; ALU2X<='1'; end if;
  end if;
end process;

```

```
if ((STA or STX) = '1') then ALU2A <= '0'; updateC <= '0'; end if;
    -- only update NZ flags
end if;

if rd_md_wr then
    -- control signals for rd_md_wr ALU/shifter operations
    selMd2 <= '0'; -- op2 is always zero for rd_md_wr
    if (mode /= inha) then
        ALU2A <= '0'; selA <= '0';                                -- turn off defaults
        if mode = inhx then ALU2X <= '1'; selX <= '1'; -- op1 = X
        else     ALU2Md <= '1'; selMd1 <= '1'; end if;      -- op1 = Md
    end if;

if NEG='1' then Cin<='1'; com1 <= '1'; end if;
if COM='1' then com1 <= '1'; end if;
if DEC ='1' then updateC<='0'; com2 <= '1'; end if; -- op2_com = -1
if LSR ='1' then rsh<= '1'; end if;
if ROR1 ='1' then rsh<='1'; shiftin<=C; end if;
if ASR ='1' then rsh<='1'; shiftin<=alu9(7); end if;
if LSL ='1' then lsh<='1'; end if;
if ROL1 ='1' then lsh<='1'; shiftin<=C; end if;
if INC ='1' then Cin<='1'; updateC<='0'; end if;
if CLR ='1' then and2 <= '1'; updateC<='0'; end if; -- and with 0 to
                                         clear
    if TST ='1' then updateC<='0'; end if;
end if; -- rd_md_wr

if (mode = inh2) then
    selMd2 <= '0'; updateC <= '0'; updateNZ <= '0'; -- op2 is zero
    -- A is always reloaded with A since ALU2A = '1'
    if TAX='1' then ALU2X<='1'; end if;
    if TXA='1' then selX <= '1'; selA <= '0'; end if;
    if CLC='1' then clrC <='1'; updateC <='1'; end if;
    if SEC='1' then setC <='1'; updateC <='1'; end if;
    if CLI='1' then clri<='1'; end if;
    if SEI='1' then setI1<='1'; end if;
    if RSP='1' then setSP1<='1'; end if;
end if;
end process;
```

```

CPU_control: process (ST, rst_b, opd, mode, IRQ, SCint, CCR, MAR, X,
PC, Md)
-- CPU state machine
variable reg_mem, hw_interrupt, BR : boolean;
begin
nST <= reset; BR := FALSE; wr <= '0'; update <= '0';
xadd1 <= (others => '0'); xadd2 <= (others => '0'); va <= "000";
db2A <= '0'; db2X <= '0'; db2Md <= '0'; db2CCR <= '0'; db2opcode <= '0';
incPC <= '0'; xadd2PC <= '0'; db2PCH <= '0'; MARH2PCH <= '0';
MARL2PCL <= '0';
clrPCH <= '0'; db2PCL <= '0'; X2PCL <= '0'; xadd2MAR <= '0';
va2MAR <= '0';
db2MARH <= '0'; clrMARH <= '0'; db2MARL <= '0'; X2MARL <= '0';
incMAR <= '0';
A2db <= '0'; X2db <= '0'; CCR2db <= '0'; PCH2db <= '0'; PCL2db <= '0';
Md2db <= '0'; MAR2ab <= '0'; PC2ab <= '0'; SP2ab <= '0'; incSP <= '0';
decSP <= '0'; setI <= '0'; setSP <= '0';

reg_mem:= (mode = imm) or (mode = dir) or (mode = ext) or (mode = ix) or
          (mode = ix1) or (mode = ix2);
hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');

if (rst_b = '0') then nST <= reset;
else
case ST is
when reset =>
    setSP <= '1';
    if (rst_b = '1') then nST <= cycle8; end if;
when fetch =>
    if (reg_mem and JMP = '0' and JSR = '0')
        then update <= '1'; end if; -- update registers if not JMP or JSR
    if hw_interrupt then nST <= push1;
        else PC2ab<='1'; db2opcode<='1'; incPC<='1'; -- read next opcode
            nST <= addr1; end if;

when addr1 =>
case mode is
    when inha | inhx => update <= '1'; nST<= fetch;
    when imm => PC2ab<= '1'; db2Md<='1'; incPC<='1';
        nST <= fetch;
    when inhl =>
        if SWI = '1' then nST <= push1;
        elsif RTS = '1' then nST <= pop2; incSP<='1';
        elsif RTI = '1' then nST <= pop5; incSP<='1';
        end if;

```

```

when inh2 => update <= '1'; nST <= fetch;
when dir => PC2ab<='1';
  if JMP='1' then db2PCL<='1'; clrPCH<='1'; nST<=fetch;
  else db2MARL<='1'; clrMARH<='1'; incPC <= '1';
    if JSR='1' then nST<=push1; else nST<=data; end if;
  end if;
when dirM => PC2ab<='1'; db2MARL<='1'; clrMARH<='1';
  incPC<='1'; nST<=data;
when ix =>
  if JMP='1' then X2PCL<='1'; clrPCH<='1'; nST<=fetch;
  else X2MARL<='1'; clrMARH<='1';
    if JSR='1' then nST<=push1; else nST<=data; end if;
  end if;
when ixm => X2MARL<='1'; clrMARH<='1'; nST<=data;
when ext | ix2 => PC2ab<='1'; db2MARH<='1';
  incPC<='1'; nST<=addr2;
when ix1 | ix1m => PC2ab<='1'; db2MARL<='1'; clrMARH<='1';
  incPC<='1'; nST<=addX;
when rel => PC2ab<='1'; db2Md<='1';
  incPC<='1'; nST<=testBR;
when others => null;
end case;

when addr2 =>      PC2ab<='1';
  if (mode = ix2) then db2MARL<='1';incPC<='1'; nST <= addX; --all [ix2]
  elsif (JMP = '1') then db2PCL<='1'; MARH2PCH<='1'; nST <= fetch;
    -- JMP [ext]
  else db2MARL<='1';incPC<='1';                                --JSR/others [ext]
    if (JSR = '1') then nST<=push1; else nST <= data; end if;
  end if;
when addX => xadd1<=MAR; xadd2<=zero&X;
  if JMP='1' then xadd2PC<='1';  nST <= fetch;
  else xadd2MAR<='1';
    if JSR='1' then nST<= push1; else nST<=data; end if;
  end if;

when data => MAR2ab<='1'; --nST <= fetch;
  if STA = '1' then wr<='1'; A2db<='1'; update <= '1'; -- update NZ flags
  elsif STX = '1' then wr<='1'; X2db<='1'; update <= '1'; -- update NZ flags
  else db2Md <= '1'; end if;                                     -- read from data bus
  if ((mode = dirM) or (mode = ixm) or (mode = ix1m)) then
    nST <= rd_mod_wr; else nST <= fetch; end if;

when rd_mod_wr => update <= '1'; nST <= writeback; -- update Md

when writeback => wr<='1'; MAR2ab<='1'; Md2db<='1'; -- write Md to memory
  nST <= fetch;

```

```

when testBR =>
  if BRA = '1' then BR := TRUE; end if;
  if BRN = '1' then BR := FALSE; end if;
  if BHI = '1' then BR := (C or Z) = '0'; end if;
  if BLS = '1' then BR := (C or Z) = '1'; end if;
  if BCC = '1' then BR := C = '0'; end if;
  if BCS = '1' then BR := C = '1'; end if;
  if BNE = '1' then BR := Z = '0'; end if;
  if BEQ = '1' then BR := Z = '1'; end if;
  if BPL = '1' then BR := N = '0'; end if;
  if BMI = '1' then BR := N = '1'; end if;
  if BMC = '1' then BR := I = '0'; end if;
  if BMS = '1' then BR := I = '1'; end if;
-- set inputs to address adder
xadd1<=PC; xadd2<=Md(7)&Md(7)&Md(7)&Md(7)&Md(7)&Md; -- sign extend Md
if BR then xadd2PC<='1'; end if;                                -- PC <= xadd1 + xadd2
nST <= fetch;

when push1 => wr<='1'; SP2ab<='1'; PCL2db<='1';
decSP <= '1'; nST <= push2;
when push2 => wr<='1'; SP2ab<='1'; PCH2db<='1'; decSP <= '1';
  if (hw_interrupt or SWI = '1') then nST <= push3;
  else MARH2PCH <= '1'; MARL2PCL <= '1'; nST <= fetch; end if;
-- PC <= MAR (execute JSR)
when push3 => wr<='1'; SP2ab<='1'; X2db<='1';
decSP <= '1'; nST <= push4;
when push4 => wr<='1'; SP2ab<='1'; A2db<='1';
decSP <= '1'; nST <= push5;
when push5 => wr<='1'; SP2ab<='1'; CCR2db<='1';
decSP <= '1'; nST <= cycle8;
when cycle8 =>
  if SWI = '1' then va <= "110"; -- 3 bits of interrupt vector addr
  elsif IRQ = '1' then va <= "101";
  elsif SCint = '1' then va <= "011";
  else va <= "111";                                -- default for reset vector
  end if;
  va2MAR <= '1'; setI <= '1'; nST <= cycle9;
when cycle9 => MAR2ab <= '1'; db2PCH <= '1'; -- get interrupt vector
incMAR <= '1'; nST <= cycle10;
when cycle10 => MAR2ab <= '1'; db2PCL <= '1';
nST <= fetch;
when pop5 => SP2ab<='1'; db2CCR<='1';           -- restore registers
  incSP<='1'; nST <=pop4;
when pop4 => SP2ab<='1'; db2A<='1';
  incSP<='1'; nST <=pop3;
when pop3 => SP2ab<='1'; db2X<='1';
  incSP<='1'; nST <=pop2;
when pop2 => SP2ab<='1'; db2PCH<='1';
  incSP<='1'; nST <=pop1;

```

```
when pop1 => SP2ab<='1'; db2PCL<='1';
    nST <=fetch ;
end case;
end if; -- if (rst_b = '0')

update_reg: process
begin
wait until CLK'event and CLK='1';
ST <= nST;
if incPC = '1' then PC <= PC + 1; end if;
if xadd2PC = '1' then PC <= xadd; end if;
if db2PCH = '1' then PCH <= dbus(4 downto 0); end if;
if MARH2PCH = '1' then PCH <= MARH; end if;
if MARL2PCL = '1' then PCL <= MARL; end if;
if clrPCH = '1' then PCH <= "00000"; end if;
if db2PCL = '1' then PCL <= dbus; end if;
if X2PCL = '1' then PCL <= X; end if;

if db2opcode= '1' then Opcode <= TO_BITVECTOR(dbus); end if;
if incSP = '1' then SP <= SP+1; end if;
if decSP = '1' then SP <= SP - 1; end if;
if (setSP = '1' or setSP1 = '1') then SP <= "111111"; end if;

if xadd2MAR = '1' then MAR <= xadd; end if;
if va2MAR = '1' then MAR <= "11111111"&va&'0'; end if;
if db2MARH = '1' then MARH <= dbus(4 downto 0); end if;
if clrMARH = '1' then MARH <= "00000"; end if;
if db2MARL = '1' then MARL <= dbus; end if;
if X2MARL = '1' then MARL <= X; end if;
if incMAR = '1' then MAR(0) <= '1'; end if;
-- MAR(0) is always '0' at this time so incrementer is not needed

if db2A='1' then A <= dbus; end if;
if db2X='1' then X<=dbus; end if;
if db2Md = '1' then Md <=dbus; end if;
if db2CCR = '1' then CCR<= dbus(3 downto 0); end if;

if (update = '1') then
    if (ALU2A = '1') then A <= Shiftout; end if;
    if (ALU2X = '1') then X <= Shiftout; end if;
    if (ALU2Md = '1') then Md <= Shiftout; end if;
    if updateNZ='1' then N <= Shiftout(7);
        if Shiftout = "00000000" then Z <= '1'; else Z <= '0'; end if;
    end if;
    if updateC='1' then C <= newC; end if;
end if;
if (setI = '1' or setIl = '1') then I <= '1'; end if;
if clrI = '1' then I <= '0'; end if;
end process;

end cpul;
```

# APPENDIX F

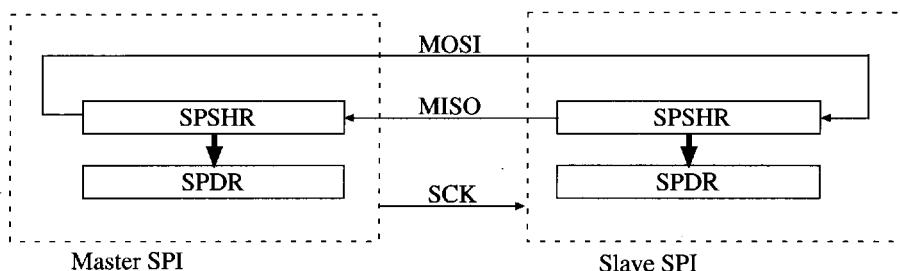
## PROJECTS

For each of these projects, choose an appropriate FPGA or CPLD as a target device and carry out the following steps:

1. Work out an overall design strategy for the system and draw block diagrams. Divide the system into modules if appropriate. Develop an algorithm, SM charts, or state graphs as appropriate for each module.
2. Write VHDL code for each module, simulate it, and debug it.
3. Integrate the VHDL code for the modules, simulate, and test the overall system.
4. Make any needed changes and synthesize the VHDL code for the target device. Simulate the system after synthesis.
5. Generate a bit file for the target device and download it. Verify that the hardware works correctly.

**P1** Design a push-button door lock that uses a standard telephone keypad as input. Use the keypad scanner designed in Chapter 3 as a module. The length of the combination is 4 to 7 digits. To unlock the door, enter the combination followed by the # key. As long as # is held down, the door will remain unlocked and can be opened. When # is released, the door is relocked. To change the combination, first enter the correct combination followed by the \* key. The lock is then in the “store” mode. The “store” indicator light comes on and remains on until the combination has been successfully changed. Next enter the new combination (4 to 7 digits) followed by #. Then enter the new combination a second time followed by #. If the second time does not match the first time, the new combination must be entered two times again. Store the combination in an array of eight 4-bit registers or in a small RAM. Store the 4-bit key codes followed by the code for the # key. Also provide a reset button that is not part of the keypad. When the reset button is pushed, the system enters the “store” state and a new combination may be entered. Use a separate counter for counting the inputs as they come in. A four-bit code, a key-down signal ( $Kd$ ), and a valid data signal ( $V$ ) are available from the keypad module.

**P2** Design an SPI (synchronous serial peripheral interface) module suitable for use with a microcontroller. The SPI allows synchronous serial communication with peripheral devices or with other microcontrollers. The SPI contains four registers—*SPCR* (SPI control), *SPSR* (SPI status), *SPDR* (SPI data), and *SPSHR* (SPI shift register). The following diagram shows how two SPIs can be connected for serial communications. One SPI operates as a master and one as a slave. The master provides the clock for synchronizing transmit and receive operations. When a byte of data is loaded into the master *SPSHR*, it initiates serial transmission and supplies a serial clock (*SCK*). Data is exchanged between the master and slave shift registers in 8 clocks. As soon as transmission is complete, data from each *SPSHR* is transferred to the corresponding *SPDR*, and the SPI flag (*SPIF*) in the *SPSR* is set.



The function of the pins depends on whether the device is in master or slave mode:

*MOSI*—output for master, input for slave

*MISO*—input for master, output for slave

*SCK*—output for master, input for slave

The *SPDR* and *SPSHR* are mapped to the same address. Reading from this address reads the *SPDR*, but writing loads the *SPSHR*. *SPSR* bit 7 is the SPI flag (*SPIF*). *SPSR* may also contain error flags, but we will omit them from this design. The following sequence will clear *SPIF*:

Read *SPSR* when *SPIF* is set.

Read or write to the *SPDR* address.

The *SPCR* register contains the following bits:

*SPIE*—enable SPI interrupt

*SPE*—enable the SPI

*MSTR*—set to '1' for master mode, '0' for slave mode

*SPR1* and *SPR0*—set *SCLK* rate as follows:

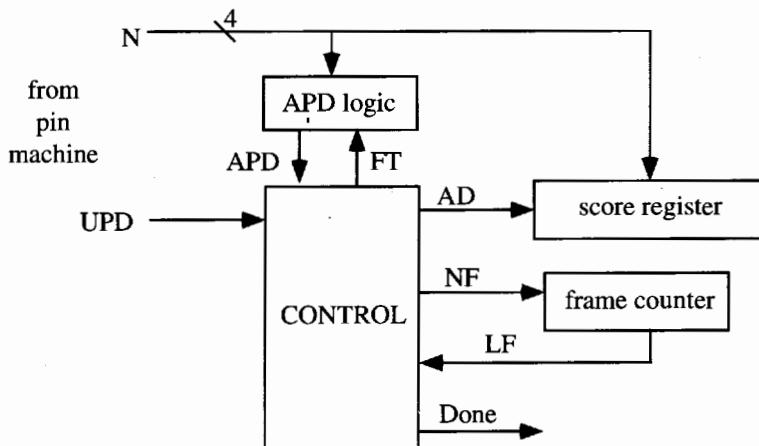
*SPR1&SPR0 = 00*    *SCK rate = Sysclk rate /2*

*SPR1&SPR0 = 01*    *SCK rate = Sysclk rate /4*

*SPR1&SPR0 = 10*    *SCK rate = Sysclk rate /16*

*SPR1&SPR0 = 11*    *SCK rate = Sysclk rate /32*

**P3** The digital system shown below will be used to keep score for a bowling game. The score keeping system will score the game according to the following (regular) rules of bowling: A game of bowling is divided into ten frames. During each frame, the player gets two tries to knock down all of the bowling pins. At the beginning of a frame, ten pins are set up. If the bowler knocks all ten pins down on his or her first throw, then the frame is scored as a *strike*. If some (or all) of the pins remain standing after the first throw, the bowler gets a second try. If the bowler knocks down all of the pins on the second try, the frame is scored as a *spare*. Otherwise, the frame is scored as the total number of pins knocked down during that frame.



The total score for a game is the sum of the number of pins knocked down plus bonuses for scoring strikes and spares. A strike is worth 10 points (for knocking down all ten pins) plus the number of pins knocked down on the next two *throws* (not frames). A spare is worth 10 points (for knocking down ten pins) plus the number of pins knocked down on the next throw. If the bowler gets a spare on the tenth frame, then he/she gets one more throw. The number of pins knocked down from this extra throw are added to the current score to get the final score. If the bowler gets a strike on the last frame, then he/she gets two more throws, and the number of pins knocked down are added to the score. If the bowler gets a strike in frame 9 and 10, then he/she also gets two more throws, but the score from the first bonus throw is added into the total *twice* (once for the strike in frame 9, once for the strike in frame 10), and the second bonus throw is added in once. The maximum score for a perfect game (all strikes) is 300. An example of bowling game scoring follows:

Frame	First Throw	Second Throw	Result	Score
1	3	4	7	7
2	5	5	spare	$7 + 10 = 17$
3	7	1	8	$17 + 7$ (bonus for spare in 2) + 8 = 32
...	...	...	...	87
9	10	-	strike	$87 + 10 = 97$
10	10	-	strike	$97 + 10$ (for this throw) + 10 (bonus for strike in 9)
-	6	3	-	$117 + 6$ (bonus for stroke in 9) + 6 (bonus for strike in 10) + 3 ( bonus for strike in 10) = 32

The score keeping system has the form shown above. The control network has three inputs: *APD* (All Pins Down), *LF* (Last Frame), and *UPD* (update). *APD* is 1 if the bowler has knocked all ten pins down (in either one or two throws). *LF* is 1 if the frame counter is in state 9 (frame 10). *UPD* is a signal to the network that causes it to update the score. *UPD* is 1 for exactly one clock cycle after every throw the bowler makes. There are many clock cycles between updates.

The control network has four outputs: *AD*, *NF*, *FT*, and *Done*. *N* represents the number of pins knocked down on the current throw. If *AD* is 1, *N* will be added to the score register on the rising edge of the next clock. If *NF* is 1, the frame counter will increment on the rising edge of the next clock. *FT* is 1 when the first throw in a frame is made. *Done* should be set to 1 when all ten frames and bonus throws, if applicable, are complete.

Use a 10-bit score register and keep the score in BCD form rather than in binary. That is, a score of 197 would be represented as 01 1001 0111. The lower two decimal digits of the register should be displayed using two 7-segment LED indicators, and the upper two bits can be connected to two single LEDs. When *ADD* = 1 and the register is clocked, *N* should be added to the register. *N* is a 4-bit binary number in the range 0 through 10. Use a 4-bit BCD counter module for the middle BCD digit. Note that in the lower four bits, you will add a binary number to a BCD digit to give a BCD digit and a carry.

**P4** Design a simple microcomputer for 8-bit signed binary numbers. Use a keypad for data entry and a 256 X 8 static RAM memory. The microcomputer should have the following 8-bit registers: *A* (accumulator), *B* (multiplier), *MDR* (memory data register), *PC* (program counter), and *MAR* (memory address register). The *IR* (instruction register) may be 5 to 8 bits, depending on how the instructions are encoded. The *B* register is connected to the *A* register so that *A* and *B* can be shifted together during the multiply. Only one 8-bit adder and one completer is allowed. The microcomputer should have a 256-word-by-8-bit memory for storing instructions and data. It should have two modes: (a) memory load and (b) execute program. Use a DIP switch to select the mode.

Memory load mode operates as follows: Select mode = 0 and reset the system. Then press two keys on the keypad followed by pushing a button to load each word in memory. The first word is loaded at address 0, the second word at address 1, etc. Data should be

loaded immediately following the program. Execution mode operates as follows: Select mode = 1 and press reset. Execution begins with the instruction at address 0.

Each instruction will be one or two words long. The first word will be the opcode, and the second word (if any) will be an 8-bit memory address or immediate operand. One bit in the opcode should distinguish between memory address or immediate operand mode. Represent negative numbers in 2's complement. Implement the following instructions:

LDA <memadd>	load A from the specified memory address
LDA <imm>	load A with immediate data
STA <memadd>	store A at the specified memory address
ADD <memadd>	add data from memory address to A, set carry flag if carry, set V if 2's complement overflow
ADD <imm>	add immediate data to A, set carry flag if carry, set V if overflow
SUB <memadd>	subtract data from memory address from A, set carry flag if borrow, set V if 2's complement overflow
SUB <imm>	subtract immediate data from A, set carry flag if borrow, set V if overflow
MUL <memadd>	multiply data from memory address by B, result in A & B
MUL <imm>	multiply immediate data by B
SWAP	swap A and B
PAUSE	pause until a button is pressed and released <i>(note: A register should always be displayed on LEDs.)</i>
JZ <target addr>	jump to target address if A = 0
JC <target addr>	jump to target address if carry flag (CF) is set
JV <target addr>	jump to target address if overflow flag (V) is set

The control module should be implemented as a linked state machine, with a separate state machine for the multiplier control. Try to keep the number of states small. (A good solution should have about ten states for the main control.) The multiplier control should use a separate counter to count the number of shifts. Assume that the clock speed is slow enough so that memory can be accessed in one clock period.

**P5** Design a stack-based calculator for 8-bit signed binary numbers. Input data to the calculator can come from a keypad or from DIP switches with a separate push-button to enter the data. The calculator should have the following operations:

enter	push the 8-bit input data onto the stack
0 - clear	clear the top of the stack, reset the stack counter, reset overflow, etc.
1 - add	replace the top two data entries on the stack with their sum
2 - sub	replace the top two data entries on the stack with their difference (stack top – next entry)
3 - mul	replace the top two data entries on the stack with their product (8 bits × 8 bits to give 8-bit product)
4 - div	replace the top two data entries on the stack with their quotient (stack top / next entry) (8 bits divided by 8 bits to give 8 bit quotient)

- |          |  |
|----------|--|
| 5 - xchg | exchange the top two data entries on the stack       |
| 6 - neg  | replace the top of the stack with its 2's complement |

Negative numbers should be represented in 2's complement. Provide an overflow indicator for 2's complement overflow. This indicator should be also be set if the product requires more than 8 bits including sign or if divide by 0 is attempted.

Implement a stack module that has four 8-bit words. The stack should have the following operations: push, pop, and exchange the top two words on the stack. The top of the stack should always be displayed on eight LEDs. Include an indicator for stack overflow (attempt to push a fifth word) and stack underflow (attempt to pop an empty stack or to exchange the top of stack with an empty location).

Design the control unit for the calculator using linked state machines. Draw a main SM chart with separate SM charts for the multiplier and divider control. When you design the arithmetic unit, try to avoid adding unnecessary registers. You should be able to implement the arithmetic unit with three registers (8 or 9 bits each), an adder, two complementers, etc.

**P6** Design a floating-point arithmetic unit. Each floating-point number should have a 4-bit fraction and a 4-bit exponent, with negative numbers represented in 2's complement. (This is the notation used in the examples in Chapter 7.) The unit should accept the following floating-point instructions:

- 001 FPL—load floating-point accumulator (fraction and exponent)
- 010 FPA—add floating-point operand to accumulator
- 011 FBS—subtract floating-point operand from accumulator
- 100 FPM—multiply accumulator by floating-point operand
- 101 FPD—(optional) divide floating-point accumulator by floating-point operand

The result of each operation (4-bit fraction and 4-bit exponent) should be in the floating-point accumulator. All output should be properly normalized. The accumulator should always be displayed as hex digits on 7-segment LEDs. Use an LED to indicate an overflow.

The input to the floating-point unit will come from a  $4 \times 4$  hexadecimal keypad, using a scanner similar to the one designed in Chapter 3. Each instruction will be represented by three hex digits from the keypad—the opcode, the fraction, and the exponent. For example,  $FPA\ 1.011 \times 2^{-3}$  is coded as 2 B D = 0010 1011 1101. Assume that all inputs are properly normalized or zero. Your design should include the following modules: fraction unit, exponent unit, control module, and 4-bit binary to 7-segment display conversion logic.

## REFERENCES

---

References 14, 15, 19, 29, and 32 are general references on digital logic and digital system design. References 2, 18, 30, 31, and 33 provide information on PLDs, FPGAs, and CPLDs. References 8, 12, 15, 20, 21, 28, and 31 provide a basic introduction to VHDL. References 3, 4, 5, 6, 7, 11, 13, 16, 17, 25, and 26 cover more advanced VHDL topics. References 1, 9, 24, and 27 relate to hardware testing and design for testability. The M6805 microcontroller family is described in references 22 and 23.

1. Abromovici, M., Breuer, M., and Frideman, F. *Digital Systems Testing and Testable Design*. Los Alamitos, Calif.: Computer Science Press, 1990.
2. Altera Corporation, *Altera Data Book*, 1996. (<http://www.altera.com>).
3. Armstrong, James, and Gary, G. *Structured Logic Design with VHDL*. Upper Saddle River, N.J.: Prentice Hall, 1993.
4. Ashenden, Peter J. *The Designer's Guide to VHDL*. San Francisco, Calif.: Morgan Kaufmann, 1996.
5. Baker, Louis. *VHDL Programming: With Advanced Topics*. Upper Saddle River, N.J.: Prentice Hall, 1993.
6. Berge, F., and Maginot, R. J. *VHDL Designer's Reference*. Boston, Mass.: Kluwer Academic Publishers, 1992.
7. Bhasker, J. *A Guide to VHDL Syntax*. Upper Saddle River, N.J.: Prentice Hall, 1995.
8. Bhasker, J. *A VHDL Primer*, rev. ed. Upper Saddle River, N.J.: Prentice Hall, 1992.
9. Bleeker, H., van den Eijnden, P., and de Jong, Frans. *Boundary Scan Test—A Practical Approach*. Boston, Mass.: Kluwer Academic Publishers, 1993.
10. Brayton, Robert K. et al. *Logic Minimization for Algorithms for VLSI Synthesis*. Boston, Mass.: Kluwer Academic Publishers, 1984.

11. Chang, K.C. *Digital Design and Modeling with VHDL and Synthesis*. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.
12. Coelho, David R. *The VHDL Handbook*. Boston, Mass.: Kluwer Academic Publishers, 1989.
13. Cohen, Ben. *VHDL—Coding Styles and Methodologies*. Boston, Mass.: Kluwer Academic Publishers, 1995.
14. Comer, David J. *Digital Logic and State Machine Design*, 3rd ed. New York: Saunders, 1995.
15. Dewey, Allen. *Analysis and Design of Digital Systems with VHDL*. Boston, Mass.: PWS, 1997.
16. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std\_logic\_1164)*. New York: The Institute of Electrical and Electronics Engineers, 1993.
17. *IEEE Standard VHDL Language Reference Manual*. New York: The Institute of Electrical and Electronics Engineers, 1993.
18. Jenkins, Jesse H. *Designing with FPGAs and CPLDs*. Upper Saddle River, N.J.: Prentice Hall, 1994.
19. Katz, Randy H. *Contemporary Logic Design*. Menlo Park, Calif.: Benjamin/Cummings, 1994.
20. Lipsett, Roger, Schaefer, Carl F., and Ussery, Cary. *VHDL: Hardware Description & Design*. Boston, Mass.: Kluwer Academic Publishers, 1989.
21. Mazor, Stanley, and Langstraat, Patricia. *A Guide to VHDL*, 2d ed. Boston, Mass.: Kluwer Academic Publishers, 1993.
22. *MC68HC05 Microcontroller Applications Guide*. Phoenix, Ariz.: Motorola, Inc., 1995.
23. *MC68HC705C4A Technical Data*. Phoenix, Ariz.: Motorola, Inc., 1996.
24. McCluskey, E. J. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Upper Saddle River, N.J.: Prentice Hall, 1986.
25. Navabi, Zainalabedin. *VHDL—Analysis and Modeling of Digital Systems*. New York: McGraw-Hill, 1993.
26. Ott, Douglas E., and Wilderotter, Thomas J. *A Designer's Guide to VHDL Synthesis*. Boston, Mass.: Kluwer Academic Publishers, 1994.

27. Parker, Kenneth P. *The Boundary Scan Handbook*. Boston, Mass.: Kluwer Academic Publishers, 1992.
28. Perry, Douglas. *VHDL*, 2d ed. New York: McGraw-Hill, 1994.
29. Roth, Charles H. *Fundamentals of Logic Design*, 4th ed. Boston, Mass.: PWS, 1992.
30. Salcic, C., and Smailagic, A. *Digital Systems Design and Prototyping Using Field Programmable Logic*. Boston, Mass.: Kluwer Academic Publishers, 1997. (Includes VHDL software for Altera products.)
31. Skahill, Kenneth, and Cypress Semiconductor. *VHDL for Programmable Logic*. Reading, Mass.: Addison-Wesley, 1996. (Includes VHDL software for Cypress products.)
32. Wakerly, John F. *Digital Design Principles and Practices*. Upper Saddle River, N.J.: Prentice Hall, 1990.
33. XILINX, Inc. *The Programmable Logic Data Book*, 1996. (<http://www.xilinx.com>).

# INDEX

- 16R4 sequential PAL, 98-100, 182  
22CEV10 (22V10) PLD  
    implementations using, 107, 108-109, 113-114  
    structure, 101-103  
486 bus. *See* bus
- A**
- active-low signal, 303  
active-high signal, 303  
adder  
    FPGA implementation, 220-222  
    full, 2-3, 45-46  
    parallel, 107-109  
    serial, 121-123  
    VHDL model for a 4-bit, 48  
adder-subtractor, parallel, 205-207  
addition  
    bit-vectors, 76  
    floating-point, 259-260  
    VHDL model with operator overloading, 271  
address transition, 321  
addressing modes, 392-393  
algebraic simplification, 3-7  
alias, 127, 420  
Altera CPLD. *See* CPLD  
alternate clock buffer, 211  
AND  
    function for std\_logic\_vectors, 278  
    gate, 1  
        testing for stuck-at faults, 340-343  
        VHDL model, 429  
    table for IEEE 9-valued logic, 277  
    table for X01Z logic, 275  
architecture, 45-47  
    declaration, 46, 420  
array, 68-70  
    attributes, 267  
    constrained, 68  
    types, 68-70, 420  
    unconstrained, 69  
ASM chart. *See* SM chart  
assert statement, 116, 422  
    use in error checking, 267  
associative laws, 4, 7  
asynchronous design, 34  
attributes, 265-268
- B**
- baud rate generator, 383-384  
BCD (binary-coded-decimal), 18  
bed-of-nails tests, 351  
behavioral modeling in VHDL, 58  
BILBO. *See* built-in logic block observation  
binary-coded-decimal (BCD), 18  
BIST. *See* Built-In Self-Test  
bit package (bit\_pack), 77, 425-433  
bit-vector, 70  
    addition, 75  
        VHDL procedure for addition, 268  
bit\_overload package, 271  
Boolean algebra, 1-7  
boundary scan testing, 351-360  
    VHDL code for test example, 359  
    VHDL model for boundary scan, 357  
bridging fault, 343  
bubble, definition, 10  
buffer, 296  
built-in logic block observation (BILBO), 364-370  
    VHDL model of BILBO register, 367  
    VHDL model of BILBO tester system, 368  
    VHDL test bench for tester system, 389  
Built-In Self-Test (BIST), 361-370

bus, 35-36, 210, 252  
 486 bus model, 316-324  
   2-2 bus cycle, 317  
   3-3 bus cycle, 318  
   SM chart for bus interface unit, 320  
   timing specifications, 320-321  
   VHDL model, 322  
 system with interfacing to RAM, 325-334  
   test data and results, 333-334  
   VHDL model for memory controller, 329  
   VHDL model, complete, 331  
   VHDL test module, 330

**C**

carry logic, 219-224  
 case statement, 55-56, 421  
 CLB (configurable logic block)  
   Xilinx 3000 series, 201-203  
   Xilinx 4000 series, 219  
 clock, VHDL model, 45  
 CMOS PLDs, 102  
 code converter  
   BCD to excess-3, 18-22, 99-100  
   serial data (NRZ to Manchester), 24  
 codes (NRZ, NRZI, RZ, Manchester), 23  
 combinational logic  
   general, 1-3  
   testing, 339-343  
   VHDL descriptions, 44-49  
 Combinatorial Function block, 202-207  
 commutative laws, 4, 7  
 compilation, 56-58  
 complex programmable logic device. *See* CPLD  
 component declaration, 48, 77, 423  
 component instantiation, 48, 423  
 concurrent statements, definition, 44  
 conditional assignment statement, 55, 419  
 configurable logic block. *See* CLB  
 configuration memory cells, 201-203  
 consensus theorem, 4, 5, 6  
 constants, 65  
   declaration, 420  
 contact bounce, 110  
 control circuit, definition, 121  
 core logic, 351-352, 354-355  
 counter  
   74163, 78-80  
   VHDL model, 78-79, 433  
 use as state register, 187  
 VHDL model for dice game, 183

CPLD (complex programmable logic device)

  Altera FLEX 10K series 235-239  
     realizing functions, 237  
   Altera MAX 7000 series 231-235  
     logic expanders, 233  
     realizing functions 231-235  
   Altera MAX 7000S series, 235  
   Altera MAX 9000 series, 235  
 crystal oscillator for FPGA, 211

**D**

D flip-flop. *See* flip-flop  
 data flow modeling in VHDL, 61  
 data line, 303  
 data transition, 321  
 debouncing and synchronizing circuit, 110-111  
 DeMorgan's laws, 3, 4  
 dice game, 168-177  
   implementation with XC3020 FPGA, 213-216  
   implementation with PAL, 180-184  
 SM chart, 172  
 synthesis, 292  
 test bench, 174-177  
 VHDL behavioral model, 173  
 VHDL data flow model, 182  
 digital system, 31-32  
 distributive laws, 4, 6, 7  
 divider, parallel for signed binary, 148-155  
   test bench, 153  
   VHDL model, 151  
 divider, parallel for unsigned binary, 144-148  
 division  
   floating-point, 260  
   signed binary, 148-155  
   unsigned binary, 144-146  
 don't cares  
   in Karnaugh maps, 7-10  
   on state graphs, 123  
 driver, 57-58, 273-274  
 duality, 4

**E**

EA (effective address), 392  
 EAB (embedded array block), 236, 239  
 EEPROM (electrically erasable programmable ROM), 86  
 effective address (EA), 392  
 elaboration, 56-57  
 embedded array block (EAB), 236, 239  
 entity, 45-47

declaration, 46, 420  
enumeration type, 67, 420

EPROM (erasable programmable ROM), 86  
use with FPGAs, 212  
equivalent states, 25-27  
error checking, 267  
essential prime implicant, 8-9  
event, definition, 58  
excess-3 coded decimal, 18  
exclusive-OR  
definition, 1  
theorems, 6-7  
VHDL gate model, 432  
exit statement, 73, 422  
exponent overflow. *See* overflow

## F

falling-edge device, 32  
FastTrack Interconnect, 236  
fault detection, 339-343  
field programmable gate array. *See* FPGA  
file declaration, 295, 423  
files  
input and output, 295-299  
VHDL example, 299  
flip-flop, 14-17  
with clock enable, 205  
D, 14  
J-K, 15  
S-R, 16  
T, 15  
flip-flop, equation derivation  
D, 20-21  
J-K, 20-22  
for SM charts, 178  
flip-flops, VHDL models, 50-54  
D, 51, 432  
J-K, 52, 432  
floating-point arithmetic, 243-261  
addition, 259  
division, 260  
multiplication, 244-259  
subtraction, 260

floating-point numbers  
normalization, 243-244  
representation, 243-244  
for loop, 72, 421  
fourpack package, 274-275  
FPGA (field programmable gate array), 201-230  
configuration from EPROM, 212

designing with, 211-219  
global clock buffer, 211  
implementation of adder-subtractor, 205-207  
implementation of dice game, 213-216  
Xilinx 3000 series, 201-211  
F, G, and FG modes, 203-204  
realizing 6-, 7-variable functions, 216-219  
realizing functions, 202-204  
Xilinx 4000 series, 219-229  
RAM implementation, 223  
realizing functions, 219  
VHDL model, 224-227  
fractional overflow. *See* overflow  
full adder, 2-3, 45-46  
function, 72-74, 422

## G

gated control signal, 32-34  
generate statements, 282-283, 423  
generic map, 331  
generics, 280-281  
VHDL example, 281  
glitch, 28, 32

## H

hardware testing. *See* testing  
hazards, 13-14  
hold time, 29-31  
    checking, 267

## I

I/O control block, 235  
idempotent law, 4  
identifiers, 45  
IEEE-1164 standard logic. *See* standard logic  
if statement, 51, 53, 421  
implication table, 26-27  
in, definition, 75  
inertial delays, 269  
inout, definition, 75  
input-output element (IOE), 236  
input-output interface block, 201  
interface signals, 46, 420  
interrupts, 393  
inverter, 1  
    VHDL model, 432  
IOE (input-output element), 236  
iterative network, 344

**J**

J-K flip-flop. *See* flip-flop

**K**

Karnaugh maps, 7-10

  4-variable, 7

  map-entered variables $\leftarrow \alpha \% 10$

  use in reducing PLA tables, 93-94

keypad scanner, 109-117

  debouncing and synchronizing circuit, 110-111

  decoder, 112-113

  keyscan, 111-112

  VHDL model, 115-117

**L**

LAB (logic array block)

  Altera FLEX 10K series, 236

  Altera MAX 7000 series, 231

latch, 16-17

  unintended creation from VHDL synthesis, 284

LE (logic element), 236-238

library declaration, 422

libraries, 76-77

linear feedback shift registers (LFSR), 362-364

  with signature register (MISR), 362

link path, 162

linked state machines, 190-193

logic array block. *See* LAB

logic cell array (LCA), 201

logic element (LE), 236-238

logic expander, 231, 233

long lines, 209-210

**M**

M68HC05 (M6805) *See also* microcontroller

  behavioral VHDL code, 437-442

  description, 387-393

  VHDL code for synthesis, 443-451

macrocell, 231-235

map-entered variables, 9-10

  for dice game controller, 180

MAR (memory address register), 309

mask programmable ROM, 86

maxterm, definition, 3

maxterm expansion, 3

Mealy sequential network, 17-23

  definition, 17

  implementation with ROM, 86-89

  timing, 28-29

Mealy sequential network, VHDL models

  with 1 process, 63-65

  with 2 processes, 58-60

  with arrays, 69-70

  data-flow, 61

  with PLA, 95-96

  with ROM, 88-89

  structural model, 61-62

memory address register (MAR), 309

memory controller, 327-328

microcontroller (including VHDL models), 387-415

  complete design, 413-415

  CPU controller, 398-404

  description of operation, 381-398

  hardware design, 404-411

  parallel ports, 411-413

microprocessor bus interface, 316-324

microprogramming, 184-190

minimum product-of-sums from Karnaugh maps, 9

minimum sum-of-products from Karnaugh maps, 8-10

minterm, definition, 3

minterm expansion, 3

MISR (multiple-input signature register), 362, 364

modes, 46

modulo-6 counter, 216

Moore sequential network, 23-25

  definition, 17

  timing, 25

  VHDL behavioral model, 104-107

multiple-input signature register (MISR), 362

multiple-valued logic, 95-96

multiplexer

  implementation with tristate buffer, 210

  in Combinatorial Function block, 204-205

  to replace tristate buffers for bus, 252

  use with microprogramming, 184-190

  VHDL models, 54-55

  in Xilinx 4000 series CLB, 219

multiplicand, definition, 124

multiplication

  floating-point, 244

  signed binary, 132-134

  unsigned binary, 124-125

multiplier, definition, 124

multiplier, 4x4 array multiplier, 131-132

multiplier, floating-point, 245-259

  simulation results, 252

SM chart for main controller, 248

state graph for multiplier control, 249

synthesis, 292-295

VHDL behavioral model, 249-252

- multiplier, for signed fractions, 134-135  
 faster, 135-138
  - VHDL behavioral model, 136
  - VHDL model using control equations, 143
  - VHDL model with control signals, 141
 multiplier, unsigned binary, 125-130
  - with counter, 128-130
  - without counter, 125-128
  - FPGA implementation, 227-229
  - PLA table, 179
  - ROM table, 179
  - SM chart, 167
  - VHDL models, 127, 168
 multivalued logic for VHDL, 272-281
 MVLLIB, 280
- N**
- NAND gate, 11
    - networks, 10-12
    - VHDL model, 281, 431
  - NATURAL subtype, 70
  - negative logic, definition, 1
  - noise, 32-33
  - NOR gate, 11
    - VHDL model, 431
  - NOR networks, 10-12
  - nMOS PLA, 90-91
  - NOR-NOR logic in PLA, 90-91
  - normalization for floating-point, 243-244
  - NOT gate. *See* inverter
  - now, definition, 324
  - numeric\_bit package 286, 288
  - numeric\_std package 286, 288
- O**
- offset, 392
  - one-hot state assignment, 229, 255
  - operator overloading, 270-272, 288-289
  - operators, 70-71
  - OR
    - gate, 1
      - VHDL model, 430
    - table for X01Z logic, 275
    - testing for stuck-at faults, 340-343
  - out, definition, 75
  - output macrocell, 101-103
  - overflow
    - signed division, 149
    - signed exponent, 245, 247, 260
    - signed fractional, 244, 247, 259, 260
- unsigned division, 146
- overloaded operators, 270-272, 288-289
- P**
- packages, 76-77, 423
  - PAL (programmable array logic), 96-100
    - combinational, 97-98
    - for dice game controller, 180
    - sequential, 98-100
    - use with microprogramming, 184-190
  - parallel adder with accumulator, 107-109
  - parallel expanders, 233
  - parallel ports, 411-413
  - parity, 373
  - partitioning program, 212
  - PGA (programmable gate arrays). *See* FPGA
  - PLA (programmable logic arrays), 89-96
    - AND-OR array structure, 90-92, 94
    - for dice game controller, 180
    - use with microprogramming, 184-190
    - VHDL model for PLA output, 278-280
  - PLA table, 92-95
    - for dice game controller, 181, 185, 189
    - for multiplier control, 179
    - reduction of, 93-94
  - place and route program, 212
  - PLD (programmable logic device), 85-118
    - other sequential devices, 101-103
    - programmable array logic (PAL), 96-100
    - programmable logic arrays (PLA), 89-96
    - read-only memories (ROM), 85-89
  - port, 45-46
  - port map, 48, 331
  - positive logic, definition, 1
  - POSITIVE subtype, 70
  - prime implicant, 8-9
  - procedure, 74-76, 422
  - process statement,
    - with sensitivity list, 50, 421
    - without sensitivity list, 421
      - with wait statements, 63, 66
  - programmable array logic. *See* PAL
  - programmable gate arrays. *See* FPGA
  - programmable interconnect matrix, 231
  - programmable interconnects, 209
  - programmable logic arrays. *See* PLA
  - programmable logic devices. *See* PLD
  - projects, 457
  - propagation delay, 29, 326-327
  - pseudo-random pattern generator (PRPG), 363

**R**

race, 34  
**RAM** (random-access memory), 223  
  in Altera FLEX 10K, 236  
  CMOS  
    43258A-25 RAM, 306, 326  
    6116-2 RAM, 304-316  
  dynamic, definition, 304  
  interface to microprocessor bus. *See bus*  
  read cycle timing, 305  
  static, 303-316  
    definition, 304  
  test of VHDL model with timing, 314  
  test of VHDL model without timing, 309-311  
  testing by use of BIST, 361, 362  
  truth table, 304  
  VHDL model with timing, 312  
  VHDL model without timing, 308  
  write cycle timing, 307-308  
**read-only memories.** *See ROM*  
**redundant terms**, definition, 5, 6  
**register memory instruction**, 392  
**report statement**, 116, 422  
**reserved words**, definition, 45  
**resolution function** for VHDL, 272  
  for IEEE 9-valued logic, 277  
  for X01Z type, 274  
**rising-edge device**, 32-33  
**rising-edge function** for standard logic, 278  
**ROM** (read-only memory), 85-89  
  testing by use of BIST, 361, 362  
  use with microprogramming, 184-190  
**ROM truth table**, 87-88  
  for multiplier control, 179  
**row matching**, 26

**S**

**selected signal assignment statement**, 55, 419  
**sensitivity list**, 50  
**sequential logic**, definition, 1  
**sequential network**  
  asynchronous design, 34  
  synchronous design, 31-34  
**sequential statements**, 50-51  
**serial adder.** *See adder, serial*  
**serial data port**, 373  
**serial data transmission**, 373-374  
**serial-parallel multiplier.** *See multiplier, serial-parallel*  
**setup time**, 29-31  
  checking, 267

**Shannon's expansion theorem**, 217  
**sharable expanders**, 233  
**signal**, 65-67  
  assignment statement, 44, 419  
  attributes, 265-266  
  declaration, 65, 420  
  definition, 44  
  in process, 66  
**signal resolution**, 272-275  
**signature**, 362  
**signed type**, 286, 288  
**simulation**, 56-58  
  command file examples, 48, 60, 62, 106, 138,  
    140, 154, 177, 252  
  waveforms, 60, 62, 106, 315, 334  
**SM blocks**, 162-165  
**SM chart**, 161-193  
  binary multiplier, 167  
  bus interface unit, 320  
  conversion from state graph, 165  
  conversion to VHDL, 167  
  derivation, 167-177  
  dice game, 172  
    linked, 192  
    for microprogramming, 186, 188  
  floating-point multiplication, 248  
  linked, 190-193  
  memory controller, 328  
  realization of, 178  
    using microprogramming, 184-190  
  simple memory model test, 310  
  TAP controller, 354  
  timing chart, 166  
**S-R flip-flop.** *See flip-flop*  
**standard logic (std\_logic)**, 95-96, 276-280  
  AND function, 278  
  rising\_edge function, 278  
**state assignment**, 19-20  
  one-hot, 229  
  for SM charts, 178  
**state equivalence theorem**, 25-26  
**state graph**  
  for control networks, 123, 126, 128, 171  
  conversion to SM chart, 165-166  
  Mealy, 19, 123, 147, 249, 346  
  Moore, 24, 104  
  strongly connected, 346  
**state machine chart.** *See SM chart*  
**statements**  
  alias, 127, 420  
  assert, 116, 422

- case, 55-56, 421  
 concurrent, definition, 44  
 conditional assignment, 55, 419  
 entity declaration, 45, 420  
 exit, 73, 422  
 file declaration, 295, 423  
 for loop, 72, 421  
 function declaration, 72, 422  
 generate, 282, 283, 423  
 if, 51, 53, 421  
 library declaration, 77, 422  
 package declaration, 76, 423  
 procedure declaration, 74, 422  
 process, 50, 63, 66, 421  
 report, 116, 422  
 selected signal assignment, 55, 419  
 signal declaration, 65, 420  
 use, 77, 423  
 while loop, 422
- s**  
**state table**  
 Mealy, 19, 123, 148, 346  
 Moore, 25, 107  
 reduction of, 26-28
- storage device, 14  
 string, 70  
 strongly connected, definition, 346  
 structural modeling in VHDL, 61  
 stuck-at-0 (s-a-0) fault, 339-343  
 stuck-at-1 (s-a-1) fault, 339-343  
 subtraction, floating-point, 260  
 subtype, 70, 420  
 synchronous design, 31-34  
 synthesis tools, 212  
 synthesis of VHDL code, 283-295  
     case statement example, 285-286  
     dice game, 292  
     floating-point multiplier, 292-295  
     if example, 285  
     inferred latch, 284  
     microcontroller, 415  
     state machine, 289-291  
     UART, 381
- T**  
**T flip-flop.** *See* flip-flop  
**test bench**  
     definition, 144  
     examples, 116, 139, 153, 174-177, 181, 280  
**testing**, 339-370  
     boundary scan, 351-360
- built-in self-test, 361-370  
 combinational logic, 339-343  
 scan path testing, 347-350  
 sequential logic, 344-347  
**TEXTIO package**, 295-299, 435-436  
**timing charts**  
     attributes test, 266  
     chip select write to RAM, 328  
     Mealy network (code converter), 28  
     Moore network (code converter), 25  
     scan test, 349  
     SM chart, 166  
     system with falling-edge device, 32  
     system with rising-edge device, 33  
     transport and inertial delays, 269  
**timing analysis**, 258  
**traffic light controller**, 104-107  
**transition table**, 20  
**transport delays**, 57, 269-270  
**tristate buffer**, 35, 210-211  
     VHDL model, 272  
**tristate bus**, 35-36, 210, 252. *See also* bus  
**truth table**, 2, 109, 113  
     modified for PLA. *See* PLA table  
**ROM**, 87-88  
**type declaration**, 420  
**types**  
     array, 68  
     file, 296  
     IEEE std\_logic, 95-96  
     IEEE-1164 standard logic, 276-280  
     predefined, 45, 67  
     unconditional array type, 68-70  
     unconstrained array, 286  
     user-defined, 67  
     X01Z, 272-275
- U**  
**UART (universal asynchronous receiver-transmitter)**,  
     374-387  
**baud rate generator** (including VHDL model),  
     383-384  
**receiver** (including VHDL model), 379-382  
**transmitter** (including VHDL model), 375-379  
**VHDL model for complete UART**, 385  
**underflow**, 245  
**universal asynchronous receiver-transmitter.** *See*  
     UART  
**unsigned type**, 286, 288  
**use statement**, 77, 423

**V**

variables, 65-67  
  assignment statement, 419  
  declaration, 65, 420  
  in process, 66  
VHDL, general, 43-80, 265-300  
VHDL identifiers, 45

**W**

wait statements, 63, 421

weak keeper circuit, 210  
while loop, 422  
wired-AND function, 210  
worst case analysis, 326

**X**

XC3020. *See also* FPGA  
  implementation of dice game, 213-216  
Xilinx FPGAs. *See* FPGAs  
XOR. *See* exclusive-OR

## DIGITAL SYSTEMS DESIGN USING VHDL

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

Written for an advanced-level course in digital systems design, *Digital Systems Design Using VHDL* integrates the use of the industry-standard hardware description language VHDL into the digital design process. Following a review of basic concepts of logic design in Chapter 1, the author introduces the basics of VHDL in Chapter 2, and then incorporates more coverage of VHDL topics as needed, with advanced topics covered in Chapter 8. Rather than simply teach VHDL as a programming language, this book emphasizes the practical use of VHDL in the digital design process. For example, in Chapter 9, the author develops VHDL models for a RAM memory and a microprocessor bus interface; he then uses a VHDL simulation to verify that timing specifications for the interface between the memory and microprocessor bus are satisfied. The book also discusses the use of CAD tools to synthesize digital logic from a VHDL description (in Chapter 8), and stresses the use of programmable logic devices, including programmable gate arrays. Chapter 10 introduces methods for testing digital systems including boundary scan and built-in self-test.

**VHDL files supporting the book may be downloaded from this PWS web site: <http://www.pws.com/ee/roth.html>**

### Features:

- Teaches the use of VHDL in the digital design process – both digital design concepts and VHDL are covered simultaneously.
- Teaches the use of VHDL for modeling, simulating, and synthesizing digital systems.
- Design examples range in complexity from a simple adder to a complete microcontroller.
- Includes coverage (in Chapter 10) of design for testability, an increasingly important aspect of digital systems design.
- Numerous examples and exercises, reflecting varying levels of difficulty, are provided at the end of each chapter.



**PWS Publishing Company**

20 Park Plaza, Boston, MA 02116

*PWS – Redesigning Engineering Education*

I(T)P An International Thomson Publishing Company

<http://www.pws.com>

ISBN 0-534-95099-X

Universitätsbibliothek Dresden

