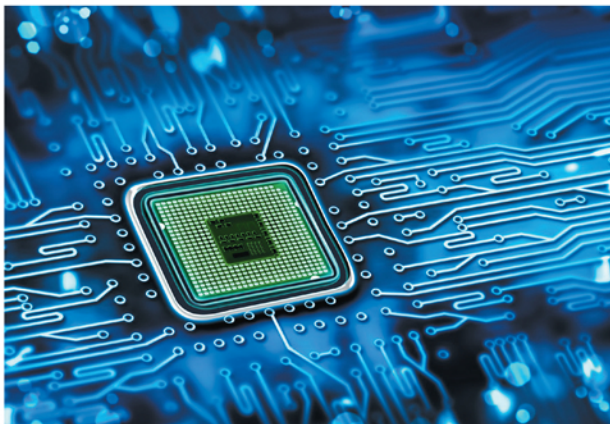


twelfth edition



# digital systems

principles and applications

NEAL S. WIDMER  
GREGORY L. MOSS  
RONALD J. TOCCI

TWELFTH EDITION

# Digital Systems

## Principles and Applications

**Neal S. Widmer**  
Purdue University

**Gregory L. Moss**  
Purdue University

**Ronald J. Tocci**  
Monroe Community College

**PEARSON**

Boston Columbus Indianapolis New York San Francisco Amsterdam  
Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto Delhi  
Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

**Editor-in-Chief:** Andrew Gilfillan  
**Product Manager:** Anthony Webster  
**Program Manager:** Holly Shufeldt  
**Project Manager:** Rex Davidson  
**Editorial Assistant:** Nancy Kesterson  
**Team Lead Project Manager:** Bryan Pirrmann  
**Team Lead Program Manager:** Laura Weaver  
**Director of Marketing:** David Gesell  
**Senior Product Marketing Manager:** Darcy Betts

**Field Marketing Manager:** Thomas Hayward  
**Procurement Specialist:** Deidra M. Skahill  
**Creative Director:** Andrea Nix  
**Art Director:** Diane Y. Ernsberger  
**Cover Designer:** Cenveo  
**Full-Service Project Management:** Philip Alexander/Integra Software Services, Pvt, Ltd.  
**Printer/Binder:** R.R. Donnelley

Unless otherwise indicated herein, any third-party trademarks that may appear in this work are the property of their respective owners and any references to third-party trademarks, logos or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson's products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc. or its affiliates, authors, licensees or distributors.

---

**Copyright © 2017 by Pearson Education, Inc. or its affiliates. All Rights Reserved.** Printed in the United States of America. This publication is protected by copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit [www.pearsoned.com/permissions/](http://www.pearsoned.com/permissions/).

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

Names: Tocci, Ronald J., author. | Widmer, Neal S., author. | Moss, Gregory L., author.  
Title: Digital systems: principles and applications/Neal S. Widmer, Purdue University, Ronald J. Tocci, Monroe Community College, Gregory L. Moss, Purdue University.  
Description: Twelfth edition. | Upper Saddle River [New Jersey]: Pearson, [2017] | Tocci's name appears first in earlier editions.  
Identifiers: LCCN 2016007313 | ISBN 9780134220130  
Subjects: LCSH: Digital electronics.  
Classification: LCC TK7868.D5 T62 2017 | DDC 621.381—dc23 LC record available at <http://lccn.loc.gov/2016007313>

10 9 8 7 6 5 4 3 2 1

**PEARSON**

[www.pearsonhighered.com](http://www.pearsonhighered.com)

ISBN 10: 0-13-422013-7  
ISBN 13: 978-0-13-422013-0



## PREFACE

This book is a comprehensive study of the principles and techniques of modern digital systems. It teaches the fundamental principles of digital systems and covers thoroughly both traditional and modern methods of applying digital design and development techniques, including how to manage a systems-level project. The book is intended for use in two- and four-year programs in technology, engineering, and computer science. It can also be used for High School STEM education courses in these topical areas. Although a background in basic electronics is helpful, most of the material requires no electronics training. Portions of the text that use electronics concepts can be skipped without adversely affecting the comprehension of the logic principles.

### What's New in This Edition?

The following list summarizes the improvements in the twelfth edition of *Digital Systems*. Details can be found in the section titled “Specific Changes” on page ix.

- Every *section* of every chapter now has a short list of expected outcomes for that section.
- Chapter 1 has been revised extensively in response to feedback from users.
- New material on troubleshooting prototype circuits using systematic fault isolation techniques applied to digital logic circuits has been added to Section 4-13.
- Quadrature Shaft Encoders used to obtain absolute shaft position serve as a real example of flip-flop applications, and timing limitations.
- More material has been added to better explain the behavior of VHDL data objects and how they are updated in sequential processes.
- Throughout the text, obsolete technology has been deleted or abbreviated to provide only content appropriate to modern systems. More modern examples are used as needed.
- Some new problems have been added and outdated problems have been removed.

## General Features

In industry today, getting a product to market very quickly is important. The use of modern design tools, CPLDs, and FPGAs allows engineers to progress from concept to functional silicon very quickly. Microcontrollers have taken over many applications that once were implemented by digital circuits, and DSP has been used to replace many analog circuits. It is amazing that microcontrollers, DSP, and all the necessary glue logic can now be consolidated onto a single FPGA using a hardware description language with advanced development tools. Today's students must be exposed to these modern tools, even in an introductory course. It is every educator's responsibility to find the best way to prepare graduates for the work they will encounter in their professional lives.

The standard SSI and MSI parts that have served as “bricks and mortar” in the building of digital systems for over 40 years are now obsolete and becoming less available. Many of the techniques that have been taught over that time have focused on optimizing circuits that are built from these outmoded devices. The topics that are uniquely suited to applying the old technology *but do not contribute to an understanding of the new technology* are being de-emphasized. From an educational standpoint, however, these small ICs do offer a way to study simple digital circuits, and the wiring of circuits using breadboards is a valuable pedagogic exercise. They help to solidify concepts such as binary inputs and outputs, physical device operation, and practical limitations, using a very simple platform. Consequently, we have chosen to continue to introduce the conceptual descriptions of digital circuits and to offer examples using conventional standard logic parts. For instructors who continue to teach the fundamentals using SSI and MSI circuits, this edition retains those qualities that have made the text so widely accepted in the past. Many hardware design tools even provide an easy-to-use design entry technique that will employ the functionality of conventional standard parts with the flexibility of programmable logic devices. A digital design can be described using a schematic drawing with pre-created building blocks that are equivalent to conventional standard parts, which can be compiled and then programmed directly into a target PLD with the added capability of easily simulating the design within the same development tool.

We believe that graduates will actually apply the concepts presented in this book using higher-level description methods and more complex programmable devices. The major shift in the field is a greater need to understand the description methods, rather than focusing on the architecture of an actual device. Software tools have evolved to the point where there is little need for concern about the inner workings of the hardware but much more need to focus on what goes in, what comes out, and how the designer can describe what the device is supposed to do. We also believe that graduates will be involved with projects using state-of-the-art design tools and hardware solutions.

This book offers a strategic advantage for teaching the vital topic of hardware description languages to beginners in the digital field. VHDL is undisputedly an industry standard language at this time, but it is also very complex and has a steep learning curve. Beginning students are often discouraged by the rigorous requirements of various data types, and they struggle with understanding edge-triggered events in VHDL. Fortunately, Altera offers AHDL, a less demanding language that uses the same basic concepts as VHDL but is much easier for beginners to master. So, instructors can opt to use AHDL to teach introductory students or VHDL for more advanced classes. This edition offers more than 40 AHDL examples, more than 40 VHDL examples, and many examples of simulation testing. All of these design files are available on the website (<http://www.pearsonhighered.com/careersresources/>).

Altera's software development system is Quartus II. The material in this text does not attempt to teach a particular hardware platform or the details of using a software development system. We have chosen to show what this tool can do, rather than train the reader how to use it.

Many laboratory hardware options are available to users of this book. Complete development boards are available that offer the normal types of inputs and outputs like logic switches, pushbuttons, clock signals, LEDs, and 7-segment displays. Many boards also offer standard connectors for readily available computer hardware, such as a standard keyboard, computer mouse, VGA video monitor, COM ports, audio in/out jacks, plus two 40-pin general-purpose I/O ribbon connectors that allow connection to any digital peripheral hardware.

Our approach to HDL and PLDs gives instructors several options:

1. The HDL material can be skipped entirely without affecting the continuity of the text.
2. HDL can be taught as a separate topic by skipping the material initially and then going back to the last sections of Chapters 3, 4, 5, 6, 7, and 9 and then covering Chapter 10.
3. HDL and the use of PLDs can be covered as the course unfolds—chapter by chapter—and woven into the fabric of the lecture/lab experience.

Among all specific hardware description languages, VHDL is clearly the industry standard and is most likely to be used by graduates in their careers. We have always felt that it is a bold proposition, however, to try to teach VHDL in an introductory course. The nature of the syntax, the subtle distinctions in object types, and the higher levels of abstraction can pose obstacles for a beginner. For this reason, we have included Altera's AHDL as the recommended introductory language for freshman and sophomore courses. We have also included VHDL as the recommended language for more advanced classes or introductory courses offered to more mature students. We do not recommend trying to cover both languages in the same course. Sections of the text that cover the specifics of a language are clearly designated with a color bar in the margin. The HDL code figures are set in a color to match the color-coded text explanation. The reader can focus only on the language of his or her choice and skip the other. Obviously, we have attempted to appeal to the diverse interests of our market, but we believe we have created a book that can be used in multiple courses and will serve as an excellent reference after graduation.

## Chapter Organization

Many instructors opt to not use the chapters of a textbook in the sequence in which they are presented. This book was written so that, for the most part, each chapter builds on previous material, but it is possible to alter the chapter sequence somewhat. The first part of Chapter 6 (arithmetic operations) can be covered right after Chapter 2 (number systems), although this will lead to a long interval before the arithmetic circuits of Chapter 6 are encountered. Much of the material in Chapter 8 (IC characteristics) can be covered earlier (e.g., after Chapter 4 or 5) without creating any serious problems.

This book can be used either in a one-term course or in a two-term sequence. In a one-term course, limits on available class hours might require omitting some topics. Obviously, the choice of deletions will depend on factors such as program or course objectives and student background. Sections

---

**FIGURE P1** Letters denote categories of problems, and asterisks indicate that corresponding solutions are provided at the end of the text.

## PROBLEMS

### SECTION 9-1

- B** 9-1. Refer to Figure 9-3. Determine the levels at each decoder output for the following sets of input conditions.
- (a)\*All inputs LOW
  - (b)\*All inputs LOW except  $E_3 = \text{HIGH}$
  - (c) All inputs HIGH except  $\bar{E}_1 = \bar{E}_2 = \text{LOW}$
  - (d) All inputs HIGH
- B** 9-2\* What is the number of inputs and outputs of a decoder that accepts 64 different input combinations?

\* Answers to problems marked with an asterisk can be found in the back of the text.

in each chapter that deal with troubleshooting, PLDs, HDLs, or microcomputer applications can be deferred to an advanced course.

**PROBLEM SETS** This edition includes six categories of problems: basic (B), challenging (C), troubleshooting (T), new (N), design (D), and HDL (H). Undesignated problems are considered to be of intermediate difficulty, between basic and challenging. Problems for which solutions are printed in the back of the text or on the website (<http://www.pearsonhighered.com/careersresources/>) are marked with an asterisk (see Figure P1).

**PROJECT MANAGEMENT AND SYSTEM-LEVEL DESIGN** Several real-world examples are included in Chapter 10 to describe the techniques used to manage projects. These applications are generally familiar to most students studying electronics, and the primary example of a digital clock is familiar to everyone. Many texts talk about top-down design, but this text demonstrates the key features of this approach and how to use the modern tools to accomplish it.

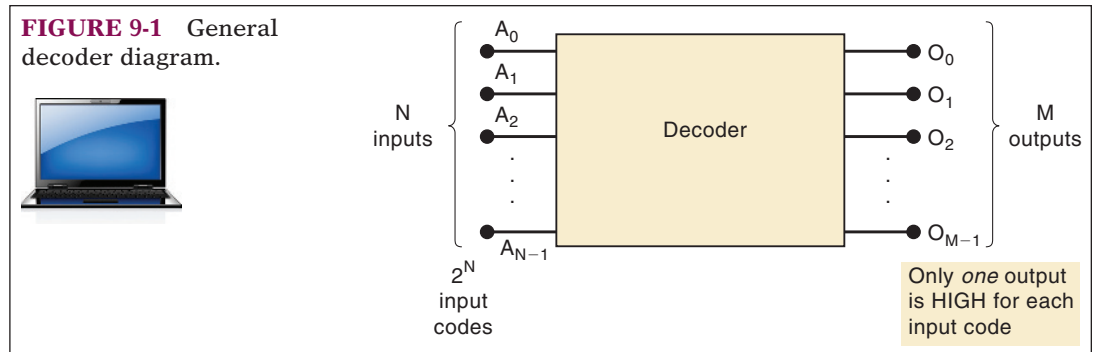
**SIMULATION FILES** This edition also includes simulation files that can be loaded into Multisim<sup>®</sup>. The circuit schematics of many of the figures throughout the text have been captured as input files for this popular simulation tool. Each file has some way of demonstrating the operation of the circuit or reinforcing a concept. In many cases, instruments are attached to the circuit and input sequences are applied to demonstrate the concept presented in one of the figures of the text. These circuits can then be modified as desired to expand on topics or create assignments and tutorials for students. All figures in the text that have a corresponding simulation file on the website are identified by the icon shown in Figure P2.

## Specific Changes

The major changes in the topical coverage are listed here.

- **Chapter 1.** Chapter 1 has been revised extensively in response to feedback from users. The significance of how Digital Systems will impact innovations of the future is emphasized.

New material focuses on interpretation of terminology and introduction to concepts used throughout the text. Basic concepts of binary



**FIGURE P2** The icon denotes a corresponding simulation file on the Web.

signals are introduced and explained through examples. New material on periodic cycles and measurements on digital waveforms is presented, setting the stage for understanding these issues in later chapters. The basics of digital signals and sampling are explained at a very introductory level.

This chapter in the 11th edition had material that has now become very outdated since its publication. Some of the historic analogies used in that edition were ineffective. The revisions have replaced or eliminated these.

- **Chapter 2.** The Gray Code is used to introduce the concept of a quadrature encoder: a device that produces a 2-bit Gray Code sequence capable of discerning the direction and angular rotation of a shaft.
- **Chapter 3.** New problems at the end of this chapter focus on logic circuits common to automobiles.
- **Chapter 4.** The material introducing PLD programming and development software has been updated and improved. The section on troubleshooting has been expanded to teach structured problem solving as it applies to hardware debugging of traditional prototyped digital circuits. The VHDL material has been enhanced to explain some subtle but very important aspects of data objects in this language. The role of the “PROCESS” is also more thoroughly covered improving the foundation that Chapter 5 builds on.
- **Chapter 5.** High-speed digital systems are easily affected by timing limitations of the circuitry. New material in this chapter explains the adverse effects caused when setup and hold time requirements are violated by explaining meta-stability. A teaching example that can be reproduced in the laboratory environment has been added. The focus is on the many applications of D flip-flops but it is presented in the context of a quadrature shaft encoder that must reliably and repeatedly keep track of absolute shaft position as it is rotated back and forth over many cycles. Design techniques from Chapter 4 are employed to design a circuit that should meet the system’s needs. The initial circuit’s marginal performance demonstrates what happens when real-timing constraints are not taken into account. A way to correct this problem is presented using even more applications of D flip-flops.
- **Chapter 6.** An Example from the 11th edition used some features of Quartus software that have since become obsolete. The example has been modified to align with more recent updates of Quartus.
- **Chapter 7.** Very few and minor changes were made to Chapter 7.
- **Chapter 8.** The section on the obsolete Emitter Coupled Logic (ECL) was deleted along with other minor updates.



- **Chapter 9.** The concept of Time Division Multiplexing is added to provide an example of how many digital signals are able to share a common data pathway. A simple system is presented that can easily be reproduced in a laboratory exercise.
- **Chapter 10.** No changes were made in Chapter 10.
- **Chapter 11.** No changes were made in Chapter 11.
- **Chapter 12.** The coverage of floating gate MOSFETS, the technology behind flash memory, is enhanced.
- **Chapter 13.** This chapter has been generalized with references to older series of CPLDs and FPGAs abbreviated.

## Retained Features

This edition retains all of the features that made the previous editions so widely accepted. It utilizes a block diagram approach to teach the basic logic operations without confusing the reader with the details of internal operation. All but the most basic electrical characteristics of the logic ICs are withheld until the reader has a firm understanding of logic principles. In Chapter 8, the reader is introduced to the internal IC circuitry. At that point, the reader can interpret a logic block's input and output characteristics and "fit" it properly into a complete system.

The treatment of each new topic or device typically follows these steps: the principle of operation is introduced; thoroughly explained examples and applications are presented, often using actual ICs; short review questions are posed at the end of the section; and finally, in-depth problems are available at the end of the chapter. These problems, ranging from simple to complex, provide instructors with a wide choice of student assignments. These problems are often intended to reinforce the material without simply repeating the principles. They require students to demonstrate comprehension of the principles by applying them to different situations. This approach also helps students to develop confidence and expand their knowledge of the material.

The material on PLDs and HDLs is distributed throughout the text, with examples that emphasize key features in each application. These topics appear at the end of each chapter, making it easy to relate each topic to the general discussion earlier in the chapter or to address the general discussion separately from the PLD/HDL coverage.

The extensive troubleshooting coverage is spread over Chapters 4 through 12 and includes presentation of troubleshooting principles and techniques, case studies, 17 troubleshooting examples, and 46 *real* troubleshooting problems. When supplemented with hands-on lab exercises, this material can help foster the development of good troubleshooting skills.

This edition offers more than 220 worked-out examples, more than 660 review questions, and more than 640 chapter problems/exercises. Some of these problems are applications that show how the logic devices presented in the chapter are used in a typical microcomputer system. Answers to a majority of the problems immediately follow the Glossary. The Glossary provides concise definitions of all terms in the text that have been highlighted in bold-face type.

An IC index is provided at the back of the book to help readers locate easily material on any IC cited or used in the text. The back endsheets provide tables of the most often used Boolean algebra theorems, logic gate summaries, and flip-flop truth tables for quick reference when doing problems or working in the lab.

## Supplements

An extensive complement of teaching and learning tools has been developed to accompany this textbook. Each component provides a unique function, and each can be used independently or in conjunction with the others.

## WEB RESOURCES

- **Quartus II Web Version software from Altera.** This development system software is available from Altera.
- **Design files from the textbook figures.** More than 40 design files in each language are presented in figures throughout the text. Students can load these into the Altera software and test them.
- **Solutions to selected problems: HDL design files.** A few of the end-of-chapter problem solutions are available to students. (All of the HDL solutions are available to instructors in the *Instructor's Resource Manual*.) Solutions for Chapter 7 problems include some large graphic and HDL files that are not published in the back of the book but are available on the web site.
- **Circuits from the text rendered in Multisim®.** Students can open and work interactively with approximately 100 circuits to increase their understanding of concepts and prepare for laboratory activities. The Multisim circuit files are provided for use by anyone who has Multisim software.

## INSTRUCTOR RESOURCES

- **Online Instructor's Resource Manual.** This manual contains worked-out solutions for all end-of-chapter problems in this textbook. (ISBN 0-13-422021-8)
- **Online PowerPoint® presentations.** Figures from the text, in addition to Lecture Notes for each chapter, are available. (ISBN 0-13-422019-6)
- **Online TestGen.** A computerized test bank is available. (ISBN 0-13-422016-1)

To access supplementary materials online, instructors need to request an instructor access code. Go to [www.pearsonhighered.com/irc](http://www.pearsonhighered.com/irc), where you can register for an instructor access code. Within 48 hours after registering, you will receive a confirming e-mail, including an instructor access code. Once you have received your code, go to the site and log on for full instructions on downloading the materials you wish to use.

## Acknowledgments

We are grateful to all those who evaluated the eleventh edition and provided answers to an extensive questionnaire:

Their comments, critiques, and suggestions were given serious consideration and were invaluable in determining the final form of the twelfth edition.

We also are greatly indebted to Professor Frank Ambrosio, Monroe Community College, for his usual high-quality work on the *Instructor's*

---

*Resource Manual*; and Professor Daniel Leon-Salas, Purdue University, for his technical review of topics and many suggestions for improvements.

A writing project of this magnitude requires conscientious and professional editorial support, and Pearson came through again in typical fashion. We thank the staff at Pearson for their help to make this publication a success.

And finally, we want to let our wives, children, and grandchildren know how much we appreciate their support and their understanding. We hope that we can eventually make up for all the hours we spent away from them while we worked on this revision.

Neal S. Widmer  
Ronald J. Tocci  
Gregory L. Moss

---



# CONTENTS

## **CHAPTER 1**   **Introductory Concepts**   **2**

- 1-1** Introduction to Digital 1s and 0s   4
- 1-2** Digital Signals   9
  - Need for Timing   10
  - Highs and Lows Over Time   11
  - Periodic/Aperiodic   11
  - Period/Frequency   11
  - Duty Cycle   12
  - Transitions   12
  - Edges/Events   12
- 1-3** Logic Circuits and Evolving Technology   13
  - Logic Circuits   13
  - Digital Integrated Circuits   14
- 1-4** Numerical Representations   14
  - Analog Representations   15
  - Digital Representations   15
- 1-5** Digital and Analog Systems   17
  - Advantages of Digital Techniques   17
  - Limitations of Digital Techniques   18
- 1-6** Digital Number Systems   19
  - Decimal System   19
  - Decimal Counting   20
  - Binary System   21
  - Binary Counting   22

- 1-7** Representing Signals with Numeric Quantities   23
- 1-8** Parallel and Serial Transmission   25
- 1-9** Memory   27
- 1-10** Digital Computers   28
  - Major Parts of a Computer   28
  - Types of Computers   29
  - Memory   30
  - Digital Progress Today and Tomorrow   31

## **CHAPTER 2**   **Number Systems and Codes**   **36**

- 2-1** Binary-to-Decimal Conversions   38
- 2-2** Decimal-to-Binary Conversions   39
  - Counting Range   41
- 2-3** Hexadecimal Number System   41
  - Hex-to-Decimal Conversion   42
  - Decimal-to-Hex Conversion   43
  - Hex-to-Binary Conversion   43
  - Binary-to-Hex Conversion   44
  - Counting in Hexadecimal   44
  - Usefulness of Hex   44
  - Summary of Conversions   45

2-4	BCD Code	46
	Binary-Coded-Decimal Code	46
	Comparison of BCD and Binary	47
2-5	The Gray Code	48
	Quadrature Encoders	50
2-6	Putting it All Together	51
2-7	The Byte, Nibble, and Word	52
	Bytes	52
	Nibbles	52
	Words	53
2-8	Alphanumeric Codes	53
	ASCII Code	54
2-9	Parity Method For Error Detection	56
	Parity Bit	57
	Error Correction	58
2-10	Applications	59

## **CHAPTER 3 Describing Logic Circuits 68**

---

3-1	Boolean Constants and Variables	71
3-2	Truth Tables	72
3-3	OR Operation with OR Gates	73
	OR Gate	74
	Summary of the OR Operation	75
3-4	AND Operation with AND Gates	77
	AND Gate	78
	Summary of the AND Operation	79
3-5	NOT Operation	80
	NOT Circuit (INVERTER)	81
	Summary of Boolean Operations	81
3-6	Describing Logic Circuits Algebraically	82
	Operator Precedence	82
	Circuits Containing INVERTERS	83
3-7	Evaluating Logic-Circuit Outputs	84
	Analysis Using a Table	85
3-8	Implementing Circuits from Boolean Expressions	87
3-9	NOR Gates and NAND Gates	88
	NOR Gate	88
	NAND Gate	90
3-10	Boolean Theorems	92
	Multivariable Theorems	93

3-11	DeMorgan's Theorems	95
	Implications of DeMorgan's Theorems	97
3-12	Universality of NAND Gates and NOR Gates	99
3-13	Alternate Logic-Gate Representations	102
	Logic-Symbol Interpretation	104
	Summary	104
3-14	Which Gate Representation to Use	105
	Which Circuit Diagram Should Be Used?	107
	Bubble Placement	107
	Analyzing Circuits	108
	Asserted Levels	110
	Labeling Active-LOW Logic Signals	110
	Labeling Bistate Signals	110
3-15	Propagation Delay	111
3-16	Summary of Methods to Describe Logic Circuits	112
3-17	Description Languages Versus Programming Languages	114
	VHDL and AHDL	115
	Computer Programming Languages	115
3-18	Implementing Logic Circuits with PLDs	117
3-19	HDL Format and Syntax	118
3-20	Intermediate Signals	121

## **CHAPTER 4 Combinational Logic Circuits 136**

---

4-1	Sum-of-Products Form	138
	Product-of-Sums	138
4-2	Simplifying Logic Circuits	139
4-3	Algebraic Simplification	140
4-4	Designing Combinational Logic Circuits	145
	Complete Design Procedure	147
4-5	Karnaugh Map Method	152
	Karnaugh Map Format	152
	Looping	154
	Looping Groups of Two (Pairs)	154
	Looping Groups of Four (Quads)	155
	Looping Groups of Eight (Octets)	156
	Complete Simplification Process	157
	Filling a K Map from an Output Expression	160

---

	Don't-Care Conditions	161		
	Summary	163		
<b>4-6</b>	Exclusive-OR and Exclusive-NOR Circuits	163		
	Exclusive-OR	163		
	Exclusive-NOR	165		
<b>4-7</b>	Parity Generator and Checker	169		
<b>4-8</b>	Enable/Disable Circuits	170		
<b>4-9</b>	Basic Characteristics of Legacy Digital ICs	173		
	Bipolar and Unipolar Digital ICs	174		
	TTL Family	175		
	CMOS Family	176		
	Power and Ground	176		
	Logic-Level Voltage Ranges	177		
	Unconnected (Floating) Inputs	177		
	Logic-Circuit Connection Diagrams	178		
<b>4-10</b>	Troubleshooting Digital Systems	180		
<b>4-11</b>	Internal Digital IC Faults	182		
	Malfunction in Internal Circuitry	182		
	Input Internally Shorted to Ground or Supply	182		
	Output Internally Shorted to Ground or Supply	183		
	Open-Circuited Input or Output	183		
	Short Between Two Pins	185		
<b>4-12</b>	External Faults	186		
	Open Signal Lines	186		
	Shorted Signal Lines	187		
	Faulty Power Supply	187		
	Output Loading	188		
<b>4-13</b>	Troubleshooting Prototyped Circuits	190		
<b>4-14</b>	Programmable Logic Devices	194		
	PLD Hardware	195		
	Programming a PLD	196		
	Development Software	197		
	Design and Development Process	200		
<b>4-15</b>	Representing Data in HDL	202		
	Bit Arrays/Bit Vectors	203		
<b>4-16</b>	Truth Tables Using HDL	207		
<b>4-17</b>	Decision Control Structures in HDL	210		
	IF/ELSE	211		
	ELSIF	215		
			<b>CHAPTER 5</b>	<b>Flip-Flops and Related Devices</b>
				<b>236</b>
<b>5-1</b>	NAND Gate Latch	239		
	Setting the Latch (FF)	240		
	Resetting the Latch (FF)	240		
	Simultaneous Setting and Resetting	241		
	Summary of NAND Latch	241		
	Alternate Representations	242		
	Terminology	242		
<b>5-2</b>	NOR Gate Latch	245		
	Flip-Flop State on Power-Up	247		
<b>5-3</b>	Troubleshooting Case Study	247		
<b>5-4</b>	Digital Pulses	249		
<b>5-5</b>	Clock Signals and Clocked Flip-Flops	251		
	Clocked Flip-Flops	252		
	Setup and Hold Times	252		
<b>5-6</b>	Clocked S-R Flip-Flop	254		
	Internal Circuitry of the Edge-Triggered S-R Flip-Flop	256		
<b>5-7</b>	Clocked J-K Flip-Flop	258		
	Internal Circuitry of the Edge-Triggered J-K Flip-Flop	259		
<b>5-8</b>	Clocked D Flip-Flop	260		
	Implementation of the D Flip-Flop	261		
	Parallel Data Transfer	262		
<b>5-9</b>	D Latch (Transparent Latch)	262		
<b>5-10</b>	Asynchronous Inputs	264		
	Designations for Asynchronous Inputs	266		
<b>5-11</b>	Flip-Flop Timing Considerations	267		
	Setup and Hold Times	267		
	Propagation Delays	268		
	Maximum Clocking Frequency, $f_{MAX}$	268		
	Clock Pulse HIGH and LOW Times	268		
	Asynchronous Active Pulse Width	269		
	Clock Transition Times	269		
<b>5-12</b>	Potential Timing Problem in FF Circuits	269		
<b>5-13</b>	Flip-Flop Applications	271		
<b>5-14</b>	Flip-Flop Synchronization	272		
<b>5-15</b>	Detecting an Input Sequence	273		
<b>5-16</b>	Detecting a Transition or "Event"	275		
<b>5-17</b>	Data Storage and Transfer	276		
	Parallel Data Transfer	277		

5-18	Serial Data Transfer: Shift Registers	278		
	Hold Time Requirement	279		
	Serial Transfer Between Registers	280		
	Shift-Left Operation	281		
	Parallel Versus Serial Transfer	281		
5-19	Frequency Division and Counting	282		
	Counting Operation	283		
	State Transition Diagram	284		
	MOD Number	284		
5-20	Application of Flip-Flops with Timing Constraints	286		
	Timing Issues	290		
5-21	Microcomputer Application	293		
5-22	Schmitt-Trigger Devices	294		
5-23	One-Shot (Monostable Multivibrator)	296		
	Nonretriggerable One-Shot	296		
	Retriggerable One-Shot	297		
	Actual Devices	298		
	Monostable Multivibrator	298		
5-24	Clock Generator Circuits	299		
	Schmitt-Trigger Oscillator	299		
	555 Timer Used as an Astable Multivibrator	299		
	Crystal-Controlled Clock Generators	302		
5-25	Troubleshooting Flip-Flop Circuits	302		
	Open Inputs	303		
	Shorted Outputs	304		
	Clock Skew	305		
5-26	Sequential Circuits in PLDs Using Schematic Entry	307		
5-27	Sequential Circuits Using HDL	311		
	The D Latch	314		
5-28	Edge-Triggered Devices	315		
5-29	HDL Circuits with Multiple Components	320		
	2's-Complement Form	345		
	Representing Signed Numbers Using 2's Complement	345		
	Sign Extension	347		
	Negation	347		
	Special Case in 2's-Complement Representation	348		
6-3	Addition in the 2's-Complement System	351		
6-4	Subtraction in the 2's-Complement System	352		
	Arithmetic Overflow	353		
	Number Circles and Binary Arithmetic	354		
6-5	Multiplication of Binary Numbers	355		
	Multiplication in the 2's-Complement System	356		
6-6	Binary Division	357		
6-7	BCD Addition	357		
	Sum Equals 9 or Less	358		
	Sum Greater than 9	358		
	BCD Subtraction	359		
6-8	Hexadecimal Arithmetic	360		
	Hex Addition	360		
	Hex Subtraction	361		
	Hex Representation of Signed Numbers	362		
6-9	Arithmetic Circuits	363		
	Arithmetic/Logic Unit	363		
6-10	Parallel Binary Adder	364		
6-11	Design of a Full Adder	366		
	K-Map Simplification	368		
	Half Adder	369		
6-12	Complete Parallel Adder with Registers	369		
	Register Notation	370		
	Sequence of Operations	371		
6-13	Carry Propagation	372		
6-14	Integrated-Circuit Parallel Adder	373		
	Cascading Parallel Adders	373		
6-15	2's-Complement Circuits	375		
	Addition	375		
	Subtraction	375		
	Combined Addition and Subtraction	377		
<b>CHAPTER 6 Digital Arithmetic: Operations and Circuits</b>		<b>340</b>		
6-1	Binary Addition and Subtraction	342		
	Binary Addition	342		
	Binary Subtraction	343		
6-2	Representing Signed Numbers	343		
	1's-Complement Form	344		

6-16	ALU Integrated Circuits	378	7-8	Decoding a Counter	440
	The 74LS382/74HC382 ALU	379		Active-HIGH Decoding	441
	Expanding the ALU	381		Active-LOW Decoding	442
	Other ALUs	382		BCD Counter Decoding	442
6-17	Troubleshooting Case Study	382	7-9	Analyzing Synchronous Counters	444
6-18	Using Altera Library Functions	384	7-10	Synchronous Counter Design	447
	Megafunction LPMs for Arithmetic Circuits	385		Basic Idea	447
	Using a Parallel Adder to Count	389		J-K Excitation Table	448
6-19	Logical Operations on Bit Arrays with HDLs	390		Design Procedure	449
6-20	HDL Adders	392		Stepper Motor Control	452
6-21	Parameterizing the Bit Capacity of a Circuit	394		Synchronous Counter Design with D FF	454
<b>CHAPTER 7 Counters and Registers</b>			7-11	Altera Library Functions for Counters	456
		<b>408</b>	7-12	HDL Counters	460
7-1	Asynchronous (Ripple) Counters	410		State Transition Description Methods	461
	Signal Flow	411		Behavioral Description	464
	MOD Number	412		Simulation of Basic Counters	467
	Frequency Division	412		Full-Featured Counters in HDL	467
	Duty Cycle	413		Simulation of Full-Featured Counter	471
7-2	Propagation Delay in Ripple Counters	414	7-13	Wiring HDL Modules Together	473
7-3	Synchronous (Parallel) Counters	416		MOD-100 BCD Counter	476
	Circuit Operation	418	7-14	State Machines	481
	Advantage of Synchronous Counters over Asynchronous	418		Simulation of State Machines	484
	Actual ICs	418		Traffic Light Controller State Machine	485
7-4	Counters with Mod Numbers $< 2^N$	419		Choosing HDL Coding Techniques	491
	State Transition Diagram	421	7-15	Register Data Transfer	493
	Displaying Counter States	421	7-16	IC Registers	493
	Changing the MOD Number	423		Parallel In/Parallel Out—The 74ALS174/74HC174	494
	General Procedure	423		Serial In/Serial Out—The 74ALS166/74HC166	496
	Decade Counters/BCD Counters	425		Parallel In/Serial Out—The 74ALS165/74HC165	498
7-5	Synchronous Down and Up/Down Counters	426		Serial In/Parallel Out—The 74ALS164/74HC164	500
7-6	Presetable Counters	428	7-17	Shift-Register Counters	502
	Synchronous Presetting	430		Ring Counter	502
7-7	IC Synchronous Counters	430		Starting a Ring Counter	502
	The 74ALS160-163/74HC160-163 Series	430		Johnson Counter	503
	The 74ALS190-191/74HC190-191 Series	434		Decoding a Johnson Counter	505
	Multistage Arrangement	439		IC Shift-Register Counters	506
			7-18	Troubleshooting	506
			7-19	Megafunction Registers	509



7-20	HDL Registers	513		
7-21	HDL Ring Counters	519		
7-22	HDL One-Shots	521		
	Nonretriggerable One-Shot Simulation	523		
	Retriggerable, Edge-Triggered One-Shots in HDL	524		
	Edge-Triggered Retriggerable One-Shot Simulation	527		
<b>CHAPTER 8 Integrated-Circuit Logic Families</b>		<b>550</b>		
<hr/>				
8-1	Digital IC Terminology	552		
	Current and Voltage Parameters (See Figure 8-1)	552		
	Fan-Out	553		
	Propagation Delays	554		
	Power Requirements	554		
	Noise Immunity	555		
	Invalid Voltage Levels	557		
	Current-Sourcing and Current-Sinking Action	557		
	IC Packages	558		
8-2	The TTL Logic Family	561		
	Circuit Operation—LOW State	561		
	Circuit Operation—HIGH State	562		
	Current-Sinking Action	564		
	Current-Sourcing Action	564		
	Totem-Pole Output Circuit	564		
	TTL NOR Gate	565		
	Summary	565		
8-3	TTL Data Sheets	566		
	Supply Voltage and Temperature Range	567		
	Voltage Levels	567		
	Maximum Voltage Ratings	568		
	Power Dissipation	568		
	Propagation Delays	568		
8-4	TTL Series Characteristics	569		
	Standard TTL, 74 Series	570		
	Schottky TTL, 74S Series	570		
	Low-Power Schottky TTL, 74LS Series (LS-TTL)	571		
	Advanced Schottky TTL, 74AS Series (AS-TTL)	571		
	Advanced Low-Power Schottky TTL, 74ALS Series	571		
	74F—Fast TTL	571		
	Comparison of TTL Series Characteristics	572		
8-5	TTL Loading and Fan-Out	573		
	Determining the Fan-Out	574		
8-6	Other TTL Characteristics	578		
	Unconnected Inputs (Floating)	578		
	Unused Inputs	578		
	Tied-Together Inputs	579		
	Biasing TTL Inputs Low	580		
	Current Transients	581		
8-7	MOS Technology	582		
	The MOSFET	583		
	Basic MOSFET Switch	583		
8-8	Complementary MOS Logic	585		
	CMOS Inverter	586		
	CMOS NAND Gate	586		
	CMOS NOR Gate	587		
	CMOS SET-RESET FF	588		
8-9	CMOS Series Characteristics	588		
	4000/14,000 Series	588		
	74HC/HCT (High-Speed CMOS)	589		
	74AC/ACT (Advanced CMOS)	589		
	74AHC/AHCT (Advanced High-Speed CMOS)	589		
	BiCMOS 5-V Logic	589		
	Power-Supply Voltage	590		
	Logic Voltage Levels	590		
	Noise Margins	590		
	Power Dissipation	591		
	$P_D$ Increases with Frequency	591		
	Fan-Out	592		
	Switching Speed	592		
	Unused Inputs	593		
	Static Sensitivity	593		
	Latch-Up	594		
8-10	Low-Voltage Technology	594		
	CMOS Family	595		
	BiCMOS Family	596		
8-11	Open-Collector/Open-Drain Outputs	597		
	Open-Collector/Open-Drain Outputs	598		

---

	Open-Collector/Open-Drain Buffer/ Drivers 600		
	IEEE/ANSI Symbol for Open-Collector/ Drain Outputs 601		
<b>8-12</b>	Tristate (Three-State) Logic Outputs 602		
	Advantage of Tristate 602		
	Tristate Buffers 603		
	Tristate ICs 605		
	IEEE/ANSI Symbol for Tristate Outputs 605		
<b>8-13</b>	High-Speed Bus Interface Logic 605		
<b>8-14</b>	CMOS Transmission Gate (Bilateral Switch) 607		
<b>8-15</b>	IC Interfacing 609		
	Interfacing 5-V TTL and CMOS 611		
	CMOS Driving TTL 612		
	CMOS Driving TTL in the HIGH State 612		
	CMOS Driving TTL in the LOW State 612		
<b>8-16</b>	Mixed-Voltage Interfacing 614		
	Low-Voltage Outputs Driving High-Voltage Loads 614		
	High-Voltage Outputs Driving Low-Voltage Loads 614		
<b>8-17</b>	Analog Voltage Comparators 616		
<b>8-18</b>	Troubleshooting 617		
	Using a Logic Pulser and Probe to Test a Circuit 618		
	Finding Shorted Nodes 618		
<b>8-19</b>	Characteristics of an FPGA 619		
	Power-Supply Voltage 619		
	Logic Voltage Levels 620		
	Power Dissipation 620		
	Maximum Input Voltage and Output Current Ratings 621		
	Switching Speed 621		
<b>CHAPTER 9 MSI Logic Circuits 638</b>			
<b>9-1</b>	Decoders 639		
	ENABLE Inputs 640		
	BCD-to-Decimal Decoders 644		
	BCD-to-Decimal Decoder/Driver 645		
	Decoder Applications 645		
<b>9-2</b>	BCD-to-7-Segment Decoder/Drivers 647		
	Common-Anode Versus Common-Cathode LED Displays 648		
<b>9-3</b>	Liquid-Crystal Displays 649		
	Driving an LCD 650		
	Types of LCDs 651		
<b>9-4</b>	Encoders 653		
	Priority Encoders 655		
	74147 Decimal-to-BCD Priority Encoder 655		
	Switch Encoder 656		
<b>9-5</b>	Troubleshooting 659		
<b>9-6</b>	Multiplexers (Data Selectors) 662		
	Basic Two-Input Multiplexer 663		
	Four-Input Multiplexer 664		
	Eight-Input Multiplexer 664		
	Quad Two-Input MUX (74ALS157/ HC157) 666		
<b>9-7</b>	Multiplexer Applications 668		
	Data Routing 668		
	Parallel-to-Serial Conversion 669		
	Operation Sequencing 669		
	Logic Function Generation 672		
<b>9-8</b>	Demultiplexers (Data Distributors) 673		
	1-Line-to-8-Line Demultiplexer 674		
	Security Monitoring System 675		
	Synchronous Data Transmission System 677		
	Time Division Multiplexing 679		
<b>9-9</b>	More Troubleshooting 683		
<b>9-10</b>	Magnitude Comparator 687		
	Data Inputs 688		
	Outputs 688		
	Cascading Inputs 688		
	Applications 689		
<b>9-11</b>	Code Converters 690		
	Basic Idea 691		
	Conversion Process 691		
	Circuit Implementation 692		
	Other Code Converter Implementations 694		
<b>9-12</b>	Data Busing 694		
<b>9-13</b>	The 74ALS173/HC173 Tristate Register 696		
<b>9-14</b>	Data Bus Operation 698		
	Data Transfer Operation 699		

	Bus Signals	700
	Simplified Bus Timing Diagram	701
	Expanding the Bus	701
	Simplified Bus Representation	703
	Bidirectional Busing	704
<b>9-15</b>	Decoders Using HDL	705
<b>9-16</b>	The HDL 7-Segment Decoder/ Driver	709
<b>9-17</b>	Encoders Using HDL	712
<b>9-18</b>	HDL Multiplexers and Demultiplexers	716
<b>9-19</b>	HDL Magnitude Comparators	720
<b>9-20</b>	HDL Code Converters	721

## **CHAPTER 10 Digital System Projects Using HDL 744**

---

<b>10-1</b>	Small-Project Management	746
	Definition	746
	Strategic Planning/Problem Decomposition	746
	Synthesis and Testing	747
	System Integration and Testing	747
<b>10-2</b>	Stepper Motor Driver Project	747
	Problem Definition	748
	Strategic Planning/Problem Decomposition	749
	Synthesis and Testing	750
<b>10-3</b>	Keypad Encoder Project	755
	Problem Analysis	755
	Strategic Planning/Problem Decomposition	757
<b>10-4</b>	Digital Clock Project	761
	Top-Down Hierarchical Design	764
	Building the Blocks from the Bottom Up	766
	MOD-12 Design	769
	Combining Blocks Graphically	773
	Combining Blocks Using Only HDL	774
<b>10-5</b>	Microwave Oven Project	778
	Definition of the Project	779
	Strategic Planning/Problem Decomposition	780
	Synthesis/Integration and Testing	784
<b>10-6</b>	Frequency Counter Project	785

## **CHAPTER 11 Interfacing with the Analog World 794**

---

<b>11-1</b>	Review of Digital Versus Analog	795
<b>11-2</b>	Digital-to-Analog Conversion	797
	Analog Output	799
	Input Weights	799
	Resolution (Step Size)	800
	Percentage Resolution	801
	What Does Resolution Mean?	802
	Bipolar DACs	804
<b>11-3</b>	DAC Circuitry	804
	Conversion Accuracy	806
	DAC with Current Output	806
	R/2R Ladder	808
<b>11-4</b>	DAC Specifications	810
	Resolution	810
	Accuracy	810
	Offset Error	811
	Settling Time	811
	Monotonicity	811
<b>11-5</b>	An Integrated-Circuit DAC	812
<b>11-6</b>	DAC Applications	813
	Control	813
	Automatic Testing	813
	Signal Reconstruction	813
	A/D Conversion	813
	Digital Amplitude Control	813
	Serial DACs	814
<b>11-7</b>	Troubleshooting DACs	814
<b>11-8</b>	Analog-to-Digital Conversion	816
<b>11-9</b>	Digital-Ramp ADC	817
	A/D Resolution and Accuracy	820
	Conversion Time, $t_C$	821
<b>11-10</b>	Data Acquisition	822
	Reconstructing a Digitized Signal	824
	Aliasing	825
	Serial ADCs	826
<b>11-11</b>	Successive-Approximation ADC	826
	Conversion Time	829
	An Actual IC: The ADC0804 Successive- Approximation ADC	829

---

11-12	Flash ADCs	834	
	Conversion Time	836	
11-13	Other A/D Conversion Methods	836	
	Dual-Slope Integrating ADC	837	
	Voltage-to-Frequency ADC	838	
	Sigma/Delta Modulation	838	
	Pipelined ADC	840	
11-14	Typical ADC Architectures for Applications	842	
11-15	Sample-and-Hold Circuits	843	
11-16	Multiplexing	844	
11-17	Digital Signal Processing (DSP)	845	
	Digital Filtering	846	
11-18	Applications of Analog Interfacing	849	
	Data Acquisition Systems	849	
	Digital Camera	850	
	Digital Cellular Telephone	850	
<b>CHAPTER 12 Memory Devices</b>		<b>866</b>	
12-1	Memory Terminology	868	
12-2	General Memory Operation	872	
	Address Inputs	873	
	The $\overline{WE}$ Input	873	
	Output Enable ( $OE$ )	874	
	Memory Enable	874	
12-3	CPU–Memory Connections	875	
12-4	Read-Only Memories	877	
	ROM Block Diagram	877	
	The Read Operation	878	
12-5	ROM Architecture	879	
	Register Array	880	
	Address Decoders	880	
	Output Buffers	880	
12-6	ROM Timing	881	
12-7	Types of ROMs	882	
	Mask-Programmed ROM	883	
	Programmable ROMs (PROMs)	885	
	Erasable Programmable ROM (EPROM)	886	
	Electrically Erasable PROM (EEPROM)	887	
12-8	Flash Memory	889	
	A Typical CMOS Flash Memory IC	890	
	Flash Technology: NOR and NAND	891	
12-9	ROM Applications	894	
	Embedded Microcontroller Program Memory	894	
	Data Transfer and Portability	894	
	Bootstrap Memory	894	
	Data Tables	895	
	Data Converter	895	
	Function Generator	895	
12-10	Semiconductor RAM	896	
12-11	RAM Architecture	897	
	Read Operation	898	
	Write Operation	898	
	Chip Select	898	
	Common Input/Output Pins	898	
12-12	Static RAM (SRAM)	899	
	Static-RAM Timing	900	
	Read Cycle	900	
	Write Cycle	902	
12-13	Dynamic RAM (DRAM)	902	
12-14	Dynamic RAM Structure and Operation	904	
	Address Multiplexing	905	
12-15	DRAM Read/Write Cycles	909	
	DRAM Read Cycle	909	
	DRAM Write Cycle	910	
12-16	DRAM Refreshing	910	
12-17	DRAM Technology	913	
	Memory Modules	913	
	FPM DRAM	914	
	EDO DRAM	914	
	SDRAM	914	
	DDRSDRAM	914	
12-18	Other Memory Technologies	915	
	Magnetic Storage	915	
	Optical Memory	916	
	Phase Change Ram (PRAM)	917	
	Ferroelectric RAM (FRAM)	917	
12-19	Expanding Word Size and Capacity	917	
	Expanding Word Size	918	
	Expanding Capacity	920	
	Incomplete Address Decoding	923	
	Combining DRAM Chips	924	

- 12-20 Special Memory Functions 925
  - Cache Memory 926
  - First-In, First-Out Memory (FIFO) 927
  - Circular Buffers 928

## **CHAPTER 13 Programmable Logic Device Architectures 940**

---

- 13-1 Digital Systems Family Tree 942
  - More on PLDs 944
- 13-2 Fundamentals of PLD Circuitry 948
  - PLD Symbolology 949

- 13-3 PLD Architectures 950
  - PROMs 950
  - Programmable Array Logic (PAL) 951
  - Field Programmable Logic Array (FPLA) 954
  - Generic Array Logic (GAL) 954
- 13-4 The Altera MAX and MAX II Families 955
- 13-5 Generations of HCPLDs 958

**Glossary 962**

**Answers to Selected Problems 975**

**Index of ICs 983**

**Index 986**

---

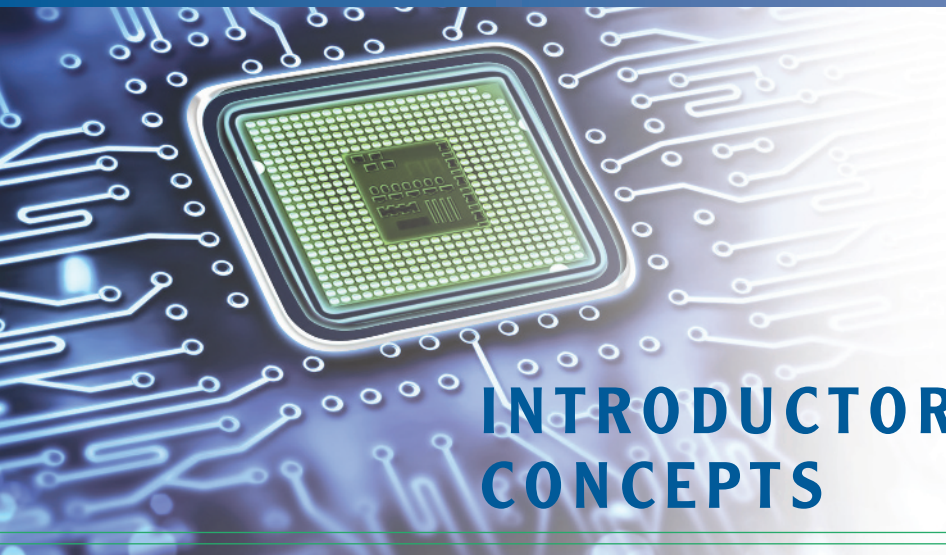
*To you, Cap, for loving me for so long; and for the million  
and one ways you brighten the lives of everyone you touch.*  
—RJT

*To my wife and best friend, Kris, who has sacrificed the most  
to complete this work. To our children John and Brooke,  
Brad and Amber, Blake and Tashi, Matt and Tamara, Katie  
and Matthew, and to our grandchildren Jersey, Judah, and  
the two we have yet to meet, who are in production along  
with this book.*

—NSW

*To my expanding family, Marita, David, Ryan, Christy,  
Jeannie, Taylor, Micah, Brayden, and Lorelei.*

—GLM



# INTRODUCTORY CONCEPTS

## ■ OUTLINE

- |     |  |      |  |
|-----|--|------|--|
| 1-1 | Introduction to Digital 1s and 0s      | 1-7  | Representing Signals with Numeric Quantities |
| 1-2 | Digital Signals                        | 1-8  | Parallel and Serial Transmission             |
| 1-3 | Logic Circuits and Evolving Technology | 1-9  | Memory                                       |
| 1-4 | Numerical Representations              | 1-10 | Digital Computers                            |
| 1-5 | Digital and Analog Systems             |      |  |
| 1-6 | Digital Number Systems                 |      |  |

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Distinguish between analog and digital representations.
- Describe how information can be represented using just two states (1s and 0s).
- Cite the advantages and drawbacks of digital techniques compared with analog.
- Describe the purpose of analog-to-digital converters (ADCs) and digital-to-analog converters (DACs).
- Recognize the basic characteristics of the binary number system.
- Convert a binary number to its decimal equivalent.
- Count in the binary number system.
- Identify typical digital signals.
- Identify a timing diagram.
- State the differences between parallel and serial transmission.
- Describe the property of memory.
- Describe the major parts of a digital computer and understand their functions.
- Distinguish among microcomputers, microprocessors, and microcontrollers.

## ■ INTRODUCTION

In today's world, the term *digital* has become part of our everyday vocabulary because of the dramatic way that digital circuits and digital techniques have become so widely used in almost all areas of life: computers, automation, robots, medical science and technology, transportation, telecommunications, entertainment, space exploration, and on and on. You are about to begin an exciting educational journey in which you will discover the fundamental principles, concepts, and operations that are common to all digital systems, from the simplest on/off switch to the most complex computer.

This chapter will introduce many of the underlying concepts that you will encounter as you learn more about your digital world. As new terms and concepts are presented, you will be directed to the chapters later in the text that expand and clarify the points. We want you to realize just how deeply digital systems impact your life. Then we want you to wonder how they work and how you might use digital systems to make the future better.

Let's go through a typical example of starting a day. The alarm clock wakes me up and I look at the time of day displayed on big bright



seven-segment LEDs (see Chapter 9). The digital alarm has compared the time of day with my alarm setting and when they were equal it activated the alarm (see Chapter 10). The alarm is “latched” on until I reset it with “off” or “snooze”(see Chapter 5 for latches). I go to the bathroom and decide to weigh myself before showering. The bathroom scales respond to the tap of my toe by awaking from its sleep mode, clearing the digital display and waiting for me to step on. It measures my weight and displays it in pounds. After a few seconds, it goes back to sleep. I grab my cordless shaver from the charger. A digital circuit inside the shaver has been controlling the charging cycle. I pick up my electric toothbrush. It can operate in three modes or “states” depending on how many times I push the button (see state machines in Chapter 7). It also keeps track of how long I brush and signals every 30 seconds in a 2 minute brush cycle. This is all controlled by a digital system inside the toothbrush hand-piece. I flip on the closet light. It has an energy saver feature that turns it off in case I forget, thanks to a small digital circuit in the light bulb (see interfacing in Chapter 8). I walk into my bedroom and turn the lights on low using the dimmer switch. The dimmer switch is an old analog circuit, but the new LED light bulbs can still be dimmed by it! This is because of a digital circuit inside the LED light bulb that controls the LEDs (see pulse width modulation in Chapter 11). I disconnect my cell phone from its charger. What a digital miracle I am holding in my hand!

I have not left the bedroom and already my life has been touched by seven digital systems. We could continue but you get the idea. Digital systems are everywhere around you and new applications are constantly being developed. All of the digital systems in the world are built from a surprisingly small number of basic circuits or building blocks. There are many instances of each block in most systems but only a few different blocks. This book will introduce you to those basic digital circuits and help you to understand the purpose, role, capabilities, and limitations of each one. Then you can use your innovation skills and the knowledge from this book to meet the next new demand.

## 1-1 INTRODUCTION TO DIGITAL 1s AND 0s

---

### OUTCOMES

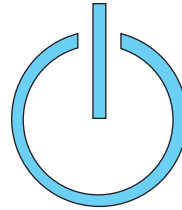
*Upon completion of this section, you will be able to:*

- Correlate new terms with their definition.
- Identify two states and assign a digit to each.
- Correlate each state with its representation in a given circuit.
- Recognize which state will activate a device in a given system.
- Identify the state of a digital signal under various physical conditions.
- Assign proper names to signals in a digital system.

Digital systems deal with things that are in one of two distinct states. The easiest example is anything that is either on or off. If you look at many devices today, you will find that the on/off switch is a single push button with the symbol shown in **Figure 1-1**. This icon represents a 1 and a 0, the numerical digits used to describe the two states in a digital system. We use numeric digits 0 and 1 to represent the two states off and on, respectively. Since there are only two digits, we call them **binary digits**, or **bits**. It is often said that digital systems are just a bunch of 1s and 0s and that is pretty

---

**FIGURE 1-1** The ubiquitous on/off symbol.



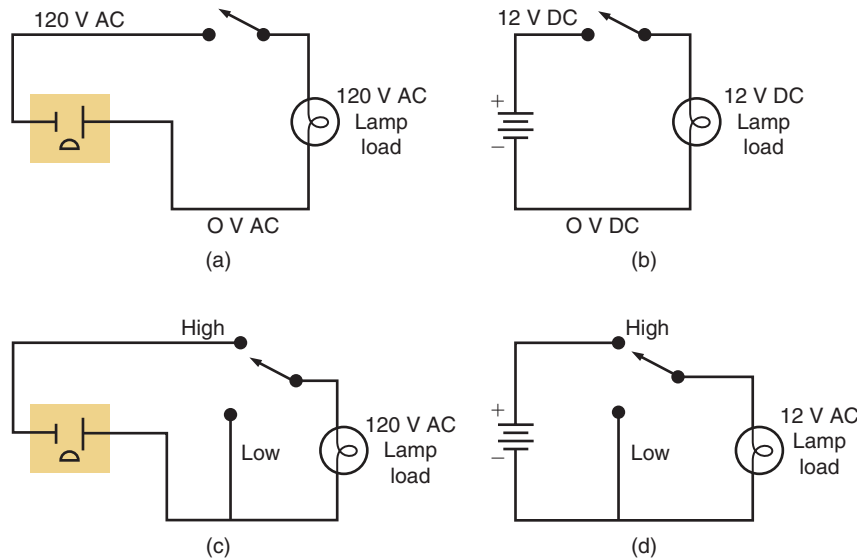
accurate. When we organize groups of numeric digits, we can create number systems and number systems are very powerful ways to represent things. As can be seen from all the digital systems around us, a lot can be done with just two possible states when circuits that can represent these two states are strategically organized.

Let's try to identify some things that must be categorized in one of two states in a system that is familiar to everyone: the automobile. The doors are either locked or unlocked. There is no such thing as being partially locked. We could also say a door is either open or closed. Now we know that a door can be partially opened, but in an automotive system the important thing to know is when the door is completely closed and safely latched. One state is considered to be closed and latched, while the other state is anything from slightly ajar to wide open. The parking brake is either set (engaged to any degree) or it is not set (completely disengaged). The engine is either running (at any speed) or it is not running. A button on the trunk lid is either pressed or not pressed. On some cars, opening the trunk when the engine is running requires the parking brake to be set, the doors unlocked, and the trunk button to be pressed. When the engine is not running the trunk can be opened whenever the trunk button is pressed and the doors are unlocked. Digital circuits observe the state of each component and make a "logical" decision to either open or not open the trunk. For this reason, these conditions are often referred to as **logic states**.

After the two states of a system component are defined, one of the digital values (1 or 0) is assigned to each state. For example, on a Ford perhaps a door that is open may be assigned a 1 (closed = 0), but on a Lexus a door that is open may be assigned a state of 0 (closed = 1). In Chapter 3, we will discuss naming conventions for digital signals that help avoid confusion regarding the meaning of 1s and 0s in any system.

How are the states of 1 and 0 represented electrically in a digital system? The answer depends on the technology of the electrical system but the simplest answer is that a 0 is generally represented by a low voltage (close to 0 V) and a 1 is generally represented by a higher voltage. Consider, as an example, common electrical circuits in a home and in an automobile. In electrical systems, a voltage must be applied to a complete circuit to cause current to flow through the active device and "turn it on." **Figure 1-2(a)** demonstrates a light bulb in your home which requires 110 V AC (alternating current) to turn the light on. When no voltage is applied (0 volts AC), the light is off. Any light bulb in your car requires 12 V DC (direct current) to turn the light on and 0 V DC to turn it off, as demonstrated in **Figure 1-2(b)**. The two systems are very similar but the technology of the systems differs. Consequently, the representations of a HIGH state (i.e., higher voltage) must match the system. In these simple wiring examples, the HIGH voltage is either connected to or disconnected from the lamp. A more accurate model of a digital logic circuit reflects that the output is always connected to either the source of the high voltage (HIGH state) or the source of the low voltage (LOW state). **Figures 1-2(c)** and **(d)** illustrate how this would look

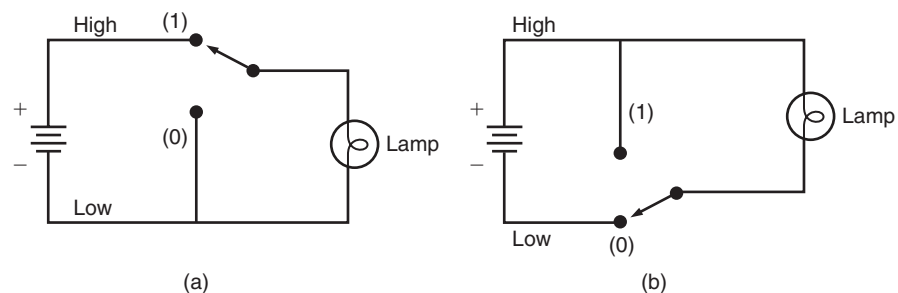
**FIGURE 1-2** (a) Typical 120 V AC house wiring; (b) typical 12 V DC automotive wiring; (c) 120 V AC model of a logic circuit; (d) 12 V AC model of a logic circuit.



for a simple light circuit. Chapter 8 will thoroughly explain why digital logic circuits operate like Figures 1-2(c) and (d) rather than like simple electrical wiring in your home or car, as depicted in Figures 1-2(a) and (b). The main point is that a 0 is typically represented by the LOW voltage or value near 0 V. The state designated as 1 is typically represented by a HIGH voltage and the value of that voltage depends on the technology of the system. These values of HIGH and LOW are often referred to as **logic levels**.

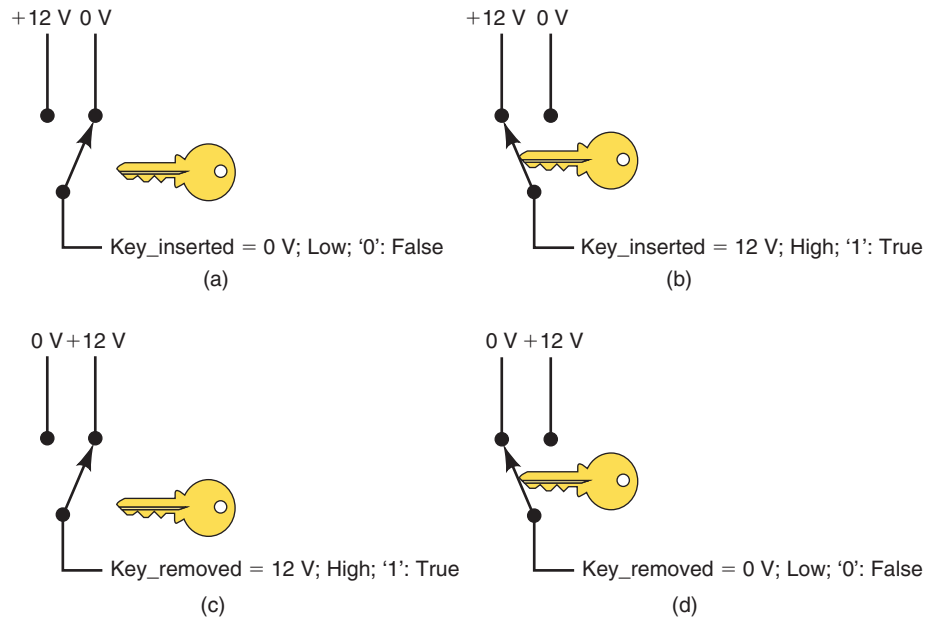
Some digital devices are activated by applying a HIGH, while others are activated by applying a LOW. **Figure 1-3** demonstrates these two scenarios for a simple light circuit. Notice that in Figure 1-3(a) the switch supplies the HIGH by connecting the voltage source which supplies current from the battery to the light and activates the light. In Figure 1-3(b), the switch supplies the LOW by connecting the return path from the light to the battery in order to activate the light. In Chapter 3, we will further investigate this concept of a device being active-HIGH or active-LOW.

**FIGURE 1-3** (a) Applying HIGH turns the lamp ON; (b) applying LOW turns the lamp ON.



Sensors that serve as inputs to digital systems also can be wired in many different ways. For example, consider a circuit that can determine if the key to a car has been inserted into the ignition switch. As we are often reminded, this piece of information is used to sound an alarm if the car door is opened when the key is still in the ignition. **Figure 1-4** demonstrates two possible ways to wire this switch and the affect each method has on the meaning of the digital output level. In Figure 1-4(a), the contacts are open, producing a LOW when no key is present. When the key is inserted, as in Figure 1-4(b), it pushes contact points to the +12 V position, producing a

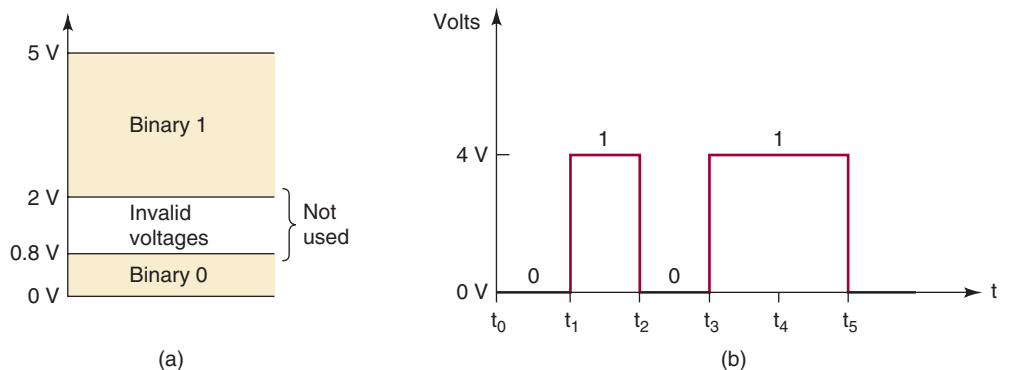
**FIGURE 1-4** Physical conditions, logic levels, and signal labels: (a) false that key is inserted, (b) true that key is inserted, (c) true that key is removed, (d) false that key is removed.



HIGH at the output. A good label for the output signal from this circuit would be *key\_inserted* because the logic level of HIGH represents the state of 1 or true. *Key\_inserted* is true when the output is HIGH. Contrast this circuit with Figure 1-4(c) in which the switch contacts are wired in the opposite way. In this case, inserting the key produces a LOW (Figure 1-4(d)) and removing the key produces a HIGH (Figure 1-4(c)). A good label for this signal is *key\_removed* because it is true that the key is removed when the output is HIGH. The name of the signal describes a physical condition which should be true when the level is HIGH or 1. Chapters 3 and 4 will expand on these concepts using HIGHS and LOWs to activate/deactivate other circuits. This is fundamental to understanding all digital systems.

Now that we know that 1s are represented by a HIGH voltage and 0s by LOW voltage, all that remains is defining how high the voltage must be to be considered a 1 and how low a voltage must be to be considered a 0. The answer to this question also depends on the technology used to implement the digital system. Electronic digital systems have gone through many changes as technology has advanced. But the principles of representing 1s and 0s remain the same. In all systems, a defined range of higher voltages is acceptable as a HIGH (1). Another defined range of lower voltages is acceptable as a LOW (0). In between is a range of voltages that is considered neither HIGH nor LOW. Voltages in this range are considered invalid. **Figure 1-5**

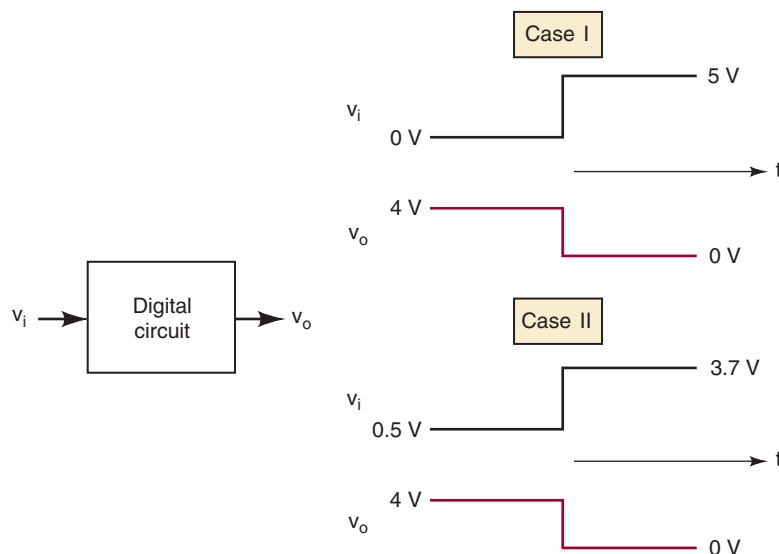
**FIGURE 1-5** Logic levels and timing (a) typical voltage ranges for a given technology of digital circuits. (b) a graph of signal levels changing over time.



demonstrates this concept for 5-volt logic systems that were based on bipolar transistor technology. Figure 1-5a indicates that in order for circuits using this technology to recognize the input as a '1' it must be a voltage greater than two but less than five. The input voltage must be less than 0.8 V to recognize it as a '0'. In the evolution of digital systems, various technologies such as electromechanical switches (relays), vacuum tubes, bipolar transistors, and MOSFET transistors have been used to implement digital logic circuits, each with their own characteristic definition of how to represent a 1 and a 0.

It is quite common and often necessary to depict the activity of a logic level over time. We called this a **timing diagram**. Figure 1-5(b) represents a typical digital waveform for the voltage ranges defined in part (a). The time axis is labeled at specific points in time,  $t_1$ ,  $t_2$ , ...  $t_5$ . Notice that the HIGH voltage level between  $t_1$  and  $t_2$  is at 4 V. In digital systems, the exact value of a voltage is not important. A HIGH voltage of 3.7 V or 4.3 V would represent the exact same information. Likewise, a LOW voltage of 0.3 V represents the same information as 0 V. This points out a significant difference between analog and digital systems. In an analog system, the exact voltage is important. For example, if the analog voltage coming from a sensor is proportional to temperature, then 3.7 V would represent a different value of temperature than 4.3 V. In other words, the voltage carries significant information in the analog system. Circuits that can preserve exact voltages are much more complicated than digital circuits that simply need to recognize a voltage in one of two ranges. Digital circuits are designed to produce output voltages that fall within the prescribed 0 and 1 voltage ranges such as those defined in Figure 1-5. Likewise, digital circuits are designed to respond predictably to input voltages that are within the defined 0 and 1 ranges. What this means is that a digital circuit will respond in the same way to all input voltages that fall within the allowed 0 range; similarly, it will not distinguish between input voltages that lie within the allowed 1 range. To illustrate, Figure 1-6 represents a typical digital circuit with input  $v_i$  and output  $v_o$ . The output is shown for two different input signal waveforms. Note that  $v_o$  is the same for both cases because the two input waveforms, while differing in their exact voltage levels, are at the same binary levels.

**FIGURE 1-6** A digital circuit responds to an input's binary level (0 or 1) and not to its actual voltage.



**OUTCOME  
ASSESSMENT  
QUESTIONS\***

1. What are the two numeric digits used to represent states in a digital system?
2. What are the two terms used to represent the two logic levels?
3. What is the abbreviation for binary digit?
4. Which binary digit value is typically represented by low (near-zero) voltage?
5. What voltage represents the binary digit value of 1?
6. Which logic level is typically assigned a value of 1?
7. What is the logic level produced in Figure 1-4(a) when the key is removed?
8. According to Figure 1-5, what is the lowest voltage that would be recognized as a logic 1?
9. According to Figure 1-5, what is the highest voltage that would be recognized as a logic 0?
10. According to Figure 1-5, how would a voltage of 1.0 V be recognized?

## 1-2 DIGITAL SIGNALS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Determine if a waveform is periodic or not.
- Measure period and frequency.
- Measure duty cycle.
- Identify events and classify edges as rising or falling.
- Recognize valid/invalid inputs.
- Recognize a timing diagram.

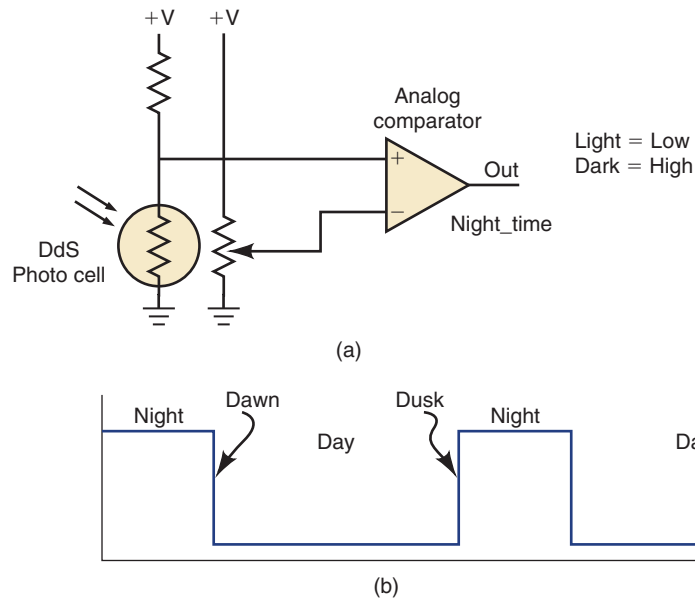
Suppose we have a light sensor that is intended to turn on the streetlights at night. An example of a circuit that could perform this task is shown in **Figure 1-7(a)**. Chapter 8 will explain more about analog comparators. This circuit's output will produce a logic 1 when no light is present (darkness). It outputs a logic 0 (0 V) when a certain level of light is present. The signal that comes from the sensor should be labelled with a signal name. It will always be either a 1 (HIGH) or a 0 (LOW) but it should be named something that informs the user about the physical condition represented by the signal. For example, if this sensor is intended to control a street lamp, the name of the output signal should be something like "night\_time". When the signal is "1" it is true that it is nighttime. When the output is "0" we can say that it is false that it is nighttime. Chapter 3 will expand on these labelling techniques.

When a circuit like this is placed in service, it will output a 1 at night and a 0 during the day. At some point around dawn, it will change from a 1 to a 0. Around dusk, it will change from a 0 to a 1. This transition between the two states is called an **edge**. *At dawn, when the signal proceeds from HIGH to LOW, it is considered a **falling edge**, or **negative edge**.* Graphing the logic state over time tells us something about the operation of the system. **Figure 1-7(b)** shows the graph over time of the output of the light sensor.

---

\*Answers to outcome assessment questions are found at the end of the chapter in which they occur.

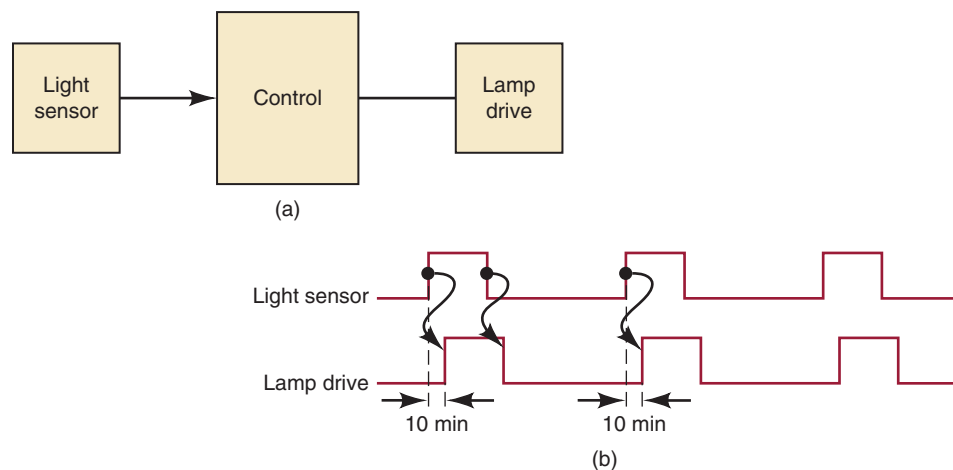
**FIGURE 1-7** (a) Darkness sensor; (b) a timing diagram of the output.



### Need for Timing

Digital circuits have inputs that are in one of two states: 1 or 0. The outputs are also either producing a 1 or a 0. In the previous section, we learned that 1s and 0s are represented by prescribed voltages and that voltage changes on the inputs result in changes in the output voltage. It can be very helpful to show the relationship between changes at the input and changes at the output in order to demonstrate the operation of the system. This means the logic states must be observed over time. Timing diagrams show the relationship, over time, between many digital “signals.” It is very important that you understand timing diagrams and can relate them to physical events in a digital circuit. For example, assume there is a circuit represented by the block diagram in **Figure 1-8** that detects the “edge” at dawn, waits 10 minutes, and then turns off the streetlamp. **Figure 1-8(b)** is a timing diagram which shows the input to the circuit as well as the output. From this diagram, we can determine the relationship between the two signals. Notice the curvy arrows. They are used to indicate the cause-and-effect relationship between input and output signals.

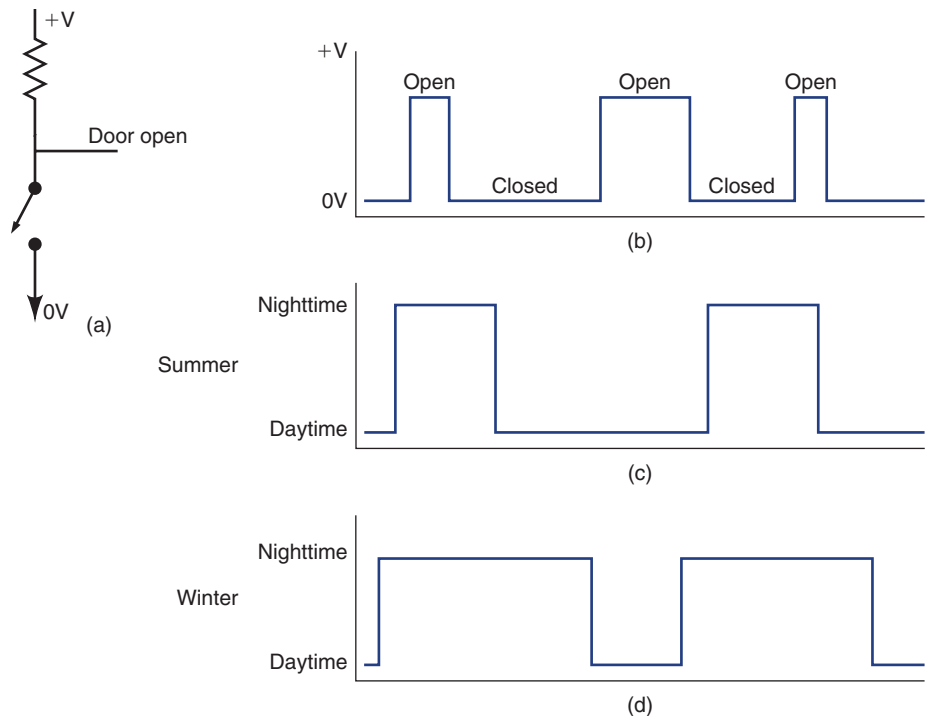
**FIGURE 1-8** Timing diagram with input and output.



## Highs and Lows Over Time

Think about a common digital input to a system that you operate all the time. A microwave oven has a switch in the door that tells the system whether the door is closed or open. This switch could be wired in several ways. Let's assume the switch is open when the door is open and closed when the door is closed. It is wired as shown in **Figure 1-9(a)**. The timing diagram in **Figure 1-9(b)** depicts the condition of the door over time. We can look at the diagram at any point in time and know the physical condition of the door.

**FIGURE 1-9** A periodic versus periodic signals with duty cycle: (a) microwave door sensor, (b) aperiodic operation of oven door, (c) periodic day/night signal summer short nights, (d) periodic day/night signal winter short days.



## Periodic/Aperiodic

Opening and closing a microwave oven door is something that happens at completely irregular intervals. If we tried to measure the length of time the door stands open, each measurement would be different. There is no regularity to the cycle of opening and closing a door. There would be no fixed period of time between events. Therefore, it is referred to as an **aperiodic** signal. Let's contrast this digital signal with a sensor that turns on and off the streetlights. To make our point in this analogy, we will disregard the effects of weather and assume cloudless days. We also assume the sensor makes one clean transition at dawn and another clean transition at dusk. The sensor tells us whether it is day or night. A timing diagram of this sensor would look like **Figure 1-9(c)** in June (central United States). In December the sensor timing diagram would look more like **Figure 1-9(d)**.

## Period/Frequency

Notice the similarities and differences in the timing waveforms of **Figures 1-9(c)** and **(d)**. The length of daylight time is different between June and December, but the time it takes for an entire day is always the same. The earth always takes 24 hours for one rotation or one complete cycle. When you measure from dawn to dawn it is always the same, regardless of the season. Likewise, notice



that the amount of time from dusk to dusk is always the same as well. When a system operates such that the time for one complete cycle is always constant, it is called a *periodic* system. Certainly, the rotation of the earth is periodic and its period is always 24 hours. The period of any wave can be defined as the amount of time per cycle (seconds/cycle). The frequency of a periodic wave is defined as the number of cycles per unit time (cycles/second). In other words, frequency ( $F$ ) and period ( $T$ ) are reciprocals.

$$F = 1/T \quad T = 1/F$$

## Duty Cycle

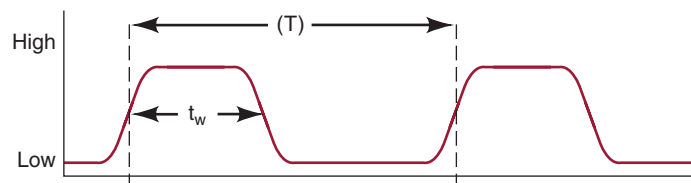
The length of daylight time and nighttime varies with the seasons but the period remains the same. If we want to measure how much of the time a digital signal is in its “active” state, then we must think about the purpose of the digital signal. In our example of a sensor whose duty is to turn the streetlights ON, we would say that this sensor is on-duty during the night when (in this example) the sensor is HIGH. The duty cycle of the street light would be the percentage of time it is dark over the course of an entire day.

$$\text{Duty Cycle} = \text{Active pulse Width/Period} = t_w/T$$

## Transitions

Just as you realize that the night does not change instantly into day, it is true that no digital signal can truly change instantly from LOW to HIGH. There is a time of transition. It is common to declare the transition as happening when the signal is half way between the two states. Measurements are taken from the 50% point of waveform. For example, the width of the HIGH pulse is measured as shown in **Figure 1-10**. The period  $T$  is also measured from 50% points as shown. Chapter 5 will have more to say about measurements of these transition times and the period of a digital waveform.

**FIGURE 1-10** Measuring pulse width and period.



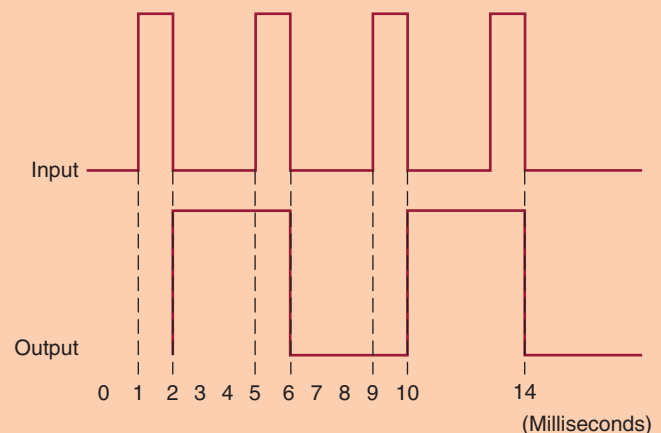
## Edges/Events

Whenever you have a system with only two states, the only thing that can be considered an “event” is when the system changes states. A transition from LOW to HIGH or HIGH to LOW is considered an “event” in digital systems. On timing diagrams, these transitions appear as sharp “edges.” Some events are rising edges and some are falling edges. We will learn in Chapter 3 that there are circuits that respond to HIGH levels (active HIGH) and circuits that respond to LOW levels (active LOW). Circuits that respond to a particular level are often considered to be *level triggered*. Other types of digital circuits respond to either rising edges or falling edges. These are called *edge triggered* circuits. Chapter 5 will introduce edge triggered devices.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Draw a timing diagram showing when a person is “on duty” over an entire week. Begin on Monday morning. The diagram will have one input representing the day/night cycle (assume equinox where length of day = length of night dawn 6:00 am, dusk 6:00 pm) and one output which goes HIGH representing when a person is “on duty.” Assume they work a typical 8–5 job Monday–Friday with Saturday and Sunday off.
2. Is the “on\_duty” waveform in the diagram from the previous question periodic or aperiodic?
3. Refer to **Figure 1-11**
  - (a) Is the input waveform periodic?
  - (b) What is the period of the input waveform in sec?
  - (c) What is the active-HIGH duty cycle of the input waveform?
  - (d) What is the frequency of the waveform in Hz?
  - (e) What type of event on the input causes a change on the output?
  - (f) What is the period of the output waveform in sec?
  - (g) What is the frequency of the output waveform in Hz?

**FIGURE 1-11** Outcome assessment question.



## 1-3 LOGIC CIRCUITS AND EVOLVING TECHNOLOGY

### OUTCOMES

Upon completion of this section, you will be able to:

- Identify acceptable digital logic levels for a given technology.
- Recognize terms describing the currently prevalent and legacy technologies for digital circuits.

### Logic Circuits

The manner in which a digital circuit responds to an input is referred to as the circuit’s *logic*. Each type of digital circuit obeys a certain set of logic rules. For this reason, digital circuits are also called **logic circuits**. We will use both terms interchangeably throughout the text. In Chapter 3, we will see more clearly what is meant by a circuit’s “logic.”

We will be studying all the types of logic circuits that are currently used in digital systems. Initially, our attention will be focused only on the logical operation that these circuits perform—that is, the relationship between

the circuit inputs and outputs. We will defer any discussion of the internal circuit operation of these logic circuits until after we have developed an understanding of their logical operation.

## Digital Integrated Circuits

Digital circuits of today's technology are primarily implemented using very sophisticated integrated circuits (ICs) that are electronically configured or tailor-made for their application. Many technologies of the past are completely obsolete. For example, the vacuum tube logic circuits would never be used today for a number of reasons such as too big, too much power, and the vacuum tubes are very hard to find. Occasionally, it makes sense to use a mature technology where it is economical and the parts will be available over the life of a product. For example, most of the ICs that made up digital systems in the 1970s are no longer being manufactured but are still available on the market from large left-over inventories. These devices will, on rare occasion, be used in a new product and they are still used for laboratory instruction for digital courses in high school and college. Throughout this text, we will try to provide enough information about the range of technologies to allow you to learn using simple devices from the past and yet introduce you to the fundamentals necessary to use the tools of the future.

Today the most common technology used to implement digital circuits (including the vast majority of computer hardware) is **CMOS**, which stands for **Complementary Metal-Oxide Semiconductor**. Other technologies have been relegated to much smaller niches in the marketplace. Prior to the advancement of CMOS technology, bipolar transistor technology was king and had a profound influence on digital systems. The major logic family that sprung from bipolar technology is referred to as **TTL (Transistor/Transistor Logic)**. You will learn about the various IC technologies, their characteristics, and relative advantages and disadvantages in Chapter 8.

### OUTCOME ASSESSMENT QUESTIONS

1. *True or false:* The exact value of an input voltage is critical for a digital circuit.
2. Can a digital circuit produce the same output voltage for different input voltage and current values?
3. A digital circuit is also referred to as a \_\_\_\_\_ circuit.
4. The most prevalent technology used for digital circuits today is abbreviated \_\_\_\_\_.
5. This acronym stands for \_\_\_\_\_.
6. Legacy systems of the past used a technology abbreviated as \_\_\_\_\_.
7. The type of transistor used in legacy systems was the \_\_\_\_\_ transistor.

## 1-4 NUMERICAL REPRESENTATIONS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Discriminate between digital and analog representations.
- Identify examples of each type of representation.

In science, technology, business, and, in fact, most other fields of endeavor, we are constantly dealing with *quantities*. Quantities are measured, monitored, recorded, manipulated arithmetically, observed, or in some other way utilized in most physical systems. It is important when dealing with various quantities that we be able to represent their values efficiently and accurately. There are basically two ways of representing the numerical value of quantities: *analog* and *digital*.

## Analog Representations

In **analog representation** a quantity is represented by a continuously variable, proportional indicator. An example is an automobile speedometer from the classic muscle cars of the 1960s and 1970s. The deflection of the needle is proportional to the speed of the car and follows any changes that occur as the vehicle speeds up or slows down. On older cars, a flexible mechanical shaft connected the transmission to the speedometer on the dashboard. It is interesting to note that on newer cars, the analog representation is usually preferred even though speed is now measured digitally.

Thermometers before the digital revolution used analog representation to measure temperature, and many are still in use today. Mercury thermometers use a column of mercury whose height is proportional to temperature. These devices are being phased out of the market because of environmental concerns, but nonetheless they are an excellent example of analog representation. Another example is an outdoor thermometer on which the position of the pointer rotates around a dial as a metal coil expands and contracts with temperature changes. The position of the pointer is proportional to the temperature. Regardless of how small the change in temperature, there will be a proportional change in the indication.

In these two examples the physical quantities (speed and temperature) are being coupled to an indicator by purely mechanical means. In electrical analog systems, the physical quantity that is being measured or processed is converted to a proportional voltage or current (electrical signal). This voltage or current is then used by the system for display, processing, or control purposes.

No matter how they are represented, analog quantities have an important characteristic: *they can vary over a continuous range of values*. The automobile speed can have *any* value between zero and, say, 100 mph. Similarly, the temperature indicated by an analog thermometer can have any value from  $-20^{\circ}$  to  $120^{\circ}$  Fahrenheit.

## Digital Representations

In **digital representation** the quantities are represented not by continuously variable indicators but by symbols called *digits*. As an example, consider a digital indoor/outdoor thermometer. It has four digits and can measure changes of  $0.1^{\circ}$ . The actual temperature gradually increases from, say,  $72.0$  to  $72.1$  but the digital representation changes suddenly from  $72.0$  to  $72.1$ . In other words, this digital representation of outdoor temperature changes in *discrete* steps, as compared with the analog representation of temperature provided by a fluid column or metal coil thermometer, where the reading changes continuously.

The major difference between analog and digital quantities, then, can be simply stated as follows:

Analog  $\equiv$  continuous

Digital  $\equiv$  discrete (step by step)

---

Because of the discrete nature of digital representations, there is no ambiguity when reading the value of a digital quantity, whereas the value of an analog quantity is often open to interpretation. In practice, when we take a measurement of an analog quantity, we always “round” to a convenient level of precision. In other words, we digitize the quantity. The digital representation is the result of assigning a number of limited precision to a continuously variable quantity.

The world around us is full of physical variables that are constantly changing. If we can measure these variables and represent them as a digital quantity, then we can record, arithmetically manipulate, or in some other way use these quantities to control things.

**EXAMPLE 1-1**

Which of the following involve analog quantities and which involve digital quantities?

- (a) Elevation using a ladder
- (b) Elevation using a ramp
- (c) Current flowing from an electrical outlet through a motor
- (d) Height of a child measured by a yard stick ruler
- (e) Height of a child measured by putting a mark on the wall
- (f) Amount of rocks in a bucket
- (g) Amount of sand in a bucket
- (h) Volume of water in a bucket

**Solution**

- (a) Digital
- (b) Analog
- (c) Analog
- (d) Digital: measured to nearest  $\frac{1}{8}$  inch
- (e) Analog
- (f) Digital: can only increase/decrease by one rock
- (g) Digital: can only increase/decrease by discrete grains of sand
- (h) Analog: (unless you want to get to the nanotechnology level!)

**OUTCOME ASSESSMENT QUESTIONS**

1. Which method of representing quantities involves discrete steps?
2. Which method of representing quantities is continuously variable?
3. Identify each as digital or analog representation:
  - (a) Time of day using a sundial
  - (b) Time of day using your cell phone
  - (c) Volume level of your flat-screen television
  - (d) Volume level of vacuum tube radio
  - (e) Measuring the circumference of a basketball in millimeters
  - (f) Measuring the circumference of a basketball by wrapping a string around it and cutting the string to length

## 1-5 DIGITAL AND ANALOG SYSTEMS

---

### OUTCOMES

Upon completion of this section, you will be able to:

- Identify advantages of digital techniques.
- Identify limitations of digital techniques.

A **digital system** is a combination of devices designed to manipulate logical information or physical quantities that are represented in digital form; that is, the quantities can take on only discrete values. These devices are most often electronic, but they can also be mechanical, magnetic, or pneumatic. Some of the more familiar digital systems include digital computers and calculators, digital audio and video equipment, and the telecommunications system.

An **analog system** contains devices that manipulate physical quantities that are represented in analog form. In an analog system, the quantities can vary over a continuous range of values. For example, the amplitude of the output signal to the speaker in a radio receiver can have any value between zero and its maximum limit.

### Advantages of Digital Techniques

An increasing majority of applications in electronics, as well as in most other technologies, use digital techniques to perform operations that were once performed using analog methods. The chief reasons for the shift to digital technology are:

1. *Digital systems are generally easier to design.* The circuits used in digital systems are *switching circuits*, where *exact* values of voltage or current are not important, only the range (HIGH or LOW) in which they fall.
  2. *Information storage is easy.* This is accomplished by special devices and circuits that can latch onto digital information and hold it for as long as necessary, and mass storage techniques that can store billions of bits of information in a relatively small physical space. Analog storage capabilities are, by contrast, extremely limited.
  3. *Accuracy and precision are easier to maintain throughout the system.* Once a signal is digitized, the degree to which it deteriorates is predictable and more easily contained within acceptable limits. In analog systems, the voltage and current signals tend to be distorted by the effects of temperature, humidity, and component tolerance variations in the circuits that process the signal.
  4. *Operations can be programmed.* It is fairly easy to design digital systems whose operation is controlled by a set of stored instructions called a *program*. Analog systems can also be *programmed*, but the variety and the complexity of the available operations are severely limited.
  5. *Digital circuits are less affected by noise.* Spurious fluctuations in voltage (noise) are not as critical in digital systems because the exact value of a voltage is not important, as long as the noise is not large enough to prevent us from distinguishing a HIGH from a LOW.
  6. *More digital circuitry can be fabricated on IC chips.* It is true that analog circuitry has also benefited from the tremendous development of IC technology, but its relative complexity and its use of devices that cannot be economically integrated (high-value capacitors, precision resistors, inductors, transformers) have prevented analog systems from achieving the same high degree of integration.
-

## Limitations of Digital Techniques

There are really very few drawbacks when using digital techniques. The two biggest problems are:

**The real world is analog and digitizing always introduces some error.**  
**Processing digitized signals takes time.**

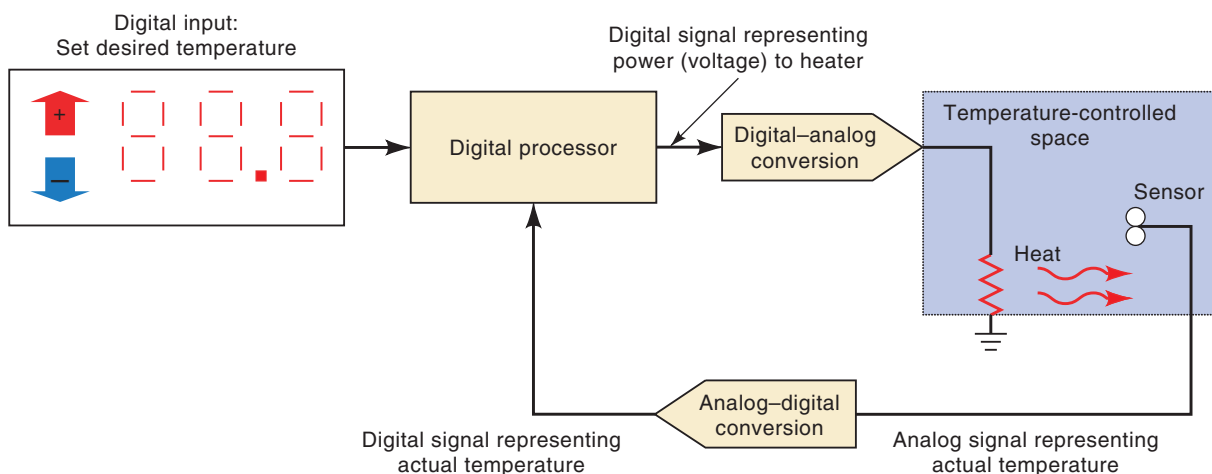
Most physical quantities are analog in nature, and these quantities are often the inputs and outputs that are being monitored, operated on, and controlled by a system. Some examples are temperature, pressure, position, velocity, liquid level, flow rate, and so on. We are in the habit of expressing these quantities *digitally*, such as when we say that the temperature is  $64^\circ$  ( $63.8^\circ$  when we want to be more precise), but we are really making a digital approximation to an inherently analog quantity.

To take advantage of digital techniques when dealing with analog inputs and outputs, four steps must be followed:

1. Convert the physical variable to an electrical signal (analog).
2. Convert the electrical (analog) signal into digital form.
3. Process (operate on) the digital information.
4. Convert the digital outputs back to real-world analog form.

An entire book could be written about step 1 alone. There are many kinds of devices that convert various physical variables into electrical analog signals (sensors). These are used to measure things that are found in our “real” analog world. On your car alone, there are sensors for fluid level (gas tank), temperature (climate control and engine), velocity (speedometer), acceleration (airbag collision detection), pressure (oil, manifold), and flow rate (fuel), to name just a few. Chapter 11 will cover the devices that convert analog to digital.

To illustrate a typical system that uses this approach Figure 1-12 describes a precision temperature regulation system. A user pushes up or down buttons to set the desired temperature in  $0.1^\circ$  increments (digital representation). A temperature sensor in the heated space converts the measured temperature to a proportional voltage. This analog voltage is



**FIGURE 1-12** Diagram of a precision digital temperature control system.

converted to a digital quantity by an **analog-to-digital converter (ADC)**. This value is then compared to the desired value and used to determine a digital value of how much heat is needed. The digital value is converted to an analog quantity (voltage) by a **digital-to-analog converter (DAC)**. This voltage is applied to a heating element, which will produce heat that is related to the voltage applied and will affect the temperature of the space.

### OUTCOME ASSESSMENT QUESTIONS

1. List three advantages of digital techniques.
2. List the two primary limitations of digital techniques.

## 1-6 DIGITAL NUMBER SYSTEMS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Identify the weight of each binary digit.
- Determine the range of binary values given the number of binary digits.
- Interpret binary numbers into decimal.
- Count in binary.

Many number systems are in use in digital technology. The most common are the decimal, binary, and hexadecimal systems. Humans operate using decimal numbers, digital systems operate using binary numbers, and **hexadecimal** is a number system that makes it easier for humans to deal with binary numbers. All three of these number systems are defined and function in the exact same way. Let's start by examining the decimal system. Because it is so familiar, we rarely stop to think about how this number system actually works. Examining its characteristics fully will help you to understand the other systems better.

### Decimal System

The **decimal system** is composed of 10 numerals or symbols. These 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; using these symbols as *digits* of a number, we can express any quantity. The decimal system, also called the *base-10* system because it has 10 digits, has evolved naturally as a result of the fact that people have 10 fingers. In fact, the word *digit* is derived from the Latin word for "finger."

The decimal system is a *positional-value system* in which the value of a digit depends on its position. For example, consider the decimal number 453. We know that the digit 4 actually represents 4 *hundreds*, the 5 represents 5 *tens*, and the 3 represents 3 *units*. In essence, the 4 carries the most weight of the three digits; it is referred to as the *most significant digit (MSD)*. The 3 carries the least weight and is called the *least significant digit (LSD)*.

Consider another example, 27.35. This number is actually equal to 2 tens plus 7 units plus 3 tenths plus 5 hundredths, or  $2 \times 10 + 7 \times 1 + 3 \times 0.1 + 5 \times 0.01$ . The decimal point is used to separate the integer and fractional parts of the number.

More rigorously, the various positions relative to the decimal point carry weights that can be expressed as powers of 10. This is illustrated in



**FIGURE 1-13** Decimal position values as powers of 10.

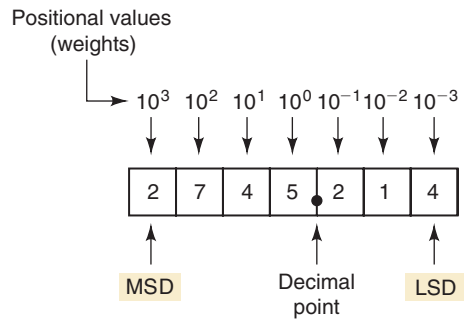


Figure 1-13, where the number 2745.214 is represented. The decimal point separates the **positive** powers of 10 from the negative powers. The number 2745.214 is thus equal to

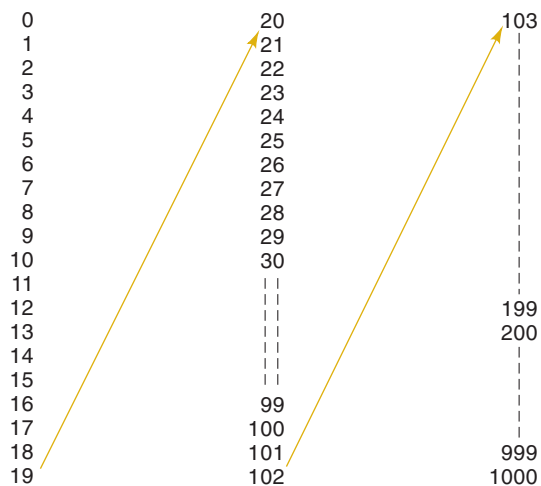
$$(2 \times 10^{+3}) + (7 \times 10^{+2}) + (4 \times 10^1) + (5 \times 10^0) + (2 \times 10^{-1}) + (1 \times 10^{-2}) + (4 \times 10^{-3})$$

In general, any number is simply the sum of the products of each digit value and its positional value.

### Decimal Counting

When counting in the decimal system, we start with 0 in the units position and take each symbol (digit) in progression until we reach 9. Then we add a 1 to the next higher position and start over with 0 in the first position (see Figure 1-14). This process continues until the count of 99 is reached. Then we add a 1 to the third position and start over with 0s in the first two positions. The same pattern is followed continuously as high as we wish to count.

**FIGURE 1-14** Decimal counting.



It is important to note that in decimal counting, the units position (LSD) changes upward with each step in the count, the tens position changes upward every 10 steps in the count, the hundreds position changes upward every 100 steps in the count, and so on.

Another characteristic of the decimal system is that using only two decimal places, we can count through  $10^2 = 100$  different numbers (0 to 99).<sup>\*</sup> With three places we can count through 1000 numbers (0 to 999), and so on. In general, with  $N$  places or digits, we can count through  $10^N$  different numbers, starting with and including zero. The largest number will always be  $10^N - 1$ .

## Binary System

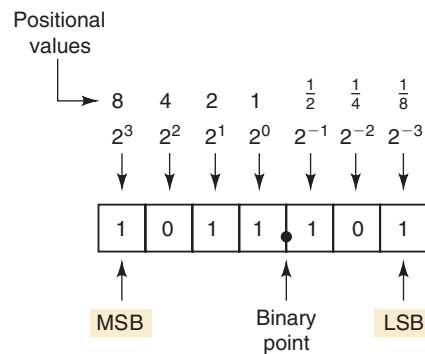
Unfortunately, the decimal number system does not lend itself to convenient implementation in digital systems. For example, it is very difficult to design electronic equipment so that it can work with 10 different voltage levels (each one representing one decimal character, 0 through 9). On the other hand, it is very easy to design simple, accurate electronic circuits that operate with only two voltage levels. For this reason, almost every digital system uses the binary (base-2) number system as the basic number system of its operations. Other number systems are often used to interpret or represent binary quantities for the convenience of the people who work with and use these digital systems.

In the binary system there are only two symbols or possible digit values, 0 and 1. Even so, this base-2 system can be used to represent any quantity that can be represented in decimal or other number systems. In general though, it will take a greater number of binary digits to express a given quantity.

All of the statements made earlier concerning the decimal system are equally applicable to the binary system. The binary system is also a positional-value system, wherein each binary digit has its own value or weight expressed as a power of 2. This is illustrated in Figure 1-15. Here, places to the left of the *binary point* (counterpart of the decimal point) are positive powers of 2 and places to the right are negative powers of 2. The number 1011.101 is shown represented in the figure. To find its equivalent in the decimal system, we simply take the sum of the products of each digit value (0 or 1) and its positional value:

$$\begin{aligned} 1011.101_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &\quad + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) \\ &= 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125 \\ &= 11.625_{10} \end{aligned}$$

**FIGURE 1-15** Binary position values as powers of 2.



<sup>\*</sup> Zero is counted as a number.

Notice in the preceding operation that subscripts (2 and 10) were used to indicate the base in which the particular number is expressed. This convention is used to avoid confusion whenever more than one number system is being employed.

In the binary system, the term *binary digit* is often abbreviated to the term *bit*, which we will use from now on. Thus, in the number expressed in Figure 1-15 there are four bits to the left of the binary point, representing the integer part of the number, and three bits to the right of the binary point, representing the fractional part. The most significant bit (MSB) is the leftmost bit (largest weight). The least significant bit (LSB) is the rightmost bit (smallest weight). These are indicated in Figure 1-15. Here, the MSB has a weight of  $2^3$ ; the LSB has a weight of  $2^{-3}$ . The weights of each digit increase by a factor of 2 as the position moves from right to left.

## Binary Counting

When we deal with binary numbers, we will usually be restricted to a specific number of bits. This restriction is based on the circuitry used to represent these binary numbers. Let's use four-bit binary numbers to illustrate the method for counting in binary.

The sequence (shown in Figure 1-16) begins with all bits at 0; this is called the *zero count*. For each successive count, the units ( $2^0$ ) position *toggles*; that is, it changes from one binary value to the other. Each time the units bit changes from a 1 to a 0, the twos ( $2^1$ ) position will toggle (change states). Each time the twos position changes from 1 to 0, the fours ( $2^2$ ) position will toggle (change states). Likewise, each time the fours position goes from 1 to 0, the eights ( $2^3$ ) position toggles. This same process would be continued for the higher order bit positions if the binary number had more than four bits.

**FIGURE 1-16** Binary counting sequence.

Weights →	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$	Decimal equivalent
	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	2
	0	0	1	1	3
	0	1	0	0	4
	0	1	0	1	5
	0	1	1	0	6
	0	1	1	1	7
	1	0	0	0	8
	1	0	0	1	9
	1	0	1	0	10
	1	0	1	1	11
	1	1	0	0	12
	1	1	0	1	13
	1	1	1	0	14
	1	1	1	1	15

↑  
LSB

The binary counting sequence has an important characteristic, as shown in Figure 1-16. The units bit (LSB) changes either from 0 to 1 or 1 to 0 with *each* count. The second bit (twos position) stays at 0 for two counts, then at 1 for two counts, then at 0 for two counts, and so on. The third bit (fours position) stays at 0 for four counts, then at 1 for four counts, and so on. The fourth bit (eights position) stays at 0 for eight counts, then at 1 for eight counts. If we wanted to count further, we would add more places, and this

pattern would continue with 0s and 1s alternating in groups of  $2^{N-1}$ . For example, using a fifth binary place, the fifth bit would alternate sixteen 0s, then sixteen 1s, and so on.

As we saw for the decimal system, it is also true for the binary system that by using  $N$  bits or places, we can go through  $2^N$  counts. For example, with two bits we can go through  $2^2 = 4$  counts ( $00_2$  through  $11_2$ ); with four bits we can go through  $2^4 = 16$  counts ( $0000_2$  through  $1111_2$ ); and so on. The last count will always be all 1s and is equal to  $2^N - 1$  in the decimal system. For example, using four bits, the last count is  $1111_2 = 2^4 - 1 = 15_{10}$ .

**EXAMPLE 1-2**

What is the largest number that can be represented using eight bits?

**Solution**

$$2^N - 1 = 2^8 - 1 = 255_{10} = 11111111_2.$$

This has been a brief introduction of the binary number system and its relation to the decimal system. We will spend much more time on these two systems and several others in the next chapter.

**OUTCOME ASSESSMENT QUESTIONS**

1. What is the decimal equivalent of  $1101011_2$ ?
2. What is the next binary number following  $10111_2$  in the counting sequence?
3. What is the largest decimal value that can be represented using 12 bits?

## 1-7 REPRESENTING SIGNALS WITH NUMERIC QUANTITIES

### OUTCOMES

*Upon completion of this section, you will be able to:*

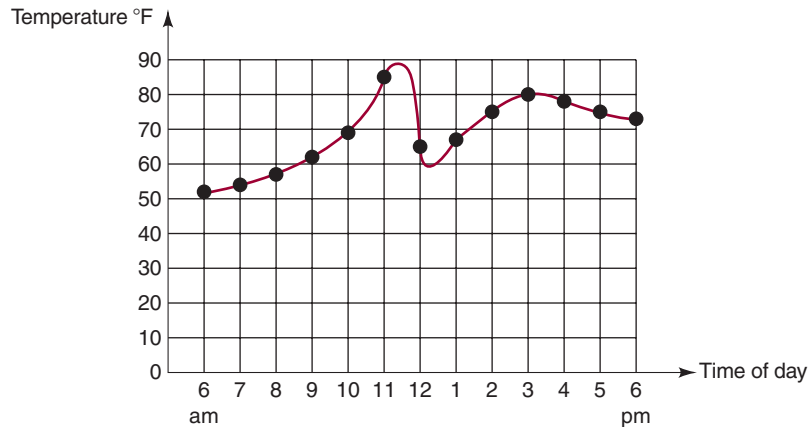
- Represent a continuous signal with a sequence of measurements.
- Evaluate the relative effects of precision in measurement.
- Evaluate the relative effects of frequency of measurement.
- Identify the acceptable range of voltages that represent 1s and 0s in a given digital system.

We will now look at the concept of representing an analog signal as a sequence of digital numbers. From the previous section, we know that any quantity can be represented by a binary number, just as easily as it can be represented by a decimal number. Suppose you are doing a science experiment that requires you to keep a record of temperature changes over a long period of time. You know that the temperature of the air is an analog quantity: continuously variable. However, you also know that temperature usually changes rather slowly, so instead of trying to continuously plot the current temperature, you decide to take a measurement at the top of every hour.

Your temperature measuring device is rather crude and can only measure in 10 degree increments. For example, any temperature between 60

and 69 (inclusive) will read 60 degrees. Any temperature between 70 and 79 (inclusive) will read 70 degrees. **Figure 1-17** shows a continuous plot of the temperature on an early summer day which starts out cool but warms up rapidly around 1:00 pm. After 1:00 pm, a thunderstorm comes through and the temperature drops very suddenly and drastically. **Figure 1-18(a)** shows the data taken from this day's temperature readings at the top of every hour. If we plot the numbers from our table, you can see in **Figure 1-18(b)** that the general shape is close to the analog signal even though some of the detailed slow changes are undetected.

**FIGURE 1-17**  
Temperature data sampled hourly. Red line indicates analog.



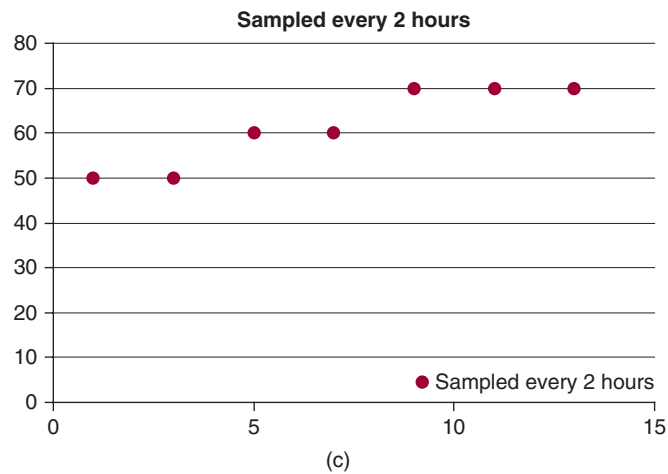
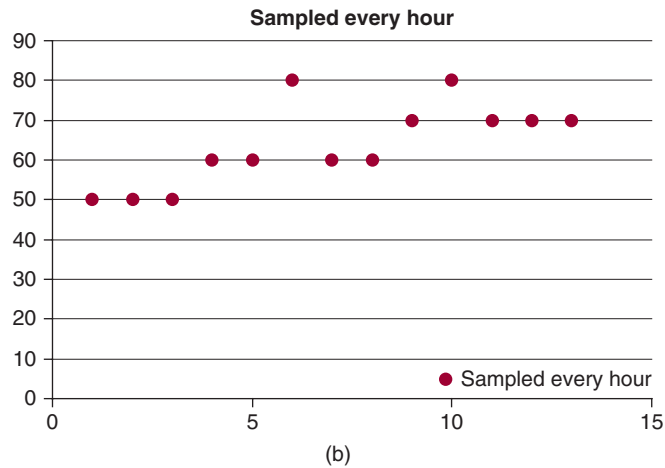
Next let's assume that we arbitrarily decided to measure the temperature every 2 hours starting at 6:00 am. These measurements are also shown in Figure 1-18a. The plot of the measurements taken every two hours is shown in **Figure 1-18c**. Notice it does not look at all like the actual analog signal. Based on these measurements, the temperature for this day appears to be very stable and quite mild. This example is intended to point out several things about representing analog signals as digital quantities.

1. The major event of the day (rapid increase in heat followed by a sudden drop in temperature) all happened in between two samples, so as far as this digital signal is concerned, the event did not happen.
2. The signal is represented as a list of measurements taken at regular intervals. These are called samples.
3. The measurements do not exactly represent the actual value at the time it is sampled due to the limitations of the measuring device (ten degree steps). This is called quantization error.
4. How often a sample is recorded has a huge impact on the accuracy of the reproduction.
5. The more frequently samples are taken, the more accurately the signal is represented.

Chapter 11 will have more to say about digital signals and how they are processed.

Sample number	1	2	3	4	5	6	7	8	9	10	11	12	13
Time of day	6	7	8	9	10	11	12	1	2	3	4	5	6
Sampled every hour	50	50	50	60	60	80	60	60	70	80	70	70	70
Sampled every 2 hours	50		50		60		60		70		70		70
Binary value stored	101	101	101	110	110	1000	110	110	111	1000	111	111	111

(a)



**FIGURE 1-18** Temperature measurements: (a) table of data; (b) graph of hourly samples; (c) graph of samples every 2 hours.

## 1-8 PARALLEL AND SERIAL TRANSMISSION

### OUTCOME

*Upon completion of this section, you will be able to:*

- Discriminate between parallel and serial transfer.

One of the most common operations that occur in any digital system is the transmission of information from one place to another. The information can be transmitted over a distance as small as a fraction of an inch on the same circuit board, or over a distance of many miles when two people are texting each other on different continents. The information that is transmitted is in binary form and is generally represented as voltages at the outputs of a sending circuit that are connected to the inputs of a receiving circuit.

**FIGURE 1-19** (a) Parallel transmission uses one connecting line per bit, and all bits are transmitted simultaneously; (b) serial transmission uses only one signal line, and the individual bits are transmitted serially (one at a time).

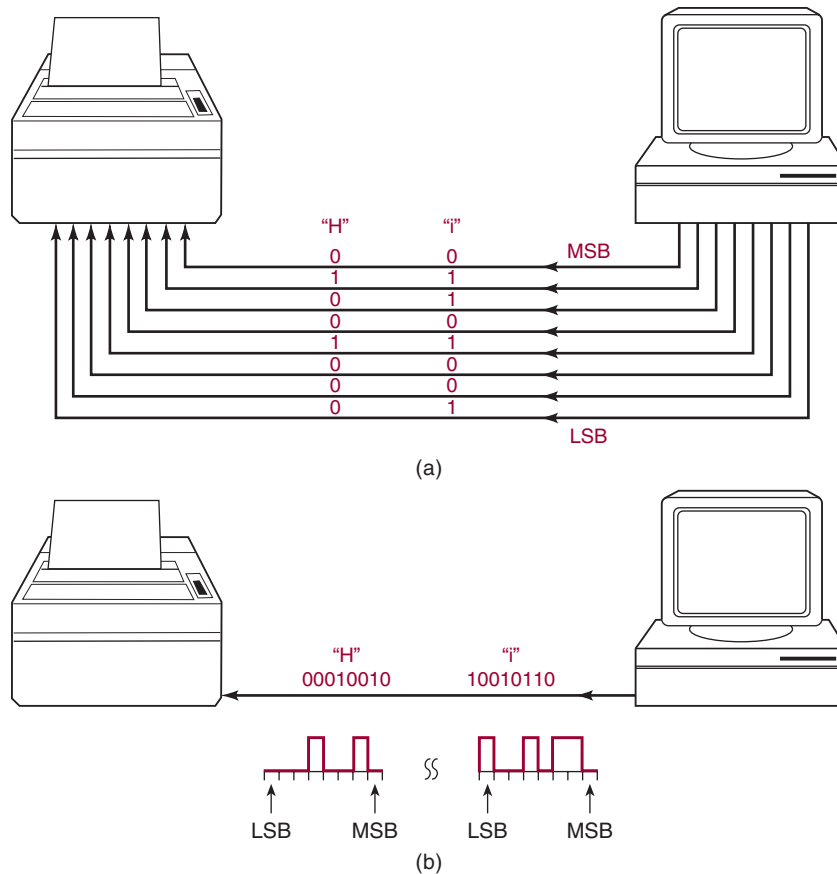


Figure 1-19 illustrates the two basic methods for digital information transmission: **parallel** and **serial**.

Figure 1-19(a) demonstrates parallel transmission of data from a computer to a printer. Parallel printer interfaces were standard in personal computers before the USB (Universal Serial Bus). In this scenario, assume we are trying to print the word “Hi” on the printer. The binary code for “H” is 01001000 and the binary code for “i” is 01101001. Each character (the “H” and the “i”) are made up of eight bits. Using parallel transmission, all eight bits are sent simultaneously over eight wires. The “H” is sent first, followed by the “i.”

Figure 1-19(b) demonstrates serial transmission such as is employed on your computer when using a USB port to send data to a printer. Although the details of the data format are much more complicated for a USB port than we show here, the point is that the data are sent one bit at a time over a single wire. The bits are shown in the diagram as though they were actually moving down the wire in the order shown. The least significant bit of “H” is sent first and the most significant bit of “i” is sent last. Of course, in reality, only one bit can be on the wire at any point in time and time is usually drawn on a graph starting at the left and advancing to the right. This produces a graph of logic bits versus time of the serial transmission called a timing diagram. Notice that in this presentation, the least significant bit is shown on the left because it was sent first.

The principal trade-off between parallel and serial representations is one of speed versus circuit simplicity. The transmission of binary data from one part of a digital system to another can be done more quickly using parallel representation because all the bits are transmitted simultaneously,

while serial representation transmits one bit at a time. On the other hand, parallel requires more signal lines connected between the sender and the receiver of the binary data than does serial. In other words, parallel is faster, and serial requires fewer signal lines. This comparison between parallel and serial methods for representing binary information will be encountered many times in discussions throughout the text.

### OUTCOME ASSESSMENT QUESTION

1. Describe the relative advantages of parallel and serial transmission of binary data.

## 1-9 MEMORY

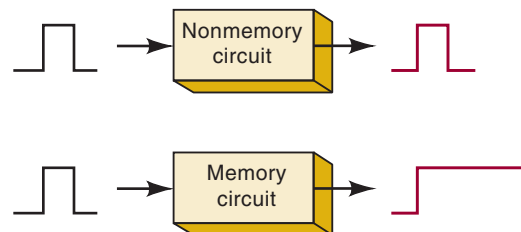
### OUTCOME

Upon completion of this section, you will be able to:

- Articulate the difference between nonmemory and memory circuits.

When an input signal is applied to most devices or circuits, the output somehow changes in response to the input, and when the input signal is removed, the output returns to its original state. These circuits do not exhibit the property of *memory* because their outputs revert back to normal. In digital circuitry certain types of devices and circuits do have memory. When an input is applied to such a circuit, the output will change its state, but it will remain in the new state even after the input is removed. This property of retaining its response to a momentary input is called **memory**. Figure 1-20 illustrates nonmemory and memory operations.

**FIGURE 1-20** Comparison of nonmemory and memory operation.



Memory devices and circuits play an important role in digital systems because they provide a means for storing binary numbers either temporarily or permanently, with the ability to change the stored information at any time. As we shall see, the various memory elements include magnetic and optical types and those that utilize electronic latching circuits (called *latches* and *flip-flops*).

### OUTCOME ASSESSMENT QUESTIONS

1. A nonmemory circuit output always depends on the \_\_\_\_\_ input (past present, or future).
2. A memory circuit output depends on \_\_\_\_\_.



## 1-10 DIGITAL COMPUTERS

---

### OUTCOMES

Upon completion of this section, you will be able to:

- Name the functional blocks of any computer.
- Name the two blocks that make up a central processing unit.
- Explain the primary strategy of improving performance/capability of computers.

Digital techniques have found their way into innumerable areas of technology, but the area of automatic **digital computers** is by far the most notable and most extensive. In simplest terms, *a computer is a system of hardware that performs arithmetic operations, manipulates data (usually in binary form), and makes decisions.*

For the most part, human beings can do whatever computers can do, but computers can do it with much greater speed and accuracy, in spite of the fact that computers perform all their calculations and operations one step at a time. For example, a person walking across the room does not think about which muscle to contract and which muscle to relax, or which direction to go, or even notice the obstacles. Our brains are processing this information all the time in parallel. A computer-controlled robot would need to take in data from sensor 1, then process this information, then output a command to an actuator that moves in a certain way. Then it would have to repeat this process for many other sensors and actuators. Of course, the fact that the computer requires only a few nanoseconds per step makes up for this apparent inefficiency.

A computer is faster and more accurate than people are, but unlike most of us, it must be given a complete set of instructions that tell it *exactly* what to do at each step of its operation. This set of instructions, called a **program**, is prepared by one or more persons for each job the computer is to do. Programs are placed in the computer's memory unit in binary-coded form, with each instruction having a unique code. The computer takes these instruction codes from memory *one at a time* and performs the operation called for by the code.

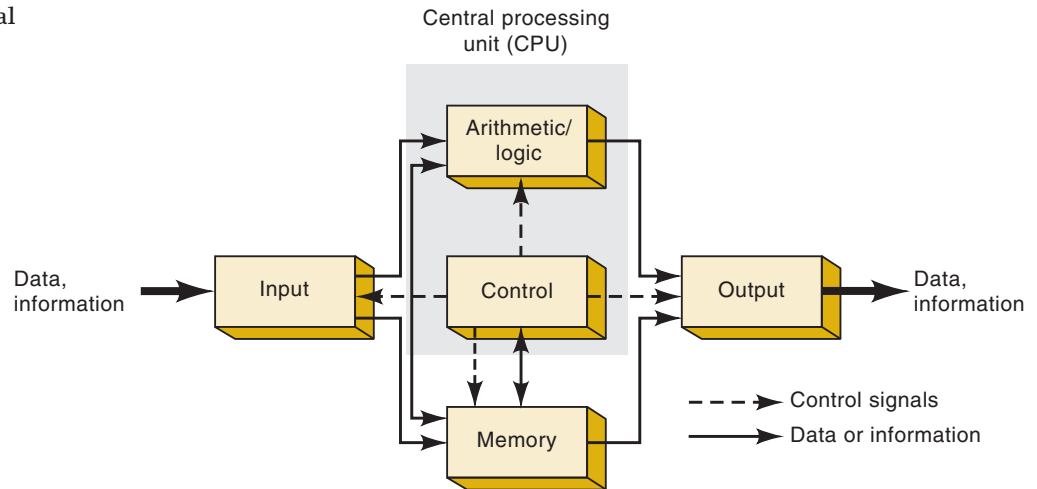
### Major Parts of a Computer

There are several types of computer systems, but each can be broken down into the same functional units. Each unit performs specific functions, and all units function together to carry out the instructions given in the program. Figure 1-21 shows the five major functional parts of a digital computer and their interaction. The solid lines with arrows represent the flow of data and information. The dashed lines with arrows represent the flow of timing and control signals.

The major functions of each unit are as follows:

1. **Input unit.** Through this unit, a complete set of instructions and data is fed into the computer system and into the memory unit, to be stored until needed. The information typically enters the input unit from a keyboard, a disk, or various sensors (in the case of a process control computer).
  2. **Memory unit.** The memory stores the instructions and data received from the input unit. It stores the results of arithmetic operations received from the arithmetic unit. It also supplies information to the output unit.
-

**FIGURE 1-21** Functional diagram of a digital computer.



3. **Control unit.** This unit takes instructions from the memory unit one at a time and interprets them. It then sends appropriate signals to all the other units to cause the specific instruction to be executed.
4. **Arithmetic/logic unit.** All arithmetic calculations and logical decisions are performed in this unit, which can then send results to the memory unit to be stored.
5. **Output unit.** This unit takes data from the memory unit and prints out, displays, or otherwise presents the information to the operator (or process, in the case of a process control computer).

As the diagram in Figure 1-21 shows, the control and arithmetic/logic units are often considered as one unit, called the **central processing unit (CPU)**. The CPU contains all of the circuitry for fetching and interpreting instructions and for controlling and performing the various operations called for by the instructions.

## Types of Computers

There are many ways to categorize computers and many names for different types of computers in each category have evolved. Some old terms are often improperly used to describe new configurations and some new terms are little more than another name for older configurations. We will describe three categories of computers and offer some names of types of computers in each category. The important thing to remember is that all of these systems in all of these categories can be broken down into the five major parts that we have described. These parts are simply arranged differently to optimize the system for a particular purpose.

The central processing unit of the first generation of computers was made up of lots of digital circuits distributed over many circuit boards. That is why in the 1970s when all of the parts of a central processing unit were “integrated” into one fairly small chip, that they were given the name “microprocessor.” These microprocessors were combined with memory, input and output circuits to produce “microcomputers.” With advances in integrated circuit technology, more and more digital circuits could fit into smaller and smaller packages and manufacturers started to offer microcomputers surrounded by specialized support hardware to make it easy to control things—all in a single integrated circuit. These came to be known as microcontrollers.

**HIGH-END COMPUTERS** The high end of computers are the very powerful systems that can handle lots of tasks and produce results very quickly. Some of the names that are often given to these computer systems are *supercomputers*, *clusters*, *servers*, and *mainframes*. Today all of these systems obtain their high speed and large throughput by using a simple strategy of dividing up the work. The overall computer is made up of many powerful microprocessors with local resources (memory, input, output) plus resources of memory and the inputs and outputs that are shared. In other words, they are made up of many computers working together to produce the intended results.

**PERSONAL COMPUTERS** The way we will define personal computer is a general-purpose computer that can run many different applications and is intended for use by individual people. Workstations, desktops, laptops, notebooks, tablets, and even cell phones fall into this category. They have a single processor, though it may be made up of multiple “cores.” Each core is really a CPU or processor that shares the instruction queue called the cache. The cores work together to grab (fetch) the next instruction and execute it as efficiently as possible.

**EMBEDDED COMPUTERS** In spite of all the high-end and personal computers that you see everywhere, the category that claims the most computer applications is the embedded computer market. These are the single chip computers which also contain built-in digital hardware to help it efficiently control things and communicate with other devices. These other devices include analog-to-digital and digital-to-analog converters, pulse width modulators, timer/counters, and serial interface circuits. These computers are embedded in so many commercial products that it is almost easier to name the products that do not contain a microcontroller. You have probably never seen one on a mousetrap but if someone ever builds a better mousetrap, it will probably include a microcontroller.

The unique thing about embedded microcontrollers is that they are an integral part of the inner workings of a system. They are not seen, though they are usually at the center of every function in the system. The designer gives an embedded controller one program of instructions and it is expected to run that program for the rest of their life.

## Memory

The fundamental purpose of all memory devices is to store a group of 1s and 0s. This fact begs two questions: How many 1s and 0s are in a group? How many groups can be stored in the memory device? Chapter 12 will explain all the different configurations of memory devices and help you to decide which size/shape will meet your needs. The goal for all memory devices (and most other electronic components) is to make it smaller, lower power, faster, and less expensive. Using different technologies, we can optimize some of these features, but no single technology is best at all of them. Consequently, computer systems use a combination of memory technologies. For example, many computers still use mechanical hard drives for long-term storage. This technology stores 1s and 0s (data) as magnetic fields on a rotating disk. The newer solid-state hard drives store data using Flash technology on special transistors. The data on these devices must be remembered, even if power is removed from the storage device. The working memory of your computer where the apps are accessed when they are active is made from dynamic RAM technology which stores data on capacitors. The working memory must store a very large number of bits. The video memory must be very

---

fast. However, it is not a problem if working memory and video memory lose their data whenever power is turned off.

To understand memory terms, think of how a typical dormitory or hotel is laid out. Each floor has the same number of rooms and each room has a number on the door. This number (usually referred to as the address) is used to locate one particular room out of all the rooms in the dormitory. The lower order digits of the room number identify where it is located within a given floor and the higher order digit identifies the floor. The number of people, characteristics of people, and assortment of stuff is unique in each room. In like manner, memory systems have an array of places to store information. These are called memory locations and they are identified with an address. The address will have upper digits that specify a general area in the device along with lower digits that identify a particular location in that area. The contents of any given memory location will be a binary number referred to as data. The number of digits that can be stored will be the same in each location, but the value of the data stored there can be unique. Chapter 12 will have much more to say about the many types of memory devices.

### Digital Progress Today and Tomorrow

Why should the contents of this book and the subject of digital systems matter to you? To answer that question, just think about the inventions that have changed how we do things since the year 2000. If you cannot think of ten examples, use the internet to look for inventions of the twenty-first century. As you look at lists generated by lots of people, ask yourself one simple question: Is any part of this invention a digital system? In almost every instance, the answer is YES! If you want to be a part of great innovations of the next 50 years, knowledge of digital systems will help you. The building blocks of digital systems have been known and understood for decades. As technology makes those building blocks faster, smaller, lower power, and less expensive, you will be able to find new ways to use them to solve the world's problems.

#### OUTCOME ASSESSMENT QUESTIONS

1. Name the five major functional units of a computer.
2. Which two units make up the CPU?
3. What is the primary strategy of improving the capabilities, speed, and throughput of computer systems?
4. What category boasts the largest number of computer applications?
5. If you discover the greatest invention of the twenty-first century, what technology will almost certainly be involved?

### SUMMARY

1. The two basic ways of representing the numerical value of physical quantities are analog (continuous) and digital (discrete).
2. Most quantities in the real world are analog, but digital techniques are generally superior to analog techniques, and most of the predicted advances will be in the digital realm.
3. The binary number system (0 and 1) is the basic system used in digital technology.
4. Digital or logic circuits operate on voltages that fall in prescribed ranges that represent either a binary 0 or a binary 1.

5. The two basic ways to transfer digital information are parallel—all bits simultaneously—and serial—one bit at a time.
6. The main parts of all computers are the input, control, memory, arithmetic/logic, and output units.
7. The combination of the arithmetic/logic unit and the control unit makes up the CPU (central processing unit).
8. General-purpose computer systems today are made up of many CPU cores (microprocessors) that work together.
9. A microcontroller is a microcomputer especially designed for dedicated (not general-purpose) control applications.

## IMPORTANT TERMS<sup>†</sup>

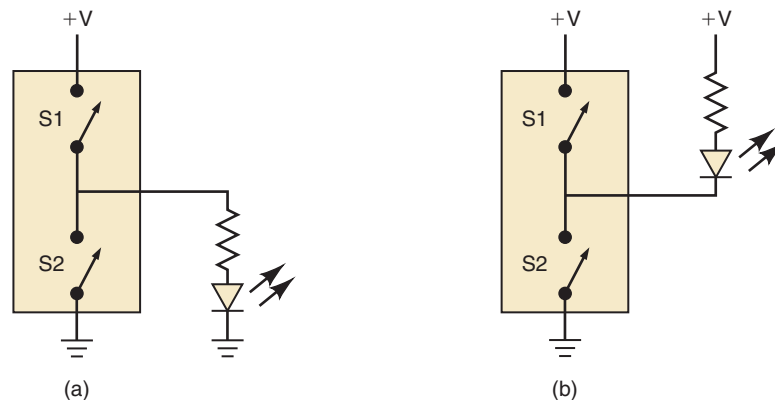
binary digit	edge triggered	parallel transmission
bit	logic circuits	serial transmission
logic states	CMOS	memory
logic levels	TTL	digital computer
timing diagram	analog representation	input unit
edge	digital representation	memory unit
negative edge	digital system	control unit
falling edge	analog system	arithmetic/logic unit
aperiodic	analog-to-digital converter (ADC)	output unit
periodic	digital-to-analog converter (DAC)	central processing unit (CPU)
frequency ( $F$ )	hexadecimal system	microprocessor
period ( $T$ )	decimal system	microcomputer
duty cycle	binary system	microcontroller
event		
level triggered		

## PROBLEMS

### SECTION 1-1

- 1-1. In Figure 1-22a, what is the logic level that must be output to turn on the LED?
- 1-2. In Figure 1-22b, what is the logic level that must be output to turn on the LED?
- 1-3. In Figure 1-22a, which switch must be closed to turn on the LED?
- 1-4. In Figure 1-22b, which switch must be closed to turn on the LED?

FIGURE 1-22 Problem 1-1.

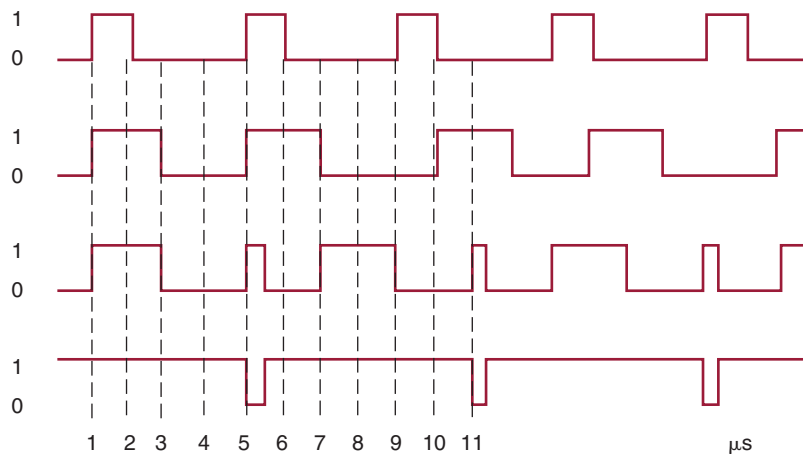


<sup>†</sup>These terms can be found in **boldface** type in the chapter and are defined in the Glossary at the end of the book. This applies to all chapters.

## SECTIONS 1-2 AND 1-3

- 1-5. Create a good label (name) for each signal described below:
- \*A sensor outputs a LOW when the elevator door is closed
  - A streetlight sensor outputs a HIGH when it detects daylight
  - \*A passenger seat sensor goes LOW when the seat is empty
  - A temperature sensor goes HIGH when the radiator fluid is critically hot
- 1-6.\* Draw two cycles of the timing diagram for a digital signal that continuously alternates between 0.2 V (binary 0) for 2 ms and 4.4 V (binary 1) for 4 ms.
- 1-7.\* Measure the period, frequency, and duty cycle of the waveform in 1-6. Assume the active part of the waveform is HIGH.
- 1-8. Draw two cycles of the timing diagram for a signal that alternates between 0.3 V (binary 0) for 5 ms and 3.9 V (binary 1) for 2 ms.
- 1-9. Measure the period, frequency, and duty cycle of the waveform in 1-8. Assume the active part of the waveform is LOW.
- 1-10. Label each waveform in Figure 1-23: periodic/aperiodic. For those that are periodic, measure  $T$ ,  $F$ , and duty cycle (assume active HIGH).

FIGURE 1-23 Problem 1-10.



## SECTIONS 1-4 AND 1-5

- 1-11.\* Which of the following are analog quantities, and which are digital?
- Number of atoms in a sample of material
  - Altitude of an aircraft
  - Pressure in a bicycle tire
  - Current through a speaker
  - Your age measured in years

---

\* Answers to problems marked with an asterisk can be found in the back of the text.

- 1-12. Which of the following are analog quantities, and which are digital?
- (a) Width of a piece of lumber
  - (b) The amount of time before the oven buzzer goes off
  - (c) The time of day displayed on a quartz watch
  - (d) Elevation measured by counting steps on a staircase
  - (e) Elevation measured by a point on a ramp

### SECTION 1-6

- 1-13.\* Convert the following binary numbers to their equivalent decimal values.
- (a)  $11001_2$
  - (b)  $1001.1001_2$
  - (c)  $10011011001.10110_2$
- 1-14. Convert the following binary numbers to decimal.
- (a)  $10011_2$
  - (b)  $1100.0101_2$
  - (c)  $10011100100.10010_2$
- 1-15.\* Using three bits, show the binary counting sequence from 000 to 111.
- 1-16. Using six bits, show the binary counting sequence from 000000 to 111111.
- 1-17.\* What is the maximum number that we can count up to using 10 bits?
- 1-18. What is the maximum number that we can count up to using 14 bits?
- 1-19.\* How many bits are needed to count up to a maximum of 511?
- 1-20. How many bits are needed to count up to a maximum of 63?

### SECTION 1-7

- 1-21. Which of the following will increase/decrease the quality of the digital signal?
- (a) Increasing the time between samples
  - (b) Increasing the number of bits in each sample
  - (c) Increasing the sampling frequency

### SECTION 1-8

- 1-22.\* Suppose that the decimal integer values from 0 to 15 are to be transmitted in binary.
- (a) How many lines will be needed if parallel representation is used?
  - (b) How many lines will be needed if serial representation is used?

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

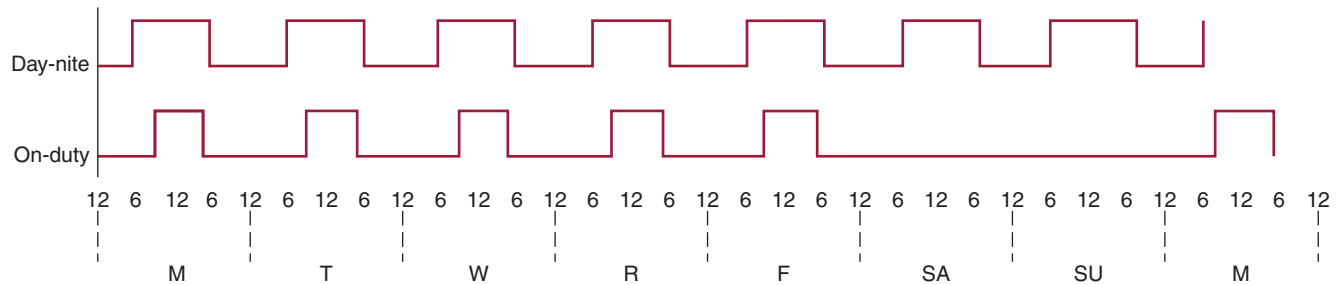
---

### SECTION 1-1

1. 0 and 1    2. LOW and HIGH    3. Bit    4. 0    5. It depends on the technology of the system.    6. HIGH    7. LOW    8. 2.0 V    9. 0.8 V    10. Invalid

**SECTION 1-2**

1. See Figure 1-24    2. Aperiodic    3. (a) Yes (b) 0.004 sec. (c) 25%  
 (d) 250 Hz (e) Falling edge (f) 0.008 sec. (g) 125 Hz.



**FIGURE 1-24** Section 1-2, outcome assessment 1-1.

**SECTION 1-3**

1. False    2. YES    3. Logic    4. CMOS    5. Complementary Metal-Oxide Semiconductor    6. TTL    7. Bipolar

**SECTION 1-4**

1. Digital    2. Analog    3. (a) Analog, (b) Digital, (c) Digital, (d) Analog, (e) Digital, (f) Analog.

**SECTION 1-5**

1. Easier to design; easier to store information; greater accuracy and precision; programmability; less affected by noise; higher degree of integration    2. Real-world physical quantities are analog. Digital processing takes time.

**SECTION 1-6**

1.  $107_{10}$     2.  $11000_2$     3.  $4095_{10}$

**SECTION 1-7**

1. A sequence of binary numbers, representing the signal's value measured at regular intervals

**SECTION 1-8**

1. Parallel is faster; serial requires only one signal line.

**SECTION 1-9**

1. present    2. past outputs and present inputs.

**SECTION 1-10**

1. Input, output, memory, arithmetic/logic, control    2. Control and arithmetic/logic





# NUMBER SYSTEMS AND CODES

## ■ OUTLINE

- 2-1 Binary-to-Decimal Conversions
- 2-2 Decimal-to-Binary Conversions
- 2-3 Hexadecimal Number System
- 2-4 BCD Code
- 2-5 The Gray Code
- 2-6 Putting It All Together
- 2-7 The Byte, Nibble, and Word
- 2-8 Alphanumeric Codes
- 2-9 Parity Method for Error Detection
- 2-10 Applications

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Convert a number from one number system (decimal, binary, hexadecimal) to its equivalent in one of the other number systems.
- Cite the advantages of the hexadecimal number system.
- Count in hexadecimal.
- Represent decimal numbers using the BCD code; cite the pros and cons of using BCD.
- Explain the difference between BCD and straight binary.
- Explain the purpose of alphanumeric codes such as the ASCII code.
- Explain the parity method for error detection.
- Determine the parity bit to be attached to a digital data string.

## ■ INTRODUCTION

The binary number system is the most important one in digital systems, but several others are also important. The decimal system is important because it is universally used to represent quantities outside a digital system. This means that there will be situations where decimal values must be converted to binary values before they are entered into the digital system. For example, when you punch a decimal number into your hand calculator (or computer), the circuitry inside the machine converts the decimal number to a binary value.

Likewise, there will be situations where the binary values at the outputs of a digital system must be converted to decimal values for presentation to the outside world. For example, your calculator (or computer) uses binary numbers to calculate answers to a problem and then converts the answers to decimal digits before displaying them.

As you will see, it is not easy to simply look at a large binary number and convert it to its equivalent decimal value. It is very tedious to enter a long sequence of 1s and 0s on a keypad, or to write large binary numbers on a piece of paper. It is especially difficult to try to convey a binary quantity while speaking to someone. The hexadecimal (base-16) number system has become a standard way of communicating numeric values in digital systems. The great advantage is that hexadecimal numbers can be converted easily to and from binary. You will find that many advanced computer tools, which are designed to help software developers troubleshoot or debug their programs, use the hexadecimal number system to enter numbers that are stored in the computer as binary and display them again as hexadecimal.

Other methods of representing decimal quantities with binary-encoded digits have been devised that are not truly number systems but offer the ease of conversion between the binary code and the decimal number system. This is referred to as binary-coded decimal. Quantities and patterns of bits might be represented by any of these methods in any given system

and throughout the written material that supports the system, so it is very important that you are able to interpret values in any system and convert between any of these numeric representations. Other codes that use 1s and 0s to represent things such as alphanumeric characters will be covered because they are so common in digital systems.

## 2-1 BINARY-TO-DECIMAL CONVERSIONS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Convert binary numbers to decimal.
- Identify the weight of each bit in a binary number.

As explained in Chapter 1, the binary number system is a positional system where each binary digit (bit) carries a certain weight based on its position relative to the LSB. Any binary number can be converted to its decimal equivalent simply by summing together the weights of the various positions in the binary number that contain a 1. To illustrate, let's change  $11011_2$  to its decimal equivalent.

$$\begin{array}{cccccc} 1 & 1 & 0 & 1 & 1 & 1_2 \\ 2^4 & + & 2^3 & + & 0 & + & 2^1 & + & 2^0 & = & 16 & + & 8 & + & 2 & + & 1 \\ & & & & & & & & & = & 27_{10} \end{array}$$

Let's try another example with a greater number of bits:

$$\begin{array}{cccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1_2 = \\ 2^7 & + & 0 & + & 2^5 & + & 2^4 & + & 0 & + & 2^2 & + & 0 & + & 2^0 & = & 181_{10} \end{array}$$

Note that the procedure is to find the weights (i.e., powers of 2) for each bit position that contains a 1, and then to add them up. Also note that the MSB has a weight of  $2^7$  even though it is the eighth bit; this is because the LSB is the first bit and has a weight of  $2^0$ .

Another method of binary-to-decimal conversion that avoids the addition of large numbers and keeping track of column weights is called the double-dabble method. The procedure is as follows:

1. Write down the left-most 1 in the binary number.
2. Double it and add the next bit to the right.
3. Write down the result under the next bit.
4. Continue with steps 2 and 3 until finished with the binary number.

Let's use the same binary numbers to verify this method.

Given:            1     1     0     1     1<sub>2</sub>

Results:        1 × 2 = 2

$$\begin{array}{r} + 1 \\ \hline 3 \times 2 = 6 \end{array}$$

$$\begin{array}{r} + 0 \\ \hline 6 \times 2 = 12 \end{array}$$

$$\begin{array}{r} + 1 \\ \hline 13 \times 2 = 26 \end{array}$$

$$\begin{array}{r} + 1 \\ \hline 27_{10} \end{array}$$

Given:            1     0     1     1     0     1     0     1<sub>2</sub>

Results:        1 → 2 → 5 → 11 → 22 → 45 → 90 → 181<sub>10</sub>

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Convert  $100011011011_2$  to its decimal equivalent by adding the products of digits and weights.
2. What is the weight of the MSB of a 16-bit number?
3. Repeat the conversion in Question 1 using the double-dabble method.

## 2-2 DECIMAL-TO-BINARY CONVERSIONS

### OUTCOMES

Upon completion of this section, you will be able to:

- Convert decimal numbers to binary.
- Identify the number of bits needed for a given range of values.
- Identify the range of values given the number of bits.

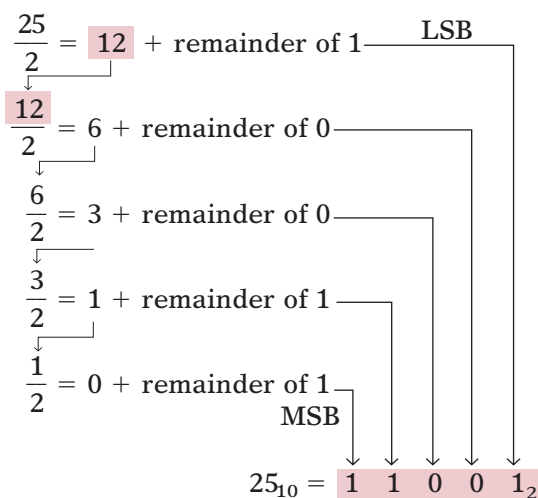
There are two ways to convert a decimal *whole* number to its equivalent binary-system representation. The first method is the reverse of the first process described in Section 2-1. The decimal number is simply expressed as a sum of powers of 2, and then 1s and 0s are written in the appropriate bit positions. To illustrate:

$$\begin{aligned} 45_{10} &= 32 + 8 + 4 + 1 = 2^5 + 0 + 2^3 + 2^2 + 0 + 2^0 \\ &= 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1_2 \end{aligned}$$

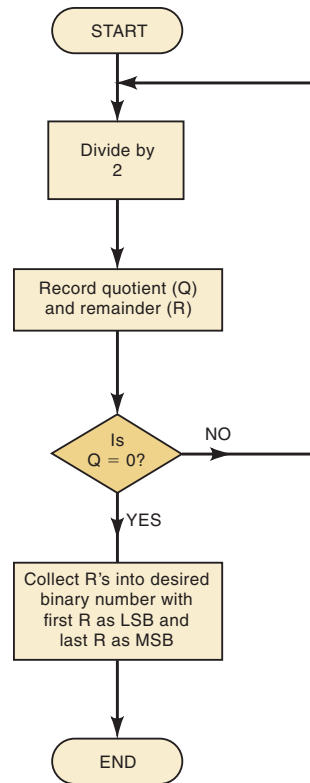
Note that a 0 is placed in the  $2^1$  and  $2^4$  positions, since all positions must be accounted for. Another example is the following:

$$\begin{aligned} 76_{10} &= 64 + 8 + 4 = 2^6 + 0 + 0 + 2^3 + 2^2 + 0 + 0 \\ &= 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0_2 \end{aligned}$$

Another method for converting decimal integers uses repeated division by 2. The conversion, illustrated below for  $25_{10}$ , requires repeatedly dividing the decimal number by 2 and writing down the remainder after each division until a quotient of 0 is obtained. Note that the binary result is obtained by writing the first remainder as the LSB and the last remainder as the MSB. This process, diagrammed in the flowchart of Figure 2-1, can also be used to convert from decimal to any other number system, as we shall see.



**FIGURE 2-1** Flowchart for repeated-division method of decimal-to-binary conversion of integers. The same process can be used to convert a decimal integer to any other number system.



### CALCULATOR HINT:

If you use a calculator to perform the divisions by 2, you can tell whether the remainder is 0 or 1 by whether or not the result has a fractional part. For instance,  $\frac{25}{2}$  would produce 12.5. Since there is a fractional part (the .5), the remainder is a 1. If there were no fractional part, such as  $\frac{12}{2} = 6$ , then the remainder would be 0. Example 2-1 illustrates this.

### EXAMPLE 2-1

Convert  $37_{10}$  to binary. Try to do it on your own before you look at the solution.

#### Solution

$$\begin{array}{r}
 \frac{37}{2} = 18.5 \longrightarrow \text{remainder of 1 (LSB)} \\
 \downarrow \\
 \frac{18}{2} = 9.0 \longrightarrow \quad \quad \quad 0 \\
 \\
 \frac{9}{2} = 4.5 \longrightarrow \quad \quad \quad 1 \\
 \\
 \frac{4}{2} = 2.0 \longrightarrow \quad \quad \quad 0 \\
 \\
 \frac{2}{2} = 1.0 \longrightarrow \quad \quad \quad 0 \\
 \\
 \frac{1}{2} = 0.5 \longrightarrow \quad \quad \quad 1 \text{ (MSB)}
 \end{array}$$

Thus,  $37_{10} = 100101_2$ .

## Counting Range

Recall that using  $N$  bits, we can count through  $2^N$  different decimal numbers ranging from 0 to  $2^N - 1$ . For example, for  $N = 4$ , we can count from  $0000_2$  to  $1111_2$ , which is  $0_{10}$  to  $15_{10}$ , for a total of 16 different numbers. Here, the largest decimal value is  $2^4 - 1 = 15$ , and there are  $2^4$  different numbers.

In general, then, we can state:

**Using  $N$  bits, we can represent decimal numbers ranging from 0 to  $2^N - 1$ , a total of  $2^N$  different numbers.**

### EXAMPLE 2-2

- (a) What is the total range of decimal values that can be represented in eight bits?
- (b) How many bits are needed to represent decimal values ranging from 0 to 12,500?

#### Solution

- (a) Here we have  $N = 8$ . Thus, we can represent decimal numbers from 0 to  $2^8 - 1 = 255$ . We can verify this by checking to see that  $11111111_2$  converts to  $255_{10}$ .
- (b) With 13 bits, we can count from decimal 0 to  $2^{13} - 1 = 8191$ . With 14 bits, we can count from 0 to  $2^{14} - 1 = 16,383$ . Clearly, 13 bits aren't enough, but 14 bits will get us up beyond 12,500. Thus, the required number of bits is 14.

### OUTCOME ASSESSMENT QUESTIONS

1. Convert  $83_{10}$  to binary using both methods.
2. Convert  $729_{10}$  to binary using both methods. Check your answer by converting back to decimal.
3. How many bits are required to count up to decimal 1 million?

## 2-3 HEXADECIMAL NUMBER SYSTEM

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Identify the weight of each hexadecimal digit.
- Convert between any of the following number systems: binary, decimal, hexadecimal.
- Count in hexadecimal.
- Identify range of numbers (in all systems) for a given number of digits.
- Identify number of digits needed for a given range of values.
- Memorize the value of each hexadecimal digit in binary and decimal.
- Cite advantages of the hexadecimal number system.

The **hexadecimal number system** uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F

as the 16 digit symbols. The digit positions are weighted as powers of 16 as shown below, rather than as powers of 10 as in the decimal system.

$16^4$	$16^3$	$16^2$	$16^1$	$16^0$	$16^{-1}$	$16^{-2}$	$16^{-3}$	$16^{-4}$
--------	--------	--------	--------	--------	-----------	-----------	-----------	-----------

Hexadecimal point

Table 2-1 shows the relationships among hexadecimal, decimal, and binary. Note that each hexadecimal digit represents a group of four binary digits. It is important to remember that hex (abbreviation for “hexadecimal”) digits A through F are equivalent to the decimal values 10 through 15.

TABLE 2-1



Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

### Hex-to-Decimal Conversion

A hex number can be converted to its decimal equivalent by using the fact that each hex digit position has a weight that is a power of 16. The LSD has a weight of  $16^0 = 1$ ; the next higher digit position has a weight of  $16^1 = 16$ ; the next has a weight of  $16^2 = 256$ ; and so on. The conversion process is demonstrated in the examples below.

#### CALCULATOR HINT:

You can use the  $y^x$  calculator function to evaluate the powers of 16.

$$\begin{aligned}
 356_{16} &= 3 \times 16^2 + 5 \times 16^1 + 6 \times 16^0 \\
 &= 768 + 80 + 6 \\
 &= 854_{10}
 \end{aligned}$$

$$\begin{aligned}
 2AF_{16} &= 2 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 \\
 &= 512 + 160 + 15 \\
 &= 687_{10}
 \end{aligned}$$

Note that in the second example, the value 10 was substituted for A and the value 15 for F in the conversion to decimal.

For practice, verify that  $1BC2_{16}$  is equal to  $7106_{10}$ .

## Decimal-to-Hex Conversion

Recall that we did decimal-to-binary conversion using repeated division by 2. Likewise, decimal-to-hex conversion can be done using repeated division by 16 (Figure 2-1). The following example contains two illustrations of this conversion.

### EXAMPLE 2-3

(a) Convert  $423_{10}$  to hex.

#### Solution

$$\begin{array}{l} \frac{423}{16} = 26 + \text{remainder of } 7 \\ \downarrow \\ \frac{26}{16} = 1 + \text{remainder of } 10 \\ \downarrow \\ \frac{1}{16} = 0 + \text{remainder of } 1 \end{array}$$

$423_{10} = 1A7_{16}$

(b) Convert  $214_{10}$  to hex.

#### Solution

$$\begin{array}{l} \frac{214}{16} = 13 + \text{remainder of } 6 \\ \downarrow \\ \frac{13}{16} = 0 + \text{remainder of } 13 \end{array}$$

$214_{10} = D6_{16}$

Again note that the remainders of the division processes form the digits of the hex number. Also note that any remainders that are greater than 9 are represented by the letters A through F.

## CALCULATOR HINT:

If a calculator is used to perform the divisions in the conversion process, the results will include a decimal fraction instead of a remainder. The remainder can be obtained by multiplying the fraction by 16. To illustrate, in Example 2-3(b), the calculator would have produced

$$\frac{214}{16} = 13.375$$

The remainder becomes  $(0.375) \times 16 = 6$ .

## Hex-to-Binary Conversion

The hexadecimal number system is used primarily as a “shorthand” method for representing binary numbers. It is a relatively simple matter to convert



a hex number to binary. *Each* hex digit is converted to its four-bit binary equivalent (Table 2-1). This is illustrated below for  $9F2_{16}$ .

$$\begin{aligned} 9F2_{16} &= && 9 && F && 2 \\ & && \downarrow && \downarrow && \downarrow \\ &= && 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ &= && 10011110010_2 \end{aligned}$$

For practice, verify that  $BA6_{16} = 101110100110_2$ .

### Binary-to-Hex Conversion

Conversion from binary to hex is just the reverse of the process above. The binary number is grouped into groups of *four* bits, and each group is converted to its equivalent hex digit. Zeros (shown shaded) are added, as needed, to complete a four-bit group.

$$\begin{aligned} 1110100110_2 &= \underbrace{0011}_3 \underbrace{1010}_A \underbrace{0110}_6 \\ &= 3A6_{16} \end{aligned}$$

To perform these conversions between hex and binary, it is necessary to know the four-bit binary numbers (0000 through 1111) and their equivalent hex digits. Once these are mastered, the conversions can be performed quickly without the need for any calculations. This is why hex is so useful in representing large binary numbers.

For practice, verify that  $10101111_2 = 15F_{16}$ .

### Counting in Hexadecimal

When counting in hex, each digit position can be incremented (increased by 1) from 0 to F. Once a digit position reaches the value F, it is reset to 0, and the next digit position is incremented. This is illustrated in the following hex counting sequences:

- (a) 38, 39, 3A, 3B, 3C, 3D, 3E, 3F, 40, 41, 42
- (b) 6F8, 6F9, 6FA, 6FB, 6FC, 6FD, 6FE, 6FF, 700

Note that when there is a 9 in a digit position, it becomes an A when it is incremented.

With  $N$  hex digit positions, we can count from decimal 0 to  $16^N - 1$ , for a total of  $16^N$  different values. For example, with three hex digits, we can count from  $000_{16}$  to  $FFF_{16}$ , which is  $0_{10}$  to  $4095_{10}$ , for a total of  $4096 = 16^3$  different values.

### Usefulness of Hex

Hex is often used in a digital system as sort of a “shorthand” way to represent strings of bits. In computer work, strings as long as 64 bits are not uncommon. These binary strings do not always represent a numerical value, but—as you will find out—can be some type of code that conveys nonnumerical information. When dealing with a large number of bits, it is more convenient and less error-prone to write the binary numbers in hex and, as we have seen, it is relatively easy to convert back and forth between binary and hex. To illustrate the advantage of hex representation of a

binary string, suppose you had in front of you a printout of the contents of 50 memory locations, each of which was a 16-bit number, and you were checking it against a list. Would you rather check 50 numbers like this one: 0110111001100111, or 50 numbers like this one: 6E67? And which one would you be more apt to read incorrectly? It is important to keep in mind, though, that digital circuits all work in binary. Hex is simply used as a convenience for the humans involved. You should memorize the four-bit binary pattern for each hexadecimal digit. Only then will you realize the usefulness of this tool in digital systems.

**EXAMPLE 2-4**

Convert decimal 378 to a 16-bit binary number by first converting to hexadecimal.

**Solution**

$$\begin{array}{r} \frac{378}{16} = 23 + \text{remainder of } 10_{10} = A_{16} \\ \downarrow \\ \frac{23}{16} = 1 + \text{remainder of } 7 \\ \downarrow \\ \frac{1}{16} = 0 + \text{remainder of } 1 \end{array}$$

Thus,  $378_{10} = 17A_{16}$ . This hex value can be converted easily to binary 000101111010. Finally, we can express  $378_{10}$  as a 16-bit number by adding four leading 0s:

$$378_{10} = 0000 \ 0001 \ 0111 \ 1010_2$$

**EXAMPLE 2-5**

Convert  $B2F_{16}$  to decimal.

**Solution**

$$\begin{aligned} B2F_{16} &= B \times 16^2 + 2 \times 16^1 + F \times 16^0 \\ &= 11 \times 256 + 2 \times 16 + 15 \\ &= 2863_{10} \end{aligned}$$

**Summary of Conversions**

Right now, your head is probably spinning as you try to keep straight all of these different conversions from one number system to another. You probably realize that many of these conversions can be done *automatically* on your calculator just by pressing a key, but it is important for you to master these conversions so that you understand the process. Besides, what happens if your calculator battery dies at a crucial time and you have no handy replacement? The following summary should help you, but nothing beats practice, practice, practice!

1. When converting from binary [or hex] to decimal, use the method of taking the weighted sum of each digit position or follow the double-dabble procedure.
2. When converting from decimal to binary [or hex], use the method of repeatedly dividing by 2 [or 16] and collecting remainders (Figure 2-1).

3. When converting from binary to hex, group the bits in groups of four, and convert each group into the correct hex digit.
4. When converting from hex to binary, convert each digit into its four-bit equivalent.

### OUTCOME ASSESSMENT QUESTIONS

1. Convert  $24CE_{16}$  to decimal.
2. Convert  $3117_{10}$  to hex, then from hex to binary.
3. Convert  $1001011110110101_2$  to hex.
4. Write the next four numbers in this hex counting sequence: E9A, E9B, E9C, E9D, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_.
5. Convert  $3527_{16}$  to binary.
6. What range of decimal values can be represented by a four-digit hex number?

## 2-4 BCD CODE

### OUTCOMES

Upon completion of this section, you will be able to:

- Convert decimal numbers to BCD code.
- Convert BCD code to decimal.
- Cite the pros and cons of using BCD.
- Cite advantages/disadvantages of BCD versus binary in digital systems.

When numbers, letters, or words are represented by a special group of symbols, we say that they are being encoded, and the group of symbols is called a *code*. Probably one of the most familiar codes is the Morse code, where a series of dots and dashes represents letters of the alphabet.

We have seen that any decimal number can be represented by an equivalent binary number. The group of 0s and 1s in the binary number can be thought of as a code representing the decimal number. When a decimal number is represented by its equivalent binary number, we call it **straight binary coding**.

Digital systems all use some form of binary numbers for their internal operation, but the external world is decimal in nature. This means that conversions between the decimal and binary systems are being performed often. We have seen that the conversions between decimal and binary can become long and complicated for large numbers. For this reason, a means of encoding decimal numbers that combines some features of both the decimal and the binary systems is used in certain situations.

### Binary-Coded-Decimal Code

If *each* digit of a decimal number is represented by its binary equivalent, the result is a code called **binary-coded decimal** (hereafter abbreviated BCD). Since a decimal digit can be as large as 9, four bits are required to code each digit (the binary code for 9 is 1001).

To illustrate the BCD code, take a decimal number such as 874. Each *digit* is changed to its binary equivalent as follows:

$$\begin{array}{ccc} 8 & 7 & 4 & \text{(decimal)} \\ \downarrow & \downarrow & \downarrow & \\ 1000 & 0111 & 0100 & \text{(BCD)} \end{array}$$

As another example, let us change 943 to its BCD-code representation:

$$\begin{array}{ccc} 9 & 4 & 3 & \text{(decimal)} \\ \downarrow & \downarrow & \downarrow & \\ 1001 & 0100 & 0011 & \text{(BCD)} \end{array}$$

Once again, each decimal digit is changed to its straight binary equivalent. Note that four bits are *always* used for each digit.

The BCD code, then, represents each digit of the decimal number by a four-bit binary number. Clearly only the four-bit binary numbers from 0000 through 1001 are used. The BCD code does not use the numbers 1010, 1011, 1100, 1101, 1110, and 1111. In other words, only 10 of the 16 possible four-bit binary code groups are used. If any of the “forbidden” four-bit numbers ever occurs in a machine using the BCD code, it is usually an indication that an error has occurred.

**EXAMPLE 2-6**

Convert 011010000111001 (BCD) to its decimal equivalent.

**Solution**

Divide the BCD number into four-bit groups and convert each to decimal.

$$\begin{array}{cccc} \underbrace{0110} & \underbrace{1000} & \underbrace{0011} & \underbrace{1001} \\ 6 & 8 & 3 & 9 \end{array}$$

**EXAMPLE 2-7**

Convert the BCD number 01111100001 to its decimal equivalent.

**Solution**

$$\begin{array}{ccc} \underbrace{0111} & \underbrace{1100} & \underbrace{0001} \\ 7 & \downarrow & 1 \end{array}$$

The forbidden code group indicates an error in the BCD number.

**Comparison of BCD and Binary**

It is important to realize that BCD is not another number system like binary, decimal, and hexadecimal. In fact, it is the decimal system with each digit encoded in its binary equivalent. It is also important to understand that a BCD number is *not* the same as a straight binary number. A straight binary number takes the *complete* decimal number and represents it in binary; the BCD code converts *each* decimal *digit* to binary individually. To illustrate, take the number 137 and compare its straight binary and BCD codes:

$$\begin{array}{l} 137_{10} = 10001001_2 \quad \text{(binary)} \\ 137_{10} = 0001\ 0011\ 0111 \quad \text{(BCD)} \end{array}$$

The BCD code requires 12 bits, while the straight binary code requires only eight bits to represent 137. BCD requires more bits than straight binary to represent decimal numbers of more than one digit because BCD does not use all possible four-bit groups, as pointed out earlier, and is therefore somewhat inefficient.

The main advantage of the BCD code is the relative ease of converting to and from decimal. Only the four-bit code groups for the decimal digits 0 through 9 need to be remembered. This ease of conversion is especially important from a hardware standpoint because in a digital system, it is the logic circuits that perform the conversions to and from decimal.

#### EXAMPLE 2-8

An automatic teller machine (ATM) at the bank allows you to enter the amount of cash you wish to withdraw in decimal by pressing decimal digit symbol keys. Does the computer convert this decimal number into straight binary or BCD? Explain.

#### Solution

The number that represents your balance (all the money you have in the bank) is stored as a straight binary number. When the withdrawn amount is entered, it must be subtracted from the balance. Since arithmetic must be performed on the numbers, both values (the balance and the withdrawn cash) must be in straight binary. It converts the decimal entry to straight binary.

#### EXAMPLE 2-9

Your cell phone allows you to enter/store a 10-decimal-digit phone number. Does the cell phone store the phone number in straight binary or BCD? Explain.

#### Solution

A phone number is a combination of many decimal digits. It is not necessary to mathematically combine the digits (i.e., you never add two phone numbers together). The phone just needs to store them in the sequence they were entered and retrieve them one at a time when you press *send*. Therefore, they will be stored as BCD digits in the memory of the computer in your cell phone.

#### OUTCOME ASSESSMENT QUESTIONS

1. Represent the decimal value 178 by its straight binary equivalent. Then encode the same decimal number using BCD.
2. How many bits are required to represent an eight-digit decimal number in BCD?
3. What is an advantage of encoding a decimal number in BCD rather than in straight binary? What is a disadvantage?

## 2-5 THE GRAY CODE

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Cite advantage of Gray code versus binary.
- Convert between Gray code and binary values.
- Generate the Gray code sequence.

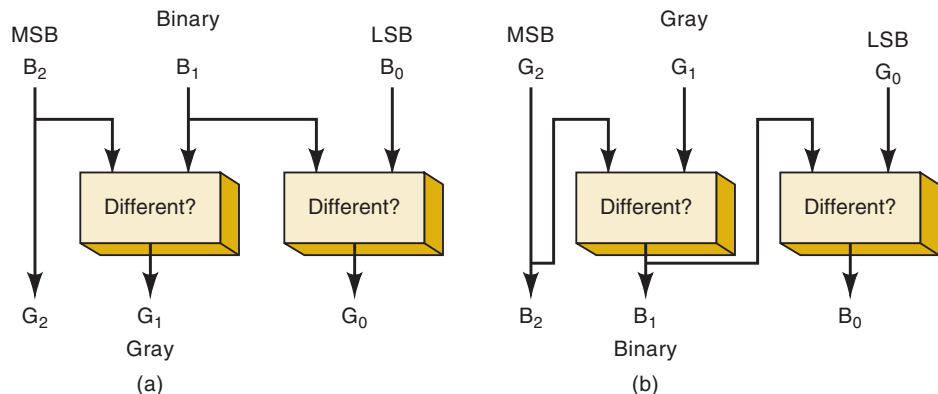
**TABLE 2-2** Three-bit binary and Gray code equivalents.

B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	G <sub>2</sub>	G <sub>1</sub>	G <sub>0</sub>
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

Digital systems operate at very fast speeds and respond to changes that occur in the digital inputs. Just as in life, when multiple input conditions are changing at the same time, the situation can be misinterpreted and cause an erroneous reaction. When you look at the bits in a binary count sequence, it is clear that there are often several bits that must change states at the same time. For example, consider when the three-bit binary number for 3 changes to 4: all three bits must change state.

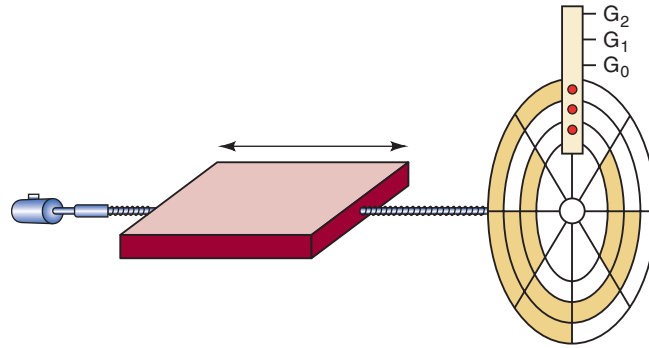
In order to reduce the likelihood of a digital circuit misinterpreting a changing input, the **Gray code** has been developed as a way to represent a sequence of numbers. The unique aspect of the Gray code is that only one bit ever changes between two successive numbers in the sequence. Table 2-2 shows the translation between three-bit binary and Gray code values. To convert binary to Gray, simply start on the most significant bit and use it as the Gray MSB as shown in Figure 2-2(a). Now compare the MSB binary with the next binary bit (B<sub>1</sub>). If they are the same, then G<sub>1</sub> = 0. If they are different, then G<sub>1</sub> = 1. G<sub>0</sub> can be found by comparing B<sub>1</sub> with B<sub>0</sub>.

**FIGURE 2-2** Converting (a) binary to Gray and (b) Gray to binary.



Conversion from Gray code back into binary is shown in Figure 2-2(b). Note that the MSB in Gray is always the same as the MSB in binary. The next binary bit is found by comparing the *binary* bit to the left with the *corresponding Gray code bit*. Similar bits produce a 0 and differing bits produce a 1. The most common application of the Gray code is in shaft position encoders as shown in Figure 2-3. These devices produce a binary value that represents the position of a rotating mechanical shaft. A practical shaft encoder would use many more bits than just three and divide the rotation into many more segments than eight, so that it could detect much smaller increments of rotation.

**FIGURE 2-3** An eight-position, three-bit shaft encoder.



**Quadrature Encoders**

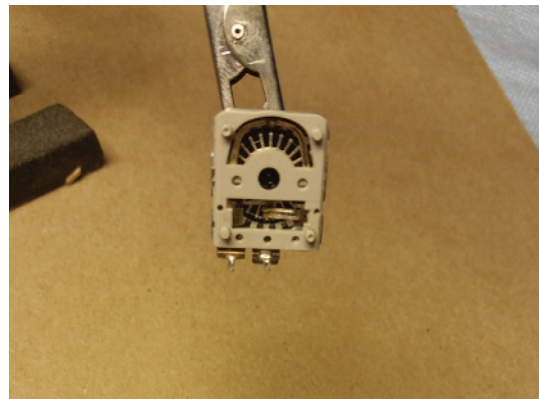
The most common application of the Gray code is the quadrature shaft encoder. As the shaft rotates, this device produces a two-bit Gray code sequence on its outputs. Clockwise rotation produces the sequence shown in Table 2-3(a) and counterclockwise rotation produces the sequence shown in Table 2-3(b). Converting these Gray code values to binary shows that they are counting up or counting down depending on the direction of rotation. The sensitivity or number of degrees of rotation represented by each state of the Gray sequence will vary between the many models of shaft encoders available. The important feature of a shaft encoder is that the *sequence* of states can be used to determine which direction the shaft is rotating.

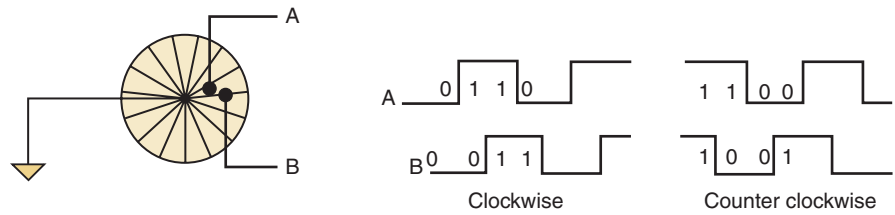
**TABLE 2-3** The two-bit Gray code from a quadrature shaft encoder.

Clockwise				Counter Clockwise			
A	B	Binary	Decimal	A	B	Binary	Decimal
0	0	00	0	0	0	00	0
0	1	01	1	1	0	11	3
1	1	10	2	1	1	10	2
1	0	11	3	0	1	01	1
(a)				(b)			

Figure 2-4 shows an inexpensive shaft encoder that can be used as a control knob on consumer electronics. This knob could be a volume control, or a tuning control on a radio receiver, for example. There are three terminals on this encoder. One terminal connects to the spoked wheel. The conducting spokes on the wheel rub against two spring metal contact arms as the shaft

**FIGURE 2-4** A mechanical contact quadrature encoder.



**FIGURE 2-5** Operation of a quadrature encoder.

rotates. The other two terminals are wired to the two spring metal contacts. The spring metal contacts are positioned such that one will always make contact with the spoke slightly before the other one as the shaft is rotated. With this encoder connected as shown in Figure 2-5, rotating the shaft clockwise and counter clockwise produces the waveforms shown. Notice that the states in these timing diagrams follow the two-bit Gray code sequence.

Quadrature encoders are referred to as incremental shaft encoders, while encoders that put out enough Gray code bits to uniquely identify any shaft position as shown in Figure 2-3 are referred to as absolute shaft encoders. Chapter 5 will demonstrate how to combine an incremental (quadrature) shaft encoder with a digital counter circuit to keep track of absolute shaft position.

### OUTCOME ASSESSMENT QUESTIONS

1. Convert the number 0101 (binary) to its Gray code equivalent.
2. Convert 0101 (Gray code) to its binary number equivalent.

## 2-6 PUTTING IT ALL TOGETHER

Table 2-4 gives the representation of the decimal numbers 1 through 15 in the binary and hex number systems and also in the BCD and Gray codes. Examine it carefully and make sure you understand how it was obtained. Especially note how the BCD representation always uses four bits for each decimal digit.

**TABLE 2-4** Number system/code equivalents.

Decimal	Binary	Hexadecimal	BCD	Gray
0	0	0	0000	0000
1	1	1	0001	0001
2	10	2	0010	0011
3	11	3	0011	0010
4	100	4	0100	0110
5	101	5	0101	0111
6	110	6	0110	0101
7	111	7	0111	0100
8	1000	8	1000	1100
9	1001	9	1001	1101
10	1010	A	0001 0000	1111
11	1011	B	0001 0001	1110
12	1100	C	0001 0010	1010
13	1101	D	0001 0011	1011
14	1110	E	0001 0100	1001
15	1111	F	0001 0101	1000



## 2-7 THE BYTE, NIBBLE, AND WORD

### OUTCOMES

Upon completion of this section, you will be able to:

- Define common terms: *byte*, *nibble*, and *word*.
- Use these words in context.
- Interpret these words in the context used.

### Bytes

Most microcomputers handle and store binary data and information in groups of eight bits, so a special name is given to a string of eight bits: it is called a **byte**. A byte always consists of eight bits, and it can represent any of numerous types of data or information. The following examples will illustrate.

#### EXAMPLE 2-10

How many bytes are in a 32-bit string (a string of 32 bits)?

#### Solution

$\frac{32}{8} = 4$ , so there are **four** bytes in a 32-bit string.

#### EXAMPLE 2-11

What is the largest decimal value that can be represented in binary using two bytes?

#### Solution

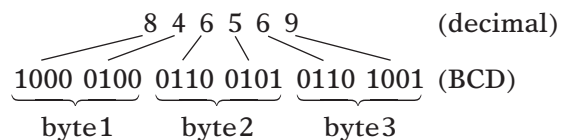
Two bytes is 16 bits, so the largest binary value will be equivalent to decimal  $2^{16} - 1 = 65,535$ .

#### EXAMPLE 2-12

How many bytes are needed to represent the decimal value 846,569 in BCD?

#### Solution

Each decimal digit converts to a four-bit BCD code. Thus, a six-digit decimal number requires 24 bits. These 24 bits are equal to **three** bytes. This is diagrammed below.



### Nibbles

Binary numbers are often broken down into groups of four bits, as we have seen with BCD codes and hexadecimal number conversions. In the early days of digital systems, a term caught on to describe a group of four bits. Because it is half as big as a byte, it was named a **nibble**. The following examples illustrate the use of this term.

**EXAMPLE 2-13**

How many nibbles are in a byte?

**Solution**

2

**EXAMPLE 2-14**

What is the hex value of the least significant nibble of the binary number 10010101?

**Solution**

1001 0101

The least significant nibble is 0101 = 5.

**Words**

*Bits*, *nibbles*, and *bytes* are terms that represent a fixed number of binary digits. As systems have grown over the years, their capacity (appetite?) for handling binary data has also grown. A **word** is a group of bits that represents a certain unit of information. The size of the word depends on the size of the data pathway in the system that uses the information. The **word size** can be defined as the number of bits in the binary word that a digital system operates on. For example, the computer in your microwave oven can probably handle only one byte at a time. It has a word size of eight bits. On the other hand, the personal computer on your desk can handle eight bytes at a time, so it has a word size of 64 bits.

**OUTCOME ASSESSMENT QUESTIONS**

1. How many bytes are needed to represent  $235_{10}$  in binary?
2. What is the largest decimal value that can be represented in BCD using two bytes?
3. How many hex digits can a nibble represent?
4. How many nibbles are in one BCD digit?

**2-8 ALPHANUMERIC CODES****OUTCOMES**

*Upon completion of this section, you will be able to:*

- Use a table to translate between ASCII codes and characters.
- Explain the purpose of alphanumeric codes such as ASCII.

In addition to numerical data, a computer must be able to handle nonnumerical information. In other words, a computer should recognize codes that represent letters of the alphabet, punctuation marks, and other special characters as well as numbers. These codes are called **alphanumeric codes**. A complete alphanumeric code would include the 26 lowercase letters, 26 uppercase letters, 10 numeric digits, 7 punctuation marks, and anywhere from 20 to 40 other characters, as +, /, #, %, \*, and so on. We can say that an alphanumeric code represents all of the various characters and functions that are found on a computer keyboard.

## ASCII Code

The most widely used alphanumeric code is the **American Standard Code for Information Interchange (ASCII)**. The ASCII (pronounced “askee”) code is a seven-bit code, and so it has  $2^7 = 128$  possible code groups. This is more than enough to represent all of the standard keyboard characters as well as the control functions such as the (RETURN) and (LINEFEED) functions. Table 2-5 shows a listing of the standard seven-bit ASCII code. The table gives the hexadecimal and decimal equivalents. The seven-bit binary code for each character can be obtained by converting the hex value to binary.

**TABLE 2-5** Standard ASCII codes.

Character	Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decimal
NUL (null)	0	0	Space	20	32	@	40	64	.	60	96
Start Heading	1	1	!	21	33	A	41	65	a	61	97
Start Text	2	2	“	22	34	B	42	66	b	62	98
End Text	3	3	#	23	35	C	43	67	c	63	99
End Transmit.	4	4	\$	24	36	D	44	68	d	64	100
Enquiry	5	5	%	25	37	E	45	69	e	65	101
Acknowledge	6	6	&	26	38	F	46	70	f	66	102
Bell	7	7	`	27	39	G	47	71	g	67	103
Backspace	8	8	(	28	40	H	48	72	h	68	104
Horiz. Tab	9	9	)	29	41	I	49	73	i	69	105
Line Feed	A	10	*	2A	42	J	4A	74	j	6A	106
Vert. Tab	B	11	+	2B	43	K	4B	75	k	6B	107
Form Feed	C	12	,	2C	44	L	4C	76	l	6C	108
Carriage Return	D	13	-	2D	45	M	4D	77	m	6D	109
Shift Out	E	14	.	2E	46	N	4E	78	n	6E	110
Shift In	F	15	/	2F	47	O	4F	79	o	6F	111
Data Link Esc	10	16	0	30	48	P	50	80	p	70	112
Direct Control 1	11	17	1	31	49	Q	51	81	q	71	113
Direct Control 2	12	18	2	32	50	R	52	82	r	72	114
Direct Control 3	13	19	3	33	51	S	53	83	s	73	115
Direct Control 4	14	20	4	34	52	T	54	84	t	74	116
Negative ACK	15	21	5	35	53	U	55	85	u	75	117
Synch Idle	16	22	6	36	54	V	56	86	v	76	118
End Trans Block	17	23	7	37	55	W	57	87	w	77	119
Cancel	18	24	8	38	56	X	58	88	x	78	120
End of Medium	19	25	9	39	57	Y	59	89	y	79	121
Substitutue	1A	26	:	3A	58	Z	5A	90	z	7A	122
Escape	1B	27	;	3B	59	[	5B	91	{	7B	123
Form Separator	1C	28	<	3C	60	\	5C	92		7C	124
Group Separator	1D	29	=	3D	61	]	5D	93	}	7D	125
Record Separator	1E	30	>	3E	62	^	5E	94	~	7E	126
Unit Separator	1F	31	?	3F	63	_	5F	95	Delete	7F	127

**EXAMPLE 2-15**

Use Table 2-5 to find the seven-bit ASCII code for the backslash character (\).

**Solution**

The hex value given in Table 2-5 is 5C. Translating each hex digit into four-bit binary produces 0101 1100. The lower seven bits represent the ASCII code for \, or 1011100.

The ASCII code is used for the transfer of alphanumeric information between a computer and the external devices such as a printer or another computer. A computer also uses ASCII internally to store the information that an operator types in at the computer's keyboard. The following example illustrates this.

**EXAMPLE 2-16**

An operator is typing in a C language program at the keyboard of a certain microcomputer. The computer converts each keystroke into its ASCII code and stores the code as a byte in memory. Determine the binary strings that will be entered into memory when the operator types in the following C statement:

```
if (x>3)
```

**Solution**

Locate each character (including the space) in Table 2-5 and record its ASCII code.

i	69	0110	1001
f	66	0110	0110
space	20	0010	0000
(	28	0010	1000
x	78	0111	1000
>	3E	0011	1110
3	33	0011	0011
)	29	0010	1001

Note that a 0 was added to the leftmost bit of each ASCII code because the codes must be stored as bytes (eight bits). This adding of an extra bit is called *padding with 0s*.

**OUTCOME ASSESSMENT QUESTIONS**

1. Encode the following message in ASCII code using the hex representation: "COST = \$72."
2. The following padded ASCII-coded message is stored in successive memory locations in a computer:

```
01010011 01010100 01001111 01010000
```

What is the message?

## 2-9 PARITY METHOD FOR ERROR DETECTION

### OUTCOMES

Upon completion of this section, you will be able to:

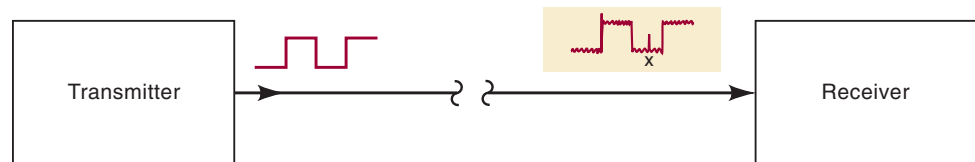
- Use even or odd parity schemes.
- Add the proper parity bit for either scheme.
- Explain the parity method for error detection.
- Determine if an error has occurred using either scheme.

The movement of binary data and codes from one location to another is the most frequent operation performed in digital systems. Here are just a few examples:

- The transmission of digitized voice over a microwave link
- The storage of data in and retrieval of data from external memory devices such as magnetic and optical disk
- The transmission of digital data from a computer to a remote computer over telephone lines (i.e., using a modem). This is one of the major ways of sending and receiving information on the Internet.

Whenever information is transmitted from one device (the transmitter) to another device (the receiver), there is a possibility that errors can occur such that the receiver does not receive the identical information that was sent by the transmitter. The major cause of any transmission errors is *electrical noise*, which consists of spurious fluctuations in voltage or current that are present in all electronic systems to varying degrees. Figure 2-6 is a simple illustration of a type of transmission error.

**FIGURE 2-6** Example of noise causing an error in the transmission of digital data.



The transmitter sends a relatively noise-free serial digital signal over a signal line to a receiver. However, by the time the signal reaches the receiver, it contains a certain degree of noise superimposed on the original signal. Occasionally, the noise is large enough in amplitude that it will alter the logic level of the signal, as it does at point *x*. When this occurs, the receiver may incorrectly interpret that bit as a logic 1, which is not what the transmitter has sent.

Most modern digital equipment is designed to be relatively error-free, and the probability of errors such as the one shown in Figure 2-6 is very low. However, we must realize that digital systems often transmit thousands, even millions, of bits per second, so that even a very low rate of occurrence of errors can produce an occasional error that might prove to be bothersome, if not disastrous. For this reason, many digital systems employ some method for detection (and sometimes correction) of errors. One of the simplest and most widely used schemes for error detection is the **parity method**.

## Parity Bit

A **parity bit** is an extra bit that is attached to a code group that is being transferred from one location to another. The parity bit is made either 0 or 1, depending on the number of 1s that are contained in the code group. Two different methods are used.

In the *even-parity* method, the value of the parity bit is chosen so that the total number of 1s in the code group (including the parity bit) is an *even* number. For example, suppose that the group is 100011. This is the ASCII character “C.” The code group has *three* 1s. Therefore, we will add a parity bit of 1 to make the total number of 1s an even number. The *new* code group, *including the parity bit*, thus becomes

1 1 0 0 0 1 1  
 ↑  
 ————— added parity bit\*

If the code group contains an even number of 1s to begin with, the parity bit is given a value of 0. For example, if the code group were 100001 (the ASCII code for “A”), the assigned parity bit would be 0, so that the new code, *including the parity bit*, would be 0100001.

The *odd-parity* method is used in exactly the same way except that the parity bit is chosen so the total number of 1s (including the parity bit) is an *odd* number. For example, for the code group 100001, the assigned parity bit would be a 1. For the code group 100011, the parity bit would be a 0.

Regardless of whether even parity or odd parity is used, the parity bit becomes an actual part of the code word. For example, adding a parity bit to the seven-bit ASCII code produces an eight-bit code. Thus, the parity bit is treated just like any other bit in the code.

The parity bit is issued to detect any *single-bit* errors that occur during the transmission of a code from one location to another. For example, suppose that the character “A” is being transmitted and *odd* parity is being used. The transmitted code would be

1 1 0 0 0 0 1

When the receiver circuit receives this code, it will check that the code contains an odd number of 1s (including the parity bit). If so, the receiver will assume that the code has been correctly received. Now, suppose that because of some noise or malfunction the receiver actually receives the following code:

1 1 0 0 0 0 0

The receiver will find that this code has an *even* number of 1s. This tells the receiver that there must be an error in the code because presumably the transmitter and receiver have agreed to use odd parity. There is no way, however, that the receiver can tell which bit is in error because it does not know what the code is supposed to be.

It should be apparent that this parity method would not work if *two* bits were in error, because two errors would not change the “oddness” or “evenness” of the number of 1s in the code. In practice, the parity method is used only in situations where the probability of a single error is very low and the probability of double errors is essentially zero.

When the parity method is being used, the transmitter and the receiver must have agreement, in advance, as to whether odd or even parity is being

---

\*The parity bit can be placed at either end of the code group, but it is usually placed to the left of the MSB.

used. There is no advantage of one over the other, although even parity seems to be used more often. The transmitter must attach an appropriate parity bit to each unit of information that it transmits. For example, if the transmitter is sending ASCII-coded data, it will attach a parity bit to each seven-bit ASCII code group. When the receiver examines the data that it has received from the transmitter, it checks each code group to see that the total number of 1s (including the parity bit) is consistent with the agreed-upon type of parity. This is often called *checking the parity* of the data. In the event that it detects an error, the receiver may send a message back to the transmitter asking it to retransmit the last set of data. The exact procedure that is followed when an error is detected depends on the particular system.

**EXAMPLE 2-17**

Computers often communicate with other remote computers over telephone lines. For example, this is how dial-up communication over the internet takes place. When one computer is transmitting a message to another, the information is usually encoded in ASCII. What actual bit strings would a computer transmit to send the message HELLO, using ASCII with even parity?

**Solution**

First, look up the ASCII codes for each character in the message. Then for each code, count the number of 1s. If it is an even number, attach a 0 as the MSB. If it is an odd number, attach a 1. Thus, the resulting eight-bit codes (bytes) will all have an even number of 1s (including parity).

	attached even-parity bits
	↓
H =	0 1 0 0 1 0 0 0
E =	1 1 0 0 0 1 0 1
L =	1 1 0 0 1 1 0 0
L =	1 1 0 0 1 1 0 0
O =	1 1 0 0 1 1 1 1

**Error Correction**

Error detection is beneficial because the system that receives a datum containing an error knows it has received “damaged goods.” Wouldn’t it be great if somehow the receiver could also know which bit was wrong? If a binary bit is wrong, then the correct value is simply its complement. Several methods have been developed to accomplish this. In each case, it requires that several bits of “error detection/correction codes” be applied to each transmitted packet of information. As the packet is received, a digital circuit can detect if errors have occurred (even multiple errors) and correct them. This technology is used for massive transfer of high speed data in such applications as magnetic disk drives, flash drives, CD, DVD, Blu-ray Disc, digital television, and broadband Internet networks.

**OUTCOME ASSESSMENT QUESTIONS**

1. Attach an odd-parity bit to the ASCII code for the \$ symbol, and express the result in hexadecimal.
2. Attach an even-parity bit to the BCD code for decimal 69.
3. Why can’t the parity method detect a double error in transmitted data?

## 2-10 APPLICATIONS

Here are several applications that will serve as a review of some of the concepts covered in this chapter. These applications should give a sense of how the various number systems and codes are used in the digital world. More applications are presented in the end-of-chapter problems.

### APPLICATION 2-1

A typical CD-ROM can store 650 megabytes of digital data. Since mega =  $2^{20}$ , how many bits of data can a CD-ROM hold?

#### Solution

Remember that a byte is eight bits. Therefore, 650 megabytes is  $650 \times 2^{20} \times 8 = 5,452,595,200$  bits.

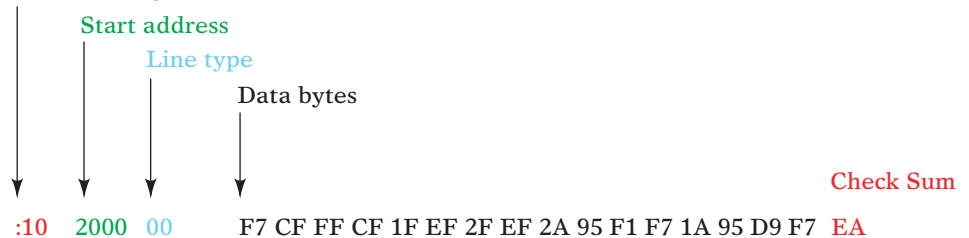
### APPLICATION 2-2

In order to program many microcontrollers, the binary instructions are stored in a file on a personal computer in a special way known as Intel Hex Format. The hexadecimal information is encoded into ASCII characters so it can be displayed easily on the PC screen, printed, and easily transmitted one character at a time over a standard PC's serial COM port. One line of an Intel Hex Format file is shown below:

```
:10200000F7CFFFCF1FEF2FEF2A95F1F71A95D9F7EA
```

#### Intel hex format:

Number of bytes of data in this line



The first character sent is the ASCII code for a colon, followed by a 1. Each has an even-parity bit appended as the most significant bit. A test instrument captures the binary bit pattern as it goes across the cable to the microcontroller.

- What should the binary bit pattern (including parity) look like? (MSB – LSB)
- The value 10, following the colon, represents the total hexadecimal number of bytes that are to be loaded into the micro's memory. What is the decimal number of bytes being loaded?
- The number 2000 is a four-digit hex value representing the address where the first byte is to be stored. What is the biggest address possible? How many bits would it take to represent this address?
- The value of the first data byte is F7. What is the value (in binary) of the least significant nibble of this byte?



**Solution**

(a) ASCII codes are 3A (for:) and 31 (for 1) 00111010 10110001  
 even-parity bit ↑ ↑

(b)  $10 \text{ hex} = 1 \times 16 + 0 \times 1 = 16$  decimal bytes

(c) FFFF is the biggest possible value. Each hex digit is 4 bits, so we need 16 bits.

FFFF            1111 1111 1111 1111            16 bits

(d) The least significant nibble (4 bits) is represented by hex 7. In binary, this would be 0111.

**APPLICATION 2-3**

A small process-control computer uses hexadecimal codes to represent its 16-bit memory addresses.

- (a) How many hex digits are required?
- (b) What is the range of addresses in hex?
- (c) How many memory locations are there?

**Solution**

(a) Since 4 bits convert to a single hex digit,  $\frac{16}{4} = 4$  hex digits are needed.

(b) The binary range is  $0000000000000000_2$  to  $1111111111111111_2$ . In hex, this becomes  $0000_{16}$  to  $FFFF_{16}$ .

(c) With 4 hex digits, the total number of addresses is  $16^4 = 65,536$ .

**APPLICATION 2-4**

Numbers are entered into a microcontroller-based system in BCD, but stored in straight binary. As a programmer, you must decide whether you need a one-byte or two-byte storage location.

- (a) How many bytes do you need if the system takes a two-digit decimal entry?
- (b) What if you needed to be able to enter three digits?

**Solution**

(a) With two digits, you can enter values up to 99 ( $1001\ 1001_{\text{BCD}}$ ). In binary this value is 01100011, which will fit into an eight-bit memory location. Thus you can use a single byte.

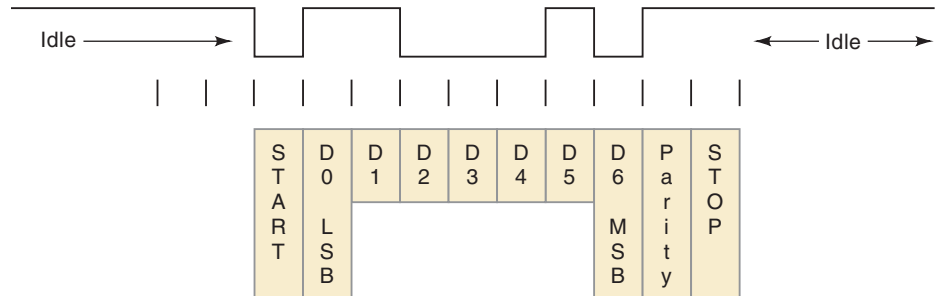
(b) Three digits can represent up to 999 ( $1001\ 1001\ 1001$ ). In binary, this value is 1111100111 (10 bits). Thus you cannot use a single byte; you need two bytes.

**APPLICATION 2-5**

When ASCII characters must be transmitted between two independent systems (such as between a computer and a modem), there must be a way of telling the receiver when a new character is coming in. There is often a need to detect errors in the transmission as well. The method of transfer is called asynchronous data communication. The normal resting state of the transmission line is logic 1. When the transmitter sends an ASCII character, it must be “framed” so the receiver knows where the data begins and ends. The first bit must always be a start bit (logic 0). Next the ASCII code is sent

LSB first and MSB last. After the MSB, a parity bit is appended to check for transmission errors. Finally, the transmission is ended by sending a stop bit (logic 1). A typical asynchronous transmission of a seven-bit ASCII code for the pound sign # (23 Hex) with even parity is shown in Figure 2-7.

**FIGURE 2-7** Asynchronous serial data with even parity.



### APPLICATION 2-6

Your PC encounters an error when running an application. The dialog box reports information about the addresses that it could not read or write. What number system is used to report the address area?

#### Solution

These numbers will normally be reported in hexadecimal. Rather than using the subscript 16 as we have done in this text, other methods may be used to indicate hexadecimal (e.g., attaching a 0x prefix to the number).

## SUMMARY

1. The hexadecimal number system is used in digital systems and computers as an efficient way of representing binary quantities.
2. In conversions between hex and binary, each hex digit corresponds to four bits.
3. The repeated-division method is used to convert decimal numbers to binary or hexadecimal.
4. Using an  $N$ -bit binary number, we can represent decimal values from 0 to  $2^N - 1$ .
5. The BCD code for a decimal number is formed by converting each digit of the decimal number to its four-bit binary equivalent.
6. The Gray code defines a sequence of bit patterns in which only one bit changes between successive patterns in the sequence.
7. A byte is a string of eight bits. A nibble is four bits. The word size depends on the system.
8. An alphanumeric code is one that uses groups of bits to represent all of the various characters and functions that are part of a typical computer's keyboard. The ASCII code is the most widely used alphanumeric code.
9. The parity method for error detection attaches a special parity bit to each transmitted group of bits.

## IMPORTANT TERMS

---

hexadecimal number system	byte nibble	American Standard Code for Information Interchange (ASCII)
straight binary coding	word	parity method
binary-coded-decimal (BCD) code	word size	parity bit
Gray code	alphanumeric code	

## PROBLEMS

---

### SECTIONS 2-1 AND 2-2

2-1. Convert these binary numbers to decimal.

- |                  |                |             |
|------------------|----------------|-------------|
| (a)*10110        | (e)*11111111   | (i)* 100110 |
| (b) 10010101     | (f) 01101111   | (j) 1101    |
| (c)*100100001001 | (g)*1111010111 | (k)*111011  |
| (d) 01101011     | (h) 11011111   | (l) 1010101 |

2-2. Convert the following decimal values to binary.

- |          |          |         |
|----------|----------|---------|
| (a)*37   | (e)*77   | (i)*511 |
| (b) 13   | (f) 390  | (j) 25  |
| (c)*189  | (g)*205  | (k) 52  |
| (d) 1000 | (h) 2133 | (l) 47  |

2-3. What is the largest decimal value that can be represented by (a)\* an eight-bit binary number? (b) A 16-bit number?

### SECTION 2-4

2-4. Convert each hex number to its decimal equivalent.

- |          |          |         |
|----------|----------|---------|
| (a)*743  | (e)*165  | (i) E71 |
| (b) 36   | (f) ABCD | (j) 89  |
| (c)*37FD | (g)*7FF  | (k) 58  |
| (d) 2000 | (h) 1204 | (l) 72  |

2-5. Convert each of the following decimal numbers to hex.

- |          |            |         |
|----------|------------|---------|
| (a)*59   | (e)*771    | (i) 29  |
| (b) 372  | (f) 2313   | (j) 33  |
| (c)*919  | (g)*65,536 | (k) 100 |
| (d) 1024 | (h) 255    | (l) 200 |

2-6. Convert each of the hex values from Problem 2-4 to binary.

2-7. Convert the binary numbers in Problem 2-1 to hex.

2-8. List the hex numbers in sequence from  $175_{16}$  to  $180_{16}$ .

2-9.\*When a large decimal number is to be converted to binary, it is sometimes easier to convert it first to hex, and then from hex to binary. Try this procedure for  $2133_{10}$  and compare it with the procedure used in Problem 2-2(h).

2-10. How many hex digits are required to represent decimal numbers up to 20,000? up to 40,000?

---

\*Answers to problems marked with an asterisk can be found in the back of the text.

2-11. Convert these hex values to decimal.

- |          |          |        |
|----------|----------|--------|
| (a)*92   | (e)*000F | (i) 19 |
| (b) 1A6  | (f) 55   | (j) 42 |
| (c)*315A | (g)*2C0  | (k) CA |
| (d) A02D | (h) 7F   | (l) F1 |

2-12. Convert these decimal values to hex.

- |          |            |         |
|----------|------------|---------|
| (a)*75   | (e)*7245   | (i) 95  |
| (b) 314  | (f) 498    | (j) 89  |
| (c)*2048 | (g)*25,619 | (k) 128 |
| (d) 24   | (h) 4095   | (l) 256 |

2-13. Take each four-bit binary number in the order they are written and write the equivalent hex digit without performing a calculation by hand or by calculator.

- |          |          |          |          |
|----------|----------|----------|----------|
| (a) 1001 | (e) 1111 | (i) 1011 | (m) 0001 |
| (b) 1101 | (f) 0010 | (j) 1100 | (n) 0101 |
| (c) 1000 | (g) 1010 | (k) 0011 | (o) 0111 |
| (d) 0000 | (h) 1001 | (l) 0100 | (p) 0110 |

2-14. Take each hex digit and write its four-bit binary value without performing any calculations by hand or by calculator.

- |       |       |       |       |
|-------|-------|-------|-------|
| (a) 6 | (e) 4 | (i) 9 | (m) 0 |
| (b) 7 | (f) 3 | (j) A | (n) 8 |
| (c) 5 | (g) C | (k) 2 | (o) D |
| (d) 1 | (h) B | (l) F | (p) 9 |

2-15. What is the largest value that can be represented by three hex digits?

2-16.\*Convert the hex values in Problem 2-11 to binary.

2-17.\*List the hex numbers in sequence from 280 to 2A0.

2-18. How many hex digits are required to represent decimal numbers up to 1 million? 4 million?

#### SECTION 2-4

2-19. Encode these decimal numbers in BCD.

- |          |            |         |
|----------|------------|---------|
| (a)*47   | (e)*13     | (i)* 72 |
| (b) 962  | (f) 529    | (j) 38  |
| (c)*187  | (g)*89,627 | (k)*61  |
| (d) 6727 | (h) 1024   | (l) 90  |

2-20. How many bits are required to represent the decimal numbers in the range from 0 to 999 using (a) straight binary code? (b) BCD code?

2-21. The following numbers are in BCD. Convert them to decimal.

- |                      |                  |
|----------------------|------------------|
| (a)*1001011101010010 | (f) 010101010101 |
| (b) 000110000100     | (g) 10111        |
| (c)*011010010101     | (h) 010110       |
| (d) 0111011101110101 | (i) 1110101      |
| (e)*010010010010     |                  |

**SECTION 2-7**

- 2-22. (a) How many bits are contained in eight bytes?  
 (b) What is the largest hex number that can be represented in four bytes?  
 (c) What is the largest BCD-encoded decimal value that can be represented in three bytes?
- 2-23. (a) Refer to Table 2-5. What is the most significant nibble of the ASCII code for the letter X?  
 (b) How many nibbles can be stored in a 16-bit word?  
 (c) How many bytes does it take to make up a 24-bit word?

**SECTIONS 2-8 AND 2-9**

- 2-24. Represent the statement “ $X = 3 \times Y$ ” in ASCII code. Attach an odd-parity bit.
- 2-25.\* Attach an *even*-parity bit to each of the ASCII codes for Problem 2-24, and give the results in hex.
- 2-26. The following bytes (shown in hex) represent a person’s name as it would be stored in a computer’s memory. Each byte is a padded ASCII code. Determine the name of each person.  
 (a) \*42 45 4E 20 53 4D 49 54 48  
 (b) 4A 6F 65 20 47 72 65 65 6E
- 2-27. Convert the following decimal numbers to BCD code and then attach an *odd*-parity bit.  
 (a) \*74                      (c) \*8884                      (e) \*165                      (g) 11  
 (b) 38                      (d) 275                      (f) 9201                      (h) 51
- 2-28.\* In a certain digital system, the decimal numbers from 000 through 999 are represented in BCD code. An *odd*-parity bit is also included at the end of each code group. Examine each of the code groups below, and assume that each one has just been transferred from one location to another. Some of the groups contain errors. Assume that *no more than* two errors have occurred for each group. Determine which of the code groups have a single error and which of them *definitely* have a double error. (*Hint*: Remember that this is a BCD code.)  
 (a) 1001010110000  
     ↑MSB  LSB↑↑\_\_\_\_\_ Parity bit  
 (b) 0100011101100  
 (c) 0111110000011  
 (d) 1000011000101
- 2-29. Suppose that the receiver received the following data from the transmitter of Example 2-17:

```

01001000
11000101
11001100
11001000
11001100

```

What errors can the receiver determine in these received data?

**DRILL QUESTIONS**

2-30. \*Perform each of the following conversions. For some of them, you may want to try several methods to see which one works best for you. For example, a binary-to-decimal conversion may be done directly, or it may be done as a binary-to-hex conversion followed by a hex-to-decimal conversion.

- (a)  $1417_{10} = \text{_____}_2$
- (b)  $255_{10} = \text{_____}_2$
- (c)  $11010001_2 = \text{_____}_{10}$
- (d)  $1110101000100111_2 = \text{_____}_{10}$
- (e)  $2497_{10} = \text{_____}_{16}$
- (f)  $511_{10} = \text{_____}(\text{BCD})$
- (g)  $235_{16} = \text{_____}_{10}$
- (h)  $4316_{10} = \text{_____}_{16}$
- (i)  $7A9_{16} = \text{_____}_{10}$
- (j)  $3E1C_{16} = \text{_____}_{10}$
- (k)  $1600_{10} = \text{_____}_{16}$
- (l)  $38,187_{10} = \text{_____}_{16}$
- (m)  $865_{10} = \text{_____}(\text{BCD})$
- (n)  $100101000111(\text{BCD}) = \text{_____}_{10}$
- (o)  $465_{16} = \text{_____}_2$
- (p)  $B34_{16} = \text{_____}_2$
- (q)  $01110100(\text{BCD}) = \text{_____}_2$
- (r)  $111010_2 = \text{_____}(\text{BCD})$

2-31. \*Represent the decimal value 37 in each of the following ways.

- (a) Straight binary
- (b) BCD
- (c) Hex
- (d) ASCII (i.e., treat each digit as a character)

2-32. \*Fill in the blanks with the correct word or words.

- (a) Conversion from decimal to \_\_\_\_\_ requires repeated division by 16.
  - (b) Conversion from decimal to binary requires repeated division by \_\_\_\_\_.
  - (c) In the BCD code, each \_\_\_\_\_ is converted to its four-bit binary equivalent.
  - (d) The \_\_\_\_\_ code has the characteristic that only one bit changes in going from one step to the next.
  - (e) A transmitter attaches a \_\_\_\_\_ to a code group to allow the receiver to detect \_\_\_\_\_.
  - (f) The \_\_\_\_\_ code is the most common alphanumeric code used in computer systems.
  - (g) \_\_\_\_\_ is often used as a convenient way to represent large binary numbers.
  - (h) A string of eight bits is called a \_\_\_\_\_.
-

- 2-33. Write the binary number that results when each of the following numbers is incremented by one.  
 (a) \*0111            (b) 010011            (c) 1011            (d) 1111
- 2-34. Decrement each binary number by one.  
 (a) \*1100            (b) 101000            (c) 1110            (d) 1001 0000
- 2-35. Write the number that results when each of the following is incremented.  
 (a) \*7779<sub>16</sub>            (c) \*OFFF<sub>16</sub>            (e) \*9FF<sub>16</sub>            (g) F<sub>16</sub>  
 (b) 9999<sub>16</sub>            (d) 2000<sub>16</sub>            (f) 100A<sub>16</sub>            (h) FE<sub>16</sub>
- 2-36. \*Repeat Problem 2-35 for the decrement operation.

### CHALLENGING EXERCISES

- 2-37. \*In a microcomputer, the *addresses* of memory locations are binary numbers that identify each memory circuit where a byte is stored. The number of bits that make up an address depends on how many memory locations there are. Since the number of bits can be very large, the addresses are often specified in hex instead of binary.
- (a) If a microcomputer uses a 20-bit address, how many different memory locations are there?
- (b) How many hex digits are needed to represent the address of a memory location?
- (c) What is the hex address of the 256th memory location? (*Note:* The first address is always 0.)
- (d) The computer program is stored in the lowest 2 kbyte block of memory. Give the start and end address of this block.
- 2-38. In an audio CD, the audio voltage signal is typically sampled about 44,000 times per second, and the value of each sample is recorded on the CD surface as a binary number. In other words, each recorded binary number represents a single voltage point on the audio signal waveform.
- (a) If the binary numbers are six bits in length, how many different voltage values can be represented by a single binary number? Repeat for eight bits and ten bits.
- (b) If ten-bit numbers are used, how many bits will be recorded on the CD in 1 second?
- (c) If a CD can typically store 5 billion bits, how many seconds of audio can be recorded when ten-bit numbers are used?
- 2-39. \*A black-and-white digital camera lays a fine grid over an image and then measures and records a binary number representing the level of gray it sees in each cell of the grid. For example, if four-bit numbers are used, the value of black is set to 0000 and the value of white to 1111, and any level of gray is somewhere between 0000 and 1111. If six-bit numbers are used, black is 000000, white is 111111, and all grays are between the two.
- Suppose we wanted to distinguish among 254 different levels of gray within each cell of the grid. How many bits would we need to use to represent these levels?
- 2-40. A 3-Megapixel digital camera stores an eight-bit number for the brightness of each of the primary colors (red, green, blue) found in each picture element (pixel). If every bit is stored (no data compression),

how many pictures can be stored on a 128-Megabyte memory card?  
(Note: In digital systems, Mega means  $2^{20}$ .)

- 2-41. Construct a table showing the binary, hex, and BCD representations of all decimal numbers from 0 to 15. Compare your table with Table 2-4.

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

---

### SECTION 2-1

1. 2267    2. 32768    3. 2267

### SECTION 2-2

1. 1010011    2. 1011011001    3. 20 bits

### SECTION 2-3

1. 9422    2. C2D; 110000101101    3. 97B5    4. E9E, E9F, EA0, EA1  
5. 11010100100111    6. 0 to 65,535

### SECTION 2-4

1.  $10110010_2$ ; 000101111000 (BCD)    2. 32    3. Advantage: ease of conversion.  
Disadvantage: BCD requires more bits.

### SECTION 2-5

1. 0111    2. 0110

### SECTION 2-7

1. One    2. 9999    3. One    4. One

### SECTION 2-8

1. 43, 4F, 53, 54, 20, 3D, 20, 24, 37, 32    2. STOP

### SECTION 2-9

1. A4    2. 001101001    3. Two errors in the data would not change the oddness or evenness of the number of 1s in the data.
-





# DESCRIBING LOGIC CIRCUITS

## ■ OUTLINE

- 3-1 Boolean Constants and Variables
- 3-2 Truth Tables
- 3-3 OR Operation with OR Gates
- 3-4 AND Operation with AND Gates
- 3-5 NOT Operation
- 3-6 Describing Logic Circuits Algebraically
- 3-7 Evaluating Logic-Circuit Outputs
- 3-8 Implementing Circuits from Boolean Expressions
- 3-9 NOR Gates and NAND Gates
- 3-10 Boolean Theorems
- 3-11 DeMorgan's Theorems
- 3-12 Universality of NAND Gates and NOR Gates
- 3-13 Alternate Logic-Gate Representations
- 3-14 Which Gate Representation to Use
- 3-15 Propagation Delay
- 3-16 Summary of Methods to Describe Logic Circuits
- 3-17 Description Languages Versus Programming Languages
- 3-18 Implementing Logic Circuits with PLDs
- 3-19 HDL Format and Syntax
- 3-20 Intermediate Signals

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Perform the three basic logic operations.
- Describe the operation of and construct the truth tables for the AND, NAND, OR, and NOR gates, and the NOT (INVERTER) circuit.
- Draw timing diagrams for the various logic-circuit gates.
- Write the Boolean expression for the logic gates and combinations of logic gates.
- Implement logic circuits using basic AND, OR, and NOT gates.
- Use Boolean algebra to simplify complex logic circuits.
- Use DeMorgan's theorems to simplify logic expressions.
- Use either of the universal gates (NAND or NOR) to implement a circuit represented by a Boolean expression.
- Explain the advantages of constructing a logic-circuit diagram using the alternate gate symbols versus the standard logic-gate symbols.
- Describe the concept of active-LOW and active-HIGH logic signals.
- Describe and measure propagation delay time.
- Use several methods to describe the operation of logic circuits.
- Interpret simple circuits defined by a hardware description language (HDL).
- Explain the difference between an HDL and a computer programming language.
- Create an HDL file for a simple logic gate.
- Create an HDL file for combinational circuits with intermediate variables.

## ■ INTRODUCTION

Chapters 1 and 2 introduced the concepts of logic levels and logic circuits. In logic, only two possible conditions exist for any input or output: true and false. The binary number system uses only two digits, 1 and 0, so it is perfect for representing logical relationships. Digital logic circuits use predefined voltage ranges to represent these binary states. Using these concepts, we can create circuits made of little more than processed beach sand and wire that make consistent, intelligent, logical decisions. It is vitally important that we have a method to describe the logical decisions made by these circuits. In other words, we must describe how they operate. In this chapter, we will discover many ways to describe their operation. Each description method is important because all these methods commonly

appear in technical literature and system documentation and are used in conjunction with modern design and development tools.

Life is full of examples of circumstances that are in one state or another. For example, a creature is either alive or dead, a light is either on or off, a door is locked or unlocked, and it is either raining or it is not. In 1854, a mathematician named George Boole wrote *An Investigation of the Laws of Thought*, in which he described the way we make logical decisions based on true or false circumstances. The methods he described are referred to today as Boolean logic, and the system of using symbols and operators to describe these decisions is called Boolean algebra. In the same way we use symbols such as  $x$  and  $y$  to represent unknown numerical values in regular algebra, Boolean algebra uses symbols to represent a logical expression that has one of two possible values: true or false. The logical expression might be *door is closed*, *button is pressed*, or *fuel is low*. Writing these expressions is very tedious, and so we tend to substitute symbols such as  $A$ ,  $B$ , and  $C$ .

The main purpose of these logical expressions is to describe the relationship between a logic circuit's output (the decision) and its inputs (the circumstances). In this chapter, we will study the most basic logic circuits—*logic gates*—which are the fundamental building blocks from which all other logic circuits and digital systems are constructed. We will see how the operation of the different logic gates and the more complex circuits formed from combinations of logic gates can be described and analyzed using Boolean algebra. We will also get a glimpse of how Boolean algebra can be used to simplify a circuit's Boolean expression so that the circuit can be rebuilt using fewer logic gates and/or fewer connections. Much more will be done with circuit simplification in Chapter 4.

Boolean algebra is not only used as a tool for analysis and simplification of logic systems. It can also be used as a tool to create a logic circuit that will produce the desired input/output relationship. This process is often called synthesis of logic circuits as opposed to analysis. Other techniques have been used in the analysis, synthesis, and documentation of logic systems and circuits including truth tables, schematic symbols, timing diagrams, and—last but by no means least—language. To categorize these methods, we could say that Boolean algebra is a mathematic tool, truth tables are data organizational tools, schematic symbols are drawing tools, timing diagrams are graphing tools, and language is the universal description tool.

Today, any of these tools can be used to provide input to computers. The computers can be used to simplify and translate between these various forms of description and ultimately provide an output in the form necessary to implement a digital system. To take advantage of the powerful benefits of computer software, we must first fully understand the acceptable ways for describing these systems in terms the computer can understand. This chapter will lay the groundwork for further study of these vital tools for synthesis and analysis of digital systems.

Clearly the tools described here are invaluable tools in describing, analyzing, designing, and implementing digital circuits. The student who expects to work in the digital field must work hard at understanding and becoming comfortable with Boolean algebra (believe us, it's much, much easier than conventional algebra) and all the other tools. Do *all* of the examples, exercises, and problems, even the ones your instructor doesn't assign. When those run out, make up your own. The time you spend will be well worth it because you will see your skills improve and your confidence grow.

---

## 3-1 BOOLEAN CONSTANTS AND VARIABLES

### OUTCOMES

Upon completion of this section, you will be able to:

- Differentiate between Boolean variables and constants.
- State the possible values of a Boolean variable or constant.
- Define Boolean algebra.

Boolean algebra differs in a major way from ordinary algebra because Boolean constants and variables are allowed to have only two possible values, 0 or 1. A Boolean variable is a quantity that may, at different times, be equal to either 0 or 1. Boolean variables are often used to represent the voltage level present on a wire or at the input/output terminals of a circuit. For example, in a certain digital system, the Boolean value of 0 might be assigned to any voltage in the range from 0 to 0.8 V, while the Boolean value of 1 might be assigned to any voltage in the range 2 to 5 V.\*

Thus, Boolean 0 and 1 do not represent actual numbers but instead represent the state of a voltage variable, or what is called its **logic level**. A voltage in a digital circuit is said to be at the logic 0 level or the logic 1 level, depending on its actual numerical value. In digital logic, several other terms are used synonymously with 0 and 1. Some of the more common ones are shown in Table 3-1. We will use the 0/1 and LOW/HIGH designations most of the time.

**TABLE 3-1** Common logic terms.

Logic 0	Logic 1
False	True
Off	On
LOW	HIGH
No	Yes
Open switch	Closed switch

As we said in the introduction, **Boolean algebra** is a means for expressing the relationship between a logic circuit's inputs and outputs. The inputs are considered logic variables whose logic levels at any time determine the output levels. In all our work to follow, we shall use letter symbols to represent logic variables. For example, the letter *A* might represent a certain digital circuit input or output, and at any time we must have either  $A = 0$  or  $A = 1$ : if not one, then the other.

Because only two values are possible, Boolean algebra is relatively easy to work with compared with ordinary algebra. In Boolean algebra, there are no fractions, decimals, negative numbers, square roots, cube roots, logarithms, imaginary numbers, and so on. In fact, in Boolean algebra there are only *three* basic operations: *OR*, *AND*, and *NOT*.

These basic operations are called *logic operations*. Digital circuits called *logic gates* can be constructed from diodes, transistors, and resistors connected so that the circuit output is the result of a basic logic operation (*OR*, *AND*, *NOT*) performed on the inputs. We will be using Boolean algebra first

\*Voltages between 0.8 and 2 V are undefined (neither 0 nor 1) and should not occur under normal circumstances.

to describe and analyze these basic logic gates, then later to analyze and design combinations of logic gates connected as logic circuits.

A Boolean constant represents a point in the circuit where the logic level never changes. In other words, each bit is “hard-wired” to either a logic 1 or a logic 0. A good name for a constant logic 1 would be VCC or HIGH. VCC is a common name given to the positive voltage supply of digital systems. A good name for a constant logic 0 would be LOW or GND. GND stands for *ground* which is the term used for the negative side of the power supply for digital circuits. Recall from Chapter 1 that these voltage levels are used to represent 1s and 0s.

### OUTCOME ASSESSMENT QUESTIONS

1. A circuit has more inputs than your application needs. The extra inputs will not affect your application if they are LOW. Should you apply a variable or a constant? What would be a good name for this point in the circuit?
2. The quadrature encoder (two-bit Gray code) described in Chapter 2 has its two channels A and B connected as inputs to a logic circuit. Should these inputs be labeled as variables or constants? What would be a good name for each input?
3. Define Boolean algebra.

## 3-2 TRUTH TABLES

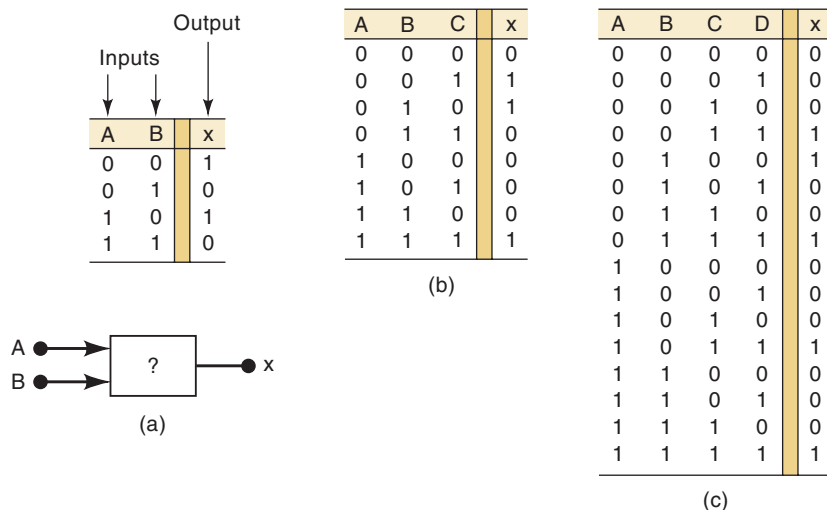
### OUTCOMES

Upon completion of this section, you will be able to:

- Construct a truth table.
- For any input, identify the correct output.
- Determine the size of the truth table based on the number of variables.

A **truth table** is a means for describing how a logic circuit’s output depends on the logic levels present at the circuit’s inputs. Figure 3-1(a) illustrates a truth table for one type of two-input logic circuit. The table lists all possible combinations of logic levels present at inputs *A* and *B*, along with the corresponding output level *x*. The first entry in the table shows that when *A* and *B*

**FIGURE 3-1** Example truth tables for (a) two-input, (b) three-input, and (c) four-input circuits.



are both at the 0 level, the output  $x$  is at the 1 level or, equivalently, in the 1 state. The second entry shows that when input  $B$  is changed to the 1 state, so that  $A = 0$  and  $B = 1$ , the output  $x$  becomes a 0. In a similar way, the table shows what happens to the output state for any set of input conditions.

Figures 3-1(b) and (c) show samples of truth tables for three- and four-input logic circuits. Again, each table lists all possible combinations of input logic levels on the left, with the resultant logic level for output  $x$  on the right. Of course, the actual values for  $x$  will depend on the type of logic circuit.

Note that there are 4 table entries for the two-input truth table, 8 entries for a three-input truth table, and 16 entries for the four-input truth table. The number of input combinations will equal  $2^N$  for an  $N$ -input truth table. Also note that the list of all possible input combinations follows the binary counting sequence, and so it is an easy matter to write down all of the combinations without missing any.

### OUTCOME ASSESSMENT QUESTIONS

1. What is the output state of the four-input circuit represented in Figure 3-1(c) when all inputs except  $B$  are 1?
2. Repeat question 1 for the following input conditions:  $A = 1, B = 0, C = 1, D = 0$ .
3. How many table entries are needed for a five-input circuit?

## 3-3 OR OPERATION WITH OR GATES

### OUTCOMES

Upon completion of this section, you will be able to:

- Define the OR logic function.
- Write Boolean equations using the OR function.
- Draw the logic symbol for the OR function.
- Write a truth table describing the OR function.
- Draw a timing diagram that demonstrates the OR function.
- Use any of the above methods to infer the correct output of a logic circuit based on its input.

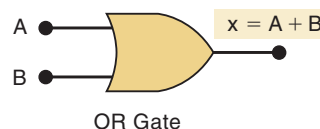
The **OR operation** is the first of the three basic Boolean operations to be learned. An example can be found in the kitchen oven. The light inside the oven should turn on if either the *oven light switch is on* OR if the *door is opened*. The letter  $A$  could be used to represent the *oven light switch is on* (true or false) and  $B$  could represent *door is opened* (true or false). The letter  $x$  could represent the *light is on* (true or false). The truth table in Figure 3-2(a) shows

**FIGURE 3-2** (a) Truth table defining the OR operation; (b) circuit symbol for a two-input OR gate.



OR		
A	B	$x = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

(a)



(b)

what happens when two logic inputs,  $A$  and  $B$ , are combined using the OR operation to produce the output  $x$ . The table shows that  $x$  is a logic 1 for every combination of input levels where one or more inputs are 1. The only case where  $x$  is a 0 is when both inputs are 0.

The Boolean expression for the OR operation is

$$x = A + B$$

In this expression, the  $+$  sign does not stand for ordinary addition; it stands for the OR operation. The OR operation is similar to ordinary addition except for the case where  $A$  and  $B$  are both 1; the OR operation produces  $1 + 1 = 1$ , not  $1 + 1 = 2$ . In Boolean algebra, 1 is as high as we go, so we can never have a result greater than 1. The same holds true for combining three inputs using the OR operation. Here we have  $x = A + B + C$ . If we consider the case where all three inputs are 1, we have

$$x = 1 + 1 + 1 = 1$$

The expression  $x = A + B$  is read as “ $x$  equals  $A$  OR  $B$ ,” which means that  $x$  will be 1 when  $A$  or  $B$  or both are 1. Likewise, the expression  $x = A + B + C$  is read as “ $x$  equals  $A$  OR  $B$  OR  $C$ ,” which means that  $x$  will be 1 when  $A$  or  $B$  or  $C$  or any combination of them are 1. To describe this circuit in the English language, we could say that  $x$  is true (1) WHEN  $A$  is true (1) OR  $B$  is true (1) OR  $C$  is true (1).

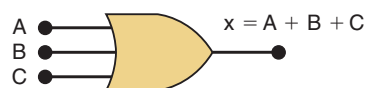
## OR Gate

In digital circuitry, an **OR gate**\* is a circuit that has two or more inputs and whose output is equal to the OR combination of the inputs. Figure 3-2(b) is the logic symbol for a two-input OR gate. The inputs  $A$  and  $B$  are logic voltage levels, and the output  $x$  is a logic voltage level whose value is the result of the OR operation on  $A$  and  $B$ ; that is,  $x = A + B$ . In other words, the OR gate operates so that its output is HIGH (logic 1) if either input  $A$  or  $B$  or both are at a logic 1 level. The OR gate output will be LOW (logic 0) only if all its inputs are at logic 0.

This same idea can be extended to more than two inputs. Figure 3-3 shows a three-input OR gate and its truth table. Examination of this truth table shows again that the output will be 1 for every case where one or more inputs are 1. This general principle is the same for OR gates with any number of inputs.

Using the language of Boolean algebra, the output  $x$  can be expressed as  $x = A + B + C$ , where again it must be emphasized that the  $+$  represents

**FIGURE 3-3** Symbol and truth table for a three-input OR gate.



A	B	C	$x = A + B + C$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

\*The term *gate* comes from the inhibit/enable operation discussed in Chapter 4.

the OR operation. The output of any OR gate, then, can be expressed as the OR combination of its various inputs. We will put this to use when we subsequently analyze logic circuits.

### Summary of the OR Operation

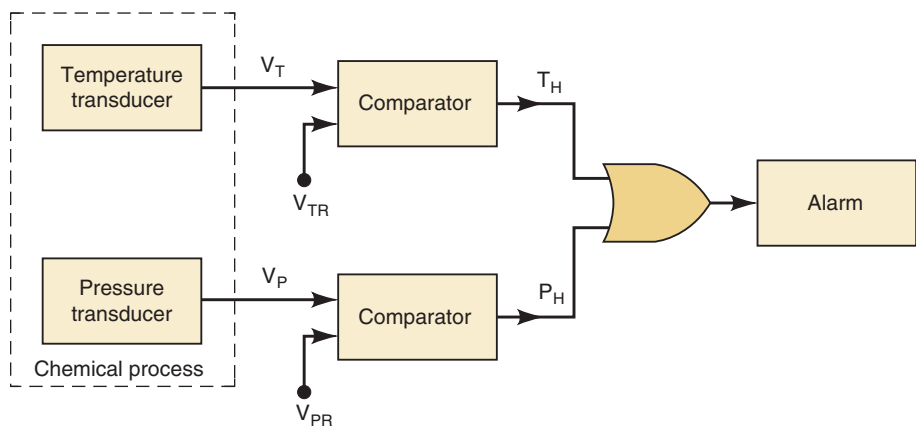
The important points to remember concerning the OR operation and OR gates are:

1. The OR operation produces a result (output) of 1 whenever *any* input is a 1. Otherwise the output is 0.
2. An OR gate is a logic circuit that performs an OR operation on the circuit's inputs.
3. The expression  $x = A + B$  is read as “ $x$  equals  $A$  OR  $B$ .”

#### EXAMPLE 3-1

In many industrial control systems, it is required to activate an output function whenever any one of several inputs is activated. For example, in a chemical process it may be desired that an alarm be activated whenever the process temperature exceeds a maximum value *or* whenever the pressure goes above a certain limit. Figure 3-4 is a block diagram of this situation. The temperature transducer circuit produces an output voltage proportional to the process temperature. This voltage,  $V_T$ , is compared with a temperature reference voltage,  $V_{TR}$ , in a voltage comparator circuit. The comparator output,  $T_H$ , is normally a low voltage (logic 0), but it switches to a high voltage (logic 1) when  $V_T$  exceeds  $V_{TR}$ , indicating that the process temperature is too high. A similar arrangement is used for the pressure measurement, so that its associated comparator output,  $P_H$ , goes from LOW to HIGH when the pressure is too high. What is the purpose of the OR gate?

**FIGURE 3-4** Example of the use of an OR gate in an alarm system.



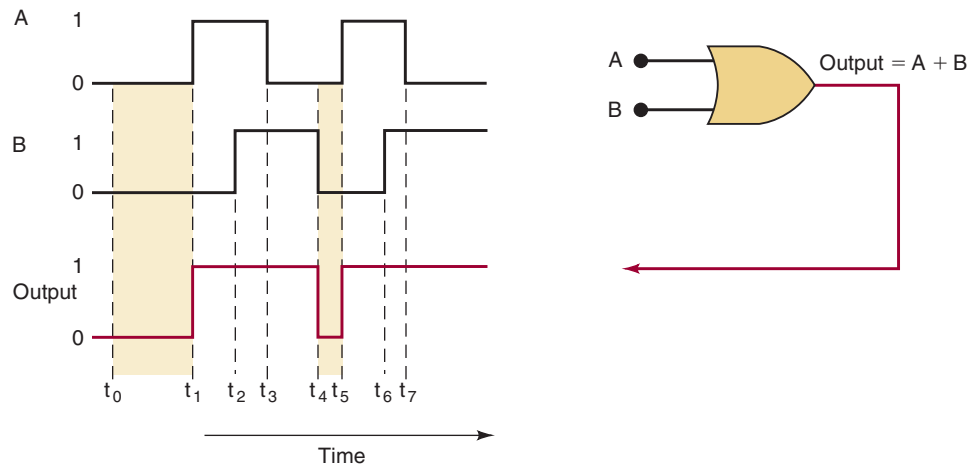
#### Solution

Since we want the alarm to be activated when either temperature *or* pressure is too high, it should be apparent that the two comparator outputs can be fed to a two-input OR gate. The OR gate output thus goes HIGH (1) for either alarm condition and will activate the alarm. This same idea can obviously be extended to situations with more than two process variables.



**EXAMPLE 3-2**

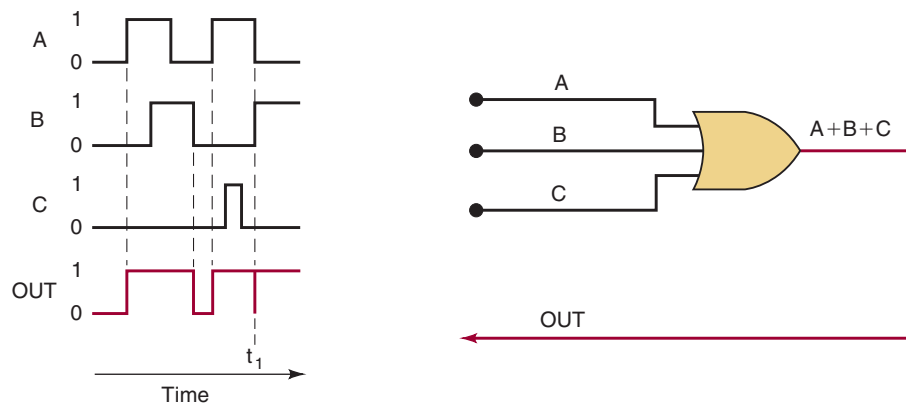
Determine the OR gate output in Figure 3-5. The OR gate inputs  $A$  and  $B$  are varying according to the timing diagrams shown. For example,  $A$  starts out LOW at time  $t_0$ , goes HIGH at  $t_1$ , back to LOW at  $t_3$ , and so on.

**FIGURE 3-5** Example 3-2.**Solution**

The OR gate output will be HIGH whenever *any* input is HIGH. Between time  $t_0$  and  $t_1$ , both inputs are LOW, so  $OUTPUT = LOW$ . At  $t_1$ , input  $A$  goes HIGH while  $B$  remains LOW. This causes  $OUTPUT$  to go HIGH at  $t_1$  and stay HIGH until  $t_4$  because, during this interval, one or both inputs are HIGH. At  $t_4$ , input  $B$  goes from 1 to 0 so that now both inputs are LOW, and this drives  $OUTPUT$  back to LOW. At  $t_5$ ,  $A$  goes HIGH, sending  $OUTPUT$  back HIGH, where it stays for the rest of the shown time span.

**EXAMPLE 3-3A**

For the situation depicted in Figure 3-6, determine the waveform at the OR gate output.

**FIGURE 3-6** Examples 3-3A and B.**Solution**

The three OR gate inputs  $A$ ,  $B$ , and  $C$  are varying, as shown by their waveform diagrams. The OR gate output is determined by realizing that it will be HIGH whenever *any* of the three inputs is at a HIGH level. Using this reasoning, the OR output waveform is as shown in the figure. Particular

attention should be paid to what occurs at time  $t_1$ . The diagram shows that at that instant of time, input  $A$  is going from HIGH to LOW while input  $B$  is going from LOW to HIGH. Since these inputs are making their transitions at approximately the same time, and since these transitions take a certain amount of time, there is a short interval when these OR gate inputs are both in the undefined range between 0 and 1. When this occurs, the OR gate output also becomes a value in this range, as evidenced by the glitch or spike on the output waveform at  $t_1$ . The occurrence of this glitch and its size (amplitude and width) depend on the speed with which the input transitions occur.

**EXAMPLE 3-3B**

What would happen to the glitch in the output in Figure 3-6 if input  $C$  sat in the HIGH state while  $A$  and  $B$  were changing at time  $t_1$ ?

**Solution**

With the  $C$  input HIGH at  $t_1$ , the OR gate output will remain HIGH, regardless of what is occurring at the other inputs, because any HIGH input will keep an OR gate output HIGH. Therefore, the glitch will not appear in the output.

**OUTCOME ASSESSMENT QUESTIONS**

1. What is the only set of input conditions that will produce a LOW output for any OR gate?
2. Write the Boolean expression for a six-input OR gate.
3. If the  $A$  input in Figure 3-6 is permanently kept at the 1 level, what will the resultant output waveform be?

**3-4 AND OPERATION WITH AND GATES****OUTCOMES**

*Upon completion of this section, you will be able to:*

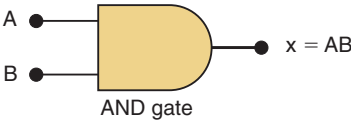
- Define the AND logic function.
- Write Boolean equations using the AND function.
- Draw the logic symbol for the AND function.
- Write a truth table describing the AND function.
- Draw a timing diagram that demonstrates the AND function.
- Use any of the above methods to infer the correct output of a logic circuit based on its input.

The **AND operation** is the second basic Boolean operation. As an example of the use of AND logic, consider a typical clothes dryer. It is drying clothes (heating, tumbling) only if the *timer is set above zero* AND the *door is closed*. Let's assign  $A$  to represent *timer is set* (T/F),  $B$  to represent *door is closed* (T/F), and  $x$  can represent the *heater and motor are on* (T/F). The truth table in Figure 3-7(a) shows what happens when two logic inputs,  $A$  and  $B$ , are combined using the AND operation to produce output  $x$ . The table shows that  $x$  is a logic 1 only when both  $A$  and  $B$  are at the logic 1 level. For any case where one of the inputs is 0, the output is 0.

**FIGURE 3-7** (a) Truth table for the AND operation; (b) AND gate symbol.

AND		
A	B	$x = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

(a)



(b)

The Boolean expression for the AND operation is

$$x = A \cdot B$$

In this expression, the  $\cdot$  sign stands for the Boolean AND operation and not the multiplication operation. However, the AND operation on Boolean variables operates the same as ordinary multiplication, as examination of the truth table shows, so we can think of them as being the same. This characteristic can be helpful when evaluating logic expressions that contain AND operations.

The expression  $x = A \cdot B$  is read as “ $x$  equals  $A$  AND  $B$ ,” which means that  $x$  will be 1 only when  $A$  and  $B$  are both 1. The  $\cdot$  sign is usually omitted so that the expression simply becomes  $x = AB$ . For the case when three inputs are ANDed, we have  $x = A \cdot B \cdot C = ABC$ . This is read as “ $x$  equals  $A$  AND  $B$  AND  $C$ ,” which means that  $x$  will be 1 only when  $A$  and  $B$  and  $C$  are all 1.

### AND Gate

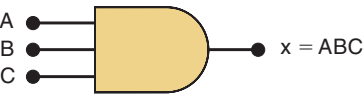
The logic symbol for a two-input **AND gate** is shown in Figure 3-7(b). The AND gate output is equal to the AND product of the logic inputs; that is,  $x = AB$ . In other words, the AND gate is a circuit that operates so that its output is HIGH only when all its inputs are HIGH. For all other cases, the AND gate output is LOW.

This same operation is characteristic of AND gates with more than two inputs. For example, a three-input AND gate and its accompanying truth table are shown in Figure 3-8. Once again, note that the gate output is 1 only for the case where  $A = B = C = 1$ . The expression for the output is  $x = ABC$ . For a four-input AND gate, the output is  $x = ABCD$ , and so on.

**FIGURE 3-8** Truth table and symbol for a three-input AND gate.



A	B	C	$x = ABC$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



Note the difference between the symbols for the AND gate and the OR gate. Whenever you see the AND symbol on a logic-circuit diagram, it tells you that the output will go HIGH *only* when *all* inputs are HIGH. Whenever you see the OR symbol, it means that the output will go HIGH when *any* input is HIGH.

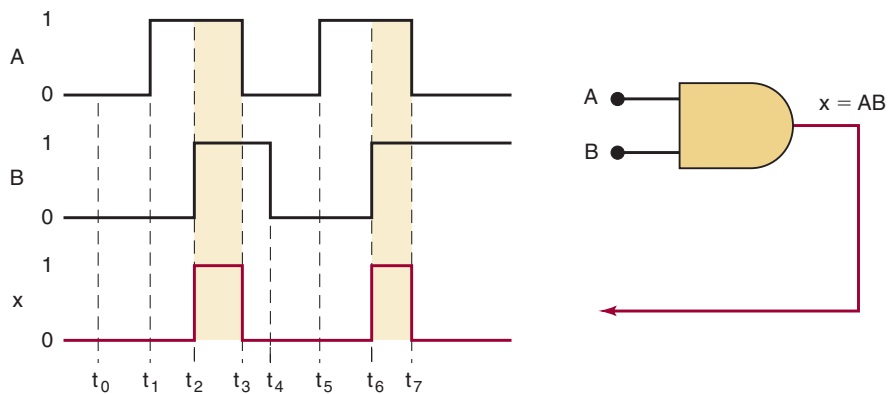
## Summary of the AND Operation

1. The AND operation is performed the same as ordinary multiplication of 1s and 0s.
2. An AND gate is a logic circuit that performs the AND operation on the circuit's inputs.
3. An AND gate output will be 1 *only* for the case when *all* inputs are 1; for all other cases, the output will be 0.
4. The expression  $x = AB$  is read as “ $x$  equals  $A$  AND  $B$ .”

### EXAMPLE 3-4

Determine the output  $x$  from the AND gate in Figure 3-9 for the given input waveforms.

FIGURE 3-9 Example 3-4.



### Solution

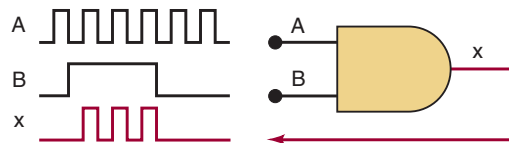
The output of an AND gate is determined by realizing that it will be HIGH only when all inputs are HIGH at the same time. For the input waveforms given, this condition is met only during intervals  $t_2 - t_3$  and  $t_6 - t_7$ . At all other times, one or more of the inputs are 0, thereby producing a LOW output. Note that input level changes that occur while the other input is LOW have no effect on the output.

### EXAMPLE 3-5A



Determine the output waveform for the AND gate shown in Figure 3-10.

FIGURE 3-10 Examples 3-5A and B.



### Solution

The output  $x$  will be at 1 only when  $A$  and  $B$  are both HIGH at the same time. Using this fact, we can determine the  $x$  waveform as shown in the figure.

Notice that the  $x$  waveform is 0 whenever  $B$  is 0, regardless of the signal at  $A$ . Also notice that whenever  $B$  is 1, the  $x$  waveform is the same as  $A$ . Thus, we can think of the  $B$  input as a *control* input whose logic level determines whether or not the  $A$  waveform gets through to the  $x$  output. In this situation, the AND gate is used as an *inhibit circuit*. We can say that  $B = 0$  is the inhibit condition producing a 0 output. Conversely,  $B = 1$  is the *enable* condition, which enables  $A$  to reach the output. This inhibit operation is an important application of AND gates, which will be encountered later.

**EXAMPLE 3-5B**

What will happen to the  $x$  output waveform in Figure 3-10 if the  $B$  input is kept at the 0 level?

**Solution**

With  $B$  kept LOW, the  $x$  output will also stay LOW. This can be reasoned in two different ways. First, with  $B = 0$  we have  $x = A \cdot B = A \cdot 0 = 0$  because anything multiplied (ANDed) by 0 will be 0. Another way to look at it is that an AND gate requires that all inputs be HIGH for the output to be HIGH, and this cannot happen if  $B$  is kept LOW.

**OUTCOME ASSESSMENT QUESTIONS**

1. What is the only input combination that will produce a HIGH at the output of a five-input AND gate?
2. What logic level should be applied to the second input of a two-input AND gate if the logic signal at the first input is to be inhibited (prevented) from reaching the output?
3. *True or false:* An AND gate output will always differ from an OR gate output for the same input conditions.

**3-5 NOT OPERATION****OUTCOMES**

*Upon completion of this section, you will be able to:*

- Define the NOT logic function.
- Write Boolean equations using the NOT function.
- Draw the logic symbol for the NOT function.
- Write a truth table describing the NOT function.
- Draw a timing diagram that demonstrates the NOT function.

The **NOT operation** is unlike the OR and AND operations because it can be performed on a single input variable. For example, if the variable  $A$  is subjected to the NOT operation, the result  $x$  can be expressed as

$$x = \bar{A}$$

where the overbar represents the NOT operation. This expression is read as “ $x$  equals NOT  $A$ ” or “ $x$  equals the *inverse* of  $A$ ” or “ $x$  equals the *complement* of  $A$ .” Each of these is in common usage, and all indicate that the logic value of  $x = \bar{A}$  is *opposite* to the logic value of  $A$ . The truth table in Figure 3-11(a) clarifies this for the two cases  $A = 0$  and  $A = 1$ . That is,

$$0 = \bar{1} \quad \text{because 0 is not 1}$$

and

$$1 = \bar{0} \quad \text{because 1 is not 0}$$

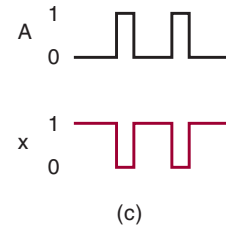
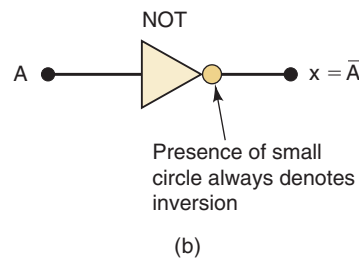
The NOT operation is also referred to as **inversion** or **complementation**, and these terms will be used interchangeably throughout the book. Although we will always use the overbar indicator to represent inversion, it is important

**FIGURE 3-11** (a) Truth table; (b) symbol for the INVERTER (NOT circuit); (c) sample waveforms.



NOT	
A	$x = \bar{A}$
0	1
1	0

(a)



to mention that another indicator for inversion is the prime symbol ( $'$ ). That is,

$$A' = \bar{A}$$

Both should be recognized as indicating the inversion operation.

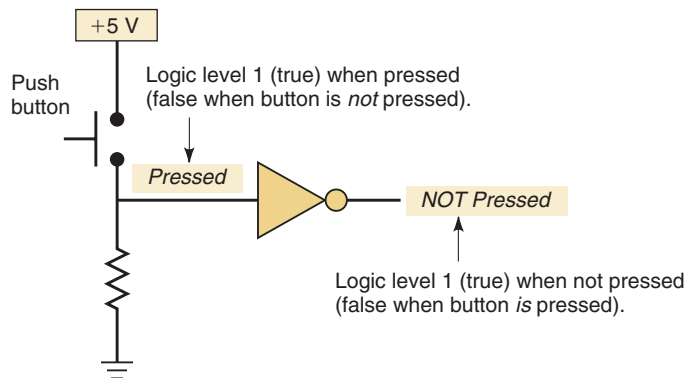
### NOT Circuit (INVERTER)

Figure 3-11(b) shows the symbol for a **NOT circuit**, which is more commonly called an **INVERTER**. This circuit *always* has only a single input, and its output logic level is always opposite to the logic level of this input. Figure 3-11(c) shows how the INVERTER affects an input signal. It inverts (complements) the input signal at all points on the waveform so that whenever the input = 0, output = 1, and vice versa.

#### APPLICATION 3-1

Figure 3-12 shows a typical application of the NOT gate. The push button is wired to produce a logic 1 (true) when it is pressed. Sometimes we want to know if the push button is not being pressed, and so this circuit provides an expression that is true when the button is not pressed.

**FIGURE 3-12** A NOT gate indicating a button is *not* pressed when its output is true.



### Summary of Boolean Operations

The rules for the OR, AND, and NOT operations may be summarized as follows:

<b>OR</b>	<b>AND</b>	<b>NOT</b>
$0 + 0 = 0$	$0 \cdot 0 = 0$	$\bar{0} = 1$
$0 + 1 = 1$	$0 \cdot 1 = 0$	$\bar{1} = 0$
$1 + 0 = 1$	$1 \cdot 0 = 0$	
$1 + 1 = 1$	$1 \cdot 1 = 1$	

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. The output of the INVERTER of Figure 3-11 is connected to the input of a second INVERTER. Determine the output level of the second INVERTER for each level of input  $A$ .
2. The output of the AND gate in Figure 3-7 is connected to the input of an INVERTER. Write the truth table showing the INVERTER output,  $y$ , for each combination of inputs  $A$  and  $B$ .

### 3-6 DESCRIBING LOGIC CIRCUITS ALGEBRAICALLY

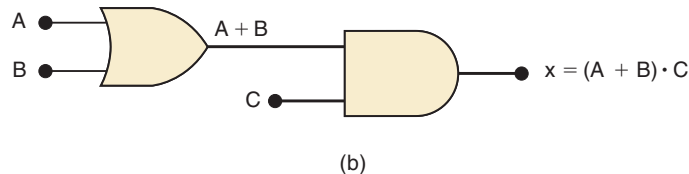
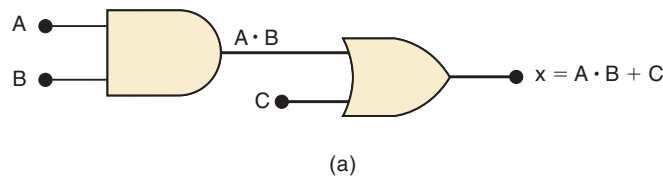
#### OUTCOME

Upon completion of this section, you will be able to:

- Translate logic diagrams into Boolean algebraic expressions.

Any logic circuit, no matter how complex, can be described completely using the three basic Boolean operations because the OR gate, AND gate, and NOT circuit are the basic building blocks of digital systems. For example, consider the circuit in Figure 3-13(a). This circuit has three inputs,  $A$ ,  $B$ , and  $C$ , and a single output,  $x$ . Utilizing the Boolean expression for each gate, we can easily determine the expression for the output.

**FIGURE 3-13** (a) Logic circuit with its Boolean expression; (b) logic circuit whose expression requires parentheses.



The expression for the AND gate output is written  $A \cdot B$ . This AND output is connected as an input to the OR gate along with  $C$ , another input. The OR gate operates on its inputs so that its output is the OR sum of the inputs. Thus, we can express the OR output as  $x = A \cdot B + C$ . (This final expression could also be written as  $x = C + A \cdot B$  because it does not matter which term of the OR sum is written first.)

#### Operator Precedence

Occasionally, there may be confusion about which operation in an expression is performed first. The expression  $A \cdot B + C$  can be interpreted in two different ways: (1)  $A \cdot B$  is ORed with  $C$ , or (2)  $A$  is ANDed with the term  $B + C$ . To avoid this confusion, it will be understood that if an expression contains both AND and OR operations, the AND operations are performed first, unless there are *parentheses* in the expression, in which case the operation inside the parentheses is to be performed first. This is the same rule that is used in ordinary algebra to determine the order of operations.

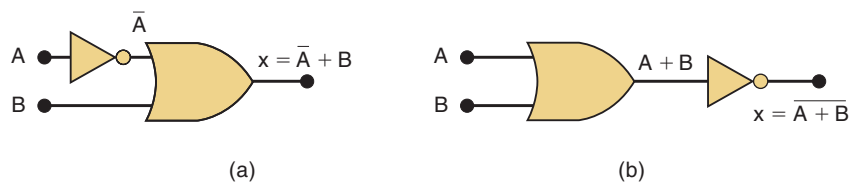
To illustrate further, consider the circuit in Figure 3-13(b). The expression for the OR gate output is simply  $A + B$ . This output serves as an input

to the AND gate along with another input,  $C$ . Thus, we express the output of the AND gate as  $x = (A + B) \cdot C$ . Note the use of parentheses here to indicate that  $A$  and  $B$  are ORed *first*, before their OR sum is ANDed with  $C$ . Without the parentheses it would be interpreted *incorrectly*, because  $A + B \cdot C$  means that  $A$  is ORed with the product  $B \cdot C$ .

### Circuits Containing INVERTERS

Whenever an INVERTER is present in a logic-circuit diagram, its output expression is simply equal to the input expression with a bar over it. Figure 3-14 shows two examples using INVERTERS. In Figure 3-14(a), input  $A$  is fed through an INVERTER, whose output is therefore  $\bar{A}$ . The INVERTER output is fed to an OR gate together with  $B$ , so that the OR output is equal to  $\bar{A} + B$ . Note that the bar is over the  $A$  alone, indicating that  $A$  is first inverted and then ORed with  $B$ .

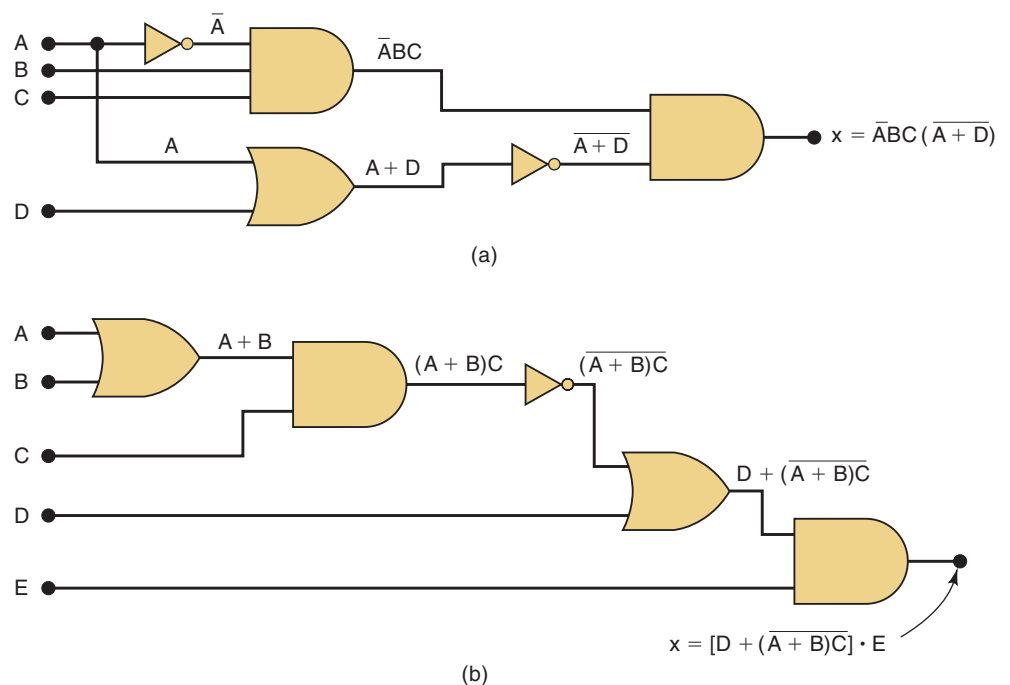
**FIGURE 3-14** Circuits using INVERTERS.



In Figure 3-14(b), the output of the OR gate is equal to  $A + B$  and is fed through an INVERTER. The INVERTER output is therefore equal to  $\overline{(A + B)}$  because it inverts the *complete* input expression. Note that the bar covers the entire expression  $(A + B)$ . This is important because, as will be shown later, the expressions  $\overline{(A + B)}$  and  $\bar{A} + \bar{B}$  are *not* equivalent. The expression  $\overline{(A + B)}$  means that  $A$  is ORed with  $B$  and then their OR sum is inverted, whereas the expression  $\bar{A} + \bar{B}$  indicates that  $A$  is inverted and  $B$  is inverted and the results are then ORed together.

Figure 3-15 shows two more examples, which should be studied carefully. Note especially the use of *two* separate sets of parentheses in Figure 3-15(b).

**FIGURE 3-15** More examples.





Also notice in Figure 3-15(a) that the input variable  $A$  is connected as an input to two different gates.

### OUTCOME ASSESSMENT QUESTIONS

1. In Figure 3-15(a), change each AND gate to an OR gate, and change the OR gate to an AND gate. Then write the expression for output  $x$ .
2. In Figure 3-15(b), change each AND gate to an OR gate, and each OR gate to an AND gate. Then write the expression for  $x$ .

## 3-7 EVALUATING LOGIC-CIRCUIT OUTPUTS

### OUTCOMES

Upon completion of this section, you will be able to:

- Evaluate any circuit diagram or Boolean equation output given a specific input.
- Translate Boolean equations or logic diagrams into a truth table.

Once we have the Boolean expression for a circuit output, we can obtain the output logic level for any set of input levels. For example, suppose that we want to know the logic level of the output  $x$  for the circuit in Figure 3-15(a) for the case where  $A = 0$ ,  $B = 1$ ,  $C = 1$ , and  $D = 1$ . As in ordinary algebra, the value of  $x$  can be found by “plugging” the values of the variables into the expression and performing the indicated operations as follows:

$$\begin{aligned} x &= \overline{ABC(A + D)} \\ &= \overline{0 \cdot 1 \cdot 1 \cdot (0 + 1)} \\ &= \overline{1 \cdot 1 \cdot 1 \cdot (0 + 1)} \\ &= \overline{1 \cdot 1 \cdot 1 \cdot (1)} \\ &= \overline{1 \cdot 1 \cdot 1 \cdot 0} \\ &= \overline{0} \\ &= 1 \end{aligned}$$

As another illustration, let us evaluate the output of the circuit in Figure 3-15(b) for  $A = 0$ ,  $B = 0$ ,  $C = 1$ ,  $D = 1$ , and  $E = 1$ .

$$\begin{aligned} x &= [D + \overline{(A + B)C}] \cdot E \\ &= [1 + \overline{(0 + 0) \cdot 1}] \cdot 1 \\ &= [1 + \overline{0 \cdot 1}] \cdot 1 \\ &= [1 + \overline{0}] \cdot 1 \\ &= [1 + 1] \cdot 1 \\ &= 1 \cdot 1 \\ &= 1 \end{aligned}$$

In general, the following rules must always be followed when evaluating a Boolean expression:

1. First, perform all inversions of single terms; that is,  $\overline{0} = 1$  or  $\overline{1} = 0$ .
2. Then perform all operations within parentheses.
3. Perform an AND operation before an OR operation unless parentheses indicate otherwise.
4. If an expression has a bar over it, perform the operations inside the expression first and then invert the result.

For practice, determine the outputs of both circuits in Figure 3-15 for the case where all inputs are 1. The answers are  $x = 0$  and  $x = 1$ , respectively.

### Analysis Using a Table

Whenever you have a circuit made up of multiple logic gates and you want to know how it works, the best way to analyze it is to use a truth table. The advantages of this method are:

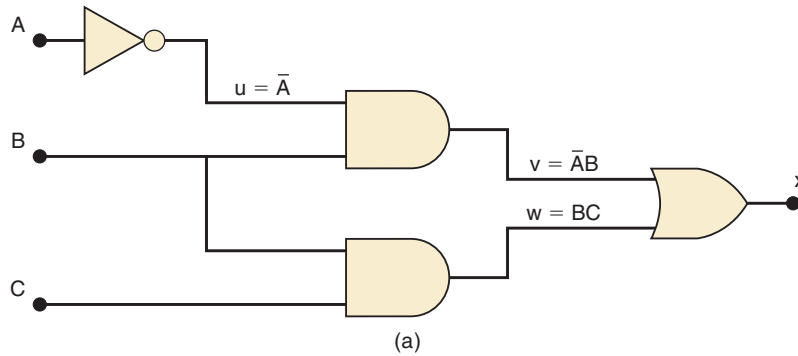
It allows you to analyze one gate or logic combination at a time.

It allows you to easily double-check your work.

When you are done, you have a table that is of tremendous benefit in troubleshooting the logic circuit.

Recall that a truth table lists all the possible input combinations in numerical order. For each possible input combination, you can determine the logic state at every point (node) in the logic circuit including the output. For example refer to Figure 3-16(a). There are several intermediate nodes in this circuit that are neither inputs nor outputs to the circuit. They are simply connections between one gate's output and another gate's input.

**FIGURE 3-16** Analysis of a logic circuit using truth tables.



A	B	C	$u = \bar{A}$	$v = \bar{A}B$	$w = BC$	$x = v+w$
0	0	0	1			
0	0	1	1			
0	1	0	1			
0	1	1	1			
1	0	0	0			
1	0	1	0			
1	1	0	0			
1	1	1	0			

(b)

A	B	C	$u = \bar{A}$	$v = \bar{A}B$	$w = BC$	$x = v+w$
0	0	0	1	0		
0	0	1	1	0		
0	1	0	1	1		
0	1	1	1	1		
1	0	0	0	0		
1	0	1	0	0		
1	1	0	0	0		
1	1	1	0	0		

(c)

A	B	C	$u = \bar{A}$	$v = \bar{A}B$	$w = BC$	$x = v+w$
0	0	0	1	0	0	
0	0	1	1	0	0	
0	1	0	1	1	0	
0	1	1	1	1	1	
1	0	0	0	0	0	
1	0	1	0	0	0	
1	1	0	0	0	0	
1	1	1	0	0	1	

(d)

A	B	C	$u = \bar{A}$	$v = \bar{A}B$	$w = BC$	$x = v+w$
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	1	0	1	1	0	1
0	1	1	1	1	1	1
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	1	1

(e)

In this diagram they have been labeled  $u$ ,  $v$ , and  $w$ . The first step after listing all the input combinations is to create a column in the truth table for each intermediate signal (node) as shown in Figure 3-16(b). Node  $u$  has been filled in as the complement of  $A$ .

The next step is to fill in the values for column  $v$  as shown in Figure 3-16(c). From the diagram we can see that  $v = \bar{A}B$ . The node  $v$  should be HIGH when  $\bar{A}$  (node  $u$ ) is HIGH AND  $B$  is HIGH. This occurs whenever  $A$  is LOW AND  $B$  is HIGH. The third step is to predict the values at node  $w$  which is the logical product of  $BC$ . This column is HIGH whenever  $B$  is HIGH AND  $C$  is HIGH as shown in Figure 3-16(d). The final step is to logically combine columns  $v$  and  $w$  to predict the output  $x$ . Since  $x = v + w$ , the  $x$  output will be HIGH when  $v$  is HIGH OR  $w$  is HIGH as shown in Figure 3-16(e).

If you built this circuit and it was not producing the correct output for  $x$  under all conditions, this table could be used to find the trouble. The general procedure is to test the circuit under each combination of inputs. If any input combination produces an incorrect output (i.e., a fault), compare the actual logic state of each intermediate node in the circuit with the correct theoretical value in the table while applying that input condition. If the logic state for an intermediate node is *correct*, the problem must be farther to the right of that node. If the logic state for an intermediate node is *incorrect*, the problem must be to the left of that node (or that node is shorted to something). Detailed troubleshooting procedures and possible circuit faults will be covered more extensively in Chapter 4.

### EXAMPLE 3-6

Analyze the operation of Figure 3-15(a) by creating a table showing the logic state at each node of the circuit.

#### Solution

Fill in the column for  $t$  by entering a 1 for all entries where  $A = 0$  and  $B = 1$  and  $C = 1$ .

Fill in the column for  $u$  by entering a 1 for all entries where  $A = 1$  or  $D = 1$ .

Fill in the column for  $v$  by complementing all entries in column  $u$ .

Fill in the column for  $x$  by entering a 1 for all entries where  $t = 1$  and  $v = 1$ .

A	B	C	D	$t = \bar{A}BC$	$u = A + D$	$v = \bar{u}$	$x = tv$
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	0	1	0
0	0	1	1	0	1	0	0
0	1	0	0	0	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	1
0	1	1	1	1	1	0	0
1	0	0	0	0	1	0	0
1	0	0	1	0	1	0	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	1	0	0
1	1	0	1	0	1	0	0
1	1	1	0	0	1	0	0
1	1	1	1	0	1	0	0

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Use the expression for  $x$  to determine the output of the circuit in Figure 3-15(a) for the conditions  $A = 0, B = 1, C = 1,$  and  $D = 0$ .
2. Use the expression for  $x$  to determine the output of the circuit in Figure 3-15(b) for the conditions  $A = B = E = 1, C = D = 0$ .
3. Determine the answers to Questions 1 and 2 by finding the logic levels present at each gate output using a table as in Figure 3-16.

### 3-8 IMPLEMENTING CIRCUITS FROM BOOLEAN EXPRESSIONS

#### OUTCOME

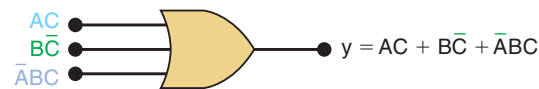
Upon completion of this section, you will be able to:

- Translate any Boolean equation into a logic diagram.

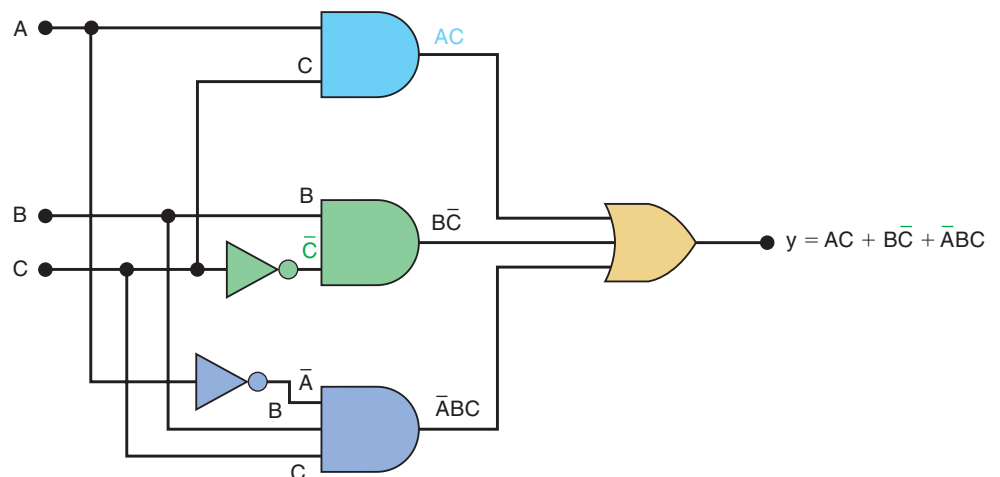
When the operation of a circuit is defined by a Boolean expression, we can draw a logic-circuit diagram directly from that expression. For example, if we needed a circuit that was defined by  $x = A \cdot B \cdot C$ , we would immediately know that all that was needed was a three-input AND gate. If we needed a circuit that was defined by  $x = A + \bar{B}$ , we would use a two-input OR gate with an INVERTER on one of the inputs. The same reasoning used for these simple cases can be extended to more complex circuits.

Suppose that we wanted to construct a circuit whose output is  $y = AC + B\bar{C} + \bar{A}BC$ . This Boolean expression contains three terms ( $AC, B\bar{C}, \bar{A}BC$ ), which are ORed together. This tells us that a three-input OR gate is required with inputs that are equal to  $AC, B\bar{C}$ , and  $\bar{A}BC$ . This is illustrated in Figure 3-17(a), where a three-input OR gate is drawn with inputs labeled as  $AC, B\bar{C}$ , and  $\bar{A}BC$ .

**FIGURE 3-17**  
Constructing a logic circuit from a Boolean expression.



(a)



(b)

Each OR gate input is an AND product term, which means that an AND gate with appropriate inputs can be used to generate each of these terms. This is shown in Figure 3-17(b), which is the final circuit diagram. Note the use of INVERTERS to produce the  $\bar{A}$  and  $\bar{C}$  terms required in the expression.

This same general approach can always be followed, although we shall find that there are some clever, more efficient techniques that can be employed. For now, however, this straightforward method will be used to minimize the number of new items that are to be learned.

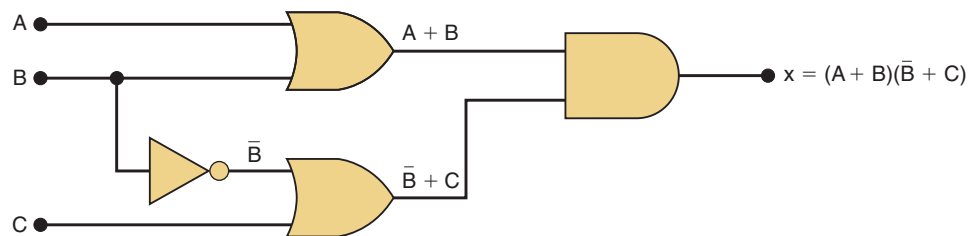
**EXAMPLE 3-7**

Draw the circuit diagram to implement the expression  $x = (A + B)(\bar{B} + C)$ .

**Solution**

This expression shows that the terms  $A + B$  and  $\bar{B} + C$  are inputs to an AND gate, and each of these two terms is generated from a separate OR gate. The result is drawn in Figure 3-18.

**FIGURE 3-18**  
Example 3-7.

**OUTCOME ASSESSMENT QUESTIONS**

1. Draw the circuit diagram that implements the expression  $x = \bar{A}BC(\bar{A} + \bar{D})$  using gates with no more than three inputs.
2. Draw the circuit diagram for the expression  $y = AC + B\bar{C} + \bar{A}BC$ .
3. Draw the circuit diagram for  $x = [D + (\bar{A} + B)\bar{C}] \cdot E$ .

**3-9 NOR GATES AND NAND GATES****OUTCOMES**

Upon completion of this section, you will be able to:

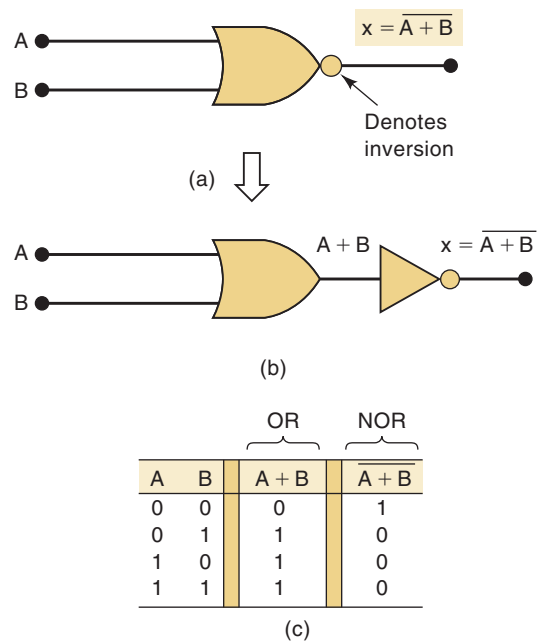
- Define the NOR and NAND logic functions.
- Write Boolean equations using the NOR and NAND function.
- Draw the logic symbol for the NOR and NAND function.
- Write a truth table describing the NOR and NAND function.
- Draw a timing diagram that demonstrates the NOR and NAND function.
- Use any of the above methods to infer the correct output of a logic circuit based on its input.

Two other types of logic gates, NOR gates and NAND gates, are widely used in digital circuits. These gates actually combine the basic AND, OR, and NOT operations, so it is a relatively simple matter to write their Boolean expressions.

**NOR Gate**

The symbol for a two-input **NOR gate** is shown in Figure 3-19(a). It is the same as the OR gate symbol except that it has a small circle on the output. The small circle represents the inversion operation. Thus, the NOR gate

**FIGURE 3-19** (a) NOR symbol; (b) equivalent circuit; (c) truth table.



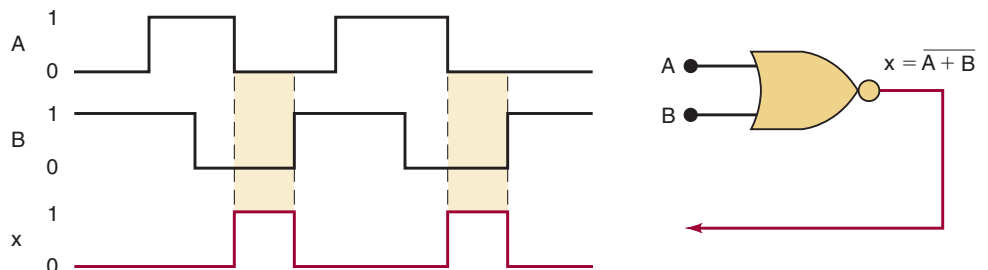
operates like an OR gate followed by an INVERTER, so that the circuits in Figures 3-19(a) and (b) are equivalent, and the output expression for the NOR gate is  $x = \overline{A + B}$ .

The truth table in Figure 3-19(c) shows that the NOR gate output is the exact inverse of the OR gate output for all possible input conditions. An OR gate output goes HIGH when any input is HIGH; the NOR gate output goes LOW when any input is HIGH. This same operation can be extended to NOR gates with more than two inputs.

### EXAMPLE 3-8

Determine the waveform at the output of a NOR gate for the input waveforms shown in Figure 3-20.

**FIGURE 3-20** Example 3-8.



### Solution

One way to determine the NOR output waveform is to find first the OR output waveform and then invert it (change all 1s to 0s, and vice versa). Another way utilizes the fact that a NOR gate output will be HIGH *only* when all inputs are LOW. Thus, you can examine the input waveforms, find those time intervals where they are all LOW, and make the NOR output HIGH for those intervals. The NOR output will be LOW for all other time intervals. The resultant output waveform is shown in the figure.

**EXAMPLE 3-9**

Determine the Boolean expression for a three-input NOR gate followed by an INVERTER.

**Solution**

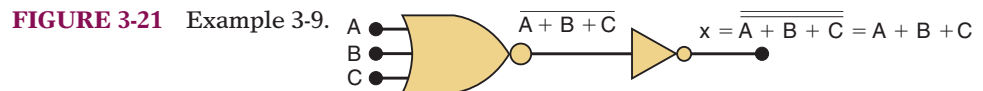
Refer to Figure 3-21, where the circuit diagram is shown. The expression at the NOR output is  $(\overline{A + B + C})$ , which is then fed through an INVERTER to produce

$$x = \overline{\overline{A + B + C}}$$

The presence of the double inversion signs indicates that the quantity  $(A + B + C)$  has been inverted and then inverted again. It should be clear that this simply results in the expression  $(A + B + C)$  being unchanged. That is,

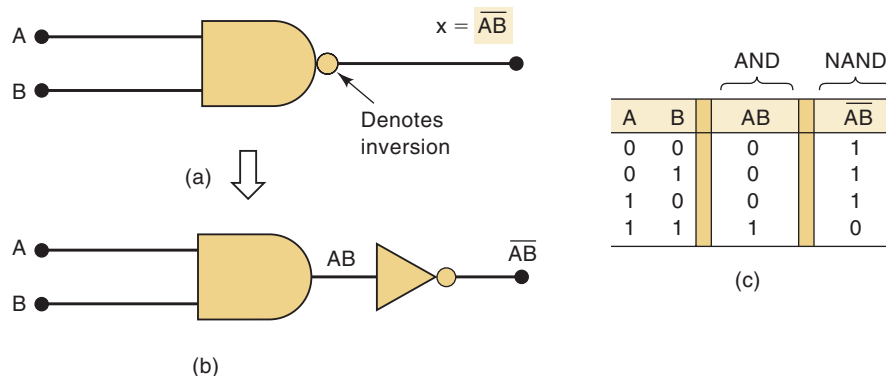
$$x = \overline{\overline{A + B + C}} = (A + B + C)$$

Whenever two inversion bars are over the same variable or quantity, they cancel each other out, as in the example above. However, in cases such as  $\overline{A + B}$  the inversion bars do not cancel. This is because the smaller inversion bars invert the single variables  $A$  and  $B$ , while the wide bar inverts the quantity  $(\overline{A + B})$ . Thus,  $\overline{A + B} \neq A + B$ . Similarly,  $\overline{A B} \neq AB$ .

**NAND Gate**

The symbol for a two-input **NAND** gate is shown in Figure 3-22(a). It is the same as the AND gate symbol except for the small circle on its output. Once again, this small circle denotes the inversion operation. Thus, the NAND operates like an AND gate followed by an INVERTER, so that the circuits of Figures 3-22(a) and (b) are equivalent, and the output expression for the NAND gate is  $x = \overline{AB}$ .

**FIGURE 3-22** (a) NAND symbol; (b) equivalent circuit; (c) truth table.

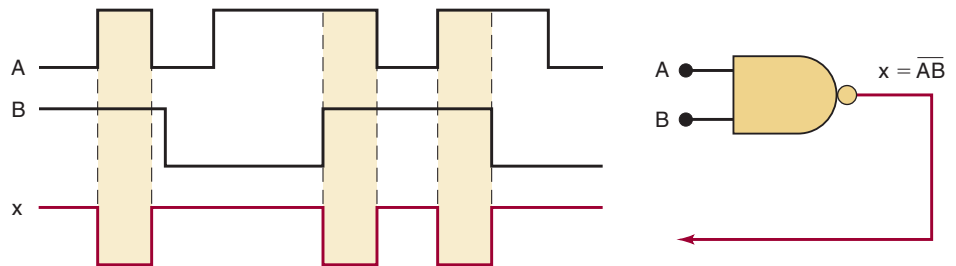


The truth table in Figure 3-22(c) shows that the NAND gate output is the exact inverse of the AND gate for all possible input conditions. The AND output goes HIGH only when all inputs are HIGH, while the NAND output goes LOW only when all inputs are HIGH. This same characteristic is true of NAND gates having more than two inputs.

## EXAMPLE 3-10

Determine the output waveform of a NAND gate having the input waveforms shown in Figure 3-23.

FIGURE 3-23  
Example 3-10.



## Solution

One way is to draw first the output waveform for an AND gate and then invert it. Another way utilizes the fact that a NAND output will be LOW only when all inputs are HIGH. Thus, you can find those time intervals during which the inputs are all HIGH, and make the NAND output LOW for those intervals. The output will be HIGH at all other times.

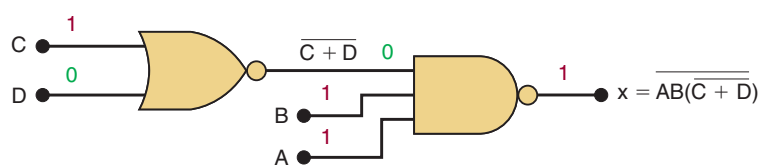
## EXAMPLE 3-11

Implement the logic circuit that has the expression  $x = \overline{AB \cdot (\overline{C + D})}$  using only NOR and NAND gates.

## Solution

The  $(\overline{C + D})$  term is the expression for the output of a NOR gate. This term is ANDed with  $A$  and  $B$ , and the result is inverted; this, of course, is the NAND operation. Thus, the circuit is implemented as shown in Figure 3-24. Note that the NAND gate first ANDs the  $A$ ,  $B$ , and  $(\overline{C + D})$  terms, and then it inverts the *complete* result.

FIGURE 3-24 Examples  
3-11 and 3-12.



## EXAMPLE 3-12

Determine the output level in Figure 3-24 for  $A = B = C = 1$  and  $D = 0$ .

## Solution

In the first method, we use the expression for  $x$ .

$$\begin{aligned} x &= \overline{AB(\overline{C + D})} \\ &= \overline{1 \cdot 1 \cdot (\overline{1 + 0})} \\ &= \overline{1 \cdot 1 \cdot (\overline{1})} \\ &= \overline{1 \cdot 1 \cdot 0} \\ &= \overline{0} = 1 \end{aligned}$$

In the second method, we write down the input logic levels on the circuit diagram (shown in color in Figure 3-24) and follow these levels through each



gate to the final output. The NOR gate has inputs of 1 and 0 to produce an output of 0 (an OR would have produced an output of 1). The NAND gate thus has input levels of 0, 1, and 1 to produce an output of 1 (an AND would have produced an output of 0).

### OUTCOME ASSESSMENT QUESTIONS

1. What is the only set of input conditions that will produce a HIGH output from a three-input NOR gate?
2. Determine the output level in Figure 3-24 for  $A = B = 1, C = D = 0$ .
3. Change the NOR gate of Figure 3-24 to a NAND gate, and change the NAND to a NOR. What is the new expression for  $x$ ?

## 3-10 BOOLEAN THEOREMS

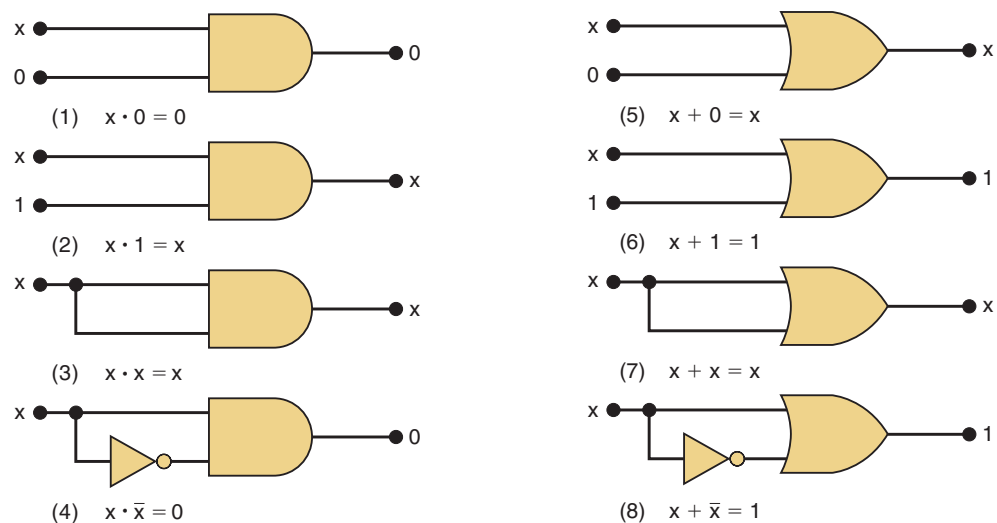
### OUTCOMES

Upon completion of this section, you will be able to:

- Correlate common algebraic theorems to Boolean algebra.
- Employ theorems of Boolean algebra to simplify expressions.
- Prove equivalence of two expressions by generating and comparing truth tables.

We have seen how Boolean algebra can be used to help analyze a logic circuit and express its operation mathematically. We will continue our study of Boolean algebra by investigating the various **Boolean theorems** (rules) that can help us to simplify logic expressions and logic circuits. The first group of theorems is given in Figure 3-25. In each theorem,  $x$  is a logic variable that can be either a 0 or a 1. Each theorem is accompanied by a logic-circuit diagram that demonstrates its validity.

Theorem (1) states that if any variable is ANDed with 0, the result must be 0. This is easy to remember because the AND operation is just like ordinary multiplication, where we know that anything multiplied by 0 is 0. We



**FIGURE 3-25** Single-variable theorems.

also know that the output of an AND gate will be 0 whenever any input is 0, regardless of the level on the other input.

Theorem (2) is also obvious by comparison with ordinary multiplication.

Theorem (3) can be proved by trying each case. If  $x = 0$ , then  $0 \cdot 0 = 0$ ; if  $x = 1$ , then  $1 \cdot 1 = 1$ . Thus,  $x \cdot x = x$ .

Theorem (4) can be proved in the same manner. However, it can also be reasoned that at any time either  $x$  or its inverse  $\bar{x}$  must be at the 0 level, and so their AND product always must be 0.

Theorem (5) is straightforward, since 0 added to anything does not affect its value, either in regular addition or in OR addition.

Theorem (6) states that if any variable is ORed with 1, the result will always be 1. We check this for both values of  $x$ :  $0 + 1 = 1$  and  $1 + 1 = 1$ . Equivalently, we can remember that an OR gate output will be 1 when *any* input is 1, regardless of the value of the other input.

Theorem (7) can be proved by checking for both values of  $x$ :  $0 + 0 = 0$  and  $1 + 1 = 1$ .

Theorem (8) can be proved similarly, or we can just reason that at any time either  $x$  or  $\bar{x}$  must be at the 1 level so that we are always ORing a 0 and a 1, which always results in 1.

Before introducing any more theorems, we should point out that when theorems (1) through (8) are applied, the variable  $x$  may actually represent an expression containing more than one variable. For example, if we have  $\overline{AB}(AB)$ , we can invoke theorem (4) by letting  $x = \overline{AB}$ . Thus, we can say that  $\overline{AB}(AB) = 0$ . The same idea can be applied to the use of any of these theorems.

## Multivariable Theorems

The theorems presented below involve more than one variable:

- (9)  $x + y = y + x$
- (10)  $x \cdot y = y \cdot x$
- (11)  $x + (y + z) = (x + y) + z = x + y + z$
- (12)  $x(yz) = (xy)z = xyz$
- (13a)  $x(y + z) = xy + xz$
- (13b)  $(w + x)(y + z) = wy + xy + wz + xz$
- (14)  $x + xy = x$
- (15a)  $x + \bar{x}y = x + y$
- (15b)  $\bar{x} + xy = \bar{x} + y$

Theorems (9) and (10) are called the *commutative laws*. These laws indicate that the order in which we OR or AND two variables is unimportant; the result is the same.

Theorems (11) and (12) are the *associative laws*, which state that we can group the variables in an AND expression or OR expression any way we want.

Theorem (13) is the *distributive law*, which states that an expression can be expanded by multiplying term by term just the same as in ordinary algebra. This theorem also indicates that we can factor an expression. That is, if we have a sum of two (or more) terms, each of which contains a common variable, the common variable can be factored out just as in ordinary algebra. For example, if we have the expression  $\overline{A}BC + \overline{A}\overline{B}\overline{C}$ , we can factor out the  $\overline{A}$  variable:

$$\overline{A}BC + \overline{A}\overline{B}\overline{C} = \overline{A}(BC + \overline{B}\overline{C})$$

As another example, consider the expression  $ABC + ABD$ . Here the two terms have the variables  $A$  and  $B$  in common, and so  $A \cdot B$  can be factored out of both terms. That is,

$$ABC + ABD = AB(C + D)$$

Theorems (9) to (13) are easy to remember and use because they are identical to those of ordinary algebra. Theorems (14) and (15), on the other hand, do not have any counterparts in ordinary algebra. Each can be proved by trying all possible cases for  $x$  and  $y$ . This is illustrated (for theorem 14) by creating an analysis table for the equation  $x + xy$  as follows:

x	y	xy	x + xy
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Notice that the value of the entire expression ( $x + xy$ ) is always the same as  $x$ .

Theorem (14) can also be proved by factoring and using theorems (6) and (2) as follows:

$$\begin{aligned} x + xy &= x(1 + y) \\ &= x \cdot 1 && \text{[using theorem (6)]} \\ &= x && \text{[using theorem (2)]} \end{aligned}$$

All of these Boolean theorems can be useful in simplifying a logic expression—that is, in reducing the number of terms in the expression. When this is done, the reduced expression will produce a circuit that is less complex than the one that the original expression would have produced. A good portion of the next chapter will be devoted to the process of circuit simplification. For now, the following examples will serve to illustrate how the Boolean theorems can be applied. **Note:** You can find all the Boolean theorems on the inside back cover.

### EXAMPLE 3-13

Simplify the expression  $y = A\bar{B}D + A\bar{B}\bar{D}$ .

#### Solution

Factor out the common variables  $A\bar{B}$  using theorem (13):

$$y = A\bar{B}(D + \bar{D})$$

Using theorem (8), the term in parentheses is equivalent to 1. Thus,

$$\begin{aligned} y &= A\bar{B} \cdot 1 \\ &= A\bar{B} && \text{[using theorem (2)]} \end{aligned}$$

**EXAMPLE 3-14**

Simplify  $z = (\bar{A} + B)(A + B)$ .

**Solution**

The expression can be expanded by multiplying out the terms [theorem (13)]:

$$z = \bar{A} \cdot A + \bar{A} \cdot B + B \cdot A + B \cdot B$$

Invoking theorem (4), the term  $\bar{A} \cdot A = 0$ . Also,  $B \cdot B = B$  [theorem (3)]:

$$z = 0 + \bar{A} \cdot B + B \cdot A + B = \bar{A}B + AB + B$$

Factoring out the variable  $B$  [theorem (13)], we have

$$z = B(\bar{A} + A + 1)$$

Finally, using theorems (2) and (6),

$$z = B$$

**EXAMPLE 3-15**

Simplify  $x = ACD + \bar{A}BCD$ .

**Solution**

Factoring out the common variables  $CD$ , we have

$$x = CD(A + \bar{A}B)$$

Utilizing theorem (15a), we can replace  $A + \bar{A}B$  by  $A + B$ , so

$$\begin{aligned} x &= CD(A + B) \\ &= ACD + BCD \end{aligned}$$

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Use theorems (13) and (14) to simplify  $y = A\bar{C} + ABC\bar{C}$ .
2. Use theorems (13) and (8) to simplify  $y = \bar{A}BC\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D}$ .
3. Use theorems (13) and (15b) to simplify  $y = \bar{A}D + ABD$ .

**3-11 DEMORGAN'S THEOREMS****OUTCOMES**

Upon completion of this section, you will be able to:

- Express DeMorgan's theorem algebraically.
- Use DeMorgan's theorem to simplify algebraic expressions.
- Draw DeMorgan equivalent circuits.

Two of the most important theorems of Boolean algebra were contributed by a great mathematician named DeMorgan. **DeMorgan's theorems** are

extremely useful in simplifying expressions in which a product or sum of variables is inverted. The two theorems are:

$$(16) \quad \overline{(x + y)} = \bar{x} \cdot \bar{y}$$

$$(17) \quad \overline{(x \cdot y)} = \bar{x} + \bar{y}$$

Theorem (16) says that when the OR sum of two variables is inverted, this is the same as inverting each variable individually and then ANDing these inverted variables. Theorem (17) says that when the AND product of two variables is inverted, this is the same as inverting each variable individually and then ORing them. Each of DeMorgan's theorems can readily be proven by checking for all possible combinations of  $x$  and  $y$ . This will be left as an end-of-chapter exercise.

Although these theorems have been stated in terms of single variables  $x$  and  $y$ , they are equally valid for situations where  $x$  and/or  $y$  are expressions that contain more than one variable. For example, let's apply them to the expression  $\overline{(A\bar{B} + C)}$  as shown below:

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

Note that we used theorem (16) and treated  $A\bar{B}$  as  $x$  and  $C$  as  $y$ . The result can be further simplified because we have a product  $A\bar{B}$  that is inverted. Using theorem (17), the expression becomes

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

Notice that we can replace  $\bar{\bar{B}}$  by  $B$ , so that we finally have

$$(\bar{A} + B) \cdot \bar{C} = \bar{A}\bar{C} + B\bar{C}$$

This final result contains only inverter signs that invert a single variable.

### EXAMPLE 3-16

Simplify the expression  $z = \overline{(\bar{A} + C)} \cdot \overline{(B + \bar{D})}$  to one having only single variables inverted.

#### Solution

Using theorem (17), and treating  $(\bar{A} + C)$  as  $x$  and  $(B + \bar{D})$  as  $y$ , we have

$$z = \overline{(\bar{A} + C)} + \overline{(B + \bar{D})}$$

We can think of this as breaking the large inverter sign down the middle and changing the AND sign ( $\cdot$ ) to an OR sign ( $+$ ). Now the term  $\overline{(\bar{A} + C)}$  can be simplified by applying theorem (16). Likewise,  $\overline{(B + \bar{D})}$  can be simplified:

$$\begin{aligned} z &= \overline{(\bar{A} + C)} + \overline{(B + \bar{D})} \\ &= (\bar{\bar{A}} \cdot \bar{C}) + \bar{B} \cdot \bar{\bar{D}} \end{aligned}$$

Here we have broken the larger inverter signs down the middle and replaced the ( $+$ ) with a ( $\cdot$ ). Canceling out the double inversions, we have finally

$$z = A\bar{C} + \bar{B}D$$

Example 3-16 points out that when using DeMorgan's theorems to reduce an expression, we may break an inverter sign at any point in the expression and change the operator sign at that point in the expression to its opposite (+ is changed to ·, and vice versa). This procedure is continued until the expression is reduced to one in which only single variables are inverted. Two more examples are given below.

**Example 1**

$$\begin{aligned} z &= \overline{A + \overline{B} \cdot C} \\ &= \overline{A} \cdot \overline{\overline{B} \cdot C} \\ &= \overline{A} \cdot (\overline{\overline{B}} + \overline{C}) \\ &= \overline{A} \cdot (B + \overline{C}) \end{aligned}$$

**Example 2**

$$\begin{aligned} \omega &= \overline{(A + BC) \cdot (D + EF)} \\ &= \overline{(A + BC)} + \overline{(D + EF)} \\ &= (\overline{A} \cdot \overline{BC}) + (\overline{D} \cdot \overline{EF}) \\ &= [\overline{A} \cdot (\overline{B} + \overline{C})] + [\overline{D} \cdot (\overline{E} + \overline{F})] \\ &= \overline{A}\overline{B} + \overline{A}\overline{C} + \overline{D}\overline{E} + \overline{D}\overline{F} \end{aligned}$$

DeMorgan's theorems are easily extended to more than two variables. For example, it can be proved that

$$\begin{aligned} \overline{x + y + z} &= \overline{x} \cdot \overline{y} \cdot \overline{z} \\ \overline{x \cdot y \cdot z} &= \overline{x} + \overline{y} + \overline{z} \end{aligned}$$

Here, we see that the large inverter sign is broken at *two* points in the expression and the operator sign is changed to its opposite. This can be extended to any number of variables. Again, realize that the variables can themselves be expressions rather than single variables. Here is another example.

$$\begin{aligned} x &= \overline{\overline{AB} \cdot \overline{CD} \cdot \overline{EF}} \\ &= \overline{\overline{AB}} + \overline{\overline{CD}} + \overline{\overline{EF}} \\ &= AB + CD + EF \end{aligned}$$

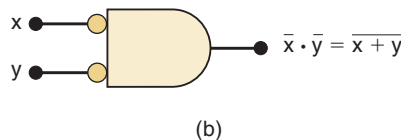
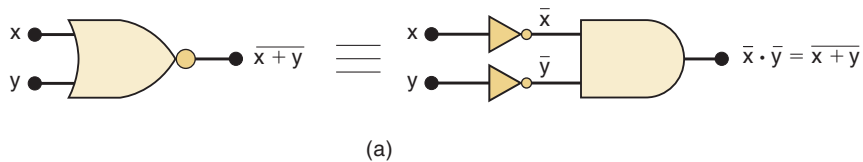
**Implications of DeMorgan's Theorems**

Let us examine theorems (16) and (17) from the standpoint of logic circuits. First, consider theorem (16):

$$\overline{x + y} = \overline{x} \cdot \overline{y}$$

The left-hand side of the equation can be viewed as the output of a NOR gate whose inputs are  $x$  and  $y$ . The right-hand side of the equation, on the other hand, is the result of first inverting both  $x$  and  $y$  and then putting them through an AND gate. These two representations are equivalent and are illustrated in Figure 3-26(a). What this means is that an AND gate with INVERTERS on each of its inputs is equivalent to a NOR gate. In fact, both

**FIGURE 3-26**  
(a) Equivalent circuits implied by theorem (16);  
(b) alternative symbol for the NOR function.



representations are used to represent the NOR function. When the AND gate with inverted inputs is used to represent the NOR function, it is usually drawn as shown in Figure 3-26(b), where the small circles on the inputs represent the inversion operation.

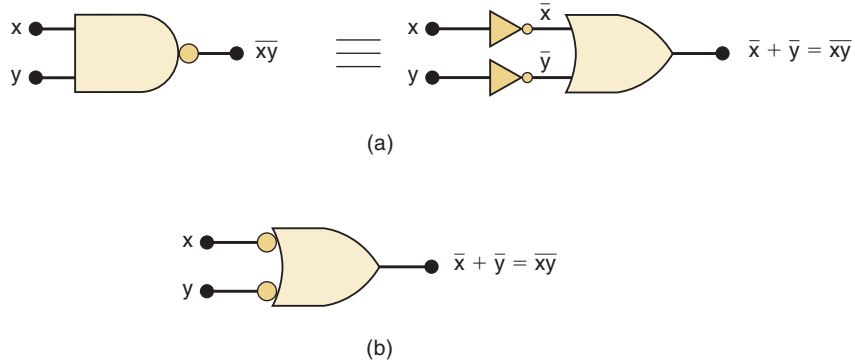
Now consider theorem (17):

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

The left side of the equation can be implemented by a NAND gate with inputs  $x$  and  $y$ . The right side can be implemented by first inverting inputs  $x$  and  $y$  and then putting them through an OR gate. These two equivalent representations are shown in Figure 3-27(a). The OR gate with INVERTERS on each of its inputs is equivalent to the NAND gate. In fact, both representations are used to represent the NAND function. When the OR gate with inverted inputs is used to represent the NAND function, it is usually drawn as shown in Figure 3-27(b), where the circles again represent inversion.

**FIGURE 3-27**

(a) Equivalent circuits implied by theorem (17);  
 (b) alternative symbol for the NAND function.

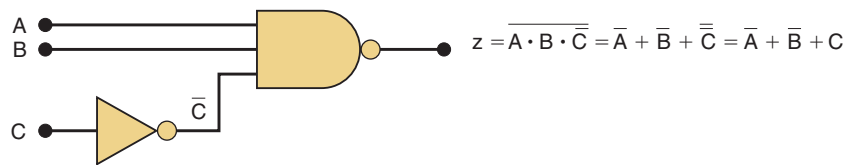


**EXAMPLE 3-17**

Determine the output expression for the circuit of Figure 3-28 and simplify it using DeMorgan's theorems.

**FIGURE 3-28**

Example 3-17.



**Solution**

The expression for  $z$  is  $z = \overline{ABC}$ . Use DeMorgan's theorem to break the large inversion sign:

$$z = \bar{A} + \bar{B} + \bar{\bar{C}}$$

Cancel the double inversions over  $C$  to obtain

$$z = \bar{A} + \bar{B} + C$$

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Use DeMorgan's theorems to convert the expression  $z = \overline{(A + B) \cdot \overline{C}}$  to one that has only single-variable inversions.
2. Repeat question 1 for the expression  $y = \overline{RST} + \overline{Q}$ .
3. Implement a circuit having output expression  $z = \overline{A} \overline{B} C$  using only a NOR gate and an INVERTER.
4. Use DeMorgan's theorems to convert  $y = \overline{A + \overline{B} + \overline{C} D}$  to an expression containing only single-variable inversions.

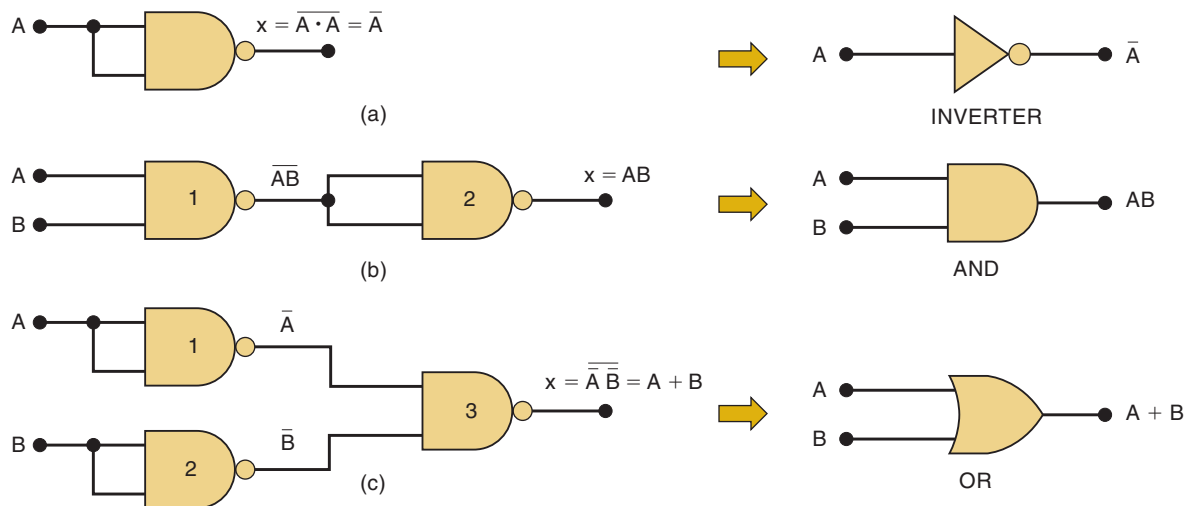
### 3-12 UNIVERSALITY OF NAND GATES AND NOR GATES

#### OUTCOMES

Upon completion of this section, you will be able to:

- Use NAND gates to create AND, OR, and NOT logic functions.
- Use NOR gates to create AND, OR, and NOT logic functions.
- Demonstrate reduction of total ICs using DIP package logic technology.

All Boolean expressions consist of various combinations of the basic operations of OR, AND, and INVERT. Therefore, any expression can be implemented using combinations of OR gates, AND gates, and INVERTERS. It is possible, however, to implement any logic expression using *only* NAND gates and no other type of gate. This is because NAND gates, in the proper combination, can be used to perform each of the Boolean operations OR, AND, and INVERT. This is demonstrated in Figure 3-29.



**FIGURE 3-29** NAND gates can be used to implement any Boolean function.

First, in Figure 3-29(a), we have a two-input NAND gate whose inputs are purposely connected together so that the variable  $A$  is applied to both. In this configuration, the NAND simply acts as INVERTER because its output is  $x = \overline{A \cdot A} = \overline{A}$ .

In Figure 3-29(b), we have two NAND gates connected so that the AND operation is performed. NAND gate 2 is used as an INVERTER to change  $\overline{AB}$  to  $\overline{\overline{AB}} = AB$ , which is the desired AND function.

The OR operation can be implemented using NAND gates connected as shown in Figure 3-29(c). Here NAND gates 1 and 2 are used as INVERTERS



to invert the inputs, so that the final output is  $x = \overline{\overline{A} \cdot \overline{B}}$ , which can be simplified to  $x = A + B$  using DeMorgan's theorem.

In a similar manner, it can be shown that NOR gates can be arranged to implement any of the Boolean operations. This is illustrated in Figure 3-30. Part (a) shows that a NOR gate with its inputs connected together behaves as an INVERTER because the output is  $x = \overline{A + A} = \overline{A}$ .

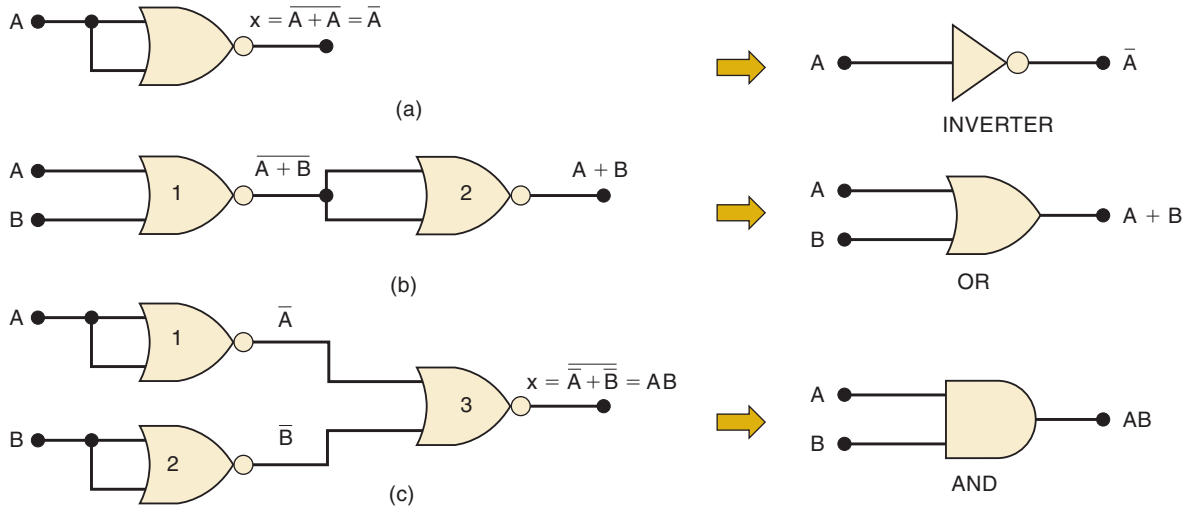


FIGURE 3-30 NOR gates can be used to implement any Boolean operation.

In Figure 3-30(b), two NOR gates are arranged so that the OR operation is performed. NOR gate 2 is used as an INVERTER to change  $\overline{A + B}$  to  $\overline{\overline{A + B}} = A + B$ , which is the desired OR function.

The AND operation can be implemented with NOR gates as shown in Figure 3-30(c). Here, NOR gates 1 and 2 are used as INVERTERS to invert the inputs, so that the final output is  $x = \overline{\overline{A} + \overline{B}}$ , which can be simplified to  $x = A \cdot B$  by use of DeMorgan's theorem.

Since any of the Boolean operations can be implemented using only NAND gates, any logic circuit can be constructed using only NAND gates. The same is true for NOR gates. This characteristic of NAND and NOR gates can be very useful in logic-circuit design, as Example 3-18 illustrates.

**EXAMPLE 3-18**

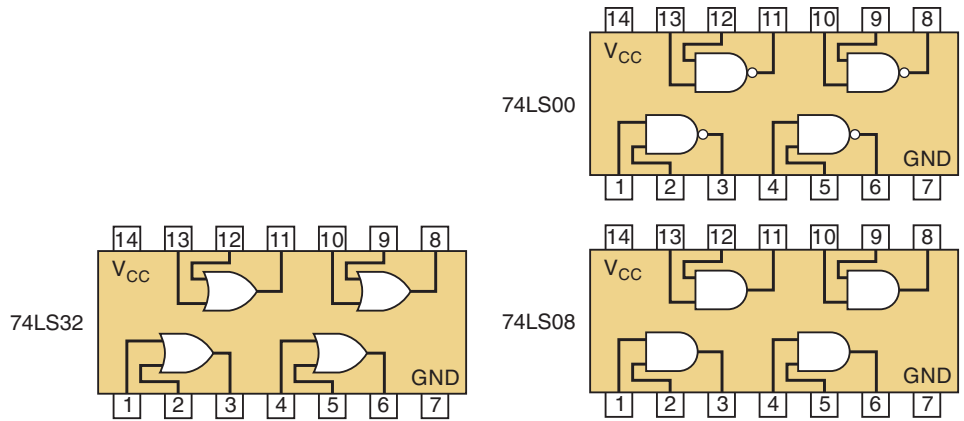
In a certain manufacturing process, a conveyor belt will shut down whenever specific conditions occur. These conditions are monitored and reflected by the states of four logic signals as follows: signal *A* will be HIGH whenever the conveyor belt speed is too fast; signal *B* will be HIGH whenever the collection bin at the end of the belt is full; signal *C* will be HIGH when the belt tension is too high; signal *D* will be HIGH when the manual override is off.

A logic circuit is needed to generate a signal *x* that will go HIGH whenever conditions *A* and *B* exist simultaneously or whenever conditions *C* and *D* exist simultaneously. Clearly, the logic expression for *x* will be  $x = AB + CD$ . The circuit is to be implemented with a minimum number of ICs. The TTL integrated circuits shown in Figure 3-31 are available. Each IC is a *quad*, which means that it contains *four* identical gates on one chip.

**Solution**

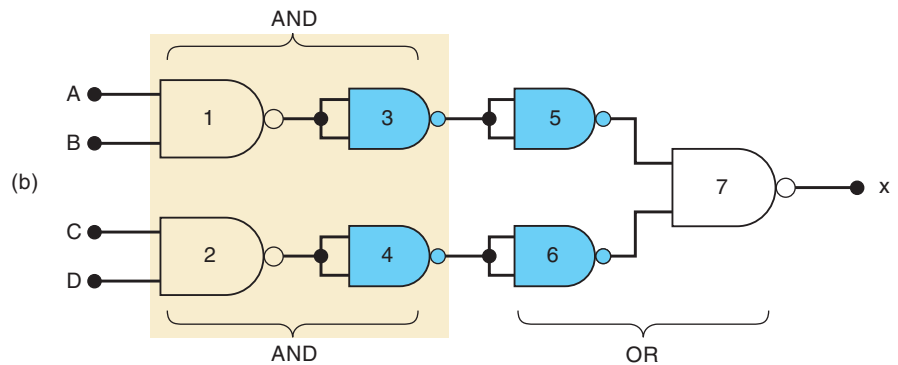
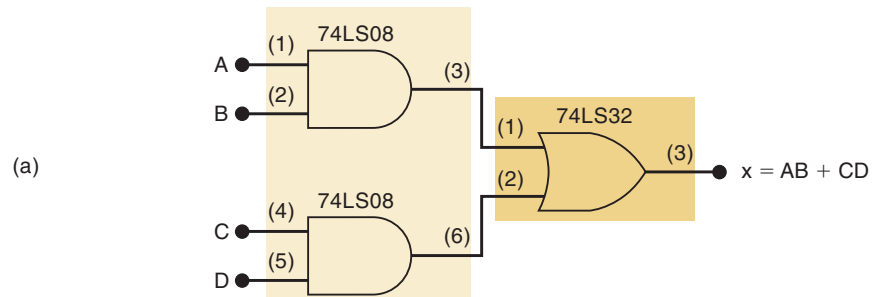
The straightforward method for implementing the given expression uses two AND gates and an OR gate, as shown in Figure 3-32(a). This implementation uses two gates from the 74LS08 IC and a single gate from the 74LS32 IC.

**FIGURE 3-31** ICs available for Examples 3-18.

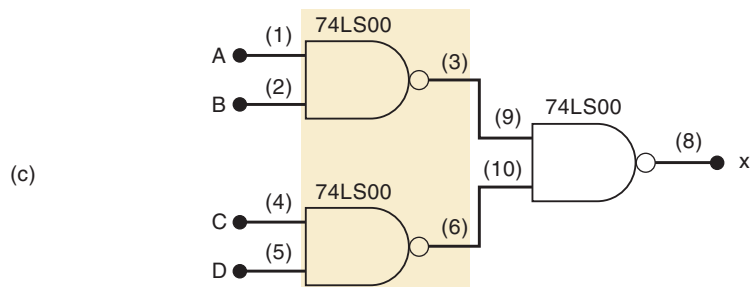


The numbers in parentheses at each input and output are the pin numbers of the respective IC. These are always shown on any logic-circuit wiring diagram. For our purposes, most logic diagrams will not show pin numbers unless they are needed in the description of circuit operation.

**FIGURE 3-32** Possible implementations for Example 3-18.



After eliminating double inversions



Another implementation can be accomplished by taking the circuit of Figure 3-32(a) and replacing each AND gate and OR gate by its equivalent NAND gate implementation from Figure 3-29. The result is shown in Figure 3-32(b).

At first glance, this new circuit looks as if it requires seven NAND gates. However, NAND gates 3 and 5 are connected as INVERTERS in series and can be eliminated from the circuit because they perform a double inversion of the signal out of NAND gate 1. Similarly, NAND gates 4 and 6 can be eliminated. The final circuit, after eliminating the double INVERTERS, is drawn in Figure 3-32(c).

This final circuit is more efficient than the one in Figure 3-32(a) because it uses three two-input NAND gates that can be implemented from one IC, the 74LS00.

### OUTCOME ASSESSMENT QUESTIONS

1. How many different ways do we now have to implement the inversion operation in a logic circuit?
2. Implement the expression  $x = (A + B)(C + D)$  using OR and AND gates. Then implement the expression using only NOR gates by converting each OR and AND gate to its NOR implementation from Figure 3-30. Which circuit is more efficient?
3. Write the output expression for the circuit of Figure 3-32(c), and use DeMorgan's theorems to show that it is equivalent to the expression for the circuit of Figure 3-32(a).
4. Refer Figure 3-32(a). If the  $D$  input needed to be inverted to produce  $x = AB + \overline{CD}$ , how many ICs would be needed?
5. Assuming the change described above, how many ICs would be needed using NAND only as in Figure 3-32(c)?

## 3-13 ALTERNATE LOGIC-GATE REPRESENTATIONS

### OUTCOMES

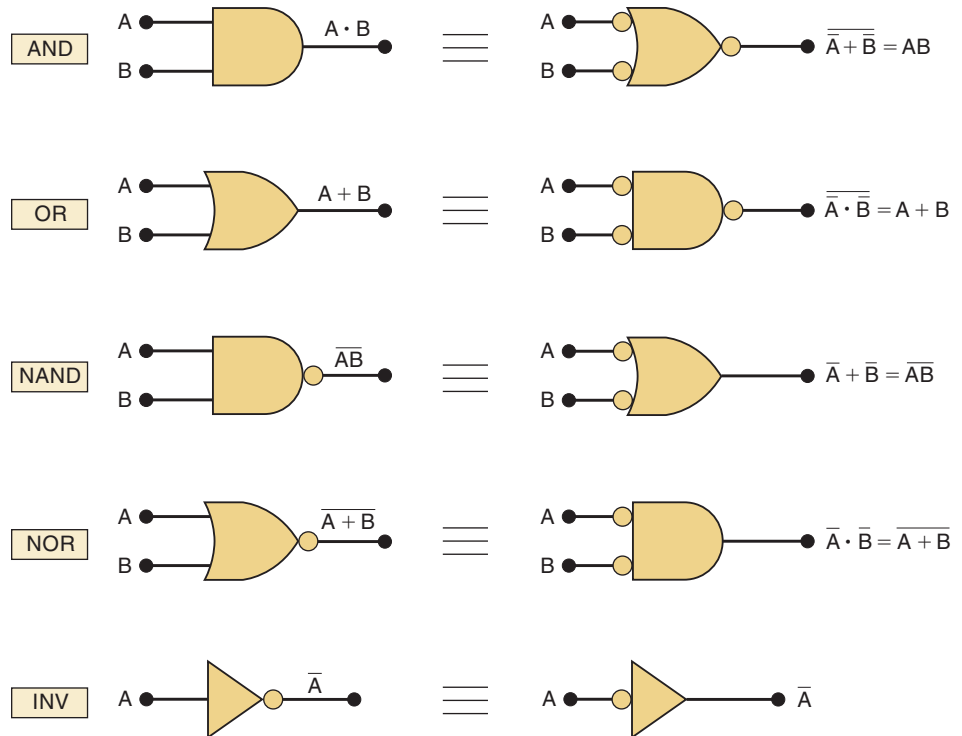
*Upon completion of this section, you will be able to:*

- Interpret logic diagrams based on active levels of the inputs.
- Represent logic functions in the most meaningful way using alternate symbols.

We have introduced the five basic logic gates (AND, OR, INVERTER, NAND, and NOR) and the standard symbols used to represent them on logic-circuit diagrams. Although you may find that some circuit diagrams still use these standard symbols exclusively, it has become increasingly more common to find circuit diagrams that utilize **alternate logic symbols** *in addition* to the standard symbols.

Before discussing the reasons for using an alternate symbol for a logic gate, we will present the alternate symbols for each gate and show that they are equivalent to the standard symbols. Refer to Figure 3-33; the left side of the illustration shows the standard symbol for each logic gate, and the

**FIGURE 3-33** Standard and alternative symbols for various logic gates and inverter.



right side shows the alternate symbol. The alternate symbol for each gate is obtained from the standard symbol by doing the following:

1. Invert each input and output of the standard symbol. This is done by adding bubbles (small circles) on input and output lines that do not have bubbles and by removing bubbles that are already there.
2. Change the operation symbol from AND to OR, or from OR to AND. (In the special case of the INVERTER, the operation symbol is not changed.)

For example, the standard NAND symbol is an AND symbol with a bubble on its output. Following the steps outlined above, remove the bubble from the output, and add a bubble to each input. Then change the AND symbol to an OR symbol. The result is an OR symbol with bubbles on its inputs.

We can easily prove that this alternate symbol is equivalent to the standard symbol by using DeMorgan's theorems and recalling that the bubble represents an inversion operation. The output expression from the standard NAND symbol is  $\overline{AB} = \overline{A} + \overline{B}$ , which is the same as the output expression for the alternate symbol. This same procedure can be followed for each pair of symbols in Figure 3-33.

Several points should be stressed regarding the logic symbol equivalences:

1. The equivalences can be extended to gates with *any* number of inputs.
2. None of the standard symbols have bubbles on their inputs, and all the alternate symbols do.
3. The standard and alternate symbols for each gate represent the same physical circuit; *there is no difference in the circuits represented by the two symbols.*
4. NAND and NOR gates are inverting gates, and so both the standard and the alternate symbols for each will have a bubble on *either* the input or the output. AND and OR gates are *noninverting* gates, and so the alternate symbols for each will have bubbles on *both* inputs and output.

## Logic-Symbol Interpretation

Each of the logic-gate symbols of Figure 3-33 provides a unique interpretation of how the gate operates. Before we can demonstrate these interpretations, we must first establish the concept of **active logic levels**.

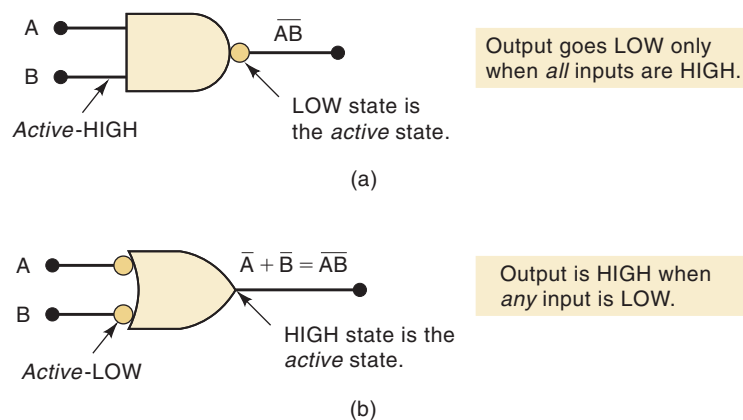
When an input or output line on a logic circuit symbol has *no bubble* on it, that line is said to be **active-HIGH**. When an input or output line *does* have a *bubble* on it, that line is said to be **active-LOW**. The presence or absence of a bubble, then, determines the active-HIGH/active-LOW status of a circuit's inputs and output, and is used to interpret the circuit operation.

To illustrate, Figure 3-34(a) shows the standard symbol for a NAND gate. The standard symbol has a bubble on its output and no bubbles on its inputs. Thus, it has an active-LOW output and active-HIGH inputs. The logic operation represented by this symbol can therefore be interpreted as follows:

**The output goes LOW only when *all* of the inputs are HIGH.**

Note that this says that the output will go to its active state only when *all* of the inputs are in their active states. The word *all* is used because of the AND symbol.

**FIGURE 3-34**  
Interpretation of the two  
NAND gate symbols.



The alternate symbol for a NAND gate shown in Figure 3-34(b) has an active-HIGH output and active-LOW inputs, and so its operation can be stated as follows:

**The output goes HIGH when *any* input is LOW.**

This says that the output will be in its active state whenever *any* of the inputs is in its active state. The word *any* is used because of the OR symbol.

With a little thought, you can see that the two interpretations for the NAND symbols in Figure 3-34 are different ways of saying the same thing.

## Summary

At this point you are probably wondering why there is a need to have two different symbols and interpretations for each logic gate. We hope the reasons will become clear after reading the next section. For now, let us summarize the important points concerning the logic-gate representations.

1. To obtain the alternate symbol for a logic gate, take the standard symbol and change its operation symbol (OR to AND, or AND to OR), and change the bubbles on both inputs and output (i.e., delete bubbles that are present, and add bubbles where there are none).

- To interpret the logic-gate operation, first note which logic state, 0 or 1, is the active state for the inputs and which is the active state for the output. Then realize that the output's active state is produced by having *all* of the inputs in their active state (if an AND symbol is used) or by having *any* of the inputs in its active state (if an OR symbol is used).

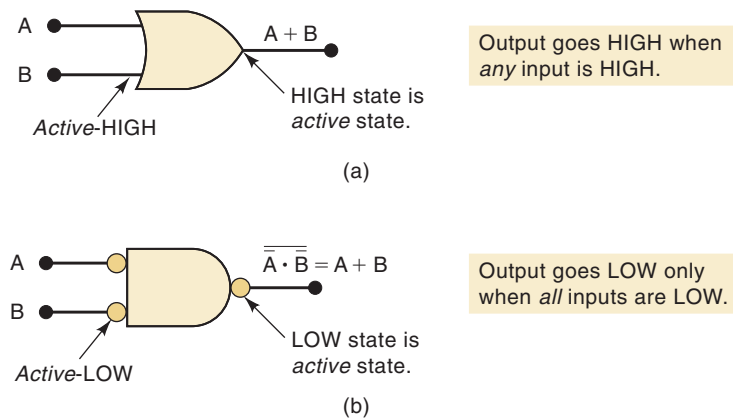
**EXAMPLE 3-19**

Give the interpretation of the two OR gate symbols.

**Solution**

The results are shown in Figure 3-35. Note that the word *any* is used when the operation symbol is an OR symbol and the word *all* is used when it includes an AND symbol.

**FIGURE 3-35**  
Interpretation of the two OR gate symbols.

**OUTCOME ASSESSMENT QUESTIONS**

- Write the interpretation of the operation performed by the standard NOR gate symbol in Figure 3-33.
- Repeat question 1 for the alternate NOR gate symbol.
- Repeat question 1 for the alternate AND gate symbol.
- Repeat question 1 for the standard AND gate symbol.

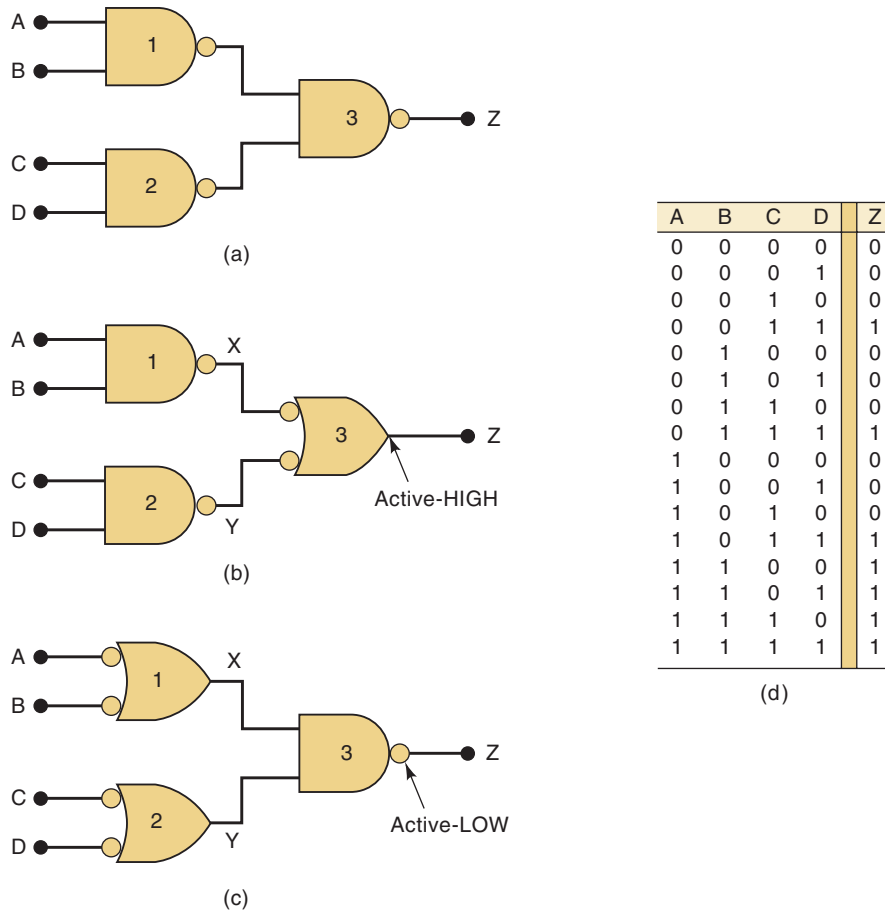
**3-14 WHICH GATE REPRESENTATION TO USE****OUTCOMES**

Upon completion of this section, you will be able to:

- Choose the most descriptive logic symbols.
- Quickly analyze circuit operation based on the diagram.

Some logic-circuit designers and some textbooks use only the standard logic-gate symbols in their circuit schematics. While this practice is logically correct, it does nothing to make the circuit operation easier to follow. Proper use of the alternate gate symbols in the circuit diagram can make the circuit operation much clearer. This can be illustrated by considering the example shown in Figure 3-36.

**FIGURE 3-36** (a) Original circuit using standard NAND symbols; (b) equivalent representation where output Z is active-HIGH; (c) equivalent representation where output Z is active-LOW; (d) truth table.



The circuit in Figure 3-36(a) contains three NAND gates connected to produce an output Z that depends on inputs A, B, C, and D. The circuit diagram uses the standard symbol for each of the NAND gates. While this diagram is logically correct, it does not facilitate an understanding of how the circuit functions. The circuit representations given in Figures 3-36(b) and (c), however, can be analyzed more easily to determine the circuit operation.

The representation of Figure 3-36(b) is obtained from the original circuit diagram by replacing NAND gate 3 with its alternate symbol. In this diagram, output Z is taken from a NAND gate symbol that has an active-HIGH output. Thus, we can say that Z will go HIGH when either X or Y is LOW. Now, since X and Y each appear at the output of NAND symbols having active-LOW outputs, we can say that X will go LOW only if A = B = 1, and Y will go LOW only if C = D = 1. Putting this all together, we can describe the circuit operation as follows:

**Output Z will go HIGH whenever either  $A = B = 1$  or  $C = D = 1$  (or both).**

This description can be translated to truth-table form by setting  $Z = 1$  for those cases where  $A = B = 1$  and for those cases where  $C = D = 1$ . For all other cases, Z is made a 0. The resultant truth table is shown in Figure 3-36(d).

The representation of Figure 3-36(c) is obtained from the original circuit diagram by replacing NAND gates 1 and 2 by their alternate symbols. In this equivalent representation, the Z output is taken from a NAND gate that has an active-LOW output. Thus, we can say that Z will go LOW only when  $X = Y = 1$ . Because X and Y are active-HIGH outputs, we can say that

$X$  will be HIGH when either  $A$  or  $B$  is LOW, and  $Y$  will be HIGH when either  $C$  or  $D$  is LOW. Putting this all together, we can describe the circuit operation as follows:

**Output  $Z$  will go LOW only when  $A$  or  $B$  is LOW and  $C$  or  $D$  is LOW.**

This description can be translated to truth-table form by making  $Z = 0$  for all cases where at least one of the  $A$  or  $B$  inputs is LOW at the same time that at least one of the  $C$  or  $D$  inputs is LOW. For all other cases,  $Z$  is made a 1. The resultant truth table is the same as that obtained for the circuit diagram of Figure 3-36(b).

### Which Circuit Diagram Should Be Used?

The answer to this question depends on the particular function being performed by the circuit output. If the circuit is being used to cause some action (e.g., turn on an LED or activate another logic circuit) when output  $Z$  goes to the 1 state, then we say that  $Z$  is to be active-HIGH, and the circuit diagram of Figure 3-36(b) should be used. On the other hand, if the circuit is being used to cause some action when  $Z$  goes to the 0 state, then  $Z$  is to be active-LOW, and the diagram of Figure 3-36(c) should be used.

Of course, there will be situations where *both* output states are used to produce different actions and either one can be considered to be the active state. For these cases, either circuit representation can be used.

### Bubble Placement

Refer to the circuit representation of Figure 3-36(b) and note that the symbols for NAND gates 1 and 2 were chosen to have active-LOW outputs to match the active-LOW inputs of NAND gate 3. Refer to the circuit representation of Figure 3-36(c) and note that the symbols for NAND gates 1 and 2 were chosen to have active-HIGH outputs to match the active-HIGH inputs of NAND gate 3. This leads to the following general rule for preparing logic-circuit schematics:

**Whenever possible, choose gate symbols so that bubble outputs are connected to bubble inputs, and nonbubble outputs to nonbubble inputs.**

The following examples will show how this rule can be applied.

#### EXAMPLE 3-20

The logic circuit in Figure 3-37(a) is being used to activate an alarm when its output  $Z$  goes HIGH. Modify the circuit diagram so that it represents the circuit operation more effectively.

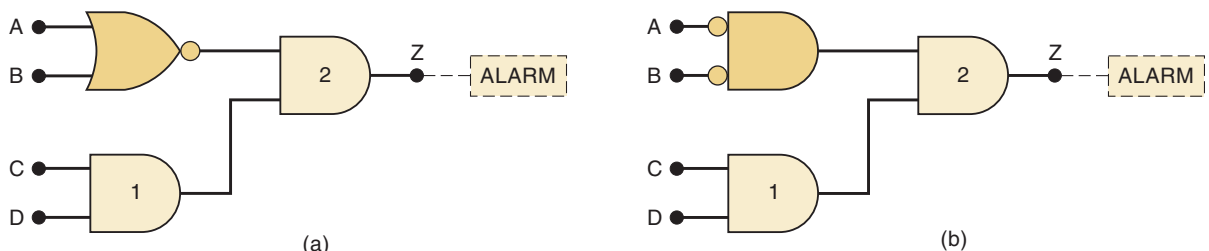


FIGURE 3-37 Example 3-20.



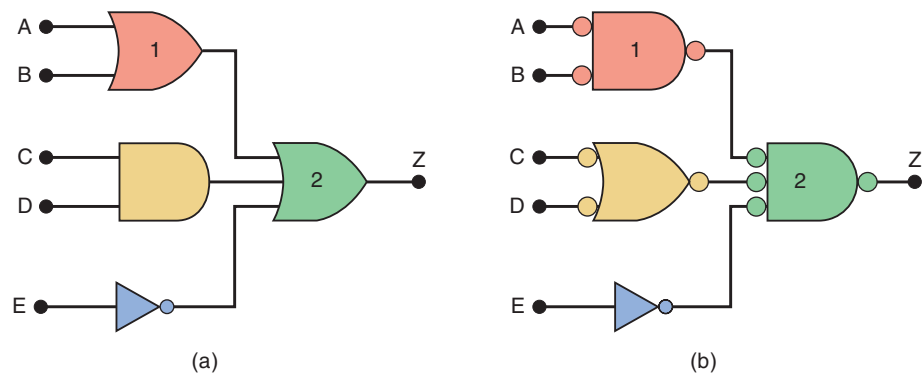
**Solution**

Because  $Z = 1$  will activate the alarm,  $Z$  is to be active-HIGH. Thus, the AND gate 2 symbol does not have to be changed. The NOR gate symbol should be changed to the alternate symbol with a nonbubble (active-HIGH) output to match the nonbubble input of AND gate 2, as shown in Figure 3-37(b). Note that the circuit now has nonbubble outputs connected to the nonbubble inputs of gate 2.

**EXAMPLE 3-21**

When the output of the logic circuit in Figure 3-38(a) goes LOW, it activates another logic circuit. Modify the circuit diagram to represent the circuit operation more effectively.

**FIGURE 3-38**  
Example 3-21.

**Solution**

Because  $Z$  is to be active-LOW, the symbol for OR gate 2 must be changed to its alternate symbol, as shown in Figure 3-38(b). The new OR gate 2 symbol has bubble inputs, and so the AND gate and OR gate 1 symbols must be changed to bubbled outputs, as shown in Figure 3-38(b). The INVERTER already has a bubble output. Now the circuit has all bubble outputs connected to bubble inputs of gate 2.

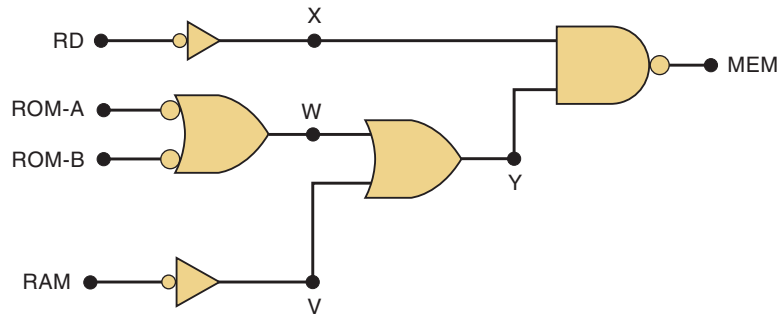
**Analyzing Circuits**

When a logic-circuit schematic is drawn using the rules we followed in these examples, it is much easier for an engineer or technician (or student) to follow the signal flow through the circuit and to determine the input conditions that are needed to activate the output. This will be illustrated in the following examples—which, incidentally, use circuit diagrams taken from the logic schematics of an actual microcomputer.

**EXAMPLE 3-22**

The logic circuit in Figure 3-39 generates an output, *MEM*, that is used to activate the memory ICs in a particular microcomputer. Determine the input conditions necessary to activate *MEM*.

**FIGURE 3-39**  
Example 3-22.



### Solution

One way to do this would be to write the expression for *MEM* in terms of the inputs *RD*, *ROM-A*, *ROM-B*, and *RAM*, and to evaluate it for the 16 possible combinations of these inputs. While this method would work, it would require a lot more work than is necessary.

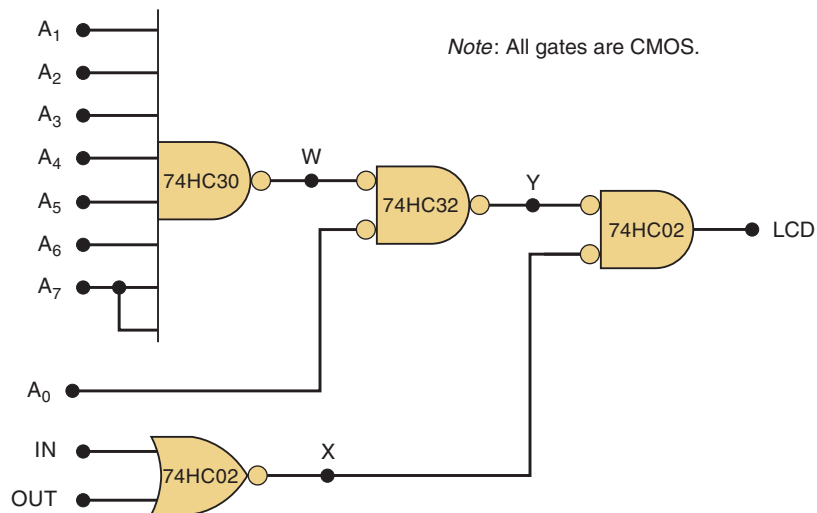
A more efficient method is to interpret the circuit diagram using the ideas we have been developing in the last two sections. These are the steps:

1. *MEM* is active-LOW, and it will go LOW only when *X* and *Y* are HIGH.
2. *X* will be HIGH only when *RD* = 0.
3. *Y* will be HIGH when either *W* or *V* is HIGH.
4. *V* will be HIGH when *RAM* = 0.
5. *W* will be HIGH when either *ROM-A* or *ROM-B* = 0.
6. Putting this all together, *MEM* will go LOW only when *RD* = 0 and at least one of the three inputs *ROM-A*, *ROM-B*, or *RAM* is LOW.

### EXAMPLE 3-23

The logic circuit in Figure 3-40 is used to enable the liquid crystal display (LCD) of a handheld electronic device when the microcontroller is sending data to or receiving data from the LCD controller. The circuit will enable the display when *LCD* = 1. Determine the input conditions necessary to enable the LCD.

**FIGURE 3-40**  
Example 3-23.



**Solution**

Once again, we will interpret the diagram in a step-by-step fashion:

1. *LCD* is active-HIGH, and it will go HIGH only when  $X = Y = 0$ .
2.  $X$  will be LOW when either *IN* or *OUT* is HIGH.
3.  $Y$  will be LOW only when  $W = 0$  and  $A_0 = 0$ .
4.  $W$  will be LOW only when  $A_1$  through  $A_7$  are all HIGH.
5. Putting this all together, *LCD* will be HIGH when  $A_1 = A_2 = A_3 = A_4 = A_5 = A_6 = A_7 = 1$  and  $A_0 = 0$ , and either *IN* or *OUT* or both are 1.

Note the strange symbol for the eight-input CMOS NAND gate (74HC30); also note that signal  $A_7$  is connected to two of the NAND inputs.

**Asserted Levels**

We have been describing logic signals as being active-LOW or active-HIGH. For example, the output *MEM* in Figure 3-39 is active-LOW, and the output *LCD* in Figure 3-40 is active-HIGH because these are the output states that cause something to happen. Similarly, Figure 3-40 has active-HIGH inputs  $A_1$  to  $A_7$  and active-LOW input  $A_0$ .

When a logic signal is in its active state, it can be said to be **asserted**. For example, when we say that input  $A_0$  is asserted, we are saying that it is in its active-LOW state. When a logic signal is not in its active state, it is said to be **unasserted**. Thus, when we say that *LCD* is unasserted, we mean that it is in its inactive state (low).

Clearly, the terms *asserted* and *unasserted* are synonymous with *active* and *inactive*, respectively:

$$\begin{aligned}\text{asserted} &= \text{active} \\ \text{unasserted} &= \text{inactive}\end{aligned}$$

Both sets of terms are in common use in the digital field, so you should recognize both ways of describing a logic signal's active state.

**Labeling Active-LOW Logic Signals**

It has become common practice to use an overbar to label active-LOW signals. The overbar serves as another indication that the signal is active-LOW; of course, the absence of an overbar means that the signal is active-HIGH.

To illustrate, all of the signals in Figure 3-39 are active-LOW, and so they can be labeled as follows:

$$\overline{RD}, \overline{ROM-A}, \overline{ROM-B}, \overline{RAM}, \overline{MEM}$$

Remember, the overbar is simply a way to emphasize that these are active-LOW signals. We will employ this convention for labeling logic signals whenever appropriate.

**Labeling Bistate Signals**

Very often, an output signal will have two active states; that is, it will have one important function in the HIGH state and another in the LOW state. It is customary to label such signals so that both active states are apparent. A common example is the read/write signal,  $\overline{RD}/\overline{WR}$ , which is interpreted

as follows: when this signal is HIGH, the read operation ( $RD$ ) is performed; when it is LOW, the write operation ( $\overline{WR}$ ) is performed.

### OUTCOME ASSESSMENT QUESTIONS

1. Use the method of Examples 3-22 and 3-23 to determine the input conditions needed to activate the output of the circuit in Figure 3-37(b).
2. Repeat question 1 for the circuit of Figure 3-38(b).
3. How many NAND gates are shown in Figure 3-39?
4. How many NOR gates are shown in Figure 3-40?
5. What will be the output level in Figure 3-38(b) when all of the inputs are asserted?
6. What inputs are required to assert the alarm output in Figure 3-37(b)?
7. Which of the following signals is active-LOW:  $RD$ ,  $\overline{W}$ ,  $R/\overline{W}$ ?

## 3-15 PROPAGATION DELAY

### OUTCOMES

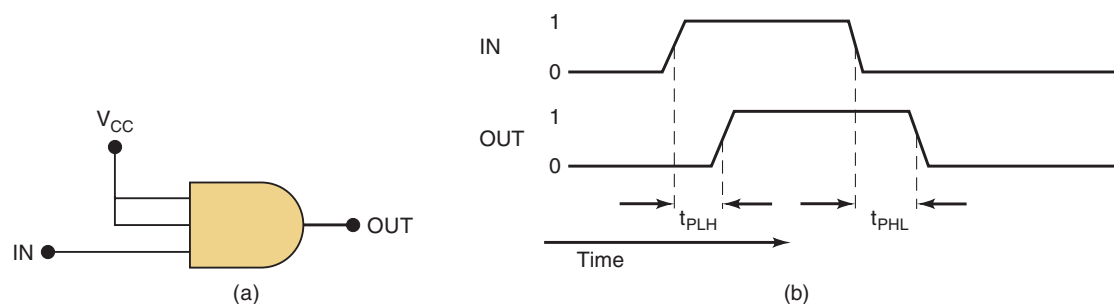
Upon completion of this section, you will be able to:

- Predict the affect of propagation delay.
- Measure actual propagation delay.
- Use standard parameter descriptors for propagation delay.

**Propagation delay** can be simply defined as the time it takes for a system to produce the appropriate output after it receives an input. Think about a typical vending machine. You put your money into the machine and press a button to make a selection. You do not receive the product immediately; it takes a little time for the product to be moved off the rack and dropped to the vending door. This is propagation delay. A biologic example can be found in our reflexes. From the time you see brake lights on the car in front of you until you get your foot on the brakes, there is a measurable delay or reaction time.

Real digital circuits have a measurable propagation delay time as well. The reasons will become clearer when you study the real characteristics of circuits and semiconductors (transistors) instead of just their idealized operation. Chapter 8 will provide more information about the inner workings of logic ICs. An AND gate, as shown in Figure 3-41(a), serves as an example that propagation delay does exist and that it can be measured.

When the IN signal goes HIGH, it causes the OUT signal to go HIGH a short time later. Likewise, when the IN signal goes LOW, it causes the OUT



**FIGURE 3-41** Measuring propagation delay in a logic gate.

signal to go LOW a short time later. Two things are important to note from the timing diagram in Figure 3-41(b):

1. Transitions are not truly vertical (instantaneous) so we measure from the 50% point on the input to the 50% point on the output.
2. The time it takes to make the *output* go HIGH is not necessarily the same as the time to make the *output* go LOW. These delay times are called  $t_{PLH}$  (time propagation LOW to HIGH) and  $t_{PHL}$  (time propagation HIGH to LOW).

The speed of a logic circuit is related to this characteristic of propagation delay. Whatever part is chosen to implement the logic circuit will have a data sheet that states the value of propagation delay. This information is used to assure that the circuit can operate fast enough for the application.

### OUTCOME ASSESSMENT QUESTIONS

1. Why are the transitions not vertical when measuring propagation delay?
2. Where are time measurements taken when transitions are not vertical?
3. What is the parameter that measures the time after the input changes until the output can switch from HIGH to LOW?
4. What is the parameter that measures the time after the input changes until the output can switch from LOW to HIGH?

## 3-16 SUMMARY OF METHODS TO DESCRIBE LOGIC CIRCUITS

### OUTCOME

*Upon completion of this section, you will be able to:*

- List the methods used to describe the operation of a logic circuit.

The topics we have covered so far in this chapter have all centered around just three simple logic functions that we refer to as AND, OR, and NOT. The concept is not new to anyone because we all use these logical functions every day as we make decisions. Here are some logical examples of how you might think: If it is raining OR the newspaper says that it could rain, then I will take my umbrella; if I get my paycheck today AND I make it to the bank, then I will have money to spend this evening; if I have a passing grade in lecture AND I have NOT failed in lab, then I will pass my digital class. At this point, you may be wondering why we have spent so much effort in describing such familiar concepts. The answer can be summed up in two key points:

1. We must be able to represent these logical decisions.
2. We must be able to combine these logic functions and implement a decision-making system.

We have learned how to represent each of the basic logic functions using:

- Logical statements in our own language
- Truth tables
- Traditional graphic logic symbols
- Boolean algebra expressions
- Timing diagrams

**EXAMPLE 3-24**

The following English expression describes the way a logic circuit needs to operate in order to drive a seatbelt warning indicator in a car.

**If the driver is present AND the driver is NOT buckled up AND the ignition switch is on, THEN turn on the warning light.**

Describe the circuit using Boolean algebra, schematic diagrams with logic symbols, truth tables, and timing diagrams.

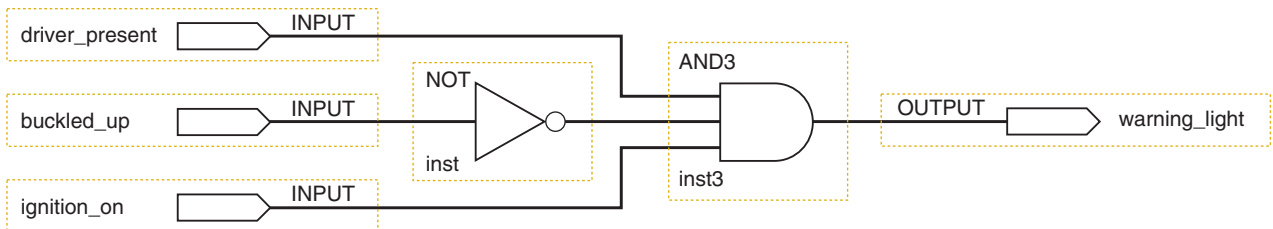
**Solution**

See Figure 3-42.

**Boolean expression**

$$\text{warning\_light} = \text{driver\_present} \cdot \overline{\text{buckled\_up}} \cdot \text{ignition\_on}$$

(a)

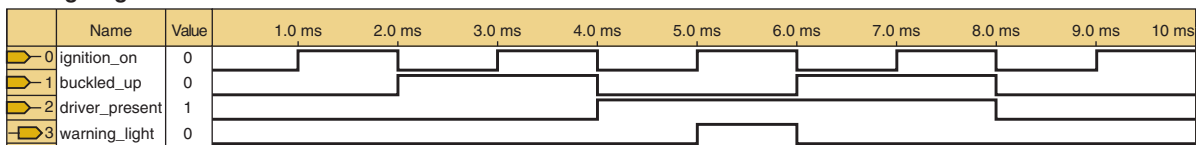
**Schematic diagram**

(b)

**Truth table**

driver_present	buckled_up	ignition_on	warning_light
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

(c)

**Timing diagram**

(d)

**FIGURE 3-42** Methods of describing logic circuits: (a) Boolean expression; (b) schematic diagram; (c) truth table; (d) timing diagram.

Figure 3-42 shows four different ways of representing the logic circuit that was described in English as the problem statement of Example 3-24. There are many other ways in which we could represent the logic of this decision. As an example we could dream up an entirely new set of graphic

symbols, or state the logical relationship in French or Japanese. Of course, we cannot cover all the possible ways of describing a logic circuit, but we must understand the most common methods to be able to communicate with others in this profession. Furthermore, certain situations are easier to describe using one method over another. In some cases, a picture is worth a thousand words, and in other cases words are concise enough and are more easily communicated to others. The important point here is that we need ways to describe and communicate the operation of digital systems.

Many tools have been developed to allow a designer to enter a circuit description into a computer for the purpose of documenting the circuit, simulating the circuit, and ultimately creating a functional circuit. The tool we recommend is from Altera Corporation, one of the top digital circuit suppliers in the world. Its Quartus II software is available free of charge and can be downloaded from its web site. It is easy to learn how to use, especially through the use of the tutorials that are available to owners of this book at [www.pearsonhighered.com/electronics](http://www.pearsonhighered.com/electronics). Quartus II offers a way for you to describe a circuit by drawing a logic diagram. The logic diagram in Figure 3-42(b) is a block description file (.bdf) that was generated using the Quartus II software. Notice that this diagram is made up of labeled input symbols, labeled output symbols, and logic gate symbols. All of these symbols are provided in a library of components included in Quartus II. The components are easily connected using a wire drawing tool.

After the designer draws a block description file (.bdf), she can open a simulation file in the form of a timing diagram. She creates the input waveforms, and the simulator draws the output waveform. The timing diagram shown in Figure 3-42(d) is an example of a Quartus II timing diagram simulation.

#### OUTCOME ASSESSMENT QUESTIONS

1. Name five ways to describe the operation of logic circuits.
2. Name two tools available in Quartus II software.

### 3-17 DESCRIPTION LANGUAGES VERSUS PROGRAMMING LANGUAGES\*

#### OUTCOMES

*Upon completion of this section, you will be able to:*

- Articulate the difference between hardware description language and computer programming languages.
- State the source of origin of VHDL and AHDL.

Recent trends in the field of digital systems are favoring text-based language description of digital circuits. You probably noticed that each description method in Figure 3-42 offers challenges to computer entry, whether it is due to overbars, symbols, formatting, or line-drawing issues. In this section, we will begin to learn some of the more advanced tools that professionals in the digital field use to describe the circuits that implement their ideas. These tools are referred to as **hardware description languages (HDLs)**. Even with the powerful computers we have today, it is not possible to describe a logic circuit in English

\*All sections covering hardware description languages may be skipped without loss of continuity in Chapters 3–13.

prose and expect the computer to understand what you mean. Computers need a more rigidly defined language. We will focus on two languages in this text: **Altera hardware description language (AHDL)** and **very high speed integrated circuit (VHSIC) hardware description language (VHDL)**.

## VHDL and AHDL

VHDL is not a new language. It was developed by the Department of Defense in the early 1980s as a concise way to document the designs in the very high speed integrated circuit (VHSIC) program. Appending HDL onto this acronym was too much, even for the military, and so the language was abbreviated to VHDL. Computer programs were developed to take the VHDL language files and simulate the operation of the circuits. With the growth of complex programmable logic devices in digital systems, VHDL has evolved into one of the primary high-level hardware description languages for designing and implementing digital circuits (synthesis). The language has been standardized by the IEEE, making it universally appealing for engineers as well as the makers of software tools that translate designs into the bit patterns used to program actual devices.

AHDL is a language that the Altera Corporation developed to provide a convenient way to configure the logic devices that they offer. Altera was one of the first companies to introduce logic devices that can be reconfigured electronically. These devices are called **programmable logic devices (PLDs)**. Unlike VHDL, AHDL is not intended to be used as a universal language for describing any logic circuit. It is intended to be used for programming complex digital systems into Altera PLDs in a language that is generally perceived to be easier to learn yet very similar to VHDL. It also has features that take full advantage of the architecture of Altera devices. All of the examples in this text will use the Altera Quartus II software to develop both AHDL and VHDL design files. You will see the advantage of using Altera's development system for both languages when you program an actual device. The Altera system makes circuit development very easy and contains all the necessary tools to translate from the HDL design file to a file ready to load into an Altera PLD. It also allows you to develop building blocks using schematic entry, AHDL, VHDL, and other methods and then interconnect them to form a complete system.

Other HDLs are available that are more suitable for programming simple programmable logic devices. You will find any of these languages easy to use after learning the basics of AHDL or VHDL as covered in this text.

## Computer Programming Languages

It is important to distinguish between hardware description languages intended to describe the hardware configuration of a circuit and programming languages that represent a sequence of instructions intended to be carried out by a computer to accomplish some task. In both cases, we use a *language* to *program* a device. However, computers are complex digital systems that are made up of logic circuits. Computers operate by following a laundry list of tasks (i.e., instructions, or "the program"), each of which must be done in sequential order. The speed of operation is determined by how fast the computer can execute each instruction. For example, if a computer were to respond to four different inputs, it would require at least four separate instructions (sequential tasks) to detect and identify which input changed state. A digital logic circuit, on the other hand, is limited in its speed only by how quickly the circuitry can change the outputs in

---



response to changes in the inputs. It is monitoring all inputs **concurrently** (at the same time) and responding to any changes.

The following analogy will help you understand the difference between computer operation and digital logic circuit operation and the role of language elements used to describe what the systems do. Consider the challenge of describing what is done to an Indy 500 car during a pit stop. If a single person performed all the necessary tasks one at a time, he or she would need to be very fast at each task. This is the way a computer operates: one task at a time but very quickly. Of course, at Indy, there is an entire pit crew that swarms the car, and each member of the crew does his or her task while the others do theirs. All crew members operate concurrently, like the elements of a digital circuit. Now consider how you would describe to someone else what is being done to the Indy car during the pit stop using (1) the individual-mechanic approach or (2) the pit-crew approach. Wouldn't the two English language descriptions of what is being done sound very similar? As we will see, the languages used to describe digital hardware (HDL) are very similar to languages that describe computer programs (e.g., BASIC, C, JAVA), even though the resulting implementation operates quite differently. Knowledge of any of these computer programming languages is not necessary to understand HDL. The important thing is that when you have learned both an HDL and a computer language, you must understand their different roles in digital systems.

### EXAMPLE 3-25

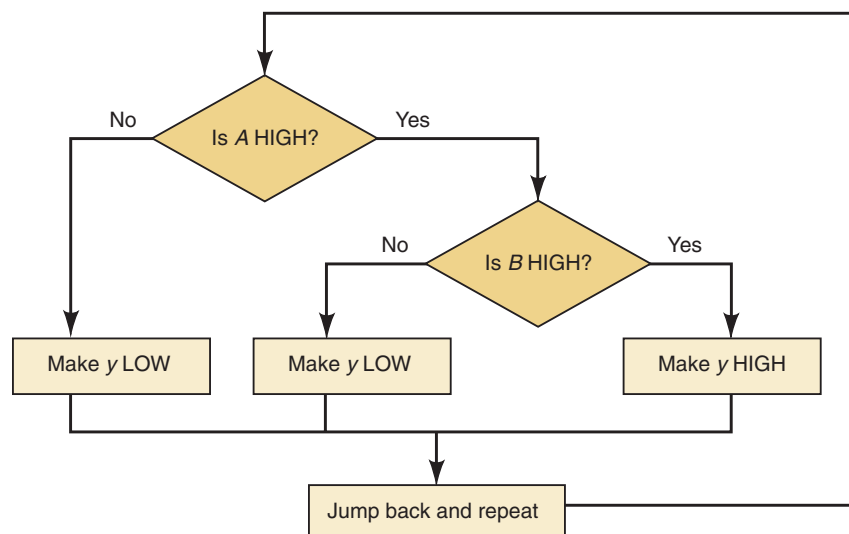
Compare the operation of a computer and a logic circuit in performing the simple logical operation of  $y = AB$ .

#### Solution

The logic circuit is a simple AND gate. The output  $y$  will be HIGH within approximately 10 nanoseconds of the point when  $A$  and  $B$  are HIGH simultaneously. Within approximately 10 nanoseconds after either input goes LOW, the output  $y$  will be LOW.

The computer must run a program of instructions that makes decisions. Suppose each instruction takes 20 ns (that's pretty fast!). Each shape in the flowchart shown in Figure 3-43 represents one instruction. Clearly, it will take a minimum of two or three instructions (40–60 ns) to respond to changes in the inputs.

**FIGURE 3-43** Decision process of a computer program.



**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What does HDL stand for?
2. What is the purpose of an HDL?
3. What is the purpose of a computer programming language?
4. What is the key difference between HDL and computer programming languages?
5. Who created AHDL?
6. Who created VHDL?

### 3-18 IMPLEMENTING LOGIC CIRCUITS WITH PLDs

#### OUTCOMES

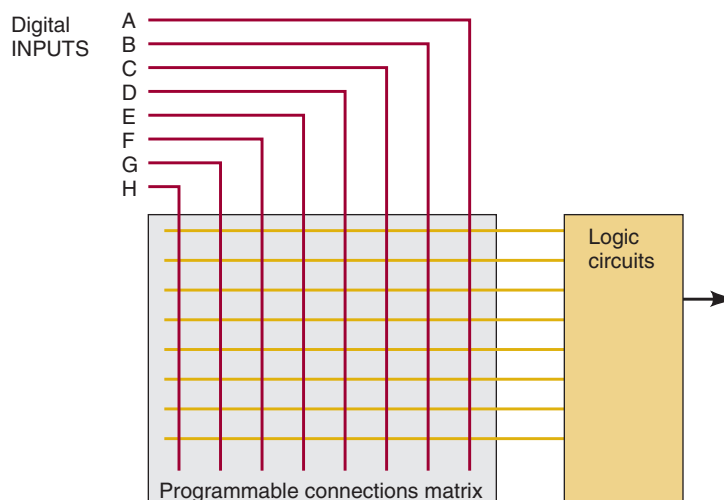
Upon completion of this section, you will be able to:

- Define PLD.
- Explain “programming” of a PLD.
- Explain the role of compiling.

Many digital circuits today are implemented using programmable logic devices (PLDs). These devices are not like microcomputers or microcontrollers that “run” the program of instructions. Instead, they are configured electronically, and their internal circuits are “wired” together electronically to form a logic circuit. This programmable wiring can be thought of as thousands of connections that are either connected (1) or not connected (0). Figure 3-44 shows a small area of programmable connections. Each intersection between a row (horizontal wire) and a column (vertical wire) is a programmable connection. You can imagine how difficult it would be to try to configure these devices by placing 1s and 0s in a grid manually (which is how they did it back in the 1970s).

The role of the hardware description language is to provide a concise and convenient way for the designer to describe the operation of the circuit in a format that a personal computer can handle and store conveniently. The computer runs a special software application called a **compiler** to translate from the hardware description language into the grid of 1s and 0s that can

**FIGURE 3-44** Configuring hardware connections with programmable logic devices.



be loaded into the PLD. If a person can master the higher level hardware description language, it actually makes programming the PLDs much easier than trying to use Boolean algebra, schematic drawings, or truth tables. In much the same way that you learned the English language, we will start by expressing simple things and gradually learn the more complicated aspects of these languages. Our objective is to learn enough of HDL to be able to communicate with others and perform simple tasks. A full understanding of all the details of these languages is beyond the scope of this text and can really be mastered only by regular use.

In the sections throughout this book that cover the HDLs, we will present both AHDL and VHDL in a format that allows you to skip over one language and concentrate on the other without missing important information. Of course, this setup means there will be some redundant information presented if you choose to read about both languages. We feel this redundancy is worth the extra effort to provide you with the flexibility of focusing on either of the two languages or learning both by comparing and contrasting similar examples. The recommended way to use the text is to focus on one language. It is true that the easiest way to become bilingual, and fluent in both languages, is to be raised in an environment where both languages are spoken routinely. It is also very easy, however, to confuse details, so we will keep the specific examples separate and independent. We hope this format provides you with the opportunity to learn one language now and then use this book as a reference later in your career should you need to pick up the second language.

#### OUTCOME ASSESSMENT QUESTIONS

1. What does PLD stand for?
2. How are the circuits reconfigured electronically in a PLD?
3. What does a compiler do?

## 3-19 HDL FORMAT AND SYNTAX

### OUTCOMES

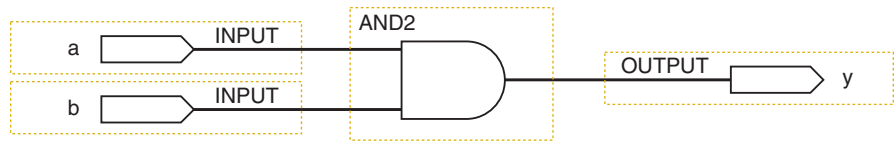
*Upon completion of this section, you will be able to:*

- Identify key words that are reserved by AHDL or VHDL.
- Use HDL syntax correctly.
- Write a simple source file.

Any language has its unique properties, similarities to other languages, and its proper syntax. When we study grammar in school, we learn conventions such as the order of words as elements in a sentence and proper punctuation. This is referred to as the **syntax** of language. A language designed to be interpreted by a computer must follow strict rules of syntax. A computer is just an assortment of processed beach sand and wire that has no idea what you “meant” to say, so you must present the instructions using the exact syntax that the computer language expects and understands. The basic format of any hardware circuit description (in any language) involves two vital elements:

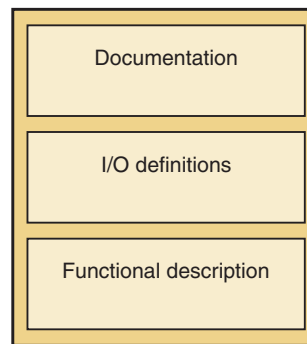
1. The definition of what goes into it and what comes out of it (i.e., input/output specs)
2. The definition of how the outputs respond to the inputs (i.e., its operation)

**FIGURE 3-45** A schematic diagram description.



A circuit schematic diagram such as Figure 3-45 can be read and understood by a competent engineer or technician because both would understand the meaning of each symbol in the drawing. If you understand how each element works and how the elements are connected to each other, you can understand how the circuit operates. On the left side of the diagram is the set of inputs and on the right is the set of outputs. The symbols in the middle define its operation. The text-based language must convey the same information. All HDLs use the format shown in Figure 3-46.

**FIGURE 3-46** Format of HDL files.



In a text-based language, the circuit being described must be given a name. The inputs and outputs (sometimes called ports) must be assigned names and defined according to the nature of the port. Is it a single bit from a toggle switch? Or is it a four-bit number coming from a keypad? The text-based language must somehow convey the nature of these inputs and outputs. The **mode** of a port defines whether it is input, output, or both. The **type** refers to the number of bits and how those bits are grouped and interpreted. If the *type* of input is a single bit, then it can have only two possible values: 0 and 1. If the type of input is a four-bit binary number from a keypad, it can have any one of 16 different values ( $0000_2$ – $1111_2$ ). The type determines the range of possible values. The definition of the circuit's operation in a text-based language is contained in a set of statements that follow the circuit input/output (I/O) definition. The following two sections describe the very simple circuit of Figure 3-45 and illustrate the critical elements of AHDL and VHDL.

## BOOLEAN DESCRIPTION USING AHDL

Refer to Figure 3-47. The keyword **SUBDESIGN** gives a name to the circuit block, which in this case is *and\_gate*. The name of the file must also be *and\_gate.tdf*. Notice that the keyword **SUBDESIGN** is capitalized. This is not required by the software, but use of a consistent style in capitalization makes the code much easier to read. The style guide that is provided with the Altera compiler for AHDL suggests the use of capital letters for the keywords in the language. Variables that are named by the designer should be lowercase.

**FIGURE 3-47** Essential elements in AHDL.

```

SUBDESIGN and_gate
(
  a, b      :INPUT;
  y         :OUTPUT;
)
BEGIN
  y = a & b;
END;

```

The SUBDESIGN section defines the inputs and outputs of the logic circuit block. Something must enclose the circuit that we are trying to describe, much the same way that a block diagram encloses everything that makes up that part of the design. In AHDL, this input/output definition is enclosed in parentheses. The list of variables used for inputs to this block are separated by commas and followed by :INPUT;. In AHDL, the single-bit type is assumed unless the variable is designated as multiple bits. The single-output bit is declared with the mode :OUTPUT;. We will learn the proper way to describe other types of inputs, outputs, and variables as we need to use them.

The set of statements that describe the operation of the AHDL circuit are contained in the logic section between the keywords BEGIN and END. END must be followed by a semicolon, similar to the way you would always end a paragraph with a period. In this example, the operation of the hardware is described by a very simple Boolean algebra equation that states that the output ( $y$ ) is assigned ( $=$ ) the logic level produced by  $a$  AND  $b$ . This Boolean algebra equation is referred to as a **concurrent assignment statement**. Any statements (there is only one in this example) between BEGIN and END are evaluated constantly and concurrently. The order in which they are listed makes no difference. The basic Boolean operators are:

&	AND
#	OR
!	NOT
\$	XOR

#### OUTCOME ASSESSMENT QUESTIONS

1. What appears inside the parentheses ( ) after SUBDESIGN?
2. What appears between BEGIN and END?

### BOOLEAN DESCRIPTION USING VHDL

Refer to Figure 3-48. The keyword **ENTITY** gives a name to the circuit block, which in this case is `and_gate`. Notice that the keyword ENTITY is capitalized but `and_gate` is not. This is not required by the software, but use of a consistent style in capitalization makes the code much easier to read. The style guide provided with the Altera compiler for VHDL suggests using capital letters for the keywords in the language. Variables that are named by the designer should be lowercase.

**FIGURE 3-48** Essential elements in VHDL.

```

ENTITY and_gate IS
PORT (  a, b  :IN BIT;
        y      :OUT BIT);
END and_gate;
ARCHITECTURE ckt OF and_gate IS
BEGIN
        y <= a AND b;
END ckt;

```

The ENTITY declaration can be thought of as a block description. Something must enclose the circuit we are trying to describe, much the same way a block diagram encloses everything that makes up that part of the design. In VHDL, the keyword PORT tells the compiler that we are defining inputs and outputs to this circuit block. The names used for inputs (separated by commas) are listed, ending with a colon and a description of the mode and type of input (:IN BIT;). In VHDL, the **BIT** description tells the compiler that each variable in the list is a single bit. We will learn the proper way to describe other types of inputs, outputs, and variables as we need to use them. The line containing END and\_gate; terminates the ENTITY declaration.

The ARCHITECTURE declaration is used to describe the operation of everything inside the block. The designer makes up a name for this architectural description of the inner workings of the ENTITY block (ckt in this example). Every ENTITY must have at least one ARCHITECTURE associated with it. The words OF and IS are keywords in this declaration. The body of the architecture description is enclosed between the BEGIN and END keywords. END is followed by the name that has been assigned to this architecture. This line must be punctuated with a semicolon, similar to the way you end a paragraph with a period. Within the body (between BEGIN and END) is the description of the block's operation. In this example, the operation of the hardware is described by a very simple Boolean algebra equation that states that the output (*y*) is assigned ( $\leq$ ) the logic level produced by *a* AND *b*. This is referred to as a **concurrent assignment statement**, which means that all the statements (there is only one in this example) between BEGIN and END are evaluated constantly and concurrently. The order in which they are listed makes no difference.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the role of the ENTITY declaration?
2. Which key section defines the operation of the circuit?
3. What is the assignment operator used to give a value to a logic signal?

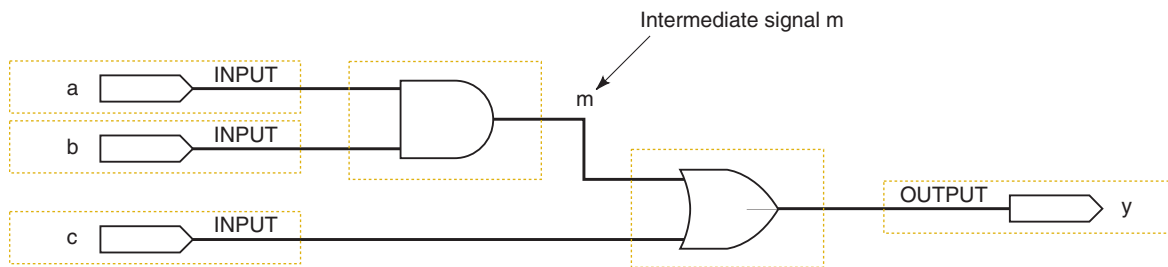
## 3-20 INTERMEDIATE SIGNALS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Define variables in HDL.
- Use variables in HDL code.
- Document the source file.

In many designs, there is a need to define signal points “inside” the circuit block. They are points in the circuit that are neither inputs nor outputs for the block but may be useful as a reference point. It may be a signal that needs to be connected to many other places within the block. In an analog or digital schematic diagram, they would be called test points or *nodes*. In an HDL, they are referred to as **buried nodes** or **local signals**. Figure 3-49 shows a very simple circuit that uses an intermediate signal named *m*. In the HDL, these nodes (signals) are not defined with the inputs and outputs but rather in the section that describes the operation of the block. The inputs and outputs are available to other circuit blocks in the system, but these local signals are recognized only within this block.



**FIGURE 3-49** A logic circuit diagram with an intermediate variable.

In the example code that follows, notice the information at the top. The purpose of this information is strictly for documentation purposes. It is absolutely vital that the design is documented thoroughly. At a minimum, it should describe the project it is being used in, who wrote it, and the date. This information is often referred to as a header. We are keeping our headers brief to make this book a little lighter to carry to class, but remember: memory space is cheap and information is valuable. So don't be afraid to *document thoroughly!* There are also comments next to many of the statements in the code. These comments help the designer remember what she or he was trying to do and help any other person to understand what was intended.

## AHDL BURIED NODES

The AHDL code that describes the circuit in Figure 3-49 is shown in Figure 3-50. The **comments** in AHDL can be enclosed between % characters, as you can see in the figure between lines 1 and 4. This section of the code allows the designer to write many lines of information that will be ignored by computer programs using this file but can be read by any person trying to decipher the code. Notice that the comments at the end of lines 7, 8, 11, 13, and 14 are preceded by two dashes (--). The text following the dashes is for documentation only. Either type of comment symbol may be used, but percent signs must be used in pairs to open and close a comment. Double dashes indicate a comment that extends to the end of the line.

In AHDL, local signals are declared in the **VARIABLE** section, which is placed between the **SUBDESIGN** section and the logic section. The intermediate signal *m* is defined on line 11, following the keyword **VARIABLE**. The keyword **NODE** designates the nature of the variable. Notice that a colon separates the variable name from its node designation. In the hardware

**FIGURE 3-50**

Intermediate variables in AHDL described in Figure 3-49.

```

1      % Intermediate variables in AHDL (Figure 3-49)
2      Digital Systems 12th ed
3      NS Widmer
4      May 27, 2015          %
5      SUBDESIGN fig3_50
6      (
7          a,b,c      :INPUT;    -- define inputs to block
8          y          :OUTPUT;   -- define block output
9      )
10     VARIABLE
11     m              :NODE;      -- name an intermediate signal
12     BEGIN
13         m = a & b;           -- generate buried product term
14         y = m # c;          -- generate sum on output
15     END;
```

description on line 13, the intermediate variable is assigned (connected to) a value ( $m = a \& b$ ;) and then  $m$  is used in the second statement on line 14 to assign (connect) a value to  $y$  ( $y = m \# c$ ;) . Remember that the assignment statements are concurrent and, thus, the order in which they are given does not matter. For human readability, it may seem more logical to assign values to intermediate variables before they are used in other assignment statements, as shown here.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the designation used for intermediate variables?
2. Where are these variables declared?
3. Does it matter whether the  $m$  or  $y$  equation comes first?
4. What character is used to limit a block of comments?
5. What characters are used to comment a single line?

## VHDL LOCAL SIGNALS

The VHDL code that describes the circuit in Figure 3-49 is shown in Figure 3-51. The **comments** in VHDL follow two dashes (--). Typing two successive dashes allows the designer to write information from that point to the end of the line. The information following the two successive dashes will be ignored by computer programs using this file, but can be read by any person trying to decipher the code.

The intermediate signal  $m$  is defined on line 13 following the keyword SIGNAL. The keyword BIT designates the type of the signal. Notice that a colon separates the signal name from its type designation. In the hardware description on line 16, the intermediate signal is assigned (connected to) a value ( $m \leq a \text{ AND } b$ ;) and then  $m$  is used in the statement on line 17



```

1  -- Intermediate variables in VHDL (Figure 3-49)
2  -- Digital Systems 12th ed
3  -- NS Widmer
4  -- May 27, 2015
5
6  ENTITY fig3_51 IS
7  PORT( a, b, c   :IN BIT;    -- define inputs to block
8        y         :OUT BIT); -- define block output
9  END fig3_51;
10
11 ARCHITECTURE ckt OF fig3_51 IS
12
13     SIGNAL m     :BIT;      -- name an intermediate signal
14
15 BEGIN
16     m <= a AND b;          -- generate buried product term
17     y <= m OR c;          -- generate sum on output
18 END ckt;

```

**FIGURE 3-51** Intermediate signals in VHDL described in Figure 3-49.

to assign (connect) a value to  $y$  ( $y \leq m \text{ OR } c$ ). Remember that the assignment statements are concurrent and, thus, the order in which they are given does not matter. For human readability, it may seem more logical to assign values to intermediate signals before they are used in other assignment statements, as shown here.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the designation used for intermediate signals?
2. Where are these signals declared?
3. Does it matter whether the  $m$  or  $y$  equation comes first?
4. What characters are used to comment a single line?

## SUMMARY

1. Boolean algebra is a mathematical tool used in the analysis and design of digital circuits.
2. The basic Boolean operations are the OR, AND, and NOT operations.
3. An OR gate produces a HIGH output when any input is HIGH. An AND gate produces a HIGH output only when all inputs are HIGH. A NOT circuit (INVERTER) produces an output that is the opposite logic level compared to the input.
4. A NOR gate is the same as an OR gate with its output connected to an INVERTER. A NAND gate is the same as an AND gate with its output connected to an INVERTER.

5. Boolean theorems and rules can be used to simplify the expression of a logic circuit and can lead to a simpler way of implementing the circuit.
6. NAND gates can be used to implement any of the basic Boolean operations. NOR gates can be used likewise.
7. Either standard or alternate symbols can be used for each logic gate, depending on whether the output is to be active-HIGH or active-LOW.
8. Propagation delay is the time between an input transition and the circuit's resulting response.
9. Hardware description languages have become an important method of describing digital circuits.
10. HDL code should always contain comments that document its vital characteristics so a person reading it later can understand what it does.
11. Every HDL circuit description contains a definition of the inputs and outputs, followed by a section that describes the circuit's operation.
12. In addition to inputs and outputs, intermediate connections that are buried within the circuit can be defined. These intermediate connections are called nodes or signals.

## IMPORTANT TERMS

---

logic level	active logic levels	concurrent
Boolean algebra	active-HIGH	compiler
truth table	active-LOW	syntax
OR operation	asserted	mode
OR gate	unasserted	type
AND operation	propagation delay	SUBDESIGN
AND gate	hardware description	concurrent
NOT operation	languages (HDLs)	assignment
inversion	Altera hardware	statement
(complementation)	description language	ENTITY
NOT circuit	(AHDL)	BIT
(INVERTER)	very high speed	ARCHITECTURE
NOR gate	integrated circuit	buried nodes (local
NAND gate	(VHSIC) hardware	signals)
Boolean theorems	description language	comments
DeMorgan's theorems	(VHDL)	VARIABLE
alternate logic	programmable logic	NODE
symbols	devices (PLDs)	

## PROBLEMS

---

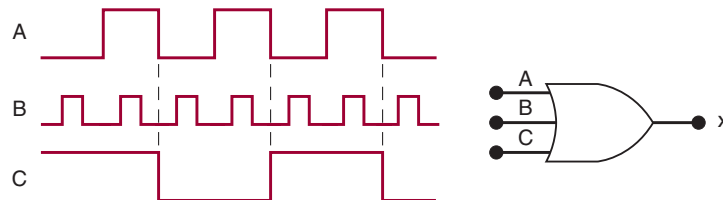
The color letters preceding some of the problems are used to indicate the nature or type of problem as follows:

- B** basic problem
  - T** troubleshooting problem
  - D** design or circuit-modification problem
  - N** new concept or technique not covered in text
  - C** challenging problem
  - H** HDL problem
-

## SECTION 3-3

- B** 3-1.\* (a) Draw the output waveform for the OR gate of Figure 3-52.
- B** (b) Suppose that the  $A$  input in Figure 3-52 is unintentionally shorted to ground (i.e.,  $A = 0$ ). Draw the resulting output waveform.
- B** (c) Suppose that the  $A$  input in Figure 3-52 is unintentionally shorted to the +5V supply line (i.e.,  $A = 1$ ). Draw the resulting output waveform.

FIGURE 3-52



- 3-2. A three-input OR gate should be producing a logic 0 at its output but instead it is producing a logic 1. How can you determine which of the three inputs is incorrect?
- C** 3-3. Read the statements below concerning an OR gate. At first, they may appear to be valid, but after some thought you should realize that neither one is *always* true. Prove this by showing a specific example to refute each statement.
- (a) If the output waveform from an OR gate is the same as the waveform at one of its inputs, the other input is being held permanently LOW.
- (b) If the output waveform from an OR gate is always HIGH, one of its inputs is being held permanently HIGH.
- B** 3-4. How many different sets of input conditions will produce a HIGH output from a five-input OR gate?

## SECTION 3-4

- 3-5. A three-input AND gate should be producing a logic 1 at its output but instead it is producing a logic 0. How can you determine which of the three inputs is incorrect?
- B** 3-6. Change the OR gate in Figure 3-52 to an AND gate.
- (a)\* Draw the output waveform.
- (b) Draw the output waveform if the  $A$  input is permanently shorted to ground.
- (c) Draw the output waveform if  $A$  is permanently shorted to +5V.
- D** 3-7.\* Refer to Figure 3-4. Modify the circuit so that the alarm is to be activated only when the pressure and the temperature exceed their maximum limits at the same time.
- B** 3-8.\* Change the OR gate in Figure 3-6 to an AND gate and draw the output waveform.
- B** 3-9. Suppose that you have an unknown two-input gate that is either an OR gate or an AND gate. What combination of input levels should you apply to the gate's inputs to determine which type of gate it is?

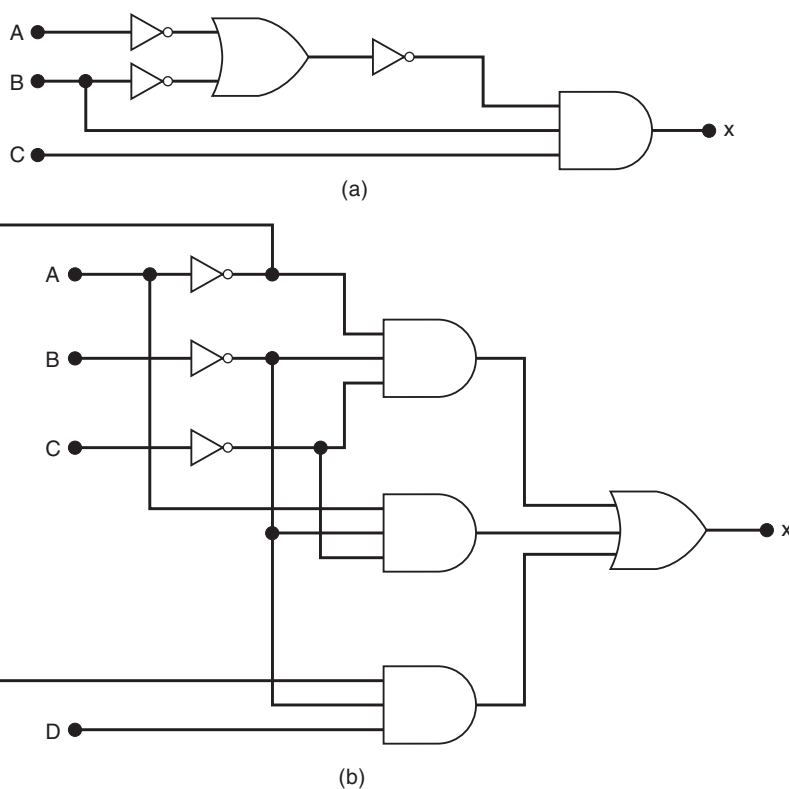
\*Answers to problems marked with an asterisk can be found in the back of the text.

- B** 3-10. *True or false:* No matter how many inputs it has, an AND gate will produce a HIGH output for only one combination of input levels.

**SECTIONS 3-5 TO 3-7**

- B** 3-11. Apply the *A* waveform from Figure 3-23 to the input of an INVERTER. Draw the output waveform. Repeat for waveform *B*.
- B** 3-12. (a)\* Write the Boolean expression for output *x* in Figure 3-53(a). Determine the value of *x* for all possible input conditions, and list the values in a truth table.  
(b) Repeat for the circuit in Figure 3-53(b).

**FIGURE 3-53**



- B** 3-13.\* Create a complete analysis table for the circuit of Figure 3-15(b) by finding the logic levels present at each gate output for each of the 32 possible input combinations.
- B** 3-14. (a)\*Change each OR to an AND, and each AND to an OR, in Figure 3-15(b). Then write the expression for the output.  
(b) Complete an analysis table.
- B** 3-15. Create a complete analysis table for the circuit in Figure 3-15(a) by finding the logic levels present at each gate output for each of the 16 possible input combinations.

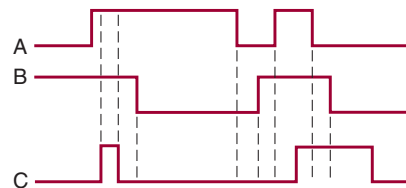
**SECTION 3-8**

- B** 3-16. For each of the following expressions, construct the corresponding logic circuit, using AND and OR gates and INVERTERS.
  - (a)\* $x = \overline{AB(C + D)}$
  - (b)\* $z = A + B + \overline{CDE} + \overline{BCD}$
  - (c)  $y = (\overline{M} + \overline{N} + \overline{PQ})$

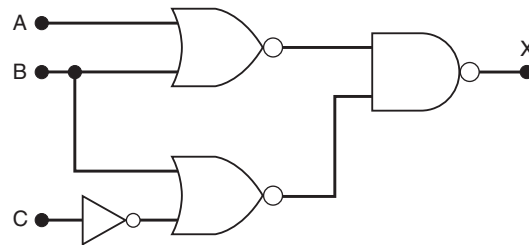
- (d)  $x = \overline{W + P\overline{Q}}$   
 (e)  $z = MN(P + \overline{N})$   
 (f)  $x = (A + B)(\overline{A} + \overline{B})$   
 (g)  $g = AC + \overline{BC}$   
 (h)  $h = \overline{AB} + \overline{CD}$

**SECTION 3-9**

- B** 3-17.\*(a) Apply the input waveforms of Figure 3-54 to a NOR gate, and draw the output waveform.  
 (b) Repeat with C held permanently LOW.  
 (c) Repeat with C held HIGH.

**FIGURE 3-54**

- B** 3-18. Repeat Problem 3-17 for a NAND gate.  
**C** 3-19.\*Write the expression for the output of Figure 3-55, and use it to determine the complete truth table. Then apply the waveforms of Figure 3-54 to the circuit inputs, and draw the resulting output waveform.

**FIGURE 3-55**

- B** 3-20. Determine the truth table for the circuit of Figure 3-24.  
**B** 3-21. Modify the circuits that were constructed in Problem 3-16 so that NAND gates and NOR gates are used wherever appropriate.

**SECTION 3-10**

- C** 3-22. Prove theorems (15a) and (15b) by trying all possible cases.

- B** 3-23.\***DRILL QUESTION**  
 Complete each expression.

- |   |  |
|---|--|
| (a) $A + 1 = \underline{\hspace{2cm}}$                | (f) $D \cdot 1 = \underline{\hspace{2cm}}$         |
| (b) $A \cdot A = \underline{\hspace{2cm}}$            | (g) $D + 0 = \underline{\hspace{2cm}}$             |
| (c) $B \cdot \overline{B} = \underline{\hspace{2cm}}$ | (h) $C + \overline{C} = \underline{\hspace{2cm}}$  |
| (d) $C + C = \underline{\hspace{2cm}}$                | (i) $G + GF = \underline{\hspace{2cm}}$            |
| (e) $x \cdot 0 = \underline{\hspace{2cm}}$            | (j) $y + \overline{w}y = \underline{\hspace{2cm}}$ |

- C** 3-24. (a)\*Simplify the following expression using theorems (13b), (3), and (4):

$$x = (M + N)(\bar{M} + P)(\bar{N} + \bar{P})$$

- (b) Simplify the following expression using theorems (13a), (8), and (6):

$$z = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + \bar{B}\bar{C}D$$

**SECTIONS 3-11 AND 3-12**

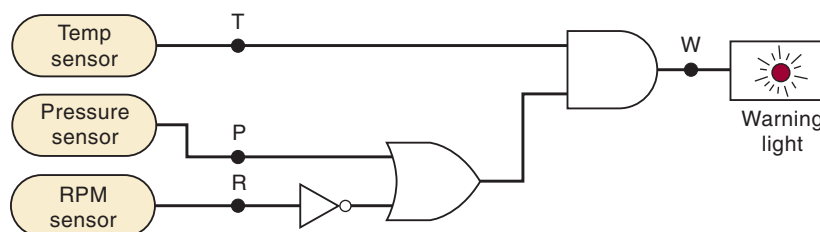
- C** 3-25. Prove DeMorgan's theorems by trying all possible cases.
- B** 3-26. Simplify each of the following expressions using DeMorgan's theorems.
- |                                     |  |   |
|-------------------------------------|--|---|
| (a)* $\overline{ABC}$               | (d) $\overline{A + B}$                       | (g)* $\overline{A(B + \bar{C})D}$           |
| (b) $\overline{\bar{A} + \bar{B}C}$ | (e)* $\overline{\bar{A}B}$                   | (h) $\overline{(M + \bar{N})(\bar{M} + N)}$ |
| (c)* $\overline{AB\bar{C}D}$        | (f) $\overline{\bar{A} + \bar{C} + \bar{D}}$ | (i) $\overline{\overline{ABCD}}$            |
- B** 3-27.\*Use DeMorgan's theorems to simplify the expression for the output of Figure 3-55.
- C** 3-28. Convert the circuit of Figure 3-53(b) to one using only NAND gates. Then write the output expression for the new circuit, simplify it using DeMorgan's theorems, and compare it with the expression for the original circuit.
- C** 3-29. Convert the circuit of Figure 3-53(a) to one using only NOR gates. Then write the expression for the new circuit, simplify it using DeMorgan's theorems, and compare it with the expression for the original circuit.
- B** 3-30. Show how a two-input NAND gate can be constructed from two-input NOR gates.
- B** 3-31. Show how a two-input NOR gate can be constructed from two-input NAND gates.
- C** 3-32. A jet aircraft employs a system for monitoring the rpm, pressure, and temperature values of its engines using sensors that operate as follows:

*RPM* sensor output = 0 only when speed < 4800 rpm  
*P* sensor output = 0 only when pressure < 220 psi  
*T* sensor output = 0 only when temperature < 200° F

Figure 3-56 shows the logic circuit that controls a cockpit warning light for certain combinations of engine conditions. Assume that a HIGH at output *W* activates the warning light.

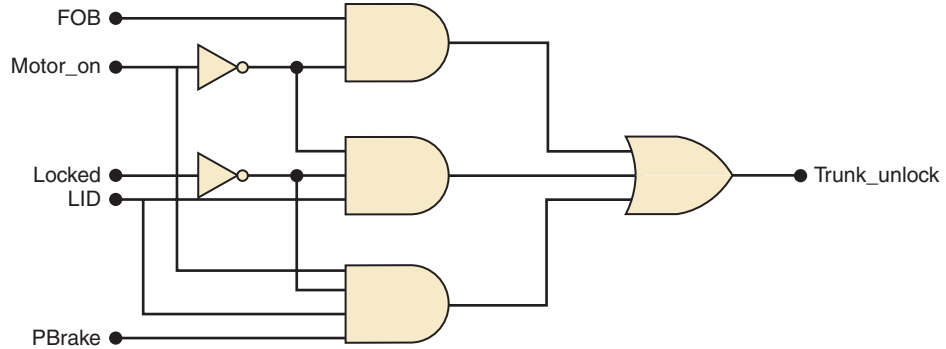
- (a)\*Determine what engine conditions will give a warning to the pilot.
- (b) Change this circuit to one using all NAND gates.

**FIGURE 3-56**



3-33. The trunk of an automobile is opened in one of two ways: by pressing a button on the trunk lid or by pressing the trunk button on the key fob. However, these buttons only open the trunk under certain conditions for safety and security purposes. The logic diagram for this circuit is shown in Figure 3-57.

FIGURE 3-57



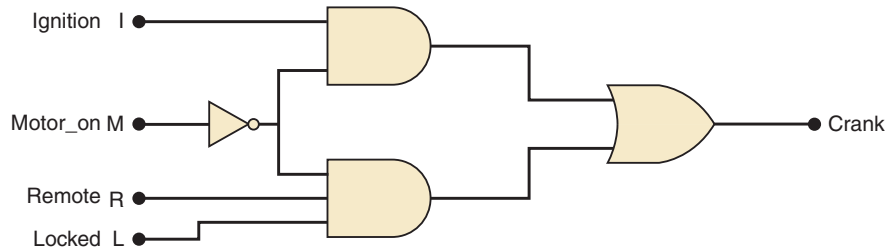
The output is Trunk\_unlock  
 HIGH activates the latch release and opens the trunk.  
 The inputs are defined as follows:

Button on trunk lid	LID	LOW = not pressed	HIGH = pressed
Button on key fob	FOB	LOW = not pressed	HIGH = pressed
Condition of door locks	Locked	Low = unlocked	HIGH = locked
Parking brake	PBrake	Low = not set (off)	HIGH = brake set
Engine status	Motor_on	Low = off	HIGH = motor on

- (a) Write the conditions in English that will open the trunk.
- (b) Write the Boolean equation using the signal names given.
- (c) Redraw the circuit using all NAND gates (assume you have up to four-input NAND gates available).

3-34. The remote start for an automobile will crank the engine under certain conditions. The logic circuit is shown in Figure 3-58. Inputs are defined as follows:

FIGURE 3-58



I	Ignition	Ignition switch in START position = HIGH
M	Motor_on	Engine running = HIGH
R	Remote	Remote start button on FOB pressed = HIGH
L	Locked	Doors locked = HIGH

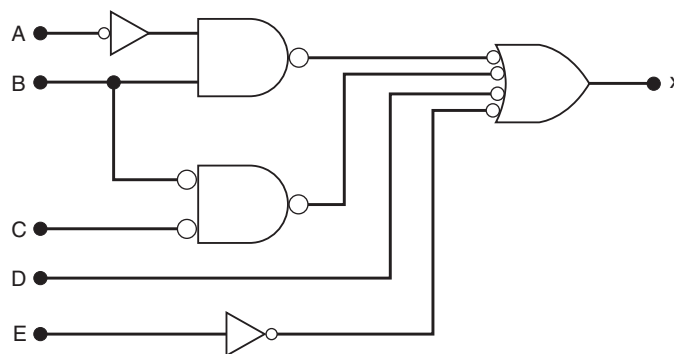
- (a) Write the Boolean expression from the circuit diagram.
- (b) Draw the truth table for this circuit.
- (c) Write the unsimplified SOP expression (using all four-variable product terms).

- (d) Use Boolean algebra to simplify the SOP expression in (c) to match the expression in (a)
- (e) Implement this circuit using only NAND gates.

### SECTIONS 3-13 AND 3-14

- B** 3-35.\* For each statement below, draw the appropriate logic-gate symbol—standard or alternate—for the given operation.
- (a) A HIGH output occurs only when all three inputs are LOW.
- (b) A LOW output occurs when any of the four inputs is LOW.
- (c) A LOW output occurs only when all eight inputs are HIGH.
- B** 3-36. Draw the standard representations for each of the basic logic gates. Then draw the alternate representations.
- C** 3-37. The circuit of Figure 3-55 is supposed to be a simple digital combination lock whose output will generate an active-LOW  $\overline{UNLOCK}$  signal for only one combination of inputs.
- (a)\* Modify the circuit diagram so that it represents more effectively the circuit operation.
- (b) Use the new circuit diagram to determine the input combination that will activate the output. Do this by working back from the output using the information given by the gate symbols, as was done in Examples 3-22 and 3-23. Compare the results with the truth table obtained in Problem 3-19.
- C** 3-38. (a) Determine the input conditions needed to activate output Z in Figure 3-37(b). Do this by working back from the output, as was done in Examples 3-22 and 3-23.
- (b) Assume that it is the LOW state of Z that is to activate the alarm. Change the circuit diagram to reflect this, and then use the revised diagram to determine the input conditions needed to activate the alarm.
- D** 3-39. Modify the circuit of Figure 3-40 so that  $A_1 = 0$  is needed to produce  $LCD = 1$  instead of  $A_1 = 1$ .
- B** 3-40.\* Determine the input conditions needed to cause the output in Figure 3-59 to go to its active state.

FIGURE 3-59

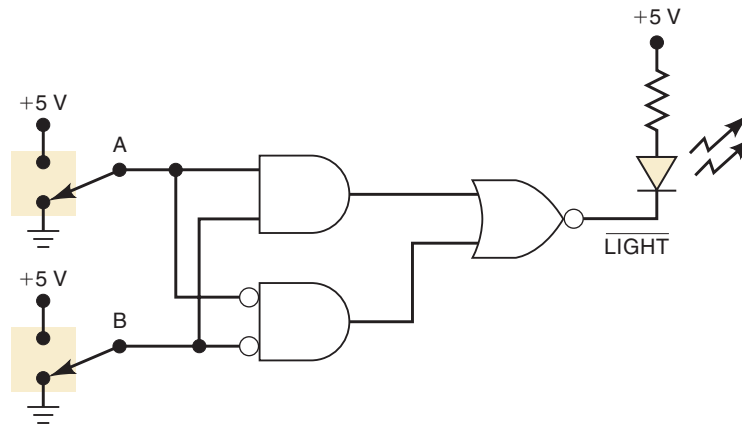


- B** 3-41.\* (a) What is the asserted state for the output of Figure 3-59?
- (b) What is the asserted state for the output of Figure 3-36(c)?
- B** 3-42. Use the results of Problem 3-40 to obtain the complete truth table for the circuit of Figure 3-59.



- N** 3-43.\* Figure 3-60 shows an application of logic gates that simulates a two-way switch like the ones used in our homes to turn a light on or off from two different switches. Here the light is an LED that will be ON (conducting) when the NOR gate output is LOW. Note that this output is labeled  $\overline{LIGHT}$  to indicate that it is active-LOW. Determine the input conditions needed to turn on the LED. Then verify that the circuit operates as a two-way switch using switches  $A$  and  $B$ . (In Chapter 4, you will learn how to design circuits like this one to produce a given relationship between inputs and outputs.)

FIGURE 3-60



## SECTION 3-15

- B** 3-44. A 7406 TTL inverter has a maximum  $t_{PLH}$  of 15 ns and a  $t_{PHL}$  of 23 ns. A positive pulse that lasts 100 ns is applied to the input.
- Draw the input and output waveforms. Scale the X-axis such that the end time is 200 ns.
  - Label  $t_{PLH}$  and  $t_{PHL}$  on the graph.
  - What is the pulse width of the output if worst case propagation delays occur?

## SECTION 3-17

## HDL DRILL QUESTIONS

- H** 3-45.\* True or false:
- VHDL is a computer programming language.
  - VHDL can accomplish the same thing as AHDL.
  - AHDL is an IEEE standard language.
  - Each intersection in a switch matrix can be programmed as an open or short circuit between a row and column wire.
  - The first item that appears at the top of an HDL listing is the functional description.
  - The type of an object indicates if it is an input or an output.
  - The mode of an object determines if it is an input or an output.
  - Buried nodes are nodes that have been deleted and will never be used again.
  - Local signals are another name for intermediate variables.
  - The header is a block of comments that document vital information about the project.

**SECTION 3-18**

- B** 3-46. Redraw the programmable connection matrix from Figure 3-44. Label the output signals (horizontal lines) from the connection matrix (from top row to bottom row) as follows: AAABADHE. Draw an X in the appropriate intersections to short-circuit a row to a column and create these connections to the logic circuit.
- H** 3-47.\* Write the HDL code in the language of your choice that will produce the following output functions:

$$\begin{aligned} X &= A + B \\ Y &= \overline{AB} \\ Z &= A + B + C \end{aligned}$$

- H** 3-48. Write the HDL code in the language of your choice that will implement the logic circuit of Figure 3-39.
- (a) Use a single Boolean equation.
- (b) Use the intermediate variables  $V$ ,  $W$ ,  $X$ , and  $Y$ .

**MICROCOMPUTER APPLICATION**

- C** 3-49.\* Refer to Figure 3-40 in Example 3-23. Inputs  $A_7$  through  $A_0$  are *address* inputs that are supplied to this circuit from outputs of the microprocessor chip in a microcomputer. The eight-bit address code  $A_7$  to  $A_0$  selects which device the microprocessor wants to activate. In Example 3-23, the required address code to activate the LCD was  $A_7$  through  $A_0 = 11111110_2 = FE_{16}$ .
- Modify the circuit so that the microprocessor must supply an address code of  $4A_{16}$  to activate the LCD.

**CHALLENGING EXERCISES**

- C** 3-50. Show how  $x = ABC$  can be implemented with one two-input NOR and one two-input NAND gate.
- C** 3-51.\* Implement  $y = ABCD$  using only two-input NAND gates.

**ANSWERS TO OUTCOME ASSESSMENT QUESTIONS****SECTION 3-1**

1. Constant; GND    2. Variables;  $A$ ,  $B$     3. See glossary

**SECTION 3-2**

1.  $x = 1$     2.  $x = 0$     3. 32

**SECTION 3-3**

1. All inputs LOW.    2.  $x = A + B + C + D + E + F$     3. Constant HIGH.

**SECTION 3-4**

1. All five inputs = 1.    2. A LOW input will keep the output LOW.    3. False; see truth table of each gate.

**SECTION 3-5**

1. Output of second INVERTER will be the same as input  $A$ .    2.  $y$  will be LOW only for  $A = B = 1$ .

**SECTION 3-6**

$$1. x = \bar{A} + B + C + \bar{AD} \quad 2. x = D(\overline{AB + C}) + E$$

**SECTION 3-7**

$$1. x = 1 \quad 2. x = 1 \quad 3. x = 1 \text{ for both.}$$

**SECTION 3-8**

1. See Figure 3-15(a).    2. See Figure 3-17(b).    3. See Figure 3-15(b).

**SECTION 3-9**

$$1. \text{ All inputs LOW.} \quad 2. x = 0 \quad 3. x = \overline{A + B + \bar{CD}}$$

**SECTION 3-10**

$$1. y = AC \quad 2. y = \bar{A}\bar{B}\bar{D} \quad 3. y = \bar{AD} + BD$$

**SECTION 3-11**

1.  $z = \bar{A}\bar{B} + C$     2.  $y = (\bar{R} + S + \bar{T})Q$     3. Same as Figure 3-28 except NAND is replaced by NOR.    4.  $y = \bar{AB}(C + \bar{D})$

**SECTION 3-12**

1. Three.    2. NOR circuit is more efficient because it can be implemented with only three NOR gates.    3.  $x = (\overline{AB})(\overline{CD}) = \overline{AB} + \overline{CD} = AB + CD$     4. 3    5. 1

**SECTION 3-13**

1. Output goes LOW when any input is HIGH.    2. Output goes HIGH only when all inputs are LOW.    3. Output goes LOW when any input is LOW.    4. Output goes HIGH only when all inputs are HIGH.

**SECTION 3-14**

1. Z will go HIGH when  $A = B = 0$  and  $C = D = 1$     2. Z will go LOW when  $A = B = 0, E = 1$ , and either C or D or both are 0.    3. Two    4. Two  
5. LOW    6.  $A = B = 0, C = D = 1$     7.  $\bar{W}$

**SECTION 3-15**

1. The time scale is in nanoseconds and it takes a finite amount of time to change states.    2. From 50% point on the input to 50% point on the output    3.  $t_{PHL}$   
4.  $t_{PLH}$

**SECTION 3-16**

1. Boolean equation, truth table, logic diagram, timing diagram, language.  
2. Schematic entry of .bdf files and simulation using timing diagrams.

**SECTION 3-17**

1. Hardware description language    2. To describe a digital circuit and its operation.    3. To give a computer a sequential list of tasks.    4. HDL describes concurrent hardware circuits; computer instructions execute one at a time.  
5. Altera Corporation.    6. U.S. Dept. of Defense.

**SECTION 3-18**

1. Programmable logic device.    2. By making and breaking connections in a switching matrix.    3. It translates HDL code into a pattern of bits to configure the switching matrix.

**SECTION 3-19****AHDL**

1. The input and output definitions.
2. The description of how it operates.

**VHDL**

1. To give a name to the circuit and define its inputs and outputs.
2. The ARCHITECTURE description.
3. <=

**SECTION 3-20****AHDL**

1. NODE
2. After the I/O definition and before BEGIN.
3. No
4. %
5. --

**VHDL**

1. SIGNAL
  2. Inside ARCHITECTURE before BEGIN.
  3. No
  4. --
-



# COMBINATIONAL LOGIC CIRCUITS

## ■ OUTLINE

- |     |   |      |                                     |
|-----|---|------|-------------------------------------|
| 4-1 | Sum-of-Products Form                    | 4-10 | Troubleshooting Digital Systems     |
| 4-2 | Simplifying Logic Circuits              | 4-11 | Internal Digital IC Faults          |
| 4-3 | Algebraic Simplification                | 4-12 | External Faults                     |
| 4-4 | Designing Combinational Logic Circuits  | 4-13 | Troubleshooting Prototyped Circuits |
| 4-5 | Karnaugh Map Method                     | 4-14 | Programmable Logic Devices          |
| 4-6 | Exclusive-OR and Exclusive-NOR Circuits | 4-15 | Representing Data in HDL            |
| 4-7 | Parity Generator and Checker            | 4-16 | Truth Tables Using HDL              |
| 4-8 | Enable/Disable Circuits                 | 4-17 | Decision Control Structures in HDL  |
| 4-9 | Basic Characteristics of Digital ICs    |      |                                     |

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Convert a logic expression into a sum-of-products expression.
- Perform the necessary steps to reduce a sum-of-products expression to its simplest form.
- Use Boolean algebra and the Karnaugh map as tools to simplify and design logic circuits.
- Explain the operation of both exclusive-OR and exclusive-NOR circuits.
- Design simple logic circuits without the help of a truth table.
- Describe how to implement enable circuits.
- Cite the basic characteristics of TTL and CMOS digital ICs.
- Use the basic troubleshooting rules of digital systems.
- Deduce from observed results the faults of malfunctioning combinational logic circuits.
- Describe the fundamental idea of programmable logic devices (PLDs).
- Describe the steps involved in programming a PLD to perform a simple combinational logic function.
- Describe hierarchical design methods.
- Identify proper data types for single-bit, bit array, and numeric value variables.
- Describe logic circuits using HDL control structures IF/ELSE, IF/ELSIF, and CASE.
- Select the appropriate HDL control structure for a given problem.

## ■ INTRODUCTION

In Chapter 3, we studied the operation of all the basic logic gates, and we used Boolean algebra to describe and analyze circuits that were made up of combinations of logic gates. These circuits can be classified as *combinational* logic circuits because, at any time, the logic level at the output depends on the combination of logic levels present at the inputs. A combinational circuit has no *memory* characteristic, so its output depends *only* on the current value of its inputs.

In this chapter, we will continue our study of combinational circuits. To start, we will go further into the simplification of logic circuits. Two methods will be used: one uses Boolean algebra theorems; the other uses a *mapping* technique. In addition, we will study simple techniques for designing combinational logic circuits to satisfy a given set of requirements. A complete study of logic-circuit design is not one of our objectives, but the methods we introduce will provide a good introduction to logic design.

A good portion of this chapter is devoted to the topic of *troubleshooting*. This term has been adopted as a general description of the process of isolating a problem or fault in any system and identifying a way of fixing it. The analytical skills and efficient methods of troubleshooting are equally applicable to any system whether it is a plumbing problem, a problem with your car, a health issue, or a digital circuit. Digital systems, implemented using TTL-integrated circuits, have for decades provided an exceptional vehicle for the study of efficient, systematic troubleshooting methods. As with any system, the practical characteristics of the pieces that make up the system must be understood in order to effectively analyze its normal operation, locate the trouble, and propose a remedy. We will present some basic characteristics and typical failure modes of logic ICs in the TTL and CMOS families that are still commonly used for laboratory instruction in introductory digital courses and take advantage of this technology to teach some fundamental troubleshooting principles.

In the last sections of this chapter, we will extend our knowledge of programmable logic devices and hardware description languages. The concept of programmable hardware connections will be reinforced, and we will provide more details regarding the role of the development system. You will learn the steps followed in the design and development of digital systems today. Enough information will be provided to allow you to choose the correct types of data objects for use in simple projects to be presented later in this text. Finally, several control structures will be explained, along with some instruction regarding their appropriate use.

## 4-1 SUM-OF-PRODUCTS FORM

---

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Identify the form of a sum-of-products (SOP) expression.
- Identify the form of a product-of-sums (POS) expression.

The methods of logic-circuit simplification and design that we will study require the logic expression to be in a **sum-of-products (SOP)** form. Some examples of this form are:

1.  $ABC + \overline{A}B\overline{C}$
2.  $AB + \overline{A}B\overline{C} + \overline{C}\overline{D} + D$
3.  $\overline{A}B + \overline{C}\overline{D} + EF + GK + H\overline{L}$

Each of these sum-of-products expressions consists of two or more AND terms (products) that are ORed together. Each AND term consists of one or more variables *individually* appearing in either complemented or uncomplemented form. For example, in the sum-of-products expression  $ABC + \overline{A}B\overline{C}$ , the first AND product contains the variables  $A$ ,  $B$ , and  $C$  in their uncomplemented (not inverted) form. The second AND term contains  $A$  and  $C$  in their complemented (inverted) form. Note that in a sum-of-products expression, one inversion sign *cannot* cover more than one variable in a term (e.g., we cannot have  $\overline{ABC}$  or  $\overline{RST}$ ).

### Product-of-Sums

Another general form for logic expressions is sometimes used in logic-circuit design. Called the **product-of-sums (POS)** form, it consists of two or more OR

---

terms (sums) that are ANDed together. Each OR term contains one or more variables in complemented or uncomplemented form. Here are some product-of-sum expressions:

1.  $(A + \bar{B} + C)(A + C)$
2.  $(A + \bar{B})(\bar{C} + D)F$
3.  $(A + C)(B + \bar{D})(\bar{B} + C)(A + \bar{D} + \bar{E})$

The methods of circuit simplification and design that we will be using are based on the sum-of-products form, so we will not be doing much with the product-of-sums form. It will, however, occur from time to time in some logic circuits that have a particular structure.

### OUTCOME ASSESSMENT QUESTIONS

1. Which of the following expressions is in SOP form?
  - (a)  $AB + CD + E$
  - (b)  $AB(C + D)$
  - (c)  $(A + B)(C + D + F)$
  - (d)  $\overline{MN} + PQ$
2. Repeat question 1 for the POS form.

## 4-2 SIMPLIFYING LOGIC CIRCUITS

### OUTCOMES

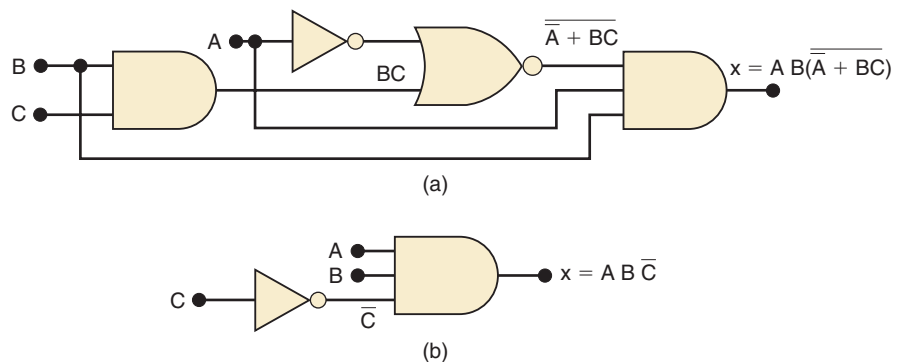
Upon completion of this section, you will be able to:

- Justify the use of simplification.
- Name two simplification techniques for digital circuits.

Once the expression for a logic circuit has been obtained, we may be able to reduce it to a simpler form containing fewer terms or fewer variables in one or more terms. The new expression can then be used to implement a circuit that is equivalent to the original circuit but that contains fewer gates and connections.

To illustrate, the circuit of Figure 4-1(a) can be simplified to produce the circuit of Figure 4-1(b). Both circuits perform the same logic, so it should be obvious that the simpler circuit is more desirable because it contains fewer

**FIGURE 4-1** It is often possible to simplify a logic circuit such as that in part (a) to produce a more efficient implementation, shown in (b).





gates and will therefore be smaller and cheaper than the original. Furthermore, the circuit reliability will improve because there are fewer interconnections that can be potential circuit faults.

Another strategic advantage of simplifying logic circuits involves the operational speed of circuits. Recall from previous discussions that logic gates are subject to propagation delay. If practical logic circuits are configured such that logical changes in the inputs must propagate through many layers of gates in order to determine the output, they cannot possibly operate as fast as circuits with fewer layers of gates. For example, compare the circuits of Figure 4-1(a) and (b). In Figure 4-1(a), the longest path a signal must travel involves three gates. In Figure 4-1(b), the longest signal path (C) only involves two gates. Working toward a common form such as SOP or POS assures similar propagation delay for all signals in the system and helps determine the maximum operating speed of the system.

In subsequent sections, we will study two methods for simplifying logic circuits. One method will utilize the Boolean algebra theorems and, as we shall see, is greatly dependent on inspiration and experience. The other method (Karnaugh mapping) is a systematic, step-by-step approach. Some instructors may wish to skip over this latter method because it is somewhat mechanical and probably does not contribute to a better understanding of Boolean algebra. This can be done without affecting the continuity or clarity of the rest of the text.

#### OUTCOME ASSESSMENT QUESTIONS

1. List two advantages of simplification.
2. List two methods of simplification.

### 4-3 ALGEBRAIC SIMPLIFICATION

#### OUTCOMES

*Upon completion of this section, you will be able to:*

- Apply Boolean algebra theorems and properties to reduce Boolean expressions.
- Manipulate expressions into POS or SOP form.

We can use the Boolean algebra theorems that we studied in Chapter 3 to help us simplify the expression for a logic circuit. Unfortunately, it is not always obvious which theorems should be applied to produce the simplest result. Furthermore, there is no easy way to tell whether the simplified expression is in its simplest form or whether it could have been simplified further. Thus, algebraic simplification often becomes a process of trial and error. With experience, however, one can become adept at obtaining reasonably good results.

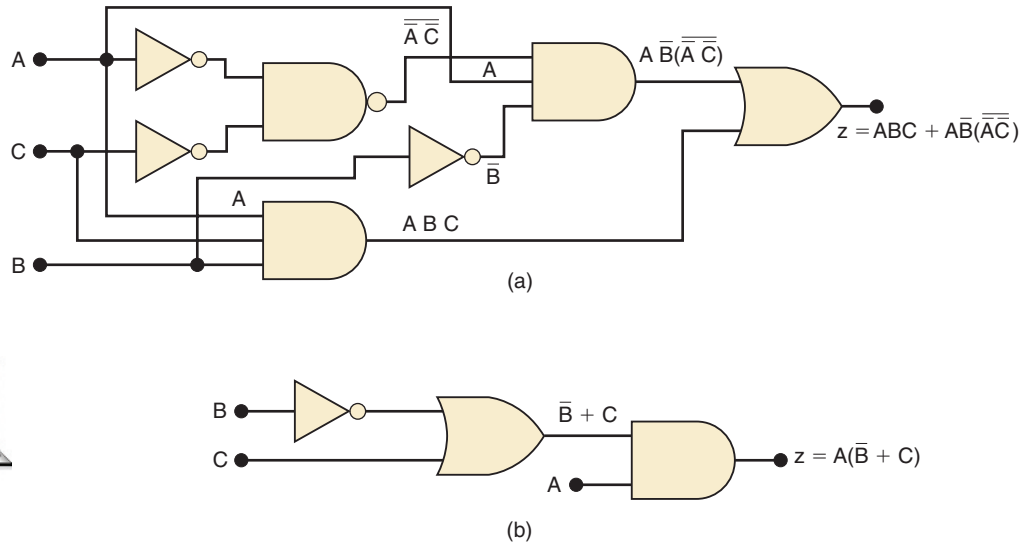
The examples that follow will illustrate many of the ways in which the Boolean theorems can be applied in trying to simplify an expression. You should notice that these examples contain two essential steps:

1. The original expression is put into SOP form by repeated application of DeMorgan's theorems and multiplication of terms.

2. Once the original expression is in SOP form, the product terms are checked for common factors, and factoring is performed wherever possible. The factoring should result in the elimination of one or more terms.

**EXAMPLE 4-1**

Simplify the logic circuit shown in Figure 4-2(a).



**FIGURE 4-2** Example 4-1.

**Solution**

The first step is to determine the expression for the output using the method presented in Section 3-6. The result is

$$z = ABC + A\bar{B} \cdot (\bar{A}\bar{C})$$

Once the expression is determined, it is usually a good idea to break down all large inverter signs using DeMorgan's theorems and then multiply out all terms.

$$\begin{aligned} z &= ABC + A\bar{B}(\bar{A} + \bar{C}) && \text{[theorem (17)]} \\ &= ABC + A\bar{B}(A + C) && \text{[cancel double inversions]} \\ &= ABC + A\bar{B}A + A\bar{B}C && \text{[multiply out]} \\ &= ABC + A\bar{B} + A\bar{B}C && \text{[} A \cdot A = A \text{]} \end{aligned}$$

With the expression now in SOP form, we should look for common variables among the various terms with the intention of factoring. The first and third terms above have  $AC$  in common, which can be factored out:

$$z = AC(B + \bar{B}) + A\bar{B}$$

Since  $B + \bar{B} = 1$ , then

$$\begin{aligned} z &= AC(1) + A\bar{B} \\ &= AC + A\bar{B} \end{aligned}$$

We can now factor out  $A$ , which results in

$$z = A(C + \bar{B})$$

This result can be simplified no further. Its circuit implementation is shown in Figure 4-2(b). It is obvious that the circuit in Figure 4-2(b) is a great deal simpler than the original circuit in Figure 4-2(a).

#### EXAMPLE 4-2

Simplify the expression  $z = A\bar{B}\bar{C} + A\bar{B}C + ABC$ .

#### Solution

The expression is already in SOP form.

*Method 1:* The first two terms in the expression have the product  $A\bar{B}$  in common. Thus,

$$\begin{aligned} z &= A\bar{B}(\bar{C} + C) + ABC \\ &= A\bar{B}(1) + ABC \\ &= A\bar{B} + ABC \end{aligned}$$

We can factor the variable  $A$  from both terms:

$$z = A(\bar{B} + BC)$$

Invoking theorem (15b):

$$z = A(\bar{B} + C)$$

*Method 2:* The original expression is  $z = A\bar{B}\bar{C} + A\bar{B}C + ABC$ . The first two terms have  $A\bar{B}$  in common. The last two terms have  $AC$  in common. How do we know whether to factor  $A\bar{B}$  from the first two terms or  $AC$  from the last two terms? Actually, we can do both by using the  $A\bar{B}C$  term *twice*. In other words, we can rewrite the expression as:

$$z = A\bar{B}\bar{C} + A\bar{B}C + A\bar{B}C + ABC$$

where we have added an extra term  $A\bar{B}C$ . This is valid and will not change the value of the expression because  $A\bar{B}C + A\bar{B}C = A\bar{B}C$  [theorem (7)]. Now we can factor  $A\bar{B}$  from the first two terms and  $AC$  from the last two terms:

$$\begin{aligned} z &= A\bar{B}(C + \bar{C}) + AC(\bar{B} + B) \\ &= A\bar{B} \cdot 1 + AC \cdot 1 \\ &= A\bar{B} + AC = A(\bar{B} + C) \end{aligned}$$

Of course, this is the same result obtained with method 1. This trick of using the same term twice can always be used. In fact, the same term can be used more than twice if necessary.

## EXAMPLE 4-3

Simplify  $z = \overline{AC}(\overline{ABD}) + \overline{ABC}\overline{D} + \overline{ABC}$ .

**Solution**

First, use DeMorgan's theorem on the first term:

$$z = \overline{AC}(A + \overline{B} + \overline{D}) + \overline{ABC}\overline{D} + \overline{ABC} \quad (\text{step 1})$$

Multiplying out yields

$$z = \overline{ACA} + \overline{ACB} + \overline{ACD} + \overline{ABC}\overline{D} + \overline{ABC} \quad (2)$$

Because  $\overline{A} \cdot A = 0$ , the first term is eliminated:

$$z = \overline{A}\overline{BC} + \overline{ACD} + \overline{ABC}\overline{D} + \overline{ABC} \quad (3)$$

This is the desired SOP form. Now we must look for common factors among the various product terms. The idea is to check for the largest common factor between any two or more product terms. For example, the first and last terms have the common factor  $\overline{BC}$ , and the second and third terms share the common factor  $\overline{AD}$ . We can factor these out as follows:

$$z = \overline{BC}(\overline{A} + A) + \overline{AD}(C + \overline{BC}) \quad (4)$$

Now, because  $\overline{A} + A = 1$ , and  $C + \overline{BC} = C + B$  [theorem (15a)], we have

$$z = \overline{BC} + \overline{AD}(B + C) \quad (5)$$

This same result could have been reached with other choices for the factoring. For example, we could have factored  $C$  from the first, second, and fourth product terms in step 3 to obtain

$$z = C(\overline{A}\overline{B} + \overline{AD} + \overline{A}\overline{B}) + \overline{ABC}\overline{D}$$

The expression inside the parentheses can be factored further:

$$z = C(\overline{B}[\overline{A} + A] + \overline{AD}) + \overline{ABC}\overline{D}$$

Because  $\overline{A} + A = 1$ , this becomes

$$z = C(\overline{B} + \overline{AD}) + \overline{ABC}\overline{D}$$

Multiplying out yields

$$z = \overline{BC} + \overline{ACD} + \overline{ABC}\overline{D}$$

Now we can factor  $\overline{AD}$  from the second and third terms to get

$$z = \overline{BC} + \overline{AD}(C + \overline{BC})$$

If we use theorem (15a), the expression in parentheses becomes  $B + C$ . Thus, we finally have

$$z = \overline{BC} + \overline{AD}(B + C)$$

This is the same result that we obtained earlier, but it took us many more steps. This illustrates why you should look for the largest common factors: it will generally lead to the final expression in the fewest steps.

Example 4-3 illustrates the frustration often encountered in Boolean simplification. Because we have arrived at the same equation (which appears irreducible) by two different methods, it might seem reasonable to conclude that this final equation is the simplest form. In fact, the simplest form of this equation is

$$z = \overline{A}B\overline{D} + \overline{B}C$$

But there is no apparent way to reduce step (5) to reach this simpler version. In this case, we missed an operation earlier in the process that could have led to the simpler form. The question is, “How could we have known that we missed a step?” Later in this chapter, we will examine a mapping technique that will always lead to the simplest SOP form.

#### EXAMPLE 4-4

Simplify the expression  $x = (\overline{A} + B)(A + B + D)\overline{D}$ .

#### Solution

The expression can be put into sum-of-products form by multiplying out all the terms. The result is

$$x = \overline{A}A\overline{D} + \overline{A}B\overline{D} + \overline{A}D\overline{D} + BA\overline{D} + BB\overline{D} + BDD\overline{D}$$

The first term can be eliminated because  $\overline{A}A = 0$ . Likewise, the third and sixth terms can be eliminated because  $D\overline{D} = 0$ . The fifth term can be simplified to  $B\overline{D}$  because  $BB = B$ . This gives us

$$x = \overline{A}B\overline{D} + AB\overline{D} + B\overline{D}$$

We can factor  $B\overline{D}$  from each term to obtain

$$x = B\overline{D}(\overline{A} + A + 1)$$

Clearly, the term inside the parentheses is always 1, so we finally have

$$x = B\overline{D}$$

#### EXAMPLE 4-5

Simplify the circuit of Figure 4-3(a).

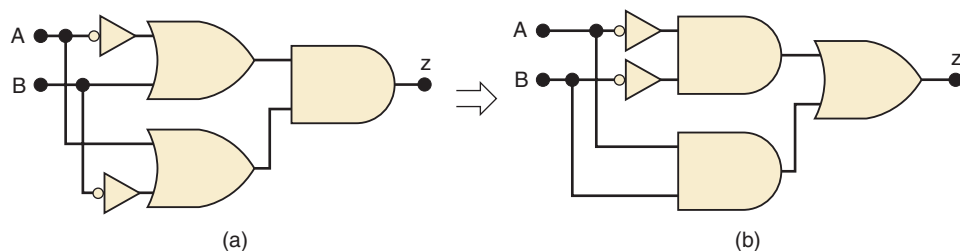


FIGURE 4-3 Example 4-5.

**Solution**

The expression for output  $z$  is

$$z = (\bar{A} + B)(A + \bar{B})$$

Multiplying out to get the sum-of-products form, we obtain

$$z = \bar{A}A + \bar{A}\bar{B} + BA + B\bar{B}$$

We can eliminate  $\bar{A}A = 0$  and  $B\bar{B} = 0$  to end up with

$$z = \bar{A}\bar{B} + AB$$

This expression is implemented in Figure 4-3(b), and if we compare it with the original circuit, we see that both circuits contain the same number of gates and connections. In this case, the simplification process produced an equivalent, but not simpler, circuit.

**EXAMPLE 4-6**

Simplify  $x = \bar{A}\bar{B}C + \bar{A}BD + \bar{C}\bar{D}$ .

**Solution**

You can try, but you will not be able to simplify this expression any further.

**OUTCOME ASSESSMENT QUESTIONS**

- State which of the following expressions are *not* in the sum-of-products form:
  - $R\bar{S}\bar{T} + \bar{R}S\bar{T} + \bar{T}$
  - $\bar{A}\bar{C}\bar{D} + \bar{A}CD$
  - $MN\bar{P} + (M + \bar{N})P$
  - $AB + \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}\bar{C}D$
- Simplify the circuit in Figure 4-1(a) to arrive at the circuit of Figure 4-1(b).
- Change each AND gate in Figure 4-1(a) to a NAND gate. Determine the new expression for  $x$  and simplify it.

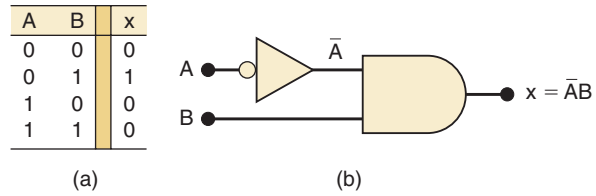
**4-4 DESIGNING COMBINATIONAL LOGIC CIRCUITS****OUTCOMES**

Upon completion of this section, you will be able to:

- Systematically design a circuit to perform any logic function of up to four variables.
- List the steps of the design process.

When the desired output level of a logic circuit is given for all possible input conditions, the results can be conveniently displayed in a truth table. The Boolean expression for the required circuit can then be derived from the truth table. For example, consider Figure 4-4(a), where a truth table is shown for a circuit that has two inputs,  $A$  and  $B$ , and output  $x$ . The table shows that output  $x$  is to be at the 1 level *only* for the case where  $A = 0$

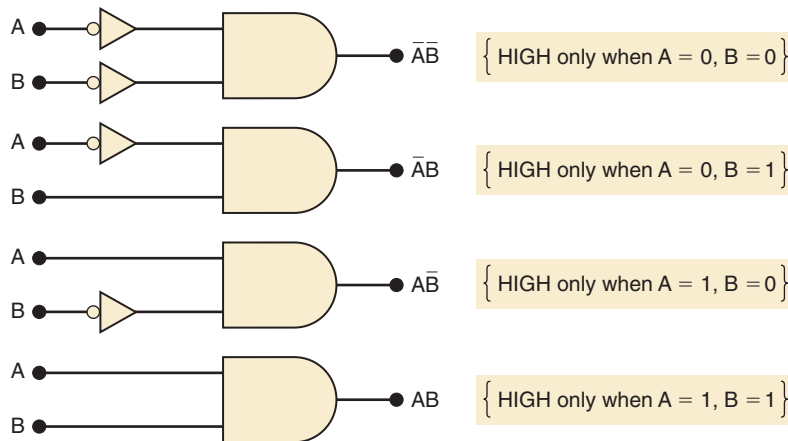
**FIGURE 4-4** Circuit that produces a 1 output only for the  $A = 0, B = 1$  condition.



and  $B = 1$ . It now remains to determine what logic circuit will produce this desired operation. It should be apparent that one possible solution is that shown in Figure 4-4(b). Here an AND gate is used with inputs  $\bar{A}$  and  $B$ , so that  $x = \bar{A} \cdot B$ . Obviously  $x$  will be 1 *only if* both inputs to the AND gate are 1, namely,  $\bar{A} = 1$  (which means that  $A = 0$ ) and  $B = 1$ . For all other values of  $A$  and  $B$ , the output  $x$  will be 0.

A similar approach can be used for the other input conditions. For instance, if  $x$  were to be HIGH only for the  $A = 1, B = 0$  condition, the resulting circuit would be an AND gate with inputs  $A$  and  $\bar{B}$ . In other words, for any of the four possible input conditions, we can generate a HIGH  $x$  output by using an AND gate with appropriate inputs to generate the required AND product. The four different cases are shown in Figure 4-5. Each of the AND gates shown generates an output that is HIGH *only* for one given input condition and the output is LOW for all other conditions. It should be noted that the AND inputs are inverted or not inverted depending on the values that the variables have for the given condition. If the variable is LOW for the given condition, it is inverted before entering the AND gate.

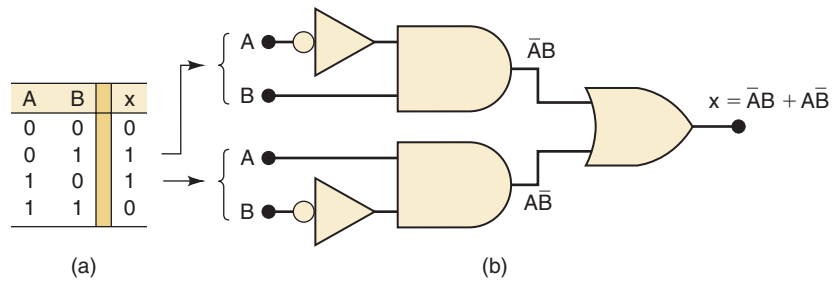
**FIGURE 4-5** An AND gate with appropriate inputs can be used to produce a HIGH output for a specific set of input levels.



Let us now consider the case shown in Figure 4-6(a), where we have a truth table that indicates that the output  $x$  is to be 1 for two different cases:  $A = 0, B = 1$  and  $A = 1, B = 0$ . How can this be implemented? We know that the AND term  $\bar{A} \cdot B$  will generate a 1 only for the  $A = 0, B = 1$  condition, and the AND term  $A \cdot \bar{B}$  will generate a 1 for the  $A = 1, B = 0$  condition. Because  $x = 1$  for *either* condition, it should be clear that these terms should be ORed together to produce the desired output,  $x$ . This implementation is shown in Figure 4-6(b), where the resulting expression for the output is  $x = \bar{A}B + A\bar{B}$ .

In this example, an AND term is generated for each case in the table where the output  $x = 1$ . The AND gate outputs are then ORed together to produce the total output  $x$ , which will be 1 when either AND term is 1. This same procedure can be extended to examples with more than two inputs. Consider the truth table for a three-input circuit (Table 4-1). Here there

**FIGURE 4-6** Each set of input conditions that is to produce a 1 output is implemented by a separate AND gate. The AND outputs are ORed to produce the final output.



are three cases where the output  $x = 1$ . The required AND term for each of these cases is shown. Again, note that for each case where a variable is 0, it appears inverted in the AND term. The sum-of-products expression for  $x$  is obtained by ORing the three AND terms.

$$x = \bar{A}B\bar{C} + \bar{A}BC + ABC$$

**TABLE 4-1** Generating AND terms.

A	B	C	x	
0	0	0	0	
0	0	1	0	
0	1	0	1	$\rightarrow \bar{A}B\bar{C}$
0	1	1	1	$\rightarrow \bar{A}BC$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	$\rightarrow ABC$

## Complete Design Procedure

Any logic problem can be solved using the following step-by-step procedure.

1. Interpret the problem and set up a truth table to describe its operation.
2. Write the AND (product) term for each case where the output is 1.
3. Write the sum-of-products (SOP) expression for the output.
4. Simplify the output expression if possible.
5. Implement the circuit for the final, simplified expression.

The following example illustrates the complete design procedure.

### EXAMPLE 4-7

Design a logic circuit that has three inputs,  $A$ ,  $B$ , and  $C$ , and whose output will be HIGH only when a majority of the inputs are HIGH.

#### Solution

**Step 1.** Set up the truth table.

On the basis of the problem statement, the output  $x$  should be 1 whenever two or more inputs are 1; for all other cases, the output should be 0 (Table 4-2).



**TABLE 4-2** Example 4-7:  
truth table with AND terms.

A	B	C	x	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\rightarrow \bar{A}BC$
1	0	0	0	
1	0	1	1	$\rightarrow A\bar{B}C$
1	1	0	1	$\rightarrow AB\bar{C}$
1	1	1	1	$\rightarrow ABC$

**Step 2.** Write the AND term for each case where the output is a 1.

There are four such cases. The AND terms are shown next to the truth table (Table 4-2). Again note that each AND term contains each input variable in either inverted or noninverted form.

**Step 3.** Write the sum-of-products expression for the output.

$$x = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

**Step 4.** Simplify the output expression.

This expression can be simplified in several ways. Perhaps the quickest way is to realize that the last term  $ABC$  has two variables in common with each of the other terms. Thus, we can use the  $ABC$  term to pair with each of the other terms. The expression is rewritten with the  $ABC$  term occurring three times (recall from Example 4-2 that this is legal in Boolean algebra):

$$x = \bar{A}BC + ABC + A\bar{B}C + ABC + AB\bar{C} + ABC$$

Factoring the appropriate pairs of terms, we have

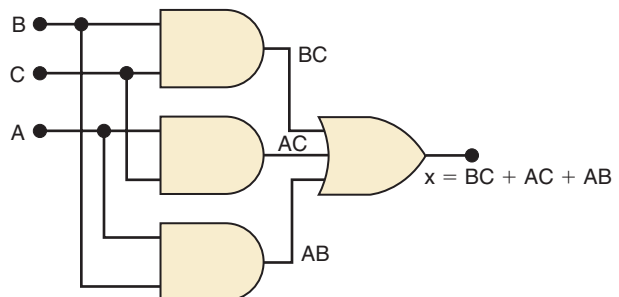
$$x = BC(\bar{A} + A) + AC(\bar{B} + B) + AB(\bar{C} + C)$$

Each term in parentheses is equal to 1, so we have

$$x = BC + AC + AB$$

**Step 5.** Implement the circuit for the final expression.

This expression is implemented in Figure 4-7. Since the expression is in SOP form, the circuit consists of a group of AND gates working into a single OR gate.

**FIGURE 4-7** Example 4-7.

## EXAMPLE 4-8

Refer to Figure 4-8(a), where an analog-to-digital converter is monitoring the dc voltage ( $V_B$ ) of a 12-V storage battery on an orbiting spaceship. The converter's output is a four-bit binary number,  $ABCD$ , corresponding to the battery voltage in steps of 1 V, with  $A$  as the MSB. The converter's binary outputs are fed to a logic circuit that is to produce a HIGH output as long as the binary value is greater than  $0110_2 = 6_{10}$ ; that is, the battery voltage is greater than 6 V. Design this logic circuit.

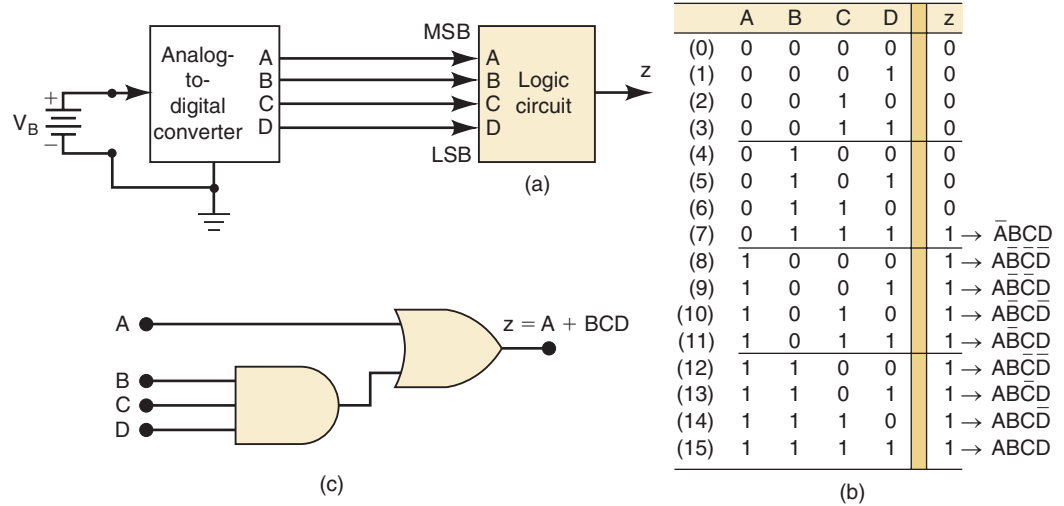


FIGURE 4-8 Example 4-8.

### Solution

The truth table is shown in Figure 4-8(b). For each case in the truth table, we have indicated the decimal equivalent of the binary number represented by the  $ABCD$  combination.

The output  $z$  is set equal to 1 for all those cases where the binary number is greater than 0110. For all other cases,  $z$  is set equal to 0. This truth table gives us the following sum-of-products expression:

$$z = \overline{A}BCD + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}CD + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}BC\overline{D} + \overline{A}BCD + A\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C}D + A\overline{B}C\overline{D} + A\overline{B}CD + AB\overline{C}\overline{D} + AB\overline{C}D + ABC\overline{D} + ABCD$$

Simplification of this expression will be a formidable task, but with a little care it can be accomplished. The step-by-step process involves factoring and eliminating terms of the form  $A + \overline{A}$ :

$$\begin{aligned} z &= \overline{A}BCD + \overline{A}\overline{B}\overline{C}(\overline{D} + D) + \overline{A}\overline{B}\overline{C}(D) + \overline{A}\overline{B}C(\overline{D} + D) + \overline{A}\overline{B}C(D) + \overline{A}B\overline{C}(\overline{D} + D) + \overline{A}B\overline{C}(D) + \overline{A}BC(\overline{D} + D) + \overline{A}BC(D) \\ &= \overline{A}BCD + \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + \overline{A}\overline{B}C + \overline{A}BC + \overline{A}BC \\ &= \overline{A}BCD + \overline{A}\overline{B}(\overline{C} + C) + \overline{A}B(\overline{C} + C) \\ &= \overline{A}BCD + \overline{A}\overline{B} + \overline{A}B \\ &= \overline{A}BCD + A(\overline{B} + B) \\ &= \overline{A}BCD + A \end{aligned}$$

This can be reduced further by invoking theorem (15a), which says that  $x + \overline{x}y = x + y$ . In this case  $x = A$  and  $y = BCD$ . Thus,

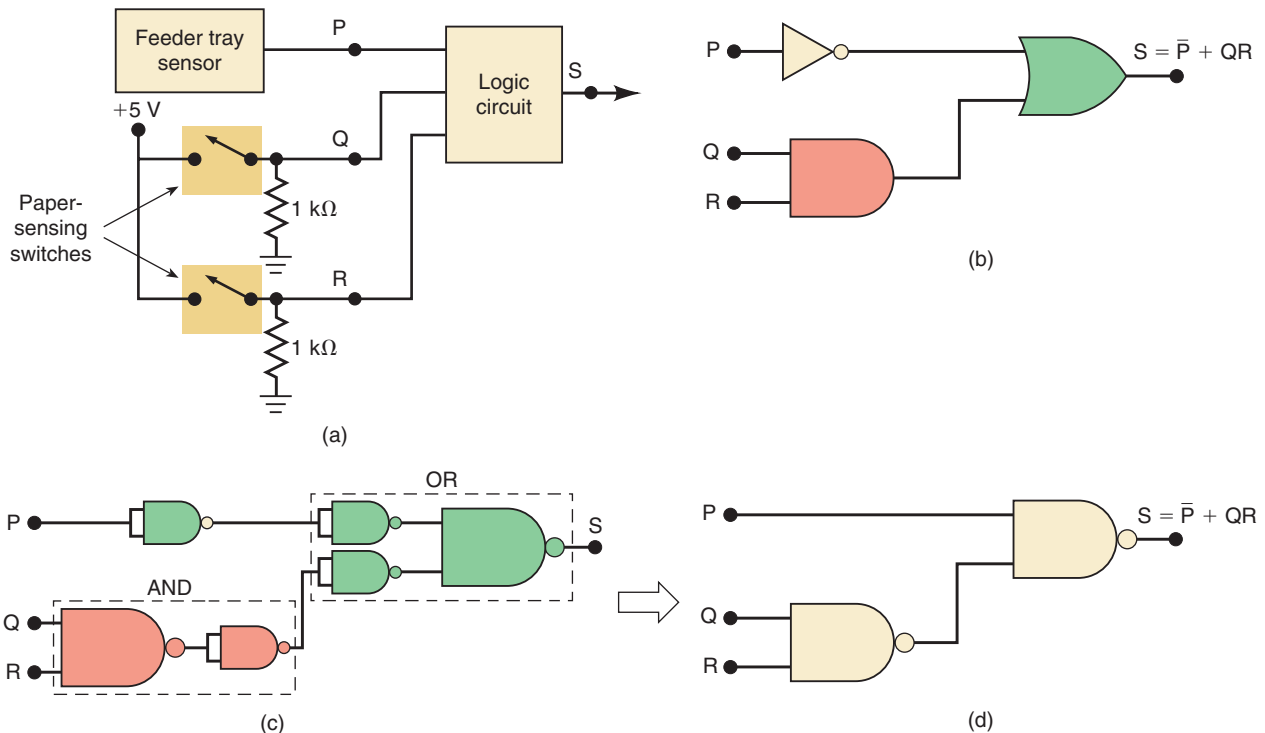
$$z = \overline{A}BCD + A = BCD + A$$

This final expression is implemented in Figure 4-8(c).

As this example demonstrates, the algebraic simplification method can be quite lengthy when the original expression contains a large number of terms. This is a limitation that is not shared by the Karnaugh mapping method, as we will see later.

**EXAMPLE 4-9**

Refer to Figure 4-9(a). In a simple copy machine, a stop signal,  $S$ , is to be generated to stop the machine operation and energize an indicator light whenever either of the following conditions exists: (1) there is no paper in the paper feeder tray; or (2) the two microswitches in the paper path are activated, indicating a jam in the paper path. The presence of paper in the feeder tray is indicated by a HIGH at logic signal  $P$ . Each of the microswitches produces a logic signal ( $Q$  and  $R$ ) that goes HIGH whenever paper is passing over the switch to activate it. Design the logic circuit to produce a HIGH at output signal  $S$  for the stated conditions, and implement it using the 74HC00 CMOS quad two-input NAND chip.



**FIGURE 4-9** Example 4-9.

**Solution**

We will use the five-step process used in Example 4-7. The truth table is shown in Table 4-3. The  $S$  output will be a logic 1 whenever  $P = 0$  because this indicates no paper in the feeder tray.  $S$  will also be a 1 for the two cases where  $Q$  and  $R$  are both 1, indicating a paper jam. As the table shows, there are five different input conditions that produce a HIGH output. **(Step 1)**

**TABLE 4-3** Example 4-9:  
Truth table with AND terms.

<i>P</i>	<i>Q</i>	<i>R</i>	<i>S</i>	
0	0	0	1	$\overline{P}\overline{Q}\overline{R}$
0	0	1	1	$\overline{P}\overline{Q}R$
0	1	0	1	$\overline{P}Q\overline{R}$
0	1	1	1	$\overline{P}QR$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	$PQR$

The AND terms for each of these cases are shown. (Step 2)  
The sum-of-products expression becomes

$$S = \overline{P}\overline{Q}\overline{R} + \overline{P}\overline{Q}R + \overline{P}Q\overline{R} + \overline{P}QR + PQR \quad \text{(Step 3)}$$

We can begin the simplification by factoring out  $\overline{P}\overline{Q}$  from terms 1 and 2 and by factoring out  $\overline{P}Q$  from terms 3 and 4:

$$S = \overline{P}\overline{Q}(\overline{R} + R) + \overline{P}Q(\overline{R} + R) + PQR$$

Now we can eliminate the  $\overline{R} + R$  terms because they equal 1:

$$S = \overline{P}\overline{Q} + \overline{P}Q + PQR$$

Factoring  $\overline{P}$  from terms 1 and 2 allows us to eliminate  $Q$  from these terms:

$$S = \overline{P} + PQR$$

Here we can apply theorem (15b) ( $\overline{x} + xy = \overline{x} + y$ ) to obtain

$$S = \overline{P} + QR \quad \text{(Step 4)}$$

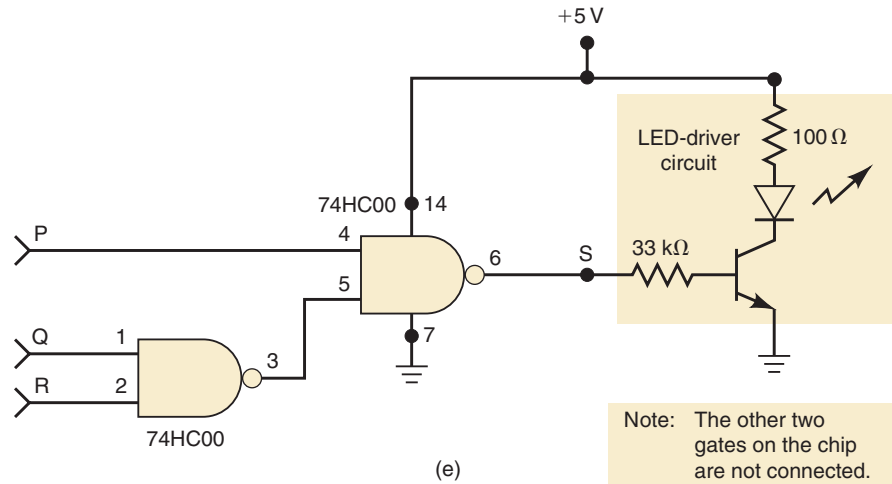
As a double check of this simplified Boolean equation, let's see if it matches the truth table that we started out with. This equation says that the output  $S$  will be HIGH whenever  $P$  is LOW OR both  $Q$  AND  $R$  are HIGH. Look at Table 4-3 and observe that the output is HIGH for all four cases when  $P$  is LOW.  $S$  is also HIGH when  $Q$  AND  $R$  are both HIGH, regardless of the state of  $P$ . This agrees with the equation.

The AND/OR implementation for this circuit is shown in Figure 4-9(b). (Step 5)

To implement this circuit using the 74HC00 quad two-input NAND chip, we must convert each gate and the INVERTER by substituting their NAND-gate equivalents (per Section 3-12). This is shown in Figure 4-9(c). Clearly, we can eliminate two pairs of double inverters to produce the NAND-gate implementation shown in Figure 4-9(d).

The final wired-up circuit is obtained by connecting two of the NAND gates on the 74HC00 chip. This CMOS chip has the same gate configuration and pin numbers as the TTL 74LS00 chip of Figure 3-31. Figure 4-10 shows the wired-up circuit with pin numbers, including the +5V and GROUND pins. It also includes an output driver transistor and LED to indicate the state of output  $S$ .

**FIGURE 4-10** Circuit of Figure 4-9(d) implemented using 74HC00 NAND chip.



### OUTCOME ASSESSMENT QUESTIONS

1. Write the sum-of-products expression for a circuit with four inputs and an output that is to be HIGH only when input *A* is LOW at the same time that exactly two other inputs are LOW.
2. Implement the expression of question 1 using all four-input NAND gates. How many are required?
3. List the steps of the systematic design process.

## 4-5 KARNAUGH MAP METHOD

### OUTCOMES

Upon completion of this section, you will be able to:

- Identify “Don’t care” conditions and use them in truth tables.
- Use K maps to generate the simplest SOP form from a truth table.

The **Karnaugh map (K map)** is a graphical tool used to simplify a logic equation or to convert a truth table to its corresponding logic circuit in a simple, orderly process. Although a K map can be used for problems involving any number of input variables, its practical usefulness is limited to five or six variables. The following discussion will be limited to problems with up to four inputs because even five- and six-input problems are too involved and are best done by a computer program.

### Karnaugh Map Format

The K map, like a truth table, is a means for showing the relationship between logic inputs and the desired output. Figure 4-11 shows three examples of K maps for two, three, and four variables, together with the corresponding truth tables. These examples illustrate the following important points:

1. The truth table gives the value of output *X* for each combination of input values. The K map gives the same information in a different

**FIGURE 4-11** Karnaugh maps and truth tables for (a) two, (b) three, and (c) four variables.

A	B	X
0	0	1 → $\bar{A}\bar{B}$
0	1	0
1	0	0
1	1	1 → $AB$

$$\left\{ x = \bar{A}\bar{B} + AB \right\}$$

(a)

	$\bar{B}$	B
$\bar{A}$	1	0
A	0	1

A	B	C	X
0	0	0	1 → $\bar{A}\bar{B}\bar{C}$
0	0	1	1 → $\bar{A}\bar{B}C$
0	1	0	1 → $\bar{A}B\bar{C}$
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1 → $AB\bar{C}$
1	1	1	0

$$\left\{ X = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + AB\bar{C} \right\}$$

(b)

	$\bar{C}$	C
$\bar{A}\bar{B}$	1	1
$\bar{A}B$	1	0
$AB$	1	0
$A\bar{B}$	0	0

A	B	C	D	X
0	0	0	0	0
0	0	0	1	1 → $\bar{A}\bar{B}\bar{C}D$
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1 → $\bar{A}B\bar{C}D$
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1 → $AB\bar{C}D$
1	1	1	0	0
1	1	1	1	1 → $ABCD$

$$\left\{ X = \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}D + AB\bar{C}D + ABCD \right\}$$

(c)

	$\bar{C}D$	$\bar{C}\bar{D}$	$CD$	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	0	0
$\bar{A}B$	0	1	0	0
$AB$	0	1	1	0
$A\bar{B}$	0	0	0	0

format. Each case in the truth table corresponds to a square in the K map. For example, in Figure 4-11(a), the  $A = 0, B = 0$  condition in the truth table corresponds to the  $\bar{A}\bar{B}$  square in the K map. Because the truth table shows  $X = 1$  for this case, a 1 is placed in the  $\bar{A}\bar{B}$  square in the K map. Similarly, the  $A = 1, B = 1$  condition in the truth table corresponds to the  $AB$  square of the K map. Because  $X = 1$  for this case, a 1 is placed in the  $AB$  square. All other squares are filled with 0s. This same idea is used in the three- and four-variable maps shown in the figure.

- The K-map squares are labeled so that horizontally adjacent squares differ only in one variable. For example, the upper left-hand square in the four-variable map is  $\bar{A}\bar{B}\bar{C}\bar{D}$ , while the square immediately to its right is  $\bar{A}\bar{B}\bar{C}D$  (only the  $D$  variable is different). Similarly, vertically adjacent squares differ only in one variable. For example, the upper left-hand square is  $\bar{A}\bar{B}\bar{C}\bar{D}$ , while the square directly below it is  $\bar{A}\bar{B}\bar{C}D$  (only the  $B$  variable is different).

Note that each square in the top row is considered to be adjacent to a corresponding square in the bottom row. For example, the  $\bar{A}\bar{B}\bar{C}\bar{D}$  square in the top row is adjacent to the  $\bar{A}\bar{B}C\bar{D}$  square in the bottom

row because they differ only in the  $A$  variable. You can think of the top of the map as being wrapped around to touch the bottom of the map. Similarly, squares in the leftmost column are adjacent to corresponding squares in the rightmost column.

3. In order for vertically and horizontally adjacent squares to differ in only one variable, the top-to-bottom labeling must be done in the order shown:  $\overline{A}B, \overline{A}\overline{B}, AB, A\overline{B}$ . The same is true of the left-to-right labeling:  $\overline{C}D, \overline{C}\overline{D}, CD, C\overline{D}$ .
4. Once a K map has been filled with 0s and 1s, the sum-of-products expression for the output  $X$  can be obtained by ORing together those squares that contain a 1. In the three-variable map of Figure 4-11(b), the  $\overline{A}\overline{B}\overline{C}$ ,  $\overline{A}B\overline{C}$ ,  $A\overline{B}\overline{C}$ , and  $ABC$  squares contain a 1, so that  $X = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$ .

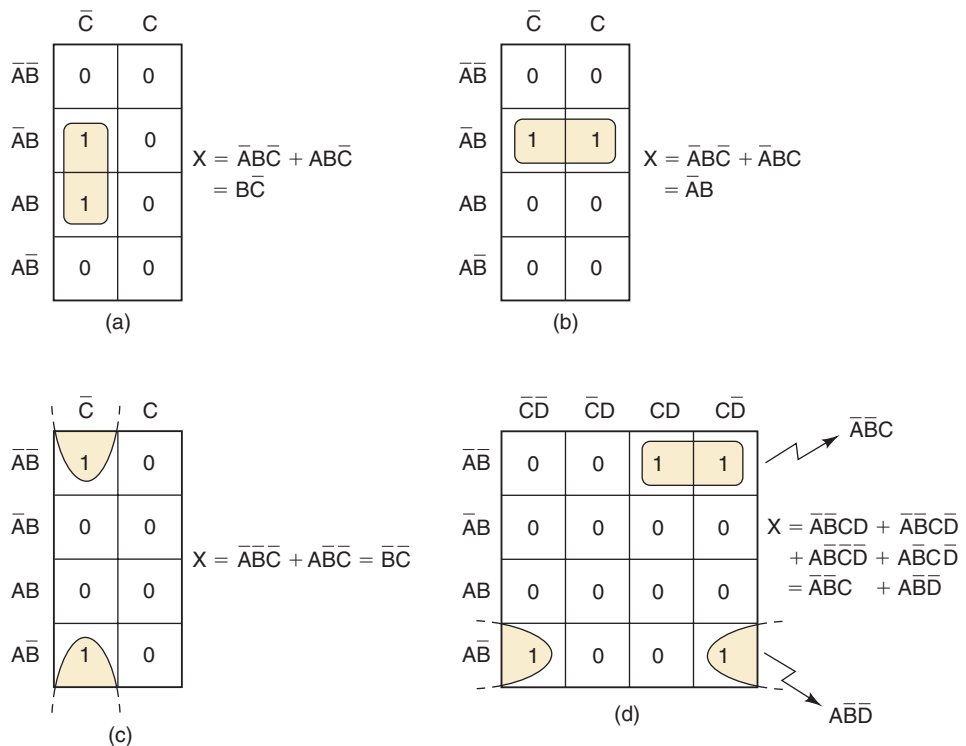
### Looping

The expression for output  $X$  can be simplified by properly combining those squares in the K map that contain 1s. The process for combining these 1s is called **looping**.

### Looping Groups of Two (Pairs)

Figure 4-12(a) is the K map for a particular three-variable truth table. This map contains a pair of 1s that are vertically adjacent to each other; the first represents  $\overline{A}B\overline{C}$ , and the second represents  $AB\overline{C}$ . Note that in these two terms only the  $A$  variable appears in both normal and complemented (inverted) form, while  $B$  and  $\overline{C}$  remain unchanged. These two terms can be

**FIGURE 4-12** Examples of looping pairs of adjacent 1s.



looped (combined) to give a resultant that eliminates the  $A$  variable because it appears in both uncomplemented and complemented forms. This is easily proved as follows:

$$\begin{aligned} X &= \overline{A}B\overline{C} + AB\overline{C} \\ &= B\overline{C}(\overline{A} + A) \\ &= B\overline{C}(1) = B\overline{C} \end{aligned}$$

This same principle holds true for any pair of vertically or horizontally adjacent 1s. Figure 4-12(b) shows an example of two horizontally adjacent 1s. These two can be looped and the  $C$  variable eliminated because it appears in both its uncomplemented and complemented forms to give a resultant of  $X = \overline{A}B$ .

Another example is shown in Figure 4-12(c). In a K map, the top row and bottom row of squares are considered to be adjacent. Thus, the two 1s in this map can be looped to provide a resultant of  $\overline{A}B\overline{C} + AB\overline{C} = B\overline{C}$ .

Figure 4-12(d) shows a K map that has two pairs of 1s that can be looped. The two 1s in the top row are horizontally adjacent. The two 1s in the bottom row are also adjacent because, in a K map, the leftmost column and the rightmost column of squares are considered to be adjacent. When the top pair of 1s is looped, the  $D$  variable is eliminated (because it appears as both  $D$  and  $\overline{D}$ ) to give the term  $\overline{A}BC$ . Looping the bottom pair eliminates the  $C$  variable to give the term  $A\overline{B}\overline{D}$ . These two terms are ORed to give the final result for  $X$ .

To summarize:

**Looping a pair of adjacent 1s in a K map eliminates the variable that appears in complemented and uncomplemented form.**

### Looping Groups of Four (Quads)

A K map may contain a group of four 1s that are adjacent to each other. This group is called a *quad*. Figure 4-13 shows several examples of quads. In Figure 4-13(a), the four 1s are vertically adjacent and in Figure 4-13(b), they are horizontally adjacent. The K map in Figure 4-13(c) contains four 1s in a square, and they are considered adjacent to each other. The four 1s in Figure 4-13(d) are also adjacent, as are those in Figure 4-13(e), because, as pointed out earlier, the top and bottom rows are considered to be adjacent to each other, as are the leftmost and rightmost columns.

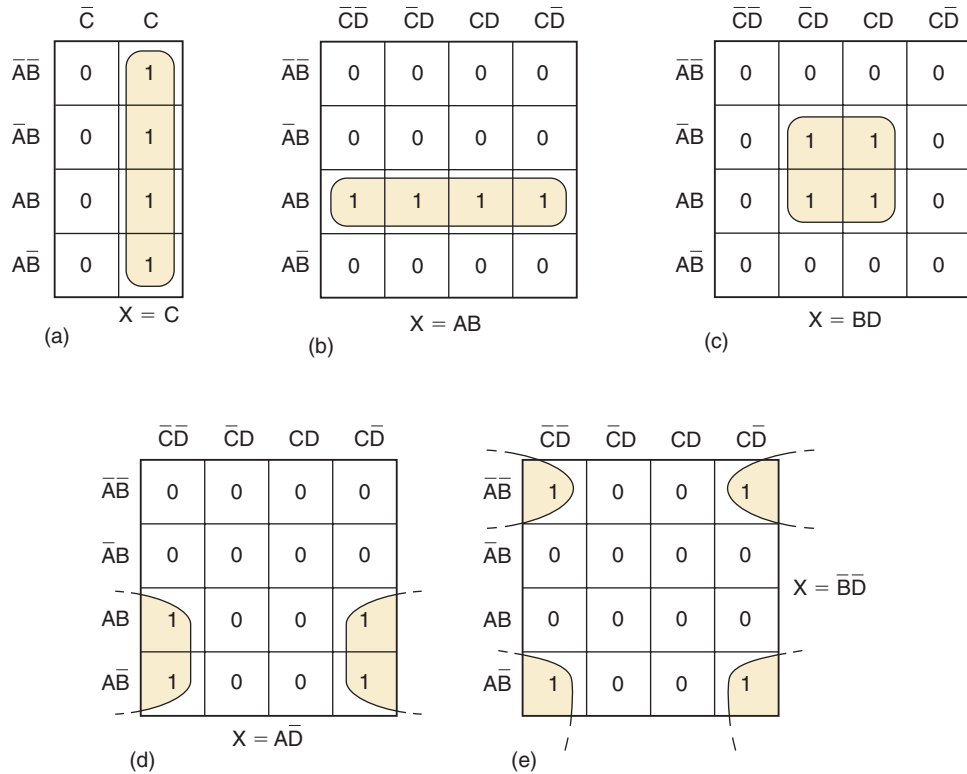
When a quad is looped, the resultant term will contain only the variables that do not change form for all the squares in the quad. For example, in Figure 4-13(a), the four squares that contain a 1 are  $\overline{A}\overline{B}C$ ,  $\overline{A}BC$ ,  $ABC$ , and  $A\overline{B}C$ . Examination of these terms reveals that only the variable  $C$  remains unchanged (both  $A$  and  $B$  appear in complemented and uncomplemented form). Thus, the resultant expression for  $X$  is simply  $X = C$ . This can be proved as follows:

$$\begin{aligned} X &= \overline{A}\overline{B}C + \overline{A}BC + ABC + A\overline{B}C \\ &= \overline{A}C(\overline{B} + B) + AC(B + \overline{B}) \\ &= \overline{A}C + AC \\ &= C(\overline{A} + A) = C \end{aligned}$$

As another example, consider Figure 4-13(d), where the four squares containing 1s are  $AB\overline{C}\overline{D}$ ,  $\overline{A}B\overline{C}\overline{D}$ ,  $AB\overline{C}D$ , and  $\overline{A}B\overline{C}D$ . Examination of these



**FIGURE 4-13** Examples of looping groups of four 1s (quads).



terms indicates that only the variables  $A$  and  $\bar{D}$  remain unchanged, so that the simplified expression for  $X$  is

$$X = A\bar{D}$$

This can be proved in the same manner that was used above. The reader should check each of the other cases in Figure 4-13 to verify the indicated expressions for  $X$ .

To summarize:

**Looping a quad of adjacent 1s eliminates the two variables that appear in both complemented and uncomplemented form.**

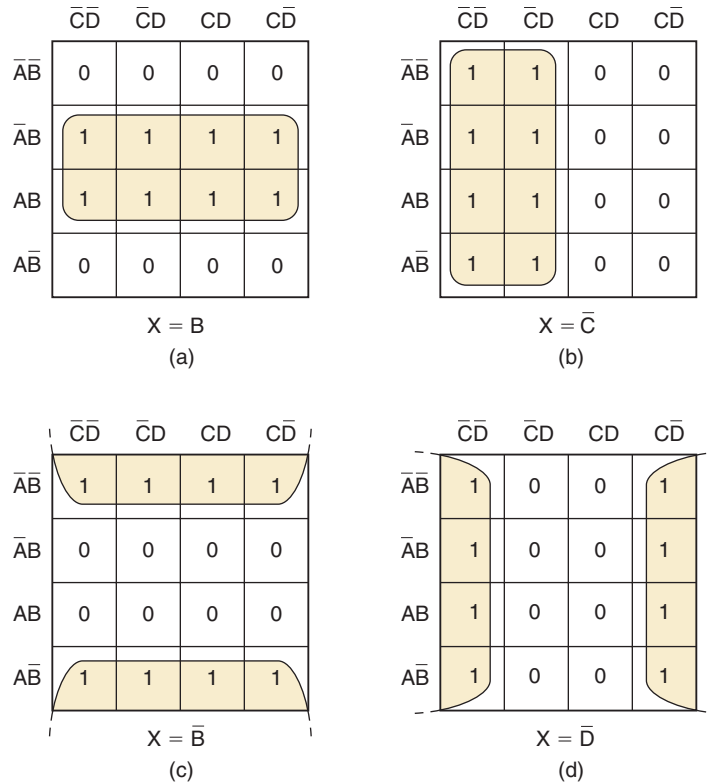
### Looping Groups of Eight (Octets)

A group of eight 1s that are adjacent to one another is called an *octet*. Several examples of octets are shown in Figure 4-14. When an octet is looped in a four-variable map, three of the four variables are eliminated because only one variable remains unchanged. For example, examination of the eight looped squares in Figure 4-14(a) shows that only the variable  $B$  is in the same form for all eight squares: the other variables appear in complemented and uncomplemented form. Thus, for this map,  $X = B$ . The reader can verify the results for the other examples in Figure 4-14.

To summarize:

**Looping an octet of adjacent 1s eliminates the three variables that appear in both complemented and uncomplemented form.**

**FIGURE 4-14** Examples of looping groups of eight 1s (octets).



### Complete Simplification Process

We have seen how looping of pairs, quads, and octets on a K map can be used to obtain a simplified expression. We can summarize the rule for loops of *any* size:

**When a variable appears in both complemented and uncomplemented form within a loop, that variable is eliminated from the expression. Variables that are the same for all squares of the loop must appear in the final expression.**

It should be clear that a larger loop of 1s eliminates more variables. To be exact, a loop of two eliminates one variable, a loop of four eliminates two variables, and a loop of eight eliminates three. This principle will now be used to obtain a simplified logic expression from a K map that contains any combination of 1s and 0s.

The procedure will first be outlined and then applied to several examples. The steps below are followed in using the K-map method for simplifying a Boolean expression:

- Step 1** Construct the K map and place 1s in those squares corresponding to the 1s in the truth table. Place 0s in the other squares.
- Step 2** Examine the map for adjacent 1s and loop those 1s that are *not* adjacent to any other 1s. These are called *isolated* 1s.
- Step 3** Next, look for those 1s that are adjacent to only one other 1. Loop *any* pair containing such a 1.
- Step 4** Loop any octet even if it contains some 1s that have already been looped.
- Step 5** Loop any quad that contains one or more 1s that have not already been looped, *making sure to use the minimum number of loops.*

**Step 6** Loop any pairs necessary to include any 1s that have not yet been looped, *making sure to use the minimum number of loops*.

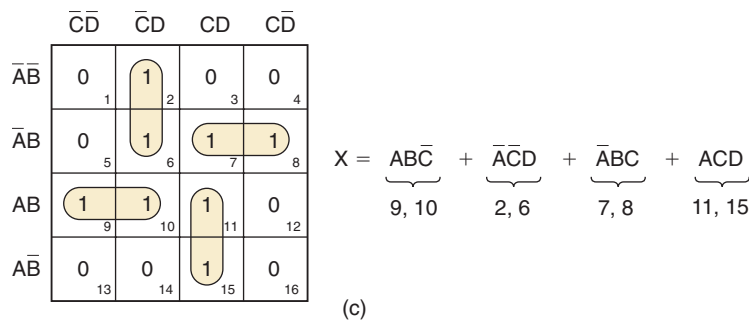
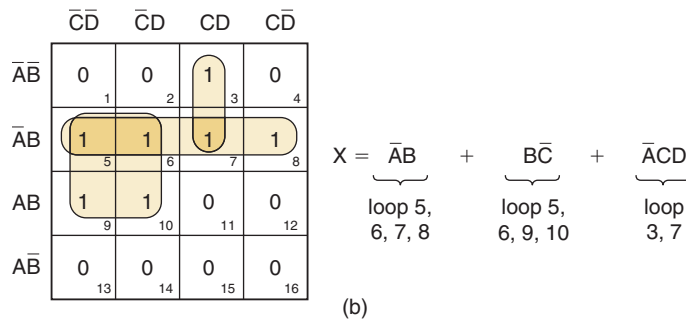
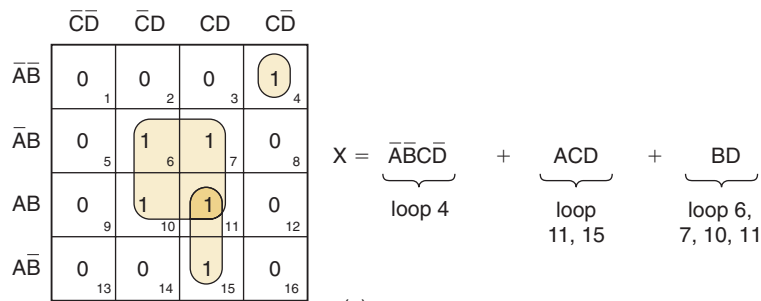
**Step 7** Form the OR sum of all the terms generated by each loop.

These steps will be followed exactly and referred to in the following examples. In each case, the resulting logic expression will be in its simplest sum-of-products form.

### EXAMPLE 4-10

Figure 4-15(a) shows the K map for a four-variable problem. We will assume that Step 1 (construct a K map from the problem truth table) has been completed. The squares are numbered for convenience in identifying each loop. Use steps 2–7 of the simplification process to reduce the K map to an SOP expression.

**FIGURE 4-15** Examples 4-10 to 4-12.



### Solution

**Step 2** Square 4 is the only square containing a 1 that is not adjacent to any other 1. It is looped and is referred to as loop 4.

**Step 3** Square 15 is adjacent *only* to square 11. This pair is looped and referred to as loop 11, 15.

**Step 4** There are no octets.

**Step 5** Squares 6, 7, 10, and 11 form a quad. This quad is looped (loop 6, 7, 10, 11). Note that square 11 is used again, even though it was part of loop 11, 15.

**Step 6** All 1s have already been looped.

**Step 7** Each loop generates a term in the expression for  $X$ . Loop 4 is simply  $\overline{A}\overline{B}\overline{C}\overline{D}$ . Loop 11, 15 is  $ACD$  (the  $B$  variable is eliminated). Loop 6, 7, 10, 11 is  $BD$  ( $A$  and  $C$  are eliminated).

**EXAMPLE 4-11**

Consider the K map in Figure 4-15(b). Once again, we can assume that step 1 has already been performed. Simplify.

**Solution**

**Step 2** There are no isolated 1s.

**Step 3** The 1 in square 3 is adjacent *only* to the 1 in square 7. Looping this pair (loop 3, 7) produces the term  $\overline{A}CD$ .

**Step 4** There are no octets.

**Step 5** There are two quads. Squares 5, 6, 7, and 8 form one quad. Looping this quad produces the term  $\overline{A}B$ . The second quad is made up of squares 5, 6, 9, and 10. This quad is looped because it contains two squares that have not been looped previously. Looping this quad produces  $B\overline{C}$ .

**Step 6** All 1s have already been looped.

**Step 7** The terms generated by the three loops are ORed together to obtain the expression for  $X$ .

**EXAMPLE 4-12**

Consider the K map in Figure 4-15(c). Simplify.

**Solution**

**Step 2** There are no isolated 1s.

**Step 3** The 1 in square 2 is adjacent only to the 1 in square 6. This pair is looped to produce  $\overline{A}\overline{C}D$ . Similarly, square 9 is adjacent only to square 10. Looping this pair produces  $AB\overline{C}$ . Likewise, loop 7, 8 and loop 11, 15 produce the terms  $\overline{A}BC$  and  $ACD$ , respectively.

**Step 4** There are no octets.

**Step 5** There is one quad formed by squares 6, 7, 10, and 11. This quad, however, is *not* looped because all the 1s in the quad have been included in other loops.

**Step 6** All 1s have already been looped.

**Step 7** The expression for  $X$  is shown in the figure.

**EXAMPLE 4-13**

Consider the two K map loopings in Figure 4-16. Is one better than the other?

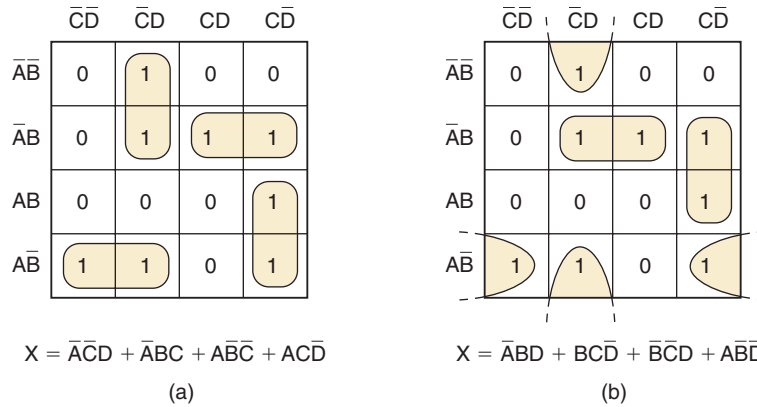
**Solution**

**Step 2** There are no isolated 1s.

**Step 3** There are no 1s that are adjacent to only one other 1.

**Step 4** There are no octets.

**FIGURE 4-16** The same K map with two equally good solutions.



**Step 5** There are no quads.

**Step 6 and 7** There are many possible pairs. The looping must use the minimum number of loops to account for all the 1s. For this map, there are *two* possible loopings, which require only four looped pairs. Figure 4-16(a) shows one solution and its resultant expression. Figure 4-16(b) shows the other. Note that both expressions are of the same complexity, and so neither is better than the other.

### Filling a K Map from an Output Expression

When the desired output is presented as a Boolean expression instead of a truth table, the K map can be filled by using the following steps:

1. Get the expression into SOP form if it is not already in that form.
2. For each product term in the SOP expression, place a 1 in each K-map square whose label contains the same combination of input variables. Place a 0 in all other squares.

The following example illustrates this procedure.

#### EXAMPLE 4-14

Use a K map to simplify  $y = \bar{C}(\bar{A}\bar{B}\bar{D} + D) + \bar{A}BC + \bar{D}$ .

#### Solution

1. Multiply out the first term to get  $y = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{C}D + \bar{A}BC + \bar{D}$ , which is now in SOP form.
2. For the  $\bar{A}\bar{B}\bar{C}\bar{D}$  term, simply put a 1 in the  $\bar{A}\bar{B}\bar{C}\bar{D}$  square of the K map (Figure 4-17). For the  $\bar{C}D$  term, place a 1 in all squares with  $\bar{C}D$  in their labels, that is,  $\bar{A}\bar{B}\bar{C}D$ ,  $\bar{A}B\bar{C}D$ ,  $AB\bar{C}D$ ,  $A\bar{B}\bar{C}D$ . For the  $\bar{A}BC$  term, place a 1 in all squares that have an  $\bar{A}BC$  in their labels, that is,  $\bar{A}BCD$ ,  $\bar{A}BC\bar{D}$ . For the  $\bar{D}$  term, place a 1 in all squares that have a  $\bar{D}$  in their labels, that is, all squares in the leftmost and rightmost columns.

The K map is now filled and can be looped for simplification. Verify that proper looping produces  $y = \bar{A}B + \bar{C} + \bar{D}$ .

**FIGURE 4-17** Example 4-14.

	$\bar{C}\bar{D}$	$\bar{C}D$	$CD$	$C\bar{D}$
$\bar{A}\bar{B}$	1	1	0	1
$\bar{A}B$	1	1	0	1
$AB$	1	1	0	1
$A\bar{B}$	1	1	1	1

$y = \bar{A}\bar{B} + \bar{C} + \bar{D}$

### Don't-Care Conditions

Some logic circuits can be designed so that there are certain input conditions for which there are no specified output levels, usually because these input conditions will never occur. In other words, there will be certain combinations of input levels where we “don’t care” whether the output is 1 or 0. This is illustrated in the truth table of Figure 4-18(a).

**FIGURE 4-18** “Don’t-care” conditions should be changed to 0 or 1 to produce K-map looping that yields the simplest expression.

A	B	C	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	x
1	0	0	x
1	0	1	1
1	1	0	1
1	1	1	1

} “don’t care”

	$\bar{C}$	$C$
$\bar{A}\bar{B}$	0	0
$\bar{A}B$	0	x
$AB$	1	1
$A\bar{B}$	x	1

(a)

⇒

	$\bar{C}$	$C$
$\bar{A}\bar{B}$	0	0
$\bar{A}B$	0	0
$AB$	1	1
$A\bar{B}$	1	1

(b)

→ z = A

(c)

Here the output  $z$  is not specified as either 0 or 1 for the conditions  $A, B, C = 1, 0, 0$  and  $A, B, C = 0, 1, 1$ . Instead, an  $x$  is shown for these conditions. The  $x$  represents the **don’t-care condition**. A don’t-care condition can come about for several reasons, the most common being that in some situations certain input combinations can never occur, and so there is no specified output for these conditions.

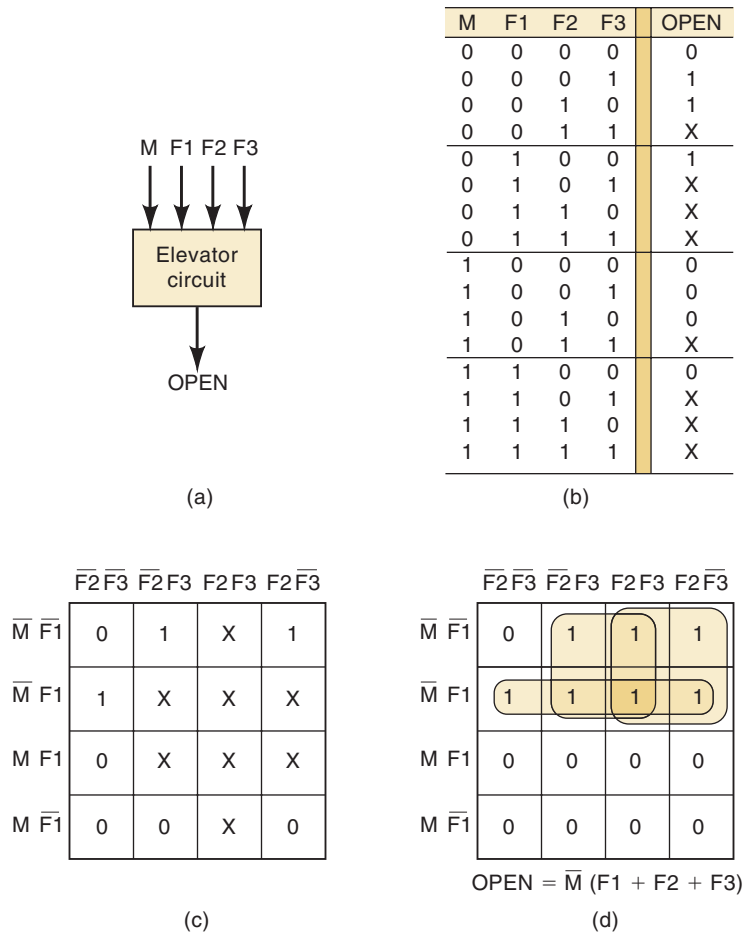
A circuit designer is free to make the output for any don’t-care condition either a 0 or a 1 to produce the simplest output expression. For example, the K map for this truth table is shown in Figure 4-18(b) with an  $x$  placed in the  $\bar{A}\bar{B}C$  and  $A\bar{B}\bar{C}$  squares. The designer here would be wise to change the  $x$  in the  $\bar{A}\bar{B}C$  square to a 1 and the  $x$  in the  $A\bar{B}\bar{C}$  square to a 0 because this would produce a quad that can be looped to produce  $z = A$ , as shown in Figure 4-18(c).

Whenever don’t-care conditions occur, we must decide which  $x$  to change to 0 and which to 1 to produce the best K-map looping (i.e., the simplest expression). This decision is not always an easy one. Several end-of-chapter problems will provide practice in dealing with don’t-care cases. Here’s another example.

#### EXAMPLE 4-15

Let’s design a logic circuit that controls an elevator door in a three-story building. The circuit in Figure 4-19(a) has four inputs.  $M$  is a logic signal that indicates when the elevator is moving ( $M = 1$ ) or stopped ( $M = 0$ ).  $F1$ ,  $F2$ , and  $F3$  are floor indicator signals that are normally LOW, and they go HIGH only when the elevator is positioned at the level of that particular

**FIGURE 4-19** Example 4-15.



floor. For example, when the elevator is lined up level with the second floor,  $F_2 = 1$  and  $F_1 = F_3 = 0$ . The circuit output is the *OPEN* signal, which is normally LOW and will go HIGH when the elevator door is to be opened.

We can fill in the truth table for the *OPEN* output [Figure 4-19(b)] as follows:

1. Because the elevator cannot be lined up with more than one floor at a time, only one of the floor inputs can be HIGH at any given time. This means that all those cases in the truth table where more than one floor input is a 1 are don't-care conditions. We can place an x in the *OPEN* output column for those eight cases where more than one *F* input is 1.
2. Looking at the other eight cases, when  $M = 1$  the elevator is moving, so *OPEN* must be a 0 because we do not want the elevator door to open. When  $M = 0$  (elevator stopped) we want  $OPEN = 1$  provided that one of the floor inputs is 1. When  $M = 0$  and all floor inputs are 0, the elevator is stopped but is not properly lined up with any floor, so we want  $OPEN = 0$  to keep the door closed.

The truth table is now complete and we can transfer its information to the K map in Figure 4-19(c). The map has only three 1s, but it has eight don't-cares. By changing four of these don't-care squares to 1s, we can produce quad loopings that contain the original 1s [Figure 4-19(d)]. This is the best we can do as far as minimizing the output expression. Verify that the loopings produce the *OPEN* output expression shown.

## Summary

The K-map process has several advantages over the algebraic method. K mapping is a more orderly process with well-defined steps compared with the trial-and-error process sometimes used in algebraic simplification. K mapping usually requires fewer steps, especially for expressions containing many terms, and it always produces a minimum expression.

Nevertheless, some instructors prefer the algebraic method because it requires a thorough knowledge of Boolean algebra and is not simply a mechanical procedure. Each method has its advantages, and although most logic designers are adept at both, being proficient in one method is all that is necessary to produce acceptable results.

There are other, more complex techniques that designers use to minimize logic circuits with more than four inputs. These techniques are especially suited for circuits with large numbers of inputs where algebraic and K-mapping methods are not feasible. Most of these techniques can be translated into a computer program that will perform the minimization from input data that supply the truth table or the unsimplified expression.

### OUTCOME ASSESSMENT QUESTIONS

1. Use K mapping to obtain the expression of Example 4-7.
2. Use K mapping to obtain the expression of Example 4-8. This should emphasize the advantage of K mapping for expressions containing many terms.
3. Obtain the expression of Example 4-9 using a K map.
4. What is a don't-care condition?

## 4-6 EXCLUSIVE-OR AND EXCLUSIVE-NOR CIRCUITS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Define the exclusive-OR and exclusive-NOR logic functions.
- Write Boolean equations using the XOR/XNOR functions.
- Draw the logic symbol for the XOR/XNOR functions.
- Write a truth table describing the XOR/XNOR functions.
- Draw a timing diagram that demonstrates the XOR/XNOR functions.
- Use any of the above methods to infer the correct output of a logic circuit based on its input.

Two special logic circuits that occur quite often in digital systems are the *exclusive-OR* and *exclusive-NOR* circuits.

### Exclusive-OR

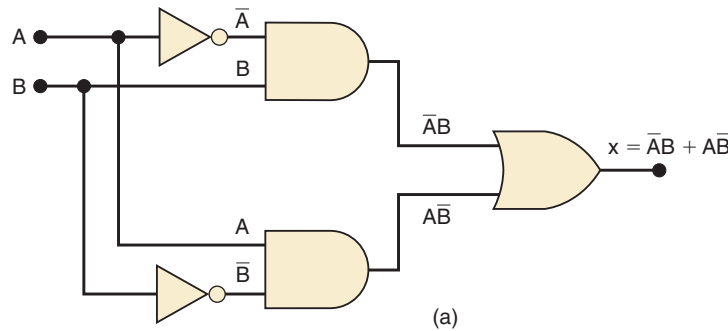
Consider the logic circuit of Figure 4-20(a). The output expression of this circuit is

$$x = \bar{A}B + A\bar{B}$$

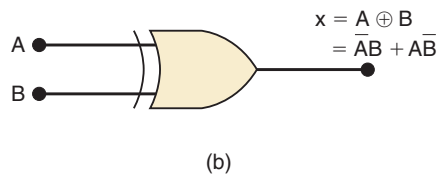


**FIGURE 4-20**

(a) Exclusive-OR circuit and truth table; (b) traditional XOR gate symbol.



A	B	x
0	0	0
0	1	1
1	0	1
1	1	0



The accompanying truth table shows that  $x = 1$  for two cases:  $A = 0, B = 1$  (the  $\bar{A}B$  term) and  $A = 1, B = 0$  (the  $A\bar{B}$  term). In other words:

**This circuit produces a HIGH output whenever the two inputs are at opposite levels.**

This is the **exclusive-OR** circuit, which will hereafter be abbreviated **XOR**.

This particular combination of logic gates occurs quite often and is very useful in certain applications. In fact, the XOR circuit has been given a symbol of its own, shown in Figure 4-20(b). This symbol is assumed to contain all of the logic contained in the XOR circuit and therefore has the same logic expression and truth table. This XOR circuit is commonly referred to as an XOR gate, and we consider it as another type of logic gate.

An XOR gate has only *two* inputs; there are no three-input or four-input XOR gates. The two inputs are combined so that  $x = \bar{A}B + A\bar{B}$ . A shorthand way that is sometimes used to indicate the XOR output expression is

$$x = A \oplus B$$

where the symbol  $\oplus$  represents the XOR gate operation.

The characteristics of an XOR gate are summarized as follows:

1. It has only two inputs and its output is

$$x = \bar{A}B + A\bar{B} = A \oplus B$$

2. Its output is *HIGH* only when the two inputs are at *different* levels.

Several ICs are available that contain XOR gates. Those listed below are *quad* XOR chips containing four XOR gates.

74LS86	Quad XOR (TTL family)
74C86	Quad XOR (CMOS family)
74HC86	Quad XOR (high-speed CMOS)

## Exclusive-NOR

The **exclusive-NOR** circuit (abbreviated **XNOR**) operates completely opposite to the XOR circuit. Figure 4-21(a) shows an XNOR circuit and its accompanying truth table. The output expression is

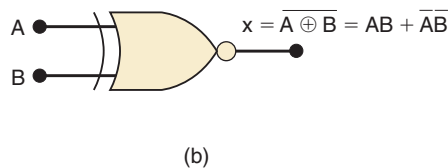
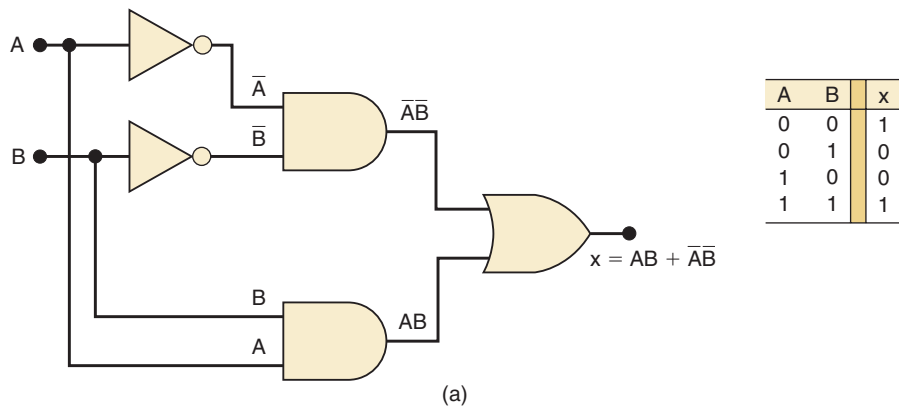
$$x = AB + \overline{A}\overline{B}$$

which indicates along with the truth table that  $x$  will be 1 for two cases:  $A = B = 1$  (the  $AB$  term) and  $A = B = 0$  (the  $\overline{A}\overline{B}$  term). In other words:

**The XNOR produces a HIGH output whenever the two inputs are at the same level.**

It should be apparent that the output of the XNOR circuit is the exact inverse of the output of the XOR circuit. The traditional symbol for an XNOR gate is obtained by simply adding a small circle at the output of the XOR symbol [Figure 4-21(b)].

**FIGURE 4-21**  
(a) Exclusive-NOR circuit;  
(b) traditional symbol for XNOR gate.



The XNOR gate also has *only two* inputs, and it combines them so that its output is

$$x = AB + \overline{A}\overline{B}$$

A shorthand way to indicate the output expression of the XNOR is

$$x = \overline{A \oplus B}$$

which is simply the inverse of the XOR operation. The XNOR gate is summarized as follows:

1. It has only two inputs and its output is

$$x = AB + \overline{A}\overline{B} = \overline{A \oplus B}$$

2. Its output is HIGH only when the two inputs are at the *same* level.

Several ICs are available that contain XNOR gates. Those listed below are quad XNOR chips containing four XNOR gates.

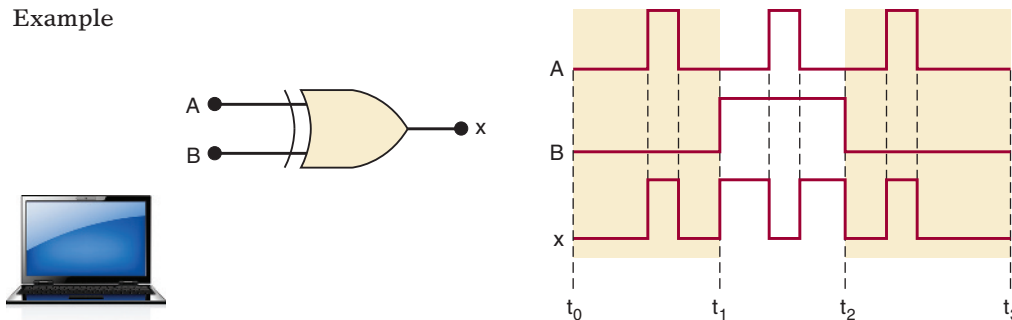
74LS266	Quad XNOR (TTL family)
74C266	Quad XNOR (CMOS)
74HC266	Quad XNOR (high-speed CMOS)

Each of these XNOR chips, however, has special output circuitry that limits its use to special types of applications. Very often, a logic designer will obtain the XNOR function simply by connecting the output of an XOR to an INVERTER.

### EXAMPLE 4-16

Determine the output waveform for the input waveforms given in Figure 4-22.

**FIGURE 4-22** Example 4-16.



### Solution

The output waveform is obtained using the fact that the XOR output will go HIGH only when its inputs are at different levels. The resulting output waveform reveals several interesting points:

1. The  $x$  waveform matches the  $A$  input waveform during those time intervals when  $B = 0$ . This occurs during the time intervals  $t_0$  to  $t_1$  and  $t_2$  to  $t_3$ .
2. The  $x$  waveform is the *inverse* of the  $A$  input waveform during those time intervals when  $B = 1$ . This occurs during the interval  $t_1$  to  $t_2$ .
3. These observations show that an XOR gate can be used as a *controlled INVERTER*; that is, one of its inputs can be used to control whether or not the signal at the other input will be inverted. This property will be useful in certain applications.

### EXAMPLE 4-17

The notation  $x_1x_0$  represents a two-bit binary number that can have any value (00, 01, 10, or 11); for example, when  $x_1 = 1$  and  $x_0 = 0$ , the binary number is 10, and so on. Similarly,  $y_1y_0$  represents another two-bit binary number. Design a logic circuit, using  $x_1$ ,  $x_0$ ,  $y_1$ , and  $y_0$  inputs, whose output will be HIGH only when the two binary numbers  $x_1x_0$  and  $y_1y_0$  are *equal*.

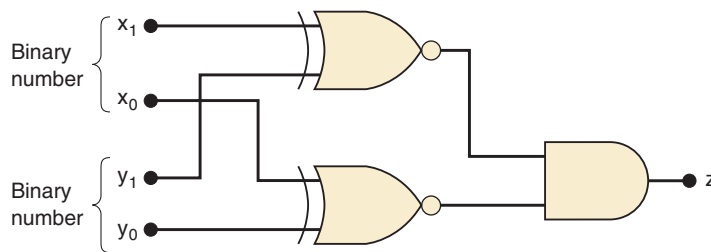
### Solution

The first step is to construct a truth table for the 16 input conditions (Table 4-4). The output  $z$  must be HIGH whenever the  $x_1x_0$  values match the  $y_1y_0$  values;

TABLE 4-4

$x_1$	$x_0$	$y_1$	$y_0$	$z$ (Output)
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

**FIGURE 4-23** Circuit for detecting equality of two two-bit binary numbers.



that is, whenever  $x_1 = y_1$  and  $x_0 = y_0$ . The table shows that there are four such cases. We could now continue with the normal procedure, which would be to obtain a sum-of-products expression for  $z$ , attempt to simplify it, and then implement the result. However, the nature of this problem makes it ideally suited for implementation using XNOR gates, and a little thought will produce a simple solution with minimum work. Refer to Figure 4-23; in this logic diagram,  $x_1$  and  $y_1$  are fed to one XNOR gate, and  $x_0$  and  $y_0$  are fed to another XNOR gate. The output of each XNOR will be HIGH only when its inputs are equal. Thus, for  $x_0 = y_0$  and  $x_1 = y_1$ , both XNOR outputs will be HIGH. This is the condition we are looking for because it means that the two two-bit numbers are equal. The AND gate output will be HIGH only for this case, thereby producing the desired output.

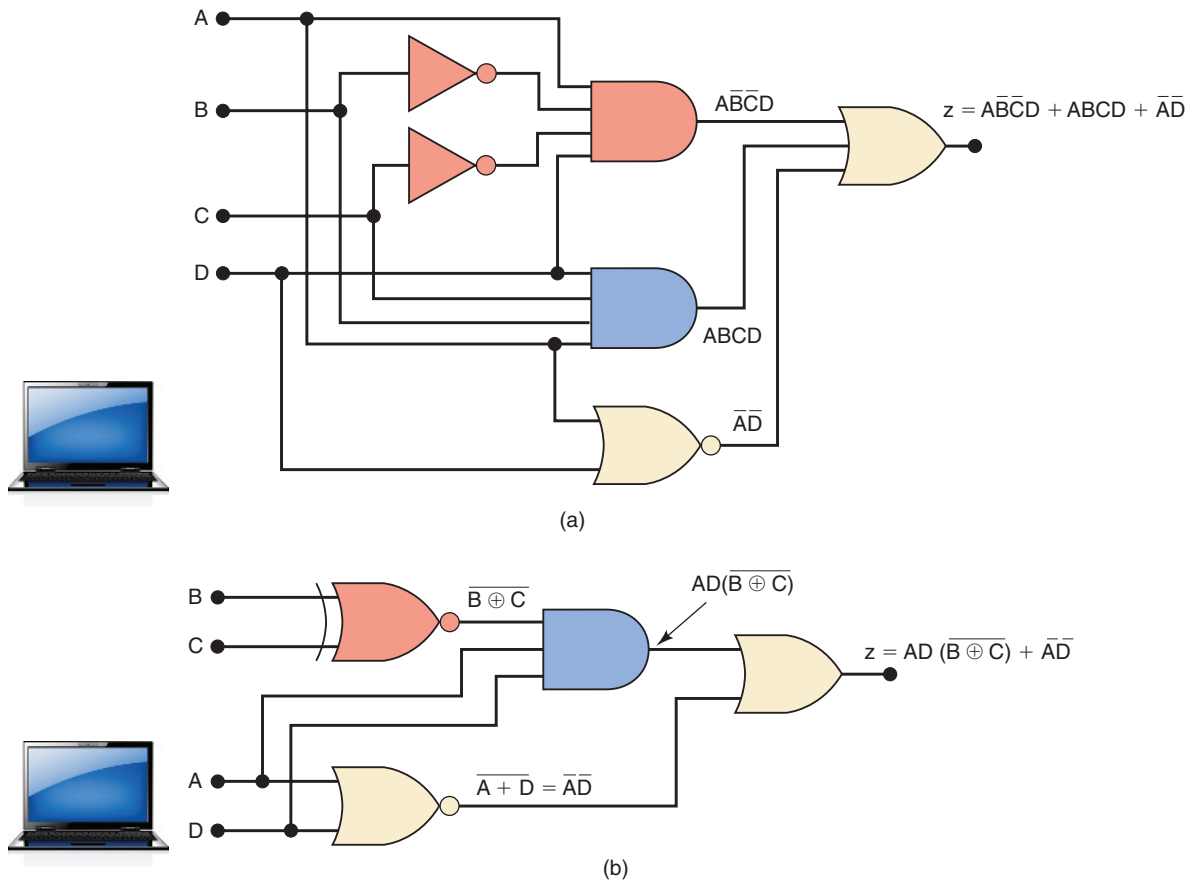
#### EXAMPLE 4-18

When simplifying the expression for the output of a combinational logic circuit, you may encounter the XOR or XNOR operations as you are factoring. This will often lead to the use of XOR or XNOR gates in the implementation of the final circuit. To illustrate, simplify the circuit of Figure 4-24(a).

#### Solution

The unsimplified expression for the circuit is obtained as

$$z = ABCD + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}D$$



**FIGURE 4-24** Example 4-18, showing how an XNOR gate may be used to simplify circuit implementation.

We can factor  $AD$  from the first two terms:

$$z = AD(BC + \overline{BC}) + \overline{AD}$$

At first glance, you might think that the expression in parentheses can be replaced by 1. But that would be true only if it were  $BC + \overline{BC}$ . You should recognize the expression in parentheses as the XNOR combination of  $B$  and  $C$ . This fact can be used to reimplement the circuit as shown in Figure 4-24(b). This circuit is much simpler than the original because it uses gates with fewer inputs and two INVERTERS have been eliminated.

### OUTCOME ASSESSMENT QUESTIONS

1. Use Boolean algebra to prove that the XNOR output expression is the exact inverse of the XOR output expression.
2. What is the output of an XNOR gate when a logic signal and its exact inverse are connected to its inputs?
3. A logic designer needs an INVERTER, and all that is available is one XOR gate from a 74HC86 chip. Does he need another chip?

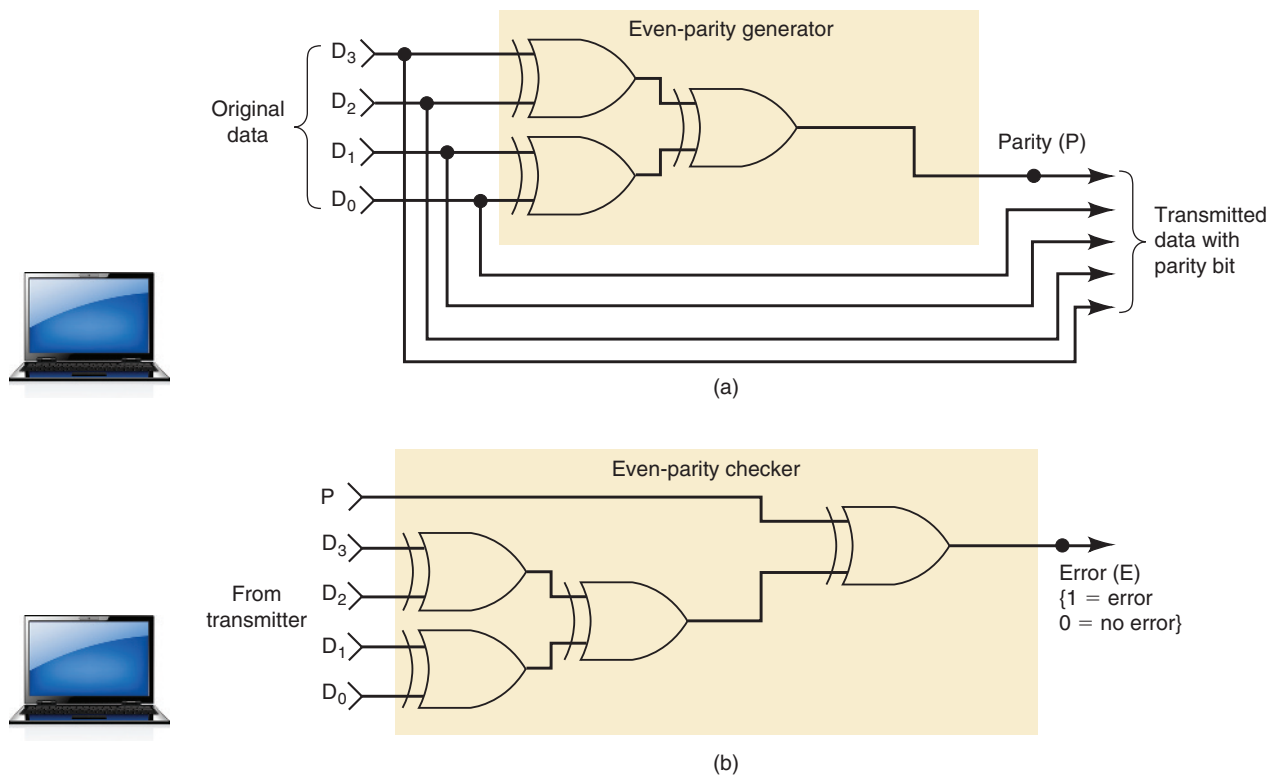
## 4-7 PARITY GENERATOR AND CHECKER

### OUTCOMES

Upon completion of this section, you will be able to:

- Apply XOR gates to make a parity generator.
- Apply XOR gates to make a parity checker.

In Chapter 2, we saw that a transmitter can attach a parity bit to a set of data bits before transmitting the data bits to a receiver. We also saw how this allows the receiver to detect any single-bit errors that may have occurred during the transmission. Figure 4-25 shows an example of one type of logic circuitry that is used for **parity generation** and **parity checking**. This particular example uses a group of four bits as the data to be transmitted, and it uses an even-parity bit. It can be readily adapted to use odd parity and any number of bits.



**FIGURE 4-25** XOR gates used to implement (a) the parity generator and (b) the parity checker for an even-parity system.

In Figure 4-25(a), the set of data to be transmitted is applied to the parity-generator circuit, which produces the even-parity bit,  $P$ , at its output. This parity bit is transmitted to the receiver along with the original data bits, making a total of five bits. In Figure 4-25(b), these five bits (data + parity) enter the receiver's parity-checker circuit, which produces an error output,  $E$ , that indicates whether or not a single-bit error has occurred.

It should not be too surprising that both of these circuits employ XOR gates when we consider that a single XOR gate operates so that it produces a 1 output if an odd number of its inputs are 1, and a 0 output if an even number of its inputs are 1.

**EXAMPLE 4-19**

Determine the parity generator's output for each of the following sets of input data,  $D_3D_2D_1D_0$ : (a) 0111; (b) 1001; (c) 0000; (d) 0100. Refer to Figure 4-25(a).

**Solution**

For each case, apply the data levels to the parity-generator inputs and trace them through each gate to the  $P$  output. The results are: (a) 1; (b) 0; (c) 0; and (d) 1. Note that  $P$  is a 1 only when the original data contain an odd number of 1s. Thus, the total number of 1s sent to the receiver (data + parity) will be even.

**EXAMPLE 4-20**

Determine the parity checker's output [see Figure 4-25(b)] for each of the following sets of data from the transmitter:

	$P$	$D_3$	$D_2$	$D_1$	$D_0$
(a)	0	1	0	1	0
(b)	1	1	1	1	0
(c)	1	1	1	1	1
(d)	1	0	0	0	0

**Solution**

For each case, apply these levels to the parity-checker inputs and trace them through to the  $E$  output. The results are: (a) 0; (b) 0; (c) 1; (d) 1. Note that a 1 is produced at  $E$  only when an odd number of 1s appears in the inputs to the parity checker. This indicates that an error has occurred because even parity is being used.

**OUTCOME ASSESSMENT QUESTIONS**

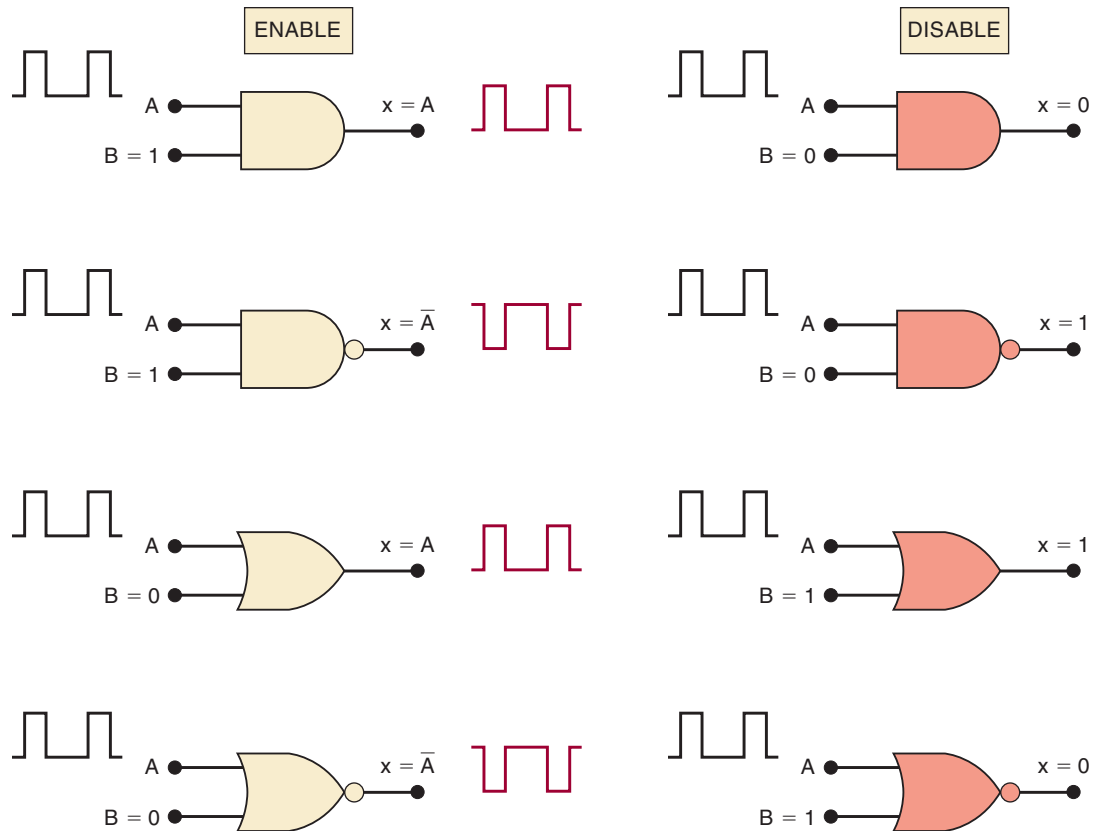
1. How many XOR gates are required to generate the parity bit for an eight-bit data value?
2. How many XOR gates are required to check the parity bit for an eight-bit data value (plus one parity bit)?

**4-8 ENABLE/DISABLE CIRCUITS****OUTCOMES**

*Upon completion of this section, you will be able to:*

- Use logic circuits to selectively enable/disable the passing of a signal.
- Select the correct logic function to assure required logic levels when disabled.

Each of the basic logic gates can be used to control the passage of an input logic signal through to the output. This is depicted in Figure 4-26, where a logic signal,  $A$ , is applied to one input of each of the basic logic gates. The other input of each gate is the control input,  $B$ . The logic level at this control input will determine whether the input signal is **enabled** to reach the output



**FIGURE 4-26** Four basic gates can either enable or disable the passage of an input signal,  $A$ , under control of the logic level at control input  $B$ .

or **disabled** from reaching the output. This controlling action is why these circuits came to be called *gates*.

Examine Figure 4-26 and you should notice that when the noninverting gates (AND, OR) are enabled, the output will follow the  $A$  signal exactly. Conversely, when the inverting gates (NAND, NOR) are enabled, the output will be the exact inverse of the  $A$  signal.

Also notice in the figure that AND and NOR gates produce a constant LOW output when they are in the disabled condition. Conversely, the NAND and OR gates produce a constant HIGH output in the disabled condition.

There will be many situations in digital-circuit design where the passage of a logic signal is to be enabled or disabled, depending on conditions present at one or more control inputs. Several are shown in the following examples.

#### EXAMPLE 4-21

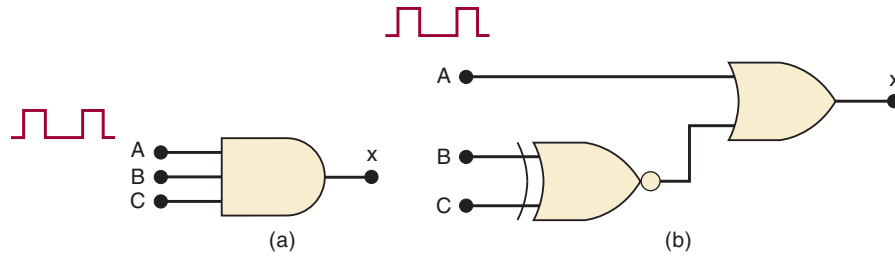
Design a logic circuit that will allow a signal to pass to the output only when control inputs  $B$  and  $C$  are both HIGH; otherwise, the output will stay LOW.

#### Solution

An AND gate should be used because the signal is to be passed without inversion, and the disable output condition is a LOW. Because the enable condition must occur only when  $B = C = 1$ , a three-input AND gate is used, as shown in Figure 4-27(a).



**FIGURE 4-27** Examples 4-21 and 4-22.



**EXAMPLE 4-22**

Design a logic circuit that allows a signal to pass to the output only when one, but not both, of the control inputs are HIGH; otherwise, the output will stay HIGH.

**Solution**

The result is drawn in Figure 4-27(b). An OR gate is used because we want the output disable condition to be a HIGH, and we do not want to invert the signal. Control inputs *B* and *C* are combined in an XNOR gate. When *B* and *C* are different, the XNOR sends a LOW to enable the OR gate. When *B* and *C* are the same, the XNOR sends a HIGH to disable the OR gate.

**EXAMPLE 4-23**

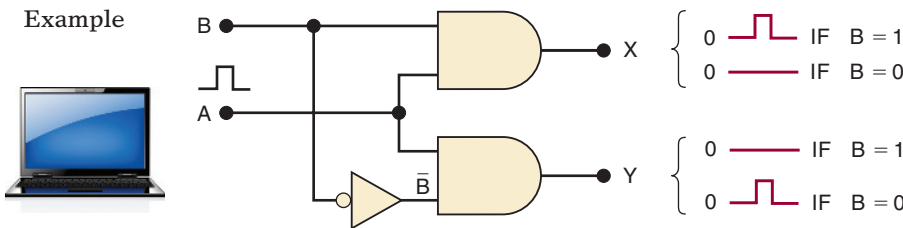
Design a logic circuit with input signal *A*, control input *B*, and outputs *X* and *Y* to operate as follows:

1. When *B* = 1, output *X* will follow input *A*, and output *Y* will be 0.
2. When *B* = 0, output *X* will be 0, and output *Y* will follow input *A*.

**Solution**

The two outputs will be 0 when they are disabled and will follow the input signal when they are enabled. Thus, an AND gate should be used for each output. Because *X* is to be enabled when *B* = 1, its AND gate must be controlled by *B*, as shown in Figure 4-28. Because *Y* is to be enabled when *B* = 0, its AND gate is controlled by  $\bar{B}$ . The circuit in Figure 4-28 is called a *pulse-steering circuit* because it steers the input pulse to one output or the other, depending on *B*.

**FIGURE 4-28** Example 4-23.



**OUTCOME ASSESSMENT QUESTIONS**

1. Design a logic circuit with three inputs *A*, *B*, *C* and an output that goes LOW only when *A* is HIGH while *B* and *C* are different.
2. Design a circuit to pass signal *A* only when *B* is HIGH and *C* is LOW. The output must be LOW when *A* is not being passed.
3. Which logic gates produce a 1 output in the disabled state?
4. Which logic gates pass the inverse of the input signal when they are enabled?

## 4-9 BASIC CHARACTERISTICS OF LEGACY DIGITAL ICs\*

### OUTCOMES

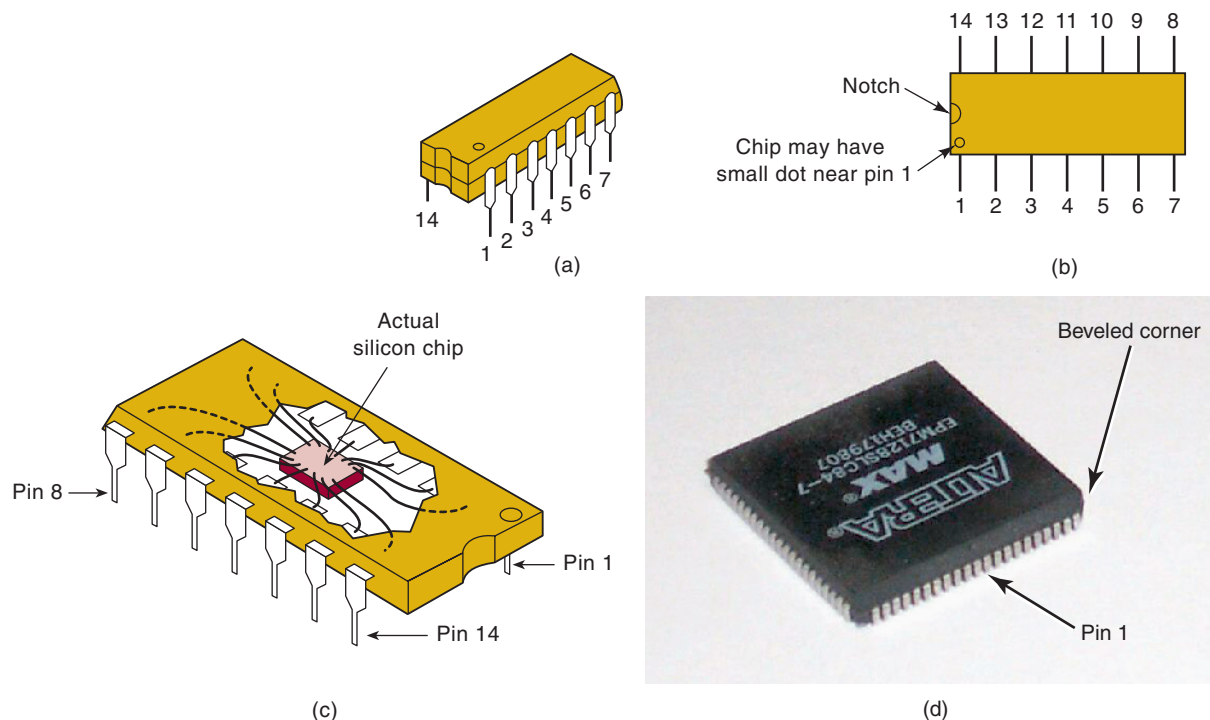
Upon completion of this section, you will be able to:

- Use legacy technology for educational purposes.
- Identify critical characteristics of TTL and CMOS circuits.
- Implement a logic circuit using SSI and MSI technology integrated circuits.

Digital ICs are a collection of resistors, diodes, and transistors fabricated on a single piece of semiconductor material (usually silicon) called a *substrate*, which is commonly referred to as a *chip*. The chip is enclosed in a protective plastic or ceramic package from which pins extend for connecting the IC to other devices. One of the more common types of package is the **dual-in-line package (DIP)**, shown in Figure 4-29(a), so called because it contains two parallel rows of pins. The pins are numbered counterclockwise when viewed from the top of the package with respect to an identifying notch or dot at one end of the package [see Figure 4-29(b)]. The DIP shown here is a 14-pin package that measures 0.75 in. by 0.25 in.; 16-, 20-, 24-, 28-, 40-, and 64-pin packages are also used.

Figure 4-29(c) shows that the actual silicon chip is much smaller than the DIP; typically, it might be a 0.05-in. square. The silicon chip is connected to the pins of the DIP by very fine wires (1-mil diameter).

The DIP is probably the most common digital IC package found in older digital equipment, but other types are becoming more and more popular. The IC shown in Figure 4-29(d) is only one of the many packages common



**FIGURE 4-29** (a) Dual-in-line package (DIP); (b) top view; (c) actual silicon chip is much smaller than the protective package; (d) PLCC package.

\*This section is included for those who use TTL ICs in Laboratory exercises. For those using FPGAs for experiments, this section can be skipped.

to modern digital circuits. This particular package uses J-shaped leads that curl under the IC. We will take a look at some of these other types of IC packages in Chapter 8.

Digital ICs are often categorized according to their circuit complexity as measured by the number of equivalent logic gates on the substrate. The levels of complexity that are commonly defined are shown in Table 4-5.

**TABLE 4-5** Categories of ICs.

Complexity	Gates per Chip
Small-scale integration (SSI)	Fewer than 12
Medium-scale integration (MSI)	12–99
Large-scale integration (LSI)	100–9999
Very large-scale integration (VLSI)	10,000–99,999
Ultra large-scale integration (ULSI)	100,000–999,999
Giga-scale integration (GSI)	1,000,000 or more

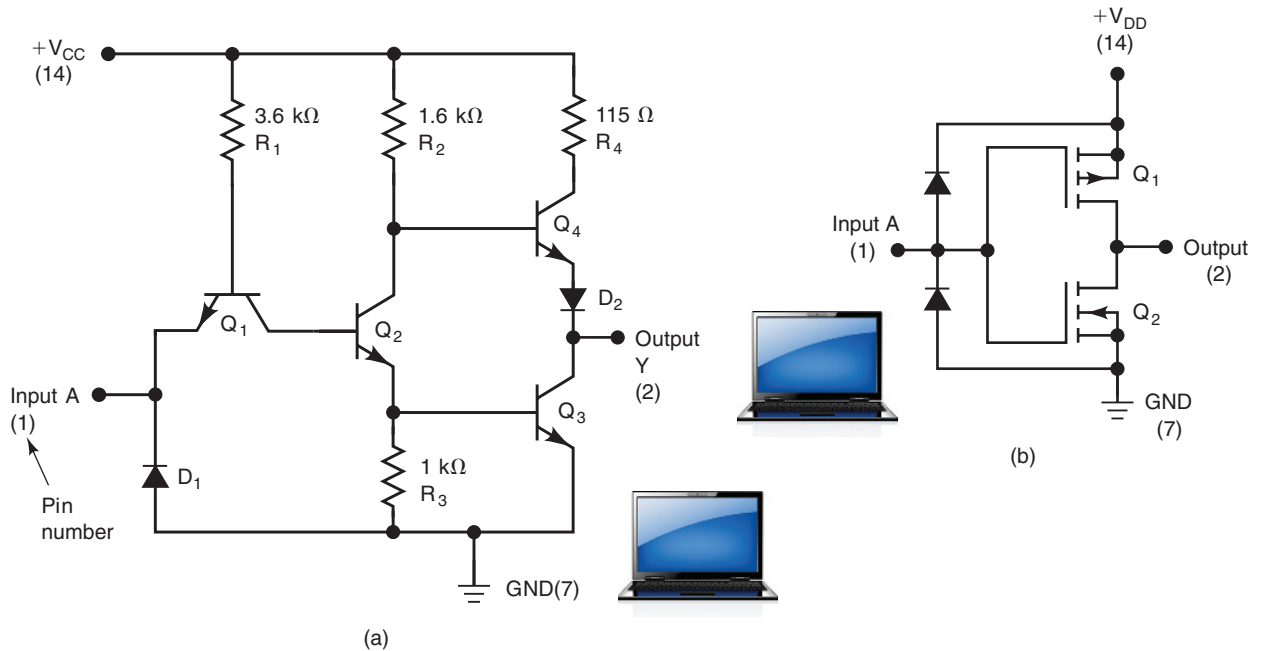
All of the specific ICs referred to in Chapter 3 and this chapter are **SSI** chips having a small number of gates. In digital systems, medium-scale integration (**MSI**) and large-scale integration devices (**LSI**, **VLSI**, **ULSI**, **GSI**) can perform most of the functions that once required several circuit boards full of SSI devices. However, SSI chips are still used as the “interface,” or “glue,” between these more complex chips. The small-scale ICs also offer an excellent way to learn the basic building blocks of digital systems. Consequently, many laboratory-based courses use these ICs to build and test small projects.

The industrial world of digital electronics has now turned to programmable logic devices (PLDs) to implement a digital system of any significant size. Some simple PLDs are available in DIP packages, but the more complex programmable logic devices require many more pins than are available in DIPs. Larger integrated circuits that may need to be removed from a circuit and replaced are typically manufactured in a plastic leaded chip carrier (PLCC) package. Figure 4-29(d) shows the Altera EPM 7128SLC84 in a PLCC package. The key features of this chip are more pins, closer spacing, and pins around the entire periphery. Notice that pin 1 is not “on the corner” like the DIP but rather at the middle of the top of the package. Most logic circuits in use today are much more complex (VLSI and greater) and require many more pins. They are not able to be removed and replaced on an experimental circuit board so they will be described in a later section.

## Bipolar and Unipolar Digital ICs

Digital ICs can also be categorized according to the principal type of electronic component used in their circuitry. *Bipolar ICs* are made using the bipolar junction transistor (NPN and PNP) as their main circuit element. *Unipolar ICs* use the unipolar field-effect transistor (P-channel and N-channel MOSFETs) as their main element.

The **transistor-transistor logic (TTL)** family has been the major family of bipolar digital ICs for over 40 years. The standard 74 series was the first series of TTL ICs. It is no longer used in new designs, having been replaced by several higher-performance TTL series, but its basic circuit arrangement forms the foundation for all the TTL series ICs. This circuit arrangement is shown in Figure 4-30(a) for the standard TTL INVERTER. Notice that the circuit contains several bipolar transistors as the main circuit element.



**FIGURE 4-30** (a) TTL INVERTER circuit; (b) CMOS INVERTER circuit. Pin numbers are given in parentheses.

TTL was the leading IC family in the SSI and MSI categories up until approximately 1990. Since then, its leading position has been challenged by CMOS technology, which has gradually displaced TTL from that position. The **complementary metal-oxide semiconductor (CMOS)** family belongs to the class of unipolar digital ICs because it uses P- and N-channel MOSFETs as the main circuit elements. Figure 4-30(b) is a standard CMOS INVERTER circuit. If we compare the TTL and CMOS circuits in Figure 4-30, it is apparent that the CMOS version uses fewer components. This is one of the main advantages of CMOS over TTL.

Because of the simplicity and compactness as well as some other superior attributes of CMOS, the modern large-scale ICs are manufactured primarily using CMOS technology. Teaching laboratories that use SSI and MSI devices often use TTL due to its durability, although some use CMOS as well. Chapter 8 will provide a comprehensive study of the circuitry and characteristics of TTL and CMOS. For now, we need to look at only a few of their basic characteristics so that we can talk about troubleshooting simple combinational circuits.

## TTL Family

The TTL logic family actually consists of several subfamilies or series. Table 4-6 lists the name of each TTL series together with the prefix designation used to identify different ICs as being part of that series. For example, ICs that are part of the standard TTL series have an identification number that starts with 74. The 7402, 7438, and 74123 are all ICs in this series. Likewise, ICs that are part of the low-power Schottky TTL series have an identification number that starts with 74LS. The 74LS02, 74LS38, and 74LS123 are examples of devices in the 74LS series.

The principal differences in the various TTL series have to do with their electrical characteristics such as power dissipation and switching speed. They

**TABLE 4-6** Various series within the TTL logic family.

TTL Series	Prefix	Example IC
Standard TTL	74	7404 (hex INVERTER)
Schottky TTL	74S	74S04 (hex INVERTER)
Low-power Schottky TTL	74LS	74LS04 (hex INVERTER)
Advanced Schottky TTL	74AS	74AS04 (hex INVERTER)
Advanced low-power Schottky TTL	74ALS	74ALS04 (hex INVERTER)

do not differ in the pin layout or logic operations performed by the circuits on the chip. For example, the 7404, 74S04, 74LS04, 74AS04, and 74ALS04 are all hex-INVERTER ICs, each containing *six* INVERTERs on a single chip.

### CMOS Family

Several CMOS series are available, and some of these are listed in Table 4-7. The 4000 series is the oldest CMOS series. This series contains many of the same logic functions as the TTL family but was not designed to be *pin-compatible* with TTL devices. For example, the 4001 quad NOR chip contains four two-input NOR gates, as does the TTL 7402 chip, but the gate inputs and outputs on the CMOS chip will not have the same pin numbers as the corresponding signals on the TTL chip.

The 74C, 74HC, 74HCT, 74AC, and 74ACT series are newer CMOS series. The first three are pin-compatible with correspondingly numbered TTL devices. For example, the 74C02, 74HC02, and 74HCT02 have the same pin layout as the 7402, 74LS02, and so on. The 74HC and 74HCT series operate at a higher speed than 74C devices. The 74HCT series is designed to be *electrically compatible* with TTL devices; that is, a 74HCT integrated circuit can be connected directly to TTL devices without any interfacing circuitry. The 74AC and 74ACT series are advanced-performance ICs. Neither is pin-compatible with TTL. The 74ACT devices are electrically compatible with TTL. We explore the various TTL and CMOS series in greater detail in Chapter 8 as well as the latest low-voltage technologies used in modern ICs.

**TABLE 4-7** Various series within the CMOS logic family.

CMOS Series	Prefix	Example IC
Metal-gate CMOS	40	4001 (quad NOR gates)
Metal-gate, pin-compatible with TTL	74C	74C02 (quad NOR gates)
Silicon-gate, pin-compatible with TTL, high speed	74HC	74HC02 (quad NOR gates)
Silicon-gate, high-speed, pin-compatible and electrically compatible with TTL	74HCT	74HCT02 (quad NOR gates)
Advanced-performance CMOS, not pin-compatible or electrically compatible with TTL	74AC	74AC02 (quad NOR)
Advanced-performance CMOS, not pin-compatible with TTL, but electrically compatible with TTL	74ACT	74ACT02 (quad NOR)

### Power and Ground

To use digital ICs, it is necessary to make the proper connections to the IC pins. The most important connections are *dc power* and *ground*. These are required for the circuits on the chip to operate correctly. In Figure 4-30, you can see that both the TTL and the CMOS circuits have a dc power supply

voltage connected to one of their pins and ground to another. The power supply pin is labeled  $V_{CC}$  for the TTL circuit and  $V_{DD}$  for the CMOS circuit. Many of the newer CMOS integrated circuits that are designed to be compatible with TTL integrated circuits also use the  $V_{CC}$  designation as their power pin.

If either the power or the ground connection is not made to the IC, the logic gates on the chip will not respond properly to the logic inputs, and the gates will not produce the expected output logic levels.

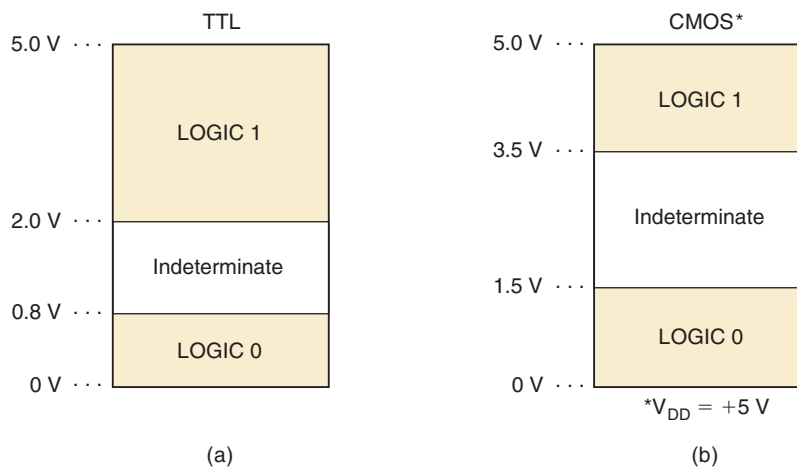
### Logic-Level Voltage Ranges

For TTL devices,  $V_{CC}$  is nominally +5V. For CMOS integrated circuits,  $V_{DD}$  can range from +3 to +18V, although +5V is most often used when CMOS integrated circuits are used in the same circuit with TTL integrated circuits.

For standard TTL devices, the acceptable input voltage ranges for the logic 0 and logic 1 levels are defined as shown in Figure 4-31(a). A logic 0 is any voltage in the range from 0 to 0.8 V; a logic 1 is any voltage from 2 to 5 V. Voltages that are not in either of these ranges are said to be **indeterminate** and should not be used as inputs to any TTL device. The IC manufacturers cannot guarantee how a TTL circuit will respond to input levels that are in the indeterminate range (between 0.8 and 2.0 V).

The logic input voltage ranges for CMOS integrated circuits operating with  $V_{DD} = +5V$  are shown in Figure 4-31(b). Voltages between 0 and 1.5 V are defined as a logic 0, and voltages from 3.5 to 5 V are defined as a logic 1. The indeterminate range includes voltages between 1.5 and 3.5 V.

**FIGURE 4-31** Logic-level input voltage ranges for (a) TTL and (b) CMOS digital ICs.



### Unconnected (Floating) Inputs

What happens when the input to a digital IC is left unconnected? An unconnected input is often called a **floating** input. The answer to this question will be different for TTL and CMOS.

A floating TTL input acts just like a logic 1. In other words, the IC will respond as if the input had a logic HIGH level applied to it. This characteristic is often used when testing a TTL circuit. A lazy technician might leave certain inputs unconnected instead of connecting them to a logic HIGH. Although this is logically correct, it is not a recommended practice, especially in final circuit designs, because the floating TTL input is extremely susceptible to picking up noise signals that will probably adversely affect the device's operation.

A floating input on some TTL gates will measure a dc level of between 1.4 and 1.8 V when checked with a VOM or an oscilloscope. Even though this is in the indeterminate range for TTL, it will produce the same response as a logic 1. Being aware of this characteristic of a floating TTL input can be valuable when troubleshooting TTL circuits.

If a CMOS input is left floating, it may have disastrous results. The IC may become overheated and eventually destroy itself. For this reason all inputs to a CMOS integrated circuit must be connected to a LOW or a HIGH level or to the output of another IC. A floating CMOS input will not measure as a specific dc voltage but will fluctuate randomly as it picks up noise. Thus, it does not act as logic 1 or logic 0, and so its effect on the output is unpredictable. Sometimes the output will oscillate as a result of the noise picked up by the floating input.

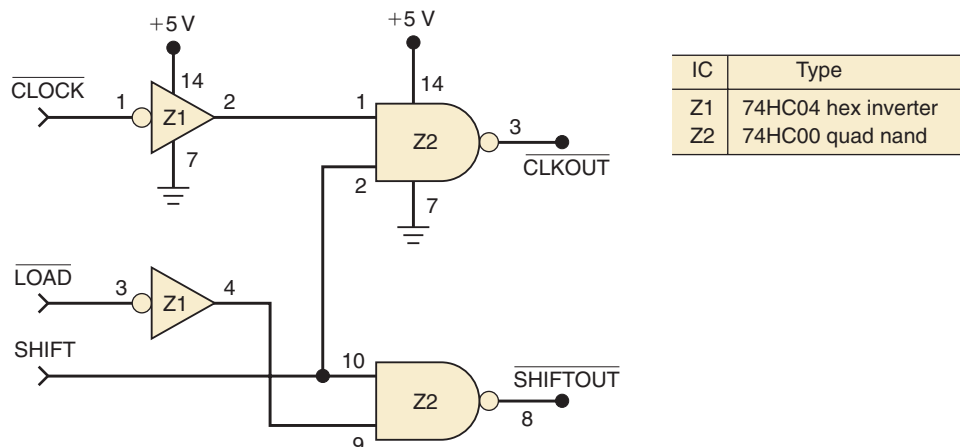
Many of the more complex CMOS ICs have circuitry built into the inputs, which reduces the likelihood of any destructive reaction to an open input. With this circuitry, it is not necessary to ground each unused pin on these large ICs when experimenting. It is still good practice, however, to tie unused inputs to HIGH or LOW (whichever is appropriate) in the final circuit implementation.

### Logic-Circuit Connection Diagrams

A connection diagram shows *all* electrical connections, pin numbers, IC numbers, component values, signal names, and power supply voltages. Figure 4-32 shows a typical connection diagram for a simple logic circuit. Examine it carefully and note the following important points:

1. The circuit uses logic gates from two different ICs. The two INVERTERS are part of a 74HC04 chip that has been given the designation Z1. The 74HC04 contains six INVERTERS; two of them are used in this circuit, and each is labeled as being part of chip Z1. Similarly, the two NAND gates are part of a 74HC00 chip that contains four NAND gates. All of the gates on this chip are designated with the label Z2. By numbering each gate as Z1, Z2, Z3, and so on, we can keep track of which gate is part of which chip. This is especially valuable in more complex circuits containing many ICs with several gates per chip.
2. Each gate input and output pin number is indicated on the diagram. These pin numbers and the IC labels are used to reference easily any

**FIGURE 4-32** Typical logic-circuit connection diagram.

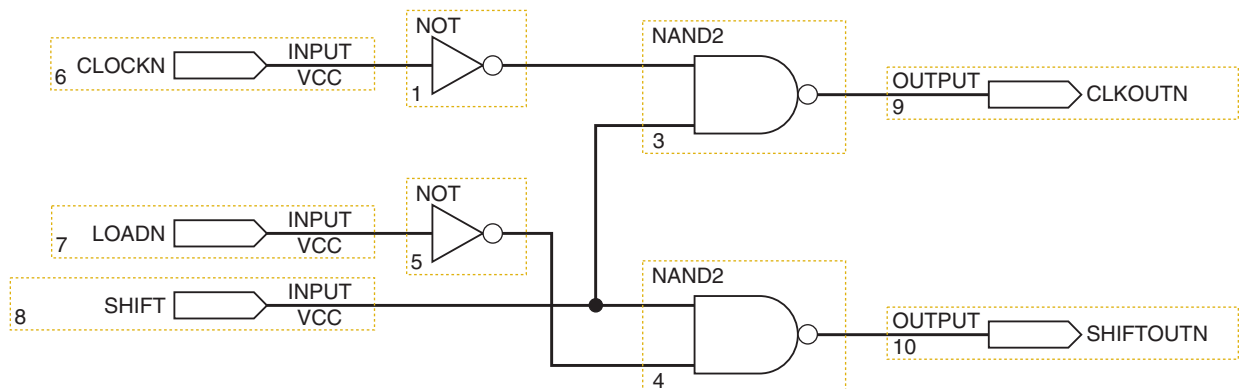


point in the circuit. For example, Z1 pin 2 refers to the output pin of the top INVERTER. Similarly, we can say that Z1 pin 4 is connected to Z2 pin 9.

3. The power and ground connections to each IC (not each gate) are shown on the diagram. For example, Z1 pin 14 is connected to +5 V, and Z1 pin 7 is connected to ground. These connections provide power to *all* of the six INVERTERS that are part of Z1.
4. For the circuit contained in Figure 4-32, the signals that are inputs are on the left. The signals that are outputs are on the right. The bar over the signal name indicates that the signal is active when LOW. The bubbles are positioned on the diagram symbols also to indicate the active-LOW state. Each signal in this case is obviously a single bit.
5. Signals are defined graphically in Figure 4-32 as inputs and outputs, and the relationship between them (the operation of the circuit) is described graphically using interconnected logic symbols.

Manufacturers of electronic equipment generally supply detailed schematics that use a format similar to that in Figure 4-32. These connection diagrams are a virtual necessity when troubleshooting a faulty circuit. We have chosen to identify individual ICs as Z1, Z2, Z3, and so on. Other designations that are commonly used are IC1, IC2, IC3, and so on, and U1, U2, U3, and so on.

In Chapter 3 we introduced the graphic entry tools of Quartus II software from Altera. An example of a logic diagram drawn using Altera software is shown in Figure 4-33. A circuit like this is not intended to be implemented using SSI or MSI logic ICs. That is why there are no pin numbers or chip designations on the logic symbols, only instance numbers. Altera's software will translate a graphic description of the logic function into a binary file that is used to configure logic circuits inside one of Altera's many digital ICs. These configurable or programmable logic circuits will be described in detail later in this chapter. Also, notice that a common convention in naming input and output signals is to use the N suffix rather than the overbar to indicate that the signal is active-LOW. For example, the LOADN input is a signal that will be LOW in order to perform the LOAD function.



**FIGURE 4-33** Logic diagram using Quartus II schematic capture.



**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Which type of transistor is used in (a) TTL and (b) CMOS?
2. Name the six common categories of digital ICs according to complexity.
3. *True or false:* A 74S74 chip will contain the same logic and pin layout as the 74LS74.
4. *True or false:* A 74HC74 chip will contain the same logic and pin layout as the 74AS74.
5. Which CMOS series are not pin-compatible with TTL?
6. What is the acceptable input voltage range of a logic 0 for TTL? What is it for a logic 1?
7. Repeat question 6 for CMOS operating at  $V_{DD} = 5\text{V}$ .
8. How does a TTL integrated circuit respond to a floating input?
9. How does a CMOS integrated circuit respond to a floating input?
10. Which CMOS series can be connected directly to TTL with no interfacing circuitry?
11. What is the purpose of pin numbers on a logic circuit connection diagram?
12. What are the key similarities of graphic design files used for programmable logic and traditional logic circuit connection diagrams?

## 4-10 TROUBLESHOOTING DIGITAL SYSTEMS

### OUTCOMES

Upon completion of this section, you will be able to:

- State three steps necessary to recover from a system fault or failure.
- Use a logic probe to determine the logic level present at any point in a circuit.

There are three basic steps in fixing a digital circuit or system that has a fault (failure):

1. *Fault detection.* Observe the circuit/system operation and compare it with the expected correct operation.
2. *Fault isolation.* Perform tests and make measurements to isolate the fault.
3. *Fault correction.* Replace the faulty component, repair the faulty connection, remove the short, and so on.

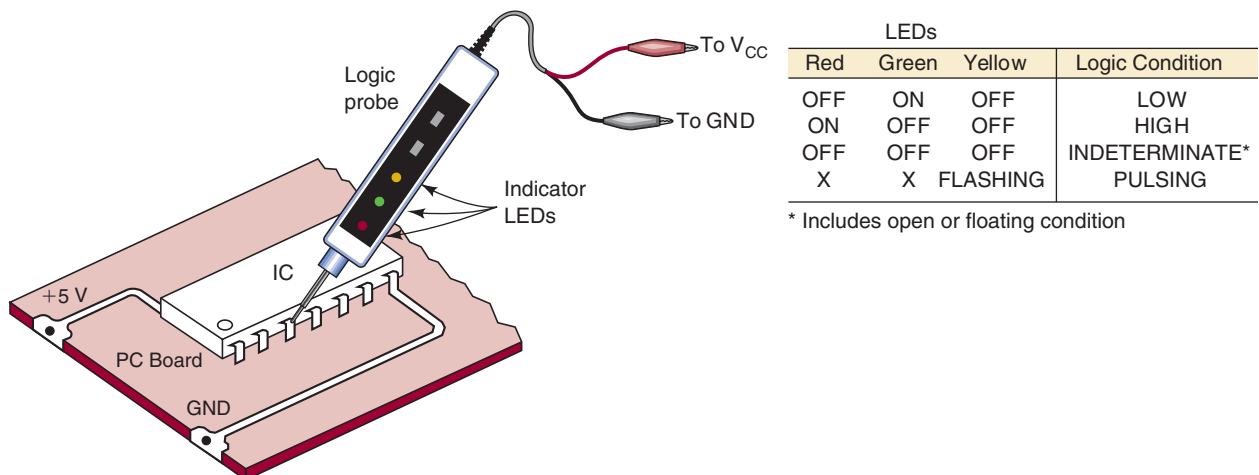
Although these steps may seem relatively apparent and straightforward, the actual troubleshooting procedure that is followed is highly dependent on the type and complexity of the circuitry, and on the kinds of troubleshooting tools and documentation that are available.

Good troubleshooting techniques can be learned only in a laboratory environment through experimentation and actual troubleshooting of faulty circuits and systems. There is absolutely no better way to become an effective troubleshooter than to do as much troubleshooting as possible, and no amount of textbook reading can provide that kind of experience. We can, however, help you to develop the analytical skills that are the most essential part of effective troubleshooting. We will describe the types of faults that are common to systems that are made primarily from digital ICs and we will tell you how to recognize them. We will then present typical case studies to illustrate the analytical processes involved in troubleshooting.

In addition, there will be end-of-chapter problems to provide you with the opportunity to go through these analytical processes to reach conclusions about faulty digital circuits.

For the troubleshooting discussions and exercises we will be doing in this book, it will be assumed that the troubleshooting technician has the basic troubleshooting tools available: *logic probe*, *oscilloscope*, *logic pulser*. Of course, the most important and effective troubleshooting tool is the technician's brain, and that's the tool we are hoping to develop by presenting troubleshooting principles and techniques, examples and problems, here and in the following chapters.

In the next three sections on troubleshooting, we will use only our brain and a **logic probe** such as the one illustrated in Figure 4-34. The logic probe has a pointy metal tip that is touched to the specific point you want to test. Here, it is shown probing (touching) pin 3 of an IC. It can also be touched to a printed circuit board trace, an uninsulated wire, a connector pin, a lead on a discrete component such as a transistor, or any other conducting point in a circuit. The logic level that is present at the probe tip will be indicated by the status of the indicator LEDs in the probe. The four possibilities are given in the table of Figure 4-34. Note that an *indeterminate* logic level produces no indicator light. This includes the condition where the probe tip is touched to a point in a circuit that is open or floating—that is, not connected to any source of voltage. This type of probe also offers a yellow LED to indicate the presence of a pulse train. Any transitions (LOW to HIGH or HIGH to LOW) will cause the yellow LED to flash on for a fraction of a second and then turn off. If the transitions are occurring frequently, the LED will continue to flash at around 3 Hz. By observing the green and red LEDs along with the flashing yellow, you can tell whether the signal is mostly HIGH or mostly LOW.



**FIGURE 4-34** A logic probe is used to monitor the logic level activity at an IC pin or any other accessible point in a logic circuit.

#### OUTCOME ASSESSMENT QUESTIONS

1. List the three steps necessary to recover from a system fault or failure.
2. List the indicators on a logic probe.

## 4-11 INTERNAL DIGITAL IC FAULTS

### OUTCOMES

Upon completion of this section, you will be able to:

- Identify common failure modes of digital integrated circuits.
- Recognize the symptoms of each failure mode.
- Understand signal contention.

The most common internal failures of digital ICs are:

1. Malfunction in the internal circuitry
2. Inputs or outputs shorted to ground or  $V_{CC}$
3. Inputs or outputs open-circuited
4. Short between two pins (other than ground or  $V_{CC}$ )

We will now describe each of these types of failure.

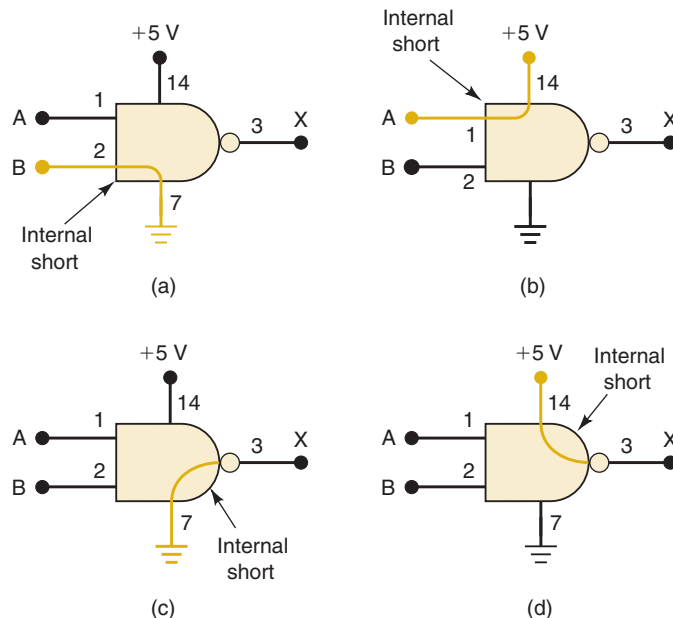
### Malfunction in Internal Circuitry

This is usually caused by one of the internal components failing completely or operating outside its specifications. When this happens, the IC outputs do not respond properly to the IC inputs. There is no way to predict what the outputs will do because it depends on what internal component has failed. Examples of this type of failure would be a base-emitter short in transistor  $Q_4$  or an extremely large resistance value for  $R_2$  in the TTL INVERTER of Figure 4-30(a). This type of internal IC failure is not as common as the other three.

### Input Internally Shorted to Ground or Supply

This type of internal failure will cause an IC input to be stuck in the LOW or HIGH state. Figure 4-35(a) shows input pin 2 of a NAND gate shorted to ground within the IC. This will cause pin 2 always to be in the LOW state. If

**FIGURE 4-35** (a) IC input internally shorted to ground; (b) IC input internally shorted to supply voltage. These two types of failures force the input signal at the shorted pin to stay in the same state. (c) IC output internally shorted to ground; (d) output internally shorted to supply voltage. These two failures do not affect signals at the IC inputs.



this input pin is being driven by a logic signal  $B$ , it will effectively short  $B$  to ground. Thus, this type of fault will affect the output of the device that is generating the  $B$  signal.

Similarly, an IC input pin could be internally shorted to +5V, as in Figure 4-35(b). This would keep that pin stuck in the HIGH state. If this input pin is being driven by a logic signal  $A$ , it will effectively short  $A$  to +5V.

### Output Internally Shorted to Ground or Supply

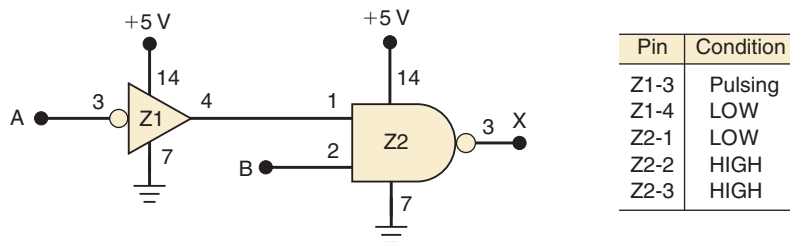
This type of internal failure will cause the output pin to be stuck in the LOW or HIGH state. Figure 4-35(c) shows pin 3 of the NAND gate shorted to ground within the IC. This output is stuck LOW, and it will not respond to the conditions applied to input pins 1 and 2; in other words, logic inputs  $A$  and  $B$  will have no effect on output  $X$ .

An IC output pin can also be shorted to +5V within the IC, as shown in Figure 4-35(d). This forces the output pin 3 to be stuck HIGH regardless of the state of the signals at the input pins. Note that this type of failure has no effect on the logic signals at the IC inputs.

#### EXAMPLE 4-24

Refer to the circuit of Figure 4-36. A technician uses a logic probe to determine the conditions at the various IC pins. The results are recorded in the figure. Examine these results and determine if the circuit is working properly. If not, suggest some of the possible faults.

FIGURE 4-36 Example 4-24.



#### Solution

Output pin 4 of the INVERTER should be pulsing because its input is pulsing. The recorded results, however, show that pin 4 is stuck LOW. Because this is connected to Z2 pin 1, this keeps the NAND output HIGH. From our preceding discussion, we can list three possible faults that could produce this operation.

First, there could be an internal component failure in the INVERTER that prevents it from responding properly to its input. Second, pin 4 of the INVERTER could be shorted to ground internal to Z1, thereby keeping it stuck LOW. Third, pin 1 of Z2 could be shorted to ground internal to Z2. This would prevent the INVERTER output pin from changing.

In addition to these possible faults, there can be external shorts to ground anywhere in the conducting path between Z1 pin 4 and Z2 pin 1. We will see how to go about isolating the actual fault in Section 4-13.

### Open-Circuited Input or Output

Sometimes the very fine conducting wire that connects an IC pin to the IC's internal circuitry will break, producing an open circuit. Figure 4-37 in Example 4-25 shows this for an input (pin 13) and an output (pin 6). If a

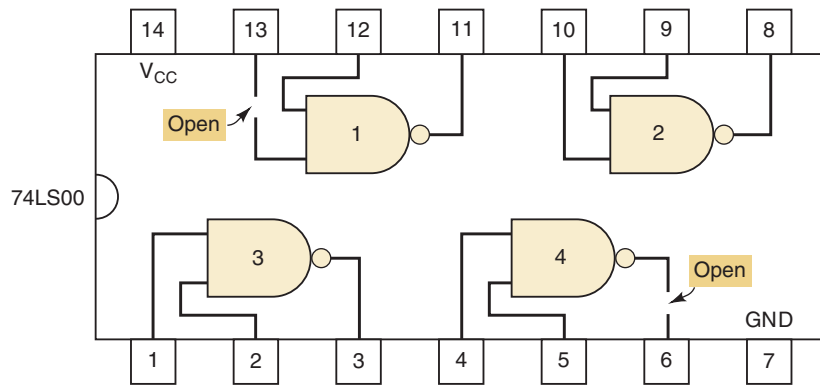
signal is applied to pin 13, it will not reach the NAND-1 gate input and so will not have an effect on the NAND-1 output. The open gate input will be in the floating state. As stated earlier, TTL devices will respond as if this floating input is a logic 1, and CMOS devices will respond erratically and may even become damaged from overheating.

The open at the NAND-4 output prevents the signal from reaching IC pin 6, so there will be no stable voltage present at that pin. If this pin is connected to the input of another IC, it will produce a floating condition at that input.

**EXAMPLE 4-25**

What would a logic probe indicate at pin 13 and at pin 6 of Figure 4-37?

**FIGURE 4-37** An IC with an internally open input will not respond to signals applied to that input pin. An internally open output will produce an unpredictable voltage at that output pin.



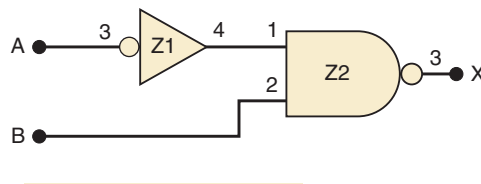
**Solution**

At pin 13, the logic probe will indicate the logic level of the external signal that is connected to pin 13 (which is not shown in the diagram). At pin 6, the logic probe will show no LED lit for an indeterminate logic level because the NAND output level never makes it to pin 6.

**EXAMPLE 4-26**

Refer to the circuit of Figure 4-38 and the recorded logic probe indications. What are some of the possible faults that could produce the recorded results? Assume that the ICs are TTL.

**FIGURE 4-38** Example 4-26.



Pin	Condition
Z1-3	HIGH
Z1-4	LOW
Z2-1	LOW
Z2-2	Pulsing
Z2-3	Pulsing

Note: V<sub>CC</sub> and ground connections to each IC are not shown

**Solution**

Examination of the recorded results indicates that the INVERTER appears to be working properly, but the NAND output is inconsistent with its inputs. The NAND output should be HIGH because its input pin 1 is LOW. This LOW should prevent the NAND gate from responding to the pulses at pin 2. It is probable that this LOW is not reaching the internal NAND

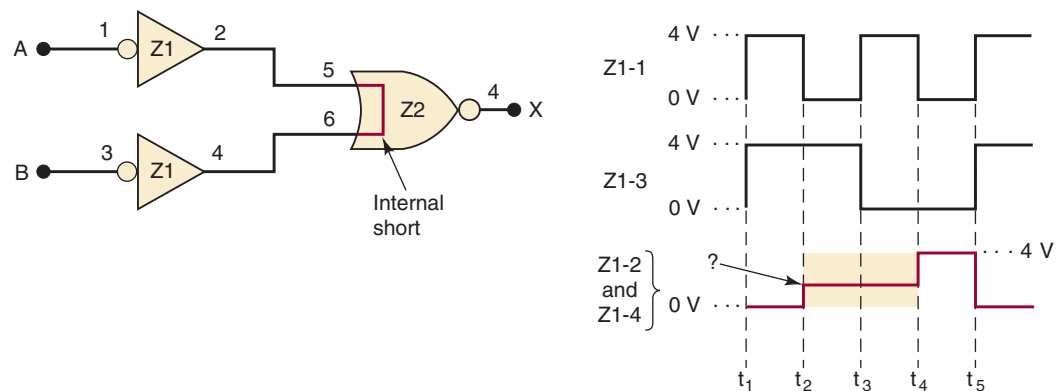
gate circuitry because of an internal open. Because the IC is TTL, this open circuit would produce the same effect as a logic HIGH at pin 1. If the IC had been CMOS, the internal open circuit at pin 1 might have produced an indeterminate output and possible overheating and destruction of the chip.

From our earlier statement regarding open TTL inputs, you might have expected that the voltage of pin 1 of Z2 would be 1.4 to 1.8 V and should have been registered as indeterminate by the logic probe. This would have been true if the open circuit had been *external* to the NAND chip. There is no open circuit between Z1 pin 4 and Z2 pin 1, and so the voltage at Z1 pin 4 is reaching Z2 pin 1, but it becomes disconnected *inside* the NAND chip.

### Short Between Two Pins

An internal short between two pins of an IC will force the logic signals at those pins always to be identical. Whenever two signals that are supposed to be different show the same logic-level variations, there is a good possibility that the signals are shorted together.

Consider the circuit in Figure 4-39, where pins 5 and 6 of the NOR gate are internally shorted together. The short causes the two INVERTER output pins to be connected together so that the signals at Z1 pin 2 and Z1 pin 4 must be identical, even when the two INVERTER input signals are trying to produce different outputs. To illustrate, consider the input waveforms shown in the diagram. Even though these input waveforms are different, the waveforms at outputs Z1-2 and Z1-4 are the same.



**FIGURE 4-39** When two input pins are internally shorted, the signals driving these pins are forced to be identical, and usually a signal with three distinct levels results.

During the interval  $t_1$  to  $t_2$ , both INVERTERS have a HIGH input and both are trying to produce a LOW output, so that their being shorted together makes no difference. During the interval  $t_4$  to  $t_5$ , both INVERTERS have a LOW input and are trying to produce a HIGH output, so that again their being shorted has no effect. However, during the intervals  $t_2$  to  $t_3$  and  $t_3$  to  $t_4$ , one INVERTER is trying to produce a HIGH output while the other is trying to produce a LOW output. This is called signal **contention** because the two signals are “fighting” each other. When this happens, the actual voltage level that appears at the shorted outputs will depend on the internal IC circuitry. For TTL devices, it will usually be a voltage in the high end of the logic 0 range (i.e., close to 0.8 V), although it may also be in the indeterminate range. For CMOS devices, it will often be a voltage in the indeterminate range.

Whenever you see a waveform like the Z1-2, Z1-4 signal in Figure 4-39 with three different levels, you should suspect that two output signals may be shorted together.

### OUTCOME ASSESSMENT QUESTIONS

1. List the different internal digital IC faults.
2. Which internal IC fault can produce signals that show three different voltage levels?
3. What would a logic probe indicate at Z1-2 and Z1-4 of Figure 4-39 if  $A = 0$  and  $B = 1$ ?
4. What is signal contention?

## 4-12 EXTERNAL FAULTS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Identify common faults/failure modes associated with circuit boards and systems.
- Recognize the symptoms of each failure mode.
- Use common methods to diagnose these failure modes.
- Understand loading effects on digital circuits.

We have seen how to recognize the effects of various faults internal to digital ICs. Many more things can go wrong external to the ICs; we will describe the most common ones in this section.

### Open Signal Lines

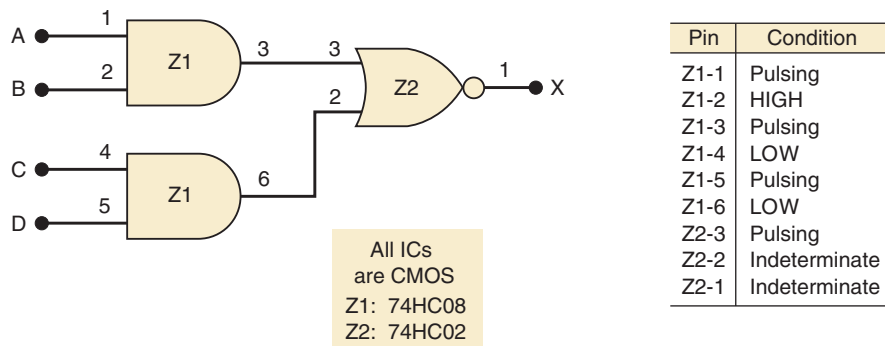
This category includes any fault that produces a break or discontinuity in the conducting path such that a voltage level or signal is prevented from going from one point to another. Some of the causes of open signal lines are:

1. Broken wire
2. Poor solder connection; loose wire-wrap connection
3. Crack or cut trace on a printed circuit board (some of these are hairline cracks that are difficult to see without a magnifying glass)
4. Bent or broken pin on an IC
5. Faulty IC socket such that the IC pin does not make good contact with the socket

This type of circuit fault can often be detected by a careful visual inspection and then verified by disconnecting power from the circuit and checking for continuity (i.e., a low-resistance path) with an ohmmeter between the two points in question.

### EXAMPLE 4-27

Consider the CMOS circuit of Figure 4-40 and the accompanying logic probe indications. What is the most probable circuit fault?

**FIGURE 4-40** Example 4-27.

### Solution

The indeterminate level at the NOR gate output is probably due to the indeterminate input at pin 2. Because there is a LOW at Z1-6, this LOW should also be at Z2-2. Clearly, the LOW from Z1-6 is not reaching Z2-2, and there must be an open circuit in the signal path between these two points. The location of this open circuit can be determined by starting at Z1-6 with the logic probe and tracing the LOW level along the signal path toward Z2-2 until it changes into an indeterminate level.

### Shorted Signal Lines

This type of fault has the same effect as an internal short between IC pins. It causes two signals to be exactly the same (signal contention). A signal line may be shorted to ground or  $V_{CC}$  rather than to another signal line. In those cases, the signal will be forced to the LOW or the HIGH state. The main causes for unexpected shorts between two points in a circuit are as follows:

1. *Sloppy wiring.* An example of this is stripping too much insulation from ends of wires that are in close proximity.
2. *Solder bridges.* These are splashes of solder that short two or more points together. They commonly occur between points that are very close together, such as adjacent pins on a chip.
3. *Incomplete etching.* The copper between adjacent conducting paths on a printed circuit board is not completely etched away.

Again, a careful visual inspection can very often uncover this type of fault, and an ohmmeter check can verify that the two points in the circuit are shorted together.

### Faulty Power Supply

All digital systems have one or more dc power supplies that supply the  $V_{CC}$  and  $V_{DD}$  voltages required by the chips. A faulty power supply or one that is overloaded (supplying more than its rated amount of current) will provide poorly regulated supply voltages to the ICs, and the ICs either will not operate or will operate erratically.

A power supply may go out of regulation because of a fault in its internal circuitry, or because the circuits that it is powering are drawing more current than the supply is designed for. This can happen if a chip or a component has a fault that causes it to draw much more current than normal.

It is good troubleshooting practice to check the voltage levels at each power supply in the system to see that they are within their specified ranges.



It is also a good idea to check them on an oscilloscope to verify that there is no significant amount of ac ripple on the dc levels and to verify that the voltage levels stay regulated during the system operation.

One of the most common signs of a faulty power supply is one or more chips operating erratically or not at all. Some ICs are more tolerant of power supply variations and may operate properly, while others do not. You should always check the power and ground levels at each IC that appears to be operating incorrectly.

### Output Loading

When a digital IC has its output connected to too many IC inputs, its output current rating will be exceeded, and the output voltage can fall into the indeterminate range. This effect is called *loading* the output signal (actually it's overloading the output signal) and is usually the result of poor design or an incorrect connection.

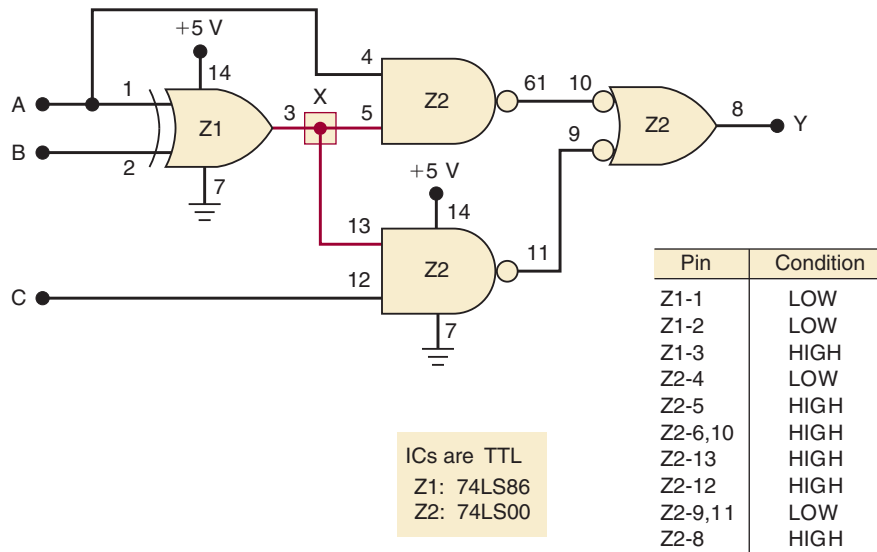
#### EXAMPLE 4-28

Consider the circuit of Figure 4-41. Output Y is supposed to go HIGH for either of the following conditions:

1.  $A = 1, B = 0$  regardless of the level on C
2.  $A = 0, B = 1, C = 1$

You may wish to verify these results for yourself.

**FIGURE 4-41** Example 4-28.



When the circuit is tested, the technician observes that output Y goes HIGH whenever A is HIGH or C is HIGH, regardless of the level at B. She takes logic probe measurements for the condition where  $A = B = 0, C = 1$  and comes up with the indications recorded in Figure 4-41.

Examine the recorded levels and list the possible causes for the malfunction. Then develop a step-by-step procedure to determine the exact fault.

### Solution

All of the NAND gate outputs are correct for the levels present at their inputs. The XOR gate, however, should be producing a LOW at output pin 3 because both of its inputs are at the same LOW level. It appears that Z1-3 is stuck HIGH, even though its inputs should produce a LOW. There are several possible causes for this:

1. An internal component failure in Z1 that prevents its output from going LOW
2. An external short to  $V_{CC}$  from any point along the conductors connected to node X (shaded in the diagram of the figure)
3. Pin 3 of Z1 internally shorted to  $V_{CC}$
4. Pin 5 of Z2 internally shorted to  $V_{CC}$
5. Pin 13 of Z2 internally shorted to  $V_{CC}$

All of these possibilities except for the first one will short node X (and every IC pin connected to it) directly to  $V_{CC}$ .

The following procedure can be used to isolate the fault. This procedure is not the only approach that can be used and, as we stated earlier, the actual troubleshooting procedure that a technician uses is very dependent on what test equipment is available.

1. Check the  $V_{CC}$  and ground levels at the appropriate pins of Z1. Although it is unlikely that the absence of either of these might cause Z1-3 to stay HIGH, it is a good idea to make this check on any IC that is producing an incorrect output.
2. Turn off power to the circuit and use an ohmmeter to check for a short (resistance less than  $1\ \Omega$ ) between node X and any point connected to  $V_{CC}$  (such as Z1-14 or Z2-14). If no short is indicated, the last four possibilities in our list can be eliminated. This means that it is very likely that Z1 has an internal failure and should be replaced.
3. If step 2 shows that there is a short from node X to  $V_{CC}$ , perform a thorough visual examination of the circuit board and look for solder bridges, unetched copper slivers, uninsulated wires touching each other, and any other possible cause of an external short to  $V_{CC}$ . A likely spot for a solder bridge would be between adjacent pins 13 and 14 of Z2. Pin 14 is connected to  $V_{CC}$  and pin 13 to node X. If an external short is found, remove it and perform an ohmmeter check to verify that node X is no longer shorted to  $V_{CC}$ .
4. If step 3 does not reveal an external short, the three possibilities that remain are internal shorts to  $V_{CC}$  at Z1-3, Z2-13, or Z2-5. One of these is shorting node X to  $V_{CC}$ .

To determine which of these IC pins is the culprit, we should disconnect each of them from node X *one at a time* and recheck for a short to  $V_{CC}$  after each disconnection. When the pin that is internally shorted to  $V_{CC}$  is disconnected, node X will no longer be shorted to  $V_{CC}$ .

The process of disconnecting each suspected pin from node X can be easy or difficult depending on how the circuit is constructed. If the ICs are in sockets, all you need to do is to pull the IC from its socket, bend out the suspected pin, and reinsert the IC into its socket. If the ICs are soldered into a printed circuit board, you will have to cut the trace that is connected to the pin and repair the cut trace when you are finished.

---

Example 4-28, although fairly simple, shows you the kinds of thinking that a troubleshooter must employ to isolate a fault. You will have the opportunity to begin developing your own troubleshooting skills by working on many end-of-chapter problems that have been designated with a **T** for troubleshooting.

#### OUTCOME ASSESSMENT QUESTIONS

1. What are the most common types of external faults?
2. List some of the causes of signal-path open circuits.
3. What symptoms are caused by a faulty power supply?
4. How might loading affect an IC output voltage level?

### 4-13 TROUBLESHOOTING PROTOTYPED CIRCUITS

#### OUTCOMES

*Upon completion of this section, you will be able to:*

- Develop systematic troubleshooting techniques to efficiently isolate problems when building and testing circuit designs.
- Recognize the unique faults that are often present in prototyped circuits.

Troubleshooting of circuits can be divided into two major categories:

1. Finding failures in a system that was previously working (i.e., repair)
2. Finding faults in a system that is being tested for the first time (i.e., prototyping)

Many of the skills required to troubleshoot in these two categories are the same but there are also key differences. The number of failure modes that are possible is much higher in the prototyping category because at this point we have not proven that the design should work. It is wise to do everything possible to validate your design before you prototype in order to save yourself a lot of frustration from building a circuit from a flawed design. For example, suppose a mistake is made in the simplification process (either by Boolean algebra or by K-mapping) that results in an incorrect equation. If this error is not realized until after building the prototype circuit, it could mean more than just rewiring. It may mean that you need more ICs or even more breadboard/circuit board space. Lots of effort building the initial prototype may have been wasted.

This is why analysis of a circuit design is so important before building a prototype. This analysis may simply mean working the problem backward. In other words, take your circuit that you have designed and intend to build and do a truth table analysis of its operation. If these results match the truth table from the design process, then it is reasonably certain that there was not a fundamental error in the design. There still could be an oversight of practical limitations of the circuits that cause problems such as propagation delays or spurious glitches. Circuit simulators that run on computers have become very popular for this reason. Most simulators are able to take into account the many practical limitations of the devices in the design and help to point out problems before the circuit is implemented in hardware.

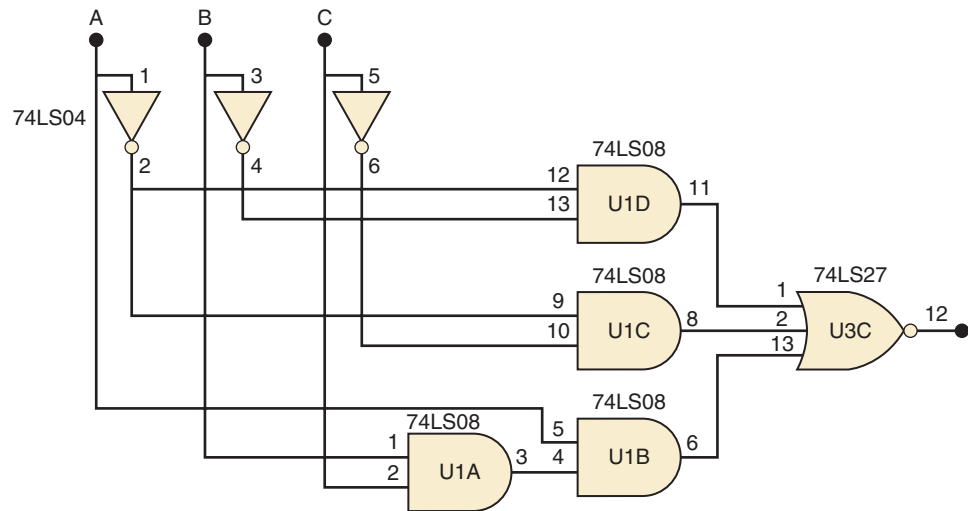
Even when the design has been thoroughly analyzed and verified, there is still a great likelihood that errors could be made in fabrication of the prototype circuits. Wires can be crossed, or connected to the wrong point in a circuit. ICs can be inserted in the wrong place. Incorrect pin numbers on a schematic diagram can be particularly frustrating. Nobody makes these mistakes on purpose so it is particularly hard to recognize all the assumptions that are in your mind that may not be valid. A systematic process can point out your incorrect assumptions and mistakes by isolating the fault to a very small area. With a few simple tests in this small area, the actual fault becomes obvious. Always remember that when you are convinced that a circuit should work, but yet it does not work, there must be a wrong assumption. Something that you think you know to be true is really false or vice versa.

The first principle in this technique is to *eliminate* as many possible locations of the fault as possible. Then focus on the parts that have not been eliminated. The second is essentially the scientific method of hypothesizing and testing. Troubleshooting combinational logic circuits is a tremendous way to exercise and master these two very critical problem solving skills.

To demonstrate this process, let's assume that a design has been completed and the analysis/simulation has verified that the design should do what we expect. The circuit in Figure 4-42 is the result of the design. This circuit is built and when tested, it is found that many of the outputs match the original truth table but some do not. At this point, we will not give an actual example of an error in this circuit, but rather describe a process that will work for any error. Remember the objective is to quickly and efficiently isolate the problem. Then we will apply these techniques to some common mistakes to show how an error can be isolated.

1. Identify the first input combination in the table that produces an incorrect output. The inputs must be put in this state in order to test for the failure. One of two possibilities exists at this point: the output of the OR gate is HIGH but it should be LOW, or the output is LOW but it should be HIGH.
  2. If the output is LOW but according to the truth table it should be HIGH, the very nature of a NOR gate can help us isolate the problem. The HIGH output of a NOR gate can be considered its unique state because it can only happen when all of its inputs are LOW. If one of the NOR gate inputs is actually HIGH, then we know the problem is in that branch of the circuit and we have eliminated  $\frac{2}{3}$  of the circuit. The hypothesis is that one of the AND gates is supplying a HIGH when it should be LOW. Use the logic probe to test IC U3C pins 1, 2, and 13. If they are all LOW, then our hypothesis is wrong which must mean the problem is with the NOR gate. The tests described in the previous sections can be used to locate the problem within this IC. On the other hand, if one of the input pins of U3C is HIGH, we must find out why. Either way, the space containing the fault has gotten much smaller.
  3. Let's assume that one of the pins on the NOR gate (e.g., pin 13) was HIGH. Look at the diagram and notice that U1B AND gate pin 6 should be driving the NOR gate pin 13. Rather than assuming that the output (pin 6) of U1B is HIGH, use the logic probe to test it. If this output is LOW but the other end of the wire is HIGH, there must not actually be wire in between the two. This is exactly the type of error that is common when prototyping. Tracing the circuit between the two should locate the problem.
-

**FIGURE 4-42** An example for troubleshooting a prototype circuit.



- Assuming the output pin 6 of U1B AND gate is HIGH (remember it should be LOW with these inputs) the logic levels on the inputs should be examined. Hypothetically, at least one of the inputs to this AND gate should be LOW, but which one? The way to find out is to examine the present input combination, find the input that is LOW, and look on the schematic to determine which pin it should be connected to. Probe that pin and see if it is LOW. Move the input HIGH, LOW, HIGH, LOW and watch your logic probe. This will verify if the input is properly wired to the gate. If this is correct, then probe the other inputs to the AND gate while changing the input variables that drive them. Trace the AND gate input that is not responding properly to the logic circuits that drive it. If all are correct and at least one of the inputs is LOW, the AND gate has a failure or is being loaded. Probe the AND gate output and disconnect the wire that it drives. Did the logic level change? If so, trace the load problem. If not, troubleshoot the AND gate chip (check  $V_{CC}$  and ground with the probe and if okay, replace the IC).

Now let's go back to the truth table. We will explore two possible causes of this effect and see if the approach described above is able to identify them.

**Fault scenario #1:** U1B pin 5 is actually connected to  $\bar{A}$ , rather than A.

Figure 4-43(a) shows the design requirements and the test results for this circuit.

**FIGURE 4-43** Figure 4-42 test results: (a) fault scenario 1; (b) fault scenario 2.

A	B	C	Required Results	Test Results
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	1

(a)

A	B	C	Required Results	Test Results
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

(b)

Troubleshooting steps to isolate the problem.

1. The fourth line in the table has the error. Set input switches to  $A = 0$ ,  $B = 1$ ,  $C = 1$ .
2. Probe the output of U3C NOR gate. It is LOW but should be HIGH.
3. Check the inputs to the NOR gate. Pin 13 is HIGH (but should be LOW).
4. U1B AND gate pin 6 is HIGH.
5. Looking at the switches, U1B pin 5 should be LOW because  $A$  is LOW. Probing U1B pin 5 shows that it is HIGH. Moving switch  $A$  to HIGH makes pin 5 go LOW.
6. Conclusion: Pin 5 of U1B is not the same as  $A$ ; it is the inverse of  $A$ . Tracing the wire shows it was accidentally wired to  $\bar{A}$  (pin 2 of the inverter). Note that fixing this problem also resolves the last line on the truth table.

**Fault scenario #2.** When inserting the IC U1 into the proto-board, pin 13 bent and curled up under the IC coming in contact with pin 14. Visually, it looks fine but electrically pin 13 of U1 is shorted to  $V_{CC}$ .

Figure 4-43(b) shows the design requirements and the test results for this scenario.

Troubleshooting steps to isolate the problem.

1. The fourth line in the table has the error. Set input switches to  $A = 0$ ,  $B = 1$ ,  $C = 1$ .
2. Probe the output of U3C NOR gate. It is LOW but should be HIGH.
3. None of the inputs to U3C should be HIGH. Probing them shows pin 1 is HIGH.
4. U1D AND gate pin 11 is HIGH.
5. Looking at the switches, U1D pin 13 should be LOW because  $B$  is HIGH. Probing U1D pin 13 shows that it is HIGH. Moving switch  $B$  to LOW has no effect on pin 13. It remains HIGH, which is wrong.
6. Probing the output pin 4 of the inverter for input  $B$  shows that it follows the complement of switch  $B$ , going HIGH and LOW. This is correct.
7. Conclusion: The wire from Inverter pin 4 must not be connected to U1D pin 13 as the diagram shows. Visual inspection indicates the wire appears to be connected properly. Yet pin 13 on IC U1D seems to be HIGH all the time. Probing the other holes of the protoboard for pin 13 indicates the signal is correct (the complement of  $B$ ). There must be a disconnect between the protoboard and the IC. When the chip is removed the problem becomes obvious: Pin 13 is bent and not making contact with the circuit board.

**Generalization:** If the output of a logic gate should be in its unique state (only one input combination produces this state) but it is actually in the other state, then one of the gate's inputs is in the wrong state and will be easy to identify. Pursue the problem. This eliminates the other inputs and the circuits that drive them.

To demonstrate, assume the output of an AND gate is LOW and should be HIGH

**Hypothesize:** One of its inputs must be LOW. Test this hypothesis.

If false (all inputs really are HIGH), the problem is in the gate itself or its load.

If true, then find the input that is LOW and determine why.

If the output of a gate should not be in its unique state, but it is in that state, then any one of the inputs to that gate could be incorrect. For example, when an AND gate output should be LOW but it is really HIGH (only HIGH when all inputs are HIGH), it is not obvious which input should be LOW. In this case, use your knowledge of the circuit driving these inputs to identify which input should be LOW under present conditions and trace it to its source.

To demonstrate, assume the output of an AND gate is HIGH and should be LOW

**Hypothesize:** All of its inputs must be HIGH. Test this hypothesis.

If false (one of the inputs really is LOW), the problem is in the gate itself or its load. Focus on potential problems with this IC.

If true (all inputs are HIGH), then at least one of them is incorrect and should be LOW. Look at the logic that drives each input to determine which should be LOW. Focus on problems in this driver circuit.

Whenever a driven node is identified as incorrect, trace back to the output of the gate that drives it. If the two ends of a wire do not agree, find the wiring error. If the output of the driving gate agrees with the input of the driven gate, then verify that the inputs to the driving gate are correct. Assuming they are correct, test for loading problems or shorts to  $V_{CC}$  or GND by disconnecting the load from the output of the driving gate. If unloading the output does not produce the proper logic level at the output of the driving gate, then there is a fault in the driving gate.

Repeated application of this approach will allow you to efficiently and systematically isolate the problem by moving from the final output back through only the branches of the circuit that are affected by the fault.

### OUTCOME ASSESSMENT QUESTIONS

1. What is the unique state of an AND gate output?
2. If the output of an AND gate is not in its unique state, but should be, what is the most likely problem?
3. What is the unique state of an OR gate output?
4. If the output of an OR gate is not in its unique state, but should be, what is the most likely problem?
5. If a circuit node has the incorrect logic level, why would you test the other end of the wire that is supposed to drive that node?
6. If you suspect an output is incorrect because of a loading problem (in the circuit it is driving), how can you prove it?

## 4-14 PROGRAMMABLE LOGIC DEVICES\*

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Explain what is done inside a PLD to “program” it.
- Describe methods used to accomplish the programming process for PLDs.

\*All sections covering PLDs may be skipped without loss of continuity.

- Define common terms associated with PLD development.
- Define hierarchical design techniques.

In the previous sections, we briefly considered the class of ICs known as programmable logic devices. In Chapter 3, we introduced the concept of describing a circuit's operation using a hardware description language. In this section, we will explore these topics further and become prepared to use the tools of the trade to develop and implement digital systems using PLDs. Of course, it is impossible to understand all the complex details of how a PLD works before grasping the fundamentals of digital circuits. As we examine new fundamental concepts, we will expand our knowledge of the PLDs and the programming methods. The material is presented in such a way that anyone who is not interested in PLDs can easily skip over these sections without loss of continuity in the coverage of the basic principles.

Let's review the process we covered earlier of designing combinational digital circuits. The input devices are identified and assigned an algebraic name like  $A$ ,  $B$ ,  $C$ , or  $LOAD$ ,  $SHIFT$ ,  $CLOCK$ . Likewise, output devices are given names like  $X$ ,  $Z$ , or  $CLOCK\_OUT$ ,  $SHIFT\_OUT$ . Then a truth table is constructed that lists all the possible input combinations and identifies the required state of the outputs under each input condition. The truth table is one way of describing how the circuit is to operate. Another way to describe the circuit's operation is the Boolean expression. From this point the designer must find the simplest algebraic relationship and select digital ICs that can be wired together to implement the circuit. You have probably experienced that these last steps are the most tedious, time consuming, and prone to errors.

Programmable logic devices allow most of these tedious steps to be automated by a computer and PLD *development software*. Using programmable logic improves the efficiency of the design and development process. Consequently, most modern digital systems are implemented in this way. The job of the circuit designer is to identify inputs and outputs, specify the logical relationship in the most convenient manner, and select a programmable device that is capable of implementing the circuit at the lowest cost. The concept behind programmable logic devices is simple: put lots of logic gates in a single IC and control the interconnection of these gates electronically.

## PLD Hardware

Recall from Chapter 3 that many digital circuits today are implemented using programmable logic devices (PLDs). These devices are configured electronically and their internal circuits are "wired" together electronically to form a logic circuit. This programmable wiring can be thought of as thousands of connections that are either connected (1) or not connected (0). It is very tedious to try to configure these devices by manually placing 1s and 0s in a grid, so the next logical question is, "How do we control the interconnection of gates in a PLD electronically?"

A common method of connecting one of many signals entering a network to one of many signal lines exiting the network is a switching matrix. Refer back to Figure 3-44, where this concept was introduced. A matrix is simply a grid of conductors (wires) arranged in rows and columns. Input signals are connected to the columns of the matrix, and the outputs are connected to the rows of the matrix. At each intersection of a row and a column is a switch that can electrically connect that row to that column. The switches that connect rows to columns can be mechanical switches, fusible



links, electromagnetic switches (relays), or transistors. This is the general structure used in many applications and will be explored further when we study memory devices in Chapter 12.

PLDs also use a switch matrix that is often referred to as a programmable array. By deciding which intersections are connected and which ones are not, we can “program” the way the inputs are connected to the outputs of the array. In Figure 4-44, a programmable array is used to select the inputs for each AND gate. Notice that in this simple matrix, we can produce any logical product combination of variables A, B at any of the AND gate outputs. A matrix or programmable array such as the one shown in the figure can also be used to connect the AND outputs to OR gates. The details of various PLD architectures will be covered thoroughly in Chapter 13.

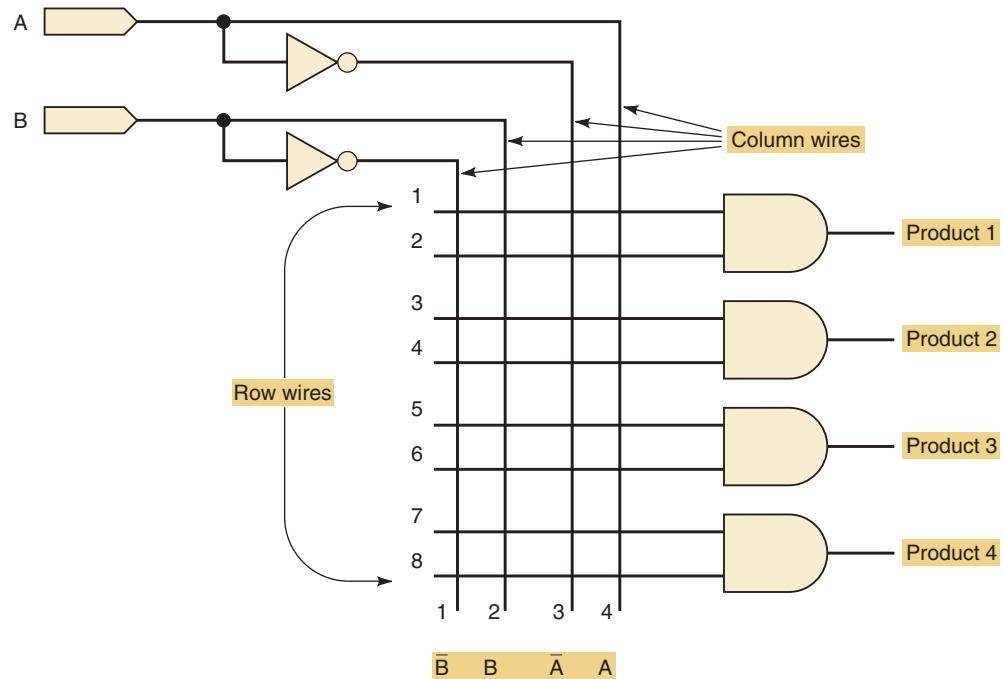


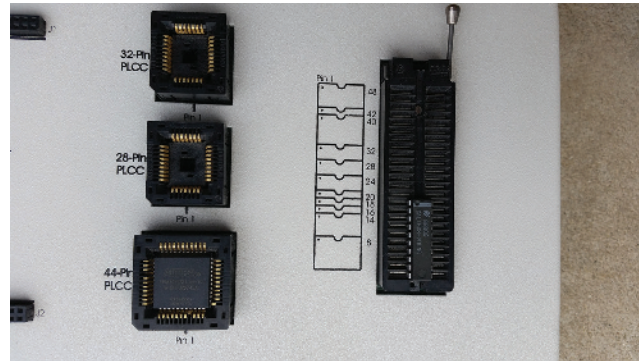
FIGURE 4-44 A programmable array for selecting inputs as product terms.

### Programming a PLD

There are two ways to “program” a PLD IC. Programming means making the actual connections in the array. In other words, it means determining which of those connections are supposed to be open (0) and which are supposed to be closed (1). One method involves removing the PLD IC chip from its circuit board. The chip is then placed in a special fixture called an all-purpose **programmer**. Most modern programmers are connected to a personal computer that is running software containing libraries of information about the many types of programmable devices available.

The programming software is invoked (called up and executed) on the PC to establish communication with the programmer. This software allows the user to set up the programmer for the type of device that is to be programmed, check if the device is blank, read the state of any programmable connection in the device, and provide instructions for the user to program a chip. Ultimately, the part is placed into a special socket that allows you to drop the chip in and then clamp the contacts onto the pins. This is called a **zero insertion force (ZIF) socket**. Figure 4-45 shows some typical

**FIGURE 4-45** ZIF sockets for DIP and PLCC packages typically found on Universal programmers.



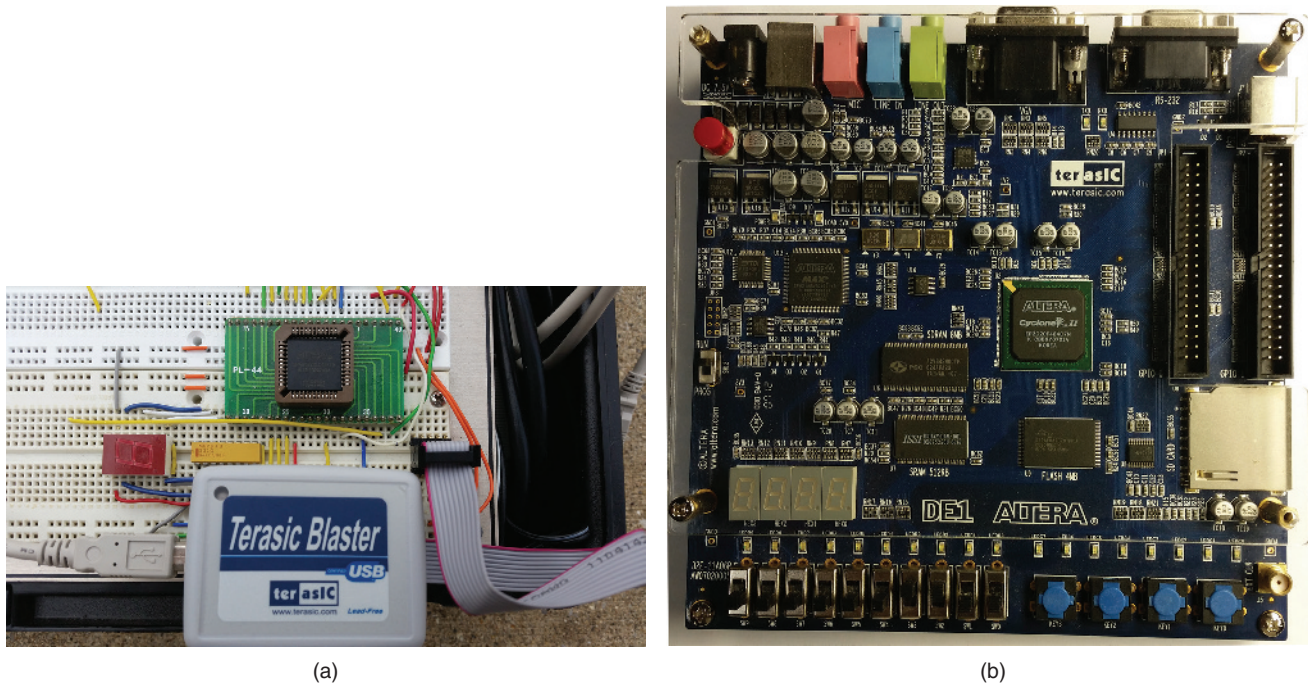
ZIF sockets used for DIP and PLCC packages. *Universal programmers* that can program any type of programmable device are available from numerous manufacturers.

Fortunately, as programmable parts began to proliferate, manufacturers saw the need to standardize pin assignments and programming methods. As a result, the Joint Electronic Device Engineering Council (**JEDEC**) was formed. One of the results was JEDEC standard 3, a format for transferring programming data for PLDs, independent of the PLD manufacturer or programming software. Pin assignments for various IC packages were also standardized, making universal programmers less complicated. Consequently, programming fixtures can program numerous types of PLDs. The software that allows the designer to specify a configuration for a PLD simply needs to produce an output file that conforms to the JEDEC standards. Then this JEDEC file can be loaded into any JEDEC-compatible PLD programmer that is capable of programming the desired type of PLD.

The more common method used today is referred to as in-system programming (**ISP**). As its name implies, the chip does not need to be removed from its circuit for storage of the programming information. A standard interface has been developed by the Joint Test Action Group (**JTAG**). The interface was developed to allow ICs to be tested without actually connecting test equipment to every pin of the IC. It also allows for internal programming. Four pins on the IC are used as a portal to store data and retrieve information about the inner condition of the IC. Many ICs, including PLDs and microcontrollers, are manufactured today to include the JTAG interface. An interface cable connects the four JTAG pins on the IC to an output port (typically USB) of a personal computer. Software running on the PC establishes contact with the IC and loads the information in the proper format. Figure 4-46(a) shows a typical USB to JTAG programmer. Development boards such as the one shown in Figure 4-46(b) typically include the USB to JTAG interface circuits that allow easy programming using the development software.

## Development Software

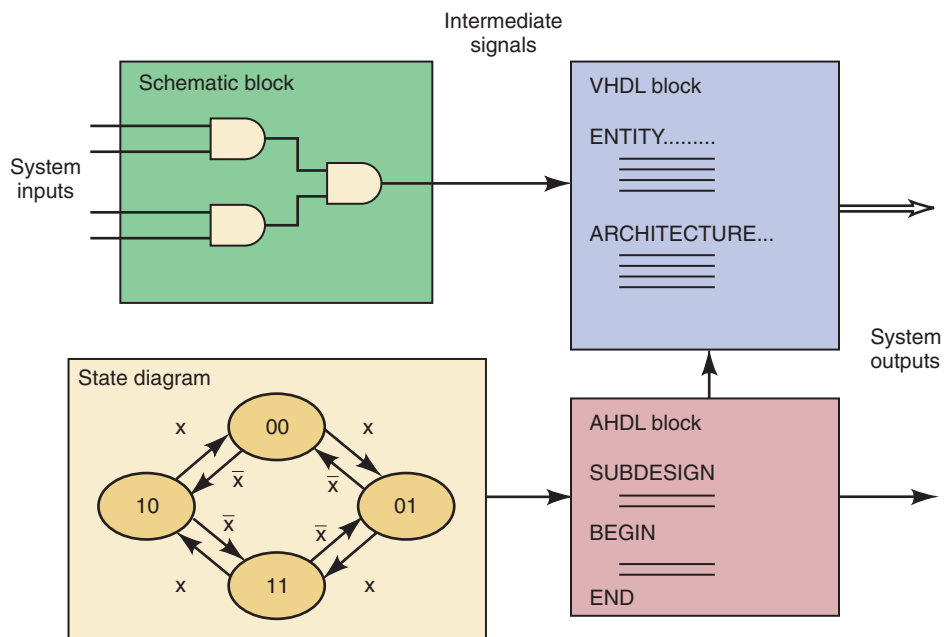
We have examined several methods of describing logic circuits including schematic capture, logic equations, timing diagrams, truth tables, and HDL. We also described the fundamental methods of storing 1s and 0s into a PLD IC to connect the logic circuits in the desired way. The biggest challenge in getting a PLD programmed is converting from any form of description into the array of 1s and 0s. Fortunately, this task is accomplished quite easily by a computer running the development software. The development software that we will be referring to and using for examples is produced by Altera.



**FIGURE 4-46** JTAG programming: (a) a typical USB-JTAG programming interface; (b) a typical development board includes USB to JTAG interface.

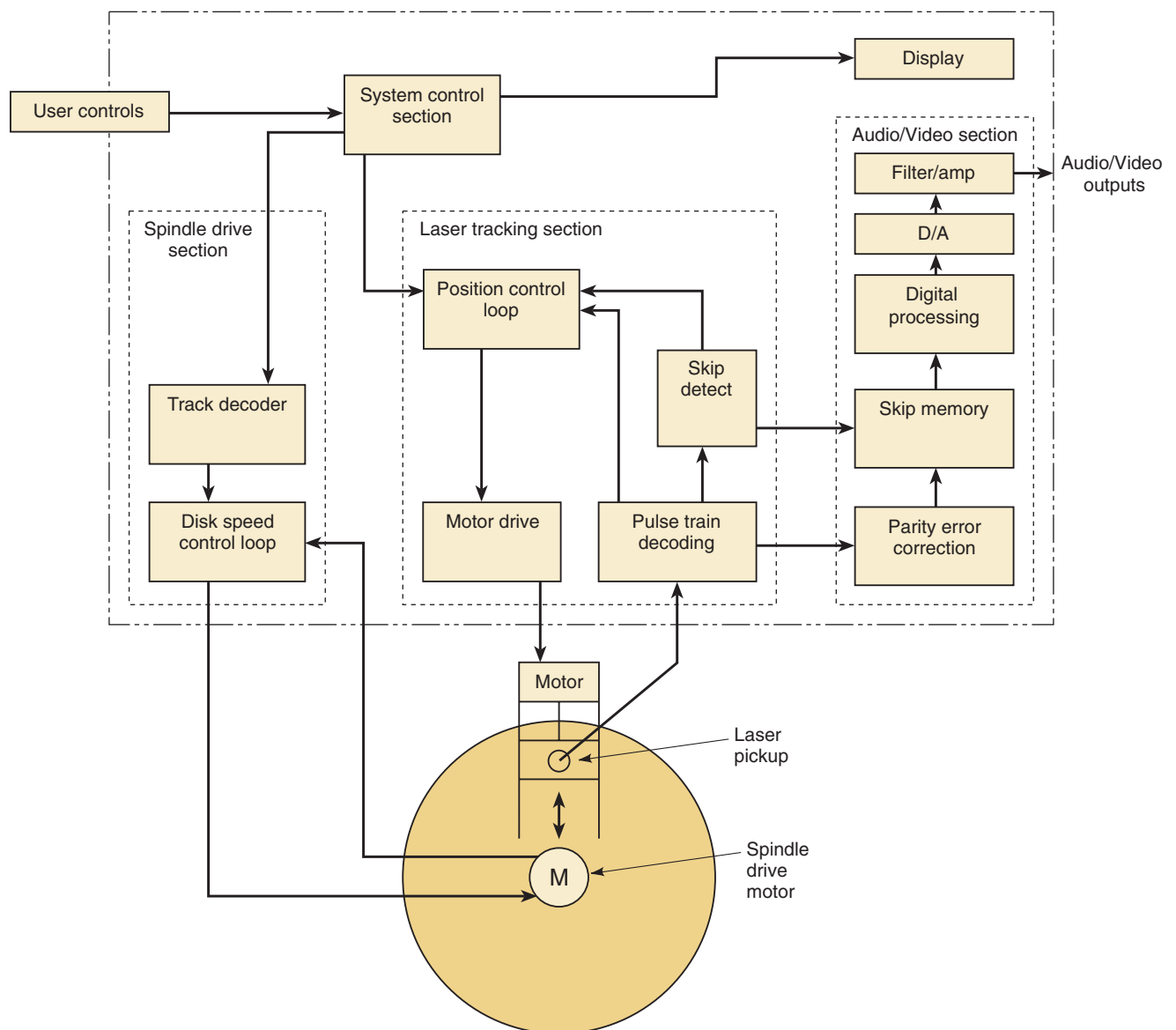
This software allows the designer to enter a circuit description in any one of the many ways we have been discussing: graphic design files (schematics), AHDL, and VHDL. It also allows the use of another HDL, called Verilog, and the option of describing the circuit in other ways such as state transition diagrams. Circuit blocks described by any of these methods can also be “connected” together to implement a much larger digital system, as shown in Figure 4-47. Any logic diagram found in this text can be redrawn using the schematic entry tools in the Altera software to create a graphic design

**FIGURE 4-47** Combining blocks that were developed using different description methods.



file. We will not focus on graphic design entry in this text because it is quite straightforward to pick up these skills in the laboratory. We will focus our examples on the methods that allow us to use HDL as an alternate means of describing a circuit. For more information on the Altera software, visit the Altera web site.

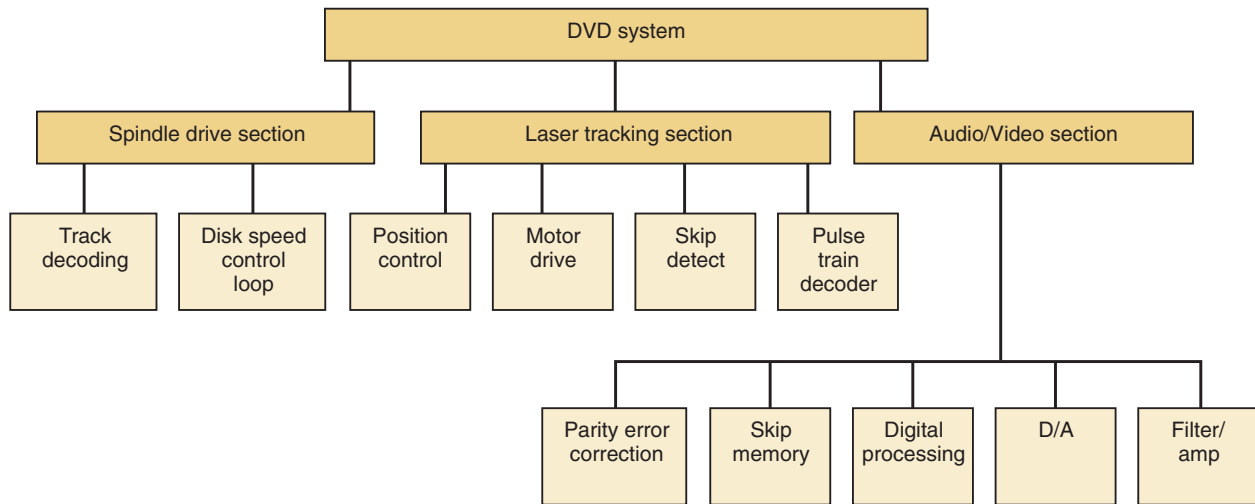
This concept of using building blocks of circuits is called **hierarchical design**. Small, useful logic circuits can be defined in whatever manner is most convenient (graphic, AHDL, VHDL, etc.) and then combined with other circuits to form a large section of a project. Sections can be combined and connected with other sections to form the whole system. Figure 4-48 shows the hierarchical structure of a DVD player using a block diagram. The outer box encloses the entire system. The dashed lines identify each major subsection, and each subsection contains individual circuits. Although it is not shown in this diagram, each circuit may be made up of smaller building blocks of common digital circuits. The Altera development software makes this type of modular, hierarchical design and development easy to accomplish.



**FIGURE 4-48** Block diagram of a DVD player.

## Design and Development Process

Another way you might see the hierarchy of a system like the DVD player just described is shown in Figure 4-49. The top level represents the entire system. It is made up of three subsections, each of which in turn is made up of the smaller circuits shown. Notice that this diagram does not show how the signals flow throughout the system but clearly identifies the various levels of the hierarchical structure of the project.



**FIGURE 4-49** An organizational hierarchy chart.

This type of diagram has led to the name for one of the most common methods of design: **top-down**. With this design approach, you start with the overall description of the entire system, such as the top box in Figure 4-49. Then you define several subsections that will make up the system. The subsections are further refined into individual circuits connected together. Every one of these hierarchy levels has defined inputs, outputs, and behavior. Each can be tested individually before it is connected to the others.

After defining the blocks from the top down, the system is built from the bottom up. Each block in this system design has a design file that describes it. The lowest level blocks must be designed by opening a design file and writing a description of its operation. The designed block is then compiled using the development tools. The compiling process determines if you have made errors in your syntax. Until your syntax is correct, the computer cannot possibly translate your description into its proper form. After it has been compiled with no syntax errors, it should be tested to see if it operates correctly. Development systems offer simulator programs that run on the PC and simulate the way your circuit responds to inputs. A simulator is a computer program that calculates the correct output logic states based on a description of the logic circuit and the current inputs. A set of hypothetical inputs and their corresponding correct outputs are developed that will prove the block works as expected. These hypothetical inputs are often called **test vectors**. Thorough testing during simulation greatly increases the likelihood of the final system working reliably. Figure 4-50 shows the simulation file for the circuit described in Figure 3-13(a) of Chapter 3. Inputs  $a$ ,  $b$ , and  $c$  were entered as test vectors, and the simulation produced output  $y$ .

When the designer is satisfied that the design works, the design can be verified by actually programming a chip and testing. For a complex PLD,

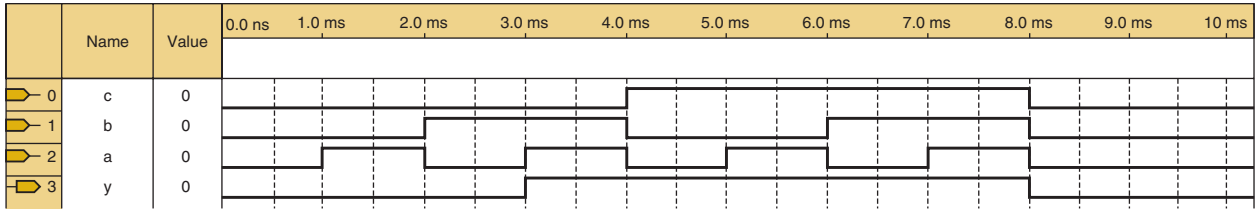
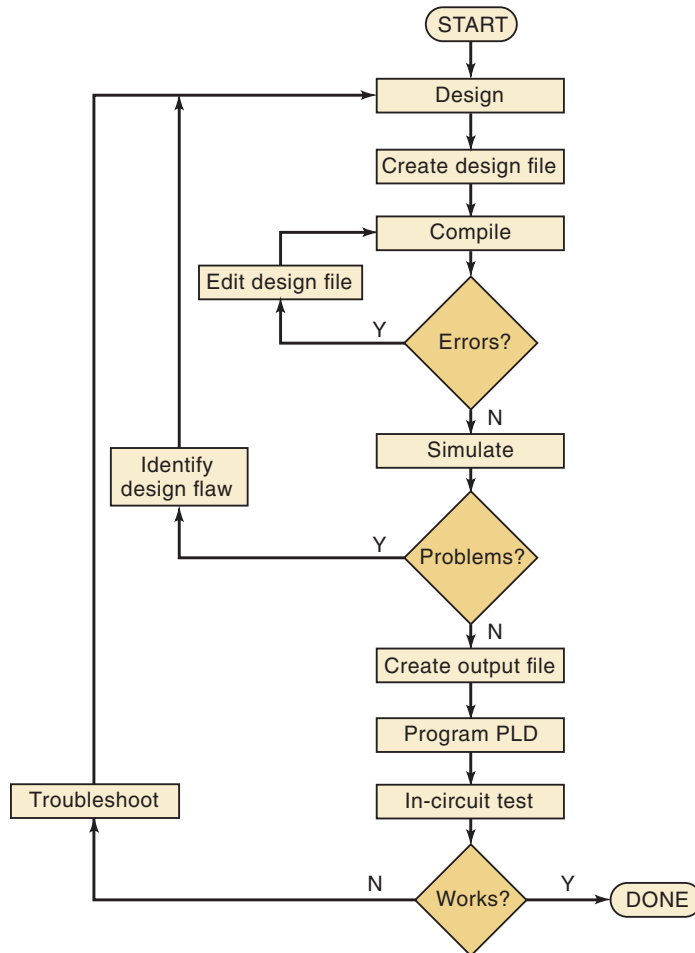


FIGURE 4-50 A timing simulation of a circuit described in HDL.

the designer can either let the development system assign pins and then lay out the final circuit board accordingly, or specify the pins for each signal using the software features. If the compiler assigns the pins, the assignments can be found in the report file or pin-out file, which provides many details about the implementation of the design. If the designer specifies the pins, it is important to know the constraints and limitations of the chip's architecture. These details will be covered in Chapter 13. The flowchart of Figure 4-51 summarizes the design process for designing each block.

After each circuit in a subsection has been tested, all can be combined and the subsection can be tested following the same process that was used for the small circuits. Then the subsections are combined and the system is tested. This approach lends itself very well to a typical project environment, where a team of people are working together, each responsible for his or her own circuits and sections that will ultimately come together to make up the system.

FIGURE 4-51 PLD development cycle flowchart.



**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is actually being “programmed” in a PLD?
2. What bits (column, row) in Figure 4-44 must be connected to make Product 1 =  $AB$ ?
3. What bits (column, row) in Figure 4-44 must be connected to make Product 3 =  $A\bar{B}$ ?
4. Define hierarchical design.
5. Most modern PLDs are programmed in-circuit using the \_\_\_\_\_ standard interface.

## 4-15 REPRESENTING DATA IN HDL

### OUTCOMES

*Upon completion of this section, you will be able to:*

- List the number systems that can be used to represent data values in AHDL and VHDL.
- Correctly use the syntax of AHDL and VHDL.
- Use bit arrays in AHDL and VHDL.
- Declare bit arrays in AHDL and VHDL.
- Select the correct data types in VHDL.
- Use IEEE standard data types where applicable.

Numeric data can be represented in various ways. We have studied the use of the hexadecimal number system as a convenient way to communicate bit patterns. We naturally prefer to use the decimal number system for numeric data, but computers and digital systems can operate only on binary information, as we studied in previous chapters. When we write in HDL, we often need to use these various number formats, and the computer must be able to understand which number system we are using. So far in this text, we have used a subscript to indicate the number system. For example,  $101_2$  was binary,  $101_{16}$  was hexadecimal, and  $101_{10}$  was decimal. Every programming language and HDL has its own unique way of identifying the various number systems, generally done with a prefix to indicate the number system. In most languages, a number with no prefix is assumed to be decimal. When we read one of these number designations, we must think of it as a symbol that represents a binary bit pattern. These numeric values are referred to as scalars or **literals**. Table 4-8 summarizes the methods of specifying values in binary, hex, and decimal for AHDL and VHDL.

**TABLE 4-8** Designating number systems in HDL.

Number System	AHDL	VHDL	Bit Pattern	Decimal Equivalent
Binary	B"101"	B"101"	101	5
Hexadecimal	H"101"	X"101"	10000001	257
Decimal	101	101	1100101	101

**EXAMPLE 4-29**

Express the following bit pattern's numeric value in binary, hex, and decimal using AHDL and VHDL notation:

11001

**Solution**

Binary is designated the same in both AHDL and VHDL: **B** "11001".

Converting the binary to hex, we have  $19_{16}$ .

In AHDL: **H** "19"

In VHDL: **X** "19"

Converting the binary to decimal, we have  $25_{10}$ .

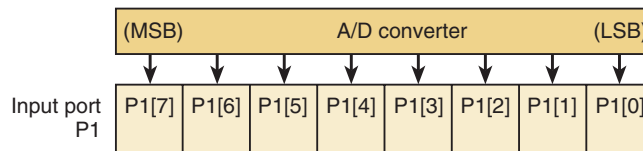
Decimal is designated the same in both AHDL and VHDL: 25.

**Bit Arrays/Bit Vectors**

In Chapter 3, we declared names for inputs to and outputs of a very simple logic circuit. These were defined as bits, or single binary digits. What if we want to represent an input, output, or signal that is made up of several bits? In an HDL, we must define the type of the signal and its range of acceptable values.

To understand the concepts used in HDLs, let's first consider some conventions for describing bits of binary words in common digital systems. Suppose we have an eight-bit number representing the current temperature, and the number is coming into our digital system through an input port that we have named P1, as shown in Figure 4-52. We can refer to the individual bits of this port as P1 bit 0 for the least significant bit, on up to P1 bit 7 for the most significant bit.

**FIGURE 4-52** Bit array notation.



We can also describe this port by saying that it is named P1, with bits numbered 7 down to 0. The terms **bit array** and **bit vector** are often used to describe this type of data structure. It simply means that the overall data structure (eight-bit port) has a name (P1) and that each individual element (bit) has a unique **index** number (0–7) to describe that bit's position (and possibly its numeric weight) in the overall structure. The HDLs and computer programming languages take advantage of this notation. For example, the third bit from the right is designated as P1[2], and it can be connected to another signal bit by using an assignment operator.

**EXAMPLE 4-30**

Assume there is an eight-bit array named P1, as shown in Figure 4-52, and another four-bit array is named P5.

- Write the bit designation for the most significant bit of P1.
- Write the bit designation for the least significant bit of P5.
- Write an expression that causes the least significant bit of P5 to drive the most significant bit of P1.



**Solution**

- (a) The name of the port is P1 and the most significant bit is bit 7. The proper designation for P1 bit 7 is P1[7].
- (b) The name of the port is P5 and the least significant bit is bit 0. The proper designation for P5 bit 0 is P5[0].
- (c) The driving signal (source) is placed on the right side of the assignment operator, and the driven signal (destination) is placed on the left: P1[7] = P5[0];

**AHDL BIT ARRAY DECLARATIONS**

In AHDL, port *p1* of Figure 4-52 is defined as an eight-bit input port, and the value on this port can be referred to using any number system, such as hex, binary, or decimal. The syntax for AHDL uses a name for the bit vector followed by the range of index designations, which are enclosed in square brackets. This declaration is included in the SUBDESIGN section. For example, to declare an eight-bit input port called *p1*, you would write

```
p1[7..0] :INPUT; --define an 8-bit input port
```

**EXAMPLE 4-31**

Declare a four-bit input named *keypad* using AHDL.

**Solution**

```
keypad[3..0] :INPUT;
```

Intermediate variables can also be declared as an array of bits. As with single bits, they are declared just after the I/O declarations in SUBDESIGN. As an example, the eight-bit temperature port *p1* can be assigned (connected) to a node named *temp*, as follows:

```
VARIABLE temp[7..0] :NODE;
BEGIN
    temp[] = p1[];
END;
```

Notice that the input port *p1* has the data applied to it, and it is driving the signal wires named *temp*. Think of the term on the right of the equals sign as the source of the data and the term on the left as the destination. The empty brackets [] mean that each of the corresponding bits in the two arrays are being connected. Individual bits can also be “connected” by specifying the bits inside the brackets. For example, to connect only the least significant bit of *p1* to the LSB of *temp*, the statement would be *temp[0] = p1[0];*

## VHDL BIT VECTOR DECLARATIONS

In VHDL, port *p1* of Figure 4-52 is defined as an eight-bit input port, and the value on this port can be referred to using only binary literals. The syntax for VHDL uses a name for the bit vector followed by the mode (:IN), the type (**BIT\_VECTOR**), and the range of index designations, which are enclosed in parentheses. This declaration is included in the ENTITY section. For example, to declare an eight-bit input port called *p1*, you would write

```
PORT (p1 :IN BIT_VECTOR (7 DOWNTO 0));
```

### EXAMPLE 4-32

Declare a four-bit input named *keypad* using VHDL.

#### Solution

```
PORT(keypad :IN BIT_VECTOR (3 DOWNTO 0));
```

Intermediate signals can also be declared as an array of bits. As with single bits, they are declared just inside the ARCHITECTURE definition. As an example, the eight-bit temperature on port *p1* can be assigned (connected) to a signal named *temp*, as follows:

```
SIGNAL    temp :BIT_VECTOR (7 DOWNTO 0);
BEGIN
    temp <= p1;
END;
```

Notice that the input port *p1* has the data applied to it, and it is driving the signal wires named *temp*. No elements in the bit vector are specified, which means that all the bits are being connected. Individual bits can also be “connected” using signal assignments and by specifying the bit numbers inside parentheses. For example, to connect only the least significant bit of *p1* to the LSB of *temp*, the statement would be `temp(0) <= p1(0);`.

## VHDL Data Objects

A very important part of VHDL that must be mastered is the use of data objects. As this name implies, a data object can be assigned values that are referred to as “data.” The nature and purpose of each data object is very important in VHDL and must be assigned a specific “type.” Examples of data objects are signals, variables, and constants.

Signals are like wires in a hardware circuit that can have a value of 1 or 0 connected to them. However, unlike wires in a circuit, a signal’s value will be “remembered” until it is explicitly updated. This concept of memory was introduced in Chapter 1 and the hardware mechanism will be much more clearly explained in the next chapter. For now it is sufficient to say that a signal that has been assigned a value will remain at that value unless a different value is assigned to it. A signal that is indeterminate will not default to 1 or 0. The operator used to assign a value to a signal is “<=”.

Variables in VHDL are similar to variables in any computer language. They are a named place to “store” a value until the next point in time (event) when it needs to change. Variables are always declared and modified within a “PROCESS.” The operator used to assign a value to a variable is “:=”.

Constants are simply names that are permanently assigned to represent a value. This practice can make the code much easier to understand.

Each data object must be assigned a data “type.” The data types included in VHDL are BIT, BIT\_VECTOR, and INTEGER. A BIT can have values of 0 or 1. A BIT\_VECTOR is a group of bits, each of which can have a value of 0 or 1. An INTEGER can have any signed integer value (i.e., -3, -2, -1, 0, 1, 2, 3 . . .).

VHDL is very particular regarding the definitions of each type of data object. The type “bit\_vector” describes an array of individual bits. This is interpreted differently than an eight-bit binary number (called a scalar quantity), which has the type **integer**. Unfortunately, VHDL does not allow us to assign an integer value to a BIT\_VECTOR signal directly because they are not the same “type.” Data can be represented by any of the types shown in Table 4-9, but data assignments and other operations must be done between objects of the same type. For example, the compiler will not allow you to take a number from a keypad declared as an integer and connect it to four LEDs that are declared as BIT\_VECTOR outputs. Notice in Table 4-9, under Possible Values, that individual BIT and STD\_LOGIC data objects (e.g., signals, variables, inputs, and outputs) are designated by single quotes, whereas values assigned to BIT\_VECTOR and STD\_LOGIC\_VECTOR types are strings of valid bit values enclosed in double quotes.

**TABLE 4-9** Common VHDL data types.

Data Type	Sample Declaration	Possible Values	Use
BIT	y :OUT BIT;	'0' '1'	y <= '0';
STD_LOGIC	driver :STD_LOGIC	'0' '1' 'z' 'x' '-'	driver <= 'z';
BIT_VECTOR	bcd_data :BIT_VECTOR (3 DOWNT0 0);	"0101" "1001" "0000"	digit <= bcd_data;
STD_LOGIC_VECTOR	dbus :STD_LOGIC_VECTOR (3 DOWNT0 0);	"0Z1X"	IF rd = '0' THEN dbus <= "zzzz";
INTEGER	z:INTEGER RANGE -32 TO 31;	-32.. -2, -1,0,1,2... 31	IF z > 5 THEN...

VHDL also offers some standardized data types that are necessary when using logic functions that are contained in the **libraries**. As you might have guessed, libraries are simply collections of little pieces of VHDL code that can be used in your hardware descriptions without reinventing the wheel. These libraries offer convenient functions, called **macrofunctions** or **max-plus2** functions, like many of the standard TTL devices that are described throughout this text. Rather than writing a new description of a familiar TTL device, we can simply pull its macrofunction out of the library and use it in our system. Of course, you need to get signals into and out of these macrofunctions, and the types of the signals in your code must match the types in the functions (which someone else wrote). This means that everyone must use the same standard data types.

When VHDL was standardized through the IEEE, many data types were created at the same time. The two that we will use in this text are **STD\_LOGIC**, which is an expansion of the BIT type, and **STD\_LOGIC\_VECTOR**, which is an expansion of the BIT\_VECTOR. As you recall, BIT type can have values of only '0' and '1'. The standard logic types are defined in the IEEE

library and have a broader range of possible values than their built-in counterparts. The possible values for a `STD_LOGIC` type or for any element in a `STD_LOGIC_VECTOR` are given in Table 4-10. The names of these categories will make much more sense after we study the characteristics of logic circuits in Chapter 8. For now, we will show examples using values of only '1' and '0'.

**TABLE 4-10** `STD_LOGIC` values.

'1'	Logic 1 (just like BIT type)
'0'	Logic 0 (just like BIT type)
'z'	High impedance*
'-'	don't care (just like you used in your K maps)
'U'	Uninitialized
'X'	Unknown
'W'	Weak unknown
'L'	Weak '0'
'H'	Weak '1'

\*We will study tristate logic in Chapter 8.

### OUTCOME ASSESSMENT QUESTIONS

1. How would you declare a six-bit input array named `push_buttons` in (a) AHDL or (b) VHDL?
2. What statement would you use to take the MSB from the array in question 1 and put it on a single-bit output port named `z`? Use (a) AHDL or (b) VHDL.
3. In VHDL, what is the IEEE standard type that is equivalent to the BIT type?
4. In VHDL, what is the IEEE standard type that is equivalent to the BIT\_VECTOR type?
5. List the number systems that can be used with AHDL and VHDL.

## 4-16 TRUTH TABLES USING HDL

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Use decision control constructs common to most programming languages to describe hardware behavior.
- Use correct syntax.
- Apply a process in VHDL.

We have learned that a truth table is another way of expressing the operation of a circuit block. It relates the output of the circuit to every possible combination of its inputs. As we saw in Section 4-4, a truth table is the starting point for a designer to define how the circuit should operate. Then a Boolean expression is derived from the truth table and simplified using K maps or Boolean algebra. Finally the circuit is implemented from the final Boolean equation. Wouldn't it be great if we could go from the truth table directly to the final circuit without all those steps? We can do exactly that by entering the truth table using HDL.

## TRUTH TABLES USING AHDL

The code in Figure 4-53 uses AHDL to implement a circuit and uses a truth table to describe its operation. The truth table for this design was presented in Example 4-7. The key point of this example is the use of the TABLE keyword in AHDL. It allows the designer to specify the operation of the circuit just like you would fill out a truth table. On the first line after TABLE, the input variables ( $a,b,c$ ) are listed exactly like you would create a column heading on a truth table. By including the three binary variables in parentheses, we tell the compiler that we want to use these three bits as a group and to refer to them as a three-bit binary number or bit pattern. The specific values for this bit pattern are listed below the group and are referred to as binary literals. The special operator ( $=>$ ) is used in truth tables to separate the inputs from the output ( $y$ ).

**FIGURE 4-53** AHDL design file for Figure 4-7.

```

SUBDESIGN Figure 4-53
(
    a,b,c :INPUT;           --a is most significant
    y     :OUTPUT;         --define block output
)
BEGIN
    TABLE
        (a,b,c)           => y;      --column headings
        (0,0,0)           => 0;
        (0,0,1)           => 0;
        (0,1,0)           => 0;
        (0,1,1)           => 1;
        (1,0,0)           => 0;
        (1,0,1)           => 1;
        (1,1,0)           => 1;
        (1,1,1)           => 1;
    END TABLE;
END;

```

The TABLE in Figure 4-53 is intended to show the relationship between the HDL code and a truth table. A more common way of representing the input data heading is to use a variable bit array to represent the value on  $a, b, c$ . This method involves a declaration of the bit array on the line before BEGIN, such as:

```
VARIABLE in_bits[2..0] :NODE;
```

Just before the TABLE keyword, the input bits can be assigned to the array, *inbits[]*:

```
in_bits[] = (a,b,c);
```

Grouping three independent bits in order like this is referred to as **concatenating**, and it is often done to connect individual bits to a bit array. The table heading on the input bit sets can be represented by *in\_bits[]*, in this case. Note that as we list the possible combinations of the inputs, we have

several options. We can make up a group of 1s and 0s in parentheses, as shown in Figure 4-53, or we can represent the same bit pattern using the equivalent binary, hex, or decimal number. It is up to the designer to decide which format is most appropriate depending on what the input variables represent.

## TRUTH TABLES USING VHDL: SELECTED SIGNAL ASSIGNMENT

The code in Figure 4-54 uses VHDL to implement a circuit using a **selected signal assignment** to describe its operation. It allows the designer to specify the operation of the circuit, just like you would fill out a truth table. The truth table for this design was presented in Example 4-7. The primary point of this example is the use of the WITH signal\_name SELECT statement in VHDL. A secondary point presented here shows how to put the data into a format that can be used conveniently with the selected signal assignment. Notice that the inputs are defined in the ENTITY declaration as three independent bits *a*, *b*, and *c*. Nothing in this declaration makes one of these more significant than another. The order in which they are listed does not matter. We want to compare the current value of these bits with each of the possible combinations that could be present. If we drew out a truth table, we would decide which bit to place on the left (MSB) and which to place on the right (LSB). This is accomplished in VHDL by **concatenating** (connecting in order) the bit variables to form a bit vector. The concatenation operator is “&”. A signal is declared as a BIT\_VECTOR to receive the ordered set of input bits and is used to compare the input’s value with the string literals contained in quotes. The output (*y*) is assigned (<=) a bit value (‘0’ or ‘1’) WHEN *in\_bits* contains the value listed in double quotes.

VHDL is very strict in the way it allows us to assign and compare objects such as signals, variables, constants, and literals. The output *y* is a BIT, and so it must be assigned a value of ‘0’ or ‘1’. The SIGNAL *in\_bits* is a three-bit

```

ENTITY Figure 4-54 IS
PORT (
    a,b,c :IN BIT;           --a is most significant
    y      :OUT BIT);
END Figure 4-54;

ARCHITECTURE truth OF Figure 4-54 IS
    SIGNAL in_bits :BIT_VECTOR(2 DOWNT0 0);
BEGIN
    in_bits <= a & b & c;    --concatenate input bits into bit_vector
    WITH in_bits SELECT
        y  <=
            '0' WHEN "000",   --Truth Table
            '0' WHEN "001",
            '0' WHEN "010",
            '1' WHEN "011",
            '0' WHEN "100",
            '1' WHEN "101",
            '1' WHEN "110",
            '1' WHEN "111";

END truth;

```

**FIGURE 4-54** VHDL design file for Figure 4-7.

BIT\_VECTOR, so it must be compared with a three-bit string literal value. VHDL will not allow *in\_bits* (a BIT\_VECTOR) to be compared with a hex number like X “5” or a decimal number like 3. These scalar quantities would be valid for assignment or comparison with integers.

**EXAMPLE 4-33**

Declare three signals in VHDL that are single bits named *too\_hot*, *too\_cold*, and *just\_right*. Combine (concatenate) these three bits into a three-bit signal called *temp\_status*, with hot on the left and cold on the right.

**Solution**

1. Declare signals first in Architecture.

```
SIGNAL too_hot, too_cold, just_right :BIT;
SIGNAL temp_status :BIT_VECTOR (2 DOWNT0 0);
```

2. Write concurrent assignment statements between BEGIN and END.

```
temp_status <= too_hot & just_right & too_cold;
```

**OUTCOME ASSESSMENT QUESTIONS**

1. How would you concatenate three bits *x*, *y*, and *z* into a three-bit array named *omega*? Use AHDL or VHDL.
2. How are truth tables implemented in AHDL?
3. How are truth tables implemented in VHDL?

**4-17 DECISION CONTROL STRUCTURES IN HDL****OUTCOMES**

*Upon completion of this section, you will be able to:*

- Choose the best control structure to describe a circuit based on its requirements.
- Differentiate between concurrent operations and sequential operations.

In this section, we will examine methods that allow us to tell the digital system how to make “logical” decisions in much the same way that we make decisions every day. In Chapter 3, we learned that concurrent assignment statements are evaluated such that the order in which they are written has no effect on the circuit being described. When using **decision control structures**, the order in which we ask the questions does matter. To summarize this concept in the terms used in HDL documentation, statements that can be written in any sequence are called **concurrent**, and statements that are evaluated in the sequence in which they are written are called **sequential**. The sequence of sequential statements affects the circuit’s operation.

The examples we have considered so far involve several individual bits. Many digital systems require inputs that represent a numeric value. Refer back to Example 4-8, in which the purpose of the logic circuit is to monitor the battery voltage measured by an A/D converter. The digital value is represented by a four-bit number coming from the A/D into the logic circuit. These inputs are not independent binary variables but rather four binary

digits of a number representing battery voltage. We need to give the data the correct type that will allow us to use it as a number.

## IF/ELSE

Truth tables are great for listing all the possible combinations of independent variables, but there are better ways to handle numeric data. As an example, when a person leaves for school or work in the morning, she must make a logical decision about wearing a coat. Let's assume she decides this issue based only on the current temperature. How many of us would reason as follows?

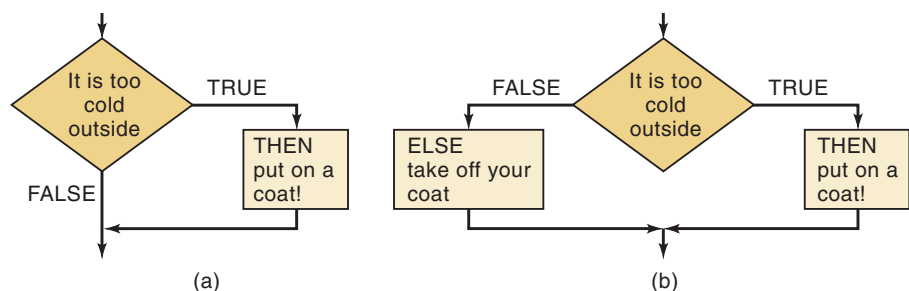
- I will wear a coat if the temperature is 0.
- I will wear a coat if the temperature is 1.
- I will wear a coat if the temperature is 2....
- I will wear a coat if the temperature is 55.
- I will *not* wear a coat if the temperature is 56.
- I will *not* wear a coat if the temperature is 57.
- I will *not* wear a coat if the temperature is 58....
- I will *not* wear a coat if the temperature is 99.

This method is similar to the truth table approach of describing the decision. For every possible input, she decides what the output should be. Of course, what she would really do is decide as follows:

- I will wear a coat if the temperature is less than 56 degrees.
- Otherwise, I will *not* wear a coat.

An HDL gives us the power to describe logic circuits using this type of reasoning. First, we must describe the inputs as a *number within a given range*, and then we can write statements that decide what to do to the outputs based on the *value* of the incoming number. In most computer programming languages, as well as HDLs, these types of decisions are made using an IF/THEN/ELSE control structure. Whenever the decision is between doing something and doing nothing, an **IF/THEN** construct is used. The keyword **IF** is followed by a statement that is true or false. **IF** it is true, **THEN** do whatever is specified. In the event that the statement is false, no action is taken. Figure 4-55(a) shows graphically how this decision works. The diamond shape represents the decision being made by evaluating the statement contained within the diamond. Every decision has two possible outcomes: true or false. In this example, if the statement is false, no action is taken.

**FIGURE 4-55** Logical flow of (a) IF/THEN and (b) IF/THEN/ELSE constructs.



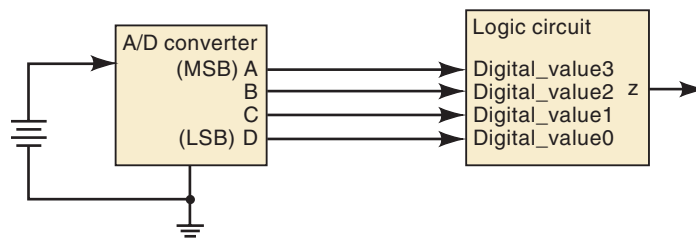


In some cases it is not enough only to decide to act or not to act, but rather we must choose between two different actions. For example, in our analogy about the decision to wear a coat or not using IF/THEN, it is assumed that the person is not initially wearing a coat when making the decision (because it was the beginning of the day). If she already has a coat on when making this decision (say at mid-day), there is no provision for her to take the coat off if it is too warm.

When decisions demand two possible actions, the IF/THEN/ELSE control structure is used, as shown in Figure 4-55(b). Here again, the statement is evaluated as true or false. The difference is that when the statement is false, a different action is performed. One of the two actions must occur with this construct. We can state it verbally as, “IF the statement is true, THEN do this. ELSE do that.” In our coat analogy, this control structure would work, regardless of whether the person’s coat was on or off initially.

Example 4-8 gave a simple example of a logic circuit that has as its input a numeric value representing battery voltage from an A/D converter. The inputs *A*, *B*, *C*, *D* are actually binary digits in a four-bit number, with *A* being the MSB and *D* being the LSB. Figure 4-56 shows the same circuit with the

**FIGURE 4-56** Logic circuit similar to Example 4-8.



inputs labeled as a four-bit number called *digital\_value*. The relationship between bits is as follows:

<i>A</i>	<i>digital_value</i> [3]	digital value bit 3 (MSB)
<i>B</i>	<i>digital_value</i> [2]	digital value bit 2
<i>C</i>	<i>digital_value</i> [1]	digital value bit 1
<i>D</i>	<i>digital_value</i> [0]	digital value bit 0 (LSB)

The input can be treated as a decimal number between 0 and 15 if we specify the correct type of the input variable.

### IF/THEN/ELSE USING AHDL

In AHDL, the inputs can be specified as a binary number made up of multiple bits by assigning a variable name followed by a list of the bit positions, as shown in Figure 4-57. The name is *digital\_value*, and the bit positions range from 3 down to 0. Notice how simple the code becomes using this method along with an IF/ELSE construct. The IF is followed by a statement that refers to the value of the entire four-bit input variable and compares it with the number 6. Of course, 6 is a decimal form of a scalar quantity and *digital\_value*[ ] actually represents a binary number. The compiler can interpret numbers in any system, so it creates a logic circuit that compares the binary value of *digital\_value* with the binary number for 6 and decides if this statement is true or false. If it is true, THEN the next statement (*z* = VCC) is used to assign *z* a value. Notice that in AHDL, we must use VCC for a logic 1

**FIGURE 4-57** AHDL version.

```

SUBDESIGN Figure 4-57
(
    digital_value[3..0] :INPUT;    -- define inputs to block
    z                   :OUTPUT;  -- define block output
)
BEGIN
    IF digital_value[] > 6 THEN
        z = VCC;                  -- output a 1
    ELSE z = GND;                 -- output a 0
    END IF;
END;

```

and GND for a logic 0 when assigning a logic level to a single bit. When *digital\_value* is 6 or less, it follows the statement after ELSE ( $z = \text{GND}$ ). The END IF; terminates the control structure.

## IF/THEN/ELSE USING VHDL

In VHDL, the critical issue is the declaration of the type of inputs. Refer to Figure 4-58. The input is treated as a single variable called *digital\_value*. Because its type is declared as INTEGER, the compiler knows to treat it as a number. By specifying a range of 0 to 15, the compiler knows it is a four-bit number. Notice that RANGE does not specify the index number of a bit vector but rather the limits of the numeric value of the integer. Integers are treated differently than bit arrays (BIT\_VECTOR) in VHDL. An integer can be compared with other numbers using inequality operators. A BIT\_VECTOR cannot be used with inequality operators.

Perhaps the most difficult aspect of VHDL is understanding the control structure called a “PROCESS.” Recall that VHDL is a language that is intended to describe the behavior of a hardware circuit with the intent that it will be evaluated by a computer for the purpose of simulation or synthesis. The PROCESS in VHDL is a segment of code that has characteristics of a computer program but whose purpose is to describe sequential events in a

**FIGURE 4-58** VHDL version.

```

ENTITY Figure 4-58 IS
    PORT( digital_value :IN INTEGER RANGE 0 TO 15; -- 4-bit input
          z             :OUT BIT);
END Figure 4-58;

ARCHITECTURE decision OF Figure 4-58 IS

BEGIN
    PROCESS (digital_value)
    BEGIN
        IF (digital_value > 6) THEN
            z <= '1';
        ELSE
            z <= '0';
        END IF;
    END PROCESS;
END decision;

```

digital circuit. It is very important to understand how the actions described in a PROCESS actually occur in the circuit.

A PROCESS is not constantly active, describing what is happening all the time. Instead, a PROCESS is a segment of code that describes the reaction of a circuit to a change in one (or more) of its inputs. These inputs that invoke the changes described in the PROCESS are identified in the *sensitivity list*. This list follows the keyword PROCESS and is contained in parentheses. The interpretation of the statements within the PROCESS (which result in changes within the circuit) is sequential, not concurrent. In other words, the order of the statements has an effect on the resulting circuit behavior.

Variables are declared and modified within a “PROCESS.” Their value is updated immediately. In other words, when a PROCESS is activated by a change in its sensitivity list and the logic within the PROCESS leads to the assignment of a new value to a variable, the effect is immediate. Therefore, these variables update in the order of the statements within the PROCESS. On the other hand, SIGNALS that are altered within a PROCESS are arbitrated at the end of the process.

#### EXAMPLE 4-34

Compare the VHDL code segments ex1 and ex2 below. Both examples use a signal, *sig*, and a variable, *var*, an input *a* with, outputs *ledseg* and *ledvar*. All are of TYPE :BIT.

```

SIGNAL sig          :BIT ;
ex1: PROCESS  (a)
    VARIABLE var    :BIT:= '0' ; -- results do not depend on this
                                init value

    BEGIN
        var := a;                -- var takes the value of input
                                a immediately
        sig <= var;              -- lone assignment to sig re-
                                sults in a wire from a
        var := not sig;          -- var is updated w/ complement
                                of input a
        ledvar <= var;           -- result: ledvar is driven by
                                NOT input a

    END PROCESS;
    ledsig <= sig;               -- result: ledsig is driven by
                                input a

ex2: PROCESS  (a)
    VARIABLE var    :BIT:= '0' ; -- results do not depend on
                                this init value

    BEGIN
        sig <= a;                -- the first signal assignment is
                                evaluated at end of process
        var := sig;              -- var is assigned by sig but
                                sig is not finalized yet
        sig <= not var;          -- this signal assignment is
                                wired, not the first line
                                (sig<=a)
        ledvar <= var;           -- result: ledvar is driven by
                                nothing!

    END PROCESS;
    ledsig <= sig;               -- result: ledsig is driven by
                                NOT ledvar

```

**Result:**

The order of assignments makes a big difference in the resulting hardware. The first example (ex1) results in `ledsig` connected to input `a` and `ledvar` connected to `NOT a`. The second example (ex2) results in a senseless circuit. `Ledsig` is driven by `NOT ledvar` but `ledvar` is not connected to input `A`. It is driven by nothing. Both `ledsig` and `ledvar` default to a logic 1 in hardware. In a timing simulation the two outputs of ex2 are indeterminate. This is because signal assignments are resolved at the end of the process where the last assignment determines the connection.

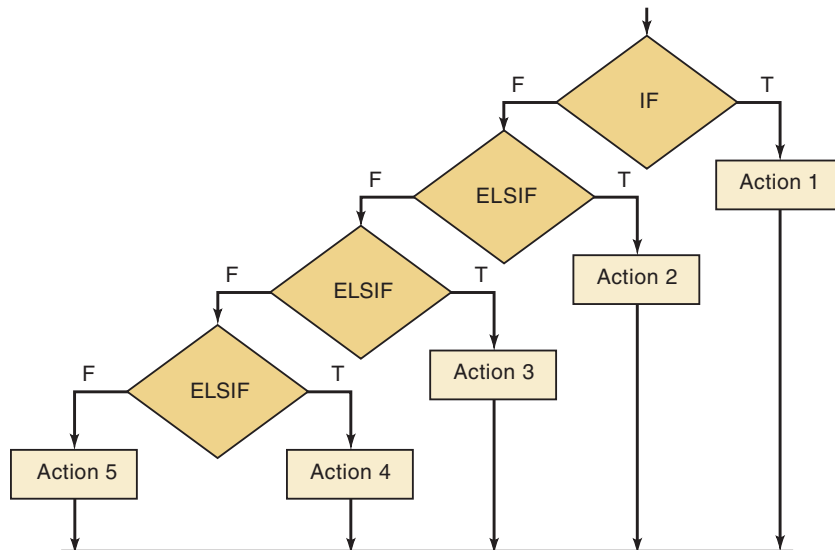
To use the IF/THEN/ELSE control structure, VHDL requires that the code be put inside a PROCESS. For example in Figure 4-58, whenever *digital\_value* changes, it causes the process code to be reevaluated. Even though we know *digital\_value* is really a four-bit binary number, the compiler will evaluate it as a number between the equivalent decimal values of 0 and 15. IF the statement in parentheses is true, THEN the next statement is applied (*z* is assigned a value of logic 1). If this statement is not true, the logic follows the ELSE clause and assigns a value of 0 to *z*. The END IF; terminates the control structure, and the END PROCESS; terminates the evaluation of the sequential statements.

**ELSIF**

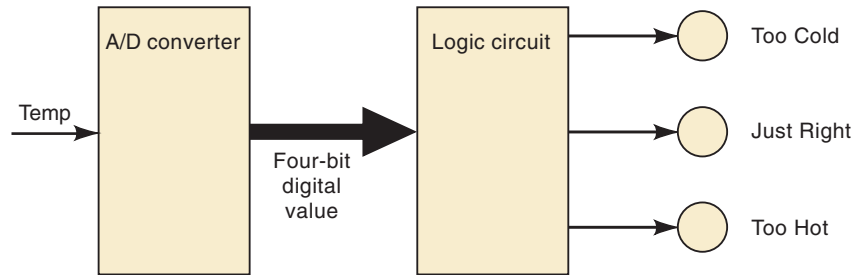
We often need to choose among many possible actions, depending on the situation. The IF construct chooses whether to perform a set of actions or not. The IF/ELSE construct selects one out of two possible actions. By combining IF and ELSE decisions, we can create a control structure referred to as **ELSIF**, which chooses one of many possible outcomes. The decision structure is shown graphically in Figure 4-59.

Notice that as each condition is evaluated, it either performs an action if true or goes on to evaluate the next condition. Each action is associated with one condition, and there is no chance to select more than one action. Note also that the conditions used to decide the appropriate action can be any expression that evaluates as true or false. This fact allows the designer to use the inequality operators to choose an action based on a range of input values. As an example of this application, let's consider the temperature-measuring system that uses an A/D converter, as described in Figure 4-60. Suppose that we want to indicate when the temperature is in a certain range, which we will refer to as Too Cold, Just Right, and Too Hot.

**FIGURE 4-59** Flowchart for multiple decisions using IF/ELSIF.



**FIGURE 4-60**  
Temperature range  
indicator circuit.



The relationship between the digital values for temperature and the categories is

<i>Digital Values</i>	<i>Category</i>
0000–1000	Too Cold
1001–1010	Just Right
1011–1111	Too Hot

We can express the decision-making process for this logic circuit as follows:

IF the digital value is less than or equal to 8, THEN light only the Too Cold indicator.

ELSE IF the digital value is greater than 8 AND less than 11, THEN light only the Just Right indicator.

ELSE light only the Too Hot indicator.

## ELSIF USING AHDL

The AHDL code in Figure 4-61 defines the inputs as a four-bit binary number. The outputs are three individual bits that drive the three range indicators. This example uses an intermediate variable (*status*) that allows us to assign a bit pattern representing the three conditions of *too\_cold*, *just\_right*, and *too\_hot*. The sequential section of the code uses the IF, ELSIF, ELSE to identify the range in which the temperature lies and assigns the correct bit pattern to *status*. In the last statement, the bits of *status* are connected to the actual output port bits. These bits have been ordered in a group that relates to the bit patterns assigned to *status[]*. This could also have been written as three concurrent statements: `too_cold = status[2]; just_right = status[1]; too_hot = status[0];`.

```

SUBDESIGN Figure 4-61
(
    digital_value[3..0]          :INPUT; --define inputs to block
    too_cold, just_right, too_hot :OUTPUT;--define outputs
)
VARIABLE
status[2..0] :NODE;--holds state of too_cold, just_right, too_hot
BEGIN
    IF      digital_value[] <= 8 THEN status[] = b"100";
    ELSIF  digital_value[] > 8 AND digital_value[] < 11 THEN
        status[] = b"010";
    ELSE  status[] = b"001";
    END IF;
    (too_cold, just_right, too_hot) = status[]; -- update output bits
END;

```

**FIGURE 4-61** Temperature range example in AHDL using ELSIF.

## ELSIF USING VHDL

The VHDL code in Figure 4-62 defines the inputs as a four-bit integer. The outputs are three individual bits that drive the three range indicators. This example uses an intermediate signal (*status*) that allows us to assign a bit pattern representing all three conditions of *too\_cold*, *just\_right*, and *too\_hot*. The process section of the code uses the IF, ELSIF, and ELSE to identify the range in which the temperature lies and assigns the correct bit pattern to *status*. In the last three statements, each bit of *status* is connected to the correct output port bit.

```

ENTITY Figure 4-62 IS
  PORT(digital_value:IN INTEGER RANGE 0 TO 15;      -- declare 4-bit input
        too_cold, just_right, too_hot :OUT BIT);
END Figure 4-62;

ARCHITECTURE howhot OF Figure 4-62 IS
  SIGNAL status      :BIT_VECTOR (2 downto 0);
BEGIN
  PROCESS (digital_value)
  BEGIN
    IF (digital_value <= 8) THEN status <= "100";
    ELSIF (digital_value > 8 AND digital_value < 11) THEN
      status <= "010";
    ELSE status <= "001";
    END IF;
  END PROCESS;
  too_cold    <= status(2);      -- assign status bits to output
  just_right  <= status(1);
  too_hot     <= status(0);
END howhot;

```

**FIGURE 4-62** Temperature range example in VHDL using ELSIF.

## CASE

One more important control structure is useful for choosing actions based on current conditions. It is called by various names, depending on the programming language, but it nearly always involves the word **CASE**. This construct determines the value of an expression or object and then goes through a list of possible values (cases) for the expression or object being evaluated. Each case has a list of actions that should take place. A CASE construct is different from an IF/ELSIF because a case correlates one unique value of an object with a set of actions. Recall that an IF/ELSIF correlates a set of actions with a true statement. There can be only one match for a CASE statement. An IF/ELSIF can have more than one statement that is true, but will THEN perform the action associated with the first true statement it evaluates.

Another important point in the HDL examples of Figures 4-63 through 4-67 is the need to combine several independent variables into a set of bits, called a bit vector. Recall that this action of linking several bits in a particular order is called *concatenation*. It allows us to consider the bit pattern as an ordered group.

## CASE USING AHDL

The AHDL example in Figure 4-63 demonstrates a case construct implementing the circuit of Figure 4-9 (see also Table 4-3). It uses individual bits as its inputs. In the first statement after BEGIN, these bits are concatenated and assigned to the intermediate variable called *status*. The CASE statement evaluates the variable *status* and finds the bit pattern (following the keyword WHEN) that matches the value of *status*. It then performs the action described following =>. In this example, it simply assigns logic 0 to the output for each of the three specified cases. All *other* cases result in a logic 1 on the output.

```

SUBDESIGN Figure 4-63
(
  p, q, r      :INPUT;      -- define inputs to block
  s            :OUTPUT;     -- define outputs
)
VARIABLE
  status[2..0] :NODE;
BEGIN
  status[] = (p, q, r); -- link input bits in order
  CASE status[] IS
    WHEN b"100"   => s = GND;
    WHEN b"101"   => s = GND;
    WHEN b"110"   => s = GND;
    WHEN OTHERS   => s = VCC;
  END CASE;
END;

```

**FIGURE 4-63** Figure 4-9 represented in AHDL.

## CASE USING VHDL

The VHDL example in Figure 4-64 demonstrates the case construct implementing the circuit of Figure 4-9 (see also Table 4-3). It uses individual bits as its inputs. In the first statement after BEGIN, these bits are concatenated and assigned to the intermediate variable called *status* using the & operator. Case statements must occur within a PROCESS in VHDL. The CASE statement evaluates the variable *status* and finds the bit pattern (following the keyword WHEN) that matches the value of *status*. It then performs the action described following =>. In this simple example, it merely assigns logic 0 to the output for each of the three specified cases. All *other* cases result in a logic 1 on the output.

### EXAMPLE 4-35

A coin detector in a vending machine accepts quarters, dimes, and nickels and activates the corresponding digital signal (*Q*, *D*, *N*) only when the correct coin is present. It is physically impossible for multiple coins to be present at the same time. A digital circuit must use the *Q*, *D*, and *N* signals as

**FIGURE 4-64** Figure 4-9 represented in VHDL.

```

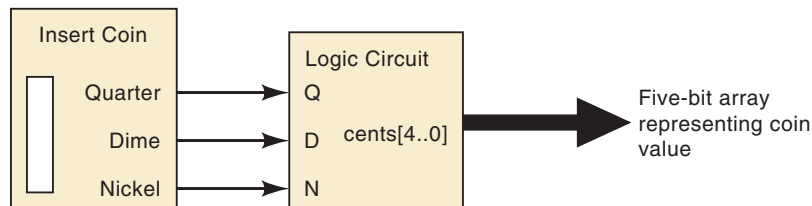
ENTITY Figure 4-64 IS
PORT( p, q, r      :IN bit;          --declare 3 bits input
      s            :OUT BIT);
END Figure 4-64;

ARCHITECTURE copy OF Figure 4-64 IS
SIGNAL status      :BIT_VECTOR (2 downto 0);
BEGIN
  status <= p & q & r;              --link bits in order
  PROCESS (status)
  BEGIN
    CASE status IS
      WHEN "100" => s <= '0';
      WHEN "101" => s <= '0';
      WHEN "110" => s <= '0';
      WHEN OTHERS => s <= '1';
    END CASE;
  END PROCESS;
END copy;

```

inputs and produce a binary number representing the value of the coin as shown in Figure 4-65. Write the AHDL and VHDL code.

**FIGURE 4-65** A coin detector circuit for a vending machine.



### Solution

This is an ideal application of the CASE construct to describe the correct operation. The outputs must be declared as five-bit numbers in order to represent up to 25 cents. Figure 4-66 shows the AHDL solution and Figure 4-67 shows the VHDL solution.

**FIGURE 4-66** An AHDL coin detector.

```

SUBDESIGN Figure 4-66
(
  q, d, n          :INPUT;          -- define quarter, dime, nickel
  cents[4..0]     :OUTPUT;         -- define binary value of coins
)
BEGIN
  CASE (q, d, n) IS                -- group coins in an ordered set
    WHEN b"001" => cents[] = 5;
    WHEN b"010" => cents[] = 10;
    WHEN b"100" => cents[] = 25;
    WHEN others => cents[] = 0;
  END CASE;
END;

```



```

ENTITY    Figure 4-67 IS
PORT( q, d, n:IN BIT;                          -- quarter, dime, nickel
      cents :OUT INTEGER RANGE 0 TO 25); -- binary value of coins
END Figure 4-67;
ARCHITECTURE detector of Figure 4-67 IS
    SIGNAL coins :BIT_VECTOR(2 DOWNT0 0);-- group the coin sensors
BEGIN
    coins <= (q & d & n);                        -- assign sensors to group
    PROCESS (coins)
    BEGIN
        CASE (coins) IS
            WHEN "001" => cents <= 5;
            WHEN "010" => cents <= 10;
            WHEN "100" => cents <= 25;
            WHEN others => cents <= 0;
        END CASE;
    END PROCESS;
END detector;

```

**FIGURE 4-67** A VHDL coin detector.

### OUTCOME ASSESSMENT QUESTIONS

1. Which control structure decides to do or not to do?
2. Which control structure decides to do this or to do that?
3. Which control structure(s) selects which one of several different actions to take?
4. *True or false:* IF/THEN/ELSE and CASE constructs must be used inside a process in VHDL.
5. *True or false:* IF/THEN/ELSE constructs can be used throughout the concurrent section of AHDL.

## SUMMARY

1. The two general forms for logic expressions are the sum-of-products form and the product-of-sums form.
2. One approach to the design of a combinatorial logic circuit is to (1) construct its truth table, (2) convert the truth table to a sum-of-products expression, (3) simplify the expression using Boolean algebra or K mapping, (4) implement the final expression.
3. The K map is a graphical method for representing a circuit's truth table and generating a simplified expression for the circuit output.
4. An exclusive-OR circuit has the expression  $x = A\bar{B} + \bar{A}B$ . Its output  $x$  will be HIGH only when inputs  $A$  and  $B$  are at opposite logic levels.
5. An exclusive-NOR circuit has the expression  $x = \bar{A}\bar{B} + AB$ . Its output  $x$  will be HIGH only when inputs  $A$  and  $B$  are at the same logic level.
6. Each of the basic gates (AND, OR, NAND, NOR) can be used to enable or disable the passage of an input signal to its output.

7. The main digital IC families are the TTL and CMOS families. Digital ICs are available in a wide range of complexities (gates per chip), from the basic to the high-complexity logic functions.
8. To perform basic troubleshooting requires—at a minimum—an understanding of circuit operation, a knowledge of the types of possible faults, a complete logic-circuit connection diagram, and a logic probe.
9. A programmable logic device (PLD) is an IC that contains a large number of logic gates whose interconnections can be programmed by the user to generate the desired logic relationship between inputs and outputs.
10. To program a PLD, you need a development system that consists of a computer, PLD development software, and a programmer fixture that does the actual programming of the PLD chip.
11. The Altera system allows convenient hierarchical design techniques using any form of hardware description.
12. The type of data objects must be specified so that the HDL compiler knows the range of numbers to be represented.
13. Truth tables can be entered directly into the source file using the features of HDL.
14. Logical control structures such as IF, ELSE, and CASE can be used to describe the operation of a logic circuit, making the code and the problem's solution much more straightforward.

## IMPORTANT TERMS

sum-of-products (SOP)	sensitivity list	integer
product-of-sums (POS)	indeterminate	objects
Karnaugh map (K map)	floating	libraries
looping	logic probe	macrofunction
don't-care condition	contention	maxplus2
exclusive-OR (XOR)	programmer	STD_LOGIC
exclusive-NOR (XNOR)	ZIF socket	STD_LOGIC_VECTOR
parity generator	JEDEC	concatenate
parity checker	ISP	selected signal assignment
enable/disable	JTAG	decision control structure
dual-in-line package (DIP)	hierarchical design	concurrent
SSI, MSI, LSI, VLSI, ULSI, GSI	top-down	sequential
transistor-transistor logic (TTL)	test vectors	IF/THEN
complementary metal- oxide semiconductor (CMOS)	literals	ELSE
	bit array	PROCESS
	bit vector	ELSIF
	index	CASE
	BIT_VECTOR	

## PROBLEMS

### SECTIONS 4-2 AND 4-3

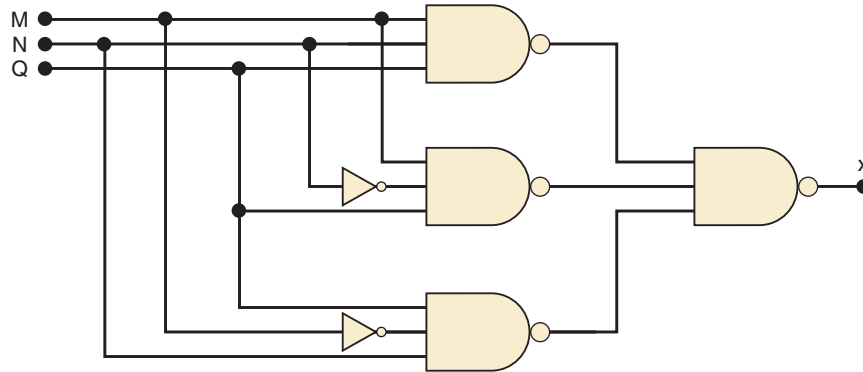
- B** 4-1.\* Simplify the following expressions using Boolean algebra.
- (a)  $x = ABC + \bar{A}C$
  - (b)  $y = (Q + R)(\bar{Q} + \bar{R})$

\*Answers to problems marked with an asterisk can be found in the back of the text.

- (c)  $w = ABC + A\bar{B}C + \bar{A}$
- (d)  $q = \overline{RST}(R + S + T)$
- (e)  $x = \bar{A}\bar{B}\bar{C} + \bar{A}BC + ABC + A\bar{B}\bar{C} + A\bar{B}C$
- (f)  $z = (B + \bar{C})(\bar{B} + C) + \bar{A} + B + \bar{C}$
- (g)  $y = (\bar{C} + \bar{D}) + \bar{A}C\bar{D} + \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C\bar{D} + AC\bar{D}$
- (h)  $x = AB(\bar{C}D) + \bar{A}BD + \bar{B}\bar{C}\bar{D}$

**B** 4-2. Simplify the circuit of Figure 4-68 using Boolean algebra.

**FIGURE 4-68** Problems 4-2 and 4-3.



**B** 4-3\* Change each gate in Problem 4-2 to a NOR gate, and simplify the circuit using Boolean algebra.

**SECTION 4-4**

**B, D** 4-4\* Design the logic circuit corresponding to the truth table shown in Table 4-11.

**TABLE 4-11**

A	B	C	x
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

**B, D** 4-5. Design a logic circuit whose output is HIGH *only* when a majority of inputs A, B, and C are LOW.

**D** 4-6. A manufacturing plant needs to have a horn sound to signal quitting time. The horn should be activated when either of the following conditions is met:

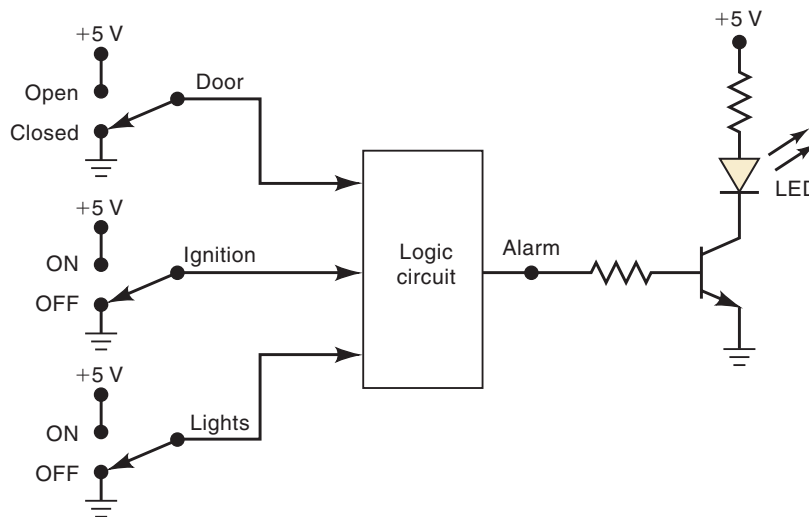
1. It's after 5 o'clock and all machines are shut down.
2. It's Friday, the production run for the day is complete, and all machines are shut down.

Design a logic circuit that will control the horn. (*Hint:* Use four logic input variables to represent the various conditions; for example,

input  $A$  will be HIGH only when the time of day is 5 o'clock or later.)

- D** 4-7\* A four-bit binary number is represented as  $A_3A_2A_1A_0$ , where  $A_3$ ,  $A_2$ ,  $A_1$ , and  $A_0$  represent the individual bits and  $A_0$  is equal to the LSB. Design a logic circuit that will produce a HIGH output whenever the binary number is greater than 0010 and less than 1000.
- D** 4-8. Figure 4-69 shows a diagram for an automobile alarm circuit used to detect certain undesirable conditions. The three switches are used

FIGURE 4-69 Problem 4-8.



to indicate the status of the door by the driver's seat, the ignition, and the headlights, respectively. Design the logic circuit with these three switches as inputs so that the alarm will be activated whenever either of the following conditions exists:

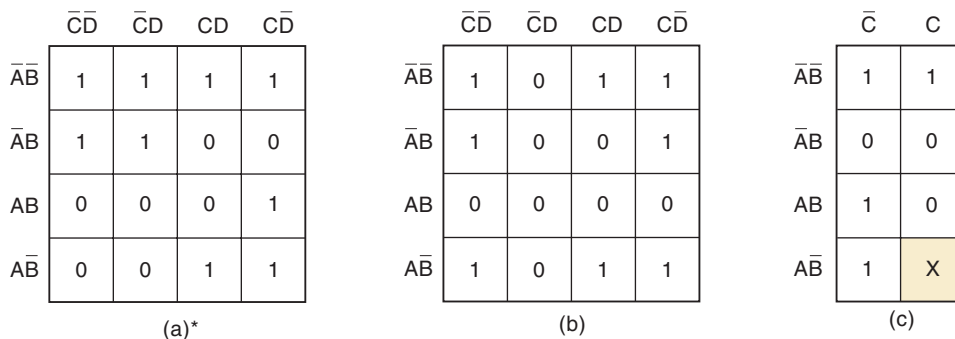
- The headlights are on while the ignition is off.
- The door is open while the ignition is on.

- 4-9\* Implement the circuit of Problem 4-4 using all NAND gates.
- 4-10. Implement the circuit of Problem 4-5 using all NAND gates.

**SECTION 4-5**

- B** 4-11. Determine the minimum expression for each K map in Figure 4-70. Pay particular attention to step 5 for the map in (a).

FIGURE 4-70 Problem 4-11.

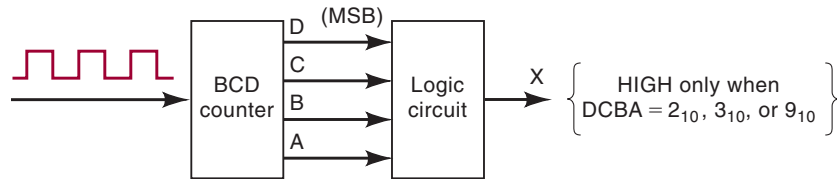


- B** 4-12. For the truth table below, create a  $2 \times 2$  K map, group terms, and simplify. Then look at the truth table again to see if the expression is true for every entry in the table.

A	B	y
0	0	1
0	1	1
1	0	0
1	1	0

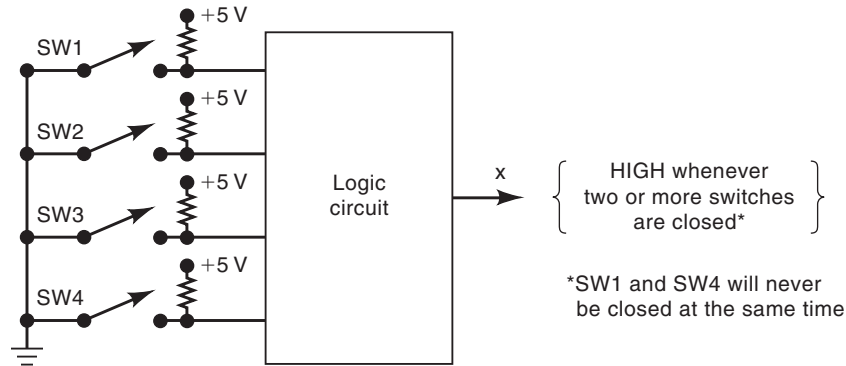
- B** 4-13. Starting with the truth table in Table 4-11, use a K map to find the simplest SOP equation.
- B** 4-14. Simplify the expression in (a)\* Problem 4-1(e) using a K map. (b) Problem 4-1(g) using a K map. (c)\* Problem 4-1(h) using a K map.
- B** 4-15\* Obtain the output expression for Problem 4-7 using a K map.
- C, D** 4-16. Figure 4-71 shows a *BCD counter* that produces a four-bit output representing the BCD code for the number of pulses that have been applied to the counter input. For example, after four pulses have occurred, the counter outputs are  $DCBA = 0100_2 = 4_{10}$ . The counter resets to 0000 on the tenth pulse and starts counting over again. In other words, the *DCBA* outputs will never represent a number greater than  $1001_2 = 9_{10}$ .
- (a)\* Design the logic circuit that produces a **HIGH** output whenever the count is 2, 3, or 9. Use K mapping and take advantage of the don't-care conditions.
- (b) Repeat for  $x = 1$  when  $DCBA = 3, 4, 5, 8$ .

FIGURE 4-71 Problem 4-16.



- D** 4-17\* Figure 4-72 shows four switches that are part of the control circuitry in a copy machine. The switches are at various points along the path of the copy paper as the paper passes through the machine. Each switch is normally open, and as the paper passes over a switch, the switch closes. It is impossible for switches SW1 and SW4 to be closed at the same time. Design the logic circuit to produce a **HIGH** output whenever *two or more* switches are closed at the same time. Use K mapping and take advantage of the don't-care conditions.

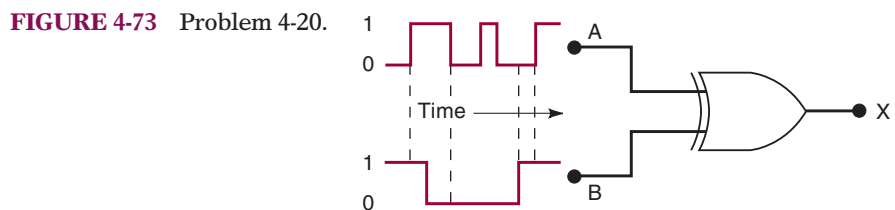
FIGURE 4-72 Problem 4-17.



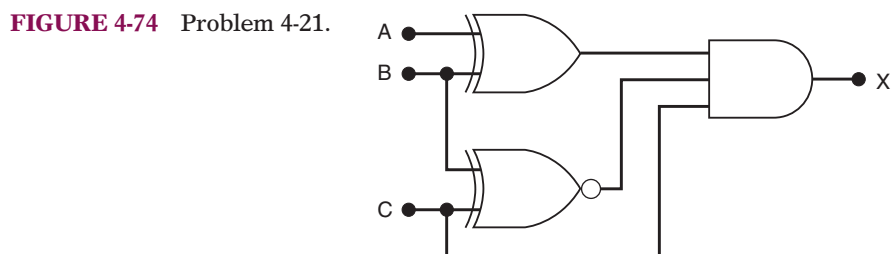
- D 4-18\* Example 4-3 demonstrated algebraic simplification. Step 3 resulted in the SOP equation  $z = \bar{A}BC + \bar{A}CD + \bar{A}BCD + ABC$ . Use a K map to prove that this equation can be simplified further than the answer shown in the example.
- C 4-19. Use Boolean algebra to arrive at the same result obtained by the K map method of Problem 4-18.

**SECTION 4-6**

- B 4-20. (a) Determine the output waveform for the circuit of Figure 4-73.  
 (b) Repeat with the  $B$  input held LOW.  
 (c) Repeat with  $B$  held HIGH.

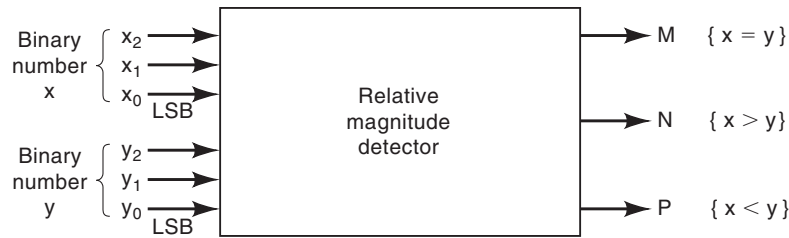


- B 4-21\* Determine the input conditions needed to produce  $x = 1$  in Figure 4-74.



- B 4-22. Design a circuit that produces a HIGH output only when all three inputs are the same level.
  - (a) Use a truth table and K map to produce the SOP solution.
  - (b) Use two-input XOR and other gates to find a solution. (*Hint:* Recall the transitive property from algebra... if  $a = b$  and  $b = c$  then  $a = c$ .)
- B 4-23\* A 7486 chip contains four XOR gates. Show how to make an XNOR gate using only a 7486 chip. *Hint:* See Example 4-16.
- B 4-24\* Modify the circuit of Figure 4-23 to compare two four-bit numbers and produce a HIGH output when the two numbers match exactly.
- B 4-25. Figure 4-75 represents a *relative-magnitude detector* that takes two three-bit binary numbers,  $x_2x_1x_0$  and  $y_2y_1y_0$ , and determines whether they are equal and, if not, which one is larger. There are three outputs, defined as follows:
  1.  $M = 1$  only if the two input numbers are equal.
  2.  $N = 1$  only if  $x_2x_1x_0$  is greater than  $y_2y_1y_0$ .
  3.  $P = 1$  only if  $y_2y_1y_0$  is greater than  $x_2x_1x_0$ .
 Design the logic circuitry for this detector. The circuit has *six* inputs and *three* outputs and is therefore much too complex to handle using the truth table approach. Refer to Example 4-17 as a hint about how you might start to solve this problem.

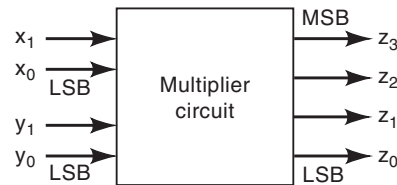
FIGURE 4-75 Problem 4-25.



### MORE DESIGN PROBLEMS

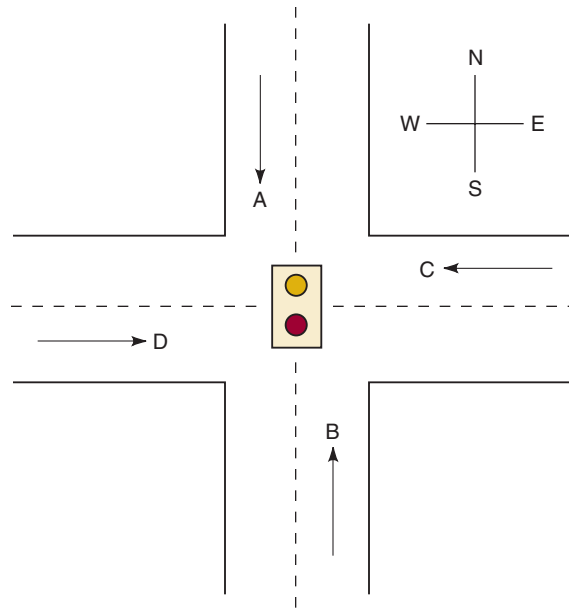
- C, D** 4-26\* Figure 4-76 represents a multiplier circuit that takes two-bit binary numbers,  $x_1x_0$  and  $y_1y_0$ , and produces an output binary number  $z_3z_2z_1z_0$  that is equal to the arithmetic product of the two input numbers. Design the logic circuit for the multiplier. (*Hint:* The logic circuit will have four inputs and four outputs.)

FIGURE 4-76 Problem 4-26.



- D** 4-27. A BCD code is being transmitted to a remote receiver. The bits are  $A_3, A_2, A_1$ , and  $A_0$ , with  $A_3$  as the MSB. The receiver circuitry includes a *BCD error detector* circuit that examines the received code to see if it is a legal BCD code (i.e.,  $\leq 1001$ ). Design this circuit to produce a HIGH for any error condition.
- D** 4-28\* Design a logic circuit whose output is HIGH whenever  $A$  and  $B$  are both HIGH as long as  $C$  and  $D$  are either both LOW or both HIGH. Try to do this without using a truth table. Then check your result by constructing a truth table from your circuit to see if it agrees with the problem statement.
- D** 4-29. Four large tanks at a chemical plant contain different liquids being heated. Liquid-level sensors are being used to detect whenever the level in tank  $A$  or tank  $B$  rises above a predetermined level. Temperature sensors in tanks  $C$  and  $D$  detect when the temperature in either of these tanks drops below a prescribed temperature limit. Assume that the liquid-level sensor outputs  $A$  and  $B$  are LOW when the level is satisfactory and HIGH when the level is too high. Also, the temperature-sensor outputs  $C$  and  $D$  are LOW when the temperature is satisfactory and HIGH when the temperature is too low. Design a logic circuit that will detect whenever the level in tank  $A$  or tank  $B$  is too high at the same time that the temperature in either tank  $C$  or tank  $D$  is too low.
- C, D** 4-30\* Figure 4-77 shows the intersection of a main highway with a secondary access road. Vehicle-detection sensors are placed along lanes  $C$  and  $D$  (main road) and lanes  $A$  and  $B$  (access road). These sensor

FIGURE 4-77 Problem 4-30.



outputs are LOW (0) when no vehicle is present and HIGH (1) when a vehicle is present. The intersection traffic light is to be controlled according to the following logic:

1. The east-west (E-W) traffic light will be green whenever *both* lanes C and D are occupied.
2. The E-W light will be green whenever *either* C or D is occupied but lanes A and B are not *both* occupied.
3. The north-south (N-S) light will be green whenever *both* lanes A and B are occupied but C and D are not *both* occupied.
4. The N-S light will also be green when *either* A or B is occupied while C and D are *both* vacant.
5. The E-W light will be green when *no* vehicles are present.

Using the sensor outputs A, B, C, and D as inputs, design a logic circuit to control the traffic light. There should be two outputs, N-S and E-W, that go HIGH when the corresponding light is to be *green*. Simplify the circuit as much as possible and show *all* steps.

#### SECTION 4-7

- D** 4-31. Redesign the parity generator and checker of Figure 4-25 to (a) operate using odd parity. (*Hint*: What is the relationship between an odd-parity bit and an even-parity bit for the same set of data bits?) (b) Operate on eight data bits.

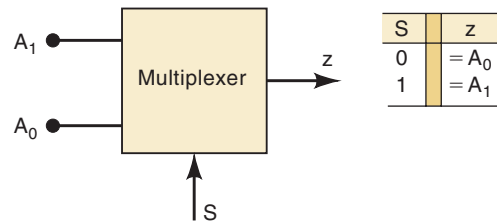
#### SECTION 4-8

- B** 4-32. (a) Under what conditions will an OR gate allow a logic signal to pass through to its output unchanged?  
 (b) Repeat (a) for an AND gate.  
 (c) Repeat for a NAND gate.  
 (d) Repeat for a NOR gate.
- B** 4-33\* (a) Can an INVERTER be used as an enable/disable circuit? Explain.  
 (b) Can an XOR gate be used as an enable/disable circuit? Explain.



- D** 4-34. Design a logic circuit that will allow input signal  $A$  to pass through to the output only when control input  $B$  is LOW while control input  $C$  is HIGH; otherwise, the output is LOW.
- D** 4-35\* Design a circuit that will *disable* the passage of an input signal only when control inputs  $B$ ,  $C$ , and  $D$  are all HIGH; the output is to be HIGH in the disabled condition.
- D** 4-36. Design a logic circuit that controls the passage of signal  $A$  according to the following requirements:
  1. Output  $X$  will equal  $A$  when control inputs  $B$  and  $C$  are the same.
  2.  $X$  will remain HIGH when  $B$  and  $C$  are different.
- D** 4-37. Design a logic circuit that has two signal inputs,  $A_1$  and  $A_0$ , and a control input  $S$  so that it functions according to the requirements given in Figure 4-78. (This type of circuit is called a *multiplexer* and will be covered in Chapter 9.)

**FIGURE 4-78** Problem 4-37.

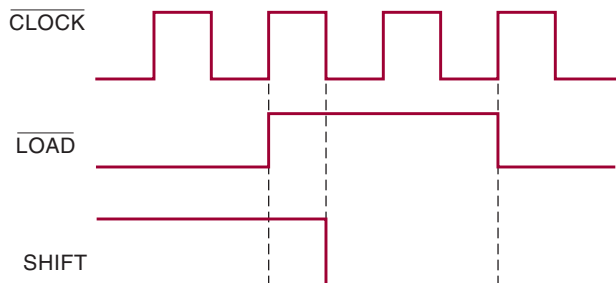


- D** 4-38\* Use K mapping to design a circuit to meet the requirements of Example 4-17. Compare this circuit with the solution in Figure 4-23. This points out that the K-map method cannot take advantage of the XOR and XNOR gate logic. The designer must be able to determine when these gates are applicable.

**SECTIONS 4-9 TO 4-13**

- T\*** 4-39. (a) A technician testing a logic circuit sees that the output of a particular INVERTER is stuck LOW while its input is pulsing. List as many possible reasons as you can for this faulty operation.
  - (b) Repeat part (a) for the case where the INVERTER output is stuck at an indeterminate logic level.
- T** 4-40\* The signals shown in Figure 4-79 are applied to the inputs of the circuit of Figure 4-32. Suppose that there is an internal open circuit at Z1-4.
  - (a) What will a logic probe indicate at Z1-4?
  - (b) What dc voltage reading would you expect at Z1-4? (Remember that the ICs are TTL.)

**FIGURE 4-79** Problem 4-40.



\*Recall that T indicates a troubleshooting exercise.

- (c) Sketch what you think the  $\overline{CLKOUT}$  and  $\overline{SHIFTOUT}$  signals will look like.
- (d) Instead of the open at Z1-4, suppose that pins 9 and 10 of Z2 are internally shorted. Sketch the probable signals at Z2-10,  $\overline{CLOCKOUT}$ , and  $\overline{SHIFTOUT}$ .

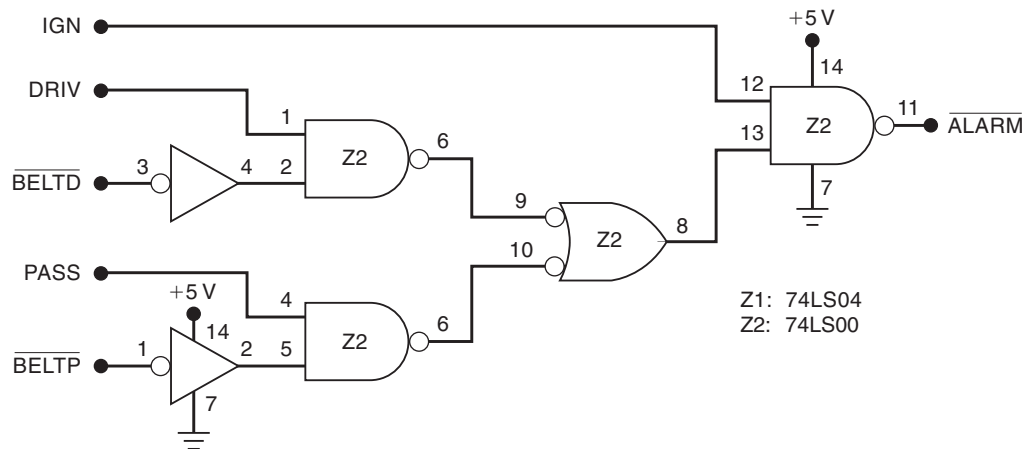
- T** 4-41. Assume that the ICs in Figure 4-32 are CMOS. Describe how the circuit operation would be affected by an open circuit in the conductor connecting Z2-2 and Z2-10.
- T** 4-42. In Example 4-24, we listed three possible faults for the situation of Figure 4-36. What procedure would you follow to determine which of the faults is the actual one?
- T** 4-43\* Refer to the circuit of Figure 4-38. Assume that the devices are CMOS. Also assume that the logic probe indication at Z2-3 is “indeterminate” rather than “pulsing.” List the possible faults, and write a procedure to follow to determine the actual fault.
- T** 4-44\* Refer to the logic circuit of Figure 4-41. Recall that output Y is supposed to be HIGH for either of the following conditions:
1.  $A = 1, B = 0$ , regardless of C
  2.  $A = 0, B = 1, C = 1$

When testing the circuit, the technician observes that Y goes HIGH only for the first condition but stays LOW for all other input conditions. Consider the following list of possible faults. For each one, write yes or no to indicate whether or not it could be the actual fault. Explain your reasoning for each no response.

- (a) An internal short to ground at Z2-13
  - (b) An open circuit in the connection to Z2-13
  - (c) An internal short to  $V_{CC}$  at Z2-11
  - (d) An open circuit in the  $V_{CC}$  connection to Z2
  - (e) An internal open circuit at Z2-9
  - (f) An open in the connection from Z2-11 to Z2-9
  - (g) A solder bridge between pins 6 and 7 of Z2
- T** 4-45. Develop a procedure for isolating the fault that is causing the malfunction described in Problem 4-44.
- T** 4-46\* Assume that the gates in Figure 4-41 are all CMOS. When the technician tests the circuit, he finds that it operates correctly except for the following conditions:
1.  $A = 1, B = 0, C = 0$
  2.  $A = 0, B = 1, C = 1$

For these conditions, the logic probe indicates indeterminate levels at Z2-6, Z2-11, and Z2-8. What do you think is the probable fault in the circuit? Explain your reasoning.

- T** 4-47. Figure 4-80 is a combinational logic circuit that operates an alarm in a car whenever the driver and/or passenger seats are occupied and the seatbelts are not fastened when the car is started. The active-HIGH signals *DRIV* and *PASS* indicate the presence of the driver and passenger, respectively, and are taken from pressure-actuated switches in the seats. The signal *IGN* is active-HIGH when the ignition switch is on. The signal  $\overline{BELTD}$  is active-LOW and indicates that the driver's seatbelt is *unfastened*; *BELTP* is the corresponding signal for the



**FIGURE 4-80** Problems 4-47, 4-48, and 4-49.

passenger seatbelt. The alarm will be activated (LOW) whenever the car is started and either of the front seats is occupied and its seatbelt is not fastened.

- Verify that the circuit will function as described.
  - Describe how this alarm system would operate if Z1-2 were internally shorted to ground.
  - Describe how it would operate if there were an open connection from Z2-6 to Z2-10.
- T** 4-48\* Suppose that the system of Figure 4-80 is functioning so that the alarm is activated as soon as the driver and/or passenger are seated and the car is started, regardless of the status of the seatbelts. What are the possible faults? What procedure would you follow to find the actual fault?
- T** 4-49\* Suppose that the alarm system of Figure 4-80 is operating so that the alarm goes on continuously as soon as the car is started, regardless of the state of the other inputs. List the possible faults and write a procedure to isolate the fault.

#### DRILL QUESTIONS ON PLDs (50 THROUGH 55)

4-50\* *True or false:*

- Top-down design begins with an overall description of the entire system and its specifications.
  - A JEDEC file can be used as the input file for a programmer.
  - If an input file compiles with no errors, it means the PLD circuit will work correctly.
  - A compiler can interpret code in spite of syntax errors.
  - Test vectors are used to simulate and test a device.
- H, B** 4-51. What are the % characters used for in the AHDL design file?
- H, B** 4-52. How are comments indicated in a VHDL design file?
- B** 4-53. What is a ZIF socket?
- B** 4-54\* Name three entry modes used to input a circuit description into PLD development software.
- B** 4-55. What do JEDEC and HDL stand for?

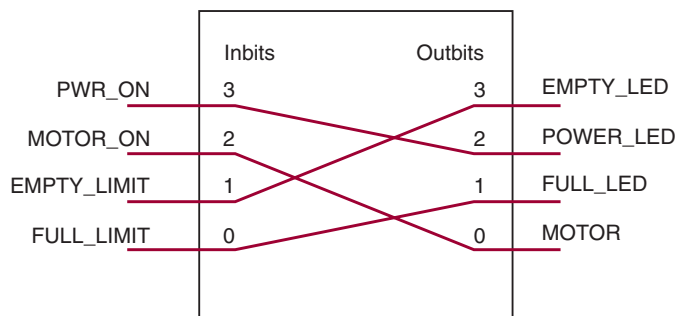
**SECTION 4-15**

- H, B** 4-56. Declare the following data objects in AHDL or VHDL.
  - (a)\*An array of eight output bits named *gadgets*.
  - (b) A single-output bit named *buzzer*.
  - (c) A 16-bit numeric input port named *altitude*.
  - (d) A single, intermediate bit within a hardware description file named *wire2*.
- H, B** 4-57. Express the following literal numbers in hex, binary, and decimal using the syntax of AHDL or VHDL.
  - (a)\* $152_{10}$
  - (b)  $1001010100_2$
  - (c)  $3C_{16}$
- H, B** 4-58\* The following similar I/O definition is given for AHDL and VHDL. Write four concurrent assignment statements that will connect the inputs to the outputs as shown in Figure 4-81.

**FIGURE 4-81** Problem 4-58.

```
SUBDESIGN hw
(
  inbits[3..0] :INPUT;
  outbits[3..0] :OUTPUT;
)

ENTITY hw IS
PORT (
  inbits :IN BIT_VECTOR (3 downto 0);
  outbits :OUT BIT_VECTOR (3 downto 0)
);
END hw;
```



**SECTION 4-16**

- H, D** 4-59. Modify the AHDL truth table of Figure 4-53 to implement  $AB + AC + \bar{A}B$ .
- H, D** 4-60\* Modify the AHDL design in Figure 4-57 so that  $z = 1$  only when the digital value is less than  $1010_2$ .
- H, D** 4-61. Modify the VHDL truth table of Figure 4-54 to implement  $AB + AC + \bar{A}B$ .

- H, D** 4-62\* Modify the VHDL design in Figure 4-58 so that  $z = 1$  only when the digital value is less than  $1010_2$ .
- H, B** 4-63. Modify the code of (a) Figure 4-57 or (b) Figure 4-58 such that the output  $z$  is LOW only when `digital_value` is between 6 and 11 (inclusive).
- H, D** 4-64. Modify (a) the AHDL design in Figure 4-63 to implement Table 4-1. (b) the VHDL design in Figure 4-64 to implement Table 4-1.
- H, D** 4-65\* Write the hardware description design file Boolean equation to implement Example 4-9.
- 4-66. Write the hardware description design file Boolean equation to implement a four-bit parity generator as shown in Figure 4-25(a).

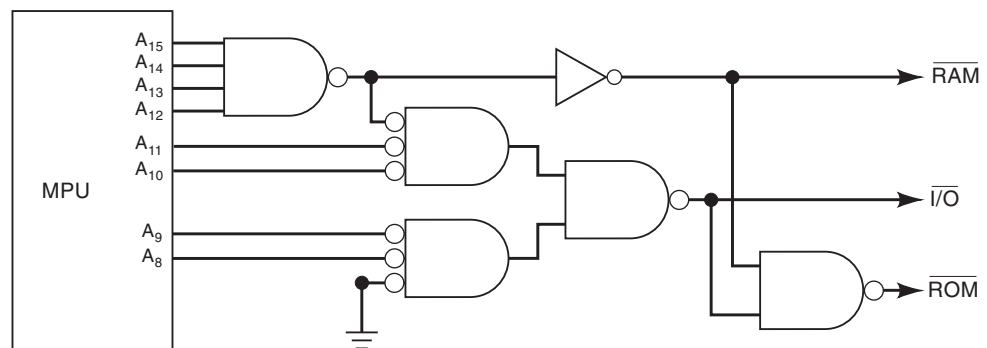
### DRILL QUESTION

- B** 4-67. Define each of the following terms.
- Karnaugh map
  - Sum-of-products form
  - Parity generator
  - Octet
  - Enable circuit
  - Don't-care condition
  - Floating input
  - Indeterminate voltage level
  - Contention
  - PLD
  - TTL
  - CMOS

### MICROCOMPUTER APPLICATIONS

- C** 4-68. In a microcomputer, the microprocessor unit (MPU) is always communicating with one of the following: (1) random-access memory (RAM), which stores programs and data that can be readily changed; (2) read-only memory (ROM), which stores programs and data that never change; and (3) external input/output (I/O) devices such as keyboards, video displays, printers, and disk drives. As it is executing a program, the MPU will generate an address code that selects which type of device (RAM, ROM, or I/O) it wants to communicate with. Figure 4-82 shows a typical arrangement where the MPU outputs an eight-bit address code  $A_{15}$  through  $A_8$ . Actually, the MPU outputs a 16-bit address

**FIGURE 4-82** Problems 4-68 and 4-69.



code, but the low-order bits  $A_7$  through  $A_0$  are not used in the device selection process. The address code is applied to a logic circuit that uses it to generate the device select signals:  $\overline{RAM}$ ,  $\overline{ROM}$ , and  $\overline{I/O}$ .

Analyze this circuit and determine the following.

- (a)\* The range of addresses  $A_{15}$  through  $A_8$  that will activate  $\overline{RAM}$
- (b) The range of addresses that activate  $\overline{I/O}$
- (c) The range of addresses that activate  $\overline{ROM}$

Express the addresses in binary and hexadecimal. For example, the answer to (a) is  $A_{15}$  to  $A_8 = 00000000_2$  to  $11101111_2 = 00_{16}$  to  $EF_{16}$ .

- C, D** 4-69. In some microcomputers, the MPU can be *disabled* for short periods of time while another device controls the RAM, ROM, and I/O. During these intervals, a special control signal ( $\overline{DMA}$ ) is activated by the MPU and is used to disable (deactivate) the device select logic so that the  $\overline{RAM}$ ,  $\overline{ROM}$ , and  $\overline{I/O}$  are all in their inactive state. Modify the circuit of Figure 4-82 so that  $\overline{RAM}$ ,  $\overline{ROM}$ , and  $\overline{I/O}$  will be deactivated whenever the signal  $\overline{DMA}$  is active, regardless of the state of the address code.

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

### SECTION 4-1

1. Only (a)    2. Only (c)

### SECTION 4-2

1. Smaller, less expensive to build circuit    2. Maintain uniform propagation delays for all circuits

### SECTION 4-3

1. Expression (b) is not in sum-of-products form because of the inversion sign over both the  $C$  and  $D$  variables (i.e., the  $\overline{ACD}$  term). Expression (c) is not in sum-of-products form because of the  $(M + \overline{N})P$  term.    2. See Example 4-1 solution    3.  $x = \overline{A} + \overline{B} + \overline{C}$

### SECTION 4-4

1.  $x = \overline{A}\overline{B}\overline{C}D + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D$     2. Eight: four inverters for  $A$ ,  $B$ ,  $C$ , and  $D$  plus four for SOP    3. Translate to truth tables, write product term for each, generate SOP

### SECTION 4-5

1.  $x = AB + AC + BC$     2.  $z = A + BCD$     3.  $S = \overline{P} + QR$     4. An input condition for which there is no specific required output condition; that is, we are free to make it 0 or 1.

### SECTION 4-6

2. A constant LOW    3. No; the available XOR gate can be used as an INVERTER by connecting one of its inputs to a constant HIGH (see Example 4-16).

### SECTION 4-7

1. seven    2. eight

**SECTION 4-8**

1.  $x = \overline{A(B \oplus C)}$    2.  $x = ABC$    3. OR, NAND   4. NAND, NOR

**SECTION 4-9**

1. (a) bipolar junction transistors (b) unipolar metal oxide semiconductor field effect transistors (MOSFETs).   2. SSI, MSI, LSI, VLSI, ULSI, GSI   3. True  
 4. True   5. 40, 74AC, 74ACT series   6. 0 to 0.8 V; 2.0 to 5.0 V   7. 0 to 1.5 V; 3.5 to 5.0 V   8. As if the input were HIGH   9. Unpredictably; it may overheat and be destroyed.   10. 74HCT and 74ACT   11. They describe exactly how to interconnect the chips for laying out the circuit and troubleshooting.   12. Inputs and outputs are defined, and logical relationships are described.

**SECTION 4-11**

1. Open inputs or outputs; inputs or outputs shorted to  $V_{CC}$ ; inputs or outputs shorted to ground; pins shorted together; internal circuit failures   2. Pins shorted together  
 3. For TTL, a LOW; for CMOS, indeterminate   4. Two or more outputs connected together

**SECTION 4-12**

1. Open signal lines; shorted signal lines; faulty power supply; output loading  
 2. Broken wires; poor solder connections; cracks or cuts in PC board; bent or broken IC pins; faulty IC sockets   3. ICs operating erratically or not at all  
 4. Logic level indeterminate

**SECTION 4-13**

1. HIGH.   2. One of its inputs is LOW   3. LOW   4. One of its inputs is HIGH.   5. If they disagree, the problem is simply in the wiring.   6. Disconnect the wire from the output. If the output is now correct, the problem is in the load.

**SECTION 4-14**

1. Electrically controlled connections are being programmed as open or closed.  
 2. (4, 1) (2, 2) or (2, 1) (4, 2)   3. (4, 5) (1, 6) or (4, 6) (1, 5)   4. See glossary  
 5. JTAG

**SECTION 4-15**

1. (a) `push_buttons[5..0]:INPUT;` (b) `push_buttons :IN BIT_VECTOR (5 DOWNT0 0),`  
 2. (a) `z = push_buttons[5];` (b) `z <= push_buttons(5);`  
 3. `STD_LOGIC`   4. `STD_LOGIC_VECTOR`   5. Decimal, binary, hexadecimal

**SECTION 4-16**

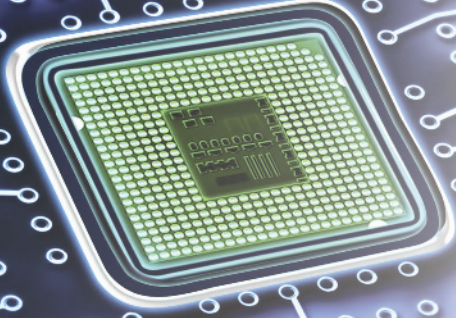
1. (AHDL) `omega[] = (x, y, z);` (VHDL) `omega <= x & y & z;`   2. Using the keyword `TABLE`   3. Using selected signal assignments

**SECTION 4-17**

1. IF/THEN   2. IF/THEN/ELSE   3. CASE or IF/ELSIF   4. True  
 5. True

*This page intentionally left blank*





# FLIP-FLOPS AND RELATED DEVICES

## ■ OUTLINE

- |      |   |      |   |
|------|---|------|---|
| 5-1  | NAND Gate Latch                         | 5-17 | Data Storage and Transfer                         |
| 5-2  | NOR Gate Latch                          | 5-18 | Serial Data Transfer: Shift Registers             |
| 5-3  | Troubleshooting Case Study              | 5-19 | Frequency Division and Counting                   |
| 5-4  | Digital Pulses                          | 5-20 | Application of Flip-Flops with Timing Constraints |
| 5-5  | Clock Signals and Clocked Flip-Flops    | 5-21 | Microcomputer Application                         |
| 5-6  | Clocked S-R Flip-Flop                   | 5-22 | Schmitt-Trigger Devices                           |
| 5-7  | Clocked J-K Flip-Flop                   | 5-23 | One-Shot (Monostable Multivibrator)               |
| 5-8  | Clocked D Flip-Flop                     | 5-24 | Clock Generator Circuits                          |
| 5-9  | D Latch (Transparent Latch)             | 5-25 | Troubleshooting Flip-Flop Circuits                |
| 5-10 | Asynchronous Inputs                     | 5-26 | Sequential Circuits in PLDs Using Schematic Entry |
| 5-11 | Flip-Flop Timing Considerations         | 5-27 | Sequential Circuits Using HDL                     |
| 5-12 | Potential Timing Problem in FF Circuits | 5-28 | Edge-Triggered Devices                            |
| 5-13 | Flip-Flop Applications                  | 5-29 | HDL Circuits with Multiple Components             |
| 5-14 | Flip-Flop Synchronization               |      |   |
| 5-15 | Detecting an Input Sequence             |      |   |
| 5-16 | Detecting a Transition or “Event”       |      |   |

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

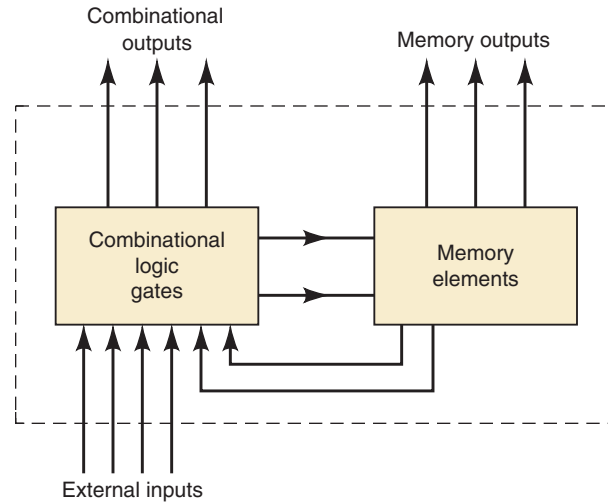
- Construct and analyze the operation of a latch flip-flop made from NAND or NOR gates.
- Describe the difference between synchronous and asynchronous systems.
- Describe the operation of edge-triggered flip-flops.
- Analyze and apply the various flip-flop timing parameters specified by the manufacturers.
- Explain the major differences between parallel and serial data transfers.
- Draw the output timing waveforms of several types of flip-flops in response to a set of input signals.
- Use state transition diagrams to describe counter operation.
- Use flip-flops in synchronization circuits.
- Connect shift registers as data transfer circuits.
- Employ flip-flops as frequency-division and counting circuits.
- Describe the typical characteristics of Schmitt triggers.
- Apply two different types of one-shots in circuit design.
- Design a free-running oscillator using a 555 timer.
- Recognize and predict the effects of clock skew on synchronous circuits.
- Troubleshoot various types of flip-flop circuits.
- Create sequential circuits with PLDs using schematic entry.
- Write HDL code for latches.
- Use logic primitives, components, and libraries in HDL code.
- Build structural level circuits from components.

## ■ INTRODUCTION

The logic circuits considered thus far have been combinational circuits whose output levels at any instant of time are dependent on the levels present at the inputs at that time. Any prior input-level conditions have no effect on the present outputs because combinational logic circuits have no memory. Most digital systems consist of both combinational circuits and memory elements.

Figure 5-1 shows a block diagram of a general digital system that combines combinational logic gates with memory devices. The combinational portion accepts logic signals from external inputs and from the outputs of

**FIGURE 5-1** General digital system diagram.



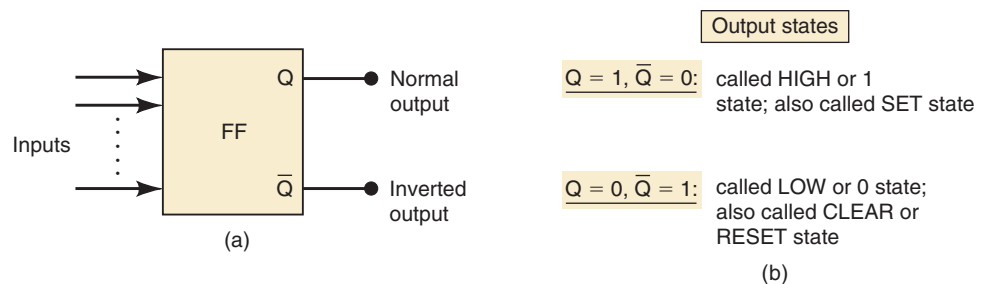
the memory elements. The combinational circuit operates on these inputs to produce various outputs, some of which are used to determine the binary values to be stored in the memory elements. The outputs of some of the memory elements, in turn, go to the inputs of logic gates in the combinational circuits. This process indicates that the external outputs of a digital system are functions of both its external inputs and the information stored in its memory elements.

The most important memory element is the **flip-flop**, which is made up of an assembly of logic gates. Even though a logic gate, by itself, has no storage capability, several can be connected together in ways that permit information to be stored. A memory element can be created by applying the concept of **feedback**. Feedback is accomplished by connecting certain gate outputs back to the appropriate gate inputs. Feedback is an extremely important engineering concept that has many applications in electronics. Several different gate arrangements are used to produce flip-flops (abbreviated FF).

Figure 5-2(a) is the general type of symbol used for a flip-flop. It shows two outputs, labeled  $Q$  and  $\bar{Q}$ , that are the inverse of each other.  $Q/\bar{Q}$  are the most common designations used for a FF's outputs. From time to time, we will use other designations such as  $X/\bar{X}$  and  $A/\bar{A}$  for convenience in identifying different FFs in a logic circuit.

The  $Q$  output is called the *normal* FF output, and  $\bar{Q}$  is the *inverted* FF output. Whenever we refer to the state of a FF, we are referring to the state of its normal ( $Q$ ) output; it is understood that its inverted output ( $\bar{Q}$ ) is in the opposite state. For example, if we say that a FF is in the HIGH (1) state, we mean that  $Q = 1$ ; if we say that a FF is in the LOW (0) state, we mean that  $Q = 0$ . Of course, the  $\bar{Q}$  state will always be the inverse of  $Q$ .

**FIGURE 5-2** General flip-flop symbol and definition of its two possible output states.



The two possible operating states for a FF are summarized in Figure 5-2(b). Note that the HIGH or 1 state ( $Q = 1/\bar{Q} = 0$ ) is also referred to as the **SET** state. Whenever the inputs to a FF cause it to go to the  $Q = 1$  state, we call this *setting* the FF; the FF has been set. In a similar way, the LOW or 0 state ( $Q = 0/\bar{Q} = 1$ ) is also referred to as the **CLEAR** or **RESET** state. Whenever the inputs to a FF cause it to go to the  $Q = 0$  state, we call this *clearing* or *resetting* the FF; the FF has been cleared (reset). As we shall see, many FFs will have a **SET** input and/or a **CLEAR (RESET)** input that is used to drive the FF into a specific output state.

As the symbol in Figure 5-2(a) implies, a FF can have one or more inputs. These inputs are used to cause the FF to switch back and forth (“flip-flop”) between its possible output states. We will find out that most FF inputs need only to be momentarily activated (pulsed) in order to cause a change in the FF output state, and the output will remain in that new state even after the input pulse is over. This is the FF’s *memory* characteristic.

The flip-flop is known by other names, including *latch* and *bistable multivibrator*. The term *latch* is used for certain types of flip-flops that we will describe. The term *bistable multivibrator* is the more technical name for a flip-flop, but it is too much of a mouthful to be used regularly.

## 5-1 NAND GATE LATCH

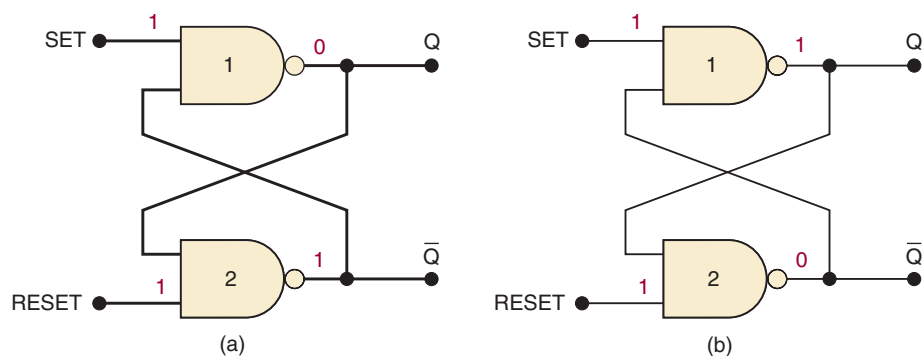
### OUTCOMES

Upon completion of this section, you will be able to:

- Define SET and RESET conditions of a latch or FF.
- Predict the output state given any changes to the inputs of a NAND latch.
- Distinguish between active-HIGH and active-LOW control inputs.

The most basic FF circuit can be constructed from either two NAND gates or two NOR gates. The NAND gate version, called a **NAND gate latch** or simply a **latch**, is shown in Figure 5-3(a). The two NAND gates are cross-coupled so that the output of NAND-1 is connected to one of the inputs of NAND-2, and vice versa. This circuit configuration provides the feedback necessary to produce the memory function. The gate outputs, labeled  $Q$  and  $\bar{Q}$ , respectively, are the latch outputs. Under normal conditions, these outputs will always be the inverse of each other. There are two latch inputs: the **SET** input is the input that *sets*  $Q$  to the 1 state; the **RESET** input is the input that *resets*  $Q$  to the 0 state.

**FIGURE 5-3** A NAND latch has two possible resting states when SET = RESET = 1.

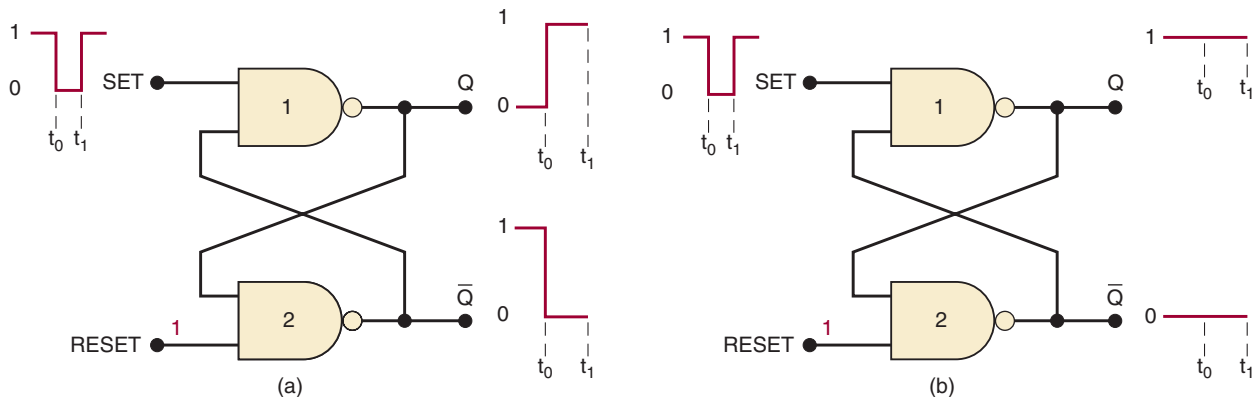


The SET and RESET inputs are both normally resting in the HIGH state, and one of them will be pulsed LOW whenever we want to change the latch outputs. We begin our analysis by showing that there are two equally likely output states when SET = RESET = 1. One possibility is shown in Figure 5-3(a), where we have  $Q = 0$  and  $\bar{Q} = 1$ . With  $Q = 0$ , the inputs to NAND-2 are 0 and 1, which produce  $\bar{Q} = 1$ . The 1 from  $\bar{Q}$  causes NAND-1 to have a 1 at both inputs to produce a 0 output at  $Q$ . In effect, what we have is the LOW at the NAND-1 output producing a HIGH at the NAND-2 output, which, in turn, keeps the NAND-1 output LOW.

The second possibility is shown in Figure 5-3(b), where  $Q = 1$  and  $\bar{Q} = 0$ . The HIGH from NAND-1 produces a LOW at the NAND-2 output, which, in turn, keeps the NAND-1 output HIGH. Thus, there are two possible output states when SET = RESET = 1; as we shall soon see, the one that actually exists will depend on what has occurred previously at the inputs.

### Setting the Latch (FF)

Now let's investigate what happens when the SET input is momentarily pulsed LOW while RESET is kept HIGH. Figure 5-4(a) shows what happens when  $Q = 0$  prior to the occurrence of the pulse. As SET is pulsed LOW at time  $t_0$ ,  $Q$  will go HIGH, and this HIGH will force  $\bar{Q}$  to go LOW so that NAND-1 now has two LOW inputs. Thus, when SET returns to the 1 state at  $t_1$ , the NAND-1 output *remains* HIGH, which, in turn, keeps the NAND-2 output LOW.



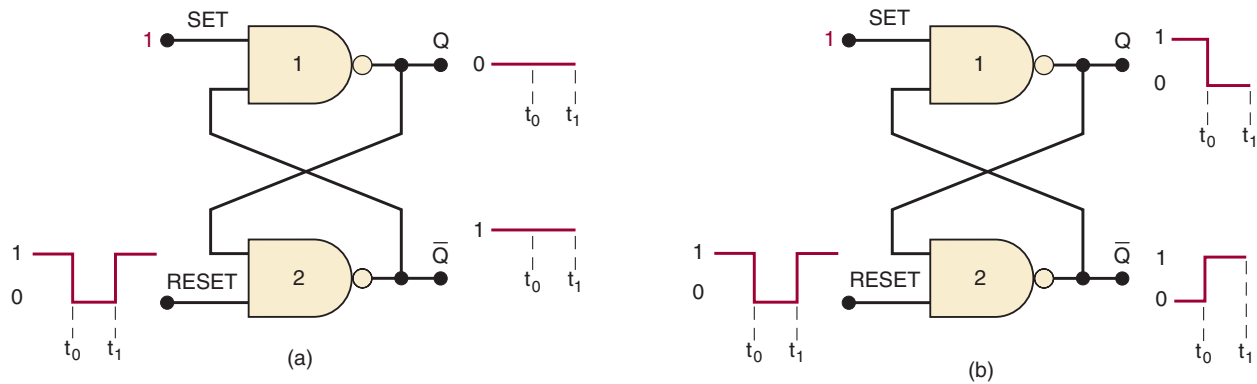
**FIGURE 5-4** Pulsing the SET input to the 0 state when (a)  $Q = 0$  prior to SET pulse; (b)  $Q = 1$  prior to SET pulse. Note that, in both cases,  $Q$  ends up HIGH.

Figure 5-4(b) shows what happens when  $Q = 1$  and  $\bar{Q} = 0$  prior to the application of the SET pulse. Since  $\bar{Q} = 0$  is already keeping the NAND-1 output HIGH, the LOW pulse at SET will not change anything. Thus, when SET returns HIGH, the latch outputs are still in the  $Q = 1$ ,  $\bar{Q} = 0$  state.

We can summarize Figure 5-4 by stating that a LOW pulse on the SET input will always cause the latch to end up in the  $Q = 1$  state. This operation is called *setting* the latch or FF.

### Resetting the Latch (FF)

Now let's consider what occurs when the RESET input is pulsed LOW while SET is kept HIGH. Figure 5-5(a) shows what happens when  $Q = 0$  and  $\bar{Q} = 1$  prior to the application of the pulse. Since  $Q = 0$  is already keeping the



**FIGURE 5-5** Pulsing the RESET input to the LOW state when (a)  $Q = 0$  prior to RESET pulse; (b)  $Q = 1$  prior to RESET pulse. In each case,  $Q$  ends up LOW.

NAND-2 output HIGH, the LOW pulse at RESET will not have any effect. When RESET returns HIGH, the latch outputs are still  $Q = 0$  and  $\bar{Q} = 1$ .

Figure 5-5(b) shows the situation where  $Q = 1$  prior to the occurrence of the RESET pulse. As RESET is pulsed LOW at  $t_0$ ,  $\bar{Q}$  will go HIGH, and this HIGH forces  $Q$  to go LOW so that NAND-2 now has two LOW inputs. Thus, when RESET returns HIGH at  $t_1$ , the NAND-2 output *remains* HIGH, which, in turn, keeps the NAND-1 output LOW.

Figure 5-5 can be summarized by stating that a LOW pulse on the RESET input will always cause the latch to end up in the  $Q = 0$  state. This operation is called *clearing* or *resetting* the latch.

### Simultaneous Setting and Resetting

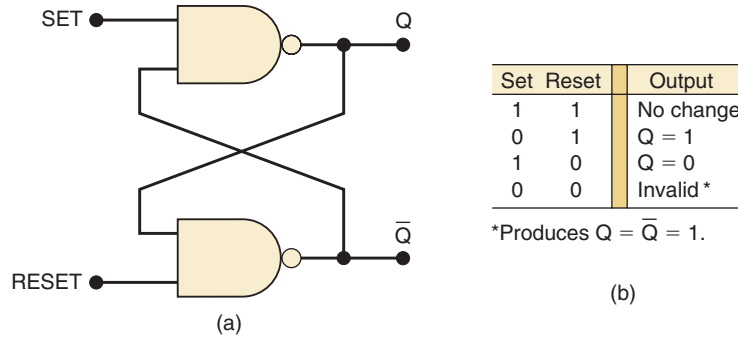
The last case to consider is the case where the SET and RESET inputs are simultaneously pulsed LOW. This will produce HIGH levels at both NAND outputs so that  $Q = \bar{Q} = 1$ . Clearly, this is an undesired condition because the two outputs are supposed to be inverses of each other. Furthermore, when the SET and RESET inputs return HIGH, the resulting output state will depend on which input returns HIGH first. Simultaneous transitions back to the 1 state will produce unpredictable results. For these reasons the  $SET = RESET = 0$  condition is normally not used for the NAND latch.

### Summary of NAND Latch

The operation described above can be conveniently placed in a function table (Figure 5-6) and is summarized as follows:

1.  $SET = RESET = 1$ . This condition is the normal resting state, and it has no effect on the output state. The  $Q$  and  $\bar{Q}$  outputs will remain in whatever state they were in prior to this input condition.
2.  $SET = 0, RESET = 1$ . This will always cause the output to go to the  $Q = 1$  state, where it will remain even after SET returns HIGH. This is called *setting* the latch.
3.  $SET = 1, RESET = 0$ . This will always produce the  $Q = 0$  state, where the output will remain even after RESET returns HIGH. This is called *clearing* or *resetting* the latch.

**FIGURE 5-6** (a) NAND latch; (b) function table.



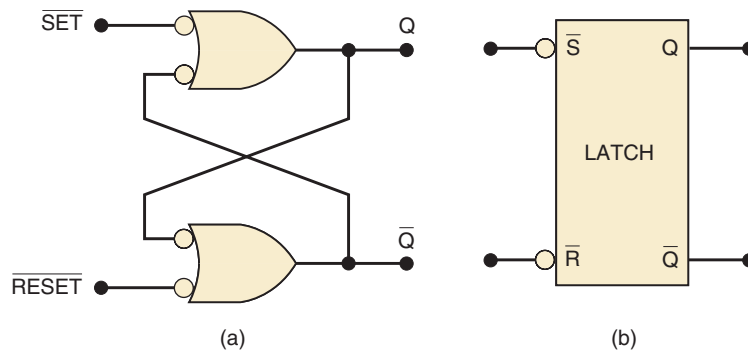
4.  $SET = RESET = 0$ . This condition tries to set and clear the latch at the same time, and it produces  $Q = \bar{Q} = 1$ . If the inputs are returned to 1 simultaneously, the resulting state is unpredictable. This input condition should not be used.

**Alternate Representations**

From the description of the NAND latch operation, it should be clear that the SET and RESET inputs are active-LOW. The SET input will set  $Q = 1$  when SET goes LOW; the RESET input will clear  $Q = 0$  when RESET goes LOW. For this reason, the NAND latch is often drawn using the alternate representation for each NAND gate, as shown in Figure 5-7(a). The bubbles on the inputs, as well as the labeling of the signals as  $\overline{SET}$  and  $\overline{RESET}$ , indicate the active-LOW status of these inputs. (You may want to review Sections 3-13 and 3-14 on this topic.)

Figure 5-7(b) shows a simplified block representation that we will sometimes use. The S and R labels represent the SET and RESET inputs, and the bubbles indicate the active-LOW nature of these inputs. Whenever we use this block symbol, it represents a NAND latch. The NAND latch and the NOR latch (presented in Section 5-2) are commonly called **S-R latches**.

**FIGURE 5-7** (a) NAND latch equivalent representation; (b) simplified block symbol.



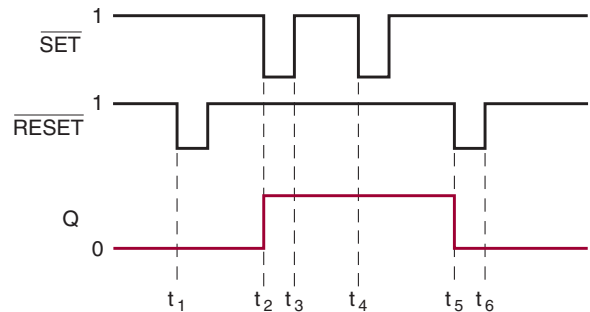
**Terminology**

The action of *resetting* a FF or a latch is also called *clearing*, and both terms are used interchangeably in the digital field. In fact, a RESET input can also be called a CLEAR input, and a SET-RESET latch can be called a SET-CLEAR latch.

## EXAMPLE 5-1

The waveforms of Figure 5-8 are applied to the inputs of the latch of Figure 5-7. Assume that initially  $Q = 0$ , and determine the  $Q$  waveform.

FIGURE 5-8 Example 5-1.



### Solution

Initially,  $\overline{\text{SET}} = \overline{\text{RESET}} = 1$  so that  $Q$  will remain in the 0 state. The LOW pulse that occurs on the  $\overline{\text{RESET}}$  input at time  $t_1$  will have no effect because  $Q$  is already in the cleared (0) state.

The only way that  $Q$  can go to the 1 state is by a LOW pulse on the  $\overline{\text{SET}}$  input. This occurs at time  $t_2$  when  $\overline{\text{SET}}$  first goes LOW. When  $\overline{\text{SET}}$  returns HIGH at  $t_3$ ,  $Q$  will remain in its new HIGH state.

At time  $t_4$  when  $\overline{\text{SET}}$  goes LOW again, there will be no effect on  $Q$  because  $Q$  is already set to the 1 state.

The only way to bring  $Q$  back to the 0 state is by a LOW pulse on the  $\overline{\text{RESET}}$  input. This occurs at time  $t_5$ . When  $\overline{\text{RESET}}$  returns to 1 at time  $t_6$ ,  $Q$  remains in the LOW state.

Example 5-1 shows that the latch output “remembers” the last input that was activated and will not change states until the opposite input is activated.

## EXAMPLE 5-2

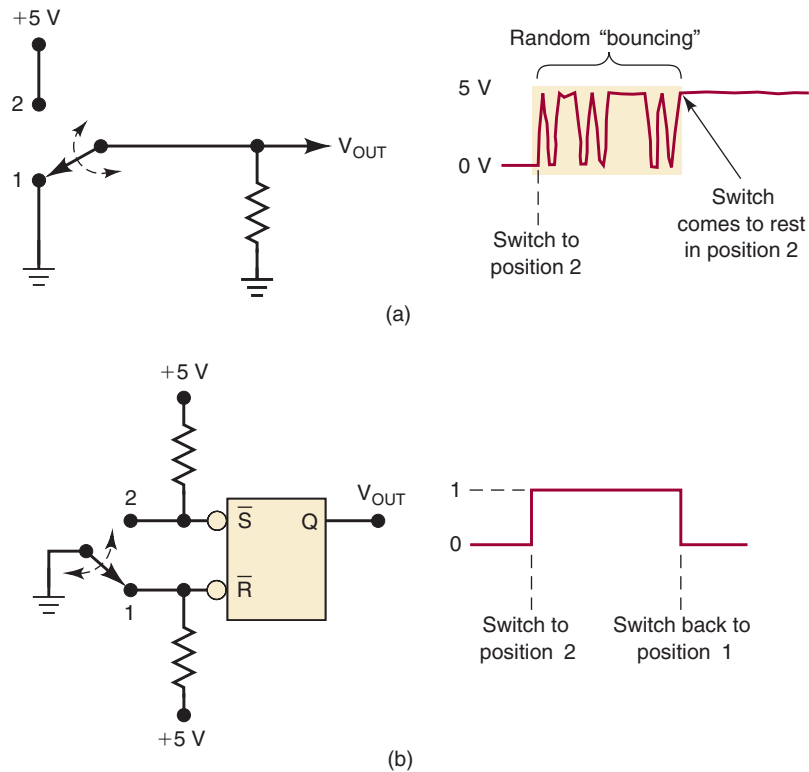
It is almost impossible to obtain a “clean” voltage transition from a mechanical switch because of the phenomenon of **contact bounce**. This is illustrated in Figure 5-9(a), where the action of moving the switch from contact position 1 to 2 produces several output voltage transitions as the switch bounces (makes and breaks contact with contact 2 several times) before coming to rest on contact 2.

The multiple transitions on the output signal generally last no longer than a few milliseconds, but they would be unacceptable in many applications. A NAND latch can be used to prevent the presence of contact bounce from affecting the output. Describe the operation of the “switch debouncing” circuit in Figure 5-9(b).



**FIGURE 5-9**

(a) Mechanical contact bounce will produce multiple transitions; (b) NAND latch used to debounce a mechanical switch.

**Solution**

Assume that the switch is resting in position 1 so that the  $\overline{RESET}$  input is LOW and  $Q = 0$ . When the switch is moved to position 2,  $\overline{RESET}$  will go HIGH, and a LOW will appear on the  $\overline{SET}$  input as the switch first makes contact. This will set  $Q = 1$  within a matter of a few nanoseconds (the response time of the NAND gate). Now if the switch bounces off contact 2,  $\overline{SET}$  and  $\overline{RESET}$  will both be HIGH, and  $Q$  will not be affected; it will stay HIGH. Thus, nothing will happen at  $Q$  as the switch bounces on and off contact 2 before finally coming to rest in position 2.

Likewise, when the switch is moved from position 2 back to position 1, it will place a LOW on the  $\overline{RESET}$  input as it first makes contact. This clears  $Q$  to the LOW state, where it will remain even if the switch bounces on and off contact 1 several times before coming to rest.

Thus, the output at  $Q$  will consist of a single transition each time the switch is moved from one position to the other.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is the normal resting state of the  $\overline{SET}$  and  $\overline{RESET}$  inputs? What is the active state of each input?
2. What will be the states of  $Q$  and  $\overline{Q}$  after a FF has been reset (cleared)?
3. *True or false:* The  $\overline{SET}$  input can never be used to make  $Q = 0$ .
4. When power is first applied to any FF circuit, it is impossible to predict the initial states of  $Q$  and  $\overline{Q}$ . What can be done to ensure that a NAND latch always starts off in the  $Q = 1$  state?

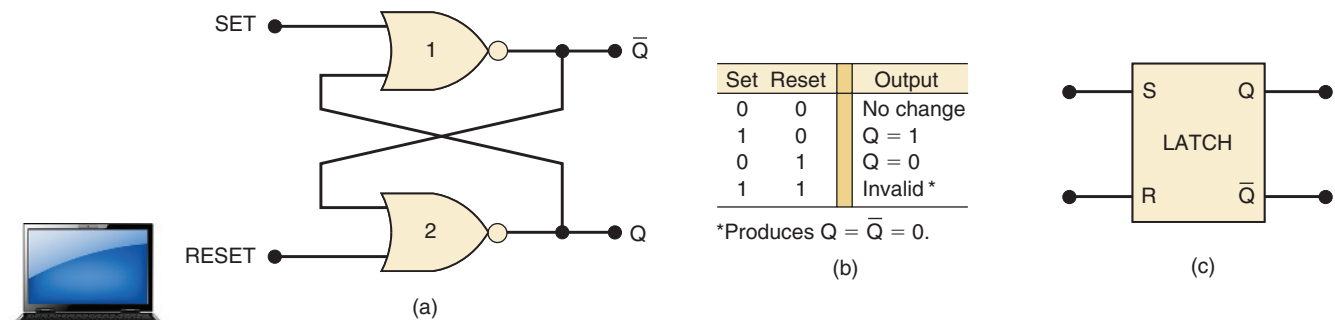
## 5-2 NOR GATE LATCH

### OUTCOMES

Upon completion of this section, you will be able to:

- Predict the output state given any changes to the inputs of a NOR latch.
- Distinguish between active-HIGH and active-LOW control inputs.

Two cross-coupled NOR gates can be used as a **NOR gate latch**. The arrangement, shown in Figure 5-10(a), is similar to the NAND latch except that the  $Q$  and  $\bar{Q}$  outputs have reversed positions.



**FIGURE 5-10** (a) NOR gate latch; (b) function table; (c) simplified block symbol.

The analysis of the operation of the NOR latch can be performed in exactly the same manner as for the NAND latch. The results are given in the function table in Figure 5-10(b) and are summarized as follows:

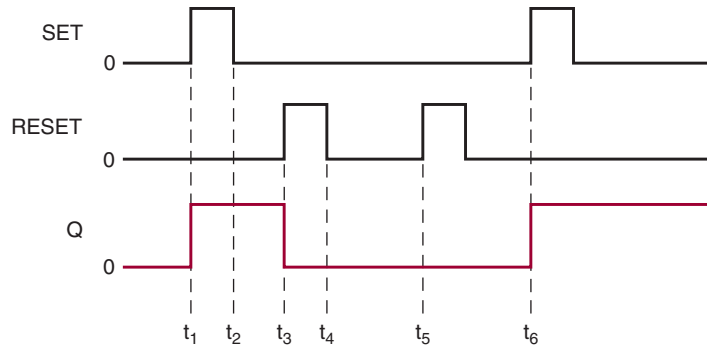
1.  $SET = RESET = 0$ . This is the normal resting state for the NOR latch, and it has no effect on the output state.  $Q$  and  $\bar{Q}$  will remain in whatever state they were in prior to the occurrence of this input condition.
2.  $SET = 1, RESET = 0$ . This will always set  $Q = 1$ , where it will remain even after  $SET$  returns to 0.
3.  $SET = 0, RESET = 1$ . This will always clear  $Q = 0$ , where it will remain even after  $RESET$  returns to 0.
4.  $SET = 1, RESET = 1$ . This condition tries to set and reset the latch at the same time, and it produces  $Q = \bar{Q} = 0$ . If the inputs are returned to 0 simultaneously, the resulting output state is unpredictable. This input condition should not be used.

The NOR gate latch operates exactly like the NAND latch except that the  $SET$  and  $RESET$  inputs are active-HIGH rather than active-LOW, and the normal resting state is  $SET = RESET = 0$ .  $Q$  will be set HIGH by a HIGH pulse on the  $SET$  input, and it will be cleared LOW by a HIGH pulse on the  $RESET$  input. The simplified block symbol for the NOR latch in Figure 5-10(c) is shown with no bubbles on the  $S$  and  $R$  inputs; this indicates that these inputs are active-HIGH.

### EXAMPLE 5-3

Assume that  $Q = 0$  initially, and determine the  $Q$  waveform for the NOR latch inputs of Figure 5-11.

FIGURE 5-11 Example 5-3.

**Solution**

Initially, SET = RESET = 0, which has no effect on Q, and Q stays LOW. When SET goes HIGH at time  $t_1$ , Q will be set to 1 and will remain there even after SET returns to 0 at  $t_2$ .

At  $t_3$  the RESET input goes HIGH and clears Q to the 0 state, where it remains even after RESET returns LOW at  $t_4$ .

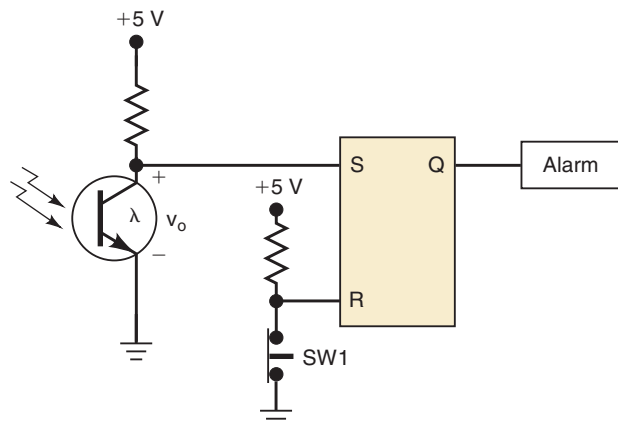
The RESET pulse at  $t_5$  has no effect on Q because Q is already LOW. The SET pulse at  $t_6$  again sets Q back to 1, where it will stay.

Example 5-3 shows that the latch “remembers” the last input that was activated, and it will not change states until the opposite input is activated.

**EXAMPLE 5-4**

Figure 5-12 shows a simple circuit that can be used to detect the interruption of a light beam. The light is focused on a phototransistor that is connected in the common-emitter configuration to operate as a switch. Assume that the latch has previously been cleared to the 0 state by momentarily opening switch SW1, and describe what happens if the light beam is momentarily interrupted.

FIGURE 5-12 Example 5-4.



### Solution

With light on the phototransistor, we can assume that it is fully conducting so that the resistance between the collector and the emitter is very small. Thus,  $v_0$  will be close to 0 V. This places a LOW on the SET input of the latch so that  $\text{SET} = \text{RESET} = 0$ .

When the light beam is interrupted, the phototransistor turns off, and its collector-emitter resistance becomes very high (i.e., essentially an open circuit). This causes  $v_0$  to rise to approximately 5 V; this activates the SET input, which sets  $Q$  HIGH and turns on the alarm.

$Q$  will remain HIGH and the alarm will remain on even if  $v_0$  returns to 0 V (i.e., the light beam was interrupted only momentarily) because SET and RESET will both be LOW, which will produce no change in  $Q$ .

In this application, the latch's memory characteristic is used to convert a momentary occurrence (beam interruption) into a constant output. The alarm will be deactivated again when the latch is reset by momentarily opening SW1 and allowing the RESET input to be pulled HIGH with the resistor. Note that if we attempt to reset the latch while the light beam is interrupted, it will produce the invalid latch input condition of  $\text{SET} = \text{RESET} = 1$ . It will be necessary to hold SW1 open until the light beam is restored to reset the alarm latch.

### Flip-Flop State on Power-Up

When power is applied to a circuit, it is not possible to predict the starting state of a flip-flop's output if its SET and RESET inputs are in their inactive state (e.g.,  $S = R = 1$  for a NAND latch,  $S = R = 0$  for a NOR latch). There is just as much chance that the starting state will be  $Q = 0$  as  $Q = 1$ . It will depend on factors such as internal propagation delays, parasitic capacitance, and external loading. If a latch or FF must start off in a particular state to ensure the proper operation of a circuit, then it must be placed in that state by momentarily activating the SET or RESET input at the start of the circuit's operation. This is often achieved by application of a pulse to the appropriate input.

#### OUTCOME ASSESSMENT QUESTIONS

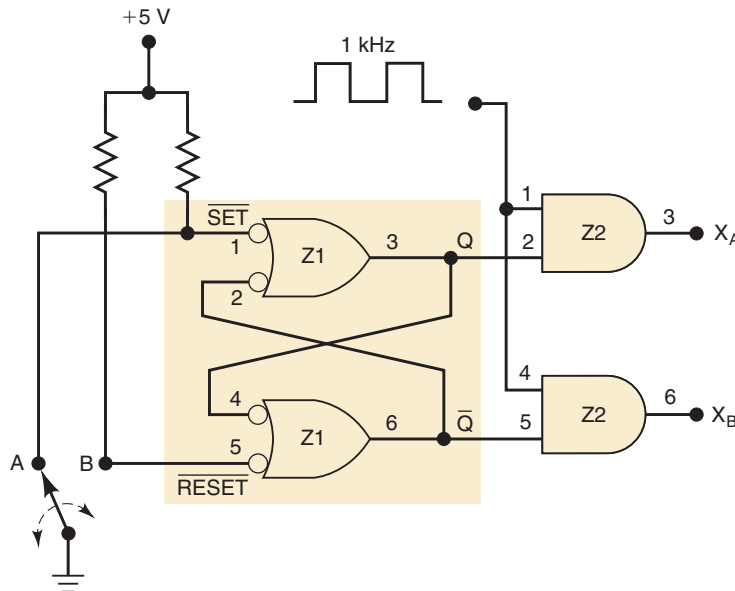
1. What is the normal resting state of the NOR latch inputs? What is the active state?
2. When a latch is set, what are the states of  $Q$  and  $\bar{Q}$ ?
3. What is the only way to cause the  $Q$  output of a NOR latch to change from 1 to 0?
4. If the NOR latch in Figure 5-12 were replaced by a NAND latch, why wouldn't the circuit work properly?

### 5-3 TROUBLESHOOTING CASE STUDY

The following two examples present an illustration of the kinds of reasoning used in troubleshooting a circuit containing a latch.

**EXAMPLE 5-5**

Analyze and describe the operation of the circuit in Figure 5-13.



**FIGURE 5-13** Examples 5-5 and 5-6.

**Solution**

The switch is used to set or clear the NAND latch to produce clean, bounce-free signals at  $Q$  and  $\bar{Q}$ . These latch outputs control the passage of the 1-kHz pulse signal through to the AND outputs  $X_A$  and  $X_B$ .

When the switch moves to position  $A$ , the latch is set to  $Q = 1$ . This enables the 1-kHz pulses to pass through to  $X_A$ , while the LOW at  $\bar{Q}$  keeps  $X_B = 0$ . When the switch moves to position  $B$ , the latch is cleared to  $Q = 0$ , which keeps  $X_A = 0$ , while the HIGH at  $\bar{Q}$  enables the pulses to pass through to  $X_B$ .

**EXAMPLE 5-6**

A technician tests the circuit of Figure 5-13 and records the observations shown in Table 5-1. He notices that when the switch is in position  $B$ , the circuit functions correctly, but in position  $A$  the latch does not set to the  $Q = 1$  state. What are the possible faults that could produce this malfunction?

**TABLE 5-1**

Switch Position	SET (Z1-1)	RESET (Z1-5)	$Q$ (Z1-3)	$\bar{Q}$ (Z1-6)	$X_A$ (Z2-3)	$X_B$ (Z2-6)
A	LOW	HIGH	LOW	HIGH	LOW	Pulses
B	HIGH	LOW	LOW	HIGH	LOW	Pulses

**Solution**

There are several possibilities:

1. An internal open connection at Z1-1, which would prevent  $Q$  from responding to the  $\overline{SET}$  input.
2. An internal component failure in NAND gate Z1 that prevents it from responding properly.
3. The  $Q$  output is stuck LOW, which could be caused by:
  - (a) Z1-3 internally shorted to ground
  - (b) Z1-4 internally shorted to ground
  - (c) Z2-2 internally shorted to ground
  - (d) The  $Q$  node externally shorted to ground

An ohmmeter check from  $Q$  to ground will determine if any of these conditions are present. A visual check should reveal any external short.

What about  $\overline{Q}$  internally or externally shorted to  $V_{CC}$ ? A little thought will lead to the conclusion that this could not be the fault. If  $\overline{Q}$  were shorted to  $V_{CC}$ , this would not prevent the  $Q$  output from going HIGH when  $\overline{SET}$  goes LOW. Because  $Q$  *does not* go HIGH, this cannot be the fault. The reason that  $\overline{Q}$  looks as if it is stuck HIGH is that  $Q$  is stuck LOW, and that keeps  $\overline{Q}$  HIGH through the bottom NAND gate.

## 5-4 DIGITAL PULSES

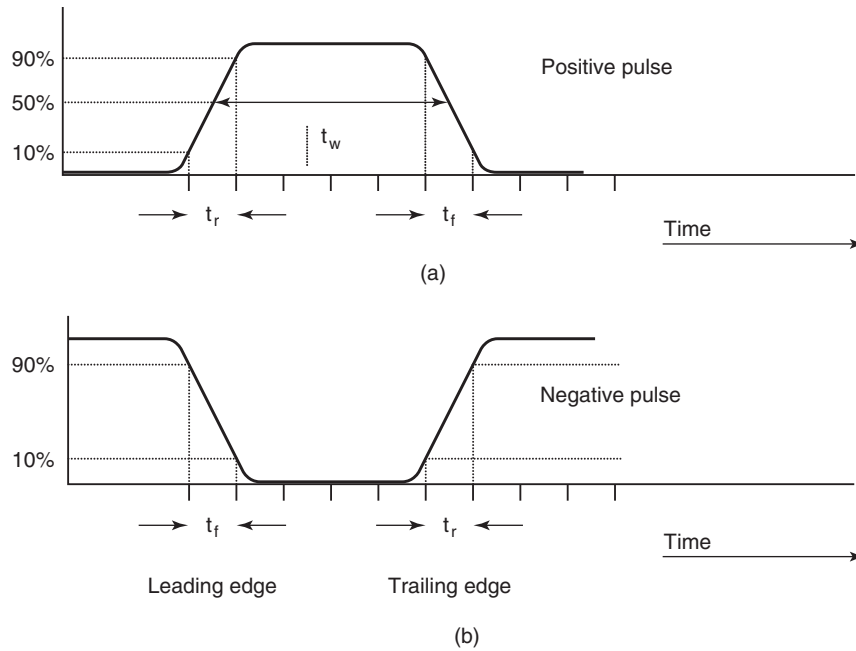
### OUTCOMES

*Upon completion of this section, you will be able to:*

- Identify the active state of a signal.
- Classify as positive or negative pulse.
- Define terms common to pulse waveforms (positive/negative pulses, leading/trailing edge, rise time, fall time, etc.).
- Measure pulse characteristics on a timing diagram or waveform.

As you can see from our discussion of S-R latches, there are situations in digital systems when a signal switches from a normal inactive state to the opposite (active) state, thus causing something to happen in the circuit. Then the signal returns to its inactive state while the effect of the recently activated signal remains in the system. These signals are called **pulses**, and it is very important to understand the terminology associated with pulses and pulse waveforms. A pulse that performs its intended function when it goes HIGH is called a *positive* pulse, and a pulse that performs its intended function when it goes LOW is called a *negative* pulse. In actual circuits it takes time for a pulse waveform to change from one level to the other. These transition times are called the rise time ( $t_r$ ) and the fall time ( $t_f$ ) and are defined as the time it takes the voltage to change between 10 and 90% of the HIGH level voltage as shown on the positive pulse in Figure 5-14(a). The transition at the beginning of the pulse is called the leading edge and the transition at the end of the pulse is the trailing edge. The duration (width) of the pulse ( $t_w$ ) is defined as the time between the points when the leading and trailing edges are at 50% of the HIGH level voltage. Figure 5-14(b) shows an active-LOW or negative pulse.

**FIGURE 5-14** (a) A positive pulse and (b) a negative pulse.



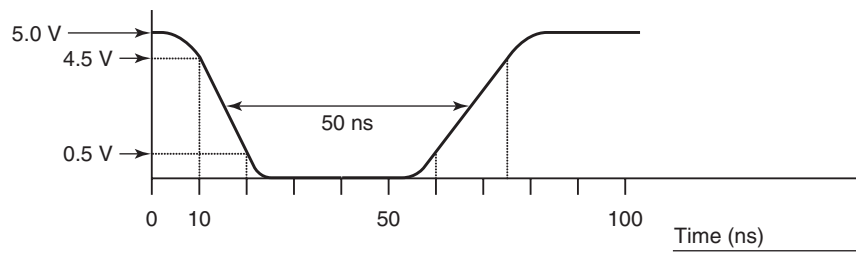
### EXAMPLE 5-7

When a microcontroller wants to access data in its external memory, it activates an active-LOW output pin called  $\overline{RD}$  (read). The data book says that the  $\overline{RD}$  pulse typically has a pulse width  $t_w$  of 50 ns, a rise time  $t_r$  of 15 ns, and a fall time  $t_f$  of 10 ns. Draw a scaled drawing of the  $\overline{RD}$  pulse.

### Solution

Figure 5-15 shows the drawing of the pulse. The  $\overline{RD}$  pulse is active-LOW, so the leading edge is a falling edge measured by  $t_f$  and the trailing edge is the rising edge measured by  $t_r$ .

**FIGURE 5-15**  
Example 5-7.



### OUTCOME ASSESSMENT QUESTIONS

1. Define the following: rise time, fall time, rising edge, falling edge, leading edge, trailing edge, positive pulse, negative pulse, pulse width.
2. Where is pulse width measured?
3. Where is rise time measured?
4. Where is fall time measured?

## 5-5 CLOCK SIGNALS AND CLOCKED FLIP-FLOPS

### OUTCOMES

Upon completion of this section, you will be able to:

- Differentiate between synchronous and asynchronous inputs for a FF.
- Define edge triggering.
- Define setup time.
- Define hold time.
- Define metastability in a FF.

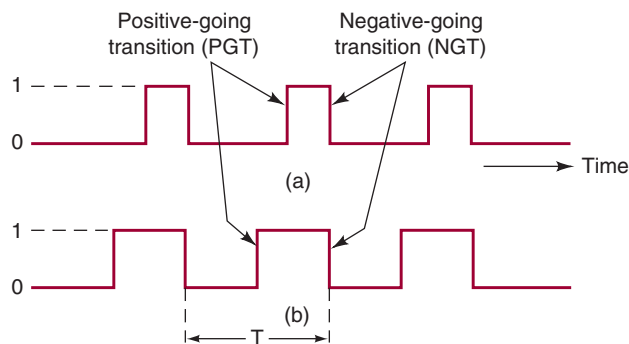
Digital systems can operate either *asynchronously* or *synchronously*. In asynchronous systems, the outputs of logic circuits can change state any time one or more of the inputs change. An asynchronous system is generally more difficult to design and troubleshoot than a synchronous system.

In synchronous systems, the exact times at which any output can change states are determined by a signal commonly called the **clock**. This clock signal is generally a rectangular pulse train or a square wave, as shown in Figure 5-16. The clock signal is distributed to all parts of the system, and most (if not all) of the system outputs can change state only when the clock makes a transition. The transitions (also called *edges*) are pointed out in Figure 5-16. When the clock changes from a 0 to a 1, this is called the **positive-going transition (PGT)**; when the clock goes from 1 to 0, this is the **negative-going transition (NGT)**. We will use the abbreviations PGT and NGT because these terms appear so often throughout the text.

Most digital systems are principally synchronous (although there are always some asynchronous parts) because synchronous circuits are easier to design and troubleshoot. They are easier to troubleshoot because the circuit outputs can change only at specific instants of time. In other words, almost everything is synchronized to the clock-signal transitions.

The synchronizing action of the clock signals is accomplished through the use of **clocked flip-flops** that are designed to change states on one or the other of the clock's transitions.

**FIGURE 5-16** Clock signals.



The speed at which a synchronous digital system operates is dependent on how often the clock cycles occur. A clock cycle is measured from one PGT to the next PGT or from one NGT to the next NGT. The time it takes to complete one cycle (seconds/cycle) is called the **period ( $T$ )**, as shown in Figure 5-16(b). The speed of a digital system is normally referred to by the number of clock cycles that happen in 1 s (cycles/second), which is known as the **frequency ( $f$ )** of the clock. The standard unit for frequency is hertz. One hertz (1 Hz) = 1 cycle/second.



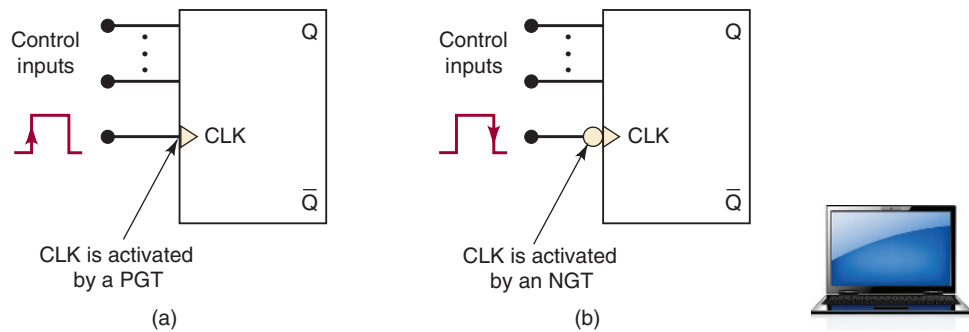
## Clocked Flip-Flops

Several types of clocked FFs are used in a wide range of applications. Before we begin our study of the different clocked FFs, we will describe the principal ideas that are common to all of them.

1. Clocked FFs have a clock input that is typically labeled *CLK*, *CK*, or *CP* (clock pulse). We will normally use *CLK*, as shown in Figure 5-17. In most clocked FFs, the *CLK* input is **edge-triggered**, which means that it is activated by a signal transition; this is indicated by the presence of a small triangle on the *CLK* input. This contrasts with the latches, which are level-triggered.

Figure 5-17(a) is a FF with a small triangle on its *CLK* input to indicate that this input is activated *only* when a positive-going transition occurs; no other part of the input pulse will have an effect on the *CLK* input. In Figure 5-17(b), the FF symbol has a bubble as well as a triangle on its *CLK* input. This signifies that the *CLK* input is activated *only* when a negative-going transition occurs; no other part of the input pulse will have an effect on the *CLK* input.

**FIGURE 5-17** Clocked FFs have a clock input (*CLK*) that is active on either (a) the PGT or (b) the NGT. The control inputs determine the effect of the active clock transition.



2. Clocked FFs also have one or more **control inputs** that can have various names, depending on their operation. The control inputs will have no effect on *Q* until the active clock transition occurs. In other words, their effect is synchronized with the signal applied to *CLK*. For this reason they are called **synchronous control inputs**.

For example, the control inputs of the FF in Figure 5-17(a) will have no effect on *Q* until the PGT of the clock signal occurs. Likewise, the control inputs in Figure 5-17(b) will have no effect until the NGT of the clock signal occurs.

3. In summary, we can say that the control inputs get the FF outputs ready to change, while the active transition at the *CLK* input actually *triggers* the change. The control inputs control the **WHAT** (i.e., what state the output will go to); the *CLK* input determines the **WHEN**.

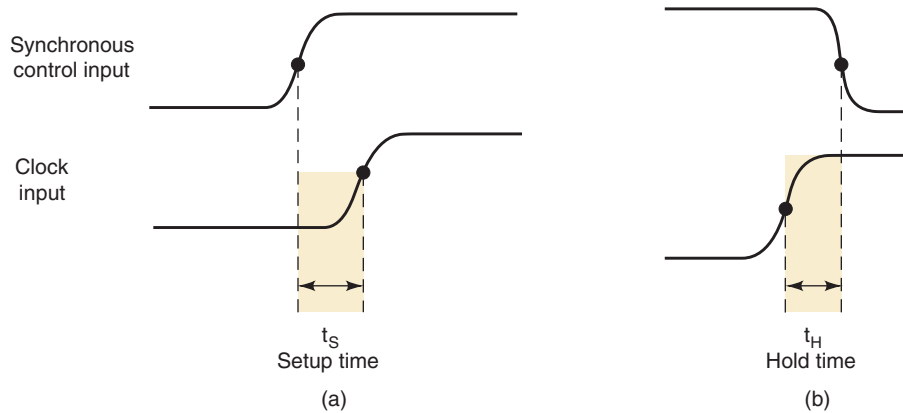
## Setup and Hold Times

A flip-flop that triggers reliably will respond to the active clock edge by reading the input and, after a predictable propagation delay, updating its output appropriately. This section investigates some timing requirements that must be met if the flip-flop is expected to trigger reliably. Unreliable triggering could mean that the output settles in the wrong state. For example, assume a flip-flop is initially reset. If we change the data on the control inputs too

closely to the clock edge, we may think we commanded the flip-flop to set but it may remain reset. Another possible response is that the flip-flop will perform in an unacceptable way before settling to either HIGH or LOW. In other words, there can be some short-term abnormal voltages present on the output that are referred to as *metastable states*. The output may begin to change from its original to the opposite state but then returns to its original state. A third possibility is that it may start to change, hesitate in the invalid region, and then proceed to settle in the opposite state. In any case, metastability can confuse the other logic circuits and cause the system to respond improperly.

Two timing requirements must be met if a clocked FF is to respond reliably to its control inputs when the active *CLK* transition occurs. These requirements are illustrated in Figure 5-18 for a FF that triggers on a PGT.

**FIGURE 5-18** Control inputs must be held stable for (a) a time  $t_S$  prior to active clock transition and for (b) a time  $t_H$  after the active clock transition.

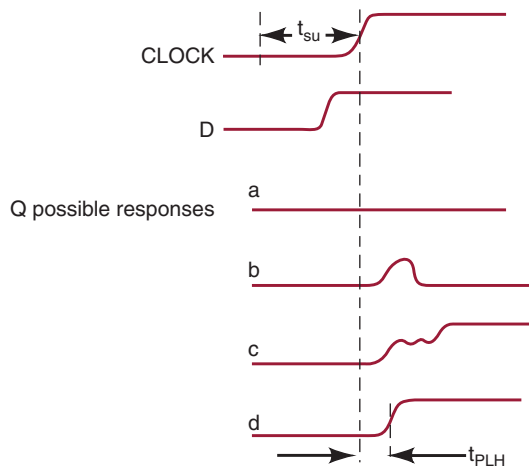


The **setup time**,  $t_S$ , is the time interval immediately preceding the active transition of the *CLK* signal during which the control input must be maintained at the proper level. IC manufacturers usually specify the minimum allowable setup time  $t_{S(\min)}$ . If this time requirement is not met, the FF may not respond reliably when the clock edge occurs.

The **hold time**,  $t_H$ , is the time interval immediately following the active transition of the *CLK* signal during which the synchronous control input must be maintained at the proper level. IC manufacturers usually specify the minimum acceptable value of hold time  $t_{H(\min)}$ . If this requirement is not met, the FF will not trigger reliably.

Figure 5-19 shows the four possible results that may occur for any given flip-flop when its published setup or hold times are violated. Possibility (a)

**FIGURE 5-19** (a) Q does not change; (b) meta-stable response, Q settles LOW; (c) metastable response, Q settles HIGH; (d) Q changes as intended.



is that the output does not respond at all to the HIGH input. Possibility (b) is that the output tries to go HIGH but falls back LOW. Possibility (c) is that the output begins to go HIGH, waivers indecisively in the invalid region, and then responds appropriately by settling HIGH. Possibility (d) is that the flip-flop may respond as intended by going HIGH.

Thus, to ensure that a clocked FF will respond properly when the active clock transition occurs, the control inputs must be stable (unchanging) for at least a time interval equal to  $t_S(\text{min})$  *prior* to the clock transition, and for at least a time interval equal to  $t_H(\text{min})$  *after* the clock transition. These time intervals are required to allow for the propagation delays of the internal gates that control the operation of the flip-flop devices.

IC flip-flops will have minimum allowable  $t_S$  and  $t_H$  values in the nanosecond range. Setup times are usually in the range of 5 to 50 ns, whereas hold times are generally from 0 to 10 ns. Notice that these times are measured between the 50 percent points on the transitions.

These timing requirements are very important in synchronous systems because, as we shall see, there will be many situations where the synchronous control inputs to a FF are changing at approximately the same time as the *CLK* input.

#### OUTCOME ASSESSMENT QUESTIONS

1. What two types of inputs does a clocked FF have?
2. What is meant by the term *edge-triggered*?
3. *True or false:* The *CLK* input will affect the FF output only when the active transition of the control input occurs.
4. Define the setup time and hold time requirements of a clocked FF.
5. *True or false:* Metastable states are the greatest benefit of using clock flip-flops.
6. What causes a flip-flop to exhibit a metastable state?

## 5-6 CLOCKED S-R FLIP-FLOP

### OUTCOMES

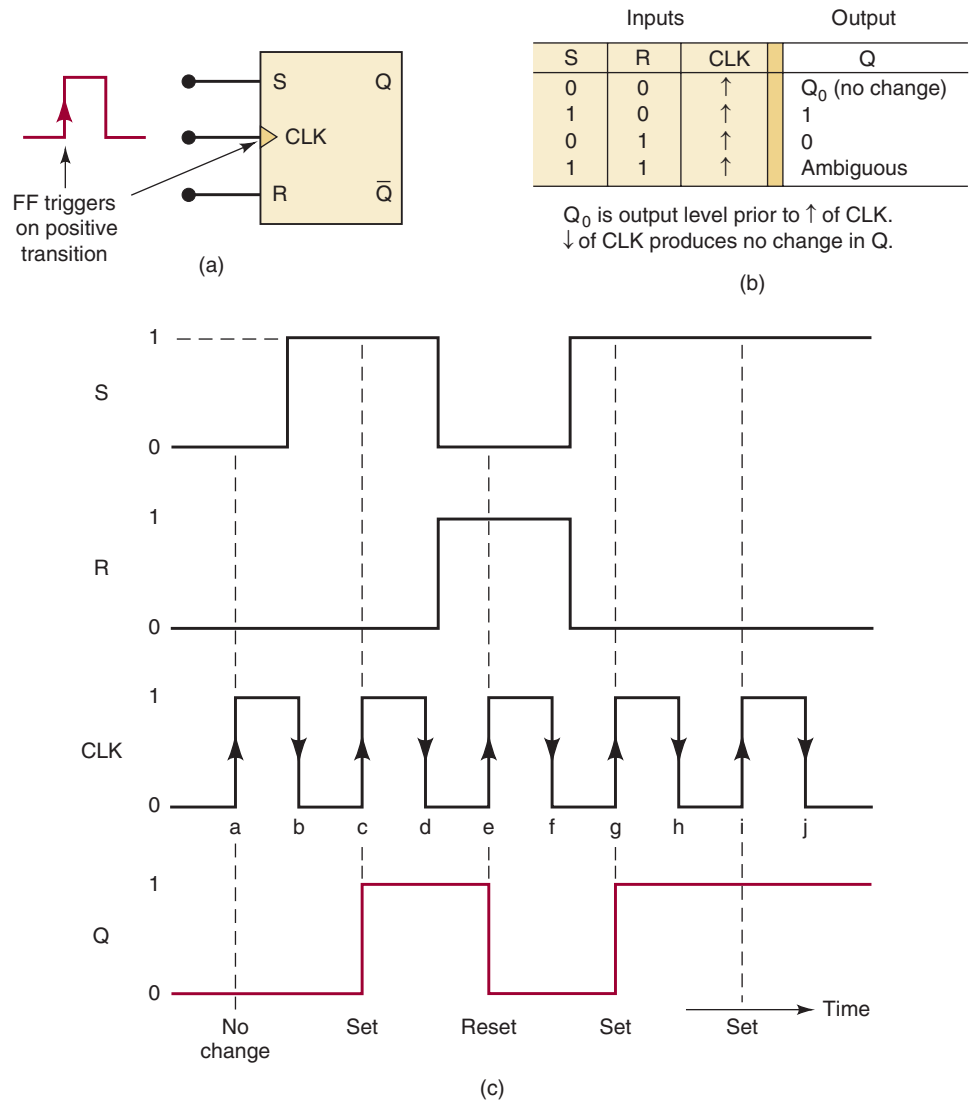
*Upon completion of this section, you will be able to:*

- Predict the *Q* output given any change on any input, *S*, *R*, or clock.
- Draw a timing diagram to demonstrate input changes and resulting output states.

Figure 5-20(a) shows the logic symbol for a **clocked S-R flip-flop** that is triggered by the positive-going edge of the clock signal. This means that the FF can change states *only* when a signal applied to its clock input makes a transition from 0 to 1. The *S* and *R* inputs control the state of the FF in the same manner as described earlier for the NOR gate latch, but the FF does not respond to these inputs until the occurrence of the PGT of the clock signal.

The function table in Figure 5-20(b) shows how the FF output will respond to the PGT at the *CLK* input for the various combinations of *S* and *R* inputs. This function table uses some new nomenclature. The up arrow ( $\uparrow$ ) indicates that a PGT is required at *CLK*; the label  $Q_0$  indicates the level

**FIGURE 5-20** (a) Clocked S-R flip-flop that responds only to the positive-going edge of a clock pulse;  
(b) function table;  
(c) typical waveforms.



at  $Q$  prior to the PGT. This nomenclature is often used by IC manufacturers in their IC data manuals.

The waveforms in Figure 5-20(c) illustrate the operation of the clocked S-R flip-flop. If we assume that the setup and hold time requirements are being met in all cases, we can analyze these waveforms as follows:

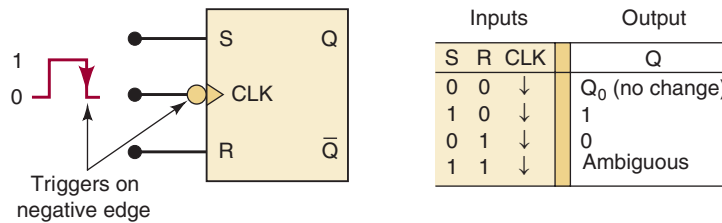
1. Initially all inputs are 0 and the  $Q$  output is assumed to be 0; that is,  $Q_0 = 0$ .
2. When the PGT of the first clock pulse occurs (point  $a$ ), the  $S$  and  $R$  inputs are both 0, so the FF is not affected and remains in the  $Q = 0$  state (i.e.,  $Q = Q_0$ ).
3. At the occurrence of the PGT of the second clock pulse (point  $c$ ), the  $S$  input is now HIGH, with  $R$  still LOW. Thus, the FF sets to the 1 state at the rising edge of this clock pulse.
4. When the third clock pulse makes its positive transition (point  $e$ ), it finds that  $S = 0$  and  $R = 1$ , which causes the FF to clear to the 0 state.
5. The fourth pulse sets the FF once again to the  $Q = 1$  state (point  $g$ ) because  $S = 1$  and  $R = 0$  when the positive edge occurs.

6. The fifth pulse also finds that  $S = 1$  and  $R = 0$  when it makes its positive-going transition. However,  $Q$  is already HIGH, so it remains in that state.
7. The  $S = R = 1$  condition should not be used because it results in an ambiguous condition.

It should be noted from these waveforms that the FF is not affected by the negative-going transitions of the clock pulses. Also, note that the  $S$  and  $R$  levels have no effect on the FF, except upon the occurrence of a positive-going transition of the clock signal. The  $S$  and  $R$  inputs are synchronous *control* inputs; they control which state the FF will go to when the clock pulse occurs. The  $CLK$  input is the *trigger* input that causes the FF to change states according to what the  $S$  and  $R$  inputs are when the active clock transition occurs.

Figure 5-21 shows the symbol and the function table for a clocked S-R flip-flop that triggers on the *negative*-going transition at its  $CLK$  input. The small circle and triangle on the  $CLK$  input indicates that this FF will trigger only when the  $CLK$  input goes from 1 to 0. This FF operates in the same manner as the positive-edge FF except that the output can change states only on the falling edge of the clock pulses (points *b*, *d*, *f*, *h*, and *j* in Figure 5-20). Both positive-edge and negative-edge triggering FFs are used in digital systems.

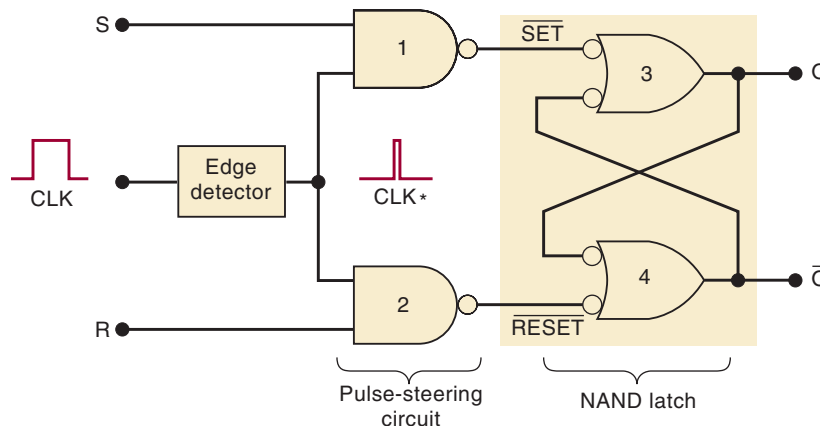
**FIGURE 5-21** Clocked S-R flip-flop that triggers only on negative-going transitions.



### Internal Circuitry of the Edge-Triggered S-R Flip-Flop

A detailed analysis of the internal circuitry of a clocked FF is not necessary because all types are readily available as ICs. Although our main interest is in the FF's external operation, our understanding of this external operation can be aided by taking a look at a simplified version of the FF's internal circuitry. Figure 5-22 shows this for an edge-triggered S-R flip-flop.

**FIGURE 5-22** Simplified version of the internal circuitry for an edge-triggered S-R flip-flop.



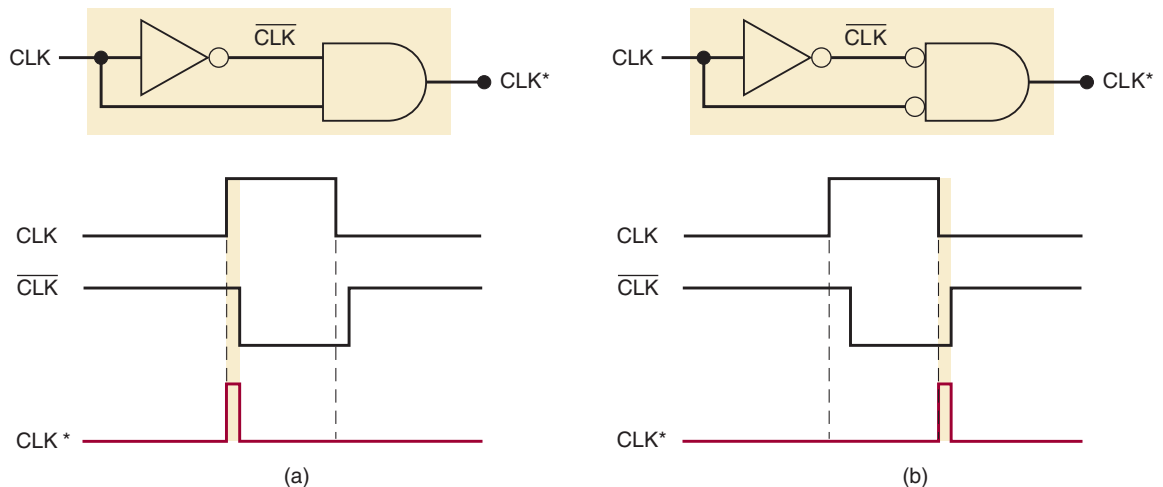
The circuit contains three sections:

1. A basic NAND gate latch formed by NAND-3 and NAND-4
2. A pulse-steering circuit formed by NAND-1 and NAND-2
3. An edge-detector circuit

As shown in Figure 5-22, the edge detector produces a narrow positive-going spike ( $CLK^*$ ) that occurs coincident with the active transition of the  $CLK$  input pulse. The pulse-steering circuit “steers” the spike through to the  $\overline{SET}$  or the  $\overline{RESET}$  input of the latch in accordance with the levels present at  $S$  and  $R$ . For example, with  $S = 1$  and  $R = 0$ , the  $CLK^*$  signal is inverted and passed through NAND-1 to produce a LOW pulse at the  $\overline{SET}$  input of the latch that sets  $Q = 1$ . With  $S = 0$ ,  $R = 1$ , the  $CLK^*$  signal is inverted and passed through NAND-2 to produce a low pulse at the  $\overline{RESET}$  input of the latch that resets  $Q = 0$ .

Figure 5-23(a) shows how the  $CLK^*$  signal is generated for edge-triggered FFs that trigger on a PGT. The INVERTER produces a delay of a few nanoseconds so that the transitions of  $\overline{CLK}$  occur a little bit after those of  $CLK$ . The AND gate produces an output spike that is HIGH only for the few nanoseconds when  $CLK$  and  $\overline{CLK}$  are both HIGH. The result is a narrow pulse at  $CLK^*$ , which occurs on the PGT of  $CLK$ . The arrangement of Figure 5-23(b) likewise produces  $CLK^*$  on the NGT of  $CLK$  for FFs that are to trigger on a NGT.

Because the  $CLK^*$  signal is HIGH for only a few nanoseconds,  $Q$  is affected by the levels at  $S$  and  $R$  only for a short time during and after the occurrence of the active edge of  $CLK$ . This is what gives the FF its edge-triggered property.



**FIGURE 5-23** Implementation of edge-detector circuits used in edge-triggered flip-flops: (a) PGT; (b) NGT. The duration of the  $CLK^*$  pulses is typically 2–5 ns.

#### OUTCOME ASSESSMENT QUESTIONS

1. Suppose that the waveforms of Figure 5-20(c) are applied to the inputs of the FF of Figure 5-21. What will happen to  $Q$  at point  $b$ ? At point  $f$ ? At point  $h$ ?
2. Explain why the  $S$  and  $R$  inputs affect  $Q$  only during the active transition of  $CLK$ .

## 5-7 CLOCKED J-K FLIP-FLOP

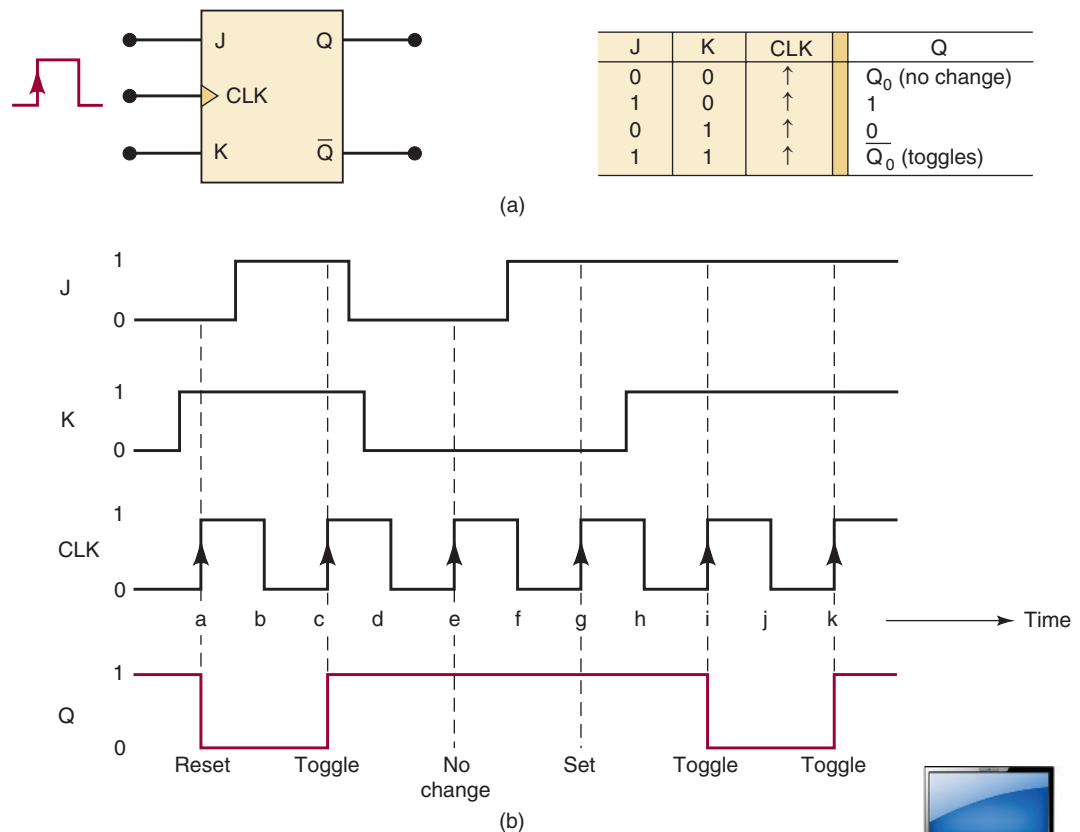
### OUTCOMES

Upon completion of this section, you will be able to:

- Identify the purpose of all J-K input combinations.
- Differentiate between a J-K and S-R FF.
- For any change in inputs,  $J$ ,  $K$ ,  $clk$ , predict the output state on  $Q$ .

Figure 5-24(a) shows a **clocked J-K flip-flop** that is triggered by the positive-going edge of the clock signal. The  $J$  and  $K$  inputs control the state of the FF in the same ways as the  $S$  and  $R$  inputs do for the clocked S-R flip-flop except for one major difference: *the  $J = K = 1$  condition does not result in an ambiguous output*. When  $J$  and  $K$  are both 1, the FF will always go to its *opposite* state upon the positive transition of the clock signal. This is called the **toggle mode** of operation. In this mode, if both  $J$  and  $K$  are left HIGH, the FF will change states (toggle) for each PGT of the clock.

The function table in Figure 5-24(a) summarizes how the J-K flip-flop responds to the PGT for each combination of  $J$  and  $K$ . Notice that the function table is the same as for the clocked S-R flip-flop (Figure 5-20) except for the  $J = K = 1$  condition. This condition results in  $Q = \bar{Q}_0$ , which means that the new value of  $Q$  will be the inverse of the value it had prior to the PGT; this is the toggle operation.



**FIGURE 5-24** (a) Clocked J-K flip-flop that responds only to the positive edge of the clock; (b) waveforms.



The operation of this FF is illustrated by the waveforms in Figure 5-24(b). Once again, we assume that the setup and hold time requirements are being met.

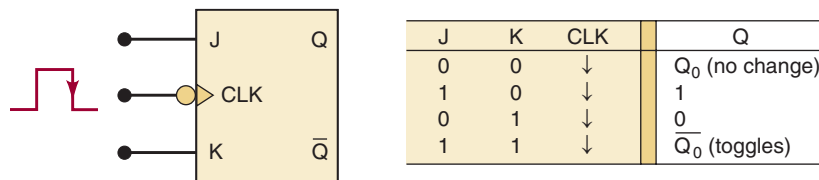
1. Initially all inputs are 0, and the  $Q$  output is assumed to be 1; that is,  $Q_0 = 1$ .
2. When the positive-going edge of the first clock pulse occurs (point  $a$ ), the  $J = 0, K = 1$  condition exists. Thus, the FF will be reset to the  $Q = 0$  state.
3. The second clock pulse finds  $J = K = 1$  when it makes its positive transition (point  $c$ ). This causes the FF to *toggle* to its opposite state,  $Q = 1$ .
4. At point  $e$  on the clock waveform,  $J$  and  $K$  are both 0, so that the FF does not change states on this transition.
5. At point  $g$ ,  $J = 1$  and  $K = 0$ . This is the condition that sets  $Q$  to the 1 state. However, it is already 1, and so it will remain there.
6. At point  $i$ ,  $J = K = 1$ , and so the FF toggles to its opposite state. The same thing occurs at point  $k$ .

Note from these waveforms that the FF is not affected by the negative-going edge of the clock pulses. Also, the  $J$  and  $K$  input levels have no effect except upon the occurrence of the PGT of the clock signal. The  $J$  and  $K$  inputs by themselves cannot cause the FF to change states.

Figure 5-25 shows the symbol for a clocked J-K flip-flop that triggers on the negative-going clock-signal transitions. The small circle on the  $CLK$  input indicates that this FF will trigger when the  $CLK$  input goes from 1 to 0. This FF operates in the same manner as the positive-edge FF of Figure 5-24 except that the output can change states only on negative-going clock-signal transitions (points  $b, d, f, h$ , and  $j$ ). Both polarities of edge-triggered J-K flip-flops are in common usage.

The J-K flip-flop is much more versatile than the S-R flip-flop because it has no ambiguous states. The  $J = K = 1$  condition, which produces the toggling operation, finds extensive use in all types of binary counters. In essence, the J-K flip-flop can do anything the S-R flip-flop can do *plus* operate in the toggle mode.

**FIGURE 5-25** J-K flip-flop that triggers only on negative-going transitions.

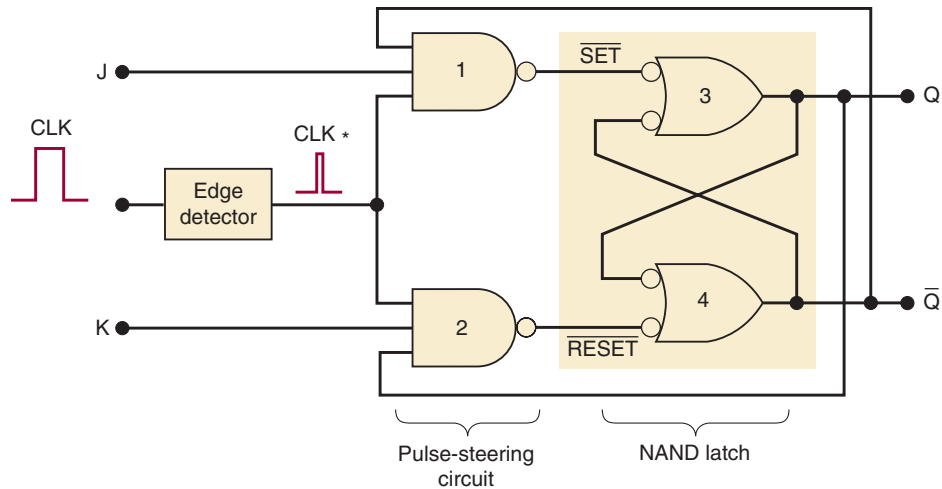


### Internal Circuitry of the Edge-Triggered J-K Flip-Flop

A simplified version of the internal circuitry of an edge-triggered J-K flip-flop is shown in Figure 5-26. It contains the same three sections as the edge-triggered S-R flip-flop (Figure 5-22). In fact, the only difference between the two circuits is that the  $Q$  and  $\overline{Q}$  outputs are fed back to the pulse-steering NAND gates. This feedback connection is what gives the J-K flip-flop its toggle operation for the  $J = K = 1$  condition.



**FIGURE 5-26** Internal circuit of the edge-triggered J-K flip-flop.



Let's examine this toggle condition more closely by assuming that  $J = K = 1$  and that  $Q$  is sitting in the LOW state when a  $CLK$  pulse occurs. With  $Q = 0$  and  $\bar{Q} = 1$ , NAND gate 1 will steer  $CLK^*$  (inverted) to the  $\overline{SET}$  input of the NAND latch to produce  $Q = 1$ . If we assume that  $Q$  is HIGH when a  $CLK$  pulse occurs, NAND gate 2 will steer  $CLK^*$  (inverted) to the  $\overline{RESET}$  input of the latch to produce  $Q = 0$ . Thus,  $Q$  always ends up in the opposite state.

In order for the toggle operation to work as described above, the  $CLK^*$  pulse must be very narrow. It must return to 0 before the  $Q$  and  $\bar{Q}$  outputs toggle to their new values; otherwise, the new values of  $Q$  and  $\bar{Q}$  will cause the  $CLK^*$  pulse to toggle the latch outputs again.

### OUTCOME ASSESSMENT QUESTIONS

1. True or false: A J-K flip-flop can be used as an S-R flip-flop, but an S-R flip-flop cannot be used as a J-K flip-flop.
2. Does a J-K flip-flop have any ambiguous input conditions?
3. What  $J$ - $K$  input condition will always set  $Q$  upon the occurrence of the active  $CLK$  transition?

## 5-8 CLOCKED D FLIP-FLOP

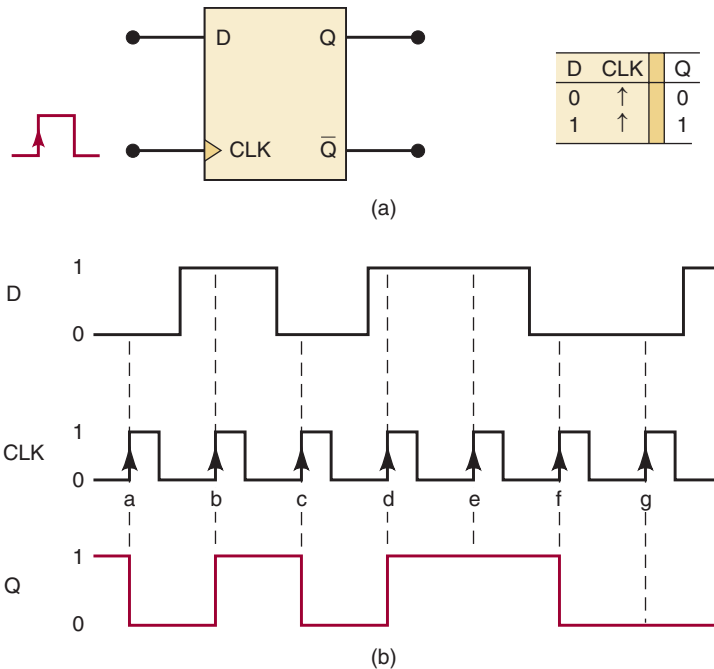
### OUTCOMES

Upon completion of this section, you will be able to:

- Predict the response of a DFF for any sequence of events on its inputs.
- Create or interpret timing diagrams that demonstrate clocked flip-flops.

Figure 5-27(a) shows the symbol and the function table for a **clocked D flip-flop** that triggers on a PGT. Unlike the S-R and J-K flip-flops, this flip-flop has only one synchronous control input,  $D$ , which stands for *data*. The operation of the D flip-flop is very simple:  $Q$  will go to the same state that is present on the  $D$  input when a PGT occurs at  $CLK$ . In other words, the level present at  $D$  will be *stored* in the flip-flop at the instant the PGT occurs. The waveforms in Figure 5-27(b) illustrate this operation.

**FIGURE 5-27** (a) D flip-flop that triggers only on positive-going transitions; (b) waveforms.



Assume that  $Q$  is initially HIGH. When the first PGT occurs at point  $a$ , the  $D$  input is LOW; thus,  $Q$  will go to the 0 state. Even though the  $D$  input level changes between points  $a$  and  $b$ , it has no effect on  $Q$ ;  $Q$  is storing the LOW that was on  $D$  at point  $a$ . When the PGT at  $b$  occurs,  $Q$  goes HIGH because  $D$  is HIGH at that time.  $Q$  stores this HIGH until the PGT at point  $c$  causes  $Q$  to go LOW because  $D$  is LOW at that time. In a similar manner, the  $Q$  output takes on the levels present at  $D$  when the PGTs occur at points  $d$ ,  $e$ ,  $f$ , and  $g$ . Note that  $Q$  stays HIGH at point  $e$  because  $D$  is still HIGH.

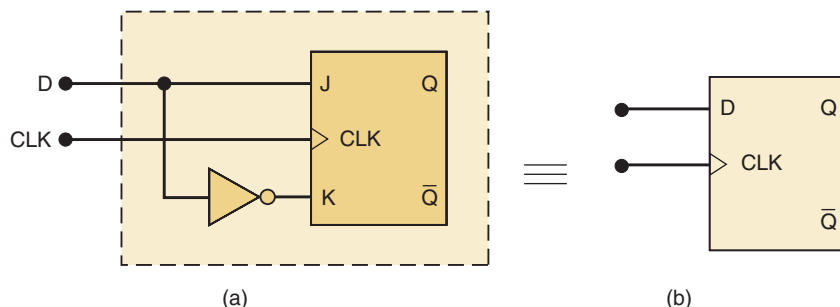
Again, it is important to remember that  $Q$  can change only when a PGT occurs. The  $D$  input has no effect between PGTs.

A negative-edge-triggered D flip-flop operates in the same way just described except that  $Q$  will take on the value of  $D$  when a NGT occurs at  $CLK$ . The symbol for the D flip-flop that triggers on NGTs will have a bubble on the  $CLK$  input.

### Implementation of the D Flip-Flop

An edge-triggered D flip-flop is easily implemented by adding a single INVERTER to the edge-triggered J-K flip-flop, as shown in Figure 5-28. If you try both values of  $D$ , you should see that  $Q$  takes on the level present at  $D$  when a PGT occurs. The same can be done to convert a S-R flip-flop to a D flip-flop.

**FIGURE 5-28** Edge-triggered D flip-flop implementation from a J-K flip-flop.

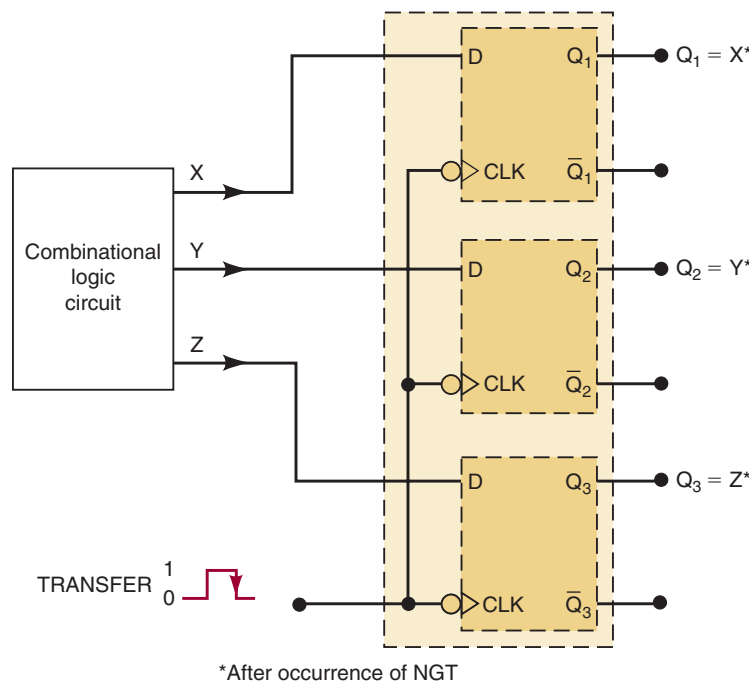


## Parallel Data Transfer

At this point you may well be wondering about the usefulness of the D flip-flop because it appears that the  $Q$  output is the same as the  $D$  input. Not quite; remember,  $Q$  takes on the value of  $D$  only at certain time instances, and so it is not identical to  $D$  (e.g., see the waveforms in Figure 5-27).

In most applications of the D flip-flop, the  $Q$  output must take on the value at its  $D$  input only at precisely defined times. One example of this is illustrated in Figure 5-29. Outputs  $X$ ,  $Y$ ,  $Z$  from a logic circuit are to be transferred to FFs  $Q_1$ ,  $Q_2$ , and  $Q_3$  for storage. Using the D flip-flops, the levels present at  $X$ ,  $Y$ , and  $Z$  will be transferred to  $Q_1$ ,  $Q_2$ , and  $Q_3$ , respectively, upon application of a TRANSFER pulse to the common  $CLK$  inputs. The FFs can store these values for subsequent processing. This is an example of **parallel data transfer** of binary data; the three bits  $X$ ,  $Y$ , and  $Z$  are all transferred *simultaneously*.

**FIGURE 5-29** Parallel transfer of binary data using D flip-flops.



### OUTCOME ASSESSMENT QUESTIONS

1. What will happen to the  $Q$  waveform in Figure 5-27(b) if the  $D$  input is held permanently LOW?
2. *True or false:* The  $Q$  output will equal the level at the  $D$  input at all times.
3. Can J-K FFs be used for parallel data transfer?

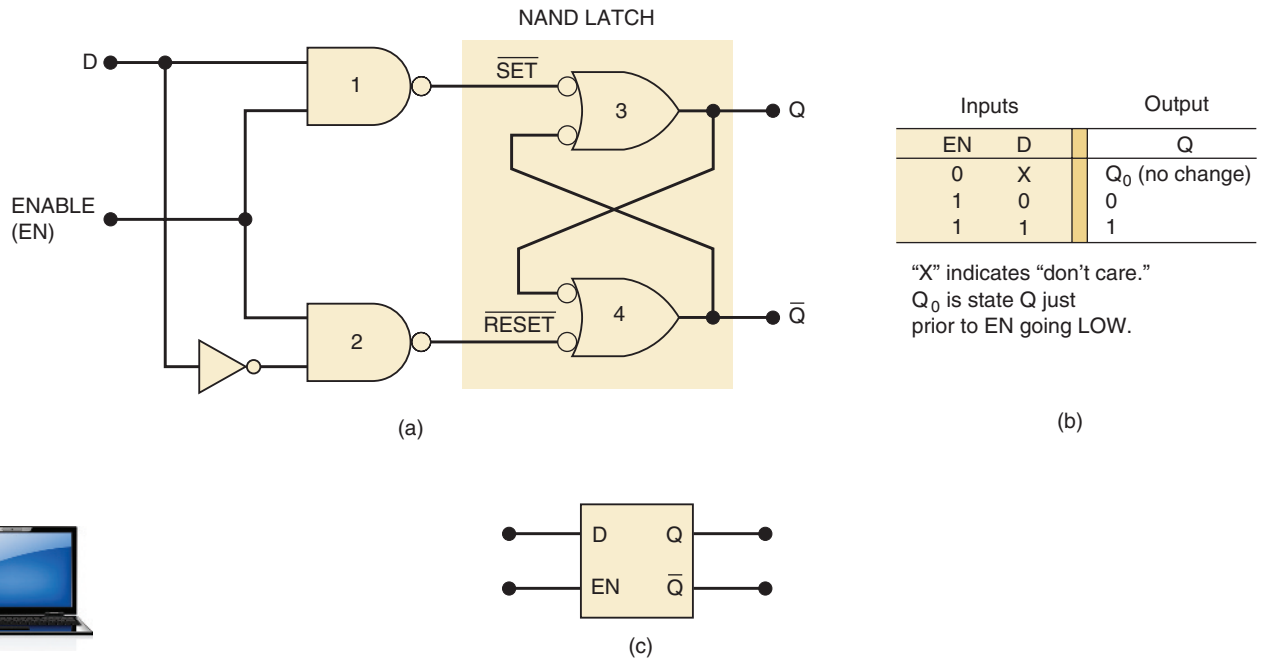
## 5-9 D LATCH (TRANSPARENT LATCH)

### OUTCOMES

Upon completion of this section, you will be able to:

- Differentiate between a D latch and a D FF operation.
- Identify the latched and transparent conditions of a D latch.
- Predict outputs for any input change in latched and transparent mode of a D latch.

The edge-triggered D flip-flop uses an edge-detector circuit to ensure that the output will respond to the  $D$  input *only* when the active transition of the clock occurs. If this edge detector is not used, the resultant circuit operates somewhat differently. It is called a **D latch** and has the arrangement shown in Figure 5-30(a).



**FIGURE 5-30** D latch: (a) structure; (b) function table; (c) logic symbol.

The circuit contains the NAND latch and the steering NAND gates 1 and 2 *without* the edge-detector circuit. The common input to the steering gates is called an *enable* input (abbreviated  $EN$ ) rather than a clock input because its effect on the  $Q$  and  $\bar{Q}$  outputs is not restricted to occurring only on its transitions. The operation of the D latch is described as follows:

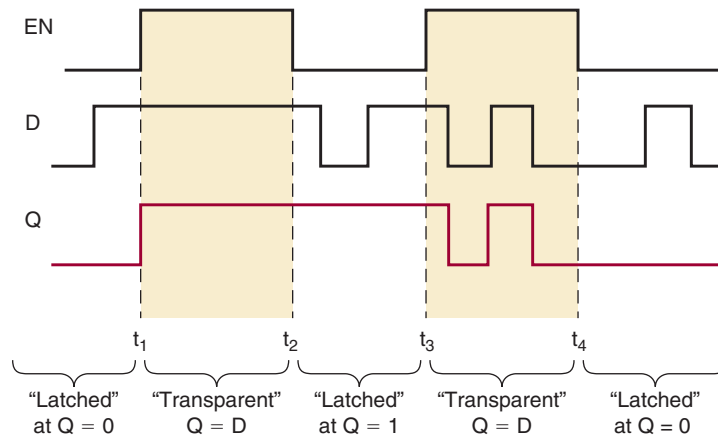
1. When  $EN$  is HIGH, the  $D$  input will produce a LOW at either the  $\overline{SET}$  or the  $\overline{RESET}$  inputs of the NAND latch to cause  $Q$  to become the same level as  $D$ . If  $D$  changes while  $EN$  is HIGH,  $Q$  will follow the changes exactly. In other words, while  $EN = 1$ , the  $Q$  output will look exactly like  $D$ ; in this mode, the D latch is said to be "transparent."
2. When  $EN$  goes LOW, the  $D$  input is inhibited from affecting the NAND latch because the outputs of both steering gates will be held HIGH. Thus, the  $Q$  and  $\bar{Q}$  outputs will stay at whatever level they had just before  $EN$  went LOW. In other words, the outputs are "latched" to their current level and cannot change while  $EN$  is LOW even if  $D$  changes.

This operation is summarized in the function table in Figure 5-30(b). The logic symbol for the D latch is given in Figure 5-30(c). Note that even though the  $EN$  input operates much like the  $CLK$  input of an edge-triggered FF, there is no small triangle on the  $EN$  input. This is because the small triangle symbol is used strictly for inputs that can cause an output change only when a transition occurs. *The D latch is not edge-triggered.*

**EXAMPLE 5-8**

Determine the  $Q$  waveform for a D latch with the  $EN$  and  $D$  inputs of Figure 5-31. Assume that  $Q = 0$  initially.

**FIGURE 5-31** Waveforms for Example 5-8 showing the two modes of operation of the transparent D latch.

**Solution**

Prior to time  $t_1$ ,  $EN$  is LOW, so that  $Q$  is “latched” at its current 0 level and cannot change even though  $D$  is changing. During the interval  $t_1$  to  $t_2$ ,  $EN$  is HIGH so that  $Q$  will follow the signal present at  $D$ . Thus,  $Q$  goes HIGH at  $t_1$  and stays there because  $D$  is not changing. When  $EN$  returns LOW at  $t_2$ ,  $Q$  will latch at the HIGH level that it has at  $t_2$  and will remain there while  $EN$  is LOW.

At  $t_3$  when  $EN$  goes HIGH again,  $Q$  will follow the changes in the  $D$  input until  $t_4$  when  $EN$  returns LOW. During the interval  $t_3$  to  $t_4$ , the D latch is “transparent” because the variations in  $D$  go through to the output  $Q$ . At  $t_4$  when  $EN$  goes LOW,  $Q$  will latch at the 0 level because that is its level at  $t_4$ . After  $t_4$  the variations in  $D$  will have no effect on  $Q$  because it is latched (i.e.,  $EN = 0$ ).

**OUTCOME ASSESSMENT QUESTIONS**

1. Describe how a D latch operates differently from an edge-triggered D flip-flop.
2. *True or false:* A D latch is in its transparent mode when  $EN = 0$ .
3. *True or false:* In a D latch, the  $D$  input can affect  $Q$  only when  $EN = 1$ .

**5-10 ASYNCHRONOUS INPUTS****OUTCOMES**

Upon completion of this section, you will be able to:

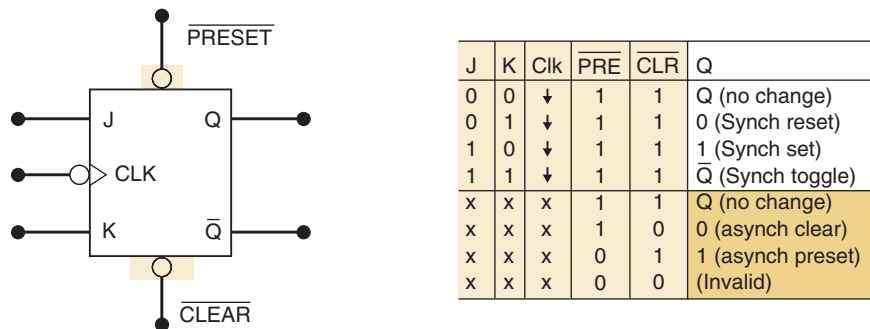
- Distinguish between synchronous and asynchronous inputs.
- For any change on any synchronous or asynchronous input, predict the output state of  $Q$ .

For the clocked flip-flops that we have been studying, the  $S$ ,  $R$ ,  $J$ ,  $K$ , and  $D$  inputs have been referred to as *control* inputs. These inputs are also called synchronous inputs because their effect on the FF output is synchronized with the  $CLK$  input. As we have seen, the synchronous control inputs must be used in conjunction with a clock signal to trigger the FF.

Most clocked FFs also have one or more **asynchronous inputs** that operate independently of the synchronous inputs and clock input. These asynchronous inputs can be used to set the FF to the 1 state or clear (reset) the FF to the 0 state *at any time, regardless of the conditions at the other inputs*. Stated in another way, the asynchronous inputs are **override inputs**, which can be used to override all the other inputs in order to place the FF in one state or the other.

Figure 5-32 shows a J-K flip-flop with two asynchronous inputs designated as  $\overline{PRESET}$  and  $\overline{CLEAR}$ . These are active-LOW inputs, as indicated by the bubbles on the FF symbol. The accompanying function table summarizes how they affect the FF output. Let's examine the various cases.

**FIGURE 5-32** Clocked J-K flip-flop with asynchronous inputs.



- $\overline{PRESET} = \overline{CLEAR} = 1$ . The asynchronous inputs are inactive and the FF is free to respond to the  $J$ ,  $K$ , and  $CLK$  inputs; in other words, the clocked operation can take place.
- $\overline{PRESET} = 0$ ;  $\overline{CLEAR} = 1$ . The  $\overline{PRESET}$  is activated and  $Q$  is *immediately* set to 1 no matter what conditions are present at the  $J$ ,  $K$ , and  $CLK$  inputs. The  $CLK$  input cannot affect the FF while  $\overline{PRESET} = 0$ .
- $\overline{PRESET} = 1$ ;  $\overline{CLEAR} = 0$ . The  $\overline{CLEAR}$  is activated and  $Q$  is *immediately* cleared to 0 independent of the conditions on the  $J$ ,  $K$ , or  $CLK$  inputs. The  $CLK$  input has no effect while  $\overline{CLEAR} = 0$ .
- $\overline{PRESET} = \overline{CLEAR} = 0$ . This condition should not be used because it can result in an ambiguous response.

It is important to realize that these asynchronous inputs respond to dc levels. This means that if a constant 0 is held on the  $\overline{PRESET}$  input, the FF will remain in the  $Q = 1$  state regardless of what is occurring at the other inputs. Similarly, a constant LOW on the  $\overline{CLEAR}$  input holds the FF in the  $Q = 0$  state. Thus, the asynchronous inputs can be used to hold the FF in a particular state for any desired interval. Most often, however, the asynchronous inputs are used to set or clear the FF to the desired state by application of a momentary pulse.

Many clocked FFs that are available as ICs will have both of these asynchronous inputs; some will have only the  $\overline{CLEAR}$  input. Some FFs will have asynchronous inputs that are active-HIGH rather than active-LOW. For these FFs the FF symbol would not have a bubble on the asynchronous inputs.

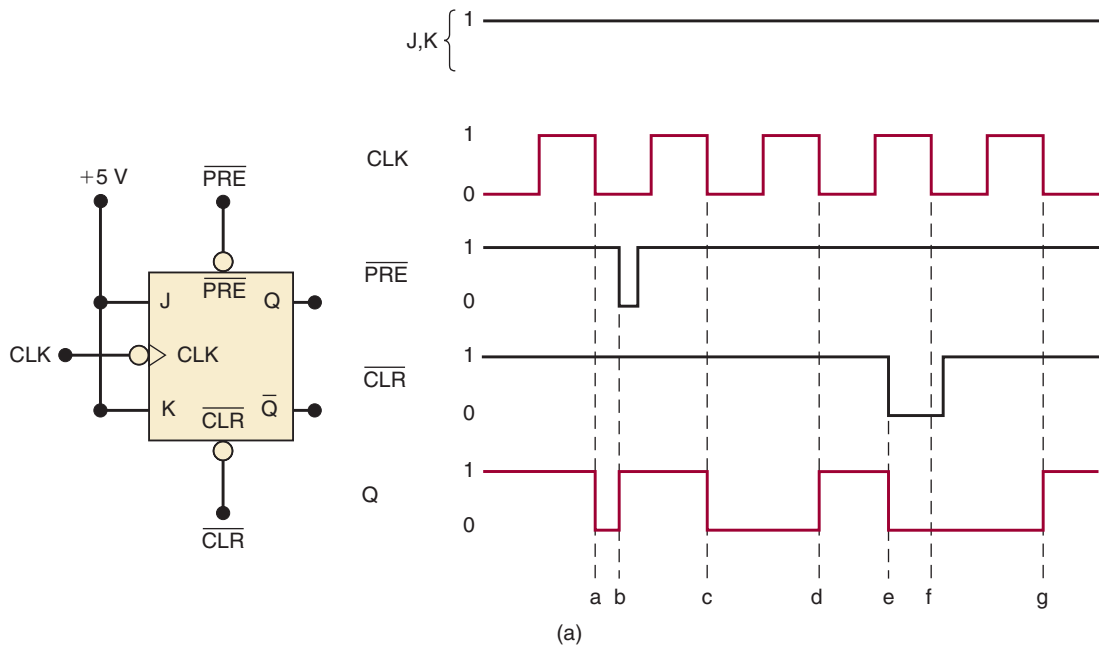
### Designations for Asynchronous Inputs

IC manufacturers do not all agree on the nomenclature to use for these asynchronous inputs. The most common designations are *PRE* (short for PRESET) and *CLR* (short for CLEAR). These labels clearly distinguish them from the synchronous SET and RESET inputs. Other labels such as *S<sub>D</sub>* (direct SET) and *R<sub>D</sub>* (direct RESET) are also used. From now on, we will use the labels *PRE* and *CLR* to represent the asynchronous inputs because these seem to be the most commonly used labels. When these asynchronous inputs are active-LOW, as they generally are, we will use the overbar to indicate their active-LOW status, that is,  $\overline{PRE}$  and  $\overline{CLR}$ .

Although most IC flip-flops have at least one or more asynchronous inputs, there are some circuit applications where they are not used. In such cases they are held permanently at their inactive level. Often, in our use of FFs throughout the remainder of the text, we will not show a FF's unused asynchronous inputs; it will be assumed that they are permanently connected to their inactive logic level.

**EXAMPLE 5-9**

Figure 5-33(a) shows the symbol for a J-K FF that responds to a NGT on its clock input and has active-LOW asynchronous inputs. The external active-LOW asynchronous inputs are labeled  $\overline{PRE}$  and  $\overline{CLR}$ . The bubble on an input means that the input responds to a logic LOW signal.



Point	Operation
a	Synchronous toggle on NGT of CLK
b	Asynchronous set on $\overline{PRE} = 0$
c	Synchronous toggle
d	Synchronous toggle
e	Asynchronous clear on $\overline{CLR} = 0$
f	$\overline{CLR}$ overrides the NGT of CLK
g	Synchronous toggle

(b)



**FIGURE 5-33** Waveforms for Example 5-9 showing how a clocked flip-flop responds to asynchronous inputs.

The  $J$  and  $K$  inputs are shown tied HIGH for this example. Determine the  $Q$  output in response to the input waveforms shown in Figure 5-33(a). Assume that  $Q$  is initially HIGH.

### Solution

Initially,  $\overline{PRE}$  and  $\overline{CLR}$  are in their inactive HIGH state, so that they will have no effect on  $Q$ . Thus, when the first NGT of the  $CLK$  signal occurs at point  $a$ ,  $Q$  will toggle to its opposite state; remember,  $J = K = 1$  produces the toggle operation.

At point  $b$ , the  $\overline{PRE}$  input is pulsed to its active-LOW state. This will *immediately* set  $Q = 1$ . Note that  $\overline{PRE}$  produces  $Q = 1$  without waiting for a NGT at  $CLK$ . The asynchronous inputs operate independently of  $CLK$ .

At point  $c$ , the NGT of  $CLK$  will again cause  $Q$  to toggle to its opposite state. Note that  $\overline{PRE}$  has returned to its inactive state prior to point  $c$ . Likewise, the NGT of  $CLK$  at point  $d$  will toggle  $Q$  back HIGH.

At point  $e$ , the  $\overline{CLR}$  input is pulsed to its active-LOW state and will *immediately* clear  $Q = 0$ . Again, it does this independently of  $CLK$ .

The NGT of  $CLK$  at point  $f$  will *not* toggle  $Q$  because the  $\overline{CLR}$  input is still active. The LOW at  $\overline{CLR}$  overrides the  $CLK$  input and holds  $Q = 0$ .

When the NGT of  $CLK$  occurs at point  $g$ , it will toggle  $Q$  to the HIGH state because neither asynchronous input is active at that point.

These steps are summarized in Figure 5-33(b).

### OUTCOME ASSESSMENT QUESTIONS

1. How does the operation of an asynchronous input differ from that of a synchronous input?
2. Can a D flip-flop respond to its  $D$  and  $CLK$  inputs while  $\overline{PRE} = 1$ ?
3. List the conditions necessary for a positive-edge-triggered J-K flip-flop with active-LOW asynchronous inputs to toggle to its opposite state.

## 5-11 FLIP-FLOP TIMING CONSIDERATIONS

### OUTCOMES

Upon completion of this section, you will be able to:

- Define parameters of flip-flops that limit speed of operation and reliability.
- Determine whether system signals are within the reliable operating limits of any given flip-flop.

Manufacturers of IC flip-flops will specify several important timing parameters and characteristics that must be considered before a FF is used in any circuit application. We will describe the most important of these and then give some actual examples of specific IC flip-flops from the TTL and CMOS logic families.

### Setup and Hold Times

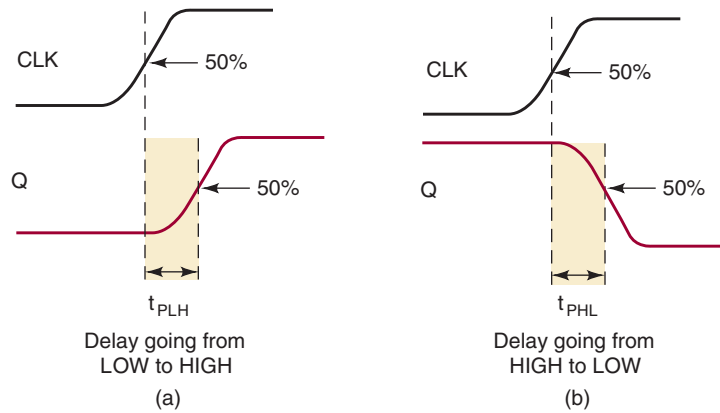
The setup and hold times have already been discussed, and you may recall from Section 5-5 that they represent requirements that must be met for reliable FF triggering. The manufacturer's IC data sheet will always specify the *minimum* values of  $t_S$  and  $t_H$ .



## Propagation Delays

Whenever a signal is to change the state of a FF's output, there is a delay from the time the signal is applied to the time when the output makes its change. Figure 5-34 illustrates the **propagation delays** that occur in response to a positive transition on the *CLK* input. Note that these delays are measured between the 50 percent points on the input and output waveforms. The same types of delays occur in response to signals on a FF's asynchronous inputs (PRESET and CLEAR). The manufacturers' data sheets usually specify propagation delays in response to all inputs, and they usually specify the *maximum* values for  $t_{PLH}$  and  $t_{PHL}$ .

**FIGURE 5-34** FF propagation delays.



Modern IC flip-flops have propagation delays that range from a few nanoseconds to around 100 ns. The values of  $t_{PLH}$  and  $t_{PHL}$  may not be the same, and they increase in direct proportion to the number of loads being driven by the *Q* output. FF propagation delays play an important part in certain situations that we will encounter later.

## Maximum Clocking Frequency, $f_{MAX}$

This is the highest frequency that may be applied to the *CLK* input of a FF and still have it trigger reliably. The  $f_{MAX}$  limit will vary from FF to FF, even with FFs having the same device number. For example, the manufacturer of the 7470 J-K flip-flop IC tests many of these FFs and may find that the  $f_{MAX}$  values fall in the range of 20 to 35 MHz. The maximum frequency will then be specified as a *minimum* of 20 MHz. This may seem confusing, but a little thought should make it clear that what the manufacturer is saying is that there is no guarantee that the 7470 flip-flops that you put in your circuit will work above 20 MHz; most of them will, but some of them will not. If they are operated below 20 MHz, however, the manufacturer guarantees that they will all work.

## Clock Pulse HIGH and LOW Times

Manufacturers will also specify the *minimum* time duration that the *CLK* signal must remain LOW before it goes HIGH, sometimes called  $t_{W(L)}$ , and the *minimum* time that *CLK* must be kept HIGH before it returns LOW, sometimes called  $t_{W(H)}$ . These times are defined in Figure 5-35(a). Failure to meet these minimum time requirements can result in unreliable triggering. Note that these time values are measured between the halfway points on the signal transitions.

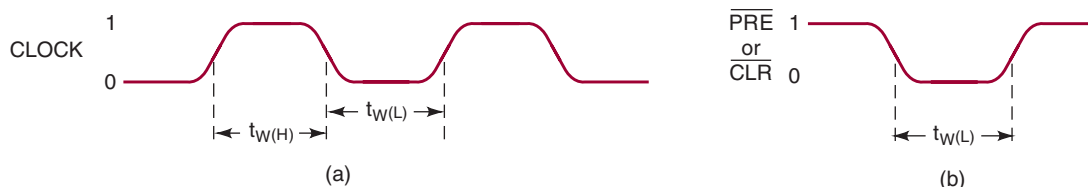


FIGURE 5-35 (a) Clock LOW and HIGH times; (b) asynchronous pulse width.

### Asynchronous Active Pulse Width

Manufacturers will also specify the *minimum* time duration that a PRESET or CLEAR input must be kept in its active state in order to set or clear the FF reliably. Figure 5-35(b) shows  $t_{W(L)}$  for active-LOW asynchronous inputs.

### Clock Transition Times

For reliable triggering, the clock waveform transition times (rise and fall times) should be kept very short. If the clock signal takes too long to make the transitions from one level to the other, the FF may trigger erratically or not at all. Manufacturers usually do not list a maximum transition time requirement for each FF integrated circuit. Instead, it is usually given as a general requirement for all ICs within a given logic family. For example, the transition times should generally be  $\leq 50$  ns for TTL devices and  $\leq 200$  ns for CMOS. These requirements will vary among the different manufacturers and among the various subfamilies within the broad TTL and CMOS logic families.

#### OUTCOME ASSESSMENT QUESTIONS

1. Which FF timing parameters indicate the time it takes the  $Q$  output to respond to an input?
2. *True or false:* A FF that has an  $f_{\text{MAX}}$  rating of 25 MHz can be reliably triggered by any  $CLK$  pulse waveform with a frequency below 25 MHz.
3. The minimum amount of time that a control input must be stable before the clock edge is called \_\_\_\_\_.
4. The minimum amount of time a control input must remain stable after a clock edge is called \_\_\_\_\_.
5. *True or false:* Clock waveforms with too great of a rise time or fall time may not trigger the flip-flop reliably.

## 5-12 POTENTIAL TIMING PROBLEM IN FF CIRCUITS

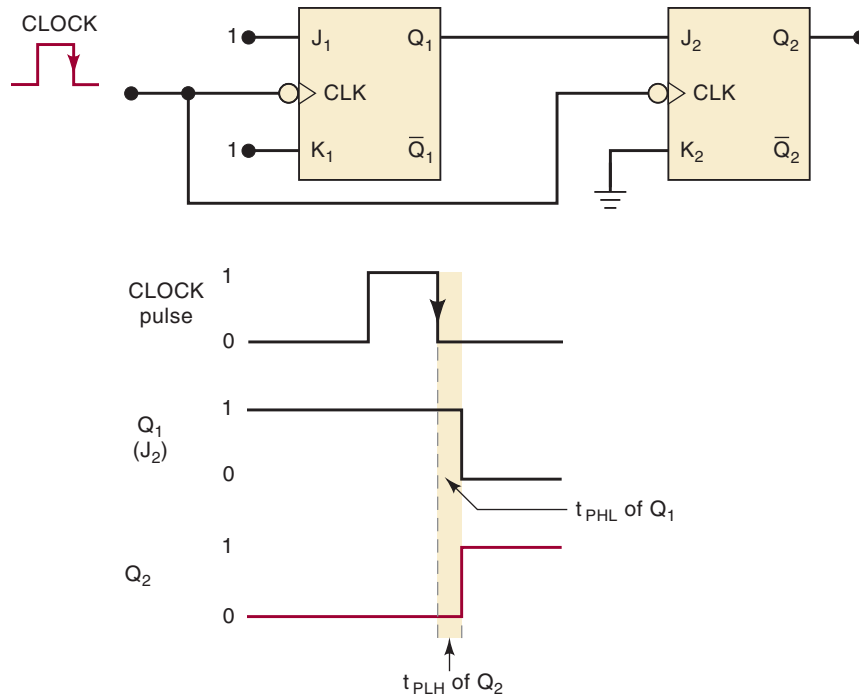
### OUTCOMES

Upon completion of this section, you will be able to:

- Predict the  $Q$  output by looking at timing waveforms of the control inputs.
- Apply timing limitations to synchronous circuits.

In many digital circuits, the output of one FF is connected either directly or through logic gates to the input of another FF, and both FFs are triggered by the same clock signal. This presents a potential timing problem.

**FIGURE 5-36**  $Q_2$  will respond properly to the level present at  $Q_1$  prior to the NGT of  $CLK$ , provided that  $Q_2$ 's hold time requirement,  $t_H$ , is less than  $Q_1$ 's propagation delay.



A typical situation is illustrated in Figure 5-36, where the output of  $Q_1$  is connected to the  $J$  input of  $Q_2$  and both FFs are clocked by the same signal at their  $CLK$  inputs.

The potential timing problem is this: because  $Q_1$  will change on the NGT of the clock pulse, the  $J_2$  input of  $Q_2$  will be changing as it receives the same NGT. This could lead to an unpredictable response at  $Q_2$ .

Let's assume that initially  $Q_1 = 1$  and  $Q_2 = 0$ . Thus, the  $Q_1$  FF has  $J_1 = K_1 = 1$ , and  $Q_2$  has  $J_2 = Q_1 = 1$ ,  $K_2 = 0$  prior to the NGT of the clock pulse. When the NGT occurs,  $Q_1$  will toggle to the LOW state, but it will not actually go LOW until after its propagation delay,  $t_{PHL}$ . The same NGT will reliably clock  $Q_2$  to the HIGH state provided that  $t_{PHL}$  is greater than  $Q_2$ 's hold time requirement,  $t_H$ . If this condition is not met, the response of  $Q_2$  will be unpredictable.

Fortunately, edge-triggered FFs have hold time requirements that are 5 ns or less; most have  $t_H = 0$ , which means that they have no hold time requirement. For these FFs, situations like that in Figure 5-36 will not be a problem.

Unless stated otherwise, in all of the FF circuits that we encounter throughout the text, we will assume that the FF's hold time requirement is short enough to respond reliably according to the following rule:

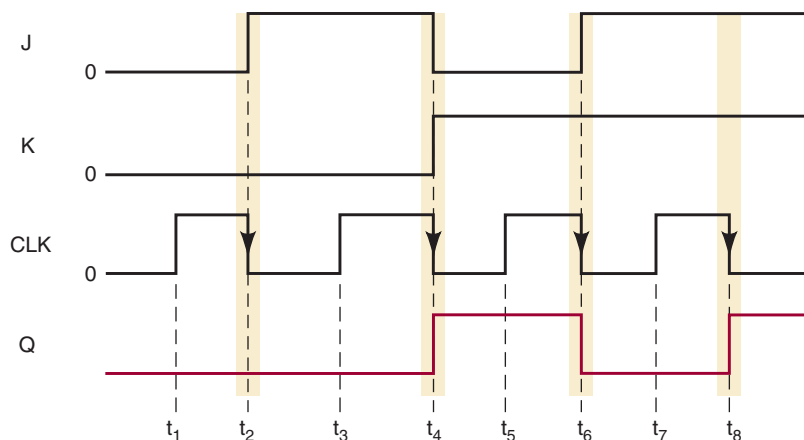
**The FF output will go to a state determined by the logic levels present at its synchronous control inputs just prior to the active clock transition.**

If we apply this rule to Figure 5-36, it says that  $Q_2$  will go to a state determined by the  $J_2 = 1$ ,  $K_2 = 0$  condition that is present just prior to the NGT of the clock pulse. The fact that  $J_2$  is changing in response to the same NGT has no effect.

## EXAMPLE 5-10

Determine the  $Q$  output for a negative-edge-triggered J-K flip-flop for the input waveforms shown in Figure 5-37. Assume that  $t_H = 0$  and that  $Q = 0$  initially.

FIGURE 5-37 Example 5-10.



## Solution

The FF will respond only at times  $t_2$ ,  $t_4$ ,  $t_6$ , and  $t_8$ . At  $t_2$ ,  $Q$  will respond to the  $J = K = 0$  condition present just prior to  $t_2$ . At  $t_4$ ,  $Q$  will respond to the  $J = 1$ ,  $K = 0$  condition present just prior to  $t_4$ . At  $t_6$ ,  $Q$  will respond to the  $J = 0$ ,  $K = 1$  condition present just prior to  $t_6$ . At  $t_8$ ,  $Q$  responds to  $J = K = 1$ .

## OUTCOME ASSESSMENT QUESTIONS

1. *True or false:* Synchronous circuits always violate set and hold times because the outputs change on the active clock edge.
2. *True or false:* The propagation delay time must exceed the hold time requirements in order for synchronous flip-flop circuit to work reliably.
3. To analyze a synchronous circuit, where can the control inputs be found that will determine the output changes?

## 5-13 FLIP-FLOP APPLICATIONS

Edge-triggered (clocked) flip-flops are versatile devices that can be used in a wide variety of applications including counting, storing of binary data, transferring binary data from one location to another, and many more. Almost all of these applications utilize the FF's clocked operation. Many of them fall into the category of **sequential circuits**. A sequential circuit is one in which the outputs follow a predetermined sequence of states, with a new state occurring each time a clock pulse occurs. Again the concept of feedback is applied, but not just to create the FF memory elements themselves. The outputs of the FFs will also generally be fed back to gates in the sequential circuit that control the operation of the FFs and, thereby, determine the new state that will occur on the next clock pulse. We will introduce some of the basic applications in the following sections, and we will expand on them in subsequent chapters.

## 5-14 FLIP-FLOP SYNCHRONIZATION

### OUTCOMES

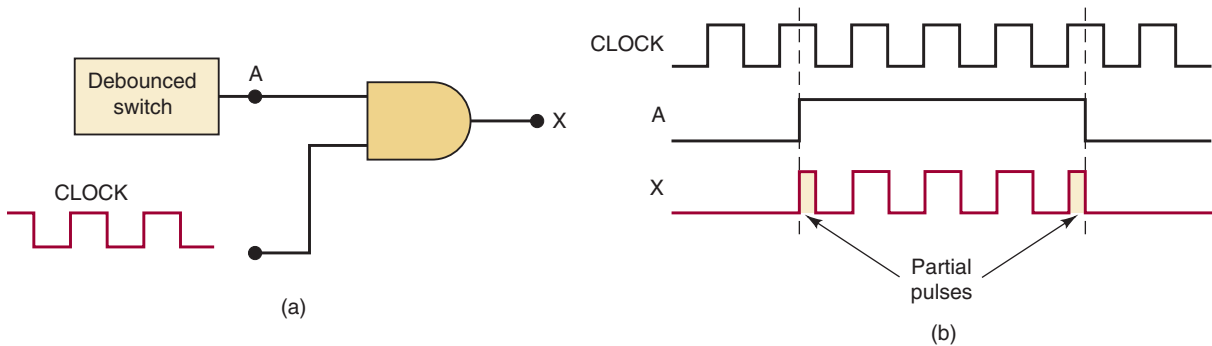
Upon completion of this section, you will be able to:

- Identify situations when synchronization is desirable.
- Synchronize external signals to the clock using a DFF.

Most digital systems are principally synchronous in their operation because most of the signals will change states in synchronism with the clock transitions. In many cases, however, there will be an external signal that is not synchronized to the clock; in other words, it is asynchronous. For example, asynchronous signals often occur as a result of a human operator's actuating an input switch at some random time relative to the clock signal. Inputs from machines and communications signals are also examples of randomly occurring transitions. This randomness can produce unpredictable and undesirable results. The following example illustrates how a FF can be used to synchronize the effect of an asynchronous input.

#### EXAMPLE 5-11

Figure 5-38(a) shows a situation where input signal  $A$  is generated from a debounced switch that is actuated by an operator (a debounced switch was first introduced in Example 5-2).  $A$  goes HIGH when the operator actuates the switch and goes LOW when the operator releases the switch. This  $A$  input is used to control the passage of the clock signal through the AND gate so that clock pulses appear at output  $X$  only as long as  $A$  is HIGH.

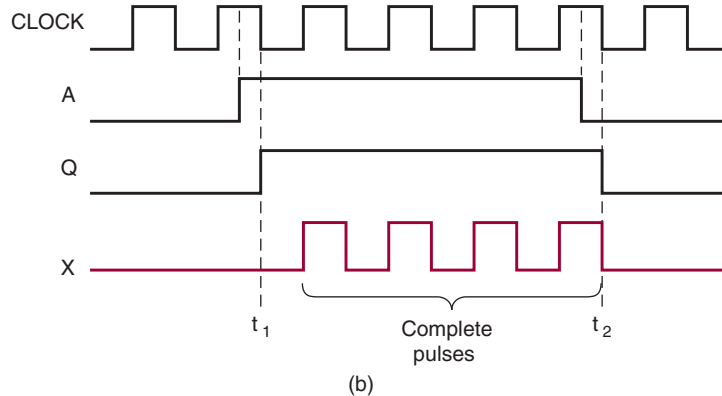
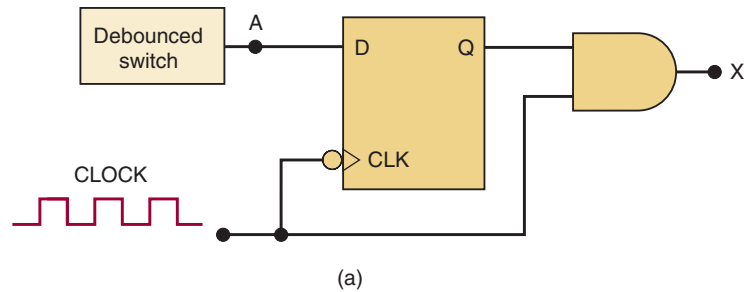


**FIGURE 5-38** Asynchronous signal  $A$  can produce partial pulses at  $X$ .

The problem with this circuit is that  $A$  is asynchronous; it can change states at any time relative to the clock signal because the exact times when the operator actuates or releases the switch are essentially random. This can produce *partial* clock pulses at output  $X$  if either transition of  $A$  occurs while the clock signal is HIGH, as shown in the waveforms of Figure 5-38(b).

This type of output is often not acceptable, so a method for preventing the appearance of partial pulses at  $X$  must be developed. One solution is shown in Figure 5-39(a). Describe how this circuit solves the problem, and draw the  $X$  waveform for the same situation as in Figure 5-38(b).

**FIGURE 5-39** An edge-triggered D flip-flop is used to synchronize the enabling of the AND gate to the NGTs of the clock.



### Solution

The  $A$  signal is connected to the  $D$  input of FF  $Q$ , which is clocked by the NGT of the clock signal. Thus, when  $A$  goes HIGH,  $Q$  will not go HIGH until the next NGT of the clock at time  $t_1$ . This HIGH at  $Q$  will enable the AND gate to pass subsequent *complete* clock pulses to  $X$ , as shown in Figure 5-39(b).

When  $A$  returns LOW,  $Q$  will not go LOW until the next NGT of the clock at  $t_2$ . Thus, the AND gate will not inhibit clock pulses until the clock pulse that ends at  $t_2$  has been passed through to  $X$ . Therefore, output  $X$  contains only complete pulses.

There is a potential problem with this circuit. Since  $A$  could go HIGH at any moment, it may by random chance violate the setup time requirement of the flip-flop. In other words, the transition of  $A$  may occur so close to the clock edge that it causes an unstable response (glitch) from the  $Q$  output. Preventing this would require a more complex synchronizing circuit.

### OUTCOME ASSESSMENT QUESTIONS

1. When should synchronization circuits be used?
2. How are DFFs used to synchronize signals?

## 5-15 DETECTING AN INPUT SEQUENCE

### OUTCOME

Upon completion of this section, you will be able to:

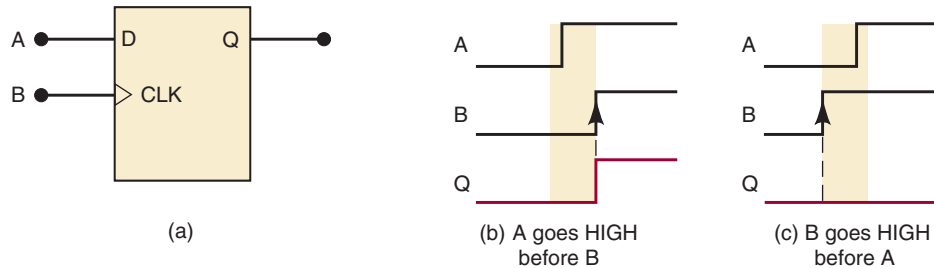
- Use a DFF to detect which of two signals changed first.

In many situations, an output is to be activated only when the inputs are activated in a certain sequence. This cannot be accomplished using pure combinational logic but requires the storage characteristic of FFs.

For example, an AND gate can be used to determine when two inputs *A* and *B* are both HIGH, but its output will respond the same regardless of which input goes HIGH first. But suppose that we want to generate a HIGH output *only* if *A* goes HIGH and then *B* goes HIGH some time later. One way to accomplish this is shown in Figure 5-40(a).

The waveforms in Figure 5-40(b) and (c) show that *Q* will go HIGH only if *A* goes HIGH before *B* goes HIGH. This is because *A* must be HIGH in order for *Q* to go HIGH on the PGT of *B*.

**FIGURE 5-40** Clocked D flip-flop used to respond to a particular sequence of inputs.



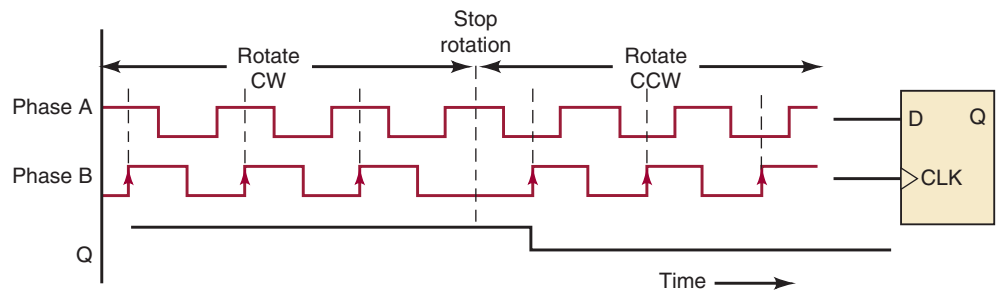
In order for this circuit to work properly, *A* must go HIGH prior to *B* by at least an amount of time equal to the setup time requirement of the FF.

**EXAMPLE 5-12**

Review the material in Section 2-5 regarding quadrature shaft encoder signals. Recall that a quadrature shaft encoder produces two output signals that are shifted 90 degrees. If the shaft rotates one direction, output *A* will lead output *B*. If it rotates in the other direction, output *A* will lag output *B*. Use a DFF to determine which direction the shaft is rotating.

**Solution**

Refer to Figure 5-41. The two quadrature outputs are connected as shown: input *A* to *D* and input *B* to *clk*. The first half of the timing diagram shows the quadrature signals when rotating clockwise. The second half of the timing diagram shows the signals when rotating counterclockwise.



**FIGURE 5-41** The sequence of quadrature encoder signals determines direction of rotation. The DFF detects this sequence.

**OUTCOME  
ASSESSMENT  
QUESTION**

1. How does a DFF identify which input changes first?

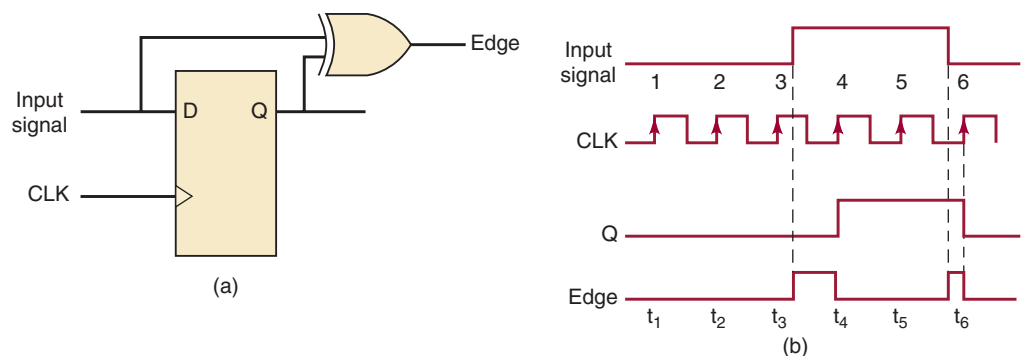
## 5-16 DETECTING A TRANSITION OR “EVENT”

### OUTCOMES

Upon completion of this section, you will be able to:

- Use a DFF and XOR gate to detect an event.
- Predict the output of the event detector for any given input signals.

As you recall we have used a number of terms (e.g., transition, edge, event) to describe a change in logic state, whether it is a change from LOW to HIGH or a change from HIGH to LOW. It is often very important for digital circuits to respond to the change when it happens. In synchronous, clocked circuits the updates always happen on a clock edge so a circuit is needed that will output a HIGH whenever its input (the signal in whose change we are interested) has changed state since the last clock edge. A D flip-flop can be used for this purpose by comparing the logic level on its  $D$  input with the logic level on its  $Q$  output. The simple circuit in Figure 5-42 demonstrates this application. At time  $t_1$ ,  $t_2$ , and  $t_3$ , the signal is LOW and there has been no change in the input since the previous clock edge. Between  $t_3$ , and  $t_4$ , the input signal changes state but the  $Q$  output of the flip-flop is still storing a LOW (the value present the last time it was clocked). The XOR gate detects that its inputs are different from each other, indicating a change in the input signal since the last clock edge. At  $t_4$ , the  $Q$  output updates to a HIGH, so the input and output are once again the same. It is important to notice that the width of the output pulse named “edge” depends on how much time there is between the input signal transition and the clock edge. This may produce very narrow output pulses as shown at  $t_6$ .



**FIGURE 5-42** Detecting an edge: (a) circuit (b) timing example.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is the purpose of the XOR gate in the event detector circuit?
2. Assume CLK is 1 MHz, 50% duty cycle. If the input signal changes state on the falling edge of CLK, what will the output *EDGE* signal look like?



## 5-17 DATA STORAGE AND TRANSFER

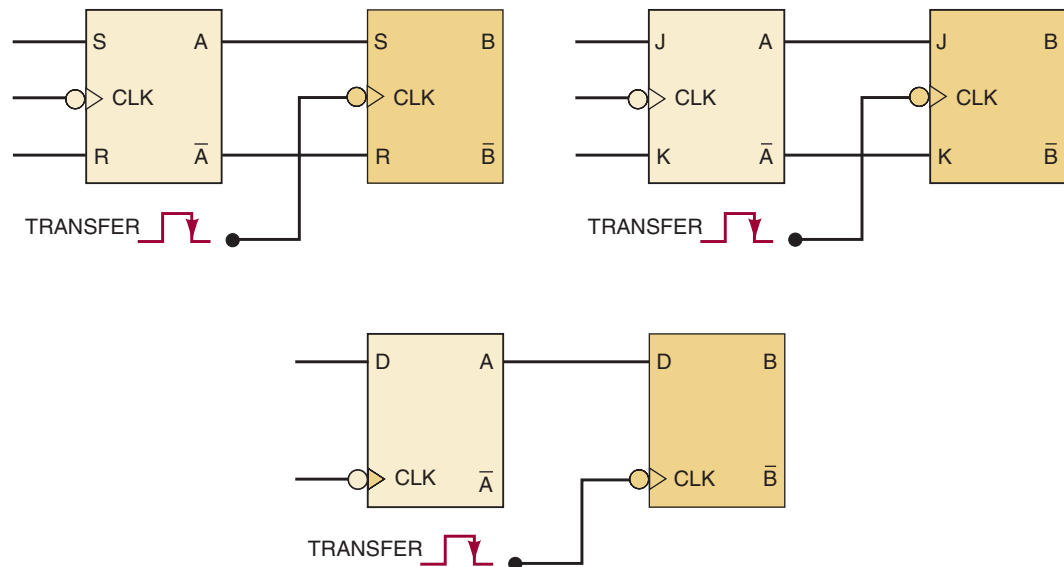
### OUTCOMES

Upon completion of this section, you will be able to:

- Use flip-flops to transfer data from one location to another.
- Distinguish between synchronous and asynchronous transfer methods.

By far the most common use of flip-flops is for the storage of data or information. The data may represent numerical values (e.g., binary numbers, BCD-coded decimal numbers) or any of a wide variety of types of data that have been encoded in binary. These data are generally stored in groups of FFs called **registers**.

The operation most often performed on data that are stored in a FF or a register is the **data transfer** operation. This involves the transfer of data from one FF or register to another. Figure 5-43 illustrates how data transfer can be accomplished between two FFs using clocked S-R, J-K, and D flip-flops. In each case, the logic value that is currently stored in FF A is transferred to FF B upon the NGT of the TRANSFER pulse. Thus, after this NGT, the B output will be the same as the A output.

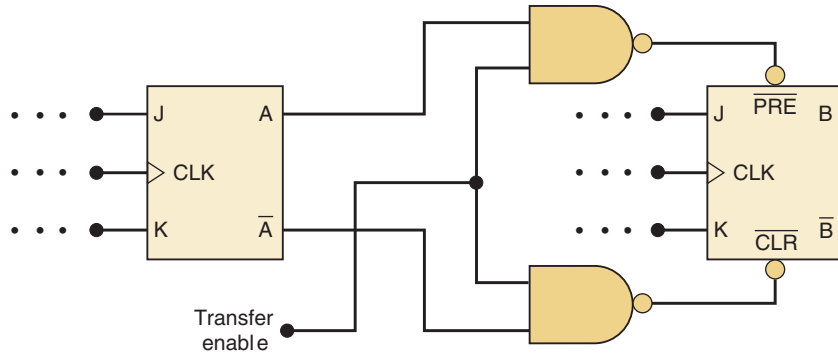


**FIGURE 5-43** Synchronous data transfer operation performed by various types of clocked FFs.

The transfer operations in Figure 5-43 are examples of **synchronous transfer** because the synchronous control and *CLK* inputs are used to perform the transfer. A transfer operation can also be obtained using the asynchronous inputs of a FF. Figure 5-44 shows how an **asynchronous transfer** can be accomplished using the PRESET and CLEAR inputs of any type of FF. Here, the asynchronous inputs respond to LOW levels. When the TRANSFER ENABLE line is held LOW, the two NAND outputs are kept HIGH, with no effect on the FF outputs. When the TRANSFER ENABLE line is made HIGH, one of the NAND outputs will go LOW, depending on the state of the A and  $\bar{A}$  outputs. This LOW will either set or clear FF B to the same state as FF A. This asynchronous transfer is done independently

**FIGURE 5-44**

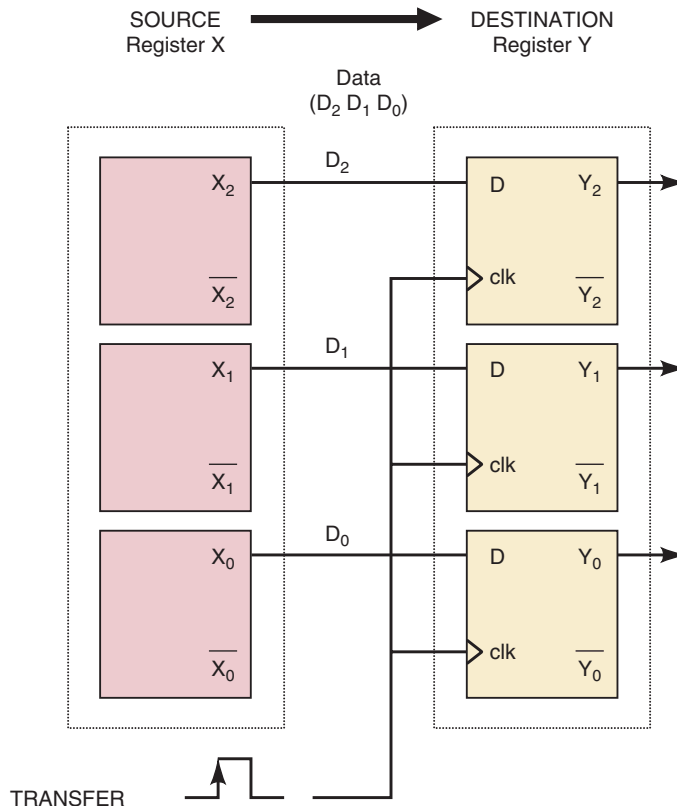
Asynchronous data transfer operation.



of the synchronous and  $CLK$  inputs of the FF. Asynchronous transfer is also called **jam transfer** because the data can be “jammed” into FF B even if its synchronous inputs are active.

### Parallel Data Transfer

Figure 5-45 illustrates data transfer from one register to another using D-type FFs. Register X consists of FFs  $X_2$ ,  $X_1$ , and  $X_0$ ; register Y consists of FFs  $Y_2$ ,  $Y_1$ , and  $Y_0$ . Upon application of the PGT of the TRANSFER pulse, the level stored in  $X_2$  is transferred to  $Y_2$ ,  $X_1$  to  $Y_1$ , and  $X_0$  to  $Y_0$ . The transfer of the contents of the X register into the Y register is a synchronous transfer. It is also referred to as a parallel transfer because the contents of  $X_2$ ,  $X_1$ , and  $X_0$  are transferred *simultaneously* into  $Y_2$ ,  $Y_1$ , and  $Y_0$ , respectively. If a **serial data transfer** were performed, the contents of the X register would be transferred to the Y register one bit at a time. This will be examined in the next section.

**FIGURE 5-45** Parallel transfer of contents of register X into register Y.

It is important to understand that parallel transfer does not change the contents of the register that is the source of data. For example, in Figure 5-45, if  $X_2X_1X_0 = 101$  and  $Y_2Y_1Y_0 = 011$  prior to the occurrence of the TRANSFER pulse, then both registers will be holding 101 after the TRANSFER pulse.

### OUTCOME ASSESSMENT QUESTIONS

1. *True or false:* Asynchronous data transfer uses the *CLK* input.
2. Which type of FF is best suited for synchronous transfer because it requires the fewest interconnections from one FF to the other?
3. If J-K flip-flops were used in the registers of Figure 5-45, how many total interconnections would be required from register *X* to register *Y*?
4. *True or false:* Synchronous data transfer requires less circuitry than asynchronous transfer.

## 5-18 SERIAL DATA TRANSFER: SHIFT REGISTERS

### OUTCOMES

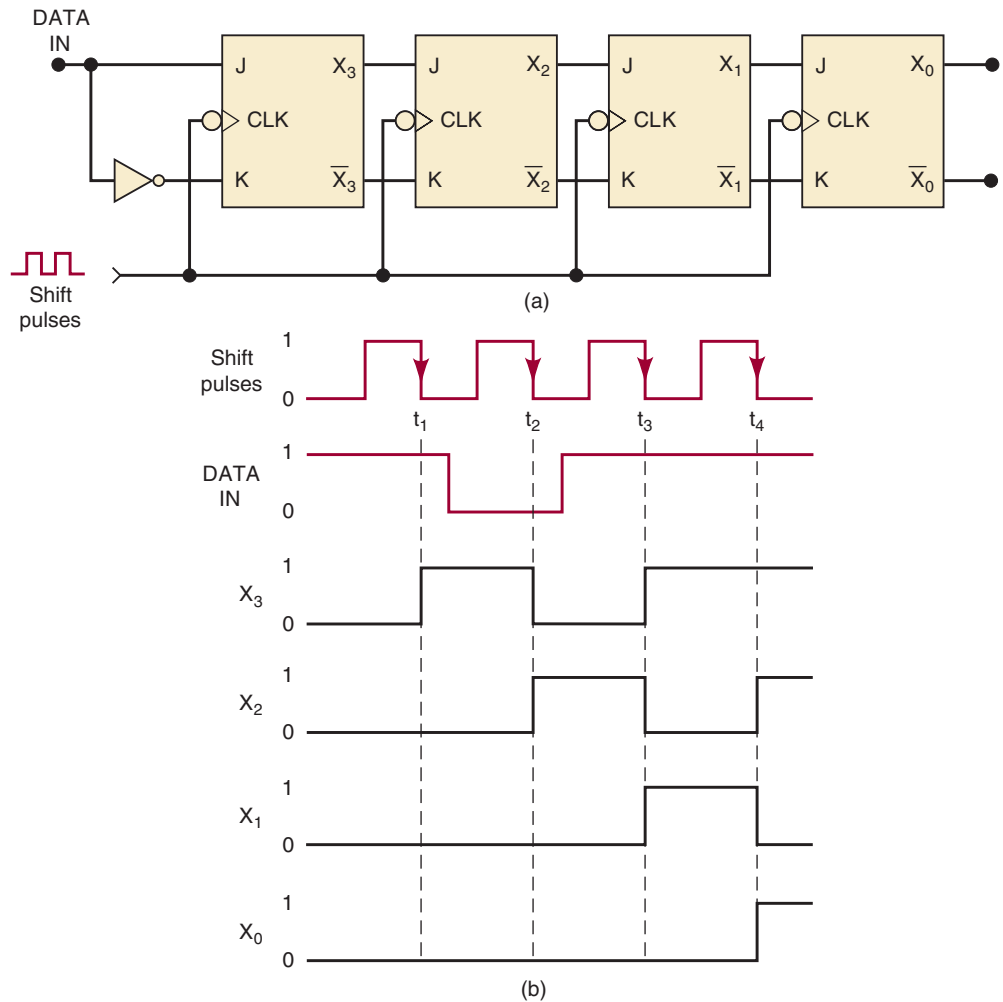
*Upon completion of this section, you will be able to:*

- Compare advantages/disadvantages between serial and parallel data transfer.
- Predict the value present on each FF in a shift register after each clock edge given the logic level of other inputs.

Before we describe the serial data transfer operation, we must first examine the basic *shift-register* arrangement. A **shift register** is a group of FFs arranged so that the binary numbers stored in the FFs are shifted from one FF to the next for every clock pulse. You have undoubtedly seen shift registers in action in devices such as an electronic calculator, where the digits shown on the display shift over each time you key in a new digit. This is the same action taking place in a shift register.

Figure 5-46(a) shows one way to arrange J-K flip-flops to operate as a four-bit shift register. Note that the FFs are connected so that the output of  $X_3$  transfers into  $X_2$ ,  $X_2$  into  $X_1$ , and  $X_1$  into  $X_0$ . What this means is that upon the occurrence of the NGT of a shift pulse, each FF takes on the value stored previously in the FF on its left. Flip-flop  $X_3$  takes on a value determined by the conditions present on its *J* and *K* inputs when the NGT occurs. For now, we will assume that  $X_3$ 's *J* and *K* inputs are fed by the DATA IN waveform shown in Figure 5-46(b). We will also assume that all FFs are in the 0 state before shift pulses are applied.

The waveforms in Figure 5-46(b) show how the input data are shifted from left to right from FF to FF as shift pulses are applied. When the first NGT occurs at  $t_1$ , each of the FFs  $X_2$ ,  $X_1$ , and  $X_0$  will have the  $J = 0$ ,  $K = 1$  condition present at its inputs because of the state of the FF on its left. Flip-flop  $X_3$  will have  $J = 1$ ,  $K = 0$  because of DATA IN. Thus, at  $t_1$ , only  $X_3$  will go HIGH, while all the other FFs remain LOW. When the second NGT occurs at  $t_2$ , flip-flop  $X_3$  will have  $J = 0$ ,  $K = 1$  because of DATA IN. Flip-flop  $X_2$



**FIGURE 5-46** Four-bit shift register.

will have  $J = 1$ ,  $K = 0$  because of the current HIGH at  $X_3$ . Flip-flops  $X_1$  and  $X_0$  will still have  $J = 0$ ,  $K = 1$ . Thus, at  $t_2$ , only FF  $X_2$  will go HIGH, FF  $X_3$  will go LOW, and FFs  $X_1$  and  $X_0$  will remain LOW.

Similar reasoning can be used to determine how the waveforms change at  $t_3$  and  $t_4$ . Note that on each NGT of the shift pulses, each FF output takes on the level that was present at the output of the FF on its left just *prior* to the NGT. Of course,  $X_3$  takes on the level that was present at DATA IN just prior to the NGT.

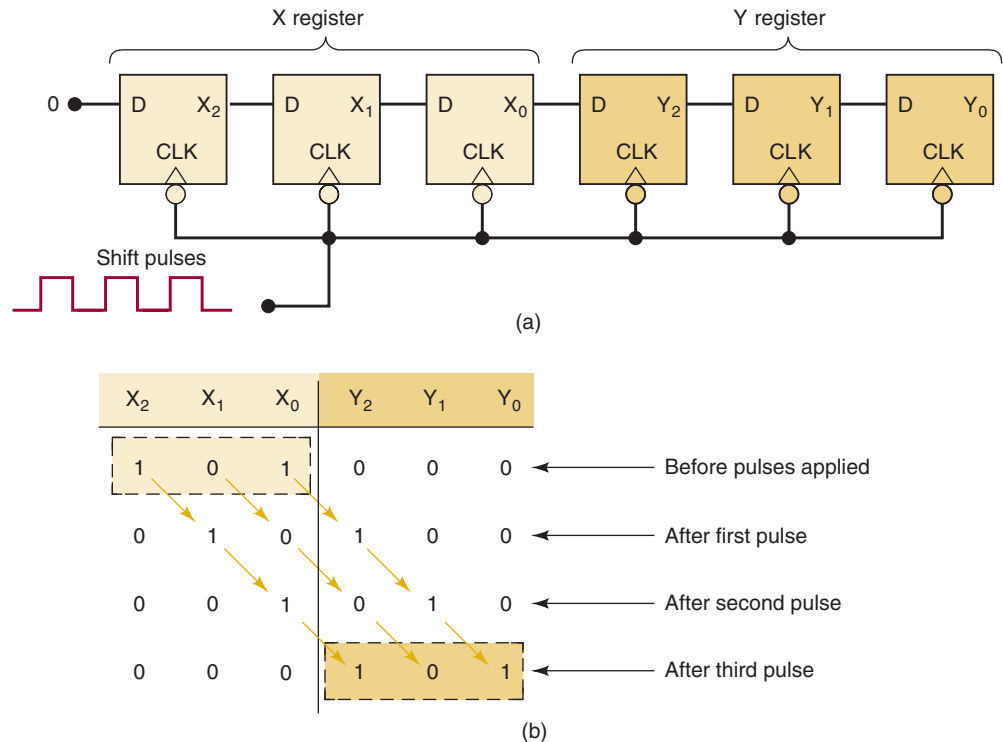
### Hold Time Requirement

In this shift-register arrangement, it is necessary that the FFs have a very small hold time requirement because there are times when the  $J$ ,  $K$  inputs are changing at about the same time as the  $CLK$  transition. For example, the  $X_3$  output switches from 1 to 0 in response to the NGT at  $t_2$ , causing the  $J$ ,  $K$  inputs of FF  $X_2$  to change while its  $CLK$  input is changing. Actually, because of the propagation delay of FF  $X_3$ , the  $J$ ,  $K$  inputs of  $X_2$  won't change for a short time after the NGT. For this reason, a shift register should be

implemented using edge-triggered FFs that have a  $t_{\text{H}}$  value less than one  $\text{CLK}$ -to-output propagation delay. This latter requirement is easily satisfied by most edge-triggered FFs.

### Serial Transfer Between Registers

Figure 5-47(a) shows two three-bit shift registers connected so that the contents of the  $X$  register will be serially transferred (shifted) into register  $Y$ . We are using D flip-flops for each shift register because this requires fewer connections than J-K flip-flops. Notice how  $X_0$ , the last FF of register  $X$ , is connected to the  $D$  input of  $Y_2$ , the first FF of register  $Y$ . Thus, as the shift pulses are applied, the information transfer takes place as follows:  $X_2 \rightarrow X_1 \rightarrow X_0 \rightarrow Y_2 \rightarrow Y_1 \rightarrow Y_0$ . Flip-flop  $X_2$  will go to a state determined by its  $D$  input. For now,  $D$  will be held LOW, so that  $X_2$  will go LOW on the first pulse and will remain there.



**FIGURE 5-47** Serial transfer of information from  $X$  register into  $Y$  register.

To illustrate, let us assume that before any shift pulses are applied, the contents of the  $X$  register are 101 (i.e.,  $X_2 = 1$ ,  $X_1 = 0$ ,  $X_0 = 1$ ) and the  $Y$  register is at 000. Refer to the table in Figure 5-47(b), which shows how the states of each FF change as shift pulses are applied. The following points should be noted:

1. On the NGT of each pulse, each FF takes on the value that was stored in the FF on its left prior to the occurrence of the pulse.
2. After *three* pulses, the 1 that was initially in  $X_2$  is in  $Y_2$ , the 0 initially in  $X_1$  is in  $Y_1$ , and the 1 initially in  $X_0$  is in  $Y_0$ . In other words, the 101 stored in the  $X$  register has now been shifted into the  $Y$  register. The  $X$  register is at 000; it has lost its original data.
3. The complete transfer of the *three* bits of data requires *three* shift pulses.

**EXAMPLE 5-13**

Assume the same initial contents of the  $X$  and  $Y$  registers in Figure 5-47. What will be the contents of each FF after the occurrence of the sixth shift pulse?

**Solution**

If we continue the process shown in Figure 5-47(b) for three more shift pulses, we will find that all of the FFs will be in the 0 state after the sixth pulse. Another way to arrive at this result is to reason as follows: the constant 0 level at the  $D$  input of  $X_2$  shifts in a new 0 with each pulse so that, after six pulses, the registers are filled up with 0s.

**Shift-Left Operation**

The FFs in Figure 5-47 can just as easily be connected so that information shifts from right to left. There is no general advantage of shifting in one direction over another; the direction chosen by a logic designer will often be dictated by the nature of the application, as we shall see.

**Parallel Versus Serial Transfer**

In parallel transfer, all of the information is transferred simultaneously upon the occurrence of a *single* transfer command pulse (Figure 5-45), no matter how many bits are being transferred. In serial transfer, as exemplified by Figure 5-47, the complete transfer of  $N$  bits of information requires  $N$  clock pulses (three bits requires three pulses, four bits requires four pulses, etc.). Parallel transfer, then, is obviously much faster than serial transfer using shift registers.

In parallel transfer, the output of each FF in register  $X$  is connected to a corresponding FF input in register  $Y$ . In serial transfer, only the last FF in register  $X$  is connected to register  $Y$ . In general, then, parallel transfer requires more interconnections between the sending register ( $X$ ) and the receiving register ( $Y$ ) than does serial transfer. This difference becomes more critical when a greater number of bits of information are being transferred. This is an important consideration when the sending and receiving registers are remote from each other because it determines how many lines (wires) are needed for the transmission of the information.

The choice of either parallel or serial transmission depends on the particular system application and specifications. Often, a combination of the two types is used to take advantage of the *speed* of parallel transfer and the *economy and simplicity* of serial transfer. More will be said later about information transfer.

**OUTCOME ASSESSMENT QUESTIONS**

1. *True or false:* The fastest method for transferring data from one register to another is parallel transfer.
2. What is the major advantage of serial transfer over parallel transfer?
3. Refer to Figure 5-47. Assume that the initial contents of the registers are  $X_2 = 0, X_1 = 1, X_0 = 0, Y_2 = 1, Y_1 = 1, Y_0 = 0$ . Also assume that the  $D$  input of FF  $X_2$  is held HIGH. Determine the value of each FF output after the occurrence of the fourth shift pulse.
4. In which form of data transfer does the source of the data not lose its data?

## 5-19 FREQUENCY DIVISION AND COUNTING

### OUTCOMES

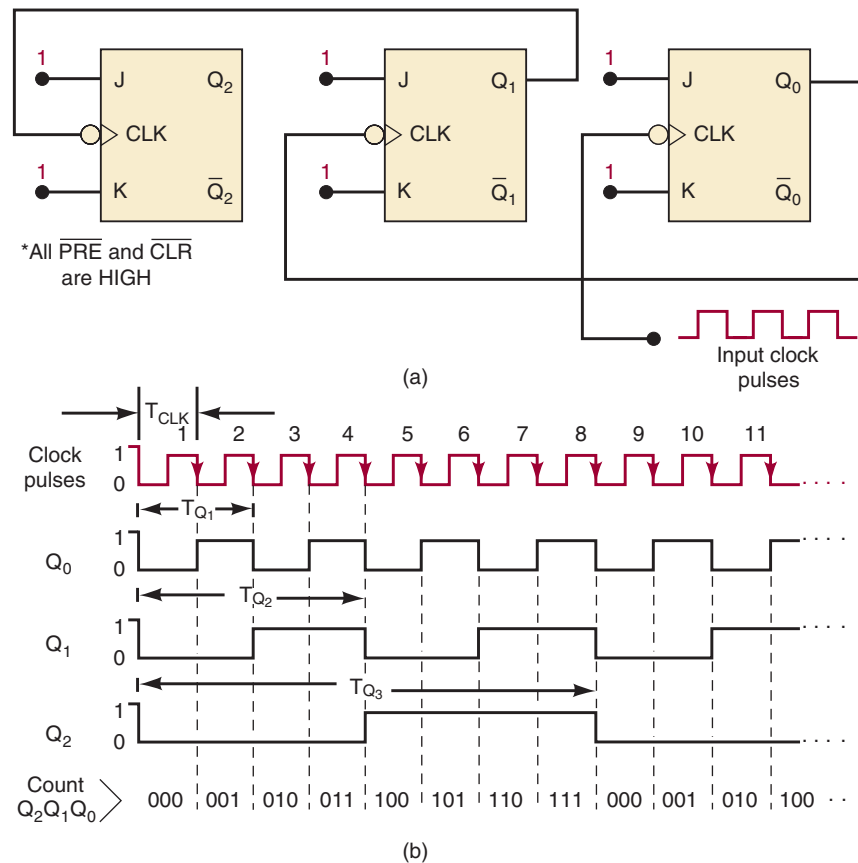
Upon completion of this section, you will be able to:

- Use flip-flops to divide a clock frequency by powers of 2.
- Use flip-flops to create a binary counter.
- Define the MOD number of a counter.
- Determine the frequency of each output of a counter.

Refer to Figure 5-48(a). Each FF has its  $J$  and  $K$  inputs at the 1 level, so that it will change states (toggle) whenever the signal on its  $CLK$  input goes from HIGH to LOW. The clock pulses are applied only to the  $CLK$  input of FF  $Q_0$ . Output  $Q_0$  is connected to the  $CLK$  input of FF  $Q_1$ , and output  $Q_1$  is connected to the  $CLK$  input of FF  $Q_2$ . The waveforms in Figure 5-48(b) show how the FFs change states as the pulses are applied. The following important points should be noted:

1. Flip-flop  $Q_0$  toggles on the negative-going transition of each input clock pulse. Thus, the  $Q_0$  output waveform has a frequency that is exactly one-half of the clock pulse frequency.
2. Flip-flop  $Q_1$  toggles each time the  $Q_0$  output goes from HIGH to LOW. The  $Q_1$  waveform has a frequency equal to exactly one-half the frequency of the  $Q_0$  output and therefore one-fourth of the clock frequency.

**FIGURE 5-48** J-K flip-flops wired as a three-bit binary counter (MOD-8).



3. Flip-flop  $Q_2$  toggles each time the  $Q_1$  output goes from HIGH to LOW. Thus, the  $Q_2$  waveform has one-half the frequency of  $Q_1$  and therefore one-eighth of the clock frequency.
4. Each FF output is a square wave (i.e., each output will be HIGH for one-half of its period).

As described above, each FF divides the frequency of its input by 2. Thus, if we were to add a fourth FF to the chain, it would have a frequency equal to one-sixteenth of the clock frequency, and so on. Using the appropriate number of FFs, this circuit could divide a frequency by any power of 2. Specifically, using  $N$  flip-flops would produce an output frequency from the last FF, which is equal to  $1/2^N$  of the input frequency.

This application of flip-flops is referred to as **frequency division**. Many applications require a frequency division. For example, your wristwatch is no doubt a “quartz” watch. The term *quartz watch* means that a quartz crystal is used to generate a very stable oscillator frequency. The natural resonant frequency of the quartz crystal in your watch is likely 1 MHz or more. In order to advance the “seconds” display once every second, the oscillator frequency is *divided* by a value that will produce a very stable and accurate 1 Hz output frequency.

## Counting Operation

In addition to functioning as a frequency divider, the circuit of Figure 5-48 also operates as a **binary counter**. This can be demonstrated by examining the sequence of states of the FFs after the occurrence of each clock pulse. Figure 5-49 presents the results in a **state table**. Let the  $Q_2Q_1Q_0$  values represent a binary number where  $Q_2$  is in the  $2^2$  position,  $Q_1$  is in the  $2^1$  position, and  $Q_0$  is in the  $2^0$  position. The first eight  $Q_2Q_1Q_0$  states in the table should be recognized as the binary counting sequence from 000 to 111. After the first NGT, the FFs are in the 001 state ( $Q_2 = 0, Q_1 = 0, Q_0 = 1$ ), which represents  $001_2$  (equivalent to decimal 1); after the second NGT, the FFs represent  $010_2$ , which is equivalent to  $2_{10}$ ; after three pulses,  $011_2 = 3_{10}$ ; after four pulses,  $100_2 = 4_{10}$ ; and so on, until after seven pulses,  $111_2 = 7_{10}$ . On the eighth NGT, the FFs return to the 000 state, and the binary sequence repeats itself for succeeding pulses.

**FIGURE 5-49** Table of flip-flop states shows binary counting sequence.

$2^2$	$2^1$	$2^0$	
$Q_2$	$Q_1$	$Q_0$	
0	0	0	Before applying clock pulses
0	0	1	After pulse #1
0	1	0	After pulse #2
0	1	1	After pulse #3
1	0	0	After pulse #4
1	0	1	After pulse #5
1	1	0	After pulse #6
1	1	1	After pulse #7
0	0	0	After pulse #8 recycles to 000
0	0	1	After pulse #9
0	1	0	After pulse #10
0	1	1	After pulse #11
.	.	.	.
.	.	.	.
.	.	.	.



Thus, for the first seven input pulses, the circuit functions as a binary counter in which the states of the FFs represent a binary number equivalent to the number of pulses that have occurred. This counter can count as high as  $111_2 = 7_{10}$  before it returns to 000.

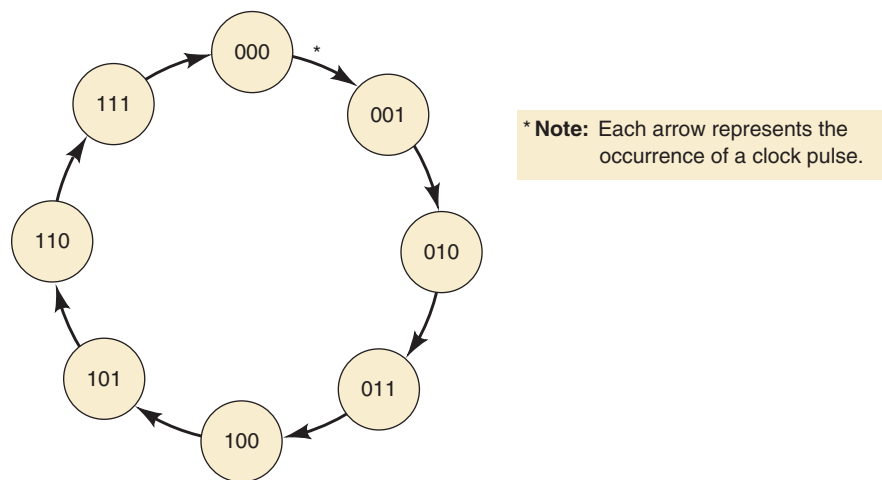
### State Transition Diagram

Another way to show how the states of the FFs change with each applied clock pulse is to use a **state transition diagram**, as illustrated in Figure 5-50. Each circle represents one possible state, as indicated by the binary number inside the circle. For example, the circle containing the number 100 represents the 100 state (i.e.,  $Q_2 = 1, Q_1 = Q_0 = 0$ ).

The arrows connecting one circle to another show how one state changes to another as a clock pulse is applied. By looking at a particular state circle, we can see which state precedes it and which state follows it. For example, looking at the 000 state, we see that this state is reached whenever the counter is in the 111 state and a clock pulse is applied. Likewise, we see that the 000 state is always followed by the 001 state.

We will use state transition diagrams to help describe, analyze, and design counters and other sequential circuits.

**FIGURE 5-50** State transition diagram shows how the states of the counter flip-flops change with each applied clock pulse.



### MOD Number

The counter of Figure 5-48 has  $2^3 = 8$  different states (000 through 111). It would be referred to as a *MOD-8 counter*, where the **MOD number** indicates the number of states in the counting sequence. If a fourth FF were added, the sequence of states would count in binary from 0000 to 1111, a total of 16 states. This would be called a *MOD-16 counter*. In general, if  $N$  flip-flops are connected in the arrangement of Figure 5-48, the counter will have  $2^N$  different states, and so it is a *MOD- $2^N$  counter*. It would be capable of counting up to  $2^N - 1$  before returning to its 0 state.

The MOD number of a counter also indicates the frequency division obtained from the last FF. For instance, a four-bit counter has four FFs, each representing one binary digit (bit), and so it is a  $\text{MOD-}2^4 = \text{MOD-}16$  counter. It can therefore count up to 15 ( $= 2^4 - 1$ ). It can also be used to divide the input pulse frequency by a factor of 16 (the MOD number).

We have looked only at the basic FF binary counter. We examine counters in much more detail in Chapter 7.

**EXAMPLE 5-14**

Assume that the MOD-8 counter in Figure 5-48 is in the 101 state. What will be the state (count) after 13 pulses have been applied?

**Solution**

Locate the 101 state on the state transition diagram. Proceed around the state diagram through eight state changes, and you should be back in the 101 state. Now continue through five more state changes (for a total of 13), and you should end up in the 010 state.

Notice that because this is a MOD-8 counter with eight states, it takes eight state transitions to make one complete excursion around the diagram back to the starting state.

**EXAMPLE 5-15**

Consider a counter circuit that contains six FFs wired in the arrangement of Figure 5-48 (i.e.,  $Q_5, Q_4, Q_3, Q_2, Q_1, Q_0$ ).

- Determine the counter's MOD number.
- Determine the frequency at the output of the last FF ( $Q_5$ ) when the input clock frequency is 1 MHz.
- What is the range of counting states for this counter?
- Assume a starting state (count) of 000000. What will be the counter's state after 129 pulses?

**Solution**

- MOD number =  $2^6 = 64$ .
- The frequency at the last FF will equal the input clock frequency divided by the MOD number. That is,

$$f(\text{at } Q_5) = \frac{1 \text{ MHz}}{64} = 15.625 \text{ kHz}$$

- The counter will count from  $000000_2$  to  $111111_2$  (0 to  $63_{10}$ ) for a total of 64 states. Note that the number of states is the same as the MOD number.
- Because this is a MOD-64 counter, every 64 clock pulses will bring the counter back to its starting state. Therefore, after 128 pulses, the count is back to 000000. The 129th pulse brings the counter to the 000001 state.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

- A 20-kHz clock signal is applied to a J-K flip-flop with  $J = K = 1$ . What is the frequency of the FF output waveform?
- How many FFs are required for a counter that will count 0 to  $255_{10}$ ?
- What is the MOD number of the counter in question 2?
- What is the frequency of the output of the eighth FF when the input clock frequency is 512 kHz?
- If this counter starts at 00000000, what will be its state after 520 pulses?

## 5-20 APPLICATION OF FLIP-FLOPS WITH TIMING CONSTRAINTS

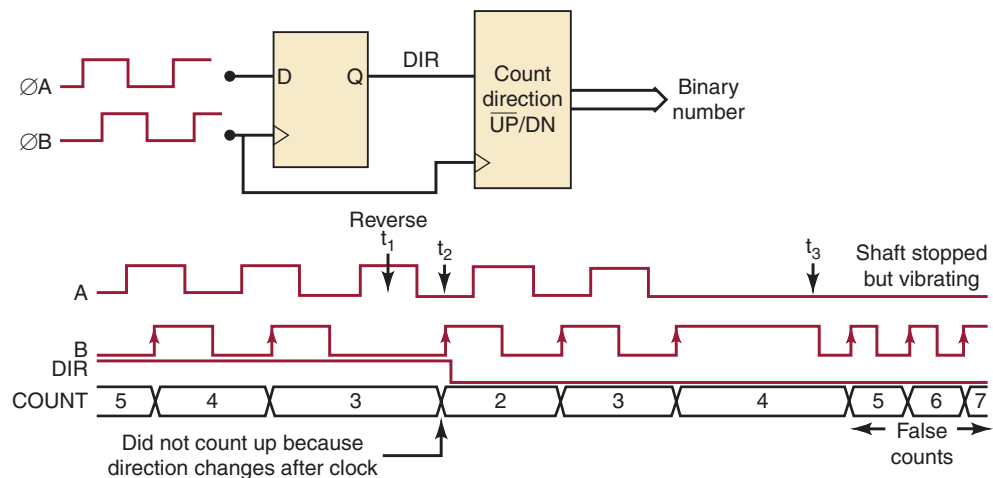
### OUTCOMES

Upon completion of this section, you will be able to:

- Identify common applications of flip-flops.
- Correlate timing limitations with performance issues.

Quadrature shaft encoders were introduced in Section 2-3 and then used in Example 5-12 to demonstrate the ability to detect a sequence. A deeper look at this example reveals that using a simple D flip-flop to keep track of a rotating shaft is inadequate. The objective is to produce a binary number on the output of a counter that always represents the physical position of the shaft. This can be conceptually accomplished by counting up while it is rotating clockwise and then counting down when it is rotating counterclockwise as shown in Figure 5-51. However, the practical reality is that the shaft could reverse at any point in this timing causing the count and the shaft position to become misaligned. Time  $t_1$  shows what can happen when the shaft encoder reverses its rotation. At  $t_2$ , the rising edge on phase  $B$  is the clocking edge for the sequence detector DFF and also for the counter. Notice that the count did not reverse on this clock edge but continued to count down. The counter does not reverse until the next rising edge on  $B$ . An even greater misalignment can happen as shown at  $t_3$ , which is an example showing the shaft that has stopped very close to the transition point of phase  $B$ . If there is a slight vibration of the shaft (very typical), then the phase  $B$  signal would be switching back and forth between LOW and HIGH. The resulting waveform would introduce a major error as the count value continues to increase even though the shaft is not actually rotating. Obviously, this simple sequence detector circuit is inadequate for keeping track of absolute shaft encoder position from a quadrature encoder.

The requirement for keeping the counter and the shaft position aligned is to have the counter update upon every transition of either phase  $A$  or phase  $B$  and to always know which direction is represented by every transition. For example, if the encoder stops very close to a transition on  $B$  as in Figure 5-51 at  $t_3$ , the counter must count up every time  $B$  changes from LOW



**FIGURE 5-51** Problems using a simple sequence detector to create an absolute shaft encoder.

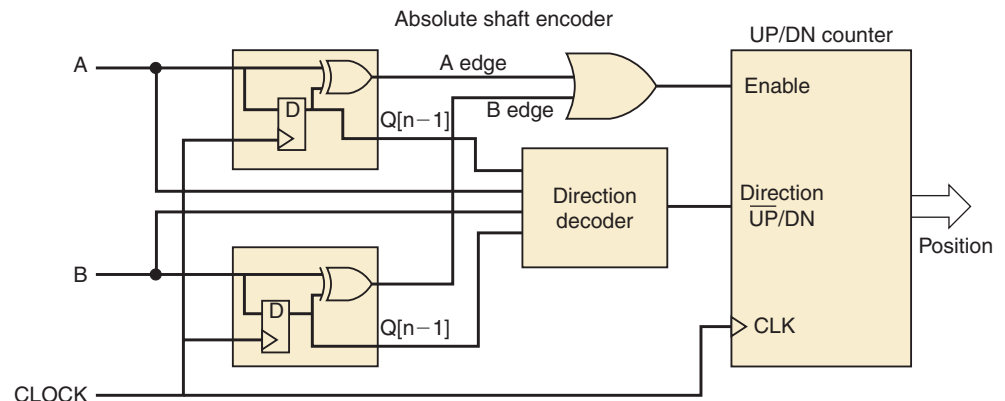
to HIGH and count down every time  $B$  changes from HIGH to LOW. This requires that we use some of the digital circuit design skills of the previous chapters and that we understand the timing limitations of real flip-flops. We also need to explore the use of a clock input that synchronously controls everything in the system. This clock must operate at a frequency that is much higher than the frequency of the input signals.

This example uses some features of counters that will be covered more thoroughly in Chapter 7. At this stage we will treat these counters as functional blocks and simply introduce you to the controls that are typical on such counters. The following definitions of the control inputs should help.

Clk clock	The counter updates its outputs on the rising edge of the clock. The other inputs (enable and direction) control exactly how the counter updates when the clock edge occurs.
Enable	The enable input must be activated for the counter to count up or down. If enable is not activated, the counter will hold its present value.
Direction	This input controls the direction (up or down) that the counter will advance on each clock pulse assuming it is enabled. For this example, when direction = 0, it counts up. When dir = 1, it counts down.

To summarize, this counter will hold its current state as long as it is not enabled (enable = 0). If it is enabled it will either count up (when direction = 0) or count down (when direction = 1) on the next rising clock edge. The clock can be a very high frequency if we simply enable the counter for one clock cycle whenever an edge occurs on phase  $A$  or  $B$ .

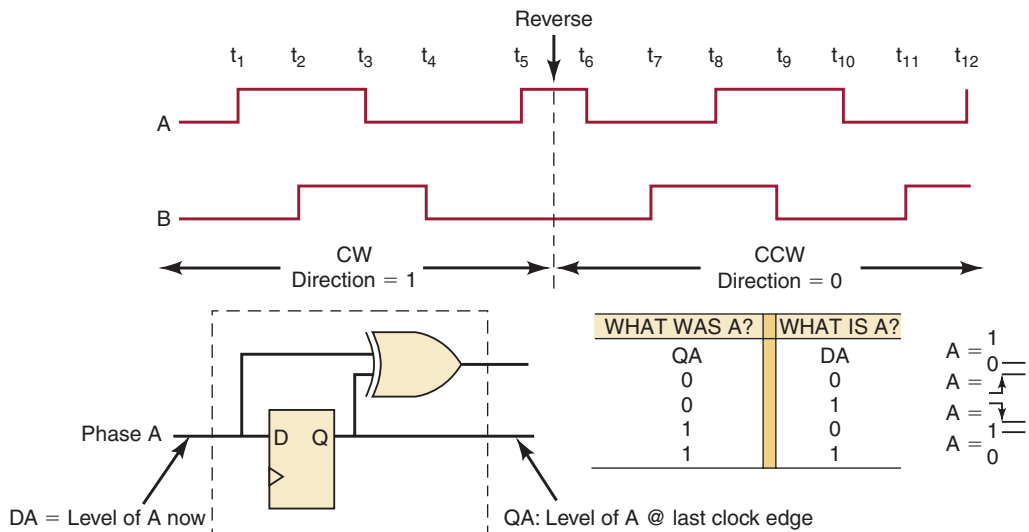
Figure 5-52 shows the block diagram of the system. The overall block has inputs from the quadrature encoder phase  $A$ , phase  $B$ , and clock. The clock should be very high frequency, let's say 50 Mhz. The output is a binary number that represents the position of the shaft. Notice the sub-blocks. The two transition detector blocks are identical and are used to recognize when a transition has occurred on  $A$  or  $B$ . The OR gate is connected to the enable control of the counter and is activated when there has been a transition on either  $A$  or  $B$ . The block in the center of the diagram is responsible for observing the quadrature signals and deciding which direction the shaft is moving. The best way to understand the inputs to this block is to consider the timing diagram of quadrature signals. To know which direction the shaft



**FIGURE 5-52** Block diagram of an absolute shaft position system.

is moving requires that you know which phase changed, which direction it changed, and the logic level of the other phase. Notice the four inputs to the direction decoder circuit. They are the present state of *A* and *B* along with the logic state that was present during the last clock edge (*QA* and *QB*). Whenever a phase changes state, the direction decoder block must determine which direction the shaft moved and tell the counter which way to count on the next rising clock edge.

This is an excellent opportunity to practice translating a real problem into a truth table by designing the direction decoder circuit. For each combination of these four inputs to the direction decoder, it must be determined what just happened (if anything) and determine which direction of rotation is represented. This seems quite complicated when looking at the timing diagram of the quadrature signals, but can be systematically defined by filling out a truth table, one line at a time. There are many ways to organize the truth table of a problem such as this, but in this case it seems best to group the output of each flip-flop with its input. Refer to Figure 5-53 which explains the meaning of the variables that are used to fill out the truth table. The *Q* output represents the level on the quadrature encoder signal on the previous clock edge. Think of this as the signal's previous logic level. The *D* input to the flip-flop has the actual encoder connected to it. Think of this as the signal's logic level now.



**FIGURE 5-53** Defining the direction decoder inputs.

The next step is to fill out the truth table. There are two conditions described in this truth table that are of no interest to us. For these conditions, we don't care whether the counter is told to go up or down.

1. There has been no change in either signal *A* or *B* (i.e.,  $QA = DA$  and  $QB = DB$ ). Note in this case the counter will not be enabled.
2. There has been a change on both signal *A* and *B* (i.e.,  $QA = \overline{DA}$  and  $QB = \overline{DB}$ ) Note that this is impossible with quadrature encoder signals.

Figure 5-54 shows the truth table set up with outputs defined for the different input conditions. For example, on the line 0 of the truth table, the last value of *A* (*QA*) is LOW and the present value of *A* (*DA*) is LOW indicating nothing has changed. The same condition exists for signal *B*. Therefore, we

**FIGURE 5-54** Direction decoder truth table.

	Q <sub>A</sub>	D <sub>A</sub>	Q <sub>B</sub>	D <sub>B</sub>	DIRECTION
(0)	0	0	0	0	X
(1)	0	0	0	1	0
(2)	0	0	1	0	1
(3)	0	0	1	1	X
(4)	0	1	0	0	1
(5)	0	1	0	1	X
(6)	0	1	1	0	X
(7)	0	1	1	1	0
(8)	1	0	0	0	0
(9)	1	0	0	1	X
(10)	1	0	1	0	X
(11)	1	0	1	1	1
(12)	1	1	0	0	X
(13)	1	1	0	1	1
(14)	1	1	1	0	0
(15)	1	1	1	1	X

enter an *x* (don't care) into the truth table output for direction. The direction does not matter because the counter is only enabled when there is a change in input *A* or *B*. Likewise, there has been no change on *A* and no change on *B* for line numbers 3, 12, and 15. Four lines of the truth table represent a change on both *A* and *B* on the same clock cycle. For example, line 5 says *A* went from LOW to HIGH and *B* went from LOW to HIGH. This is impossible for a quadrature encoder and never appears on the timing. Again, we enter an *x* in the output because this condition will never occur. Lines 6, 9, and 10 are also impossible.

The remaining eight entries in the truth table represent the critical points in the input waveforms which must determine the correct direction for the counter. Refer to the sample timing of quadrature signals in Figure 5-53. Line 1 in the truth table says that *A* does not change and *A* is LOW but *B* has changed from LOW to HIGH (a rising edge). Now find on the waveform in Figure 5-53 where this condition exists. Notice that it occurs on the right half of the diagram at  $t_7$  (direction = 0). The next line (2) of the truth table says *A* does not change and *A* is LOW but *B* has changed from HIGH to LOW (a falling edge). Look for this condition on the timing diagram and you will find it at  $t_4$  on the left half where direction = 1. Cover up the direction column in the table and try to use this technique to predict the direction output for lines 4, 7, 8, 11, 13, and 14.

The unsimplified SOP equation for this truth table (considering all *x* entries to be 0) is:

$$\text{Direction} = \overline{Q_A} \overline{D_A} Q_B \overline{D_B} + \overline{Q_A} D_A \overline{Q_B} \overline{D_B} + Q_A \overline{D_A} Q_B D_B + Q_A D_A \overline{Q_B} D_B$$

The Karnaugh map for this truth table is shown in Figure 5-55. When these groupings are made, the simplest SOP expression is

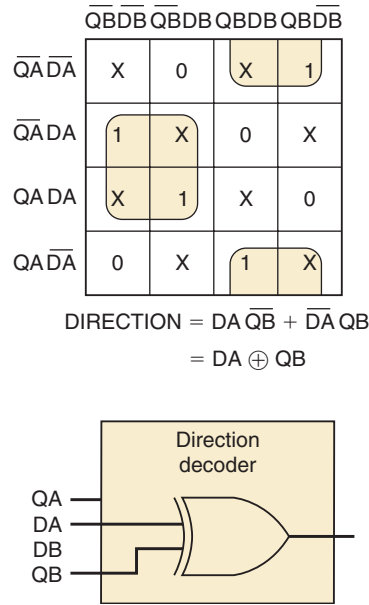
$$\text{Direction} = D_A \overline{Q_B} + \overline{D_A} Q_B$$

This simplifies further to a simple exclusive OR function:

$$\text{Direction} = D_A \oplus Q_B$$

Out of the four possible inputs to this decoder circuit (as described in the truth table), only two are needed to produce the desired result. This is the power of Boolean simplification. It should be noted that there is another

**FIGURE 5-55** K-map, minimized Boolean equation, and logic circuit for the direction decoder.



possible way to group the terms in this K map which results in an equation that uses the other two terms. Either result will work the same.

$$Direction = (\overline{Q}\overline{A}\overline{D}\overline{B}) + (Q\overline{A}D\overline{B}) = \overline{(Q\overline{A}\oplus D\overline{B})}$$

an exclusive NOR function.

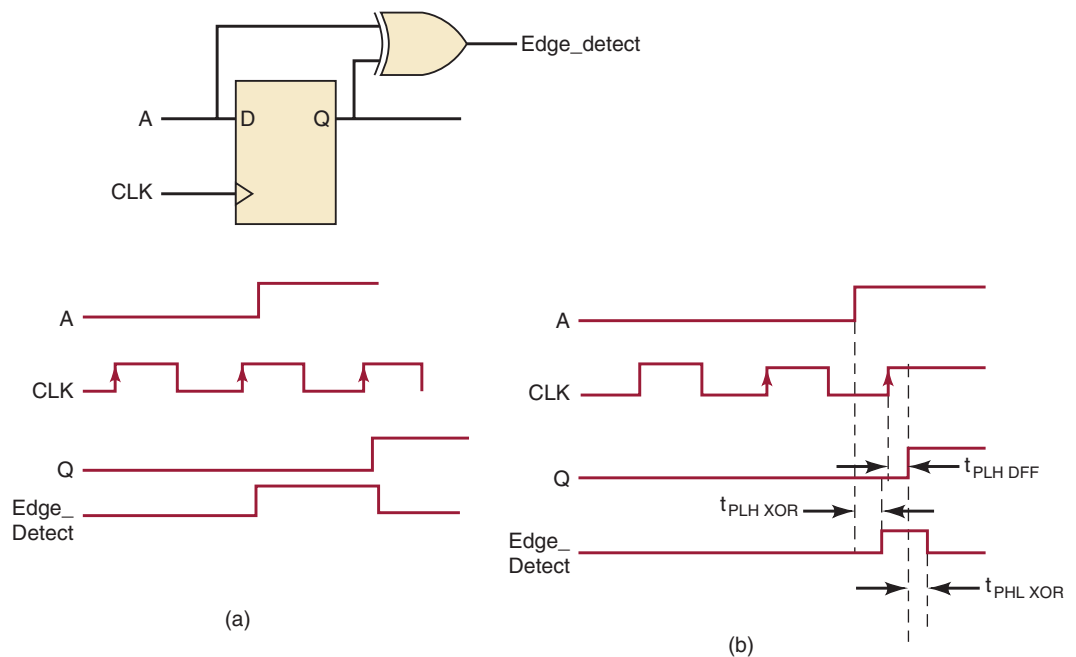
### Timing Issues

The primary goals of this design example are to show (1) applications of flip-flops, (2) combinational circuit design techniques, and to (3) demonstrate real timing constraints. The first two goals have already been accomplished as a DFF was used to detect transitions and the Direction Decoder block was designed and simplified to a simple XOR gate. Some laboratory work is required to demonstrate the timing issues. You can try this yourself, but here are the results we have experienced. If the circuit from Figure 5-52 is implemented in 20 different FPGA boards, some of them will work perfectly and some of them will have problems with repeatability. To clarify the experiment, let's assume the shaft encoder is rotated to a reference position and the counter is reset to zero. When the shaft is rotated clockwise by any number of degrees and then returned counterclockwise exactly to the reference position, the counter should again be at zero. If it is one or two counts away from zero, then the circuit did not respond to every edge perfectly. In an application where this shaft encoder measures the position of a robot arm that welds the seams of your car, the welder would eventually drift away from the seam and put the welds in the wrong place.

Why does the same design work consistently on some circuits and not on others? The answer is in the real timing constraints of the circuitry that implements the digital system. Recall that we discussed timing constraints of propagation delay time, setup time, hold time, and minimum pulse width in previous sections. We also described the need to use flip-flops for the purpose of *synchronization of signals*. With these issues in mind, let's analyze the operation of this circuit.

The problem begins with the nature of the input device and the edge detector circuit. In our example we used a 50 MHz clock, which means the clock edges happen every 20 ns. The pulses that come from the shaft

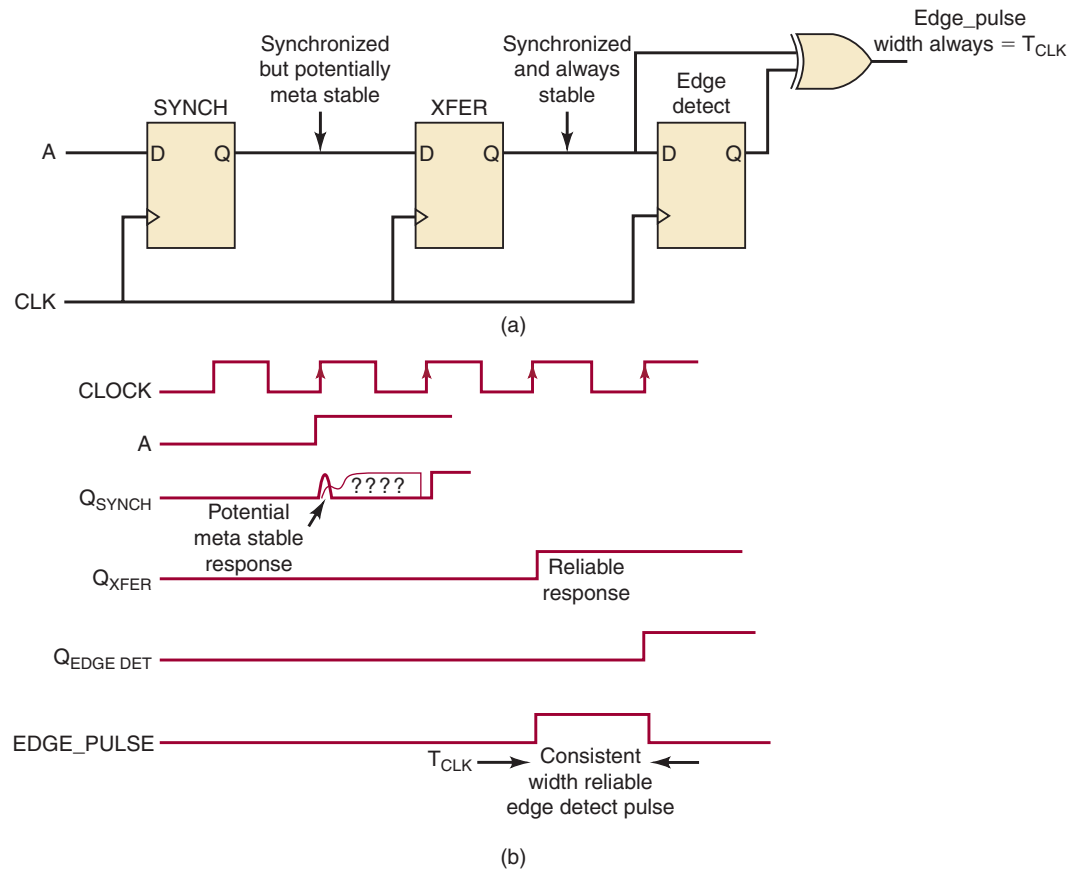
encoder can happen at any time, so it is very likely that at least some of the transitions on  $A$  and  $B$  will occur very close to a clock edge. One problem is that the setup time of the flip-flop may be violated and it will not register the transition until the next clock. In this application a delay of one clock period will not affect the operation of the system. However, consider how narrow the edge detection (output) pulse will be if the encoder changes state very close to the clock edge as shown in Figure 5-56. This situation produces an enable input to the counter circuit that is very narrow, and potentially very close to the counter's clock edge. If the setup and hold time requirements for this enable signal are not met, the counter may not be enabled and the transition will not cause the counter to increment/decrement as it should. The circuit will have "missed" this transition and will be off by one count. Random probability says that some of these transitions will be very close to the clock edge so why do some FPGAs work consistently? The published timing constraints represent the worst-case performance of all the parts manufactured with all the variables accounted for. Some of those parts will respond to very narrow pulses and have very good timing characteristics with very low setup and hold times so they will rarely miss a count. Others perform closer to the published limits and they will exhibit more errors.



**FIGURE 5-56** Timing issues:  $edge\_detect$  pulse may be (a) up to one clock period wide or (b) very narrow and very close to the next PGT of clock.

An improved design will eliminate the possibility of these errors, even for the worst-case timing characteristics of the device chosen for the design. To cure the problem we will add two more applications of flip-flops to the design: synchronization and data transfer. The first flip-flop (far left named *synch*) of Figure 5-57(a) serves the purpose of synchronization. The transition of  $A$  (which is random) will always appear on the  $Q$  output of this flip-flop immediately after the clock edge so we say it is synchronized with the clock. However, it is still possible that the occurrence of the edge on  $A$





**FIGURE 5-57** Improved quadrature encoder circuit: (a) adding synchronizing and data transfer flip-flops; (b) reliable operation depicted in a timing diagram.

will violate the setup time requirement of this synchronizing DFF, which can produce an undesired response at  $Q$  that is referred to as a metastable state. This may result in a short spurious pulse or a delayed transition to the new state. Either one can have adverse effects on the other logic circuitry's performance.

The second flip-flop (*XFER*) simply transfers data from input ( $D$ ) to output ( $Q$ ). Any invalid response from the synch flip-flop will have settled to a valid logic level before it is clocked into the *XFER* flip-flop. This will assure that the *edge-detect* flip-flop always has a clean transition on its input that always occurs exactly one clock period ahead of the clock edge as shown in Figure 5-57(b). The fact that the *edge\_pulse* output is always one clock wide assures stable *enable* and *direction* signals for the synchronous counter circuit.

### OUTCOME ASSESSMENT QUESTIONS

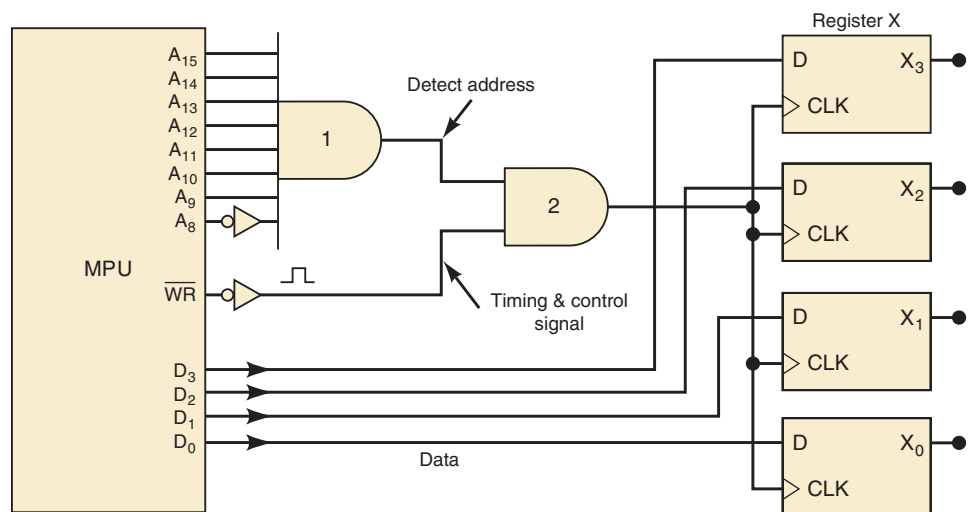
1. Name three common applications of flip-flops that were used in the quadrature encoder system.
2. Why was a single DFF serving as the sequence detector inadequate to keep track of absolute position?
3. What timing issues are at fault in Figure 5-52?

## 5-21 MICROCOMPUTER APPLICATION

Your study of digital systems is still in a relatively early stage, and you have not learned very much about microprocessors and microcomputers. However, you can get a basic idea of how FFs are employed in a typical microprocessor-controlled application without being concerned with all of the details you will need to know later.

Figure 5-58 shows a microprocessor unit (MPU) with its outputs used to transfer binary data to register X, which consists of four D flip-flops  $X_3$ ,  $X_2$ ,  $X_1$ ,  $X_0$ . One set of MPU outputs is the *address code* made up of the eight outputs  $A_{15}$ ,  $A_{14}$ ,  $A_{13}$ ,  $A_{12}$ ,  $A_{11}$ ,  $A_{10}$ ,  $A_9$ ,  $A_8$ . Most MPUs have at least 16 available address outputs, but they are not always all used. A second set of MPU outputs consists of the four *data lines*  $D_3$ ,  $D_2$ ,  $D_1$ ,  $D_0$ . Most MPUs have at least eight available data lines. The other MPU output is a timing control signal  $\overline{WR}$ , which goes LOW when the MPU is ready to write.

**FIGURE 5-58** Example of a microprocessor transferring binary data to an external register.



Recall that the MPU is the central processing unit of a microcomputer, and its main function is to execute a program of instructions stored in the computer's memory. One of the instructions it might execute could be one that tells the MPU to transfer a binary number from a storage register within the MPU to the external register X. This is called a *write cycle*. In executing this instruction, the MPU would perform the following steps:

1. Place the binary number onto its data output lines  $D_3$  through  $D_0$ .
2. Place the proper address code on its output lines  $A_{15}$  through  $A_8$  to select register X as the recipient of the data.
3. Once the data and address outputs are stabilized, the MPU generates the write pulse  $\overline{WR}$  to clock the register and complete the parallel transfer of data into register X.

There are many situations where an MPU, under the control of a program, will send data to an external register in order to control external events. For example, the individual FFs in the register can control the ON/OFF status of electromechanical devices such as solenoids, relays, or motors (through appropriate interface circuits, of course). The data sent from the MPU to the register will determine which devices are ON and which are OFF. Another common example is when the register is used to hold a binary

number for input to a digital-to-analog converter (DAC). The MPU sends the binary number to the register, and the DAC converts it to an analog voltage that may be used to control something such as the position of an electron beam on a CRT screen or the speed of a motor.

### EXAMPLE 5-16

- What address code must the MPU in Figure 5-58 generate in order for the data to be transferred into register  $X$ ?
- Assume that  $X_3-X_0 = 0110$ ,  $A_{15}-A_8 = 11111111$ , and  $D_3-D_0 = 1011$ . What will be in  $X$  after a  $\overline{WR}$  pulse occurs?

### Solution

- In order for the data to be transferred into  $X$ , the clock pulse must pass through AND gate 2 into the  $CLK$  inputs of the FFs. This will happen only if the top input of AND gate 2 is HIGH. This means that all of the inputs to AND gate 1 must be HIGH; that is,  $A_{15}$  through  $A_9$  must be 1, and  $A_8$  must be 0. Thus, the presence of address code 11111110 is needed to allow data to be transferred into register  $X$ .
- With  $A_8 = 1$ , the LOW from AND gate 1 will inhibit  $\overline{WR}$  from getting through AND gate 2, and the FFs will not be clocked. Therefore, the contents of register  $X$  will not change from 0110.

## 5-22 SCHMITT-TRIGGER DEVICES

### OUTCOMES

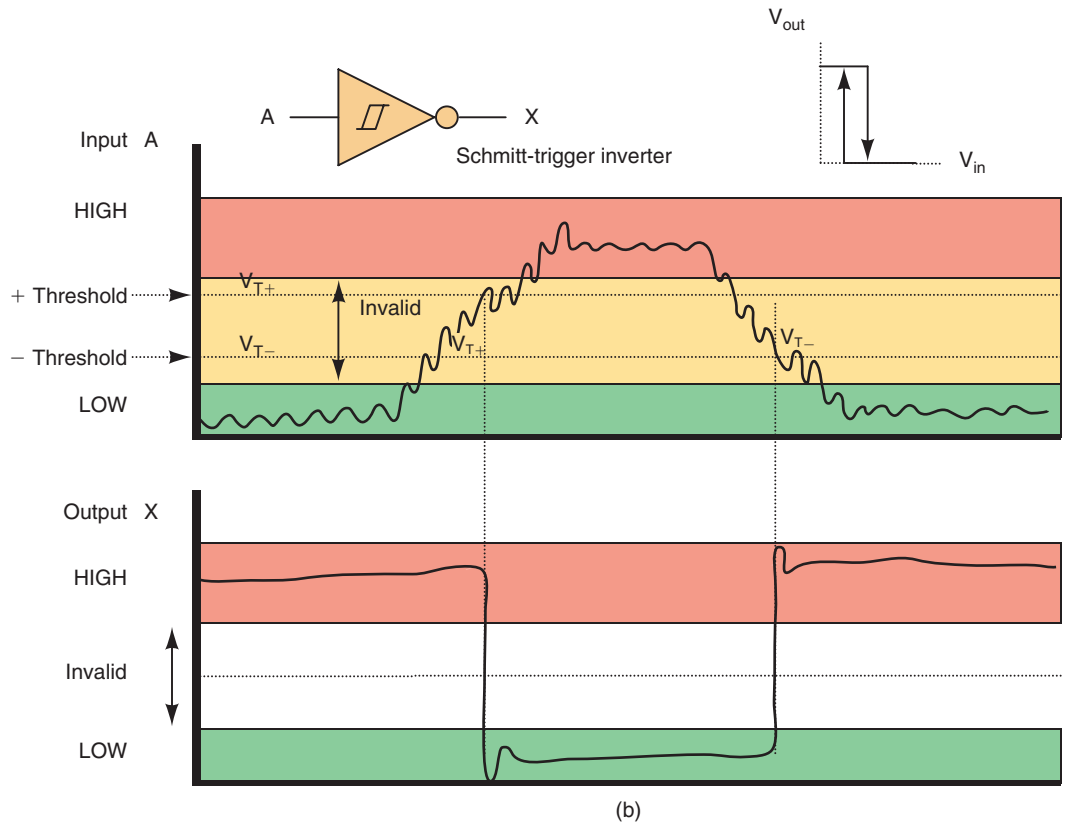
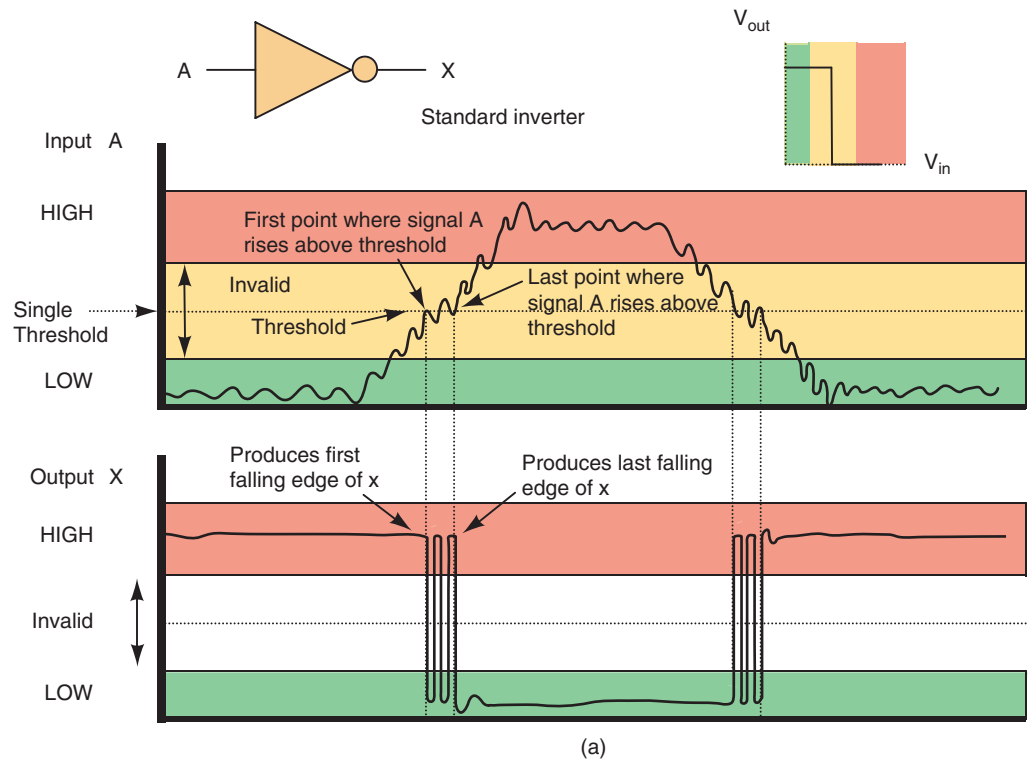
Upon completion of this section, you will be able to:

- Predict results of slow changing inputs on the output of normal logic ICs.
- Predict results of slow changing inputs on the output of Schmitt triggers.

A **Schmitt-trigger circuit** is not classified as a flip-flop, but it does exhibit a type of memory characteristic that makes it useful in certain special situations. One of those situations is shown in Figure 5-59(a). Here a standard INVERTER is being driven by a logic input that has relatively slow transition times. When these transition times exceed the maximum allowed values (this depends on the particular logic family), the outputs of logic gates and INVERTERS may produce oscillations as the input signal passes through the indeterminate range. The same input conditions can also produce erratic triggering of FFs.

A device that has a Schmitt-trigger type of input is designed to accept noisy slow-changing signals and produce an output that has oscillation-free transitions. The output will generally have very rapid transition times (typically 10 ns) that are independent of the input signal characteristics. Figure 5-59(b) shows a Schmitt-trigger INVERTER and its response to a slow-changing input.

If you examine the waveforms in Figure 5-59(b), you should note that the output does not change from HIGH to LOW until the input exceeds the *positive-going threshold* voltage,  $V_{T+}$ . Once the output goes LOW, it will remain there even when the input drops back below  $V_{T+}$  (this is its memory characteristic) until it drops all the way down below the *negative-going threshold*



**FIGURE 5-59** (a) Standard inverter response to slow noisy input, and (b) Schmitt-trigger response to slow noisy input.

voltage,  $V_{T-}$ . The values of the two threshold voltages will vary from logic family to logic family, but  $V_{T-}$  will always be less than  $V_{T+}$ .

The Schmitt-trigger INVERTER, and all other devices with Schmitt-trigger inputs, uses the distinctive symbol shown in Figure 5-59(b) to indicate that they can reliably respond to slow-changing input signals. Logic designers use ICs with Schmitt-trigger inputs to convert slow-changing signals to clean, fast-changing signals that can drive standard IC inputs.

Several ICs are available with Schmitt-trigger inputs. The 7414, 74LS14, and 74HC14 are hex INVERTER ICs with Schmitt-trigger inputs. The 7413, 74LS13, and 74HC13 are dual four-input NANDs with Schmitt-trigger inputs.

### OUTCOME ASSESSMENT QUESTIONS

1. What could occur when a slow-changing signal is applied to a standard logic IC?
2. How does a Schmitt-trigger logic device operate differently from a standard logic device?

## 5-23 ONE-SHOT (MONOSTABLE MULTIVIBRATOR)

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Predict the output of a one-shot before it is triggered.
- Distinguish between behavior of nonretriggerable and retriggerable one-shots.
- Calculate the pulse width of a one-shot.

A digital circuit that is somewhat related to the FF is the **one-shot (OS)**. Like the FF, the OS has two outputs,  $Q$  and  $\bar{Q}$ , which are the inverse of each other. Unlike the FF, the OS has only one *stable* output state (normally  $Q = 0$ ,  $\bar{Q} = 1$ ), where it remains until it is triggered by an input signal. Once triggered, the OS outputs switch to the opposite state ( $Q = 1$ ,  $\bar{Q} = 0$ ). It remains in this **quasi-stable state** for a fixed period of time,  $t_p$ , which is usually determined by an  $RC$  time constant that results from the values of external components connected to the OS. After a time  $t_p$ , the OS outputs return to their resting state until triggered again.

Figure 5-60(a) shows the logic symbol for a OS. The value of  $t_p$  is often indicated somewhere on the OS symbol. In practice,  $t_p$  can vary from several nanoseconds to several tens of seconds. The exact value of  $t_p$  is variable and is determined by the values of external components  $R_T$  and  $C_T$ .

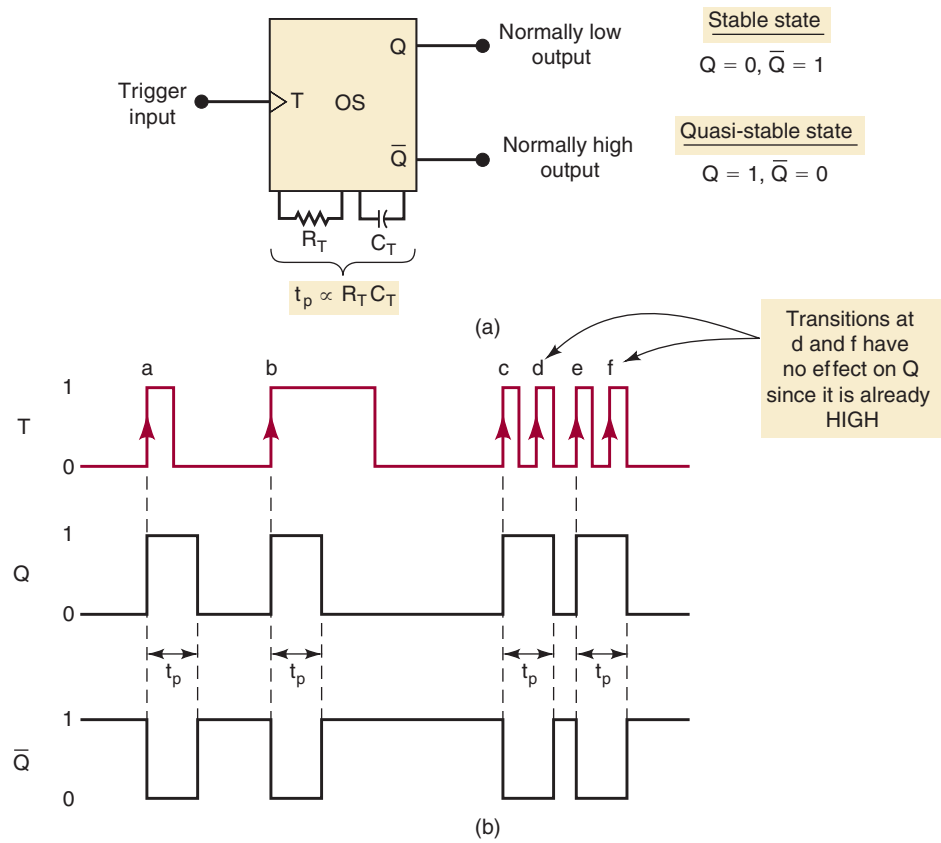
Two types of one-shots are available in IC form: the **nonretriggerable OS** and the **retriggerable OS**.

### Nonretriggerable One-Shot

The waveforms in Figure 5-60(b) illustrate the operation of a nonretriggerable OS that triggers on positive-going transitions at its trigger ( $T$ ) input. The important points to note are:

1. The PGTs at points  $a$ ,  $b$ ,  $c$ , and  $e$  will trigger the OS to its quasi-stable state for a time  $t_p$ , after which it automatically returns to the stable state.

**FIGURE 5-60** OS symbol and typical waveforms for nonretriggerable operation.



- The PGTs at points *d* and *f* have no effect on the OS because it has already been triggered to the quasi-stable state. The OS must return to the stable state before it can be triggered.
- The OS output-pulse duration is always the same, regardless of the duration of the input pulses. As stated above,  $t_p$  depends only on  $R_T$  and  $C_T$  and the internal OS circuitry. A typical OS may have a  $t_p$  given by  $t_p = 0.693R_T C_T$ .

### Retriggerable One-Shot

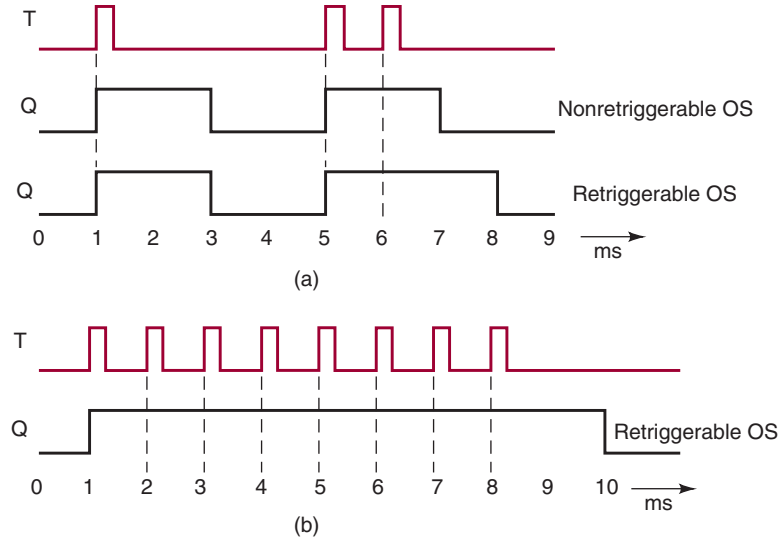
The retriggerable OS operates much like the nonretriggerable OS except for one major difference: *it can be retriggered while it is in the quasi-stable state, and it will begin a new  $t_p$  interval.* Figure 5-61(a) compares the response of both types of OS using a  $t_p$  of 2 ms. Let's examine these waveforms.

Both types of OS respond to the first trigger pulse at  $t = 1$  ms by going HIGH for 2 ms and then returning LOW. The second trigger pulse at  $t = 5$  ms triggers both one-shots to the HIGH state. The third trigger pulse at  $t = 6$  ms has no effect on the nonretriggerable OS because it is already in its quasi-stable state. However, this trigger pulse will *retrigger* the retriggerable OS to begin a new  $t_p = 2$  ms interval. Thus, it will stay HIGH for 2 ms *after* this third trigger pulse.

In effect, then, a retriggerable OS begins a new  $t_p$  interval each time a trigger pulse is applied, regardless of the current state of its  $Q$  output. In fact, trigger pulses can be applied at a rate fast enough that the OS will always be retriggered before the end of the  $t_p$  interval and  $Q$  will remain HIGH. This is shown in Figure 5-61(b), where eight pulses are applied that retrigger the OS every 1 ms.  $Q$  does not return LOW until 2 ms after the last trigger pulse.

**FIGURE 5-61**

(a) Comparison of nonretriggerable and retriggerable OS responses for  $t_p = 2$  ms.  
 (b) Retriggerable OS begins a new  $t_p$  interval each time it receives a trigger pulse.



**Actual Devices**

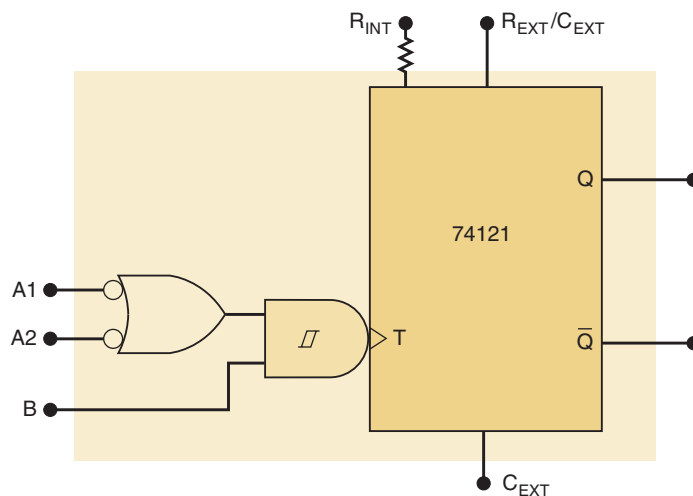
Several one-shot ICs are available in both the retriggerable and the nonretriggerable versions. The 74121 is a single nonretriggerable one-shot IC; the 74221, 74LS221, and 74HC221 are dual nonretriggerable one-shot ICs; the 74122 and 74LS122 are single retriggerable one-shot ICs; the 74123, 74LS123, and 74HC123 are dual retriggerable one-shot ICs.

Figure 5-62 shows the symbol for the 74121 nonretriggerable one-shot IC. Note that it contains internal logic gates to allow inputs  $A_1$ ,  $A_2$ , and  $B$  to trigger the OS in a variety of ways. The  $B$  input is a Schmitt-trigger type of input that is allowed to have slow transition times and still reliably trigger the OS. The pins labeled  $R_{INT}$ ,  $R_{EXT}/C_{EXT}$ , and  $C_{EXT}$  are used to connect an external resistor and capacitor to achieve the desired output pulse duration.

**Monostable Multivibrator**

Another name for the one-shot is *monostable multivibrator* because it has only one stable state. One-shots find limited application in most sequential clock-controlled systems, and experienced designers generally avoid using

**FIGURE 5-62** Logic symbol for the 74121 nonretriggerable one-shot.



them because they are prone to false triggering by spurious noise. When they are used, it is usually in simple timing applications that utilize the pre-determined  $t_p$  interval. Several of the end-of-chapter problems will illustrate how a OS is used.

### OUTCOME ASSESSMENT QUESTIONS

1. In the absence of a trigger pulse, what will be the state of a OS output?
2. *True or false:* When a nonretriggerable OS is pulsed while it is in its quasi-stable state, the output is not affected.
3. What determines the  $t_p$  value for a OS?
4. Describe how a retriggerable OS operates differently from a nonretriggerable OS.

## 5-24 CLOCK GENERATOR CIRCUITS

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the operation of RC oscillators and crystal-controlled oscillators.
- Calculate the frequency of Schmitt-trigger oscillators.
- Calculate the frequency and duty cycle for 555 timer oscillators.

Flip-flops have two stable states; therefore, we can say that they are *bistable multivibrators*. One-shots have one stable state, and so we call them *monostable multivibrators*. A third type of multivibrator has no stable states; it is called an **astable** or **free-running multivibrator**. This type of logic circuit switches back and forth (oscillates) between two unstable output states. It is useful for generating clock signals for synchronous digital circuits.

Several types of astable multivibrators are in common use. We will present three of them without any attempt to analyze their operation. They are presented here so that you can construct a clock generator circuit if needed for a project or for testing digital circuits in the lab.

### Schmitt-Trigger Oscillator

Figure 5-63 shows how a Schmitt-trigger INVERTER can be connected as an oscillator. The signal at  $V_{OUT}$  is an approximate square wave with a frequency that depends on the  $R$  and  $C$  values. The relationship between the frequency and  $RC$  values is shown in Figure 5-63 for three different Schmitt-trigger INVERTERS. Note the maximum limits on the resistance value for each device. The circuit will fail to oscillate if  $R$  is not kept below these limits.

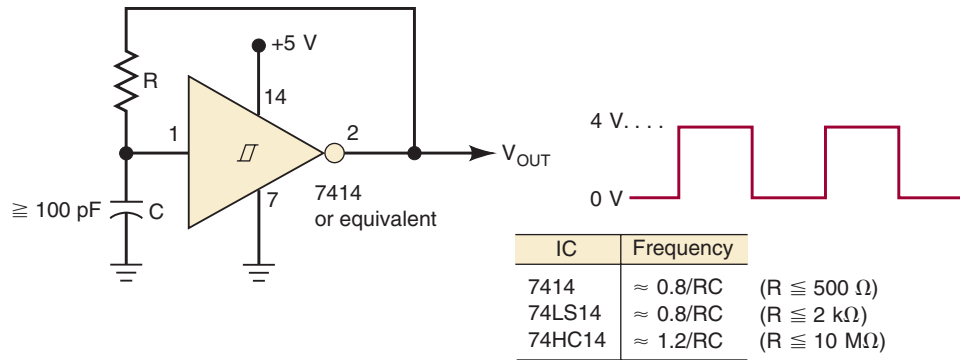
### 555 Timer Used as an Astable Multivibrator

The **555 timer** IC is a TTL-compatible device that can operate in several different modes. Figure 5-64 shows how external components can be connected to a 555 so that it operates as a free-running oscillator. Its output is a repetitive rectangular waveform that switches between two logic levels, with the time intervals at each logic level determined by the  $R$  and  $C$  values.

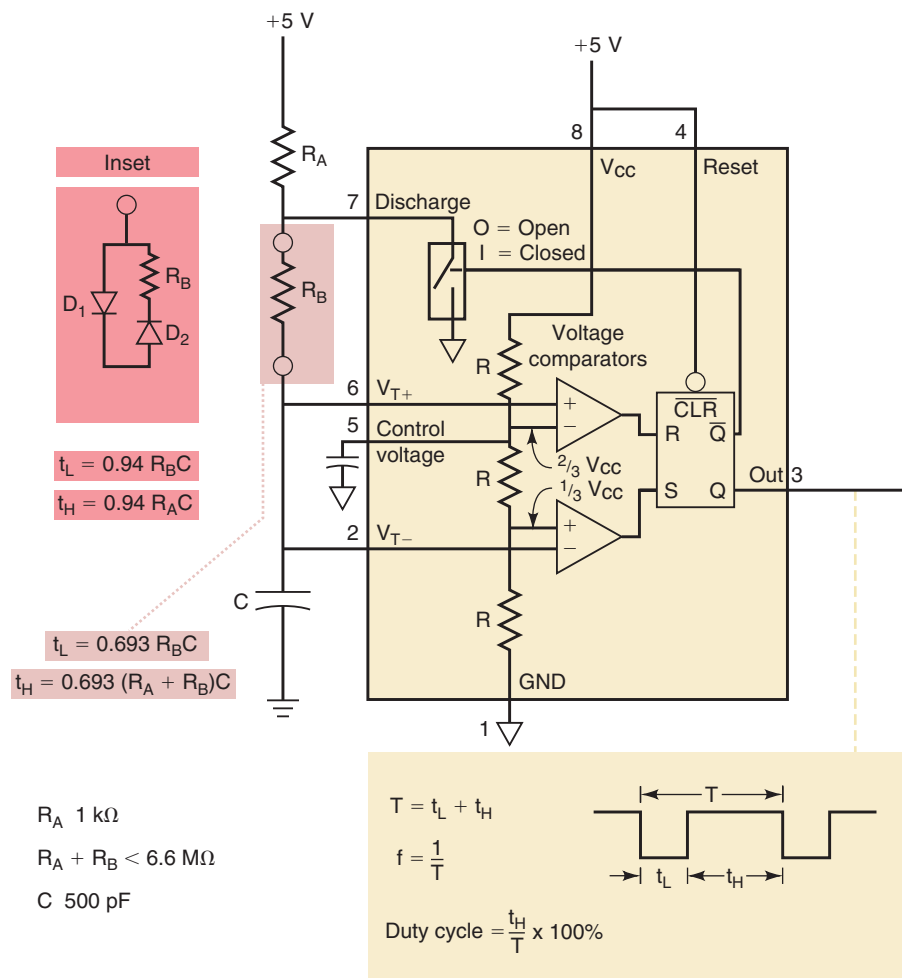


**FIGURE 5-63**

Schmitt-trigger oscillator using a 7414 INVERTER. A 7413 Schmitt-trigger NAND may also be used.



**FIGURE 5-64** 555 timer IC used as an astable multivibrator.



The heart of the 555 timer is made up of two voltage comparators and an S-R latch as shown in Figure 5-64. The voltage comparators are devices that produce a HIGH out whenever the voltage on the + input is greater than the voltage on the - input. The external capacitor (C) charges up until its voltage exceeds  $\frac{2}{3} \times V_{CC}$  as determined by the upper voltage comparator monitoring  $V_{T+}$ . When this comparator output goes HIGH, it resets the S-R latch, causing the output pin (3) to go LOW. At the same time,  $\bar{Q}$  goes HIGH, closing the discharge switch and causing the capacitor to begin to discharge

through  $R_B$ . It will continue to discharge until the capacitor voltage drops below  $\frac{1}{3} \times V_{CC}$  as determined by the lower-voltage comparator monitoring  $V_{T-}$ . When this comparator output goes HIGH, it sets the S-R latch, causing the output pin to go HIGH, opening the discharge switch, and allowing the capacitor to start charging again as the cycle repeats.

The formulas for these time intervals,  $t_L$  and  $t_H$ , and the overall period of the oscillations,  $T$ , are given in the figure. The frequency of the oscillations is, of course, the reciprocal of  $T$ . The **duty cycle** is the ratio between the pulse width (or  $t_H$ ) and the period ( $T$ ) and is expressed as a percentage. As the formulas in the diagram indicate, the  $t_L$  and  $t_H$  intervals cannot be equal unless  $R_A$  is made zero. This cannot be done without producing excess current through the device. This means that it is impossible to produce a perfect 50 percent duty-cycle square wave output with this circuit. It is possible, however, to get very close to a 50 percent duty cycle by making  $R_B \gg R_A$  (while keeping  $R_A$  greater than 1 k $\Omega$ ), so that  $t_L \approx t_H$ .

**EXAMPLE 5-17**

Calculate the frequency and the duty cycle of the 555 astable multivibrator output for  $C = 0.001 \mu\text{F}$ ,  $R_A = 2.2 \text{ k}\Omega$ , and  $R_B = 100 \text{ k}\Omega$ .

**Solution**

$$t_L = 0.693(100 \text{ k}\Omega)(0.001 \mu\text{F}) = 69.3 \mu\text{s}$$

$$t_H = 0.693(102.2 \text{ k}\Omega)(0.001 \mu\text{F}) = 70.7 \mu\text{s}$$

$$T = 69.3 + 70.7 = 140 \mu\text{s}$$

$$f = 1/140 \mu\text{s} = 7.29 \text{ kHz}$$

$$\text{duty cycle} = 70.7/140 = 50.5\%$$

Note that the duty cycle is close to 50 percent (square wave) because  $R_B$  is much greater than  $R_A$ . It can be made even closer to 50 percent by making  $R_B$  even larger compared with  $R_A$ . For instance, you should verify that if we change  $R_A$  to 1 k $\Omega$  (its minimum allowed value), the results are  $f = 7.18 \text{ kHz}$  and duty cycle = 50.3 percent.

A simple modification can be made to this circuit to allow a duty cycle of less than 50 percent. The strategy is to allow the capacitor to fill up (charge) with charged particles that flow only through  $R_A$  and empty (discharge) as charged particles flow only through  $R_B$ . This can be accomplished by simply connecting one diode ( $D_2$ ) in series with  $R_B$  and another diode ( $D_1$ ) in parallel with  $R_B$  and  $D_2$  as shown in the inset of Figure 5-64. The inset circuit replaces  $R_B$  in the drawing. Diodes are devices that allow charged particles to flow through them in only one direction, as indicated by the arrow head. Diode  $D_1$  allows all the charging current which has come through  $R_A$  to bypass  $R_B$ , and  $D_2$  ensures that none of the charging current can flow through  $R_B$ . All of the discharge current flows through  $D_2$  and  $R_B$  when the discharge switch is closed. The equations for the time high and time low for this circuit are

$$t_L = 0.94R_B C$$

$$t_H = 0.94R_A C$$

Note: The constant 0.94 is dependent upon the forward voltage drop of the diodes.

**EXAMPLE 5-18**

Using the diodes along with  $R_B$  as shown in Figure 5-64, calculate the values of  $R_A$  and  $R_B$  necessary to get a 1 kHz, 25 percent duty cycle waveform out of a 555. Assume  $C$  is a  $0.1 \mu\text{F}$  capacitor.

**Solution**

$$T = \frac{1}{f} = \frac{1}{1000} = 0.001 \text{ s} = 1 \text{ ms}$$

$$t_H = 0.25 \times T = 0.25 \times 1 \text{ ms} = 250 \mu\text{s}$$

$$R_A = \frac{t_H}{0.94 \times C} = \frac{250 \mu\text{s}}{0.94 \times 0.1 \mu\text{F}} = 2.66 \text{ k}\Omega \cong 2.7 \text{ k}\Omega \text{ (5\% tolerance)}$$

$$R_B = \frac{t_L}{0.94 \times C} = \frac{750 \mu\text{s}}{0.94 \times 0.1 \mu\text{F}} = 7.98 \text{ k}\Omega \cong 8.2 \text{ k}\Omega \text{ (5\% tolerance)}$$

**Crystal-Controlled Clock Generators**

The output frequencies of the signals from the clock-generating circuits described above depend on the values of resistors and capacitors, and thus they are not extremely accurate or stable. Even if variable resistors are used so that the desired frequency can be adjusted by “tweaking” the resistance values, changes in the  $R$  and  $C$  values will occur with changes in ambient temperature and with aging, thereby causing the adjusted frequency to drift. If frequency accuracy and stability are critical, another method of generating clock signals can be used: a **crystal-controlled clock generator**. It employs a highly stable and accurate component called a *quartz crystal*. A piece of quartz crystal can be cut to a specific size and shape to vibrate (resonate) at a precise frequency that is extremely stable with temperature and aging; frequencies from 10 kHz to 80 MHz are readily achievable. When a crystal is placed in certain circuit configurations, it can produce oscillations at an accurate and stable frequency equal to the crystal’s resonant frequency. Crystal oscillators are available as IC packages.

Crystal-controlled clock generator circuits are used in all microprocessor-based systems and microcomputers, and in any application in which a clock signal is used to generate accurate timing intervals. We will see this in some of the applications we encounter in the following chapters.

**OUTCOME ASSESSMENT QUESTIONS**

1. Determine the approximate frequency of a Schmitt-trigger oscillator that uses a 74HC14 with  $R = 10 \text{ k}\Omega$  and  $C = 0.005 \mu\text{F}$ .
2. Determine the approximate frequency and duty cycle of the 555 oscillator for  $R_A = R_B = 2.2 \text{ k}\Omega$  and  $C = 2000 \text{ pF}$ .
3. What is the advantage of crystal-controlled clock generator circuits over RC-controlled circuits?

**5-25 TROUBLESHOOTING FLIP-FLOP CIRCUITS****OUTCOMES**

Upon completion of this section, you will be able to:

- Identify common faults in flip-flop circuits.
- Describe clock skew and its effects.

Flip-flop ICs are susceptible to the same kinds of internal and external faults that occur in combinational logic circuits. All of the troubleshooting ideas that were discussed in Chapter 4 can readily be applied to circuits that contain FFs as well as logic gates.

Because of their memory characteristic and their clocked operation, FF circuits are subject to several types of faults and associated symptoms that do not occur in combinational circuits. In particular, FF circuits are susceptible to timing problems that are generally not a concern in combinational circuits. The most common types of FF circuit faults are described.

## Open Inputs

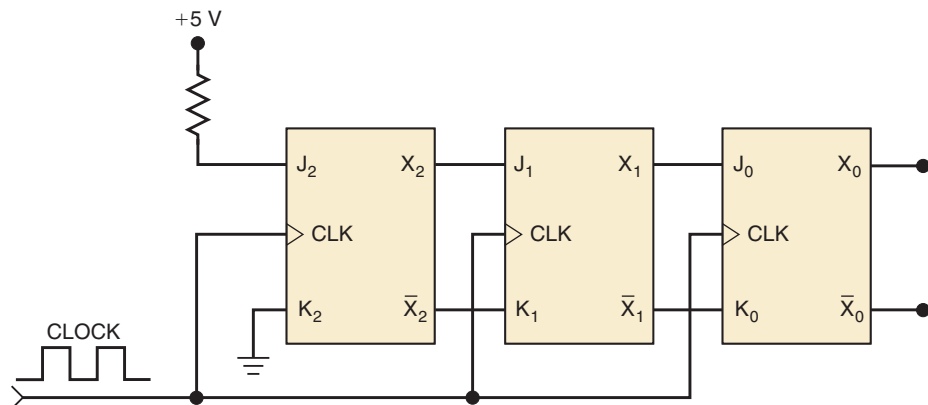
Unconnected or floating inputs of any logic circuit are particularly susceptible to picking up spurious voltage fluctuations called *noise*. If the noise is large enough in amplitude and long enough in duration, the logic circuit's output may change states in response to the noise. In a logic gate, the output will return to its original state when the noise signal subsides. In a FF, however, the output will remain in its new state because of its memory characteristic. Thus, the effect of noise pickup at any open input is usually more critical for a FF or latch than it is for a logic gate.

The most susceptible FF inputs are those that can trigger the FF to a different state—such as the *CLK*, *PRESET*, and *CLEAR*. Whenever you see a FF output that is changing states erratically, you should consider the possibility of an open connection at one of these inputs.

### EXAMPLE 5-19

Figure 5-65 shows a three-bit shift register made up of TTL flip-flops. Initially, all of the FFs are in the LOW state before clock pulses are applied. As clock pulses are applied, each PGT will cause the information to shift

**FIGURE 5-65** Example 5-19.



Clock pulse number	"Expected"			"Actual"		
	X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>
0	0	0	0	0	0	0
1	1	0	0	1	0	0
2	1	1	0	1	1	0
3	1	1	1	1	1	1
4	1	1	1	1	1	0
5	1	1	1	1	1	1
6	1	1	1	1	1	0
7	1	1	1	1	1	1
8	1	1	1	1	1	0

from each FF to the one on its right. The diagram shows the “expected” sequence of FF states after each clock pulse. Since  $J_2 = 1$  and  $K_2 = 0$ , flip-flop  $X_2$  will go HIGH on clock pulse 1 and will stay there for all subsequent pulses. This HIGH will shift into  $X_1$ , and then  $X_0$  on clock pulses 2 and 3, respectively. Thus, after the third pulse, all FFs will be HIGH and should remain there as pulses are continually applied.

Now let’s suppose that the “actual” response of the FF states is as shown in the diagram. Here the FFs change as expected for the first three clock pulses. From then on, flip-flop  $X_0$ , instead of staying HIGH, alternates between HIGH and LOW. What possible circuit fault can produce this operation?

### Solution

On the second pulse, FF  $X_1$  goes HIGH. This should make  $J_0 = 1$ ,  $K_0 = 0$  so that all subsequent clock pulses should set  $X_0 = 1$ . Instead, we see  $X_0$  changing states (toggling) on all pulses after the second one. This toggle operation would occur if  $J_0$  and  $K_0$  were both HIGH. The most probable fault is a break in the connection between  $\bar{X}_1$  and  $K_0$ . Recall that a TTL device responds to an open input as if it were a logic HIGH, so that an open at  $K_0$  is the same as a HIGH.

## Shorted Outputs

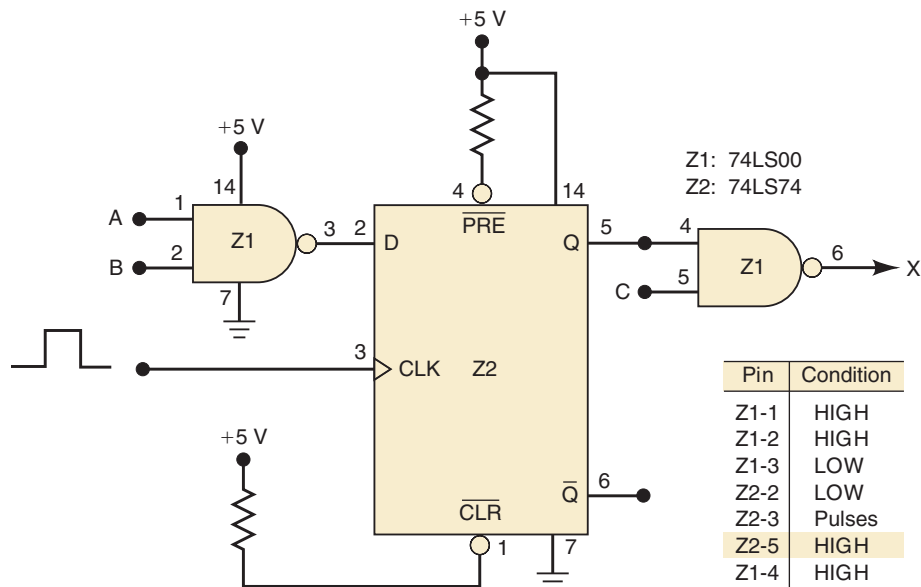
The following example illustrates how a fault in a FF circuit can cause a misleading symptom that may result in a longer time to isolate the fault.

### EXAMPLE 5-20

Consider the circuit in Figure 5-66 and examine the logic probe indications shown in the accompanying table. There is a LOW at the  $D$  input of the FF when pulses are applied to its  $CLK$  input, but the  $Q$  output fails to go to the LOW state. The technician testing this circuit considers each of the following possible circuit faults:

1. Z2-5 is internally shorted to  $V_{CC}$ .
2. Z1-4 is internally shorted to  $V_{CC}$ .

**FIGURE 5-66** Example 5-20.



3. Z2-5 or Z1-4 is externally shorted to  $V_{CC}$ .
4. Z2-4 is internally or externally shorted to GROUND. This would keep  $\overline{PRE}$  activated and would override the  $CLK$  input.
5. There is an internal failure in Z2 that prevents FF  $Q$  from responding properly to its inputs.

The technician, after making the necessary ohmmeter checks, rules out the first four possibilities. He also checks Z2's  $V_{CC}$  and GROUND pins and finds that they are at the proper voltages. He is reluctant to unsolder Z2 from the circuit until he is certain that it is faulty, and so he decides to look at the clock signal. He uses an oscilloscope to check its amplitude, frequency, pulse width, and transition times. He finds that they are all within the specifications for the 74LS74. Finally, he concludes that Z2 is faulty.

He removes the 74LS74 chip and replaces it with another one. To his dismay, the circuit with the new chip behaves in exactly the same way. After scratching his head, he decides to change the NAND gate chip, although he doesn't know why. As expected, he sees no change in the circuit operation.

Becoming more puzzled, he recalls that his electronics lab instructor emphasized the value of performing a thorough visual check on the circuit board, and so he begins to examine it carefully. While he is doing that, he detects a solder bridge between pins 6 and 7 of Z2. He removes it and tests the circuit, and it functions correctly. Explain how this fault produced the operation observed.

### Solution

The solder bridge was shorting the  $\overline{Q}$  output to GROUND. This means that  $\overline{Q}$  is permanently stuck LOW. Recall that in all latches and FFs, the  $\overline{Q}$  and  $Q$  outputs are internally cross-coupled so that the level on one will affect the other. For example, take another look at the internal circuitry for a J-K flip-flop in Figure 5-26. Note that a constant LOW at  $\overline{Q}$  would keep a LOW at one input of NAND gate 3 so that  $Q$  would have to stay HIGH, regardless of the conditions at  $J$ ,  $K$ , and  $CLK$ .

The technician learned a valuable lesson about troubleshooting FF circuits. He learned that both outputs should be checked for faults, even those that are not connected to other devices.

---

## Clock Skew

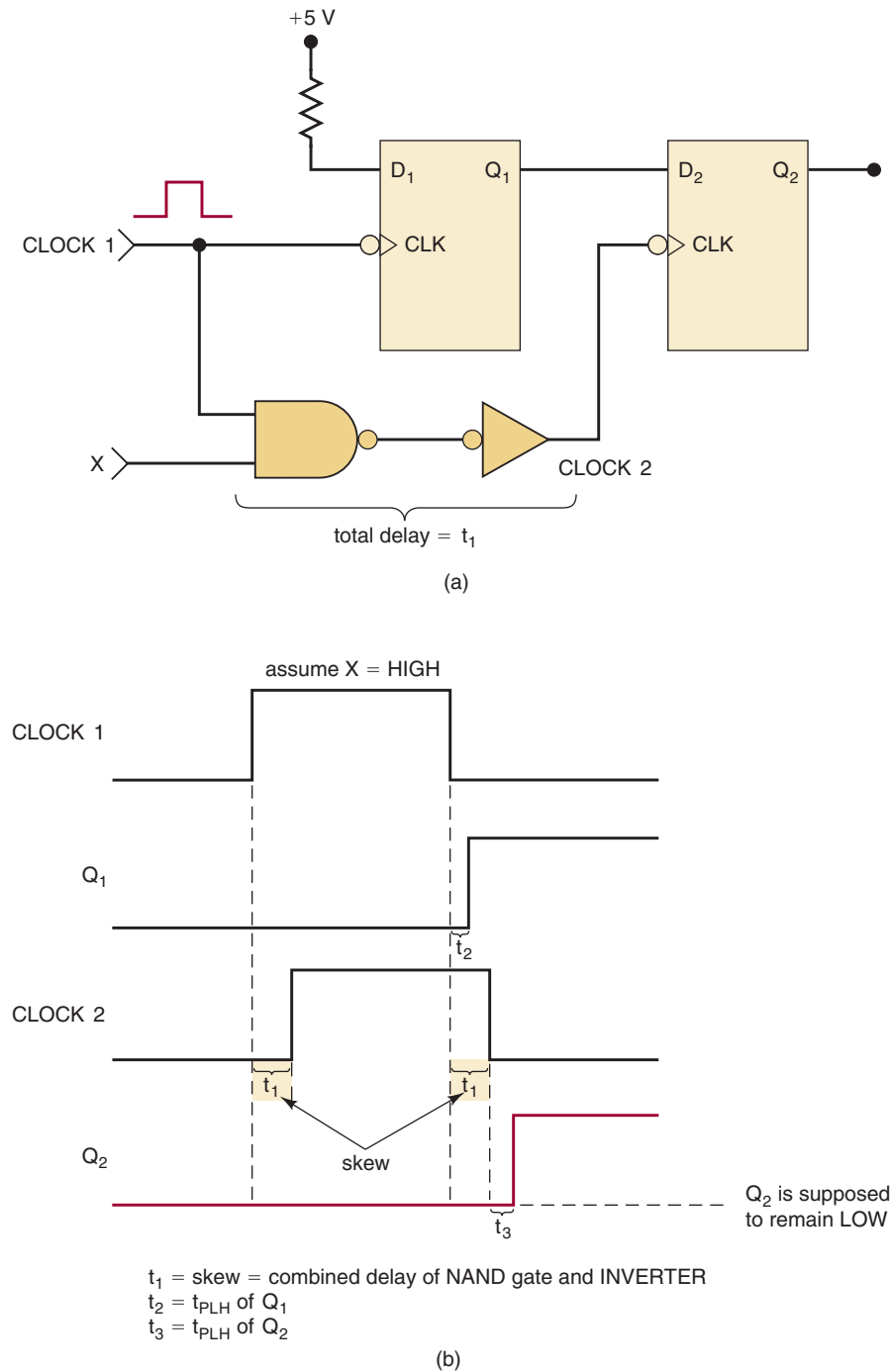
One of the most common timing problems in sequential circuits is **clock skew**. One type of clock skew occurs when a clock signal, because of propagation delays, arrives at the  $CLK$  inputs of different FFs at different times. In many situations, the skew can cause a FF to go to a wrong state. This is best illustrated with an example.

Refer to Figure 5-67(a), where the signal  $CLOCK1$  is connected directly to FF  $Q_1$ , and indirectly to  $Q_2$  through a NAND gate and INVERTER. Both FFs are supposed to be clocked by the occurrence of a NGT of  $CLOCK1$  provided that  $X$  is HIGH. If we assume that initially  $Q_1 = Q_2 = 0$  and  $X = 1$ , the NGT of  $CLOCK1$  should set  $Q_1 = 1$  and have no effect on  $Q_2$ . The waveforms in Figure 5-67(b) show how clock skew can produce incorrect triggering of  $Q_2$ .

Because of the combined propagation delays of the NAND gate and INVERTER, the transitions of the  $CLOCK2$  signal are delayed with respect to  $CLOCK1$  by an amount of time  $t_1$ . The NGT of  $CLOCK2$  arrives at  $Q_2$ 's  $CLK$  input  $t_1$  later than the NGT of  $CLOCK1$  appears at  $Q_1$ 's  $CLK$  input. This  $t_1$  is

---

**FIGURE 5-67** Clock skew occurs when two flip-flops that are supposed to be clocked simultaneously are clocked at slightly different times due to a delay in the arrival of the clock signal at the second flip-flop. (a) Extra gating circuits that can cause clock skew; (b) timing showing the later arrival of CLOCK 2.



the clock skew. The NGT of *CLOCK1* will cause *Q<sub>1</sub>* to go HIGH after a time  $t_2$  that is equal to *Q<sub>1</sub>*'s  $t_{\text{PLH}}$  propagation delay. If  $t_2$  is less than the skew  $t_1$ , *Q<sub>1</sub>* will be HIGH when the NGT of *CLOCK2* occurs, and this may incorrectly set *Q<sub>2</sub>* = 1 if its setup time requirement,  $t_s$ , is met.

For example, assume that the clock skew is 40 ns and the  $t_{\text{PLH}}$  of *Q<sub>1</sub>* is 25 ns. Thus, *Q<sub>1</sub>* will go HIGH 15 ns before the NGT of *CLOCK2*. If *Q<sub>2</sub>*'s setup time requirement is smaller than 15 ns, *Q<sub>2</sub>* will respond to the HIGH at its *D* input when the NGT of *CLOCK2* occurs, and *Q<sub>2</sub>* will go HIGH. This, of course, is not the expected response of *Q<sub>2</sub>*. It is supposed to remain LOW.

The effects of clock skew are not always easy to detect because the response of the affected FF may be intermittent (sometimes it works

correctly, sometimes it doesn't). This is because the situation is dependent on circuit propagation delays and FF timing parameters, which vary with temperature, length of connections, power supply voltage, and loading. Sometimes just connecting an oscilloscope probe to a FF or gate output will add enough load capacitance to increase the device's propagation delay so that the circuit functions correctly; then when the probe is removed, the incorrect operation reappears. This is the kind of situation that explains why some technicians are prematurely gray.

Problems caused by clock skew can be eliminated by equalizing the delays in the various paths of the clock signal so that the active transition arrives at each FF at approximately the same time. This situation is examined in Problem 5-52.

OUTCOME  
ASSESSMENT  
QUESTION

1. What is clock skew? How can it cause a problem?

## 5-26 SEQUENTIAL CIRCUITS IN PLDs USING SCHEMATIC ENTRY\*

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Use schematic entry tools to create sequential circuits in Quartus.
- Identify the libraries where common building block symbols are found in Quartus.

Logic circuits that use flip-flops and latches can be implemented with PLDs. Altera's Quartus II development system software allows the designer the option of describing the desired circuit using schematics. Quartus provides component libraries that contain flip-flop and latch devices that can be used to create the schematics. These libraries are named **primitives**, **maxplus2**, and **megafunction**. The primitives library contains the basic logic gates and all types of standard flip-flop and latch storage elements. You have probably already used this library to create combinational circuit schematics. Some of the available storage elements are dff, jkff, srff, tff, and latch.

Flip-flops and other primitives can be combined into logic circuits. The entire logic circuit can then be represented by a block symbol that simply shows its inputs and outputs. In the following chapters, you will learn how to create a block symbol for any design whether it is described by graphic symbols, AHDL, or VHDL.

Altera has also implemented equivalent versions of the ubiquitous (and also outdated) 74xxx standard logic chips to use in your PLD design schematics. These blocks can be found in the maxplus2 library and include not only the fundamental logic devices that are contained in SSI chips but also the more complex common logic functions that have been implemented as MSI chips. You can discover the functionality of the maxplus2 components, sometimes called macrofunctions, by looking up (on the web or in data books) the corresponding data sheet for the equivalent chip from various manufacturers. Some examples are 74112

\*All sections covering PLDs and HDLs may be skipped without loss of continuity.



(JK flip-flop), 74175 (four-bit register), and 74375 (four-bit latch). Altera makes this note to Quartus users, “In general, Altera recommends using megafunctions in preference to equivalent macrofunctions in all new projects. Megafunctions are easier to scale to different sizes and may offer more efficient logic synthesis and device implementation. The Quartus II software supports macrofunctions only for backward compatibility with designs created with other EDA tools.”

The megafunction library contains various high-level modules that can be used to create logic designs. Several of the included components are called **LPMs**, which refers to a subset **library of parameterized modules**. These functions do not attempt to imitate a particular standard IC like the devices in the maxplus2 library; instead, they offer a generic solution for the various types of logic functions that are useful in digital systems. The term *parameterized* means that when you instantiate a function from the library, you also specify parameters that define certain attributes for the circuit you are describing. These versatile blocks are quickly and easily customized to have the desired features and sizes using the MegaWizard Manager in Quartus. The designer simply specifies the needed device characteristics when setting up the module for use in the application. The various LPMs that are available can be found through the HELP menu under megafunctions/LPM.

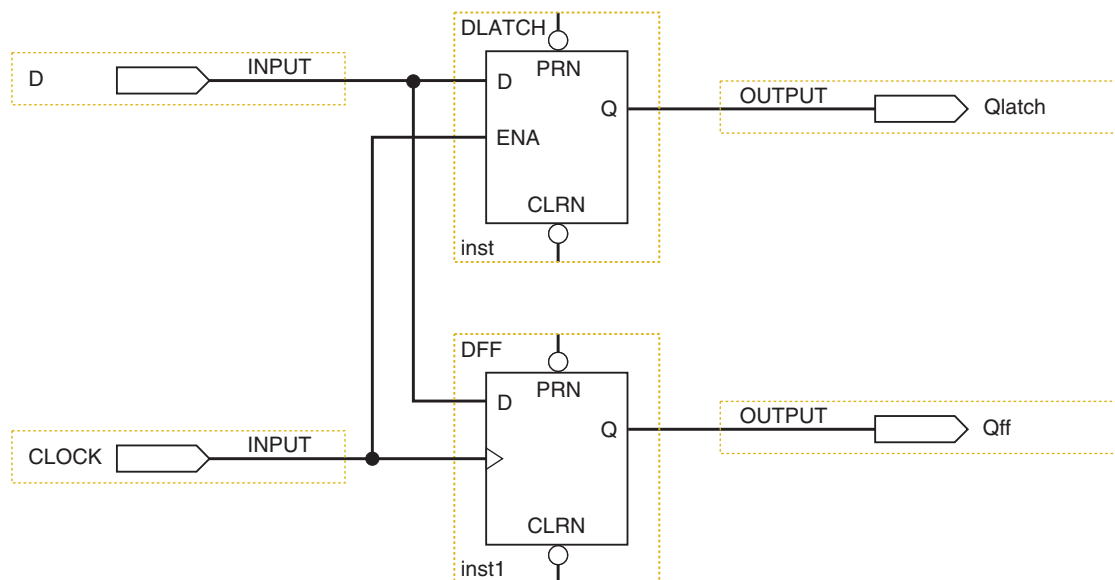
Of course, the Quartus II simulator can be used to verify the sequential circuits that have been created by schematic capture before you program a PLD for use in your design just as you can with combinational circuits.

### EXAMPLE 5-21

Compare the operation of a level-enabled D latch and an edge-triggered D flip-flop using the Quartus functional simulator.

#### Solution

Figure 5-68 shows the Quartus schematic that includes the latch and flip-flop (both from the primitives library) to be tested. These logic primitives can

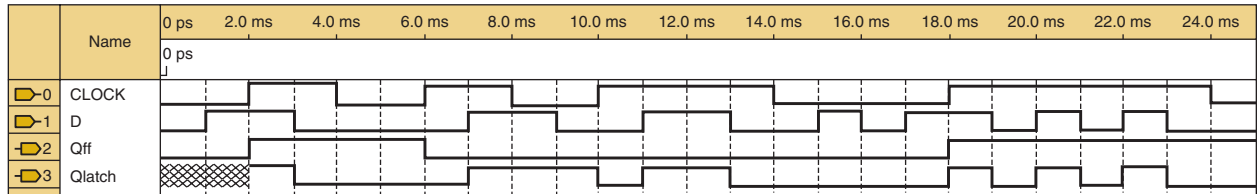


**FIGURE 5-68** D latch and D flip-flop schematic.

be found by double clicking on the bdf canvas and looking in the Libraries box under

`/quartus/libraries/primitives/storage/`

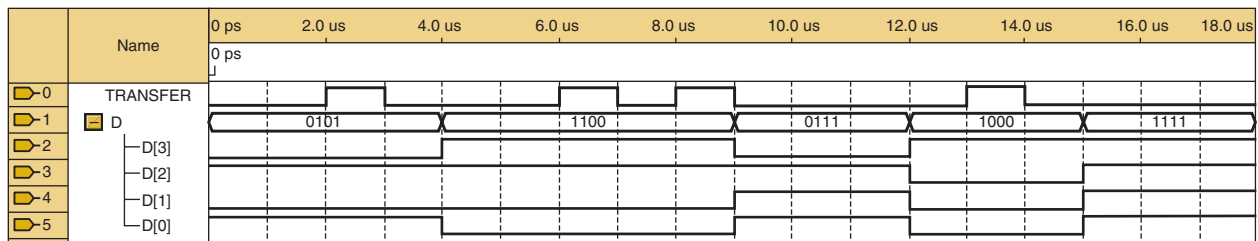
The names of the primitives are DLATCH and DFF. Figure 5-69 gives a simulation report that illustrates the operational differences between the latch and the flip-flop. The latch is “transparent” whenever it is enabled with a HIGH level and is “latched” when the enable is LOW. Whenever the latch enable is HIGH, the output will track the *D* input. On the other hand, the flip-flop will only read and store the *D* input value on the rising edge of the clock input.



**FIGURE 5-69** D latch and D flip-flop simulation report.

### EXAMPLE 5-22

Construct a block description file (bdf) of a register that consists of four D flip-flops using the DFF primitive. From this BDF file, create a block symbol and include it in a new design. The new design will have inputs named  $D[3..0]$ , TRANSFER (clock input), and  $Q[3..0]$ . Determine its operation when the input signals shown in Figure 5-70 are applied. Functionally simulate the register using Quartus to verify your prediction.



**FIGURE 5-70** Input signals for Example 5-22.

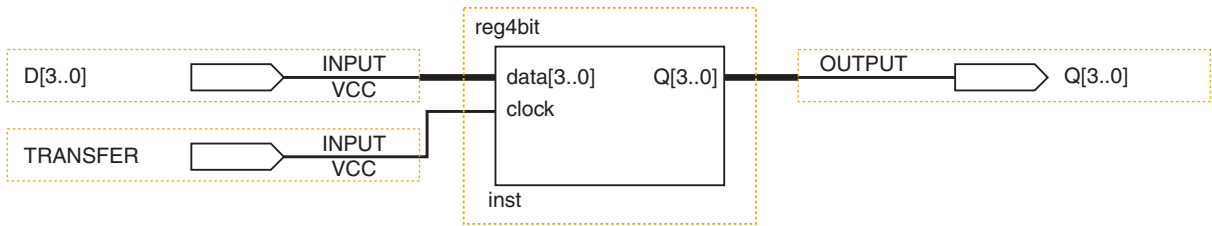
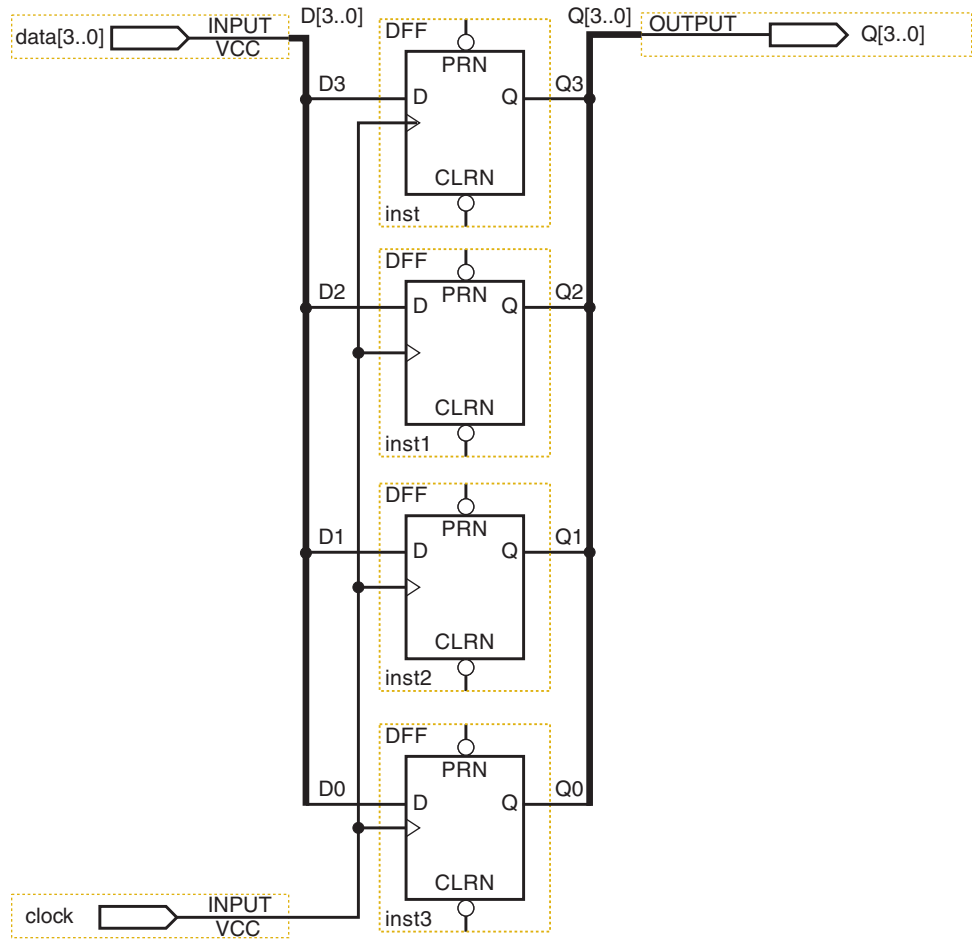
### Solution

Figure 5-71 shows the design file named reg4bit. The four-bit register was created using DFF primitives. From the Quartus menu, select:

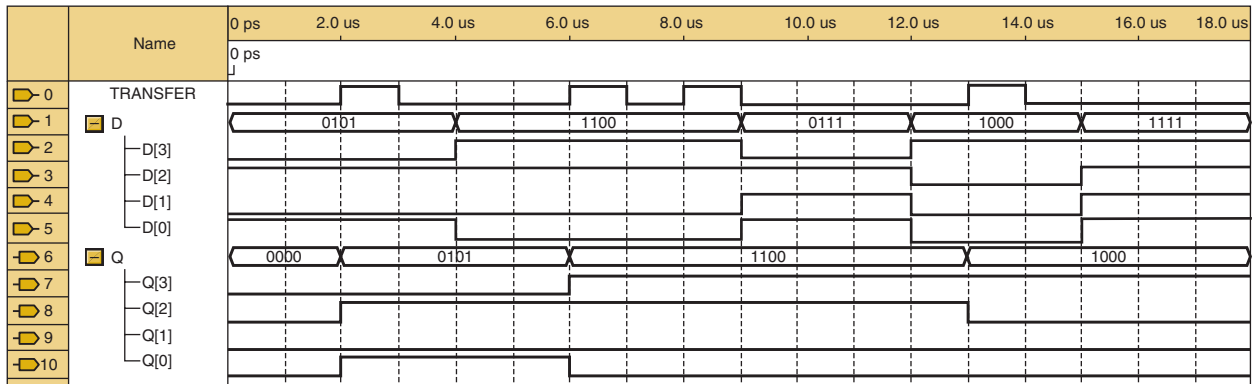
- >File
- >Create/Update
- >Create symbol files for current file.

This will produce the symbol shown in Figure 5-72. Notice that in this design file, we can name the pins in a way that makes sense for this particular application of the block symbol. The four D flip-flops will store the respective *D* input logic level on the PGT of TRANSFER. The simulation results are shown in Figure 5-73.

**FIGURE 5-71** Graphic connection of DFF primitives to form a four-bit register.



**FIGURE 5-72** Schematic using a block symbol for a four-bit DFF register.



**FIGURE 5-73** Functional simulation results for Example 5-22.

**OUTCOME  
ASSESSMENT  
QUESTION**

1. What are the names of the three Quartus libraries that contain useful functions for logic systems.

## 5-27 SEQUENTIAL CIRCUITS USING HDL

### OUTCOMES

Upon completion of this section, you will be able to:

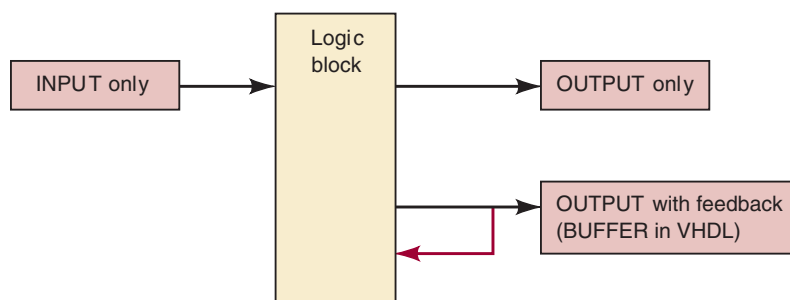
- Discriminate between combinational and sequential logic circuits.
- Describe how feedback creates sequential operation.
- Describe sequential logic circuits using AHDL and VHDL.

In Chapters 3 and 4, we used HDL to program simple combinational logic circuits. In this chapter, we have studied logic circuits that latch and clocked flip-flop circuits that sequence through various states in response to a clock edge. These latching and sequential circuits can also be implemented using PLDs and described using HDL.

Section 5-1 described a NAND gate latch. You will recall that the unique characteristic of this circuit is the fact that its outputs are cross-coupled back to its gates' inputs. This causes the circuit to respond differently depending on which state its output happens to be in. Describing circuits that have outputs that *feed back* to the input with Boolean equations or HDL involves using the output variables in the conditional portion of the description. To use Boolean equations means including output terms in the right-hand side of the equation. To use IF/THEN constructs means including output variables in the IF clause. Most PLDs have the ability to feed back the output signal to the input circuitry in order to accommodate latching action.

To write equations that use feedback, some languages, such as VHDL, can use a special designation for the output port. In these cases, the port bit is not only an output; it is an output with feedback. The difference is shown in Figure 5-74.

**FIGURE 5-74** Three input/output modes.



Rather than describing the operation of a latch using Boolean equations, let's try to think of a behavioral description of how the latch should operate. The situations we need to address are when SBAR is activated, when RBAR is activated, and when neither is activated. Recall that the invalid state occurs when both inputs are activated simultaneously. If we can describe a circuit that always recognizes one of the inputs as the winner when both are active, we can avoid the undesirable results of having an invalid input condition. To describe such a circuit, let's ask ourselves under what conditions

the latch should be set ( $Q = 1$ ). Certainly, the latch should be set if the SET input is active, but what about after SET goes back to its inactive level? How does the latch know to stay in the SET state? The description needs to use the *condition of the output now* to determine the *future* condition of the output. The following statement describes the conditions that should make the output HIGH on an S-R latch:

**IF SET is active, THEN Q should be HIGH.**

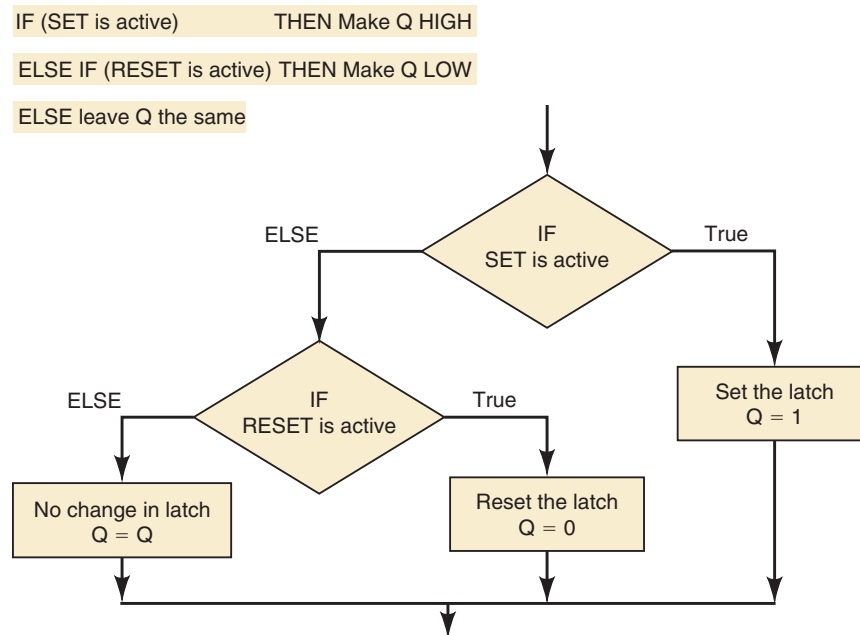
What conditions should make the output LOW?

**IF RESET is active, THEN Q should be LOW.**

What if neither input is activated? Then the output should remain the same and we can express this as  $Q = Q$ . This expression provides the feedback of the output state to be combined with input conditions for the purpose of deciding what happens next to the output.

What if both inputs are activated (i.e., the invalid input combination)? The structure of the IF/ELSE decision shown graphically in Figure 5-75 makes sure that the latch never tries to respond to both inputs. If the SET is active, regardless of what is on RESET, the output will be forced HIGH. The invalid input will always default to a set condition this way. The ELSIF clause is considered only when SET is not active. The use of the feedback term ( $Q = Q$ ) affects the operation (holding action) only when neither input is active.

**FIGURE 5-75** The logic of a behavioral description of an S-R latch.



When you design sequential circuits that feed the output value back to the inputs, it is possible to create an unstable system. A change in the output state might be fed back to the inputs, which changes the output state again, which feeds back to the inputs, which changes the output back again. This oscillation is obviously undesirable and so it is very important to make sure that no combination of inputs and outputs can make this happen. Careful analysis, simulation, and testing should be used to ensure that your circuit is stable under all conditions.

**EXAMPLE 5-23**

Describe an active-LOW input S-R latch with inputs named SBAR, RBAR, and one output named Q. It should follow the function table of a NAND latch (see Figure 5-6) and the invalid input combination should produce  $Q = 1$ .

- (a) Use AHDL.
- (b) Use VHDL.

**Solution**

- (a) Figure 5-76 shows a possible AHDL solution. Important items to note are:
  1. Q is defined as an OUTPUT, even though it is *fed back* in the circuit. AHDL allows outputs to be fed back into the circuit.
  2. The clause after IF will determine which output state occurs when both inputs are active (invalid state). In this code the SET command rules.
  3. To evaluate equality, the double equal sign is used. In other words, SBAR == 0 evaluates TRUE when SBAR is active (LOW).

```

SUBDESIGN fig5_76
(
    sbar, rbar      :INPUT;
    q               :OUTPUT;
)
BEGIN
    IF    sbar == 0 THEN q = VCC;    -- set or illegal command
    ELSIF rbar == 0 THEN q = GND;    -- reset
    ELSE                q = q;      -- hold
    END IF;
END;

```

**FIGURE 5-76** A NAND latch using AHDL.

- (d) Figure 5-77 shows a possible VHDL solution. Important items to note are:
  1. Q is defined as an OUT, even though it is fed back in the circuit. VHDL does not allow ports of mode OUT to be read within the hardware description code.
  2. A PROCESS describes what happens when the values in the sensitivity list (SBAR, RBAR) change state.

```

ENTITY fig5_77 IS
PORT (sbar, rbar      :IN BIT;
      q               :OUT BIT);
END fig5_77 ;

ARCHITECTURE behavior OF fig5_77 IS
BEGIN
PROCESS (sbar, rbar)
BEGIN
    IF sbar = '0' THEN q <= '1';    -- set or illegal command
    ELSIF rbar = '0' THEN q <= '0';  -- reset
    END IF;                          -- hold implied
END PROCESS;
END behavior;

```

**FIGURE 5-77** A NAND latch using VHDL.

3. The clause after IF will determine which output state occurs when both inputs are active (invalid state). In this code the SET command rules.
4. In VHDL, data latching (storage) is implied by intentionally leaving out the ELSE choice in an IF statement. The compiler will “understand” that when neither of the control inputs is active (LOW) the output will not change, which results in the current data bit being stored.

## The D Latch

The transparent D latch can also be easily implemented with HDLs. Quartus has a library primitive called LATCH that is available. The library contains functional definitions of digital components that are available to construct logic circuits. Primitive devices are the fundamental building blocks such as the various types of gates, flip-flops, and latches. The AHDL module below illustrates using this LATCH primitive. A latch named *q* is declared in the VARIABLE section. The latch’s output is automatically connected to an output port since *q* is also declared as an output in the SUBDESIGN. All that is needed is to connect the primitive’s enable (*.ena*) and data (*.d*) ports (see the list of standard ports for primitive memory elements in Table 5-2) to the appropriate module input signals. The VHDL module also shown below is a behavioral description of a D latch function. The VHDL language typically handles storage of data bits in a different fashion. Instead of literally declaring flip-flops or latches, the memory element is implied by an incomplete IF statement (notice the missing ELSE clause in the IF statement of this example). The VHDL compiler will interpret a false test result for the IF as a no change condition for the signal assignment to *q*, resulting in the creation of a memory element.

AHDL D latch

```
SUBDESIGN dlatch_ahdl
(enable, din      :INPUT;
 q               :OUTPUT;)

VARIABLE
 q               :LATCH;
BEGIN
 q.ena = enable;
 q.d = din;
END;
```

VHDL D latch

```
ENTITY dlatch_vhdl IS
PORT (enable, din      :IN BIT;
 q               :OUT BIT);
END dlatch_vhdl;

ARCHITECTURE v OF dlatch_vhdl IS
BEGIN
 PROCESS (enable, din)
 BEGIN
 IF enable = '1' THEN
 q <= din;
 END IF;
 END PROCESS;
END v;
```

**TABLE 5-2** Altera primitive port identifiers.

Standard Part Function	Primitive Port Name
Clock input	clk
Asynchronous preset (active-LOW)	prn
Asynchronous clear (active-LOW)	clrn
J, K, S, R, D inputs	j, k, s, r, d
Level triggered ENABLE input	ena
Q output	q

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is the distinguishing hardware characteristic of latching logic circuits?
2. What is the major characteristic of sequential circuits?

## 5-28 EDGE-TRIGGERED DEVICES

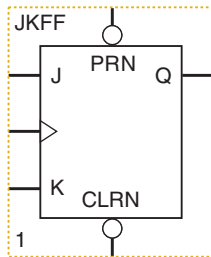
### OUTCOMES

Upon completion of this section, you will be able to:

- Define the term *logic primitives* as used by Quartus.
- Locate lists of primitives in the Quartus system.
- Identify features and syntax details that allow the use of primitives.
- Use edge-triggered primitives in AHDL.
- Describe edge-triggered flip-flops in VHDL.

Earlier in this chapter, we introduced edge-triggered devices whose outputs respond to the inputs when the clock input sees an “edge.” An edge simply means a transition from HIGH to LOW, or vice versa, and is often referred to as an **event**. If we are writing statements in the code that are concurrent, how can outputs change only when a clock input detects an edge event? The answer to this question differs substantially, depending on the HDL you use. In this section, we want to concentrate on creating clocked logic circuits in their simplest form using HDL. We will use J-K flip-flops to correlate with many of the examples found earlier in this chapter.

The J-K flip-flop is a standard building block of clocked (sequential) logic circuits known as a **logic primitive**. In its most common form, it has five inputs and one output, as shown in Figure 5-78. The input/output names can be standardized to allow us to refer to the connections of this primitive or fundamental circuit. The actual operation of the primitive circuit is defined in a library of components that is available to the HDL compiler as it generates a circuit from our description. AHDL uses logic primitives to describe flip-flop operation. VHDL offers something similar, but it also allows the designer to describe the clocked logic circuit’s operation explicitly in the code.



**FIGURE 5-78** J-K flip-flop logic primitive.

### AHDL FLIP-FLOPS

A flip-flop can be used in AHDL by declaring a register (even one flip-flop is called a register). Several different types of register primitives are available for use in AHDL, including JKFF, DFF, SRFF, and latch. Each different type of register primitive has its own official names (according to Altera software) for the ports of these primitives. These can be found by using the HELP menu in the ALTERA software and looking under Primitives. Table 5-2 listed some of these names. Registers that use these primitives are declared in the VARIABLE section of the code. The register is given an instance name, just as we have named intermediate variables or buried nodes in previous examples. Instead of declaring it as a node, however, it is declared by the type of the register primitive. For example, a J-K flip-flop can be declared as:

```
VARIABLE
    ff1 :JKFF;
```



The instance name is *ff1* (which you can make up) and the register primitive type is *JKFF* (which Altera requires you to use). Once you have declared a register, it is connected to the other logic in the design using its standard port names. The ports (or pins) on the flip-flop are referred to using the instance name, with a dot extension that designates the particular input or output. An example for a J-K flip-flop in AHDL is shown in Figure 5-79. Notice that we have made up our own input/output names for this SUBDESIGN in order to distinguish them from the primitive port names. The single flip-flop is declared on line 8, as previously described. The J input or port for this device is then labeled *ff1.j*, the K input is *ff1.k*, the clock input is *ff1.clk*, and so on. Each of the given port assignment statements will make the needed wiring connections for this design block. The *prn* and *clrn* ports are both active-LOW, asynchronous controls such as those commonly found on a standard flip-flop. In fact, these asynchronous controls on an FF primitive can be used to implement an S-R latch more efficiently than the code in Figure 5-76. The *prn* and *clrn* controls are optional in AHDL and will default to a disabled condition (at a logic 1) if they are omitted from the logic section. In other words, if lines 10 and 11 were deleted, the *prn* and *clrn* ports of *ff1* would automatically be tied to  $V_{CC}$ .

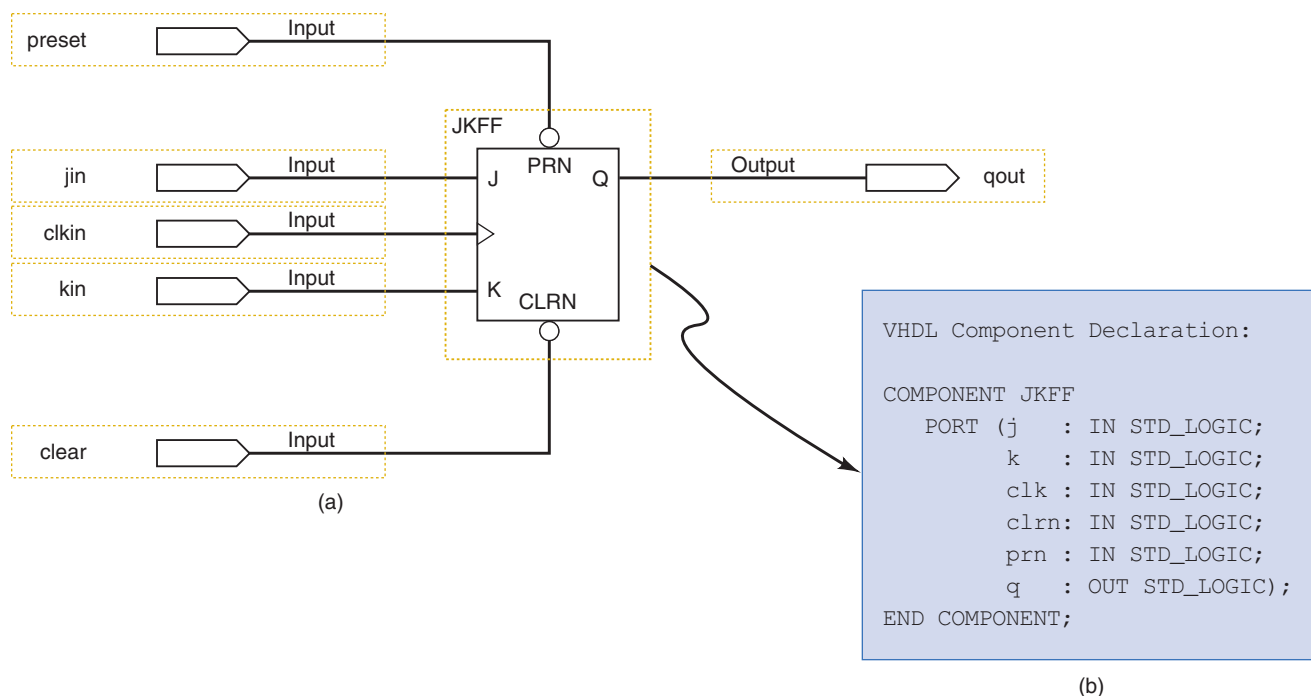
```

1  %      J-K flip-flop circuit      %
2  SUBDESIGN fig5_79
3  (
4      jin, kin, clkin, preset, clear :INPUT;
5      qout                          :OUTPUT;
6  )
7  VARIABLE
8  ff1      :JKFF;      -- define this flip-flop as a JKFF type
9  BEGIN
10     ff1.prn = preset; -- these are optional and default to vcc
11     ff1.clrn = clear;
12     ff1.j = jin;      -- connect primitive to the input signal
13     ff1.k = kin;
14     ff1.clk = clkin;
15     qout = ff1.q;    -- connect the output pin to the primitive
16  END;
```

**FIGURE 5-79** Single J-K flip-flop using AHDL.

## VHDL LIBRARY COMPONENTS

The Altera software comes with some extensive libraries of components and primitives that can be used by a designer. The graphic description of a JKFF component in the Altera library is shown in Figure 5-80(a). After placing the component on the worksheet, each of its ports is connected to inputs and outputs of the module. This same concept can be implemented in VHDL using a library component. The inputs and outputs of these library components can be found by looking under the HELP/Primitives menu. Figure 5-80(b) shows the VHDL **COMPONENT** declaration for a J-K flip-flop primitive. The key things to notice are the name of the component (JKFF) and the names of the ports. They are the same names as those used in the graphic symbol of Figure 5-80(a). Also, notice that the *type* of each input



**FIGURE 5-80** (a) Graphic representation using a component. (b) VHDL component declaration.

and output variable is `STD_LOGIC`. This is one of the IEEE standard data types defined in the library and used by many components in the library.

Figure 5-81 uses a JKFF component from the library in VHDL to create a circuit equivalent to the graphic design of Figure 5-80(a). The first two lines tell the compiler to use the IEEE library to find the definitions of the `std_logic` data types. The next two lines tell the compiler that it should look in the Altera library for any standard library components that are used later on in the code. The module inputs and outputs are declared as they

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;           --defines std_logic types
LIBRARY altera;
USE altera.maxplus2.all;              -- provides standard components

ENTITY fig5_81 IS
  PORT( clk, jin, kin, preset, clear    :IN std_logic;
        qout                          :OUT std_logic);
END fig5_81;

ARCHITECTURE a OF fig5_81 IS
  BEGIN
    ff1: JKFF PORT MAP (
      clk => clk,
      j   => jin,
      k   => kin,
      prn => preset,
      clrn => clear,
      q   => qout);
  END a;

```

**FIGURE 5-81** A J-K flip-flop using VHDL.

were in previous examples, except that the type is now `STD_LOGIC` rather than `BIT`. This is because the module port types must match the component port types. Within the architecture section, a name (`ff1`) is given to this instance of the component `JKFF`. The keywords **PORT MAP** are followed by a list of all the connections that must be made to the component ports. Notice that the component ports (e.g., `clk`) are listed on the left of the symbol `=>` and the objects they are connected to (e.g., `clkin`) are listed on the right.

## VHDL FLIP-FLOPS

Now that we have seen how to use standard components that are available in the library, let's look next at how to create our own component that can be used over and over again. For the sake of comparison, we will describe the VHDL code for a J-K flip-flop that is identical to the library component `JKFF`.

VHDL was created as a very flexible language and it allows us to define the operation of clocked devices explicitly in the code, without relying on logic primitives. The key to edge-triggered sequential circuits in VHDL is the `PROCESS`. As you recall, this keyword is followed by a sensitivity list in parentheses. Whenever a variable in the sensitivity list changes state, the code in the process block determines how the circuit should respond. This is very much like a flip-flop that does nothing until the clock input changes state, at which time it evaluates its inputs and updates its outputs. If the flip-flop needs to respond to inputs other than the clock (e.g., preset and clear), they can be added to the sensitivity list. The code in Figure 5-82 demonstrates a J-K flip-flop written in VHDL.

```

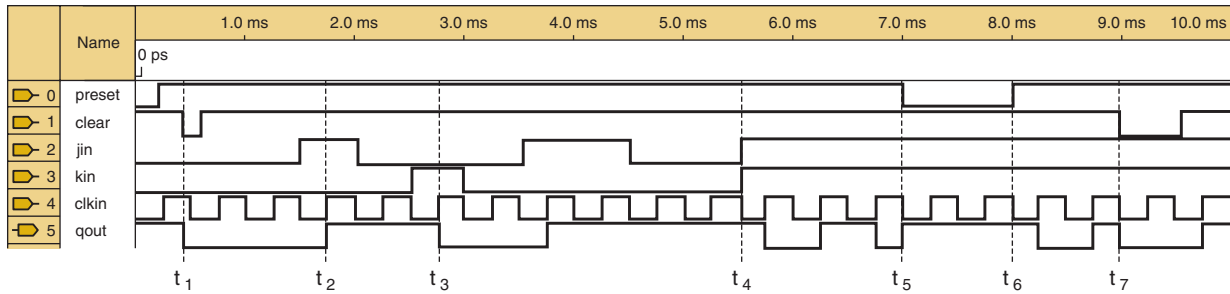
1  -- J-K Flip-Flop Circuit
2  ENTITY jk IS
3  PORT(
4      clk, j, k, prn, clrn :IN BIT;
5      q                    :OUT BIT);
6  END jk;
7
8  ARCHITECTURE a OF jk IS
9  SIGNAL qstate :BIT;
10 BEGIN
11     PROCESS(clk, prn, clrn) -- respond to any of these signals
12     BEGIN
13         IF prn = '0' THEN qstate <= '1'; -- asynch preset
14         ELSIF clrn = '0' THEN qstate <= '0'; -- asynch clear
15         ELSIF clk = '1' AND clk'EVENT THEN -- on PGT clock edge
16             IF j = '1' AND k = '1' THEN qstate <= NOT qstate;
17             ELSIF j = '1' AND k = '0' THEN qstate <= '1';
18             ELSIF j = '0' AND k = '1' THEN qstate <= '0';
19             END IF;
20         END IF;
21     END PROCESS;
22     q <= qstate; -- update output pin
23 END a;
```

**FIGURE 5-82** Single J-K flip-flop using VHDL.

On line 9 of the figure, a signal is declared with a name of *qstate*. Signals can be thought of as wires that connect two points in the circuit description, but they also have characteristics of implied “memory.” This means that once a value is assigned to the signal, it will stay at that value until a different value is assigned in the code. In VHDL, a VARIABLE is often used to implement this feature of “memory,” but variables must be declared and used within the same description block. In this example, if *qstate* were declared as a VARIABLE, it would need to be declared within the PROCESS (after line 11) and must be assigned to *q* before the end of the PROCESS (line 21). Our example uses a SIGNAL that can be declared and used throughout the architecture description.

Notice that the PROCESS sensitivity list contains the asynchronous preset and clear signals. The flip-flop must respond to these inputs as soon as they are asserted (LOW), and these inputs should override the *J*, *K*, and clock inputs. To accomplish this, we can use the sequential nature of the IF/ELSE constructs. First, the PROCESS will describe what happens only when one of the three signals—*clk*, *prn*, or *clrn*—changes state. The highest priority input in this example is *prn* because it is evaluated first in line 13. If it is asserted, *qstate* will be set HIGH and the other inputs will not even be evaluated because they are in the else branch of the decision. If *prn* is HIGH, *clrn* will be evaluated in line 14 to see if it is LOW. If it is, the flip-flop will be cleared and nothing else will be evaluated in the PROCESS. Line 15 will be evaluated only if both *prn* and *clrn* are HIGH. The term *clk*’ EVENT in line 15 evaluates as TRUE only if there has been a transition on *clk*. Because *clk* = ‘1’ must be TRUE also, this condition responds only to a rising edge transition on the clock. The next three conditions of lines 16, 17, and 18 are evaluated only following a rising edge on *clk* and serve to update the flip-flop’s state. In other words, they are nested within the ELSIF statement of line 15. Only the JK input commands for toggle, set, and reset are evaluated by the IF/ELSIF on lines 16–18. Of course, with a JK there is a fourth command, hold. The “missing” ELSE condition will be interpreted by VHDL as an implied memory device that will then hold the PRESENT state if none of the given JK conditions is TRUE. Note that each IF/ELSIF structure has its own END IF statement. Line 19 ends the decision structure that decides to set, clear, or toggle. Line 20 ends the IF/ ELSIF structure that decides among the preset, clear, and clock edge responses. As soon as the PROCESS ends, the flip-flop’s state is transferred to the output port *q*.

Regardless of whether you develop your description in AHDL or VHDL, the circuit’s proper operation can be verified using a simulator. The most important and challenging part of verification using a simulator is creating a set of hypothetical input conditions that will prove that the circuit does everything it is intended to do. There are many ways to do this, and it is up to the designer to decide which way is best. The simulation used to verify the operation of the JKFF primitive is shown in Figure 5-83. The *preset* input is initially activated and then, at  $t_1$ , the *clear* input is activated. These tests ensure that *preset* and *clear* are operating asynchronously. The *jin* input is HIGH at  $t_2$  and *kin* is HIGH at  $t_3$ . In between these points, the inputs on *jin* and *kin* are both LOW. This portion of the simulation tests the synchronous modes of set, hold, and reset. Starting at  $t_4$ , the toggle command is tested with *jin* = *kin* = 1. Notice at  $t_5$ , *preset* is asserted (LOW) to test whether *preset* overrides the toggle command. After  $t_6$ , the output starts toggling again, and at  $t_7$ , the *clear* input is shown overriding the synchronous inputs. Testing of all modes of operation and the interaction of various controls is very important when you are simulating.



**FIGURE 5-83** Simulation of the J-K flip-flop.

### OUTCOME ASSESSMENT QUESTIONS

1. What is a logic primitive?
2. What does the designer need to know in order to use a logic primitive?
3. In the Altera system, where can you find information on primitives and library functions?
4. What is the key VHDL element that allows the explicit description of clocked logic circuits?
5. Which library defines the `std_logic` data types?
6. Which library defines the logic primitives and common components?

## 5-29 HDL CIRCUITS WITH MULTIPLE COMPONENTS

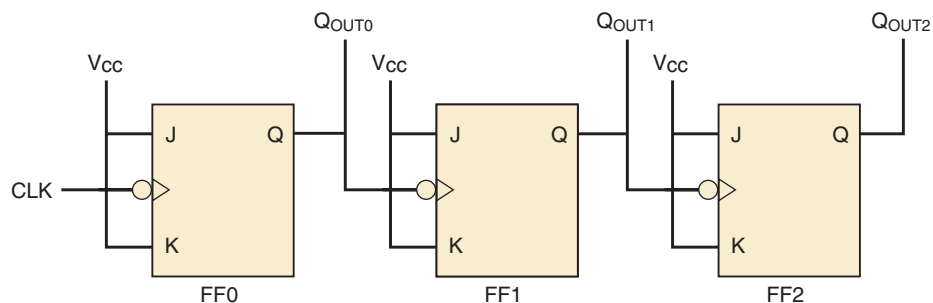
### OUTCOMES

*Upon completion of this section, you will be able to:*

- Define and declare components in HDL.
- Interconnect components using HDL.
- Create and use multiple instances of components in an HDL design.

We began this chapter by studying latches. Latches were used to make flip-flops and flip-flops were used to make many circuits, including binary counters. A graphic description (logic diagram) of a simple binary up counter is shown in Figure 5-84. This circuit is functionally the same as Figure 5-48, which was drawn with the LSB on the right to make it easier to visualize the numeric value of the binary count. The circuit has been redrawn here to show the signal flow in the more conventional format, with inputs on the left and outputs on the right. Notice that these logic symbols are negative

**FIGURE 5-84** A three-bit binary counter.



edge-triggered. These flip-flops also do not have asynchronous inputs `prn` or `clrn`. Our goal is to describe this counter circuit using HDL by interconnecting three instances of the same J-K flip-flop component.

## AHDL RIPPLE-UP COUNTER

A text-based description of this circuit requires three of the same type of flip-flop, just like the graphic description. Refer to Figure 5-85. On line 8 of the figure, bit array notation is used to declare a register of three J-K flip-flops. The name of this register is `q`, just like the name of the output port. AHDL can interpret this to mean that the output of each flip-flop should be connected to the output port. Each bit of the array `q` has all the attributes of a JKFF primitive. AHDL is very flexible in its use of indexed sets like this. As an example of the use of this set notation, notice how all the J and K inputs for all the flip-flops are tied to VCC in lines 11 and 12. If the flip-flops had been named A, B, and C rather than using a bit array, then individual assignments would be necessary for each J and K input, making the code much longer. Next, the key interconnections are made between the flip-flops to make this a ripple-up counter. The clock signal is inverted and assigned to FF0 clock input (line 13), the Q output of FF0 is inverted and assigned to FF1 clock input (line 14), and so on, forming a ripple counter.

```

1  % MOD 8 ripple up counter. %
2  SUBDESIGN fig5_85
3  (
4      clock                :INPUT;
5      q[2..0]              :OUTPUT;
6  )
7  VARIABLE
8      q[2..0]:JKFF;        -- defines three J-K FFs
9  BEGIN
10                                     -- note: prn, clrn default to vcc!
11      q[2..0].j = VCC;        -- toggle mode J=K=1 for all FFs
12      q[2..0].k = VCC;
13      q[0].clk = !clock;
14      q[1].clk = !q[0].q;
15      q[2].clk = !q[1].q;    -- connect clocks in ripple form
16  END;
```

FIGURE 5-85 MOD-8 ripple counter in AHDL.

## VHDL RIPPLE-UP COUNTER

We described in Figure 5-82 the VHDL code for a positive-edge-triggered JKFF with preset and clear controls. The counter in Figure 5-84 is negative-edge-triggered and does not require asynchronous preset or clear. Our goal now is to write the VHDL code for one of these flip-flops, represent three instances of the same flip-flop, and interconnect the ports to create the counter.

We will start by looking at the VHDL description in Figure 5-86, starting at line 18. This module of VHDL code is describing the operation of a

single J-K flip-flop component. The name of the component is `neg_jk` (line 18) and it has inputs `clk`, `j`, and `k` (line 19) and output `q` (line 20). A signal named `qstate` is used to hold the state of the flip-flop and connect it to the `q` output. On line 25, the `PROCESS` has only `clk` in its sensitivity list, so it only responds to changes in the `clk` (PGTs and NGTs). The statement that makes this flip-flop negative-edge-triggered is on line 27. `IF (clk'EVENT AND clk = '0')` is true, then a `clk` edge has just occurred and `clk` is now LOW, meaning it must have been an NGT of `clk`. The `IF/ELSE` decisions that follow implement the four states of a J-K flip-flop.

```

1  ENTITY fig5_86 IS
2  PORT (   clock      :IN BIT;
3         qout       :BUFFER BIT_VECTOR (2 DOWNTO 0));
4  END fig5_86;
5  ARCHITECTURE counter OF fig5_86 IS
6     SIGNAL high      :BIT;
7     COMPONENT neg_jk
8     PORT (   clk, j, k  :IN BIT;
9           q          :OUT BIT);
10    END COMPONENT;
11  BEGIN
12    high <= '1';      -- connect to Vcc
13    ff0: neg_jk  PORT MAP (j => high, k => high, clk => clock,  q => qout(0));
14    ff1: neg_jk  PORT MAP (j => high, k => high, clk => qout(0), q => qout(1));
15    ff2: neg_jk  PORT MAP (j => high, k => high, clk => qout(1), q => qout(2));
16  END counter;
17
18  ENTITY neg_jk IS
19  PORT (   clk, j, k    :IN BIT;
20         q             :OUT BIT);
21  END neg_jk;
22  ARCHITECTURE simple OF neg_jk IS
23     SIGNAL qstate     :BIT;
24  BEGIN
25     PROCESS (clk)
26     BEGIN
27         IF (clk'EVENT AND clk = '0') THEN
28             IF j = '1' AND k = '1' THEN qstate <= NOT qstate; -- toggle
29             ELSIF j = '1' AND k = '0' THEN qstate <= '1';    -- set
30             ELSIF j = '0' AND k = '1' THEN qstate <= '0';    -- reset
31             END IF;
32         END IF;
33     END PROCESS;
34     q <= qstate      -- connect flip-flop state to output
35  END simple;;

```

**FIGURE 5-86** MOD-8 ripple counter in VHDL.

Now that we know how one flip-flop named `neg_jk` works, let's see how we can use it three times in a circuit and hook all the ports together. Line 1 defines the `ENTITY` that will make up the three-bit counter. Lines 2–3 contain the definitions of the inputs and outputs. Notice that the outputs are in the form of a three-bit array (bit vector). On line 6 the `SIGNAL high` can be thought of as a wire used to connect points in the circuit to  $V_{CC}$ . Line 7 is very important because this is where we declare that we plan to use a component in our design whose name is `neg_jk`. In this example, the actual

code is written at the bottom of the page, but it could be in a separate file or even in a library. This declaration tells the compiler all the important facts about the component and its port names.

The final part of the description is the concurrent section of lines 12–15. First, the signal *high* is connected to  $V_{CC}$  on line 12. The next three lines are instantiations of the flip-flop components. The three instances are named ff0, ff1, and ff2. Each instance is followed by a PORT MAP which lists each port of the component and describes what it is connected to in the module.

Connecting components together using HDL is not difficult, but it is very tedious. As you can see, the file for even a very simple circuit can be quite long. This method of describing circuits is referred to as the **structural level of abstraction**. It requires the designer to account for each pin of each component and define signals for each wire that is to interconnect the components. People who are accustomed to using logic diagrams to describe circuits generally find it easy to understand the structural level, but not as easy to read at a glance as the equivalent logic circuit diagram. In fact, it is safe to say that if the structural level of description was all that was available, most people would prefer using graphic descriptions (schematics) rather than HDL. The real advantage of HDL is found in the use of higher levels of abstraction and the ability to tailor components to fit the needs of the project exactly. We will explore the use of these methods, as well as graphical tools to connect modules, in the following chapters.

### OUTCOME ASSESSMENT QUESTIONS

1. Can the same component be used more than once in the same circuit?
2. In AHDL, where are multiple instances of a component declared?
3. How do you distinguish between multiple instances of a component?
4. In AHDL, what operator is used to “connect” signals?
5. In VHDL, what serves as “wires” that connect components?
6. In VHDL, what keyword identifies the section of code where connections are specified for instances of components?

## SUMMARY

1. A flip-flop is a logic circuit with a memory characteristic such that its  $Q$  and  $\bar{Q}$  outputs will go to a new state in response to an input pulse and will remain in that new state after the input pulse is terminated.
2. A NAND latch and a NOR latch are simple FFs that respond to logic levels on their SET and RESET inputs.
3. Clearing (resetting) a FF means that its output ends up in the  $Q = 0/\bar{Q} = 1$  state. Setting a FF means that it ends up in the  $Q = 1/\bar{Q} = 0$  state.
4. Clocked FFs have a clock input ( $CLK$ ,  $CP$ ,  $CK$ ) that is edge-triggered, meaning that it triggers the FF on a positive-going transition (PGT) or a negative-going transition (NGT).
5. Edge-triggered (clocked) FFs can be triggered to a new state by the active edge of the clock input according to the state of the FF’s synchronous control inputs ( $S$ ,  $R$  or  $J$ ,  $K$  or  $D$ ).
6. Most clocked FFs also have asynchronous inputs that can set or clear the FF independently of the clock input.



7. The *D* latch is a modified NAND latch that operates like a *D* flip-flop except that it is not edge-triggered.
8. Some of the principal uses of FFs include data storage and transfer, data shifting, counting, and frequency division. They are used in sequential circuits that follow a predetermined sequence of states.
9. A one-shot (OS) is a logic circuit that can be triggered from its normal resting state ( $Q = 0$ ) to its triggered state ( $Q = 1$ ), where it remains for a time interval proportional to an *RC* time constant.
10. Circuits that have a Schmitt-trigger type of input will respond reliably to slow-changing signals and will produce outputs with clean, sharp edges.
11. A variety of circuits can be used to generate clock signals at a desired frequency, including Schmitt-trigger oscillators, a 555 timer, and a crystal-controlled oscillator.
12. A complete summary of the various types of FFs can be found inside the back cover.
13. Programmable logic devices can be programmed to operate as latching circuits and sequential circuits.
14. Fundamental building blocks called logic primitives are available in the Altera library to help implement larger systems.
15. Clocked flip-flops are available as logic primitives.
16. VHDL code can be written to describe clocked logic explicitly without using logic primitives.
17. VHDL allows HDL files to be used as components in larger systems. Prefabricated components are available in the Altera library.
18. HDL can be used to describe interconnected components in a manner much like a graphic schematic capture tool.

## IMPORTANT TERMS

---

flip-flop	metastable states	shift register
feedback	setup time, $t_S$	frequency division
SET (states/inputs)	hold time, $t_H$	binary counter
CLEAR	clocked S-R flip-flop	state table
(states/inputs)	trigger	state transition
RESET	pulse-steering circuit	diagram
(states/inputs)	edge-detector circuit	MOD number
NAND gate latch	clocked J-K flip-flop	Schmitt-trigger
S-R latch	toggle mode	circuit
contact bounce	clocked D flip-flop	one-shot (OS)
NOR gate latch	parallel data transfer	quasi-stable state
pulses	D latch	nonretriggerable OS
clock	asynchronous inputs	retriggerable OS
positive-going	override inputs	astable or free-
transition (PGT)	propagation delay	running
negative-going	sequential circuits	multivibrator
transition (NGT)	registers	555 timer
clocked flip-flop	data transfer	duty cycle
period	synchronous	crystal-controlled
frequency	transfer	clock generator
edge-triggered	asynchronous (jam)	clock skew
control inputs	transfer	primitives
synchronous control	jam transfer	maxplus2
inputs	serial data transfer	megafuction

---

library of  
parameterized  
modules (LPMs)  
EVENT

logic primitive  
COMPONENT  
PORT MAP  
nested

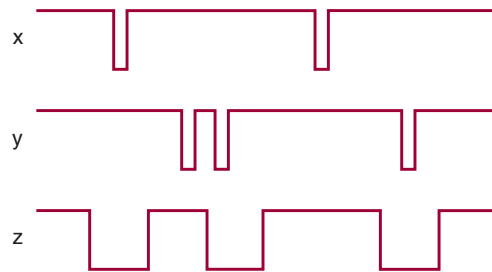
structural level of  
abstraction

## PROBLEMS

### SECTIONS 5-1 TO 5-3

- B** 5-1\* Assuming that  $Q = 0$  initially, apply the  $x$  and  $y$  waveforms of Figure 5-87 to the SET and RESET inputs of a NAND latch, and determine the  $Q$  and  $\bar{Q}$  waveforms.

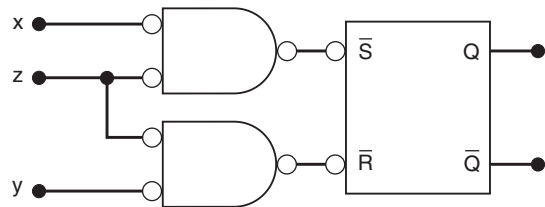
**FIGURE 5-87** Problems 5-1 to 5-3.



- B** 5-2. Invert the  $x$  and  $y$  waveforms of Figure 5-87, apply them to the SET and RESET inputs of a NOR latch, and determine the  $Q$  and  $\bar{Q}$  waveforms. Assume that  $Q = 0$  initially.

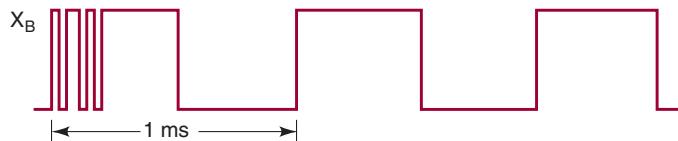
- 5-3\* The waveforms of Figure 5-87 are connected to the circuit of Figure 5-88. Assume that  $Q = 0$  initially, and determine the  $Q$  waveform.

**FIGURE 5-88** Problem 5-3.



- D** 5-4. Modify the circuit of Figure 5-9 to use a NOR gate latch.
- D** 5-5. Modify the circuit of Figure 5-12 to use a NAND gate latch.
- T** 5-6\* Refer to the circuit of Figure 5-13. A technician tests the circuit operation by observing the outputs with a storage oscilloscope while the switch is moved from  $A$  to  $B$ . When the switch is moved from  $A$  to  $B$ , the scope display of  $X_B$  appears as shown in Figure 5-89. What circuit fault could produce this result? (*Hint*: What is the function of the NAND latch?)

**FIGURE 5-89** Problem 5-6.

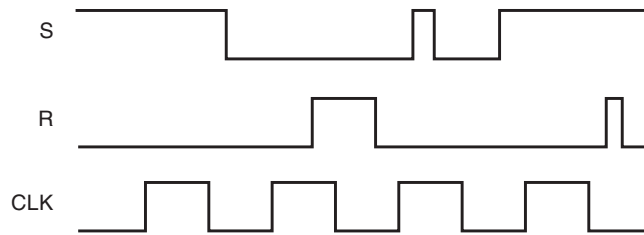


\*Answers to problems marked with an asterisk can be found in the back of the text.

## SECTIONS 5-4 THROUGH 5-6

- B** 5-7. A certain clocked FF has minimum  $t_S = 20$  ns and  $t_H = 5$  ns. How long must the control inputs be stable prior to the active clock transition?
- B** 5-8. Apply the  $S$ ,  $R$ , and  $CLK$  waveforms of Figure 5-20 to the FF of Figure 5-21, and determine the  $Q$  waveform.
- B** 5-9\* Apply the waveforms of Figure 5-90 to the FF of Figure 5-20 and determine the waveform at  $Q$ . Repeat for the FF of Figure 5-21. Assume  $Q = 0$  initially.

FIGURE 5-90 Problem 5-9.



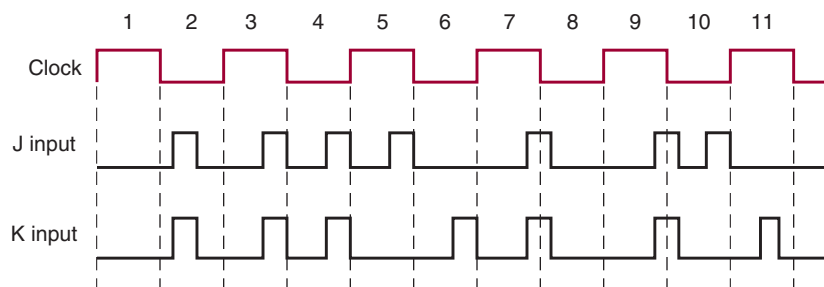
- 5-10. Draw the following digital pulse waveforms. Label  $t_r$ ,  $t_f$ , and  $t_w$ , leading edge, and trailing edge.
- (a) A negative TTL pulse with  $t_r = 20$  ns,  $t_f = 5$  ns, and  $t_w = 50$  ns.
- (b) A positive TTL pulse with  $t_r = 5$  ns,  $t_f = 1$  ns, and  $t_w = 25$  ns.
- (c) A positive pulse with  $t_w = 1$  ms whose leading edge occurs every 5 ms. Give the frequency of this waveform.

## SECTION 5-7

- B** 5-11\* Apply the  $J$ ,  $K$ , and  $CLK$  waveforms of Figure 5-24 to the FF of Figure 5-25. Assume that  $Q = 1$  initially, and determine the  $Q$  waveform.
- D** 5-12. (a)\* Show how a J-K flip-flop can operate as a *toggle* FF (changes states on each clock pulse). Then apply a 10-kHz clock signal to its  $CLK$  input and determine the waveform at  $Q$ .
- (b) Connect  $Q$  from this FF to the  $CLK$  input of a second J-K FF that also has  $J = K = 1$ . Determine the frequency of the signal at this FF's output.
- B** 5-13. The waveforms shown in Figure 5-91 are to be applied to two different FFs:
- (a) positive-edge-triggered J-K
- (b) negative-edge-triggered J-K

Draw the  $Q$  waveform response for each of these FFs, assuming that  $Q = 0$  initially. Assume that each FF has  $t_H = 0$ .

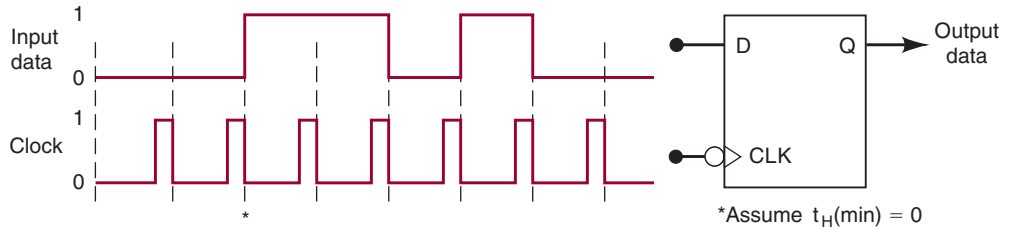
FIGURE 5-91 Problem 5-13.



**SECTION 5-8**

5-14. A D FF is sometimes used to *delay* a binary waveform so that the binary information appears at the output a certain amount of time after it appears at the *D* input.

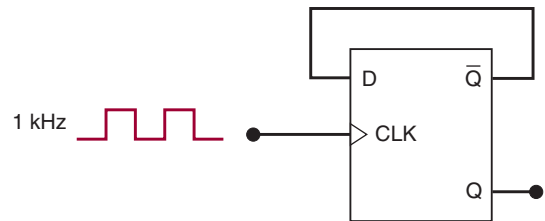
- (a)\* Determine the *Q* waveform in Figure 5-92, and compare it with the input waveform. Note that it is delayed from the input by one clock period.
- (b) How can a delay of two clock periods be obtained?



**FIGURE 5-92** Problem 5-14.

- B** 5-15. (a) Apply the *S* and *CLK* waveforms of Figure 5-90 to the *D* and *CLK* inputs of a D FF that triggers on PGTs. Then determine the waveform at *Q*.
- (b) Repeat using the *C* waveform of Figure 5-90 for the *D* input.
- B** 5-16\* An edge-triggered D flip-flop can be made to operate in the toggle mode by connecting it as shown in Figure 5-93. Assume that  $Q = 0$  initially, and determine the *Q* waveform.

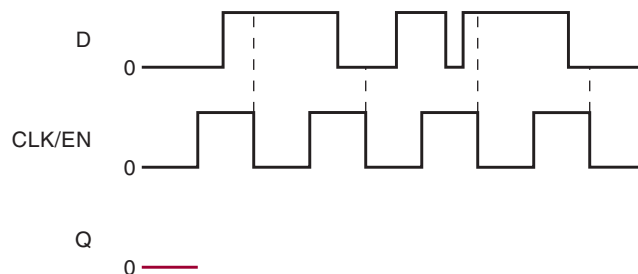
**FIGURE 5-93** D flip-flop connected to toggle (Problem 5-16).



**SECTION 5-9**

- B** 5-17. (a) Apply the *S* and *CLK* waveforms of Figure 5-90 to the *D* and *EN* inputs of a D latch, respectively, and determine the waveform at *Q*.
- (b) Repeat using the *C* waveform applied to *D*.
- 5-18. Compare the operation of the D latch with a negative-edge-triggered D flip-flop by applying the waveforms of Figure 5-94 to each and determining the *Q* waveforms.

**FIGURE 5-94** Problem 5-18.

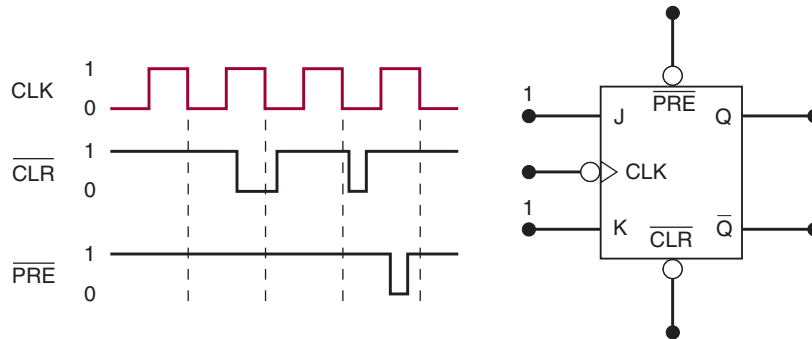


- 5-19. In Problem 5-16, we saw how an edge-triggered D flip-flop can be operated in the toggle mode. Explain why this same idea will not work for a D latch.

### SECTION 5-10

- B** 5-20. Determine the  $Q$  waveform for the FF in Figure 5-95. Assume that  $Q = 0$  initially, and remember that the asynchronous inputs override all other inputs.

**FIGURE 5-95** Problem 5-20.



- B, N** 5-21\* Apply the  $CLK$ ,  $\overline{PRE}$ , and  $\overline{CLR}$  waveforms of Figure 5-33 to a positive-edge-triggered D flip-flop with active-LOW asynchronous inputs. Assume that  $D$  is kept HIGH and  $Q$  is initially LOW. Determine the  $Q$  waveform.
- B** 5-22. Apply the waveforms of Figure 5-95 to a D flip-flop that triggers on NGTs and has active-LOW asynchronous inputs. Assume that  $D$  is kept LOW and that  $Q$  is initially HIGH. Draw the resulting  $Q$  waveform.

### SECTION 5-11

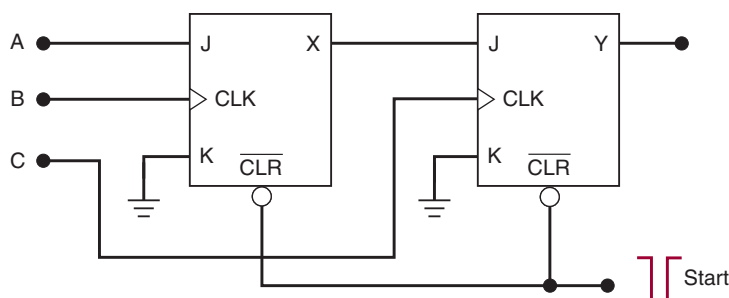
- B** 5-23. Use the Texas Instrument website to look up the 74ALS74A DFF.
- How long can it take for the  $Q$  output of a 74ALS74A to switch from 0 to 1 in response to an active  $CLK$  transition?
  - How long does the  $D$  input need to be stable before the active clock edge on the 74ALS74A?
  - What is the narrowest pulse that can be applied to the  $PRE$  of a 74ALS74A FF?
- B** 5-24. Use the Texas Instrument website to look up the 74ALS112A DFF.
- How long does it typically take to asynchronously clear a 74ALS112?
  - How long maximum does it take to asynchronously set a 74ALS112?
  - What is the shortest acceptable interval between active clock transitions for a 74ALS74A?
  - The  $J$  input of a 74ALS112A goes HIGH 15 ns before the active clock edge. The  $K$  input has been at 0. Will the flip-flop be reliably set?
  - How long does it take (after the clock edge) to synchronously store a 1 in a cleared 74ALS74A D flip-flop?

### SECTIONS 5-14 AND 5-15

- D** 5-25\* Modify the circuit of Figure 5-39 to use a J-K flip-flop.

- D 5-26. In the circuit of Figure 5-96, inputs  $A$ ,  $B$ , and  $C$  are all initially LOW. Output  $Y$  is supposed to go HIGH only when  $A$ ,  $B$ , and  $C$  go HIGH in a certain sequence.
  - (a) Determine the sequence that will make  $Y$  go HIGH.
  - (b) Explain why the START pulse is needed.
  - (c) Modify this circuit to use D FFs.

FIGURE 5-96 Problem 5-26.



SECTIONS 5-16 AND 5-17

- D 5-27\* (a) Draw a circuit diagram for the synchronous parallel transfer of data from one three-bit register to another using J-K flip-flops.
  - (b) Repeat for asynchronous parallel transfer.
- D 5-28. A *recirculating* shift register is a shift register that keeps the binary information circulating through the register as clock pulses are applied. The shift register of Figure 5-46 can be made into a circulating register by connecting flip-flop  $X_0$  to the DATA IN line. No external inputs are used. Assume that this circulating register starts out with 1011 stored in it (i.e.,  $X_3 = 1$ ,  $X_2 = 0$ ,  $X_1 = 1$ , and  $X_0 = 1$ ). List the sequence of states that the register FFs go through as eight shift pulses are applied.
- D 5-29\* Refer to Figure 5-47, where a three-bit number stored in register  $X$  is serially shifted into register  $Y$ . How can the circuit be modified so that, at the end of the transfer operation, the original number stored in  $X$  is present in both registers? (*Hint*: See Problem 5-28.)

SECTION 5-18

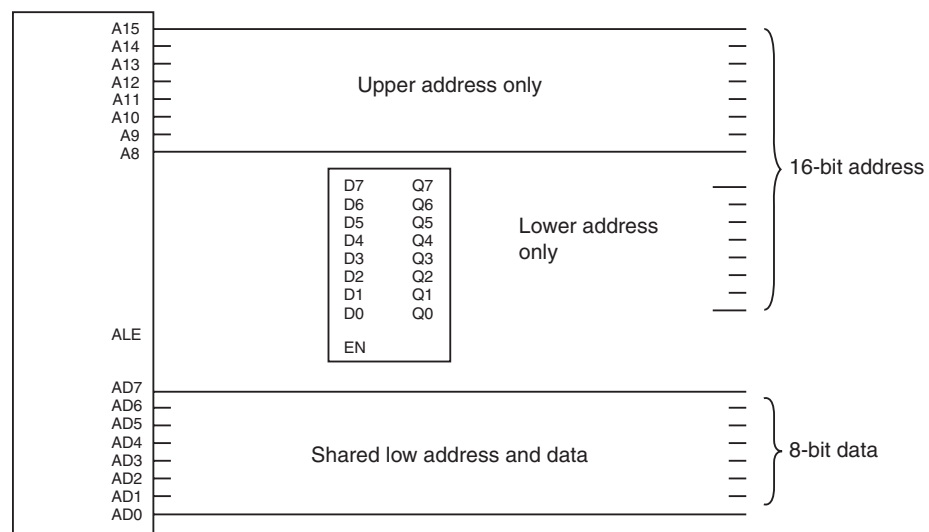
- B 5-30. Refer to the counter circuit of Figure 5-48 and answer the following:
  - (a)\* If the counter starts at 000, what will be the count after 13 clock pulses? After 99 pulses? After 256 pulses?
  - (b) If the counter starts at 100, what will be the count after 13 pulses? After 99 pulses? After 256 pulses?
  - (c) Connect a fourth J-K FF ( $X_3$ ) to this counter and draw the state transition diagram for this four-bit counter. If the input clock frequency is 80 MHz, what will the waveform at  $X_3$  look like?
- B 5-31. Refer to the binary counter of Figure 5-48. Change it by connecting  $\bar{X}_0$  to the  $CLK$  of flip-flop  $X_1$ , and  $\bar{X}_1$  to the  $CLK$  of flip-flop  $X_2$ . Start with all FFs in the 1 state, and draw the various FF output waveforms ( $X_0$ ,  $X_1$ ,  $X_2$ ,) for 16 input pulses. Then list the sequence of FF states as was done in Figure 5-49. This counter is called a *down counter*. Why?
- B 5-32. Draw the state transition diagram for this down counter, and compare it with the diagram of Figure 5-50. How are they different?

- B** 5-33\* (a) How many FFs are required to build a binary counter that counts from 0 to 1023?
- (b) Determine the frequency at the output of the last FF of this counter for an input clock frequency of 2 MHz.
- (c) What is the counter's MOD number?
- (d) If the counter is initially at zero, what count will it hold after 2060 pulses?
- B** 5-34. A binary counter is being pulsed by a 256-kHz clock signal. The output frequency from the last FF is 2 kHz.
- (a) Determine the MOD number.
- (b) Determine the counting range.
- B** 5-35. A photodetector circuit is being used to generate a pulse each time a customer walks into a certain establishment. The pulses are fed to an eight-bit counter. The counter is used to count these pulses as a means for determining how many customers have entered the store. After closing the store, the proprietor checks the counter and finds that it shows a count of  $00001001_2 = 9_{10}$ . He knows that this is incorrect because there were many more than nine people in his store. Assuming that the counter circuit is working properly, what could be the reason for the discrepancy?

### SECTION 5-19

- D** 5-36\* Modify the circuit of Figure 5-58 so that only the presence of address code 10110110 will allow data to be transferred to register X.
- T** 5-37. Suppose that the circuit of Figure 5-58 is malfunctioning so that data are being transferred to X for either of the address codes 11111110 or 11111111. What are some circuit faults that could be causing this?
- D** 5-38. Many microcontrollers share the same pins to output the lower address and transfer data. In order to hold the address constant while the data are transferred, the address information is stored in a latch which is enabled by the control signal ALE (address latch enable) as shown in Figure 5-97. Connect this latch to the microcontroller such that it takes what is on the lower address and data lines while ALE is HIGH and holds it on the lower address only lines when ALE is LOW.

**FIGURE 5-97** Problem 5-38.

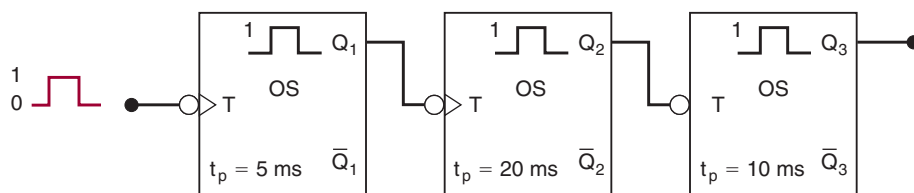


- D 5-39. Modify the circuit of Figure 5-58 so that the MPU has eight data output lines connected to transfer eight bits of data to an eight-bit register made up of two 74HC175 ICs. Show all circuit connections.

**SECTION 5-21**

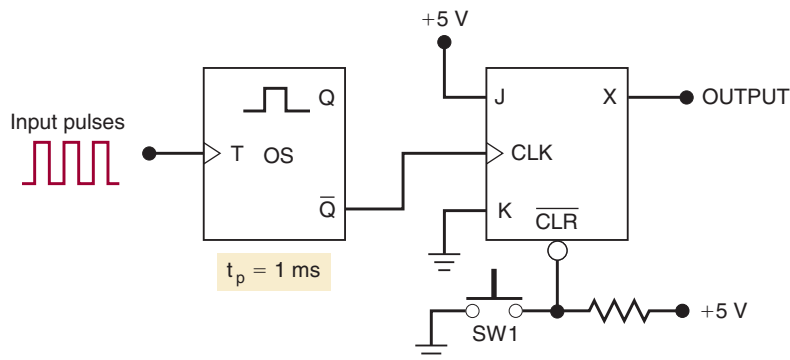
- B 5-40. Refer to the waveforms in Figure 5-61(a). Change the OS pulse duration to 0.5 ms and determine the  $Q$  output for both types of OS. Then repeat using a OS pulse duration of 1.5 ms.
- 5-41.\* Figure 5-98 shows three nonretriggerable one-shots connected in a timing chain that produces three sequential output pulses. Note the “1” in front of the pulse on each OS symbol to indicate nonretriggerable operation. Draw a timing diagram showing the relationship between the input pulse and the three OS outputs. Assume an input pulse duration of 10 ms.

**FIGURE 5-98** Problem 5-41.



- 5-42. A *retriggerable* OS can be used as a pulse-frequency detector that detects when the frequency of a pulse input is below a predetermined value. A simple example of this application is shown in Figure 5-99. The operation begins by momentarily closing switch SW1.
  - (a) Describe how the circuit responds to input frequencies above 1 kHz.
  - (b) Describe how the circuit responds to input frequencies below 1 kHz.
  - (c) How would you modify the circuit to detect when the input frequency drops below 50 kHz?

**FIGURE 5-99** Problem 5-42.



- 5-43. Refer to the logic symbol for a 74121 nonretriggerable one-shot in Figure 5-62.
  - (a)\* What input conditions are necessary for the OS to be triggered by a signal at the  $B$  input?
  - (b) What input conditions are necessary for the OS to be triggered by a signal at the  $A_1$  input?



- C, D** 5-44. The output pulse width from a 74121 OS is given by the approximate formula

$$t_p \approx 0.7R_T C_T$$

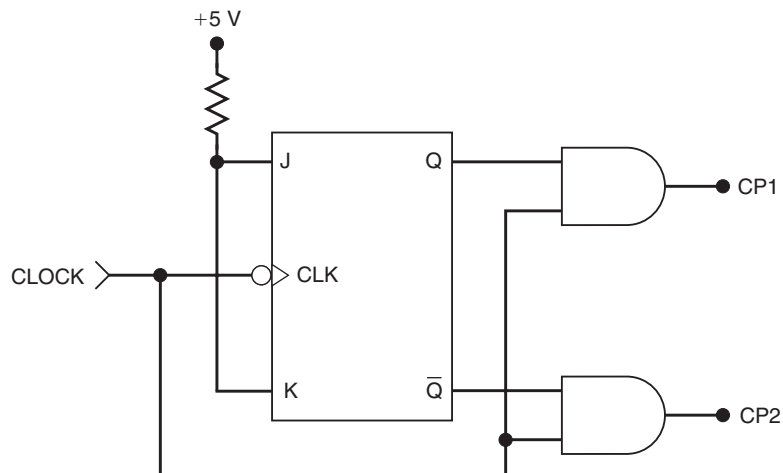
where  $R_T$  is the resistance connected between the  $R_{EXT}/C_{EXT}$  pin and  $V_{CC}$ , and  $C_T$  is the capacitance connected between the  $C_{EXT}$  pin and the  $R_{EXT}/C_{EXT}$  pin. The value for  $R_T$  can be varied between 2 and 40 k $\Omega$ , and  $C_T$  can be as large as 1000  $\mu$ F.

- (a) Show how a 74121 can be connected to produce a negative-going pulse with a 5-ms duration whenever either of two logic signals ( $E$  or  $F$ ) makes a NGT. Both  $E$  and  $F$  are normally in the HIGH state.
- (b) Modify the circuit so that a control input signal,  $G$ , can disable the OS output pulse, regardless of what occurs at  $E$  or  $F$ .

### SECTION 5-22

- B, D** 5-45\* Show how to use a 74LS14 Schmitt-trigger INVERTER to produce an approximate square wave with a frequency of 10 kHz.
- B, D** 5-46. Design a 555 free-running oscillator to produce an approximate square wave at 40 kHz.  $C$  should be kept at 500 pF or greater.
- D** 5-47. A 555 oscillator can be combined with a J-K flip-flop to produce a perfect (50 percent duty cycle) square wave. Modify the circuit of Problem 5-46 to include a J-K flip-flop. The final output is still to be a 40-kHz square wave.
- 5-48. Design a 555 timer circuit that will produce a 10 percent duty-cycle 5-kHz waveform. Choose a capacitor greater than 500 pF and resistors less than 100 k $\Omega$ . Draw the circuit diagram with pin numbers labeled.
- C** 5-49. The circuit in Figure 5-100 can be used to generate two nonoverlapping clock signals at the same frequency. These clock signals were used in early microprocessor systems that required four different clock transitions to synchronize their operations.
- (a) Draw the CP1 and CP2 timing waveforms if  $CLOCK$  is a 1-MHz square wave. Assume that  $t_{PLH}$  and  $t_{PHL}$  are 20 ns for the FF and 10 ns for the AND gates.

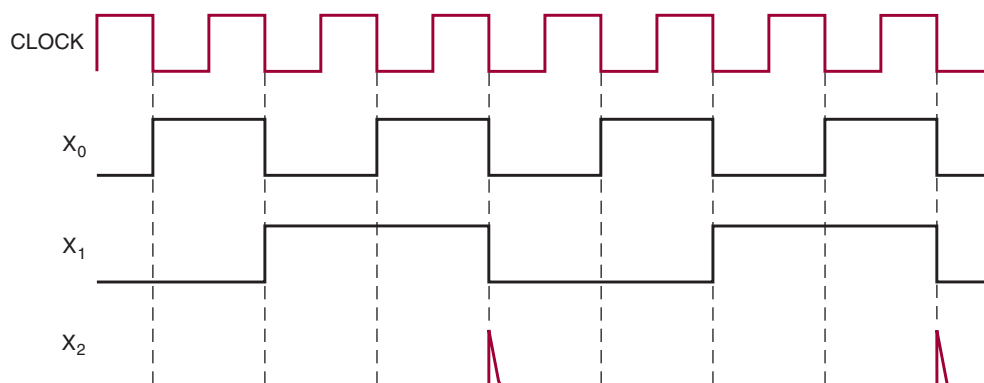
**FIGURE 5-100** Problem 5-49.



- (b) This circuit would have a problem if the FF were changed to one that responds to a PGT at  $CLK$ . Draw the CP1 and CP2 waveforms for that situation. Pay particular attention to conditions that can produce glitches.

**SECTION 5-23**

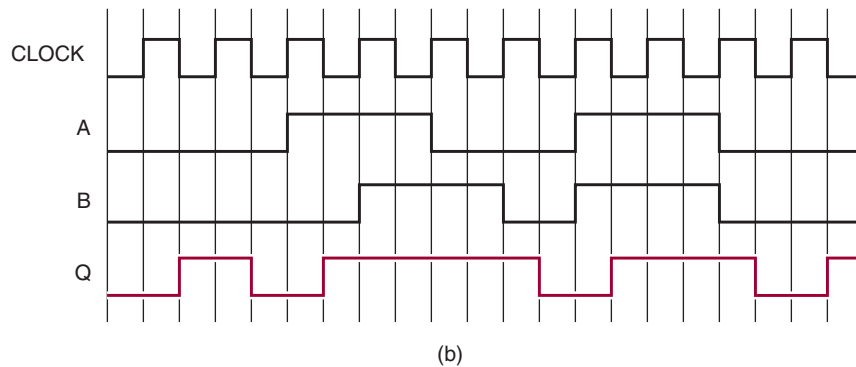
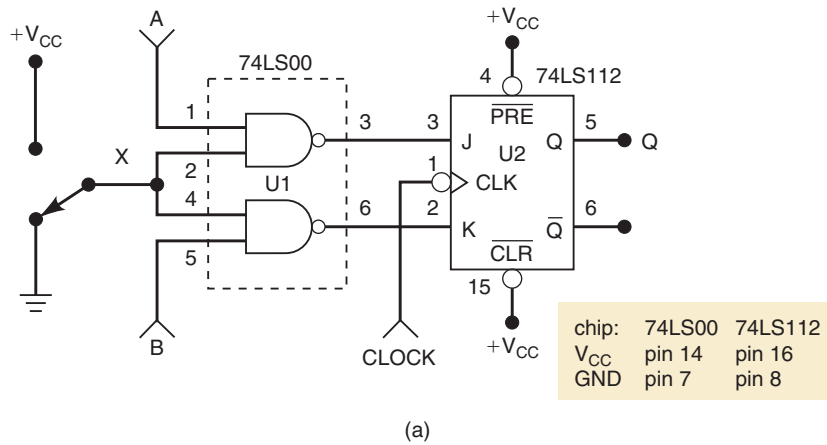
- T** 5-50. Refer to the counter circuit in Figure 5-48. Assume that all asynchronous inputs are connected to  $V_{CC}$ . When tested, the circuit waveforms appear as shown in Figure 5-101. Consider the following list of possible faults. For each one, indicate “yes” or “no” as to whether it could cause the observed results. Explain each response.
  - (a)\*  $CLR$  input of  $X_2$  is open.
  - (b)\*  $X_1$  output’s transition times are too long, possibly due to loading.
  - (c)  $X_2$  output is shorted to ground.
  - (d)  $X_2$ ’s hold time requirement is not being met.



**FIGURE 5-101** Problem 5-50.

- C, T** 5-51. Consider the situation of Figure 5-67 for each of the following sets of timing values. For each, indicate whether or not flip-flop  $Q_2$  will respond correctly.
  - (a)\* Each FF:  $t_{PLH} = 12\text{ ns}$ ;  $t_{PHL} = 8\text{ ns}$ ;  $t_S = 5\text{ ns}$ ;  $t_H = 0\text{ ns}$
  - (b) NAND gate:  $t_{PLH} = 8\text{ ns}$ ;  $t_{PHL} = 6\text{ ns}$
  - (c) INVERTER:  $t_{PLH} = 7\text{ ns}$ ;  $t_{PHL} = 5\text{ ns}$
  - (d) Each FF:  $t_{PLH} = 10\text{ ns}$ ;  $t_{PHL} = 8\text{ ns}$ ;  $t_S = 5\text{ ns}$ ;  $t_H = 0\text{ ns}$
  - (e) NAND gate:  $t_{PLH} = 12\text{ ns}$ ;  $t_{PHL} = 10\text{ ns}$
  - (f) INVERTER:  $t_{PLH} = 8\text{ ns}$ ;  $t_{PHL} = 6\text{ ns}$
- D** 5-52. Show and explain how the clock skew problem in Figure 5-67 can be eliminated by the appropriate insertion of two INVERTERS.
- T** 5-53. Refer to the circuit of Figure 5-102. Assume that the ICs are of the TTL logic family. The  $Q$  waveform was obtained when the circuit was tested with the input signals shown and with the switch in the “up” position; it is not correct. Consider the following list of faults, and for each indicate “yes” or “no” as to whether it could be the actual fault. Explain each response.
  - (a)\* Point X is always LOW due to a faulty switch.
  - (b)\* U1 pin 1 is internally shorted to  $V_{CC}$ .

**FIGURE 5-102** Problem 5-53.



(c) The connection from U1-3 to U2-3 is open.

(d) There is a solder bridge between pins 6 and 7 of U1.

- C** 5-54. The circuit of Figure 5-103 functions as a sequential combination lock. To operate the lock, proceed as follows:
1. Momentarily activate the RESET switch.
  2. Set the switches SWA, SWB, and SWC to the first part of the combination. Then momentarily toggle the ENTER switch back and forth.
  3. Set the switches to the second part of the combination, and toggle ENTER again. This should produce a HIGH at  $Q_2$  to open the lock.

If the incorrect combination is entered in either step, the operator must start the sequence over. Analyze the circuit and determine the correct sequence of combinations that will open the lock.

- C, T** 5-55\* When the combination lock of Figure 5-103 is tested, it is found that entering the correct combination does not open the lock. A logic probe check shows that entering the correct first combination sets  $Q_1$  HIGH, but entering the correct second combination produces only a momentary pulse at  $Q_2$ . Consider each of the following faults and indicate which one(s) could produce the observed operation. Explain each choice.
- (a) Switch bounce at SWA, SWB, or SWC.
  - (b) CLR input of  $Q_2$  is open.
  - (c) Connection from NAND gate 4 output to NAND gate 3 input is open.

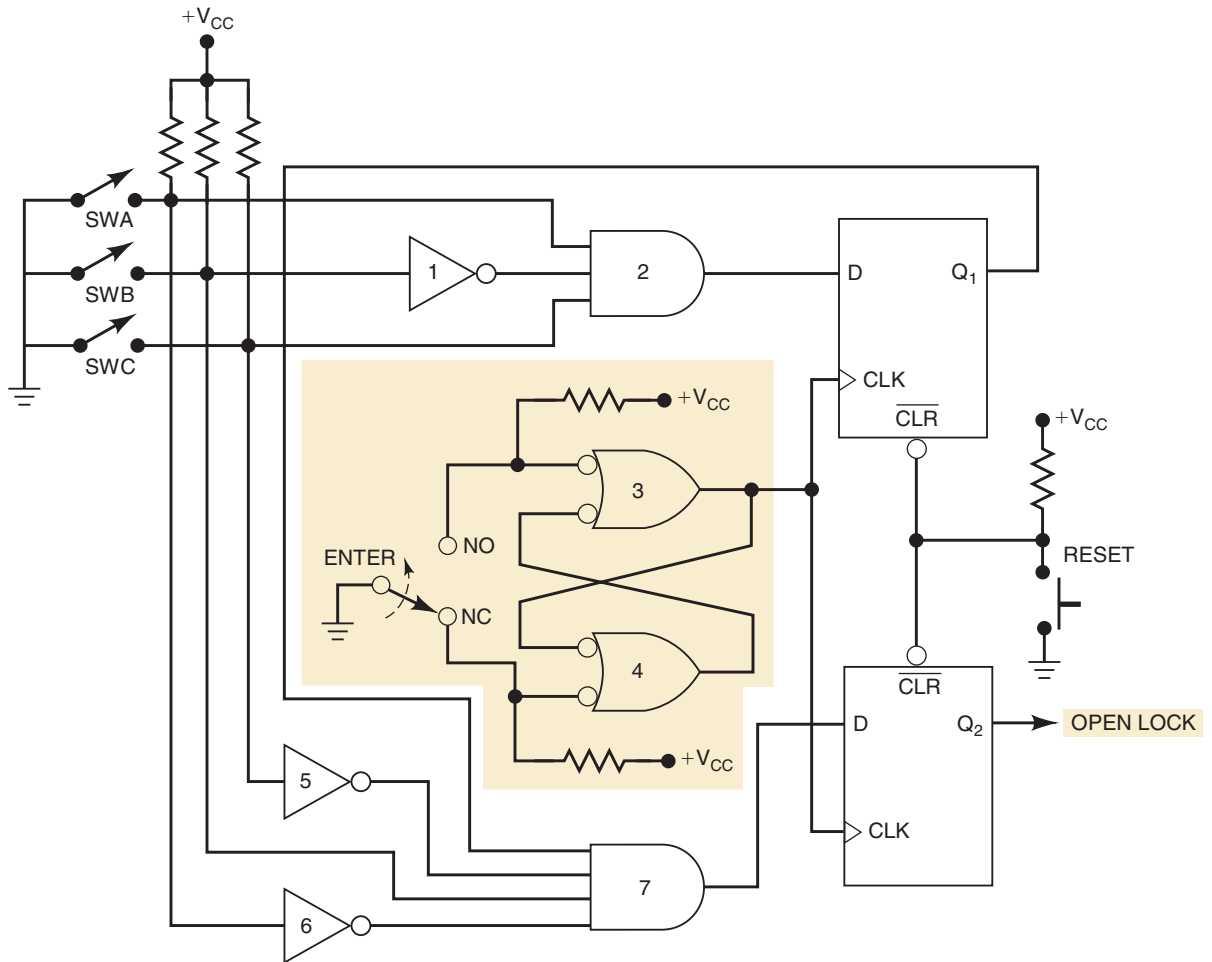


FIGURE 5-103 Problems 5-54, 5-55, and 5-69.

**DRILL QUESTIONS**

- B** 5-56. For each statement indicate what type of FF is being described.
  - (a)\* Has a SET and a CLEAR input but does not have a CLK input
  - (b)\* Will toggle on each CLK pulse when its control inputs are both HIGH
  - (c)\* Has an ENABLE input instead of a CLK input
  - (d)\* Is used to transfer data easily from one FF register to another
  - (e) Has only one control input
  - (f) Has two outputs that are complements of each other
  - (g) Can change states only on the active transition of CLK
  - (h) Is used in binary counters
- B** 5-57. Define the following terms.
  - (a) Asynchronous inputs
  - (b) Edge-triggered
  - (c) Shift register
  - (d) Frequency division
  - (e) Asynchronous (jam) transfer

- (f) State transition diagram
- (g) Parallel data transfer
- (h) Serial data transfer
- (i) Retriggerable one-shot
- (j) Schmitt-trigger inputs

### SECTIONS 5-24 THROUGH 5-27

- B** 5-58. Simulate the HDL design for a NAND latch given in Figure 5-76 (AHDL) or Figure 5-77 (VHDL). What does this S-R latch do if an “invalid” input command is applied? Since we know that any S-R latch can have an unusual output result when an invalid input command is applied, you should simulate that input condition as well as the latch’s normal set, reset, and hold commands. Some latch designs can have a tendency for the output to oscillate when an invalid command is followed by a hold command, so be sure to check for that in your simulation.
- B, H, N** 5-59\* Write an HDL design file for an active-HIGH input S-R latch. Functionally simulate the design.
- B, H** 5-60. Modify the latch description given in Figure 5-76 (AHDL) or Figure 5-77 (VHDL) to make the S-R reset if an invalid input is applied. Simulate the design.
- B, H** 5-61\* Add inverted outputs to the HDL NAND latch designs given in Figure 5-76 or Figure 5-77. Verify correct operation by simulation.
- B** 5-62. Simulate the AHDL or VHDL design for a D latch given in Section 5-25.
- D, H, N** 5-63. Create a four-bit transparent latch with one *enable* input and simulate (functional) your design.
  - (a) Use the primitive DLATCH in a schematic design file.
  - (b) Use the LPM\_LATCH in a schematic design file.
  - (c) Use an HDL design file. Modify the D latch design given in Section 5-25 by using arrays for the data inputs and outputs.
- D, H, N** 5-64. A toggle (T) flip-flop has a single control input (T). When  $T = 0$ , the flip-flop is in the no change state, similar to a JKFF with  $J = K = 0$ . When  $T = 1$ , the flip-flop is in the toggle mode, similar to a JKFF with  $J = K = 1$ . Create an HDL design for a T flip-flop and functionally simulate.
- H, N** 5-65. (a) Create the four-bit shift register in Figure 5-46(a) using the LPM\_FF megafunction in a schematic and functionally simulate.  
 (b) Create the four-bit shift register in Figure 5-46(a) using an HDL and functionally simulate.
- H, N** 5-66\* Create the two register circuit shown in Figure 5-47. Include a serial *data\_in* on the X register and functionally simulate.
  - (a) Use two LPM\_SHIFTRREG megafunctions in a schematic.
  - (b) Use an HDL.
- H** 5-67. (a) Write an AHDL design file for the FF circuit shown in Figure 5-67.  
 (b) Write a VHDL design file for the FF circuit shown in Figure 5-67.
- 5-68. Simulate (timing) the operation of the circuit in Problem 5-67. The simulation results should match the results in Figure 5-67.

- H** 5-69. (a) Write an AHDL design file to implement the entire circuit of Figure 5-103.
- (b) Write a VHDL design file to implement the entire circuit of Figure 5-103.

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

---

### SECTION 5-1

1. HIGH; LOW    2.  $Q = 0, \bar{Q} = 1$     3. True    4. Apply a momentary LOW to  $\overline{SET}$  input.

### SECTION 5-2

1. LOW, HIGH    2.  $Q = 1$  and  $\bar{Q} = 0$     3. Make RESET = 1    4.  $\overline{SET}$  and  $\overline{RESET}$  would both be normally in their active-LOW state.

### SECTION 5-4

1. See glossary    2. Time between 50% of leading edge and 50% of trailing edge    3. Time between 10 and 90% of HIGH level voltage    4. Time between 90 and 10% of HIGH level voltage

### SECTION 5-5

1. Synchronous control inputs and clock input    2. The FF output can change only when the appropriate clock transition occurs.    3. False    4. Setup time is the required interval immediately prior to the active edge of the  $CLK$  signal during which the control inputs must be held stable. Hold time is the required interval immediately following the active edge of  $CLK$  during which the control inputs must be held stable.    5. False    6. Violating setup or hold time constraints.

### SECTION 5-6

1. HIGH; LOW; HIGH    2. Because  $CLK^*$  is HIGH only for a few nanoseconds

### SECTION 5-7

1. True    2. No    3.  $J = 1, K = 0$

### SECTION 5-8

1.  $Q$  will go LOW at point  $a$  and remain LOW.    2. False. The  $D$  input can change without affecting  $Q$  because  $Q$  can change only on the active  $CLK$  edge.    3. Yes, by converting to D FFs (Figure 5-28).

### SECTION 5-9

1. In a D latch, the  $Q$  output can change while  $EN$  is HIGH. In a D flip-flop, the output can change only on the active edge of  $CLK$ .    2. False    3. True

### SECTION 5-10

1. Asynchronous inputs work independently of the  $CLK$  input.    2. Yes, because  $\overline{PRE}$  is active-LOW    3.  $J = K = 1, \overline{PRE} = \overline{CLR} = 1$ , and a PGT at  $CLK$

### SECTION 5-11

1.  $t_{PLH}$  and  $t_{PHL}$  requirements.    2. False; the waveform must also satisfy  $t_W(L)$  and  $t_W(H)$  requirements.    3. Setup time  $t_{su}$     4. Hold time  $t_h$     5. True
-

**SECTION 5-12**

1. False    2. True    3. The input levels prior to the clock edge determine the response of  $Q$ .

**SECTION 5-14**

1. Whenever there is a random external signal feeding the synchronous control inputs of a flip-flop.    2. Connect the random signal to D, system clock to clk, and use  $Q$  as the synchronized signal.

**SECTION 5-15**

1. If  $Q$  goes HIGH, it means D was already HIGH when clk went HIGH: D leads clock. If  $Q$  goes LOW, then clk leads D.

**SECTION 5-16**

1. Use an XOR gate to identify when the D input (present signal level) is different from the Q output (most recent signal level)    2. A 500 ns positive pulse.

**SECTION 5-17**

1. False    2. D flip-flop    3. Six    4. True

**SECTION 5-18**

1. True    2. Fewer interconnections between registers    3.  $X_2X_1X_0 = 111$ ;  
 $Y_2Y_1Y_0 = 101$     4. Parallel

**SECTION 5-19**

1. 10 kHz    2. Eight    3. 256    4. 2 kHz    5.  $00001000_2 = 8_{10}$

**SECTION 5-20**

1. Synchronization (first DFF in Figure 5-57), data transfer (second DFF in Figure 5-57), edge detection (third DFF in Figure 5-57), counting (included within counter block)    2. As shown in Figure 5-51, the system loses its count accuracy each time it reverses direction.    3. Signals  $A$  and  $B$  are asynchronous. They may arrive too close to the clock resulting in setup time and minimum pulse width violations.

**SECTION 5-22**

1. The output may contain oscillations.    2. It will produce clean, fast output signals even for slow-changing input signals.

**SECTION 5-23**

1.  $Q = 0, \bar{Q} = 1$     2. True    3. External  $R$  and  $C$  values    4. For a retriggerable OS, each new trigger pulse begins a new  $t_p$  interval, regardless of the state of the  $Q$  output.

**SECTION 5-24**

1. 24 kHz    2. 109.3 kHz; 66.7 percent    3. Frequency stability

**SECTION 5-25**

1. Clock skew is the arrival of a clock signal at the  $CLK$  inputs of different FFs at different times. It can cause a FF to go to an incorrect state.

**SECTION 5-26**

1. primitives, maxplus2, and megafunctions

**SECTION 5-27**

1. Feedback: The outputs are combined with the inputs to determine the next state of the outputs. 2. It progresses through a predetermined sequence of states in response to an input clock signal.

**SECTION 5-28**

1. A standard building block from a library of components that performs some fundamental logic function. 2. The names of each input and output and the primitive name that is recognized by the development system. 3. Under the HELP menu. 4. The PROCESS allows sequential IF constructs and the EVENT attribute detects transitions. 5. ieee.std\_logic\_1164. 6. altera.maxplus2

**SECTION 5-29**

1. Yes 2. In the VARIABLE section. 3. Each is assigned a variable name.  
4. = 5. SIGNALs 6. PORT MAP

---





# DIGITAL ARITHMETIC: OPERATIONS AND CIRCUITS

## ■ OUTLINE

- 6-1 Binary Addition and Subtraction
- 6-2 Representing Signed Numbers
- 6-3 Addition in the 2's-Complement System
- 6-4 Subtraction in the 2's-Complement System
- 6-5 Multiplication of Binary Numbers
- 6-6 Binary Division
- 6-7 BCD Addition
- 6-8 Hexadecimal Arithmetic
- 6-9 Arithmetic Circuits
- 6-10 Parallel Binary Adder
- 6-11 Design of a Full Adder
- 6-12 Complete Parallel Adder with Registers
- 6-13 Carry Propagation
- 6-14 Integrated-Circuit Parallel Adder
- 6-15 2's-Complement Circuits
- 6-16 ALU Integrated Circuits
- 6-17 Troubleshooting Case Study
- 6-18 Using Altera Library Functions
- 6-19 Logical Operations on Bit Arrays with HDLs
- 6-20 HDL Adders
- 6-21 Parameterizing the Bit Capacity of a Circuit

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Perform binary addition, subtraction, multiplication, and division on two binary numbers.
- Add and subtract hexadecimal numbers.
- Explain the difference between binary addition and OR addition.
- Compare the advantages and disadvantages among three different systems of representing signed binary numbers.
- Manipulate signed binary numbers using the 2's-complement system.
- Describe the BCD addition process.
- Describe the basic operation of an arithmetic/logic unit.
- Employ full adders in the design of parallel binary adders.
- Cite the advantages of parallel adders with the look-ahead carry feature.
- Explain the operation of a parallel adder/subtractor circuit.
- Use an ALU-integrated circuit to perform various logic and arithmetic operations on input data.
- Analyze troubleshooting case studies of adder/subtractor circuits.
- Use digital functions from libraries to implement more complex circuits.
- Use the Boolean equation form of description to perform operations on entire sets of bits.
- Apply software engineering techniques to expand the capacity of a hardware description.

## ■ INTRODUCTION

Digital computers and calculators perform the various arithmetic operations on numbers that are represented in binary form. The subject of digital arithmetic can be a very complex one if we want to understand all the various methods of computation and the theory behind them. Fortunately, this level of knowledge is not required by most technicians, at least not until they become experienced computer programmers. Our approach in this chapter will be to concentrate on those basic principles that are necessary for understanding how digital machines (i.e., computers) perform the basic arithmetic operations.

First, we will see how the various arithmetic operations are performed on binary numbers using “pencil and paper,” and then we will study the actual logic circuits that perform these operations in a digital system. Finally, we will learn how to describe these simple circuits using HDL techniques. Several methods of expanding the capacity of these circuits

will also be covered. The focus will be on the fundamentals of HDL, using arithmetic circuits as an example. The powerful capability of HDL combined with PLD hardware will provide the basis for further study, design, and experimentation with much more sophisticated arithmetic circuits in more advanced courses.

## 6-1 BINARY ADDITION AND SUBTRACTION

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Calculate the sum of  $n$ -bit binary numbers.
- Calculate the difference of  $n$ -bit binary numbers.

Let's start by taking a look at the simplest arithmetic operations that are performed by digital systems: addition and subtraction of two binary numbers.

### Binary Addition

The addition of two binary numbers is performed in exactly the same manner as the addition of decimal numbers. In fact, binary addition is simpler because there are fewer cases to learn. Let us first review decimal addition:

$$\begin{array}{r}
 \phantom{+} 376 \\
 + 461 \\
 \hline
 837
 \end{array}$$

↓
↓
LSD

The least-significant-digit (LSD) position is operated on first, producing a sum of 7. The digits in the second position are then added to produce a sum of 13, which produces a carry of 1 into the third position. This produces a sum of 8 in the third position.

The same general steps are followed in binary addition. However, only four cases can occur in adding the two binary digits (bits) in any position. They are:

$$\begin{aligned}
 0 + 0 &= 0 \\
 1 + 0 &= 1 \\
 1 + 1 &= 10 = 0 + \text{carry of 1 into next position} \\
 1 + 1 + 1 &= 11 = 1 + \text{carry of 1 into next position}
 \end{aligned}$$

The last case occurs when the two bits in a certain position are 1 and there is a carry from the previous position. Here are several examples of the addition of two binary numbers (decimal equivalents are in parentheses):

$$\begin{array}{r}
 011 (3) \\
 + 110 (6) \\
 \hline
 1001 (9)
 \end{array}
 \qquad
 \begin{array}{r}
 1001 (9) \\
 + 1111 (15) \\
 \hline
 11000 (24)
 \end{array}
 \qquad
 \begin{array}{r}
 11.011 (3.375) \\
 + 10.110 (2.750) \\
 \hline
 110.001 (6.125)
 \end{array}$$

It is not necessary to consider the addition of more than two binary numbers at a time because in all digital systems the circuitry that actually performs the addition can handle only two numbers at a time. When more than two numbers are to be added, the first two are added together and then their sum is added to the third number, and so on. This is not a serious drawback because digital computers can typically perform an addition operation in several nanoseconds.

Addition is the most important arithmetic operation in digital systems. As we shall see, the operations of subtraction, multiplication, and division as they are performed in most modern digital computers and calculators actually use only addition as their basic operation.

### Binary Subtraction

Likewise, binary subtraction is performed just like the subtraction of decimal numbers. There are only four possible situations that one has to deal with when subtracting one bit from another in any position of a binary number. They are:

$$\begin{aligned} 0 - 0 &= 0 \\ 1 - 1 &= 0 \\ 1 - 0 &= 1 \\ 0 - 1 &\Rightarrow \text{needs to borrow} \Rightarrow 10 - 1 = 1 \end{aligned}$$

The last case illustrates the need to borrow from the next column to the left when subtracting 1 from 0. Here are some examples of the subtraction of two binary numbers (with decimal equivalents in parentheses):

$$\begin{array}{r} 110 \text{ (6)} \\ - 010 \text{ (2)} \\ \hline 100 \text{ (4)} \end{array} \qquad \begin{array}{r} 11011 \text{ (27)} \\ - 01101 \text{ (13)} \\ \hline 1110 \text{ (14)} \end{array} \qquad \begin{array}{r} 1000.10 \text{ (8.50)} \\ - 0011.01 \text{ (3.25)} \\ \hline 101.01 \text{ (5.25)} \end{array}$$

#### OUTCOME ASSESSMENT QUESTIONS

- Add the following pairs of binary numbers.
  - $10110 + 00111$
  - $011.101 + 010.010$
  - $10001111 + 00000001$
- Subtract the following pairs of binary numbers:
  - $101101 - 010010$
  - $10001011 - 00110101$
  - $10101.1101 - 01110.0110$

## 6-2 REPRESENTING SIGNED NUMBERS

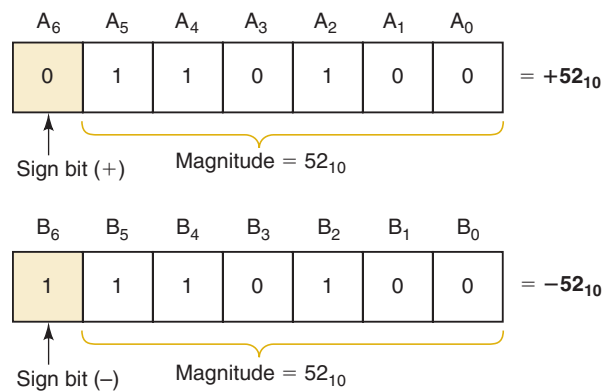
### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe sign/magnitude concept.
- Produce the 1's complement of any binary number.
- Produce the 2's complement of any binary number.
- Use the 2's complement method to represent any signed integer.
- Determine the number of bits needed for a given range.
- Determine the range for a given number of bits.
- Extend the range of any signed binary number.
- Negate any 2's-complement signed integer.

In digital computers, the binary numbers are represented by a set of binary storage devices (e.g., flip-flops). Each device represents one bit. For example, a six-bit FF register can store binary numbers ranging from 000000 to 111111 (0 to 63 in decimal). This represents the *magnitude* of the number. Because most digital computers and calculators handle negative as well as positive numbers, some means is required for representing the *sign* of the number (+ or -). This is usually done by adding to the number another bit called the **sign bit**. In general, the common convention is that a 0 in the sign bit represents a positive number and a 1 in the sign bit represents a negative number. This is illustrated in Figure 6-1. Register *A* contains the bits 0110100. The 0 in the leftmost bit ( $A_6$ ) is the sign bit that represents +. The other six bits are the magnitude of the number  $110100_2$ , which is equal to 52 in decimal. Thus, the number stored in the *A* register is +52. Similarly, the number stored in the *B* register is -52 because the sign bit is 1, representing -.

**FIGURE 6-1**  
Representation of signed numbers in sign-magnitude form.



The sign bit is used to indicate the positive or negative nature of the stored binary number. The numbers in Figure 6-1 consist of a sign bit and six magnitude bits. The magnitude bits are the true binary equivalent of the decimal value being represented. This is called the **sign-magnitude system** for representing signed binary numbers.

Although the sign-magnitude system is straightforward, calculators and computers do not normally use it because the circuit implementation is more complex than in other systems. The most commonly used system for representing signed binary numbers is the **2's-complement system**. Before we see how this is done, we must first see how to form the 1's complement and 2's complement of a binary number.

### 1's-Complement Form

The 1's complement of a binary number is obtained by changing each 0 to a 1 and each 1 to a 0. In other words, change each bit in the number to its complement. The process is shown below.

```

1 0 1 1 0 1 original binary number
  ↓ ↓ ↓ ↓ ↓ ↓
0 1 0 0 1 0 complement each bit to form 1's complement

```

Thus, we say that the 1's complement of 101101 is 010010.

## 2's-Complement Form

The 2's complement of a binary number is formed by taking the 1's complement of the number and adding 1 to the least-significant-bit position. The process is illustrated below for  $101101_2 = 45_{10}$ .

$$\begin{array}{r}
 1\ 0\ 1\ 1\ 0\ 1 \\
 0\ 1\ 0\ 0\ 1\ 0 \\
 + \quad \quad \quad 1 \\
 \hline
 0\ 1\ 0\ 0\ 1\ 1
 \end{array}
 \begin{array}{l}
 \text{binary equivalent of } 45 \\
 \text{complement each bit to form 1's complement} \\
 \text{add 1 to form 2's complement} \\
 \text{2's complement of original binary number}
 \end{array}$$

Thus, we say that 010011 is the 2's-complement representation of 101101.

Here's another example of converting a binary number to its 2's-complement representation:

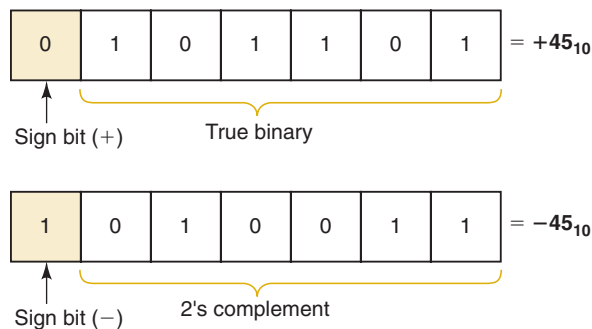
$$\begin{array}{r}
 1\ 0\ 1\ 1\ 0\ 0 \\
 0\ 1\ 0\ 0\ 1\ 1 \\
 + \quad \quad \quad 1 \\
 \hline
 0\ 1\ 0\ 1\ 0\ 0
 \end{array}
 \begin{array}{l}
 \text{original binary number} \\
 \text{1's complement} \\
 \text{add 1} \\
 \text{2's complement of original number}
 \end{array}$$

## Representing Signed Numbers Using 2's Complement

The 2's-complement system for representing signed numbers works like this:

- If the number is positive, the magnitude is represented in its true binary form, and a sign bit of 0 is placed in front of the MSB. This is shown in Figure 6-2 for  $+45_{10}$ .
- If the number is negative, the magnitude is represented in its 2's-complement form, and a sign bit of 1 is placed in front of the MSB. This is shown in Figure 6-2 for  $-45_{10}$ .

**FIGURE 6-2**  
Representation of signed numbers in the 2's-complement system.



The 2's-complement system is used to represent signed numbers because, as we shall see, it allows us to perform the operation of subtraction by actually performing addition. This is significant because it means that a digital computer can use the same circuitry both to add and to subtract, thereby realizing a saving in hardware.

**EXAMPLE 6-1**

Represent each of the following signed decimal numbers as a signed binary number in the 2's-complement system. Use a total of five bits, including the sign bit.

- (a) +13    (b) -9    (c) +3    (d) -2    (e) -8

**Solution**

- (a) The number is positive, so the magnitude (13) will be represented in its true-magnitude form, that is,  $13 = 1101_2$ . Attaching the sign bit of 0, we have

$$\begin{array}{r} +13 = 01101 \\ \text{sign bit} \leftarrow \uparrow \end{array}$$

- (b) The number is negative, so the magnitude (9) must be represented in 2's-complement form:

$$\begin{array}{r} 9_{10} = 1001_2 \\ \quad 0110 \quad (1\text{'s complement}) \\ + \quad \underline{\quad 1} \quad (add\ 1\ to\ LSB) \\ \quad 0111 \quad (2\text{'s complement}) \end{array}$$

When we attach the sign bit of 1, the complete signed number becomes

$$\begin{array}{r} -9 = 10111 \\ \text{sign bit} \leftarrow \uparrow \end{array}$$

The procedure we have just followed required two steps. First, we determined the 2's complement of the magnitude, and then we attached the sign bit. This can be accomplished in one step if we include the sign bit in the 2's-complement process. For example, to find the representation for -9, we start with the representation for +9, *including the sign bit*, and we take the 2's complement of it in order to obtain the representation for -9.

$$\begin{array}{r} +9 = 01001 \\ \quad 10110 \quad (1\text{'s complement of each bit including sign bit}) \\ + \quad \underline{\quad 1} \quad (add\ 1\ to\ LSB) \\ -9 = 10111 \quad (2\text{'s-complement representation of } -9) \end{array}$$

The result is, of course, the same as before.

- (c) The decimal value 3 can be represented in binary using only two bits. However, the problem statement requires a four-bit magnitude preceded by a sign bit. Thus, we have

$$+3_{10} = 00011$$

In many situations the number of bits is fixed by the size of the registers that will be holding the binary numbers, so that 0s may have to be added in order to fill the required number of bit positions.

- (d) Start by writing +2 using five bits:

$$\begin{array}{r} +2 = 00010 \\ \quad 11101 \quad (1\text{'s complement}) \\ + \quad \underline{\quad 1} \quad (add\ 1) \\ -2 = 11110 \quad (2\text{'s-complement representation of } -2) \end{array}$$

(e) Start with +8:

$$\begin{array}{r}
 +8 = 01000 \\
 \phantom{+8 = } 10111 \quad (\text{complement each bit}) \\
 + \phantom{+8 = } \underline{\phantom{0}1} \quad (\text{add 1}) \\
 -8 = 11000 \quad (2\text{'s-complement representation of } -8)
 \end{array}$$


---

## Sign Extension

Example 6-1 required that we use a total of five bits to represent the signed numbers. The size of a register (*number of flip-flops*) determines the number of binary digits that are stored for each number. Most digital systems today store numbers in registers sized in even multiples of four bits. In other words, the storage registers will be made up of 4, 8, 12, 16, 32, or 64 bits. In a system that stores eight-bit numbers, seven bits represent the magnitude and the MSB represents the sign. If we need to store a positive five-bit number in an eight-bit register, it makes sense to simply add leading zeros. The MSB (sign bit) is still 0, indicating a positive value.

0000 1001  
appended leading 0s          binary value for 9

What happens when we try to store five-bit negative numbers in an eight-bit register? In the previous section we found that the five-bit, 2's-complement binary representation for  $-9$  is 10111.

1 0111

If we appended leading 0s, this would no longer be a negative number in eight-bit format. The proper way to extend a negative number is to append leading 1's. Thus, the value stored for negative 9 is

111 1 0111  
2's complement magnitude  
sign in five-bit format  
sign extension to eight-bit format

## Negation

**Negation** is the operation of converting a positive number to its negative equivalent or a negative number to its positive equivalent. When signed binary numbers are represented in the 2's-complement system, negation is performed simply by performing the 2's-complement operation. To illustrate, let's start with +9 in eight-bit binary form. Its signed representation is 00001001. If we take its 2's complement we get 11110111, which represents the signed value  $-9$ . Likewise, we can start with the representation of  $-9$ , which is 11110111, and take its 2's complement to get 00001001, which represents +9. These steps are diagrammed below.

Start with	00001001	+9
2's complement (negate)	11110111	-9
negate again	00001001	+9

**Thus, we negate a signed binary number by 2's complementing it.**

---



This negation changes the number to its equivalent of opposite sign. We used negation in steps (d) and (e) of Example 6-1 to convert positive numbers to their negative equivalents.

### EXAMPLE 6-2

Each of the following numbers is a five-bit signed binary number in the 2's-complement system. Determine the decimal value in each case:

- (a) 01100    (b) 11010    (c) 10001

#### Solution

- (a) The sign bit is 0, so the number is *positive* and the other four bits represent the true magnitude of the number. That is,  $1100_2 = 12_{10}$ . Thus, the decimal number is +12.
- (b) The sign bit of 11010 is a 1, so we know that the number is *negative*, but we can't tell what the magnitude is. We can find the magnitude by negating (2's complementing) the number to convert it to its positive equivalent.

$$\begin{array}{r} 11010 \quad (\text{original negative number}) \\ 00101 \quad (\text{1's complement}) \\ + \quad \underline{\quad 1} \quad (\text{add 1}) \\ \hline 00110 \quad (+6) \end{array}$$

Because the result of the negation is  $00110 = +6$ , the original number 11010 must be equivalent to  $-6$ .

- (c) Follow the same procedure as in (b):

$$\begin{array}{r} 10001 \quad (\text{original negative number}) \\ 01110 \quad (\text{1's complement}) \\ + \quad \underline{\quad 1} \quad (\text{add 1}) \\ \hline 01111 \quad (+15) \end{array}$$

Thus,  $10001 = -15$ .

### Special Case in 2's-Complement Representation

Whenever a signed number has a 1 in the sign bit and all 0s for the magnitude bits, its decimal equivalent is  $-2^N$ , where  $N$  is the number of bits in the *magnitude*. For example,

$$\begin{array}{l} 1000 = -2^3 = -8 \\ 10000 = -2^4 = -16 \\ 100000 = -2^5 = -32 \end{array}$$

and so on. Notice that in this special case, taking the 2's complement of these numbers produces the value we started with because we are at the negative limit of the range of numbers that can be represented by this many bits. If we extend the sign of these special numbers, the normal negation procedure works fine. For example, extending the number 1000 ( $-8$ ) to 11000 (five-bit negative 8) and taking its 2's complement we get 01000 (8), which is the magnitude of the negative number.

Thus, we can state that the complete range of values that can be represented in the 2's-complement system having  $N$  magnitude bits is

$$-2^N \text{ to } +(2^N - 1)$$

There are a total of  $2^{N+1}$  different values, *including* zero.

For example, Table 6-1 lists all signed numbers that can be represented in four bits using the 2's-complement system (note there are three magnitude

TABLE 6-1

Decimal Value	Signed Binary Using 2's Complement
+7 = $2^3 - 1$	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8 = $-2^3$	1000

bits, so  $N = 3$ ). Note that the sequence starts at  $-2^N = -2^3 = -8_{10} = 1000_2$  and proceeds upward to  $+(2^N - 1) = +2^3 - 1 = +7_{10} = 0111_2$  by adding 0001 at each step as in an up counter.

**EXAMPLE 6-3**

What is the range of *unsigned* decimal values that can be represented in a byte?

**Solution**

Recall that a byte is eight bits. We are interested in unsigned numbers here, so there is no sign bit, and all of the eight bits are used for the magnitude. Therefore, the values will range from

$$00000000_2 = 0_{10}$$

to

$$11111111_2 = 255_{10}$$

This is a total of 256 different values, which we could have predicted because  $2^8 = 256$ .

**EXAMPLE 6-4**

What is the range of *signed* decimal values that can be represented in a byte?

**Solution**

Because the MSB is to be used as the sign bit, there are seven bits for the magnitude. The largest negative value is

$$10000000_2 = -2^7 = -128_{10}$$

The largest positive value is

$$01111111_2 = +2^7 - 1 = +127_{10}$$

Thus, the range is  $-128$  to  $+127$ ; this is a total of 256 different values, including zero. Alternatively, because there are seven magnitude bits ( $N = 7$ ), there are  $2^{N+1} = 2^8 = 256$  different values.

**EXAMPLE 6-5**

A certain computer is storing the following two signed numbers in its memory using the 2's-complement system:

$$\begin{aligned} 00011111_2 &= +31_{10} \\ 11110100_2 &= -12_{10} \end{aligned}$$

While executing a program, the computer is instructed to convert each number to its opposite sign; that is, change the  $+31$  to  $-31$  and change the  $-12$  to  $+12$ . How will it do this?

**Solution**

This is the negation operation whereby a signed number can have its polarity changed simply by performing the 2's-complement operation on the *complete* number, including the sign bit. The computer circuitry will take the signed number from memory, find its 2's complement, and put the result back in memory.

**OUTCOME ASSESSMENT QUESTIONS**

- Represent each of the following values as an eight-bit signed number in the 2's-complement system.  
(a)  $+13$  (b)  $-7$  (c)  $-128$
- Each of the following is a signed binary number in the 2's-complement system. Determine the decimal equivalent for each.  
(a)  $100011$  (b)  $1000000$  (c)  $01111110$
- What range of signed decimal values can be represented in 12 bits (including the sign bit)?
- How many bits are required to represent decimal values ranging from  $-50$  to  $+50$ ?
- What is the largest negative decimal value that can be represented by a two-byte number?
- Perform the 2's-complement operation on each of the following.  
(a)  $10000$  (b)  $10000000$  (c)  $1000$
- Define the negation operation.

## 6-3 ADDITION IN THE 2'S-COMPLEMENT SYSTEM

### OUTCOMES

Upon completion of this section, you will be able to:

- Calculate the sum of any two signed integers in 2's-complement format.
- Interpret sign and magnitude of the sum.

We will now investigate how the operations of addition and subtraction are performed in digital machines that use the 2's-complement representation for negative numbers. In the various cases to be considered, it is important to note that the sign bit of each number is operated on in the same manner as the magnitude bits.

**Case I: Two Positive Numbers.** The addition of two positive numbers is straightforward. Consider the addition of +9 and +4:

$$\begin{array}{r}
 +9 \rightarrow \boxed{0} 1001 \quad (\text{augend}) \\
 +4 \rightarrow \boxed{0} 0100 \quad (\text{addend}) \\
 \hline
 \boxed{0} 1101 \quad (\text{sum} = +13) \\
 \uparrow \\
 \text{sign bits}
 \end{array}$$

Note that the sign bits of the **augend** and the **addend** are both 0 and the sign bit of the sum is 0, indicating that the sum is positive. Also note that the augend and the addend are made to have the same number of bits. This must *always* be done in the 2's-complement system.

**Case II: Positive Number and Smaller Negative Number.** Consider the addition of +9 and -4. Remember that the -4 will be in its 2's-complement form. Thus, +4 (00100) must be converted to -4 (11100).

$$\begin{array}{r}
 \downarrow \text{sign bits} \\
 +9 \rightarrow \boxed{0} 1001 \quad (\text{augend}) \\
 -4 \rightarrow \boxed{1} 1100 \quad (\text{addend}) \\
 \hline
 \boxed{1} 0 0101 \\
 \uparrow \\
 \text{This carry is disregarded; the result is } 00101 \text{ (sum} = +5\text{).}
 \end{array}$$

In this case, the sign bit of the addend is 1. Note that the sign bits also participate in the addition process. In fact, a carry is generated in the last position of addition. *This carry is always disregarded*, so that the final sum is 00101, which is equivalent to +5.

**Case III: Positive Number and Larger Negative Number.** Consider the addition of -9 and +4:

$$\begin{array}{r}
 -9 \rightarrow 10111 \\
 +4 \rightarrow 00100 \\
 \hline
 11011 \quad (\text{sum} = -5) \\
 \uparrow \\
 \text{negative sign bit}
 \end{array}$$

The sum here has a sign bit of 1, indicating a negative number. Because the sum is negative, it is in 2's-complement form, so that the last four bits, 1011, actually represent the 2's complement of the sum. To find the true magnitude of the sum, we must negate (2's complement) 11011; the result is 00101 = +5. Thus, 11011 represents -5.

**Case IV: Two Negative Numbers**

$$\begin{array}{r} -9 \rightarrow 10111 \\ -4 \rightarrow 11100 \\ \hline \end{array}$$

$$\begin{array}{r} 1 \ 10011 \\ \uparrow \quad \uparrow \\ \text{sign bit} \end{array}$$

This carry is disregarded; the result is 10011 (sum = -13).

This final result is again negative and in 2's-complement form with a sign bit of 1. Negating (2's complementing) this result produces 01101 = +13.

**Case V: Equal and Opposite Numbers**

$$\begin{array}{r} -9 \rightarrow 10111 \\ +9 \rightarrow 01001 \\ \hline 0 \ 1 \ 00000 \end{array}$$

Disregard; the result is 00000 (sum = +0).

The result is obviously +0, as expected.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

Assume the 2's-complement system for both questions.

1. *True or false:* Assume that the sum falls within the valid range for the number of bits. Whenever the sum of two signed binary numbers has a sign bit of 1, the magnitude of the sum is in 2's-complement form.
2. Add the following pairs of signed numbers. Express the sum as a signed binary number and as a decimal number.
  - (a) 100111 + 111011
  - (b) 100111 + 011001

## 6-4 SUBTRACTION IN THE 2'S-COMPLEMENT SYSTEM

### OUTCOMES

Upon completion of this section, you will be able to:

- Calculate the difference of any two signed integers in 2's-complement format.
- Interpret sign and magnitude of the difference.
- Identify when arithmetic overflow has occurred.

The subtraction operation using the 2's-complement system actually involves the operation of addition and is really no different from the various cases for addition considered in Section 6-3. When subtracting one binary number (the **subtrahend**) from another binary number (the **minuend**), use the following procedure:

1. *Negate the subtrahend.* This will change the subtrahend to its equivalent value of opposite sign.
2. *Add this to the minuend.* The result of this addition will represent the *difference* between the subtrahend and the minuend.

Once again, as in all 2's-complement arithmetic operations, it is necessary that both numbers have the same number of bits in their representations.

Let us consider the case where +4 is to be subtracted from +9.

$$\begin{array}{r} \text{minuend (+9)} \rightarrow 01001 \\ \text{subtrahend (+4)} \rightarrow 00100 \end{array}$$

Negate the subtrahend to produce 11100, which represents  $-4$ . Now add this to the minuend.

$$\begin{array}{r} 01001 \quad (+9) \\ + 11100 \quad (-4) \\ \hline \overset{1}{\uparrow} 00101 \quad (+5) \\ \uparrow \text{Disregard, so the result is } 00101 = +5. \end{array}$$

When the subtrahend is changed to its 2's complement, it actually becomes  $-4$ , so that we are *adding*  $-4$  and  $+9$ , which is the same as subtracting  $+4$  from  $+9$ . This is the same as case II of Section 6-3. Any subtraction operation, then, actually becomes one of addition when the 2's-complement system is used. This feature of the 2's-complement system has made it the most widely used of the methods available because it allows addition and subtraction to be performed by the same circuitry.

Here's another example showing  $+9$  subtracted from  $-4$ :

$$\begin{array}{r} 11100 \quad (-4) \\ - 01001 \quad (+9) \end{array}$$

Negate the subtrahend ( $+9$ ) to produce  $10111$  ( $-9$ ) and add this to the minuend ( $-4$ ).

$$\begin{array}{r} 11100 \quad (-4) \\ + 10111 \quad (-9) \\ \hline \overset{1}{\uparrow} 10011 \quad (-13) \\ \uparrow \text{Disregard} \end{array}$$

Verify the results of using the above procedure for the following subtractions: (a)  $+9 - (-4)$ ; (b)  $-9 - (+4)$ ; (c)  $-9 - (-4)$ ; (d)  $+4 - (-4)$ . Remember that when the result has a sign bit of 1, it is negative and in 2's-complement form.

## Arithmetic Overflow

In each of the previous addition and subtraction examples, the numbers that were added consisted of a sign bit and four magnitude bits. The answers also consisted of a sign bit and four magnitude bits. Any carry into the sixth bit position was disregarded. In all of the cases considered, the magnitude of the answer was small enough to fit into four bits. Let's look at the addition of  $+9$  and  $+8$ .

$$\begin{array}{r} +9 \rightarrow \boxed{0} \ 1001 \\ +8 \rightarrow \boxed{0} \ 1000 \\ \hline \boxed{1} \ \underline{0001} \\ \text{incorrect sign} \quad \uparrow \quad \uparrow \quad \text{incorrect magnitude} \end{array}$$

The answer has a negative sign bit, which is obviously incorrect because we are adding two positive numbers. The answer should be  $+17$ , but the

magnitude 17 requires more than four bits and therefore *overflows* into the sign-bit position. This **overflow** condition can occur only when two positive or two negative numbers are being added, and it always produces an incorrect result. Overflow can be detected by checking to see that the sign bit of the result is the same as the sign bits of the numbers being added.

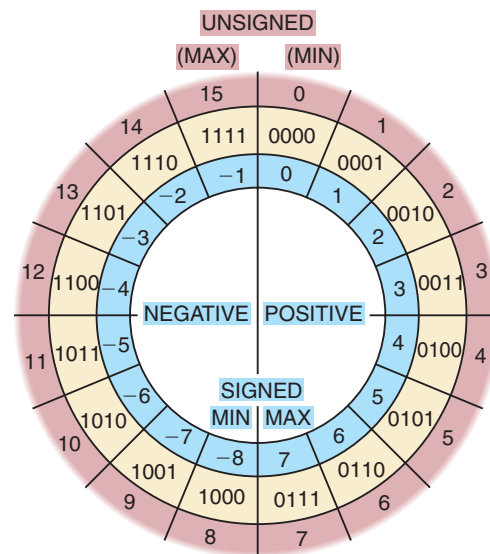
Subtraction in the 2's-complement system is performed by negating the minuend and *adding* it to the subtrahend, so overflow can occur only when the minuend and subtrahend have different signs. For example, if we are subtracting  $-8$  from  $+9$ , the  $-8$  is negated to become  $+8$  and is added to  $+9$ , just as shown above, and overflow produces an erroneous negative result because the magnitude is too large.

A computer will have a special circuit to detect any overflow condition when two numbers are added or subtracted. This detection circuit will signal the computer's control unit that overflow has occurred and the result is incorrect. We will examine such a circuit in end-of-chapter problem 6-23.

### Number Circles and Binary Arithmetic

The concept of signed arithmetic and overflow can be illustrated by taking the numbers from Table 6-1 and “bending” them into a number circle as shown in Figure 6-3. Notice that there are two ways to look at this circle. It can be thought of as a circle of unsigned numbers (as shown in the outer ring) with minimum value 0 and maximum 15, or as signed 2's-complement numbers (as shown in the inner ring) with maximum value 7 and minimum  $-8$ . To add using a number circle, simply start at the value of the augend and

**FIGURE 6-3** A four-bit number circle.



advance around the number circle clockwise by the number of spaces in the addend. For example, to add  $2 + 3$ , start at 2 (0010) and then advance clockwise three more spaces to arrive at 5 (0101). Overflow occurs when the sum is too big to fit into four-bit signed format, meaning we have exceeded the maximum value of 7. On the number circle this is indicated when adding two positive values causes us to cross the line between 0111 (max positive) and 1000 (max negative).

The number circle can also illustrate how 2's-complement subtraction really works. For example, let's perform the subtraction of 5 from 3. Of course, we know the answer is  $-2$ , but let's run the problem through the number circle. First we start at the number 3 (0011) on the number circle. The most apparent way to subtract is to move *counterclockwise* around the circle five spaces, which lands us on the number 1110 ( $-2$ ). The less obvious operation that illustrates 2's-complement arithmetic is to add  $-5$  to the number 3. Negative five (the 2's complement of 0101) is 1011, which, interpreted as an unsigned binary number, represents the value 11 (eleven) in decimal. Start at the number 3 (0011) and move clockwise around the circle 11 spaces and you will once again find yourself arriving at the number 1110 ( $-2$ ), which is the correct result.

Any subtraction operation between four-bit numbers of opposite sign that produces a result greater than 7 or less than  $-8$  is an overflow of the four-bit format and results in an incorrect answer. For example, 3 minus  $-6$  should produce the answer 9, but moving clockwise six spaces from 3 lands us on the signed number  $-7$ : an overflow condition has occurred, giving us an incorrect answer.

### OUTCOME ASSESSMENT QUESTIONS

- Perform the subtraction on the following pairs of signed numbers using the 2's-complement system. Express the results as signed binary numbers and as decimal values.
  - $01001 - 11010$
  - $10010 - 10011$
- How can arithmetic overflow be detected when signed numbers are being added? Subtracted?

## 6-5 MULTIPLICATION OF BINARY NUMBERS

### OUTCOMES

Upon completion of this section, you will be able to:

- Perform manual calculation of the product of two unsigned binary numbers.
- State the process by which signed binary numbers are multiplied to produce the signed product.

The multiplication of binary numbers is done in the same manner as the multiplication of decimal numbers. The process is actually simpler because the multiplier digits are either 0 or 1 and so we are always multiplying by 0 or 1 and no other digits. The following example illustrates for unsigned binary numbers:

$$\begin{array}{r}
 1001 \quad \leftarrow \text{multiplicand} = 9_{10} \\
 \underline{1011} \quad \leftarrow \text{multiplier} = 11_{10} \\
 1001 \\
 1001 \\
 0000 \\
 \underline{1001} \\
 110011 \quad \leftarrow \text{final product} = 99_{10}
 \end{array}$$



In this example the multiplicand and the multiplier are in true binary form and no sign bits are used. The steps followed in the process are exactly the same as in decimal multiplication. First, the LSB of the multiplier is examined; in our example, it is a 1. This 1 multiplies the multiplicand to produce 1001, which is written down as the first partial product. Next, the second bit of the multiplier is examined. It is a 1, and so 1001 is written for the second partial product. Note that this second partial product is *shifted* one place to the left relative to the first one. The third bit of the multiplier is 0, and 0000 is written as the third partial product; again, it is shifted one place to the left relative to the previous partial product. The fourth multiplier bit is 1, and so the last partial product is 1001 shifted again one position to the left. The four partial products are then summed to produce the final product.

Most digital machines can add only two binary numbers at a time. For this reason, the partial products formed during multiplication cannot all be added together at the same time. Instead, they are added together two at a time; that is, the first is added to the second, their sum is added to the third, and so on. This process is now illustrated for the example above:

$$\begin{array}{r}
 \text{Add } \left\{ \begin{array}{l} 1001 \leftarrow \text{first partial product} \\ \underline{1001} \leftarrow \text{second partial product shifted left} \end{array} \right. \\
 \\
 \text{Add } \left\{ \begin{array}{l} 11011 \leftarrow \text{sum of first two partial products} \\ \underline{0000} \leftarrow \text{third partial product shifted left} \end{array} \right. \\
 \\
 \text{Add } \left\{ \begin{array}{l} 011011 \leftarrow \text{sum of first three partial products} \\ \underline{1001} \leftarrow \text{fourth partial product shifted left} \end{array} \right. \\
 \\
 1100011 \leftarrow \text{sum of four partial products, which} \\
 \leftarrow \text{equals final total product}
 \end{array}$$

### Multiplication in the 2's-Complement System

In computers that use the 2's-complement representation, multiplication is carried on in the manner described above, provided that both the multiplicand and the multiplier are put in true binary form. If the two numbers to be multiplied are positive, they are already in true binary form and are multiplied as they are. The resulting product is, of course, positive and is given a sign bit of 0. When the two numbers are negative, they will be in 2's-complement form. The 2's complement of each is taken to convert it to a positive number, and then the two numbers are multiplied. The product is kept as a positive number and is given a sign bit of 0.

When one of the numbers is positive and the other is negative, the negative number is first converted to a positive magnitude by taking its 2's complement. The product will be in true-magnitude form. However, the product must be negative because the original numbers are of opposite sign. Thus, the product is then changed to 2's-complement form and is given a sign bit of 1.

## 6-6 BINARY DIVISION

---

### OUTCOME

Upon completion of this section, you will be able to:

- Perform manual calculation of the quotient and remainder of two unsigned binary numbers.

The process for dividing one binary number (the *dividend*) by another (the *divisor*) is the same as that followed for decimal numbers, that which we usually refer to as “long division.” The actual process is simpler in binary because when we are checking to see how many times the divisor “goes into” the dividend, there are only two possibilities, 0 or 1. To illustrate, consider the following simple division examples:

$$\begin{array}{r}
 0011 \\
 11 \overline{)1001} \\
 \underline{011} \\
 0011 \\
 \underline{11} \\
 0
 \end{array}
 \qquad
 \begin{array}{r}
 0010.1 \\
 100 \overline{)1010.0} \\
 \underline{100} \\
 100 \\
 \underline{100} \\
 0
 \end{array}$$

In the first example, we have  $1001_2$  divided by  $11_2$ , which is equivalent to  $9 \div 3$  in decimal. The resulting quotient is  $0011_2 = 3_{10}$ . In the second example,  $1010_2$  is divided by  $100_2$ , or  $10 \div 4$  in decimal. The result is  $0010.1_2 = 2.5_{10}$ .

In most digital machines, the subtractions that are part of the division operation are usually carried out using 2’s-complement subtraction, that is, taking the 2’s complement of the subtrahend and then adding.

The division of signed numbers is handled in the same way as multiplication. Negative numbers are made positive by complementing, and the division is then carried out. If the dividend and the divisor are of opposite sign, the resulting quotient is changed to a negative number by taking its 2’s complement and is given a sign bit of 1. If the dividend and the divisor are of the same sign, the quotient is left as a positive number and is given a sign bit of 0.

## 6-7 BCD ADDITION

---

### OUTCOMES

Upon completion of this section, you will be able to:

- Perform manual calculations of the sum of BCD numbers.
- Interpret the result in decimal.

In Chapter 2, we stated that many computers and calculators use the BCD code to represent decimal numbers. Recall that this code takes *each* decimal digit and represents it by a four-bit code ranging from 0000 to 1001. The addition of decimal numbers that are in BCD form can be best understood by considering the two cases that can occur when two decimal digits are added.

---

### Sum Equals 9 or Less

Consider adding 5 and 4 using BCD to represent each digit:

$$\begin{array}{r} 5 \quad 0101 \leftarrow \text{BCD for 5} \\ +4 \quad + \underline{0100} \leftarrow \text{BCD for 4} \\ \hline 9 \quad 1001 \leftarrow \text{BCD for 9} \end{array}$$

The addition is carried out as in normal binary addition, and the sum is 1001, which is the BCD code for 9. As another example, take 45 added to 33:

$$\begin{array}{r} 45 \quad 0100 \ 0101 \leftarrow \text{BCD for 45} \\ +33 \quad + \underline{0011 \ 0011} \leftarrow \text{BCD for 33} \\ \hline 78 \quad 0111 \ 1000 \leftarrow \text{BCD for 78} \end{array}$$

In this example, the four-bit codes for 5 and 3 are added in binary to produce 1000, which is BCD for 8. Similarly, adding the second-decimal-digit positions produces 0111, which is BCD for 7. The total is 01111000, which is the BCD code for 78.

In the examples above, none of the sums of the pairs of decimal digits exceeded 9; therefore, *no decimal carries were produced*. For these cases, the BCD addition process is straightforward and is actually the same as binary addition.

### Sum Greater than 9

Consider the addition of 6 and 7 in BCD:

$$\begin{array}{r} 6 \quad 0110 \leftarrow \text{BCD for 6} \\ +7 \quad + \underline{0111} \leftarrow \text{BCD for 7} \\ +13 \quad 1101 \leftarrow \text{invalid code group for BCD} \end{array}$$

The sum 1101 does not exist in the BCD code; it is one of the six forbidden or invalid four-bit code groups. This has occurred because the sum of the two digits exceeds 9. Whenever this occurs, the sum must be corrected by the addition of six (0110) to take into account the skipping of the six invalid code groups:

$$\begin{array}{r} 0110 \leftarrow \text{BCD for 6} \\ + \underline{0111} \leftarrow \text{BCD for 7} \\ \hline 1101 \leftarrow \text{invalid sum} \\ \underline{0110} \leftarrow \text{add 6 for correction} \\ \hline \underline{0001} \quad \underline{0011} \leftarrow \text{BCD for 13} \\ \hline 1 \quad 3 \end{array}$$

As shown above, 0110 is added to the invalid sum and produces the correct BCD result. Note that with the addition of 0110, a carry is produced in the second decimal position. This addition must be performed whenever the sum of the two decimal digits is greater than 9.

As another example, take 47 plus 35 in BCD:

$$\begin{array}{r} 47 \quad 0100 \ 0111 \leftarrow \text{BCD for 47} \\ +35 \quad + \underline{0011 \ 0101} \leftarrow \text{BCD for 35} \\ \hline 82 \quad 0111 \ 1100 \leftarrow \text{invalid sum in first digit} \\ \quad \quad 1 \leftarrow \text{add 6 to correct} \\ \quad \quad \quad \underline{0110} \leftarrow \text{add 6 to correct} \\ \hline \underline{1000} \quad \underline{0010} \leftarrow \text{correct BCD sum} \\ \hline 8 \quad 2 \end{array}$$

The addition of the four-bit codes for the 7 and 5 digits results in an invalid sum and is corrected by adding 0110. Note that this generates a carry of 1, which is carried over to be added to the BCD sum of the second-position digits.

Consider the addition of 59 and 38 in BCD:

$$\begin{array}{r}
 \begin{array}{r}
 59 \\
 +38 \\
 \hline
 97
 \end{array}
 \quad
 \begin{array}{r}
 0101 \\
 +0011 \\
 \hline
 1001 \\
 1001 \\
 \hline
 1001 \\
 0110 \\
 \hline
 1001 \\
 0111 \\
 \hline
 9 \quad 7
 \end{array}
 \quad
 \begin{array}{l}
 \downarrow \\
 1 \\
 \left. \begin{array}{l}
 1001 \leftarrow \text{BCD for 59} \\
 1000 \leftarrow \text{BCD for 38} \\
 0001 \leftarrow \text{perform addition} \\
 0110 \leftarrow \text{add 6 to correct} \\
 0111 \leftarrow \text{BCD for 97}
 \end{array}
 \right\}
 \end{array}
 \end{array}$$

Here, the addition of the least significant digits (LSDs) produces a sum of  $17 = 10001$ . This generates a carry into the next digit position to be added to the codes for 5 and 3. Since  $17 > 9$ , a correction factor of 6 must be added to the LSD sum. Addition of this correction does not generate a carry; the carry was already generated in the original addition.

To summarize the BCD addition procedure:

1. Using ordinary binary addition, add the BCD code groups for each digit position.
2. For those positions where the sum is 9 or less, no correction is needed. The sum is in proper BCD form.
3. When the sum of two digits is greater than 9, a correction of 0110 should be added to that sum to get the proper BCD result. This case always produces a carry into the next digit position, either from the original addition (step 1) or from the correction addition.

The procedure for BCD addition is clearly more complicated than straight binary addition. This is also true of the other BCD arithmetic operations. Perform the addition of  $275 + 641$ . Then check the correct procedure below.

$$\begin{array}{r}
 \begin{array}{r}
 275 \\
 +641 \\
 \hline
 916
 \end{array}
 \quad
 \begin{array}{r}
 0010 \quad 0111 \quad 0101 \\
 +0110 \quad 0100 \quad 0001 \\
 \hline
 1000 \quad 1011 \quad 0110 \\
 + \quad \quad 0110 \\
 \hline
 1001 \quad 0001 \quad 0110
 \end{array}
 \quad
 \begin{array}{l}
 \leftarrow \text{BCD for 275} \\
 \leftarrow \text{BCD for 641} \\
 \leftarrow \text{perform addition} \\
 \leftarrow \text{add 6 to correct second digit} \\
 \leftarrow \text{BCD for 916}
 \end{array}
 \end{array}$$

## BCD Subtraction

The process of subtracting BCD numbers is more difficult than addition. It involves a complement-then-add procedure similar to the 2's-complement method. We do not cover it in this book.

### OUTCOME ASSESSMENT QUESTIONS

1. How can you tell when a correction is needed in BCD addition?
2. Represent  $135_{10}$  and  $265_{10}$  in BCD and then perform BCD addition. Check your work by converting the result back to decimal.

## 6-8 HEXADECIMAL ARITHMETIC

### OUTCOMES

Upon completion of this section, you will be able to:

- Calculate the sum of two hexadecimal numbers.
- Calculate the difference of two hexadecimal numbers.
- Recognize the sign of a number when expressed in hexadecimal.

Hex numbers are used extensively in machine-language computer programming and in conjunction with computer memories (i.e., addresses). When working in these areas, you will encounter situations where hex numbers must be added or subtracted.

### Hex Addition

Addition of hexadecimal numbers is done in much the same way as decimal addition, as long as you remember that the largest hex digit is F instead of 9. The following procedure is suggested:

1. Add the two hex digits in decimal, mentally inserting the decimal equivalent for those digits larger than 9.
2. If the sum is 15 or less, it can be directly expressed as a hex digit.
3. If the sum is greater than or equal to 16, subtract 16 and carry a 1 to the next digit position.

The following examples will illustrate the procedure.

#### EXAMPLE 6-6

Add the hex numbers 58 and 24.

#### Solution

$$\begin{array}{r} 58 \\ +24 \\ \hline 7C \end{array}$$

Adding the LSDs (8 and 4) produces 12, which is C in hex. There is no carry into the next digit position. Adding 5 and 2 produces 7.

#### EXAMPLE 6-7

Add the hex numbers 58 and 4B.

#### Solution

$$\begin{array}{r} 58 \\ +4B \\ \hline A3 \end{array}$$

Start by adding 8 and B, mentally substituting decimal 11 for B. This produces a sum of 19. Because 19 is greater than 16, subtract 16 to get 3; write down the 3 and carry a 1 into the next position. This carry is added to the 5 and 4 to produce a sum of 10<sub>10</sub>, which is then converted to hexadecimal A.

**EXAMPLE 6-8**

Add 3AF to 23C.

**Solution**

$$\begin{array}{r} 3AF \\ +23C \\ \hline 5EB \end{array}$$

The sum of F and C is considered as  $15 + 12 = 27_{10}$ . Because this is greater than 16, subtract 16 to get  $11_{10}$ , which is hexadecimal B, and carry a 1 into the second position. Add this carry to A and 3 to obtain E. There is no carry into the MSD position.

**Hex Subtraction**

Remember that hex numbers are just an efficient way to represent binary numbers. Thus, we can subtract hex numbers using the same method we used for binary numbers. The 2's complement of the hex subtrahend will be taken and then *added* to the minuend, and any carry out of the MSD position will be disregarded.

How do we find the 2's complement of a hex number? One way is to convert it to binary, take the 2's complement of the binary equivalent, and then convert it back to hex. This process is illustrated below.

$$\begin{array}{rcccl} & & 73A & & \leftarrow \text{hex number} \\ & \swarrow & & \searrow & \\ 0111 & & 0011 & & 1010 \leftarrow \text{convert to binary} \\ & \swarrow & & \searrow & \\ 1000 & & 1100 & & 0110 \leftarrow \text{take 2's complement} \\ & \swarrow & & \searrow & \\ & & 8C6 & & \leftarrow \text{convert back to hex} \end{array}$$

There is a quicker procedure: subtract *each* hex digit from F; then add 1. Let's try this for the same hex number from the example above.

$$\begin{array}{rcccl} F & F & F & & \\ -7 & -3 & -A & & \leftarrow \text{subtract each digit from F} \\ \hline 8 & C & 5 & & \\ & & +1 & & \leftarrow \text{add 1} \\ \hline 8 & C & 6 & & \leftarrow \text{hex equivalent of 2's complement} \end{array}$$

Try either of the procedures above on the hex number E63. The correct result for the 2's complement is 19D.

**CALCULATOR HINT**

On a hex calculator, you can subtract the hex digits from a string of F's and then add one as we just demonstrated, or you can add one to the string of all F's and then subtract. For example, adding 1 to  $FFF_{16}$  yields  $1000_{16}$ . On the hex calculator enter:

$$1000 - 73A = \quad \text{The answer is } 8C6$$

**EXAMPLE 6-9**

Subtract  $3A5_{16}$  from  $592_{16}$ .

**Solution**

First, convert the subtrahend ( $3A5$ ) to its 2's-complement form by using either method presented above. The result is  $C5B$ . Then add this to the minuend ( $592$ ):

$$\begin{array}{r} 592 \\ + C5B \\ \hline \cancel{1}1ED \\ \uparrow \text{Disregard carry.} \end{array}$$

Ignore the carry out of the MSD addition; the result is  $1ED$ . We can prove that this is correct by adding  $1ED$  to  $3A5$  and checking to see that it equals  $592_{16}$ .

**Hex Representation of Signed Numbers**

The data stored in a microcomputer's internal working memory or on a hard disk or CD ROM are typically stored in bytes (groups of eight bits). The data byte stored in a particular memory location is often expressed in hexadecimal because it is more efficient and less error-prone than expressing it in binary. When the data consists of *signed* numbers, it is helpful to be able to recognize whether a hex value represents a positive or a negative number. For example, Table 6-2 lists the data stored in a small segment of memory starting at address 4000.

**TABLE 6-2**

Hex Address	Stored Binary Data	Hex Value	Decimal Value
4000	00111010	3A	+58
4001	11100101	E5	-29
4002	01010111	57	+87
4003	10000000	80	-128

Each memory location stores a single byte (eight bits), which is the binary equivalent of a signed decimal number. The table also shows the hex equivalent of each byte. For a negative data value, the sign bit (MSB) of the binary number will be a 1; this will always make the MSD of the hex number 8 or greater. When the data value is positive, the sign bit will be a 0, and the MSD of the hex number will be 7 or less. The same holds true no matter how many digits are in the hex number. *When the MSD is 8 or greater, the number being represented is negative; when the MSD is 7 or less, the number is positive.*

**OUTCOME ASSESSMENT QUESTIONS**

1. Add  $67F + 2A4$ .
2. Subtract  $67F - 2A4$ .
3. Which of the following hex numbers represent positive values:  $2F$ ,  $77EC$ ,  $C000$ ,  $6D$ ,  $FFFF$ ?

## 6-9 ARITHMETIC CIRCUITS

### OUTCOME

Upon completion of this section, you will be able to:

- Describe the hardware, at the functional block level, that performs arithmetic in computers.

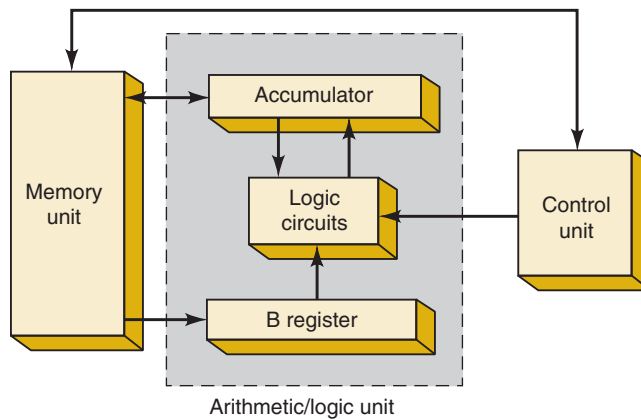
One essential function of most computers and calculators is the performance of arithmetic operations. These operations are all performed in the arithmetic/logic unit of a computer, where logic gates and flip-flops are combined so that they can add, subtract, multiply, and divide binary numbers. These circuits perform arithmetic operations at speeds that are not humanly possible.

We will now study some of the basic arithmetic circuits that are used to perform the arithmetic operations discussed earlier. In some cases, we will go through the actual design process, even though the circuits may be commercially available in integrated-circuit form, to provide more practice in the use of the techniques learned in Chapter 4.

### Arithmetic/Logic Unit

All arithmetic operations take place in the **arithmetic/logic unit (ALU)** of a computer. Figure 6-4 is a block diagram showing the major elements included in a typical ALU. The main purpose of the ALU is to accept binary data that are stored in the memory and to execute arithmetic and logic operations on these data according to instructions from the control unit.

**FIGURE 6-4** Functional parts of an ALU.



The arithmetic/logic unit contains at least two flip-flop registers: the *B register* and the **accumulator register**. It also contains combinational logic, which performs the arithmetic and logic operations on the binary numbers that are stored in the *B register* and the accumulator. A typical sequence of operations may occur as follows:

1. The control unit receives an instruction (from the memory unit) specifying that a number stored in a particular memory location (address) is to be added to the number presently stored in the accumulator register.
2. The number to be added is transferred from memory to the *B register*.



- The number in the *B* register and the number in the accumulator register are added together in the logic circuits (upon command from the control unit). The resulting sum is then sent to the accumulator to be stored.
- The new number in the accumulator can remain there so that another number can be added to it or, if the particular arithmetic process is finished, it can be transferred to memory for storage.

These steps should make it apparent how the accumulator register derives its name. This register “accumulates” the sums that occur when performing successive additions between new numbers acquired from memory and the previously accumulated sum. In fact, for any arithmetic problem containing several steps, the accumulator usually contains the results of the intermediate steps as they are completed as well as the final result when the problem is finished.

### OUTCOME ASSESSMENT QUESTIONS

- Name the three blocks of an arithmetic logic unit.
- The Accumulator and *B* are registers. What fundamental block makes up these registers?
- Why is the *A* register called an accumulator?

## 6-10 PARALLEL BINARY ADDER

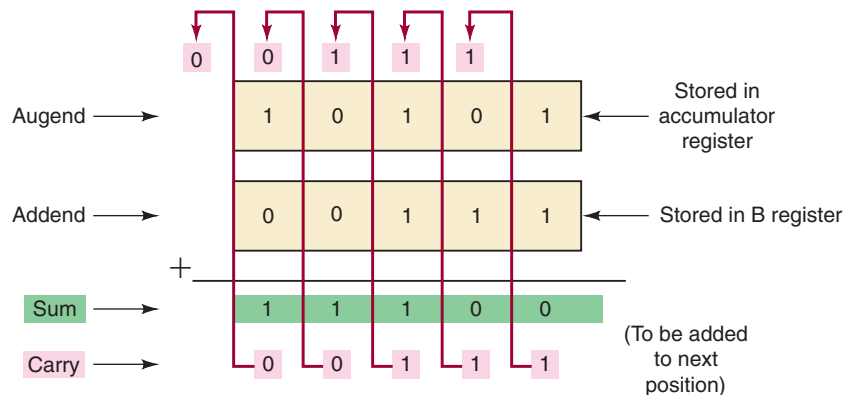
### OUTCOMES

Upon completion of this section, you will be able to:

- Decompose a binary adder to its fundamental block (Full adder), which adds one column of bits.
- Connect Full Adder blocks to form an adder for binary numbers of any size.
- Predict logic levels at any point in an adder circuit.

Computers and calculators perform the addition operation on two binary numbers at a time, where each binary number can have several binary digits. Figure 6-5 illustrates the addition of two five-bit numbers. The augend

**FIGURE 6-5** Typical binary addition process.



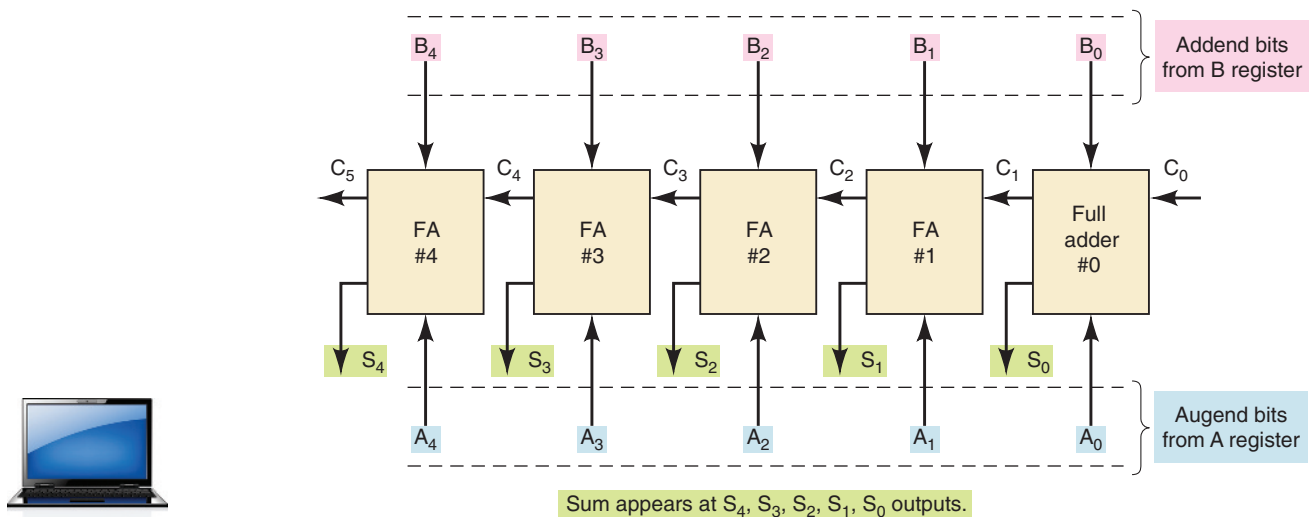
is stored in the accumulator register; that is, the accumulator contains five FFs, storing the values 10101 in successive FFs. Similarly, the addend, the number that is to be added to the augend, is stored in the *B* register (in this case, 00111).

The addition process starts by adding the least significant bits (LSBs) of the augend and addend. Thus,  $1 + 1 = 10$ , which means that the *sum* for that position is 0, with a *carry* of 1.

This carry must be added to the next position along with the augend and addend bits in that position. Thus, in the second position,  $1 + 0 + 1 = 10$ , which is again a sum of 0 and a carry of 1. This carry is added to the next position together with the augend and addend bits in that position, and so on, for the remaining positions, as shown in Figure 6-5.

At each step in this addition process, we are performing the addition of three bits: the augend bit, the addend bit, and a carry bit from the previous position. The result of the addition of these three bits produces two bits: a *sum* bit, and a *carry* bit that is to be added to the next position. It should be clear that the same process is followed for each bit position. Thus, if we can design a logic circuit that can duplicate this process, then all we have to do is to use the identical circuit for each of the bit positions. This is illustrated in Figure 6-6.

In this diagram, variables  $A_4, A_3, A_2, A_1,$  and  $A_0$  represent the bits of the augend that are stored in the accumulator (which is also called the *A* register). Variables  $B_4, B_3, B_2, B_1,$  and  $B_0$  represent the bits of the addend stored in the *B* register. Variables  $C_4, C_3, C_2, C_1,$  and  $C_0$  represent the carry bits into the corresponding positions. Variables  $S_4, S_3, S_2, S_1, S_0$  are the sum output bits for each position. Corresponding bits of the augend and addend are fed to a logic circuit called a **full adder (FA)**, along with a carry bit from the previous position. For example, bits  $A_1$  and  $B_1$  are fed into full adder 1 along with  $C_1$ , which is the carry bit produced by the addition of the  $A_0$  and  $B_0$  bits. Bits  $A_0$  and  $B_0$  are fed into full adder 0 along with  $C_0$ .  $A_0$  and  $B_0$  are the LSBs of the augend and addend, so it appears that  $C_0$  would always have to be 0 because there can be no carry into that position. We shall see, however, that there will be situations when  $C_0$  can also be 1.



**FIGURE 6-6** Block diagram of a five-bit parallel adder circuit using full adders.

The full-adder circuit used in each position has three inputs: an  $A$  bit, a  $B$  bit, and a  $C$  bit. It also produces two outputs: a sum bit and a carry bit. For example, full adder 0 has inputs  $A_0$ ,  $B_0$ , and  $C_0$ , and it produces outputs  $S_0$  and  $C_1$ . Full adder 1 had  $A_1$ ,  $B_1$ , and  $C_1$  as inputs and  $S_1$  and  $C_2$  as outputs, and so on. This arrangement is repeated for as many positions as there are in the augend and addend. Although this illustration is for five-bit numbers, in computers the numbers usually range from 8 to 64 bits.

The arrangement in Figure 6-6 is called a **parallel adder** because all of the bits of the augend and addend are present and are fed into the adder circuits *simultaneously*. This means that the additions in each position are taking place at the same time. This is different from how we add on paper, taking each position one at a time starting with the LSB. Clearly, parallel addition is extremely fast. More will be said about this later.

### OUTCOME ASSESSMENT QUESTIONS

1. How many inputs does a full adder have? How many outputs?
2. Assume the following input levels in Figure 6-6:  $A_4A_3A_2A_1A_0 = 01001$ ;  $B_4B_3B_2B_1B_0 = 00111$ ;  $C_0 = 0$ .
  - (a) What are the logic levels at the outputs of FA #2?
  - (b) What is the logic level at the  $C_5$  output?

## 6-11 DESIGN OF A FULL ADDER

### OUTCOMES

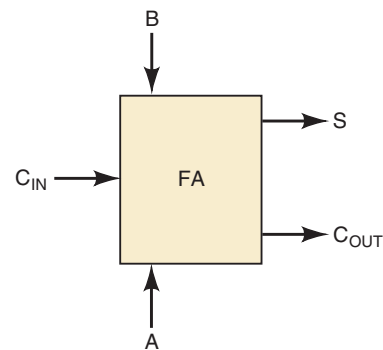
Upon completion of this section, you will be able to:

- Represent a Full Adder as a block diagram.
- Specify the function of a Full Adder using a truth table.
- Use normal combinational design techniques to produce the logic circuit that will add.

Now that we know the function of the full adder, we can design a logic circuit that will perform this function. First, we must construct a truth table showing the various input and output values for all possible cases. Figure 6-7 shows the truth table having three inputs,  $A$ ,  $B$ , and  $C_{IN}$ , and two outputs,  $S$

**FIGURE 6-7** Truth table for a full-adder circuit.

Augend bit input	Addend bit input	Carry bit input	Sum bit output	Carry bit output
A	B	$C_{IN}$	S	$C_{OUT}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



and  $C_{OUT}$ . There are eight possible cases for the three inputs, and for each case the desired output values are listed. For example, consider the case  $A = 1$ ,  $B = 0$ , and  $C_{IN} = 1$ . The full adder (FA) must add these bits to produce a sum ( $S$ ) of 0 and a carry ( $C_{OUT}$ ) of 1. Check the other cases to be sure they are understood.

Because there are two outputs, we will design the circuitry for each output individually, starting with the  $S$  output. The truth table shows that there are four cases where  $S$  is to be a 1. Using the sum-of-products method, we can write the expression for  $S$  as

$$S = \bar{A}\bar{B}C_{IN} + \bar{A}B\bar{C}_{IN} + A\bar{B}\bar{C}_{IN} + ABC_{IN} \quad (6-1)$$

We can now try to simplify this expression by factoring. Unfortunately, none of the terms in the expression has two variables in common with any of the other terms. However,  $\bar{A}$  can be factored from the first two terms, and  $A$  can be factored from the last two terms:

$$S = \bar{A}(\bar{B}C_{IN} + B\bar{C}_{IN}) + A(\bar{B}\bar{C}_{IN} + BC_{IN})$$

The first term in parentheses should be recognized as the exclusive-OR combination of  $B$  and  $C_{IN}$ , which can be written as  $B \oplus C_{IN}$ . The second term in parentheses should be recognized as the exclusive-NOR of  $B$  and  $C_{IN}$ , which can be written as  $\overline{B \oplus C_{IN}}$ . Thus, the expression for  $S$  becomes

$$S = \bar{A}(B \oplus C_{IN}) + A(\overline{B \oplus C_{IN}})$$

If we let  $X = B \oplus C_{IN}$ , this can be written as

$$S = \bar{A} \cdot X + A \cdot \bar{X} = A \oplus X$$

which is simply the exclusive-OR of  $A$  and  $X$ . Replacing the expression for  $X$ , we have

$$S = A \oplus [B \oplus C_{IN}] \quad (6-2)$$

Consider now the output  $C_{OUT}$  in the truth table of Figure 6-7. We can write the sum-of-products expression for  $C_{OUT}$  as follows:

$$C_{OUT} = \bar{A}BC_{IN} + A\bar{B}C_{IN} + AB\bar{C}_{IN} + ABC_{IN}$$

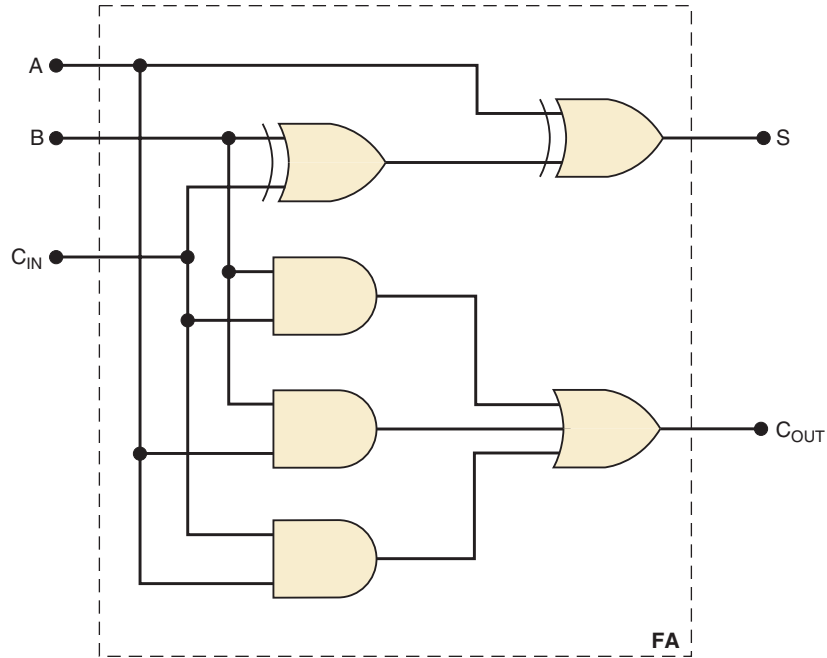
This expression can be simplified by factoring. We will employ the trick introduced in Chapter 4, whereby we will use the  $ABC_{IN}$  term *three* times because it has common factors with each of the other terms. Hence,

$$\begin{aligned} C_{OUT} &= BC_{IN}(\bar{A} + A) + AC_{IN}(\bar{B} + B) + AB(\bar{C}_{IN} + C_{IN}) \\ &= BC_{IN} + AC_{IN} + AB \end{aligned} \quad (6-3)$$

This expression cannot be simplified further.

Expressions (6-2) and (6-3) can be implemented as shown in Figure 6-8. Several other implementations can be used to produce the same expressions for  $S$  and  $C_{OUT}$ , none of which has any particular advantage over those

**FIGURE 6-8** Complete circuitry for a full adder.



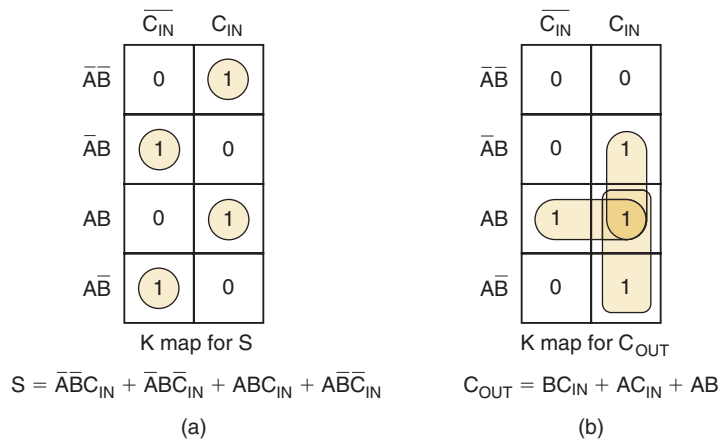
shown. The complete circuit with inputs  $A$ ,  $B$ , and  $C_{IN}$  and outputs  $S$  and  $C_{OUT}$  represents the full adder. Each of the FAs in Figure 6-6 contains this same circuitry (or its equivalent).

### K-Map Simplification

We simplified the expressions for  $S$  and  $C_{OUT}$  using algebraic methods. The K-map method can also be used. Figure 6-9(a) shows the K map for the  $S$  output. This map has no adjacent 1s, and so there are no pairs or quads to loop. Thus, the expression for  $S$  cannot be simplified using the K map. This points out a limitation of the K-map method compared to the algebraic method. We were able to simplify the expression for  $S$  through factoring and the use of XOR and XNOR operations.

The K map for the  $C_{OUT}$  output is shown in Figure 6-9(b). The three pairs that are looped will produce the same expression obtained from the algebraic method.

**FIGURE 6-9** K mappings for the full-adder outputs.



## Half Adder

The FA operates on three inputs to produce a sum and carry output. In some cases, a circuit is needed that will add only two input bits, to produce a sum and carry output. An example would be the addition of the LSB position of two binary numbers where there is no carry input to be added. A special logic circuit can be designed to take *two* input bits,  $A$  and  $B$ , and to produce sum ( $S$ ) and carry ( $C_{OUT}$ ) outputs. This circuit is called a **half adder (HA)**. Its operation is similar to that of an FA except that it operates on only two bits. We shall leave the design of the HA as an exercise at the end of the chapter.

### OUTCOME ASSESSMENT QUESTIONS

1. Without referring back to the text, repeat the design of a Full adder and a Half Adder circuit.

## 6-12 COMPLETE PARALLEL ADDER WITH REGISTERS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Use flip-flops from Chapter 5 and combinational logic circuits to create an arithmetic adder circuit with addend ( $B$ ) register and augend/accumulator register.
- Represent the operation of the arithmetic unit as a timing diagram.
- Predict the output of the arithmetic unit for any input.

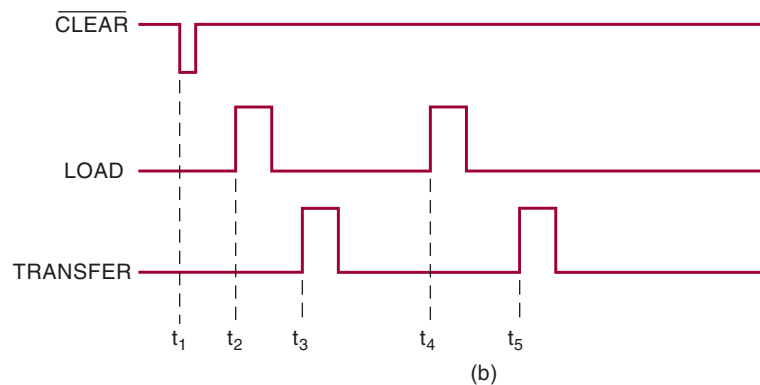
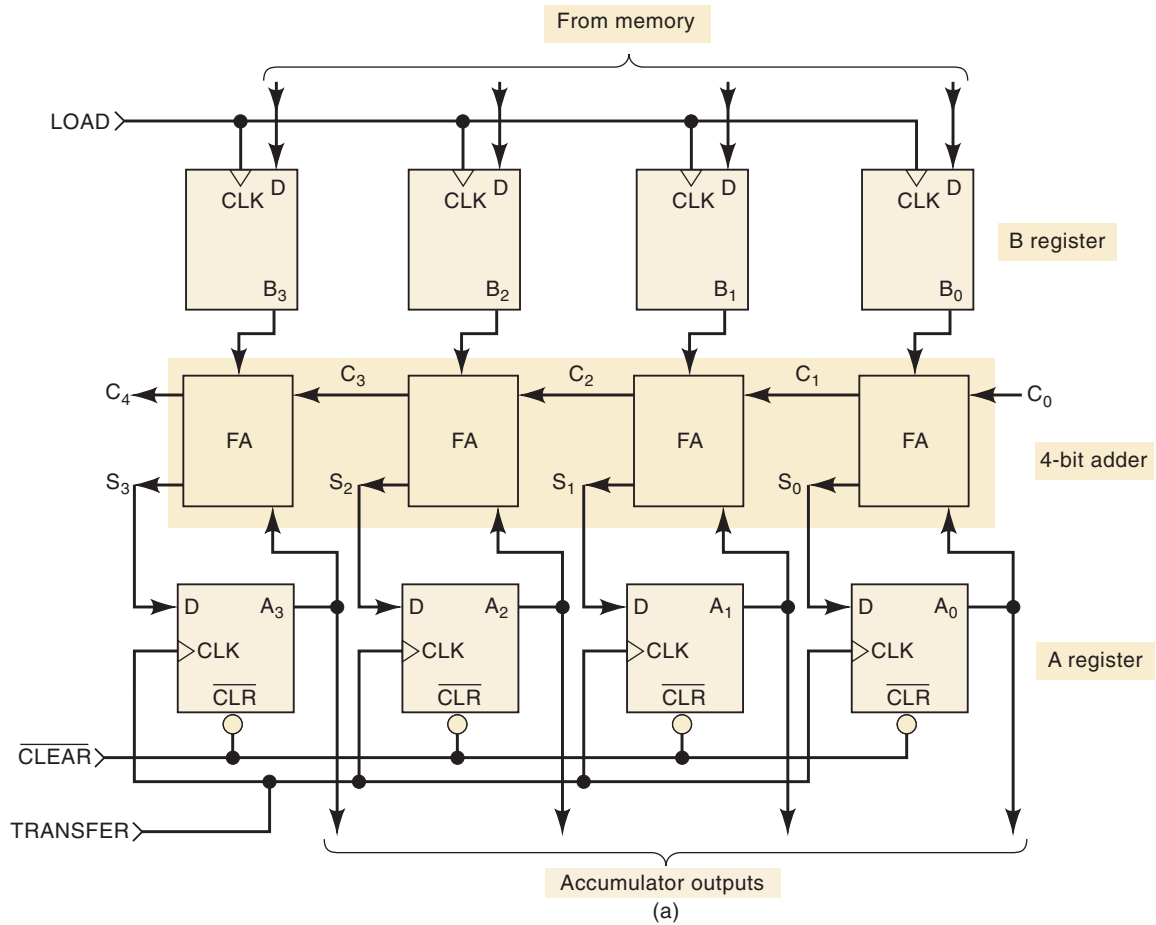
In a computer, the numbers that are to be added are stored in FF registers. Figure 6-10 shows the diagram of a four-bit parallel adder, including the storage registers. The augend bits  $A_3$  through  $A_0$  are stored in the accumulator ( $A$  register); the addend bits  $B_3$  through  $B_0$  are stored in the  $B$  register. Each of these registers is made up of D flip-flops for easy transfer of data.

The contents of the  $A$  register (i.e., the binary number stored in  $A_3$  through  $A_0$ ) is added to the contents of the  $B$  register by the four FAs, and the sum is produced at outputs  $S_3$  through  $S_0$ .  $C_4$  is the carry out of the fourth FA, and it can be used as the carry input to a fifth FA, or as an *overflow* bit to indicate that the sum exceeds 1111.

Note that the sum outputs are connected to the  $D$  inputs of the  $A$  register. This will allow the sum to be parallel-transferred into the  $A$  register on the positive-going transition (PGT) of the TRANSFER pulse. In this way, the sum can be stored in the  $A$  register.

Also note that the  $D$  inputs of the  $B$  register are coming from the computer's memory, so that binary numbers from memory will be parallel-transferred into the  $B$  register on the PGT of the LOAD pulse. In most computers, there is also provision for parallel-transferring binary numbers from memory into the accumulator ( $A$  register). For simplicity, the circuitry necessary for performing this transfer is not shown in this diagram; it will be addressed in an end-of-chapter exercise.

Finally, note that the  $A$  register outputs are available for transfer to other locations such as another computer register or the computer's memory. This will make the adder circuit available for a new set of numbers.



**FIGURE 6-10** (a) Complete four-bit parallel adder with registers; (b) signals used to add binary numbers from memory and store their sum in the accumulator.

### Register Notation

Before we go through the complete process of how this circuit adds two binary numbers, it will be helpful to introduce some notation that makes it easy to describe the contents of a register and data transfer operations.

Whenever we want to give the levels that are present at each FF in a register or at each output of a group of outputs, we will use brackets, as illustrated below:

$$[A] = 1011$$

This is the same as saying that  $A_3 = 1$ ,  $A_2 = 0$ ,  $A_1 = 1$ ,  $A_0 = 1$ . In other words, think of  $[A]$  as representing “the contents of register  $A$ .”

Whenever we want to indicate the transfer of data to or from a register, we will use an arrow, as illustrated below:

$$[B] \rightarrow [A]$$

This means that the contents of the  $B$  register have been transferred to the  $A$  register. The old contents of the  $A$  register will be lost as a result of this operation, and the  $B$  register will be unchanged. This type of notation is very common, especially in data books describing microprocessor and microcontroller operations. In many ways, it is very similar to the notation used to refer to bit-array data objects using hardware description languages.

### Sequence of Operations

We will now describe the process by which the circuit of Figure 6-10 will add the binary numbers 1001 and 0101. Assume that  $C_0 = 0$ ; that is, there is no carry into the LSB position.

1.  $[A] = 0000$ . A  $\overline{\text{CLEAR}}$  pulse is applied to the asynchronous inputs ( $\overline{\text{CLR}}$ ) of each FF in register  $A$ . This occurs at time  $t_1$ .
2.  $[M] \rightarrow [B]$ . This first binary number is transferred from memory ( $M$ ) to the  $B$  register. In this case, the binary number 1001 is loaded into register  $B$  on the PGT of the LOAD pulse at  $t_2$ .
- 3.\* $[S] \rightarrow [A]$ . With  $[B] = 1001$  and  $[A] = 0000$ , the full adders produce a sum of 1001; that is,  $[S] = 1001$ . These sum outputs are transferred into the  $A$  register on the PGT of the TRANSFER pulse at  $t_3$ . This makes  $[A] = 1001$ .
4.  $[M] \rightarrow [B]$ . The second binary number, 0101, is transferred from memory into the  $B$  register on the PGT of the second LOAD pulse at  $t_4$ . This makes  $[B] = 0101$ .
5.  $[S] \rightarrow [A]$ . With  $[B] = 0101$  and  $[A] = 1001$ , the FAs produce  $[S] = 1110$ . These sum outputs are transferred into the  $A$  register when the second TRANSFER pulse occurs at  $t_5$ . Thus,  $[A] = 1110$ .
6. At this point, the sum of the two binary numbers is present in the accumulator. In most computers, the contents of the accumulator,  $[A]$ , will usually be transferred to the computer’s memory so that the adder circuit can be used for a new set of numbers. The circuitry that performs this  $[A] \rightarrow [M]$  transfer is not shown in Figure 6-10.

#### OUTCOME ASSESSMENT QUESTIONS

1. Suppose that four different four-bit numbers are to be taken from memory and added by the circuit of Figure 6-10. How many  $\overline{\text{CLEAR}}$  pulses will be needed? How many TRANSFER pulses? How many LOAD pulses?
2. Determine the contents of the  $A$  register after the following sequence of operations:  $[A] = 0000$ ,  $[0110] \rightarrow [B]$ ,  $[S] \rightarrow [A]$ ,  $[1110] \rightarrow [B]$ ,  $[S] \rightarrow [A]$ .

\*Even though  $S$  is not a register, we will use  $[S]$  to represent the group of  $S$  outputs.



## 6-13 CARRY PROPAGATION

### OUTCOMES

Upon completion of this section, you will be able to:

- Calculate the maximum delay for an addition of  $n$  bits.
- Describe strategies to reduce these delays.

The parallel adder of Figure 6-10 performs additions at a relatively high speed because it adds the bits from each position simultaneously. However, its speed is limited by an effect called **carry propagation** or **carry ripple**, which can best be explained by considering the following addition:

$$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$

Addition of the LSB position produces a carry into the second position. This carry, when added to the bits of the second position, produces a carry into the third position. The latter carry, when added to the bits of the third position, produces a carry into the last position. The key point to notice in this example is that the sum bit generated in the *last* position (MSB) depended on the carry that was generated by the addition in the *first* position (LSB).

Looking at this from the viewpoint of the circuit of Figure 6-10,  $S_3$  out of the last full adder depends on  $C_1$  out of the first full adder. But the  $C_1$  signal must pass through three FAs before it produces  $S_3$ . What this means is that the  $S_3$  output will not reach its correct value until  $C_1$  has propagated through the intermediate FAs. This represents a time delay that depends on the propagation delay produced in each FA. For example, if each FA has a propagation delay of 40 ns, then  $S_3$  will not reach its correct level until 120 ns after  $C_1$  is generated. This means that the add command pulse cannot be applied until 160 ns after the augend and addend numbers are present in the FF registers (the extra 40 ns is due to the delay of the LSB full adder, which generates  $C_1$ ).

Obviously, the situation becomes much worse if we extend the adder circuitry to add a greater number of bits. If the adder were handling 32-bit numbers, the carry propagation delay could be  $1280 \text{ ns} = 1.28 \mu\text{s}$ . The add pulse could not be applied until at least  $1.28 \mu\text{s}$  after the numbers were present in the registers.

This magnitude of delay is prohibitive for high-speed computers. Fortunately, logic designers have come up with several ingenious schemes for reducing this delay. One of the schemes, called **look-ahead carry**, utilizes logic gates to look at the lower-order bits of the augend and addend to see if a higher-order carry is to be generated. For example, it is possible to build a logic circuit with  $B_2, B_1, B_0, A_2, A_1,$  and  $A_0$  as inputs and  $C_3$  as an output. This logic circuit would have a shorter delay than is obtained by the carry propagation through the FAs. This scheme requires a large amount of extra circuitry but is necessary to produce high-speed adders. Many high-speed adders available in integrated-circuit form utilize the look-ahead carry or a similar technique for reducing overall propagation delays.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Assume that it takes 2 ns for the output of a full adder to settle after the inputs have changed. For a 16-bit adder, what is the longest amount of time (worst case) to add two numbers?
2. What is the highest frequency that the accumulator register can be clocked?
3. How can an adder operate at higher frequencies?

## 6-14 INTEGRATED-CIRCUIT PARALLEL ADDER

### OUTCOMES

Upon completion of this section, you will be able to:

- Combine standard functional blocks to form adders of any size.
- Calculate maximum speed of operation.
- Predict logic state at any point in the circuit.

Several parallel adders are available as ICs. The most common is a four-bit parallel adder IC that contains four interconnected FAs and the look-ahead carry circuitry needed for high-speed operation. The 7483A, 74LS83A, 74LS283, and 74HC283 are all four-bit parallel-adder chips.

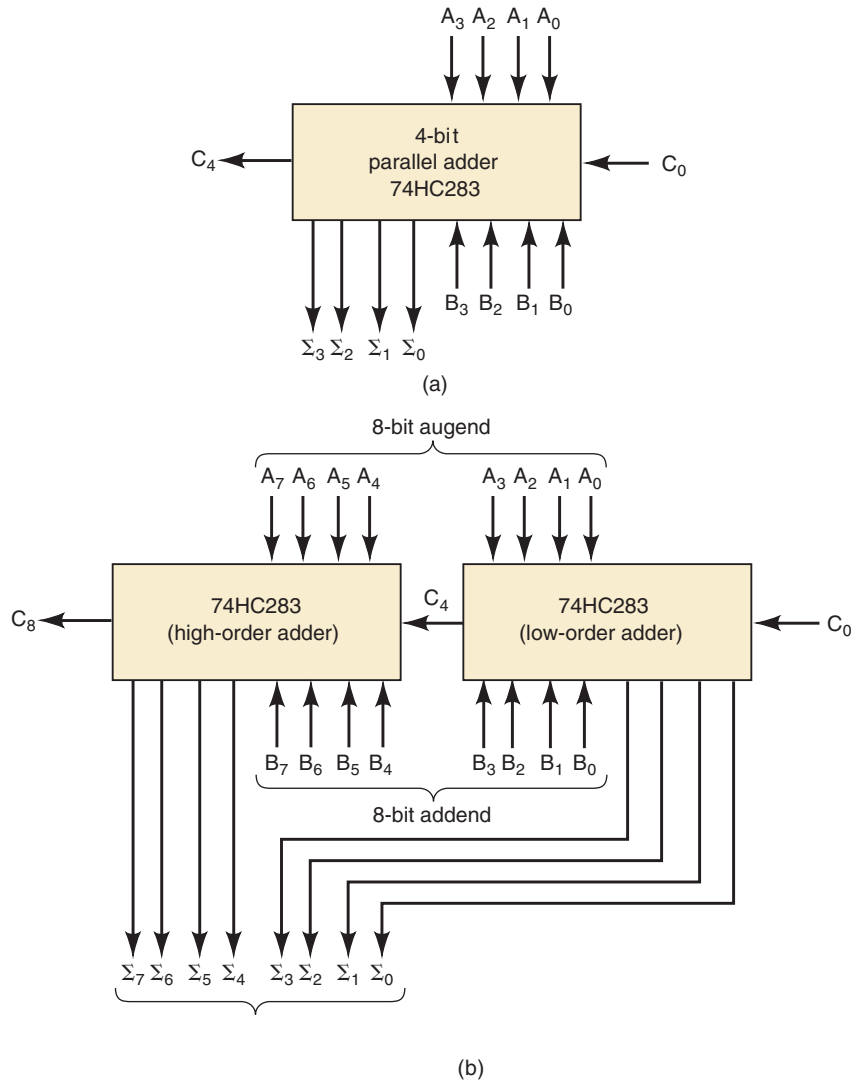
Figure 6-11(a) shows the functional symbol for the 74HC283 four-bit parallel adder (and its equivalents). The inputs to this IC are two four-bit numbers,  $A_3A_2A_1A_0$  and  $B_3B_2B_1B_0$ , and the carry,  $C_0$ , into the LSB position. The outputs are the sum bits and the carry,  $C_4$ , out of the MSB position. The sum bits are labeled  $\Sigma_3\Sigma_2\Sigma_1\Sigma_0$ , where  $\Sigma$  is the Greek capital letter *sigma*. The  $\Sigma$  label is just a common alternative to the S label for a sum bit.

### Cascading Parallel Adders

Two or more IC adders can be connected together (cascaded) to accomplish the addition of larger binary numbers. Figure 6-11(b) shows two 74HC283 adders connected to add two eight-bit numbers  $A_7A_6A_5A_4A_3A_2A_1A_0$  and  $B_7B_6B_5B_4B_3B_2B_1B_0$ . The adder on the right adds the lower-order bits of the numbers. The adder on the left adds the higher-order bits *plus* the  $C_4$  carry out of the lower-order adder. The eight sum outputs are the resultant sum of the two eight-bit numbers.  $C_8$  is the carry out of the MSB position. It can be used as the carry input to a third adder stage if larger binary numbers are to be added.

The look-ahead carry feature of the 74HC283 speeds up the operation of this two-stage adder because the logic level at  $C_4$ , the carry out of the lower-order stage, is generated more rapidly than it would be if there were no look-ahead carry circuitry on the 74HC283 chip. This allows the higher-order stage to produce its sum outputs more quickly.

**FIGURE 6-11** (a) Block symbol for the 74HC283 four-bit parallel adder; (b) cascading two 74HC283s.



**EXAMPLE 6-10**

Determine the logic levels at the inputs and outputs of the eight-bit adder in Figure 6-11(b) when  $72_{10}$  is added to  $137_{10}$ .

**Solution**

First, convert each number to an eight-bit binary number:

$$\begin{aligned} 137 &= 10001001 \\ 72 &= 01001000 \end{aligned}$$

These two binary values will be applied to the *A* and *B* inputs; that is, the *A* inputs will be 10001001 from left to right, and the *B* inputs will be 01001000 from left to right. The adder will produce the binary sum of the two numbers:

$$\begin{aligned} [A] &= 10001001 \\ [B] &= \underline{01001000} \\ [\Sigma] &= 11010001 \end{aligned}$$

The sum outputs will read 11010001 from left to right. There is no overflow into the  $C_8$  bit, and so it will be a 0.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. How many 74HC283 chips are needed to add two 20-bit numbers?
2. If a 74HC283 has a maximum propagation delay of 30 ns from  $C_0$  to  $C_4$ , what will be the total propagation delay of a 32-bit adder constructed from 74HC283s?
3. What will be the logic level at  $C_4$  in Example 6-10 ?

## 6-15 2'S-COMPLEMENT CIRCUITS

### OUTCOMES

Upon completion of this section, you will be able to:

- Combine a standard adder block with combinational circuits and registers to create an adder/subtractor.
- Predict logic levels at any point in the system.
- Interpret the results as signed integers.

Most computers use the 2's-complement system to represent negative numbers and to perform subtraction. The operations of addition and subtraction of signed numbers can be performed using only the addition operation if we use the 2's-complement form to represent negative numbers.

### Addition

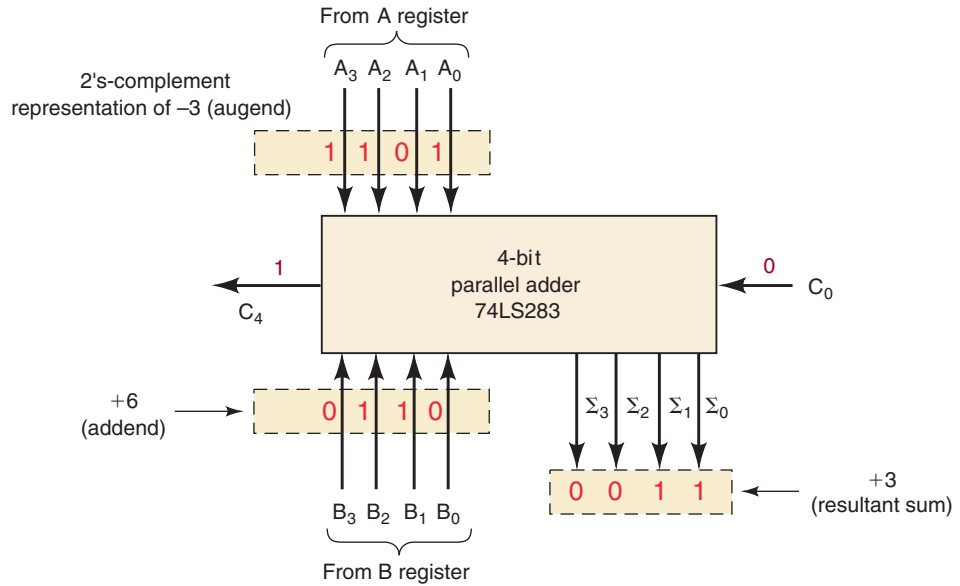
Positive and negative numbers, including the sign bits, can be added together in the basic parallel-adder circuit when the negative numbers are in 2's-complement form. This is illustrated in Figure 6-12 for the addition of  $-3$  and  $+6$ . The  $-3$  is represented in its 2's-complement form as 1101, where the first 1 is the sign bit; the  $+6$  is represented as 0110, with the first zero as the sign bit. These numbers are stored in their corresponding registers. The four-bit parallel adder produces sum outputs of 0011, which represents  $+3$ . The  $C_4$  output is 1, but remember that it is disregarded in the 2's-complement method.

### Subtraction

When the 2's-complement system is used, the number to be subtracted (the subtrahend) is changed to its 2's complement and then *added* to the minuend (the number the subtrahend is being subtracted from). For example, we can assume that the minuend is already stored in the accumulator ( $A$  register). The subtrahend is then placed in the  $B$  register (in a computer it would be transferred here from memory) and is changed to its 2's-complement form before it is added to the number in the  $A$  register. The sum outputs of the adder circuit now represent the *difference* between the minuend and the subtrahend.

The parallel-adder circuit that we have been discussing can be adapted to perform the subtraction described above if we provide a means for taking the 2's complement of the  $B$  register number. The 2's complement of a binary number is obtained by complementing (inverting) each bit and then adding 1 to the LSB. Figure 6-13 shows how this can be accomplished. The *inverted* outputs of the  $B$  register are used rather than the normal outputs;

**FIGURE 6-12** Parallel adder used to add and subtract numbers in 2's-complement system.



that is,  $\bar{B}_0, \bar{B}_1, \bar{B}_2,$  and  $\bar{B}_3$  are fed to the adder inputs (remember,  $B_3$  is the sign bit). This takes care of complementing each bit of the  $B$  number. Also,  $C_0$  is made a logical 1, so that it adds an extra 1 into the LSB of the adder; this accomplishes the same effect as adding 1 to the LSB of the  $B$  register for forming the 2's complement.

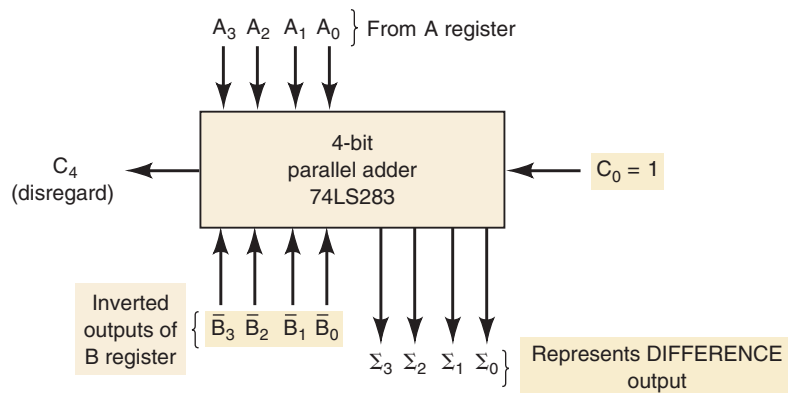
The outputs  $\Sigma_3$  to  $\Sigma_0$  represent the results of the subtraction operation. Of course,  $\Sigma_3$  is the sign bit of the result and indicates whether the result is + or -. The carry output  $C_4$  is again disregarded.

To help clarify this operation, study the following steps for subtracting +6 from +4:

1. +4 is stored in the  $A$  register as 0100.
2. +6 is stored in the  $B$  register as 0110.
3. The inverted outputs of the  $B$ -register FFs (1001) are fed to the adder.
4. The parallel-adder circuitry adds  $[A] = 0100$  to  $[\bar{B}] = 1001$  along with a carry,  $C_0 = 1$ , into the LSB. The operation is shown below.

$$\begin{array}{r}
 1 \leftarrow C_0 \\
 0100 \leftarrow [A] \\
 + 1001 \leftarrow [\bar{B}] \\
 \hline
 1110 \leftarrow [\Sigma] = [A] - [B]
 \end{array}$$

**FIGURE 6-13** Parallel adder used to perform subtraction ( $A - B$ ) using the 2's-complement system. The bits of the subtrahend ( $B$ ) are inverted, and  $C_0 = 1$  to produce the 2's complement.





of the numbers stored in the  $A$  and  $B$  registers. When the SUB level is HIGH, the circuit subtracts the  $B$ -register number from the  $A$ -register number. The operation is described as follows:

1. Assume that  $ADD = 1$  and  $SUB = 0$ . The  $SUB = 0$  *disables* (inhibits) AND gates 2, 4, 6, and 8, holding their outputs at 0. The  $ADD = 1$  *enables* AND gates 1, 3, 5, and 7, allowing their outputs to pass the  $B_0$ ,  $B_1$ ,  $B_2$ , and  $B_3$  levels, respectively.
2. The levels  $B_0$  to  $B_3$  pass through the OR gates into the four-bit parallel adder to be added to the bits  $A_0$  to  $A_3$ . The *sum* appears at the outputs  $\Sigma_0$  to  $\Sigma_3$ .
3. Note that  $SUB = 0$  causes a carry  $C_0 = 0$  into the adder.
4. Now assume that  $ADD = 0$  and  $SUB = 1$ . The  $ADD = 0$  inhibits AND gates 1, 3, 5, and 7. The  $SUB = 1$  enables AND gates 2, 4, 6, and 8, so that their outputs pass the  $\bar{B}_0$ ,  $\bar{B}_1$ ,  $\bar{B}_2$ , and  $\bar{B}_3$  levels, respectively.
5. The levels  $\bar{B}_0$  to  $\bar{B}_3$  pass through the OR gates into the adder to be added to the bits  $A_0$  to  $A_3$ . Note also that  $C_0$  is now 1. Thus, the  $B$ -register number has essentially been converted to its 2's complement.
6. The *difference* appears at the *outputs*  $\Sigma_0$  to  $\Sigma_3$ .

Circuits like the adder/subtractor of Figure 6-14 are used in computers because they provide a relatively simple means for adding and subtracting signed binary numbers. In most computers, the outputs present at the  $\Sigma$  output lines are usually transferred into the  $A$  register (accumulator) so that the results of the addition or subtraction always end up stored in the  $A$  register. This is accomplished by applying a TRANSFER pulse to the  $CLK$  inputs of register  $A$ .

### OUTCOME ASSESSMENT QUESTIONS

1. Why does  $C_0$  have to be a 1 in order to use the adder circuit in Figure 6-13 as a subtractor?
2. Assume that  $[A] = 0011$  and  $[B] = 0010$  in Figure 6-14. If  $ADD = 1$  and  $SUB = 0$ , determine the logic levels at the OR gate outputs.
3. Repeat question 2 for  $ADD = 0$ ,  $SUB = 1$ .
4. *True or false:* When the adder/subtractor circuit is used for subtraction, the 2's complement of the subtrahend appears at the input of the adder.

## 6-16 ALU INTEGRATED CIRCUITS

### OUTCOMES

Upon completion of this section, you will be able to:

- Predict the output of standard ALU blocks given any input combinations.
- Combine standard ALU blocks to produce an ALU of any size.

Several integrated circuits are called arithmetic/logic units (ALUs), even though they do not have the full capabilities of a computer's arithmetic/logic unit. These ALU chips are capable of performing several different

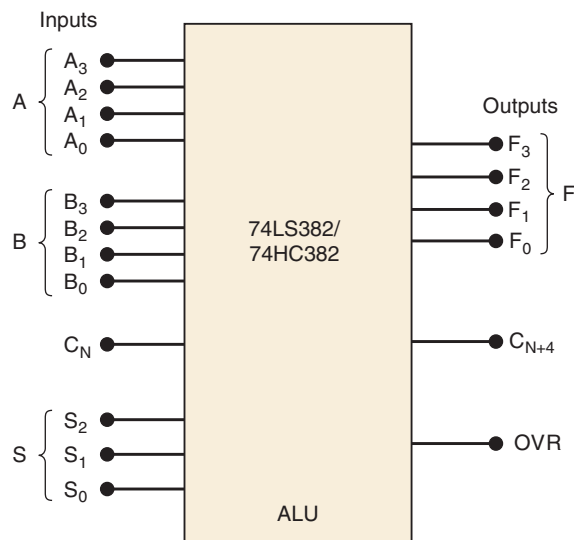
arithmetic and logic operations on binary data inputs. The specific operation that an ALU IC is to perform is determined by a specific binary code applied to its function-select inputs. Some of the ALU ICs are fairly complex, and it would require a great amount of time and space to explain and illustrate their operation. In this section, we will use a relatively simple, yet useful, ALU chip to show the basic concepts behind all ALU chips. The ideas presented here can then be extended to the more complex devices.

### The 74LS382/74HC382 ALU

Figure 6-15(a) shows the block symbol for an ALU that is available as a 74LS382 (TTL) and as a 74HC382 (CMOS). This 20-pin IC operates on two four-bit input numbers,  $A_3A_2A_1A_0$  and  $B_3B_2B_1B_0$ , to produce a four-bit output result  $F_3F_2F_1F_0$ . This ALU can perform *eight* different operations. At any given time, the operation that it is performing depends on the input code applied to the function-select inputs  $S_2S_1S_0$ . The table in Figure 6-15(b) shows the eight available operations. We will now describe each of these operations.

**CLEAR OPERATION** With  $S_2S_1S_0 = 000$ , the ALU will *clear* all of the bits of the  $F$  output so that  $F_3F_2F_1F_0 = 0000$ .

**ADD OPERATION** With  $S_2S_1S_0 = 011$  the ALU will add  $A_3A_2A_1A_0$  to  $B_3B_2B_1B_0$  to produce their sum at  $F_3F_2F_1F_0$ . For this operation,  $C_N$  is the carry into the LSB position, and it must be made a 0.  $C_{N+4}$  is the carry output from the MSB position. *OVR* is the overflow indicator output; it detects overflow when signed numbers are being used. *OVR* will be a 1



Function Table				
$S_2$	$S_1$	$S_0$	Operation	Comments
0	0	0	CLEAR	$F_3F_2F_1F_0 = 0000$
0	0	1	B minus A	} Needs $C_N = 1$
0	1	0	A minus B	
0	1	1	A plus B	Needs $C_N = 0$
1	0	0	$A \oplus B$	Exclusive-OR
1	0	1	$A + B$	OR
1	1	0	AB	AND
1	1	1	PRESET	$F_3F_2F_1F_0 = 1111$

Notes: S inputs select operation.  
OVR = 1 for signed-number overflow.

(b)

A = 4-bit input number  
B = 4-bit input number  
 $C_N$  = carry into LSB position  
S = 3-bit operation select inputs  
F = 4-bit output number  
 $C_{N+4}$  = carry out of MSB position  
OVR = overflow indicator

(a)

**FIGURE 6-15** (a) Block symbol for 74LS382/74HC382 ALU chip; (b) function table showing how select inputs ( $S$ ) determine what operation is to be performed on  $A$  and  $B$  inputs.



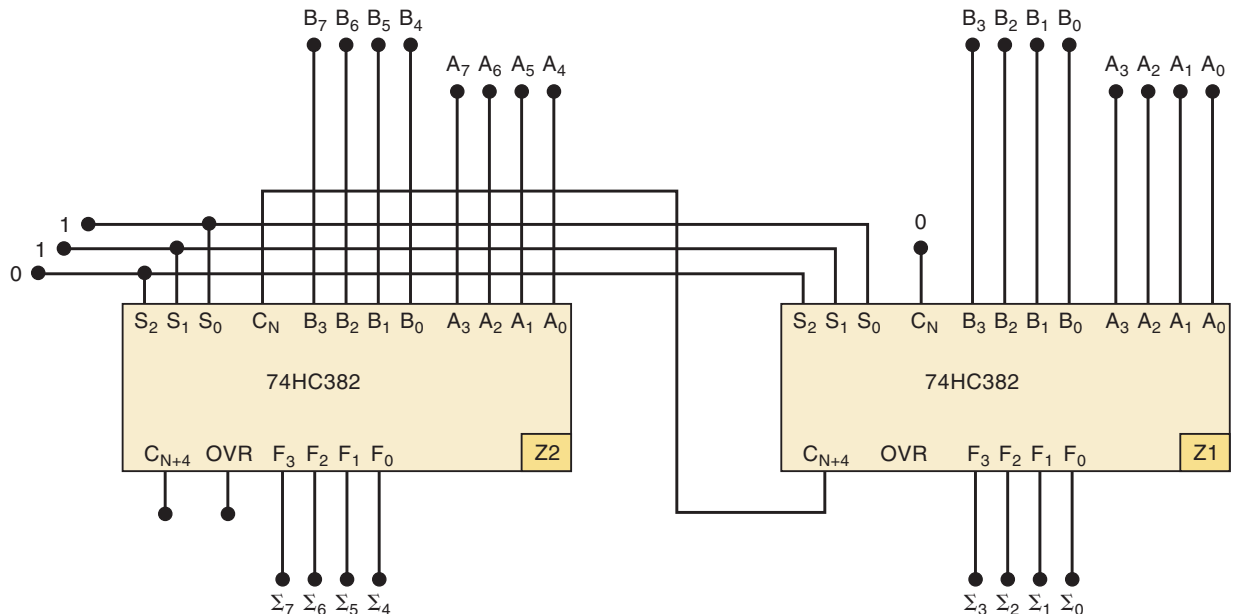


- (b) A select code of 011 will produce the sum of the  $A$  and  $B$  inputs. However, because  $C_N = 1$ , there will be a carry of 1 added into the LSB position. This will produce a result of  $F_3F_2F_1F_0 = 0110$ , which is 1 greater than  $(A + B)$ . The  $C_{N+4}$  and  $OVR$  outputs will both be 0. For the correct sum to appear at  $F$ , the  $C_N$  input must be at 0.

## Expanding the ALU

A single 74LS382 or 74HC382 operates on four-bit numbers. Two or more of these chips can be connected together to operate on larger numbers. Figure 6-16 shows how two four-bit ALUs can be combined to add two eight-bit numbers,  $B_7B_6B_5B_4B_3B_2B_1B_0$  and  $A_7A_6A_5A_4A_3A_2A_1A_0$ , to produce the output sum  $\Sigma_7\Sigma_6\Sigma_5\Sigma_4\Sigma_3\Sigma_2\Sigma_1\Sigma_0$ . Study the circuit diagram and note the following points:

1. Chip Z1 operates on the four lower-order bits of the two input numbers. Chip Z2 operates on the four higher-order bits.
2. The sum appears at the  $F$  outputs of Z1 and Z2. The lower-order bits appear at Z1, and the higher-order bits appear at Z2.
3. The  $C_N$  input of Z1 is the carry into the LSB position. For addition, it is made a 0.
4. The carry output [ $C_{N+4}$ ] of Z1 is connected to the carry input [ $C_N$ ] of Z2.
5. The  $OVR$  output of Z2 is the overflow indicator when signed eight-bit numbers are being used.
6. The corresponding select inputs of the two chips are connected together so that Z1 and Z2 are always performing the same operation. For addition, the select inputs are shown as 011.



Notes: Z1 adds lower-order bits.  
 Z2 adds higher-order bits.  
 $\Sigma_7$ - $\Sigma_0$  = 8-bit sum  
 OVR of Z2 is 8-bit overflow indicator.

**FIGURE 6-16** Two 74HC382 ALU chips connected as an eight-bit adder.

**EXAMPLE 6-12**

How would the arrangement of Figure 6-16 have to be changed in order to perform the subtraction ( $B - A$ )?

**Solution**

The select input code [see the table in Figure 6-15(b)] must be changed to 001, and the  $C_N$  input of Z1 must be made a 1.

**Other ALUs**

The 74LS181/74HC181 is another four-bit ALU. It has four select inputs that can select any of 16 different operations. It also has a mode input bit that can switch between logic operations and arithmetic operations (add and subtract). This ALU has an  $A = B$  output that is used to compare the magnitudes of the  $A$  and  $B$  inputs. When the two input numbers are exactly equal, the  $A = B$  output will be a 1; otherwise, it is a 0.

The 74LS881/74HC881 is similar to the 181 chip, but it has the capability of performing some additional logic operations.

**OUTCOME ASSESSMENT QUESTIONS**

1. Apply the following inputs to the ALU of Figure 6-15, and determine the outputs:  $S_2S_1S_0 = 001$ ,  $A_3A_2A_1A_0 = 1110$ ,  $B_3B_2B_1B_0 = 1001$ ,  $C_N = 1$ .
2. Change the select code to 011 and  $C_N$  to 0, and repeat review question 1.
3. Change the select code to 110, and repeat review question 1.
4. Apply the following inputs to the circuit of Figure 6-16, and determine the outputs:  $B = 01010011$ ,  $A = 00011000$ .
5. Change the select code to 111, and repeat review question 4.
6. How many 74HC382s are needed to add two 32-bit numbers?

**6-17 TROUBLESHOOTING CASE STUDY****OUTCOME**

*Upon completion of this section, you will be able to:*

- Improve troubleshooting skills by studying examples.

A technician is testing the adder/subtractor redrawn in Figure 6-17 and records the following test results for the various operating modes:

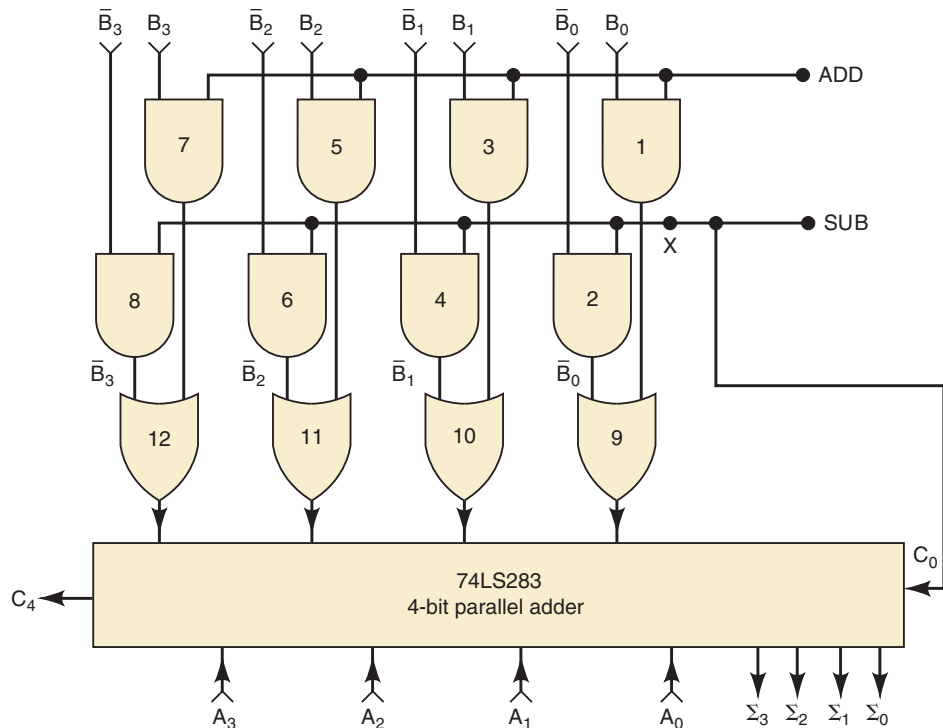
**Mode 1: ADD = 0, SUB = 0.** The sum outputs are always equal to the number in the  $A$  register *plus one*. For example, when  $[A] = 0110$ , the sum is  $[\Sigma] = 0111$ . This is incorrect because the OR outputs and  $C_0$  should all be 0 in this mode to produce  $[\Sigma] = [A]$ .

**Mode 2: ADD = 1, SUB = 0.** The sum is always 1 more than it should be. For example, with  $[A] = 0010$  and  $[B] = 0100$ , the sum output is 0111 instead of 0110.

**Mode 3: ADD = 0, SUB = 1.** The  $\Sigma$  outputs are always equal to  $[A] - [B]$ , as expected.

When she examines these test results, the technician sees that the sum outputs exceed the expected results by 1 for the first two modes of operation.

**FIGURE 6-17** Parallel adder/subtractor circuit.



At first, she suspects a possible fault in one of the LSB inputs to the adder, but she dismisses this because such a fault would also affect the subtraction operation, which is working correctly. Eventually, she realizes that there is another fault that could add an extra 1 to the results for the first two modes without causing an error in the subtraction mode.

Recall that  $C_0$  is made a 1 in the subtraction mode as part of the 2's-complement operation on  $[B]$ . For the other modes,  $C_0$  is to be a 0. The technician checks the connection between the  $SUB$  signal and the  $C_0$  input to the adder and finds that it is open due to a bad solder connection. This open connection explains the observed results because the TTL adder responds as if  $C_0$  were a constant logic 1, causing an extra 1 to be added to the result in modes 1 and 2. The open connection would have no effect on mode 3 because  $C_0$  is supposed to be a 1 anyway.

### EXAMPLE 6-13

Consider again the adder/subtractor circuit. Suppose that there is a break in the connection path between the  $SUB$  input and the AND gates at point  $X$  in Figure 6-17. Describe the effects of this open on the circuit operation for each mode.

#### Solution

First, realize that this fault will produce a logic 1 at the affected input of AND gates 2, 4, 6, and 8, which will permanently enable each of these gates to pass its  $\bar{B}$  input to the following OR gate as shown.

**Mode 1:  $ADD = 0, SUB = 0$ .** The fault will cause the circuit to perform subtraction—almost. The 1's complement of  $[B]$  will reach the OR gate outputs and be applied to the adder along with  $[A]$ . With  $C_0 = 0$ , the 2's complement of  $[B]$  will not be complete; it will be short by 1. Thus, the

adder will produce  $[A] - [B] - 1$ . To illustrate, let's try  $[A] = +6 = 0110$  and  $[B] = +3 = 0011$ . The adder will add as follows:

$$\begin{array}{r} 1\text{'s complement of } [B] = 1100 \\ [A] = 0110 \\ \text{result} = 10010 \\ \quad \quad \quad \uparrow \\ \quad \quad \quad \text{Disregard carry.} \end{array}$$

The result is  $0010 = +2$  instead of  $0011 = +3$ , as it would be for normal subtraction.

**Mode 2: ADD = 1, SUB = 0.** With  $\text{ADD} = 1$ , AND gates 1, 3, 5, and 7 will pass the  $B$  inputs to the following OR gate. Thus, each OR gate will have a  $\bar{B}$  and a  $B$  at its inputs, thereby producing a 1 output. For example, the inputs to OR gate 9 will be  $\bar{B}_0$  coming from AND gate 2 (because of the fault), and  $B_0$  coming from AND gate 1 (because  $\text{ADD} = 1$ ). Thus, OR gate 9 will produce an output of  $\bar{B}_0 + B_0$ , which will always be a logic 1.

The adder will add the 1111 from the OR gates to the  $[A]$  to produce a sum that is 1 less than  $[A]$ . Why? Because  $1111_2 = -1_{10}$ .

**Mode 3: ADD = 0, SUB = 1.** This mode will work correctly because  $\text{SUB} = 1$  is supposed to enable AND gates 2, 4, 6, and 8 anyway.

## 6-18 USING ALTERA LIBRARY FUNCTIONS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Search for standard functional blocks (macrofunctions) in the library of the Quartus Software.
- Set the parameters of any function.
- Generate blocks to meet any circuit's needs.

The adder and ALU ICs that we have looked at in this chapter are just a few of the many MSI chips that have served as the building blocks of digital systems for decades. Whenever a technology serves such a long and useful lifetime, it has a lasting impact on the field and the people who use it. TTL integrated circuits certainly fall into this category and continue in various forms today. Experienced engineers and technicians are familiar with the standard parts. Existing designs can be remanufactured and upgraded using the same basic circuits if they can be implemented in a VLSI PLD. Data sheets for these devices are still readily available, and studying these old TTL parts is one way of learning about digital system fundamentals even though it is not the standard practice today to use these parts in new designs.

For all these reasons, the Altera development software provides macrofunctions in the maxplus2 library for the user. A **macrofunction** is a self-contained description of a logic circuit with all its inputs, outputs, and operational characteristics defined. In other words, Altera engineers have gone to the trouble of writing the code necessary to get a PLD to emulate the operation of many conventional TTL MSI devices. All the designer needs to know is how to hook it into the rest of the system. Let's look at an example of how we can use standard MSI parts from the maxplus2 library to create our designs with schematic capture.

The 74382 arithmetic logic unit (ALU) is a fairly sophisticated IC. The task of describing its operation using HDL code is challenging but certainly

within our reach. Refer again to the examples of this IC and its operation, which were covered in Section 6-16. Specifically, look at Figure 6-16, which shows how to cascade two four-bit ALU chips to make an eight-bit ALU that could serve as the heart of a microcontroller's central processing unit (CPU). Figure 6-18 shows the graphic method of describing the eight-bit circuit using Altera's block diagram file and macrofunction blocks from its library of components. The 74382 symbols are simply chosen from the list in the maxplus2 library and placed on the screen. With a little experience, wiring these chips together is simple and intuitive but there is an even easier way, megafunction LPMs.

### Megafunction LPMs for Arithmetic Circuits

In Chapter 5 we discussed our logic component options when using schematic capture. To create a parallel adder using logic gates from the primitives library would require quite a few gates, so a better choice would be to use either maxplus2 macrofunctions that emulate MSI chips or megafunction LPMs.

Let's compare these two choices for designing an eight-bit parallel adder. The parallel adder will add the eight-bit values  $A[8..1]$  and  $B[8..1]$

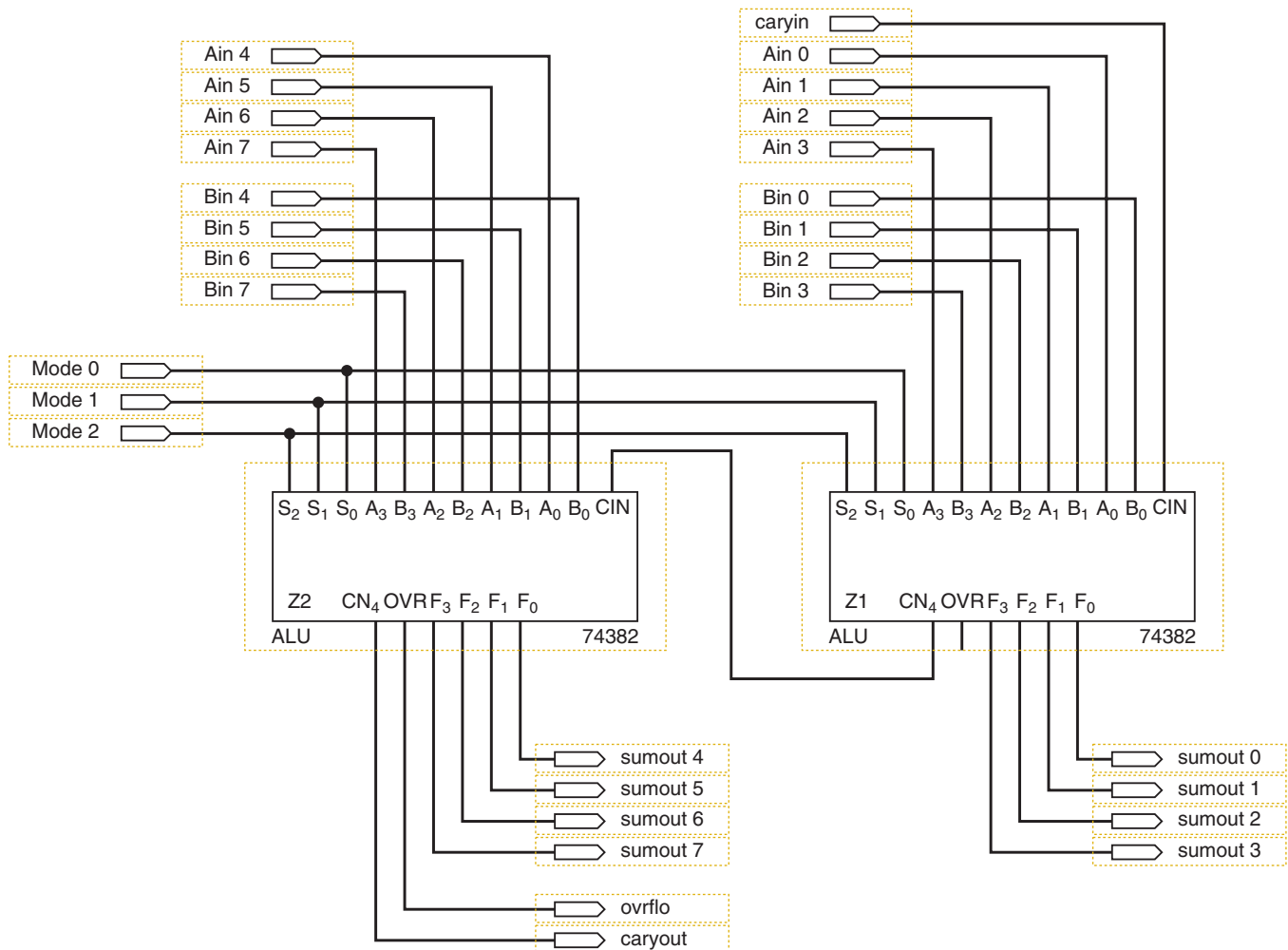
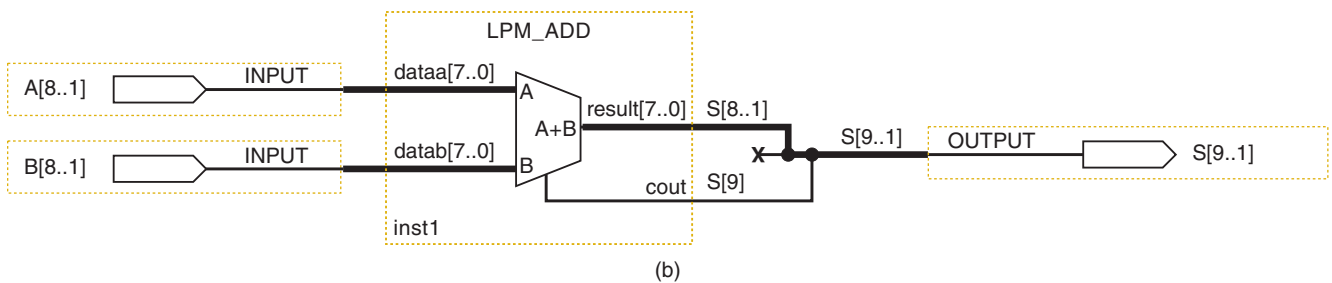
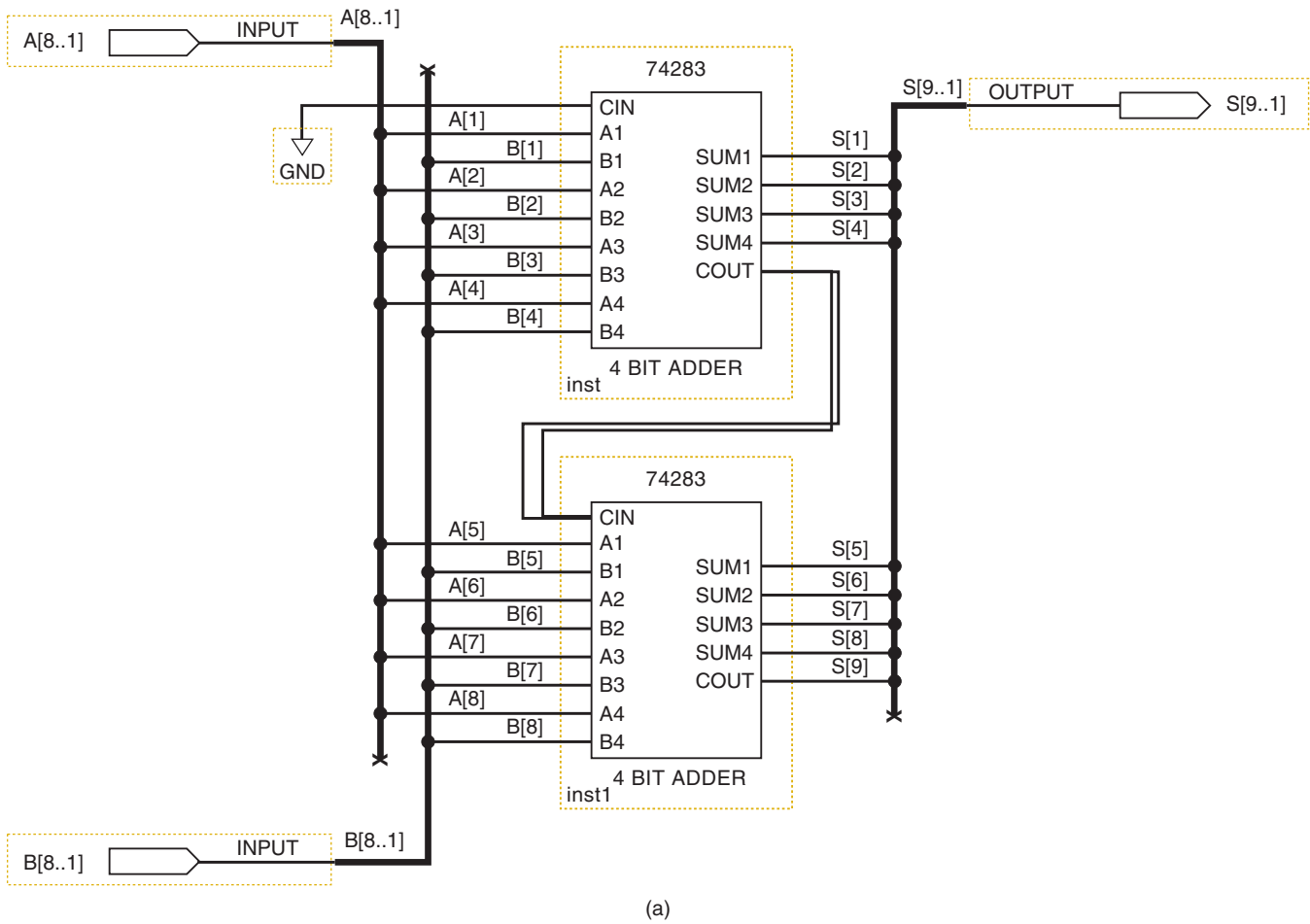


FIGURE 6-18 A Block diagram file of an eight-bit ALU.

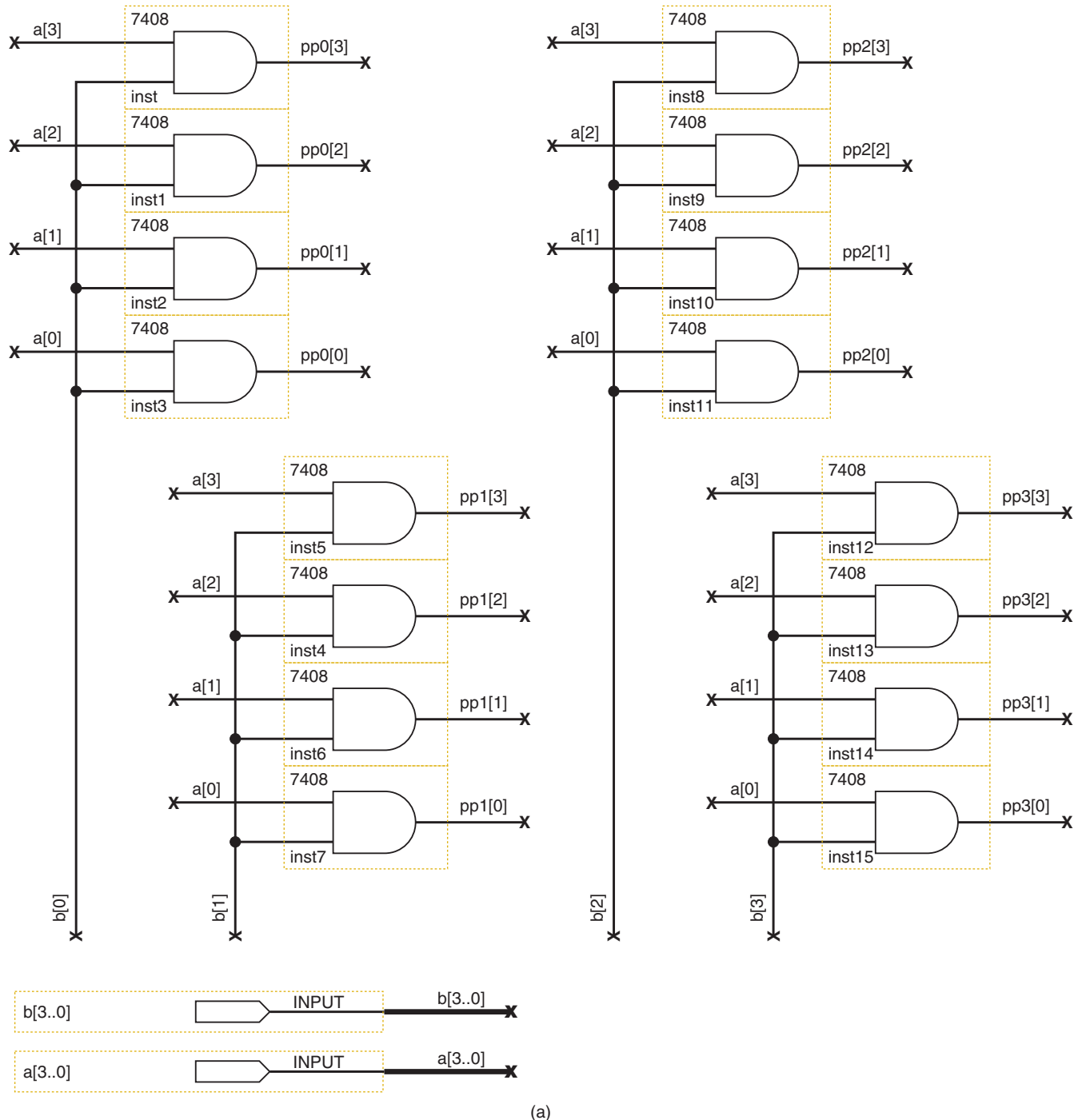
to produce the nine-bit sum  $S[9..1]$ . The 74283 macrofunction is selected for one solution. Since the 74283 is a four-bit parallel adder, we will need to cascade two of these blocks to add the eight-bit operands together (see Figure 6-11(b)). Figure 6-19(a) shows the appropriate wiring connections to construct this circuit. Note the bus splits for each of the two input data buses and the buses merge for the output data bus. Bus splits and merges require labeling of buses and signal lines. The other alternative is for us to use **LPM\_ADD\_SUB** from the megafunction library Arithmetic folder. We will use the MegaWizard Manager to configure this LPM for variable eight-bit data input buses and unsigned addition only. We will not need to include a carry input but we will need a carry output for the ninth bit of the



**FIGURE 6-19** (a) 8-bit parallel adder using 74283 macrofunction; (b) 8-bit parallel adder using LPM\_ADD\_SUB megafunction.

sum. The resultant schematic is shown in Figure 6-19(b). Both circuits will provide the same functionality.

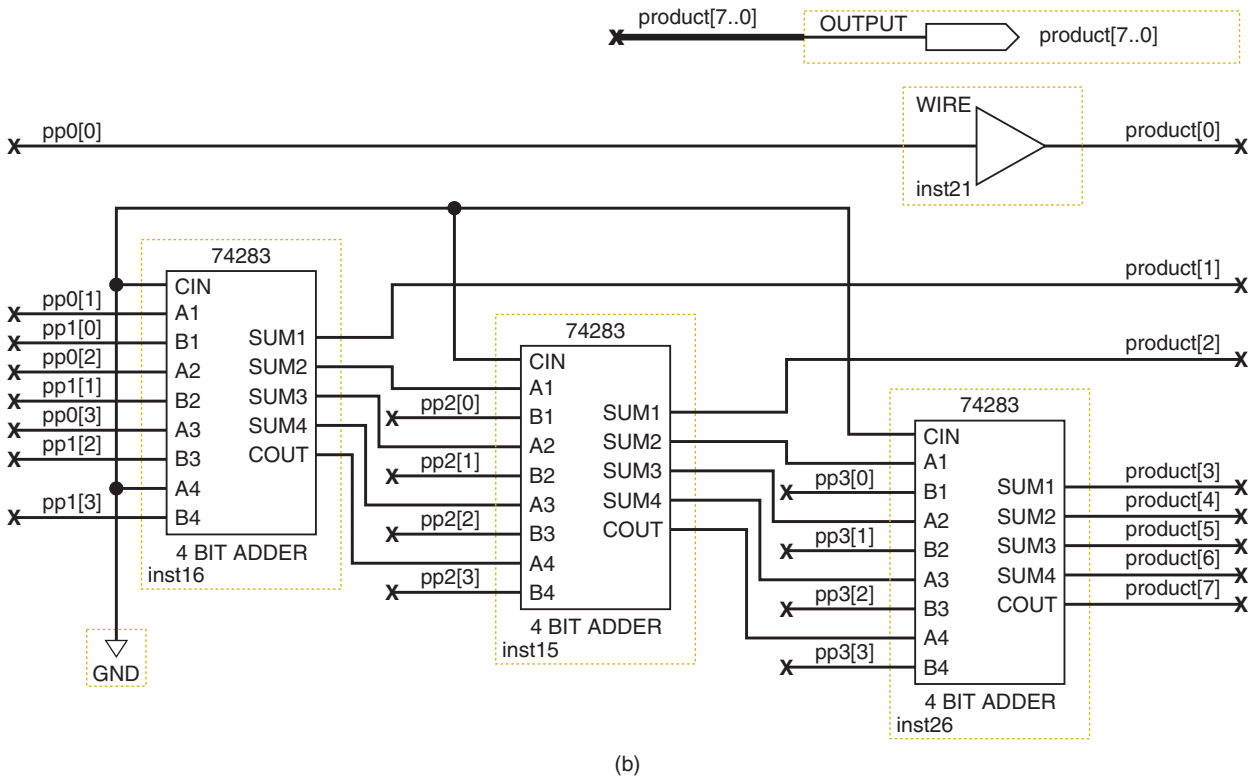
Section 6-5 illustrates the procedure for multiplying binary numbers. Let's compare schematic designs to implement a logic circuit that can multiply two unsigned four-bit numbers. One solution will use maxplus2 macrofunctions and the other will use an LPM megafunction. The macrofunction design is shown in parts (a) and (b) in Figure 6-20. Part (a) has four sets of four AND gates, one set for each of the  $b$  input bits that will produce the



**FIGURE 6-20** (a) Partial product AND gates using 7408.



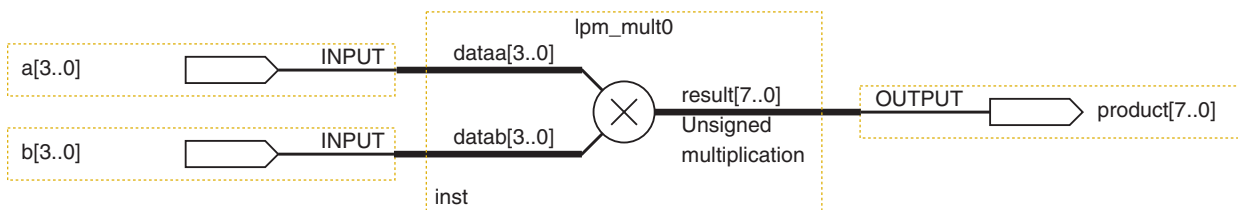
four partial products. The line and bus labels are used to make the correct wiring connections in the Quartus schematic. This technique makes a much neater schematic. The four partial products are then added together, making sure that they are properly aligned, using the 74283 macrofunctions, making sure that they are properly aligned, using the 74283 macrofunctions in part (b). The “wire” symbol allows us to have two different labels for the  $pp0[0]/product[0]$  line in the schematic. The functional simulation results shown in Figure 6-20(c) verify that we have correctly wired the circuit even though it was a bit tricky! Our alternative solution using **LPM\_MULT** is given in Figure 6-20(d). The circuit results are the same, so which solution would you prefer?



(b)

Name	Value at	0 ps	1.0 us	2.0 us	3.0 us	4.0 us	5.0 us	6.0 us	7.0 us	8.0 us	9.0 us	10.0 us	11.0 us	12.0 us	13.0 us	14.0 us	15.0 us
a	U1	1	4	7	10	13	0	3	6	9	12	15	2	5	8	11	14
b	U4	4	9	14	3	8	13	2	7	12	1	6	11	0	5	10	15
product	U4	4	36	98	30	104	0	6	42	108	12	90	22	0	40	110	210

(c)



(d)

**FIGURE 6-20** (Continued) (b) Partial product adder using 74283 macrofunctions; (c) functional simulation results; (d) LPM multiplier solution.

## Using a Parallel Adder to Count

Section 5-18 introduced how a set of flip-flops can be used to create a binary counting function. Since the process of counting up is simply adding one more to the current value stored in a register, it would seem that we should be able to connect a register and an adder together to create a binary counter. That is exactly what we have done in Figure 6-21(a) using a block symbol for a register and an LPM\_ADD\_SUB megafunction for the adder block. The block symbol for reg4bit was created from four D flip-flops, as shown in Chapter 5, Example 5-22. There are many other ways to describe the functionality of this block using AHDL, VHDL, maxplus2 macrofunctions, or megafunctions. The add1 block was created using the LPM\_ADD\_SUB megafunction. The MegaWizard Plug-in Manager was used to set the parameters as follows: four-bit input busses, addition only, set datab as a constant value of 1, unsigned addition, no optional inputs or outputs, and no pipelining. The result is a symbol that adds the constant value of 1 (input B) to the four-bit value from reg4bit (adder input A). The functional simulation results given in Figure 6-21(b) is a recycling count sequence of 0000 to 1111, for a total of 16 states, making this design a MOD-16 up counter.

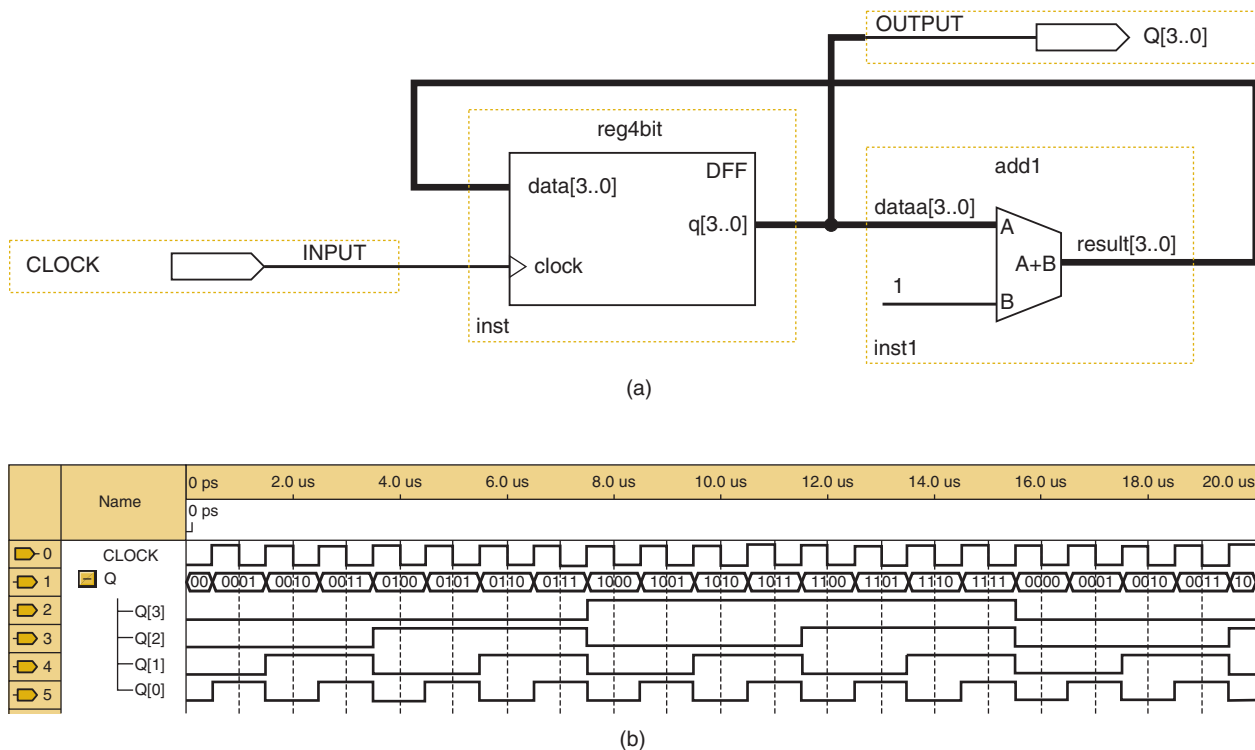


FIGURE 6-21 Binary up counter: (a) block diagram using LPMs; (b) simulation results.

### EXAMPLE 6-14

Create a mod-8 down counter using a block symbol for the register and an LPM block for the subtractor.

#### Solution

An LPM design and functional simulation results, indicating that this design is a recycling, mod-8 down counter, are shown in Figure 6-22.

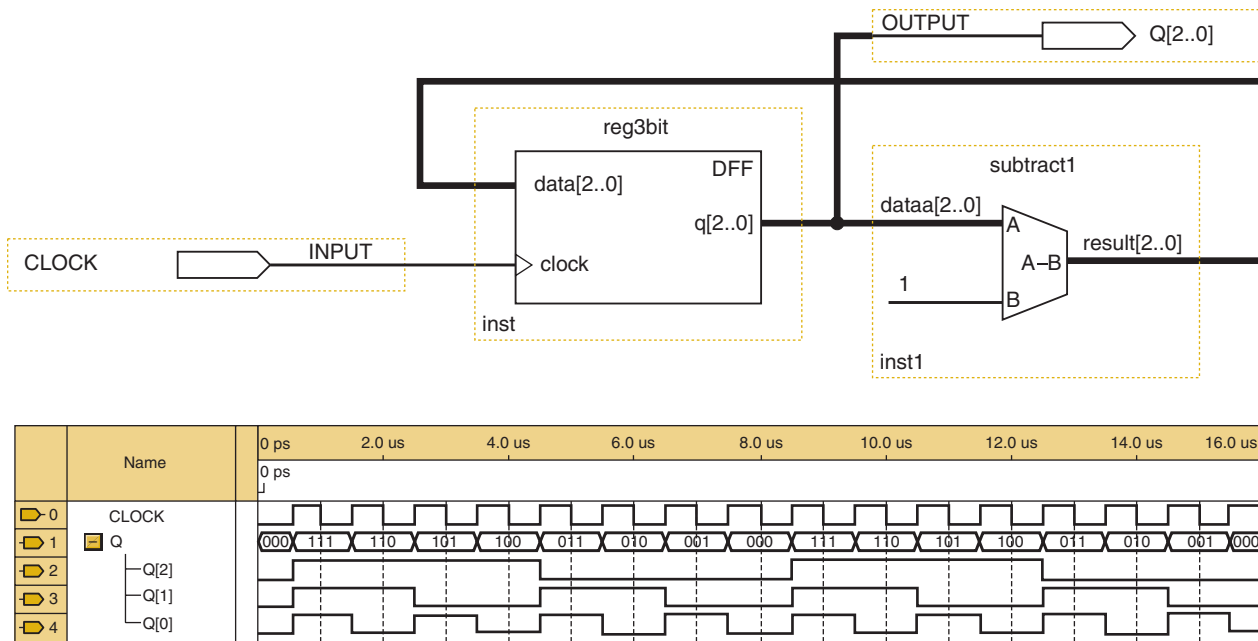


FIGURE 6-22 Mod-8 down counter block diagram and functional simulation results.

### OUTCOME ASSESSMENT QUESTIONS

1. Where can you find information about using a 74283 parallel adder macrofunction in your schematic design?
2. What is a macrofunction?

## 6-19 LOGICAL OPERATIONS ON BIT ARRAYS WITH HDLs

### OUTCOMES

Upon completion of this section, you will be able to:

- Combine bit arrays using logical operators using proper syntax in a given HDL.
- Predict result of logical combinations of bit arrays.

In Section 6-16, we examined an ALU chip that is capable of performing arithmetic and logic operations on binary input data. Now we will look at the HDL code that will perform such logic operations with bit arrays. In this section, we will expand our understanding of the HDL techniques in two main areas: specifying groups of bits in an array and using logical operations to combine arrays of bits using Boolean expressions.

In Section 6-12, we discussed register notation, which makes it easy to describe the contents of registers and signals consisting of multiple bits. The HDLs use arrays of bits in a similar notation to describe signals, as we discussed in Chapter 4. For example, in AHDL, the four-bit signal named *d* is defined as:

```
VARIABLE d[3..0] :NODE.
```

In VHDL, the same data format is expressed as:

```
SIGNAL d :BIT_VECTOR (3 DOWNTO 0).
```

Each bit in these data types is designated by an element number. In this example of a bit array named  $d$ , the bits can be referred to as  $d_3, d_2, d_1, d_0$ . Bits can also be grouped into sets. For example, if we want to refer to the three most significant bits of  $d$  as a set, we can use the expression  $d[3..1]$  in AHDL and the expression  $d(3 \text{ DOWNTO } 1)$  in VHDL. Once a value is assigned to the array and the desired set of bits is identified, logical operations can be performed on the entire set of bits. As long as the sets are the same size (the same number of bits), two sets can be combined in a logical expression, just like you would combine single variables in a Boolean equation. Each of the corresponding pairs of bits in the two sets is combined as stated in the logic equation. This allows one equation to describe the logical operation performed on each bit in a set.

**EXAMPLE 6-15**

Assume  $D_3, D_2, D_1, D_0$  has the value 1011 and  $G_3, G_2, G_1, G_0$  has the value 1100. Let's define  $D = [D_3, D_2, D_1, D_0]$  and  $G = [G_3, G_2, G_1, G_0]$ . Let's also define  $Y = [Y_3, Y_2, Y_1, Y_0]$  where  $Y$  is related to  $D$  and  $G$  as follows:

$$Y = D \cdot G;$$

What is the value of  $Y$  after this operation?

**Solution**

$D_3, D_2, D_1, D_0$	1 0 1 1	AND each bit position together
↑ ↓ ↑ ↓	↑ ↓ ↑ ↓	
$G_3, G_2, G_1, G_0$	1 1 0 0	
$Y_3, Y_2, Y_1, Y_0$	1 0 0 0	

Thus,  $Y$  is a set of four bits with value 1000.

**EXAMPLE 6-16**

For the register values described in Example 6-15, declare each  $d$ ,  $g$ , and  $y$ . Then write an expression using your favorite HDL that performs the ANDing operation on all bits.

**Solution**

```
SUBDESIGN bitwise_and
(
  d[3..0], g[3..0]      :INPUT;
  y[3..0]               :OUTPUT;)
BEGIN
  y[] = d[] & g[];
END;
```

```
ENTITY bitwise_and IS
PORT(d, g           :IN BIT_VECTOR (3 DOWNTO 0);
     y              :OUT BIT_VECTOR (3 DOWNTO 0));
END bitwise_and;
ARCHITECTURE a OF bitwise_and IS
BEGIN
  y <= d AND g;
END a;
```

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. If  $[A] = 1001$  and  $[B] = 0011$ , what is the value of (a)  $[A] \cdot [B]$ ? (b)  $[A] + [B]$ ? (Note that  $\cdot$  means AND;  $+$  means OR.)
2. If  $A[7..0] = 1010\ 1100$ , what is the value of (a)  $A[7..4]$ ? (b)  $A[5..2]$ ?
3. In AHDL, the following object is declared: `toggles[7..0]:INPUT`. Give an expression for the least significant four bits using AHDL syntax.
4. In VHDL, the following object is declared: `toggles:IN BIT_VECTOR (7 DOWNTO 0)`. Give an expression for the least significant four bits using VHDL syntax.
5. What would be the result of ORing the two registers of Example 6-15?
6. Write an HDL statement that would OR the two objects  $d$  and  $g$  together. Use your favorite HDL.
7. Write an HDL statement that would XOR the two most significant bits of  $d$  with the two least significant bits of  $g$  and put the result in the middle two bits of  $x$ .

## 6-20 HDL ADDERS

### OUTCOME

*Upon completion of this section, you will be able to:*

- Use AHDL or VHDL to describe circuits that perform arithmetic operations.

In this section, we will see how to create an 8-bit parallel adder circuit using HDL languages. The parallel adder will add the 8-bit values  $A[7..0]$  and  $B[7..0]$  to produce the 9-bit sum  $S[8..0]$ . The 9-bit sum will include the carry out as the ninth bit. One option would be for us to make the HDL design file for a 4-bit parallel adder, instruct Quartus to create the corresponding block symbol, and then use the block editor to draw a schematic that looks very similar to Figure 6-19(a) (although we would probably use array inputs and outputs in the block symbol for the HDL). However, it would be much easier for us to simply increase the size of each operand and output variable in the HDL design file and be done with it.

### AHDL EIGHT-BIT ADDER

Note that subdesign lines 3 and 4 of Figure 6-23 specify 8-bit operands and line 5 creates a 9-bit output array. In lines 8 and 9 of the AHDL code, we have declared two variable arrays of bits named  $aa$  and  $bb$ . The 9-bit arrays are inside this subdesign block and are described as “buried” nodes since they are not visible outside the subdesign as either input or output ports. The reason for defining the 9-bit arrays is to match the number of bits for the sum because we wish to also output the sum’s ninth bit (the carry out bit). Lines 11 and 12 will fill in the two buried arrays with a leading zero followed by the appropriate 8-bit input value. The two assignment statements will each concatenate a zero and the data input value together. The 9-bit output sum is produced with line 13 by adding together the two internal variables  $aa$  and  $bb$ . The compiler will be happy because line 13 has 9-bit arrays on both sides of the equals sign.

```

1  SUBDESIGN fig6_23
2  (
3  a[7..0]      :INPUT;           -- 8-bit augend
4  b[7..0]      :INPUT;           -- 8-bit addend
5  s[8..0]      :OUTPUT;         -- 9-bit sum
6  )
7  VARIABLE
8  aa[8..0]     :NODE;           -- expanded augend
9  bb[8..0]     :NODE;           -- expanded addend
10 BEGIN
11 aa[8..0] = (GND,a[7..0]);      -- concatenate leading zero
12 bb[8..0] = (GND,b[7..0]);      -- to both operands
13 s[8..0] = aa[8..0] + bb[8..0]; -- add expanded operands
14 END;
```

**FIGURE 6-23** 8-bit AHDL adder.

## VHDL EIGHT-BIT ADDER

Lines 3 and 4 in the entity declaration of Figure 6-24 will set up the 8-bit input signals and line 5 will create a 9-bit output signal. Notice that the input and output ports in the VHDL code (lines 3–5) are declared to be an integer data type. A `BIT_VECTOR` data type in VHDL is assumed to only be an array of bits with no numerical value associated with the array. An `INTEGER` data type, on the other hand, will represent a numerical value so that we can perform an arithmetic operation on it. An 8-bit binary integer can have a range of 0 to 255 while a 9-bit integer will have a range of values from 0 to 511. The signal assignment statement in line 12 will produce the sum of the two input operands *a* and *b*, which is assigned to the output port *s*.

```

1  ENTITY fig6_24 IS
2  PORT (
3      a      :IN INTEGER RANGE 0 TO 255;    -- 8-bit augend
4      b      :IN INTEGER RANGE 0 TO 255;    -- 8-bit addend
5      s      :OUT INTEGER RANGE 0 TO 511    -- 9-bit sum
6  );
7  END fig6_24;
8
9  ARCHITECTURE parallel OF fig6_24 IS
10
11 BEGIN
12     s <= a + b;                            -- add operands
13 END parallel;
```

**FIGURE 6-24** 8-bit VHDL adder.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Modify the AHDL code in Figure 6-23 to create a 4-bit parallel adder.
2. Modify the VHDL code in Figure 6-24 to create a 4-bit parallel adder.

## 6-21 PARAMETERIZING THE BIT CAPACITY OF A CIRCUIT

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe constants
- Use constants in HDL code
- Predict compiler interpretation of expressions that contain constants.

One way we have learned to expand the capacity of a circuit is to cascade stages, like we did with the 74283 parallel adder and the 74382 ALU in Section 6-18. This can be done using the Altera block design file approach (like Figure 6-18) or the text-based HDL approach. With either of these methods, we need to specify all the inputs, outputs, and interconnections between blocks. In the last section, we saw that it is very easy to simply modify the size of each operand variable when declaring the inputs and outputs in order to change the number of parallel adder bits required for an application. Regardless of whether we need a 4-bit, 8-bit, 12-bit, or any other size adder, the code that defines the circuit logic will be practically identical. Only the sizes of the inputs and outputs will change. This is just a glimpse of some of the efficiency improvements that HDL offers. The designer would, however, need to examine the code carefully and make all of the appropriate changes for the desired application size.

An important principle in software engineering is symbolic representation of the **constants** that are used throughout the code. Constants are simply fixed numbers represented by a name (symbol). If we can define a symbol (i.e., make up a name) at the top of the source code that is assigned the value for the total number of bits and then use this symbol (name) throughout the code, it is much easier to modify the circuit. Only one line of code needs to be changed to expand the capacity of the circuit. The examples that follow add this feature to the HDL code for an adder/subtractor circuit. A single input bit named *add\_sub* will control the adder/subtractor's function. The circuit will add the two operands when *add\_sub* = 0 or subtract *b* from *a* when *add\_sub* = 1.

### AHDL ADDER/SUBTRACTOR

In AHDL, using constants is very simple, as shown on line 1 of Figure 6-25. The keyword **CONSTANT** is followed by the symbolic name and the value it is to be assigned. We can allow the compiler to do some simple math calculations to establish a value for one constant based on another. We can also use this feature as we refer to the constant in the code, as shown on lines 12 through 14 and 23. For example, we can refer to *c[7]* as *c[n]* and *c[8]* as *c[n + 1]*. The size of this adder/subtractor can be changed by simply

```

1  CONSTANT n = 6;          -- user gives number of input bits
2
3  SUBDESIGN figure6_25    -- addsub
4  (
5      a[n-1..0]          :INPUT;          -- n-bit augend
6      b[n-1..0]          :INPUT;          -- n-bit addend
7      add_sub            :INPUT;          -- add or subtract
8      result[n-1..0]    :OUTPUT;         -- n-bit answer
9      carryborrow        :OUTPUT;         -- carry out
10 )
11 VARIABLE
12     aa[n..0]           :NODE;           -- expanded augend
13     bb[n..0]           :NODE;           -- expanded addend
14     rr[n..0]           :NODE;           -- expanded result
15 BEGIN
16     aa[] = (GND,a[]);    -- leading zero
17     bb[] = (GND,b[]);    -- leading zero
18     IF add_sub == GND THEN -- add if add_sub = 0
19         rr[] = aa[] + bb[]; -- calculate sum
20     ELSE rr[] = aa[] - bb[]; -- calculate difference
21     END IF;
22     result[] = rr[n-1..0]; -- get n-bit answer
23     carryborrow = rr[n]; -- get carry or borrow out
24 END;
```

**FIGURE 6-25** An  $n$ -bit adder/subtractor description in AHDL.

changing the value of the declared constant  $n$  to the desired number of bits and then recompiling.

A set of three new buried bit arrays is defined in lines 12 through 14. Each of these variables is one bit wider than the number of bits given on line 1 for the width of the parallel adder. This extra bit has been added to capture the carry or borrow output produced when the two operands are either added or subtracted. The two expanded operands (lines 12 and 13) are created because the AHDL compiler requires the same number of bits for the variables on each side of the equals sign when computing the sum (line 19) or difference (line 20). A leading zero is concatenated to each of the data inputs and assigned to the expanded variables on lines 16 and 17. The output *result* will be assigned the lower  $n$  bits from the calculation in line 22 while the *carryborrow* is assigned the bit value for the extra bit. The *carryborrow* output will be high if the sum calculation ( $a + b$ ) produces a final carry out or if the difference calculation ( $a - b$ ) subtracted a larger  $b$  value from a smaller  $a$  value and, therefore, needed to borrow from a nonexistent source. If the operands are unsigned binary values and the *carryborrow* output is high, then the  $n$ -bit result will be incorrect. This condition would indicate that more than  $n$  bits are needed for the correct sum or that a larger number was subtracted from a smaller number. As we saw in Sections 6-3 and 6-4, when using signed numbers, the *carryborrow* output is disregarded and the  $n$ -bit answer will also be a signed number. Furthermore, we will need to check for overflow for a signed number operation.



## VHDL ADDER/SUBTRACTOR

In VHDL, using constants is a little bit more involved. Constants must be included in a **PACKAGE**, as shown in Figure 6-26, lines 1–6. Packages are also used to contain component definitions and other information that must be available to all entities in the design file. Notice on line 8 that the keyword **USE** tells the compiler to use the definitions in this package throughout this design file. Inside the package, the keyword **CONSTANT** is followed by the symbolic name, its type, and the value it is to be assigned using the **:=** operator. Notice on line 3 that we can allow the compiler to do some simple math calculations to establish a value of one constant based on another. We can also use this feature as we refer to the constant in the code, as shown on lines 12, 13, 15, 23 and 32. The size of this adder/subtractor can be changed by simply changing the value of the declared constant *n* to the desired number of bits and then recompiling.

```

1  PACKAGE const IS
2      CONSTANT n      :INTEGER      := 6;      -- user gives number of input bits
3      CONSTANT m      :INTEGER      := 2**n;   -- compute combinations = 2 to the n
4      CONSTANT p      :INTEGER      := n+1;   -- add extra bit
5      CONSTANT q      :INTEGER      := 2**p;   -- compute combinations = 2 to the p
6  END const;
7
8  USE work.const.all;
9
10 ENTITY fig6_26 IS
11 PORT (
12     a          :IN INTEGER RANGE 0 TO m-1;   -- augend
13     b          :IN INTEGER RANGE 0 TO m-1;   -- addend;
14     add_sub    :IN BIT;                      -- add or subtract
15     result     :OUT INTEGER RANGE 0 TO m-1;  -- answer
16     carryborrow :OUT BIT                    -- carry borrow out
17 );
18 END fig6_26;
19
20 ARCHITECTURE parameterized OF fig6_26 IS
21 BEGIN
22 PROCESS (a, b, add_sub)
23     VARIABLE rr :INTEGER RANGE 0 TO q-1;     -- result with carry borrow
24 BEGIN
25     IF add_sub = '0' THEN                    -- add if add_sub = 0
26         rr := a + b;                         -- calculate sum
27     ELSE rr := a - b;                        -- calculate difference
28     END IF;
29     IF rr < m THEN                            -- test if extra bit was needed
30         result <= rr;                        -- get answer
31         carryborrow <= "0";                 -- get carry borrow out
32     ELSE result <= rr-m;                     -- get answer
33         carryborrow <= "1";                 -- get carry borrow out
34     END IF;
35 END PROCESS;
36 END parameterized;

```

**FIGURE 6-26** An *n*-bit adder/subtractor description in VHDL.

Since the local variable *rr*, defined in line 23, covers a number range that is twice the size (i.e., one power of 2 greater) of the input operands, it will be one bit wider than the number of parallel adder bits given on line 2. This extra bit has been added to capture the carry or borrow output produced when the two operands are either added or subtracted. This local variable will get the answer for the selected addition or subtraction calculation in lines 25 through 28. If this answer is in the same range as the input operands, the output *result* will be assigned the calculation result in line 30 while the *carryborrow* is assigned a bit value of zero (line 31). If the calculation produces a value that is greater than the range for the operands (i.e., IF statement on line 29 is false), then the weight of the next higher bit position will be subtracted from the value of *rr* (line 32) and the *carryborrow* output will be assigned a logic one (line 33). The *carryborrow* output will be high if the sum calculation ( $a + b$ ) produces a final carry out or if the difference calculation ( $a - b$ ) subtracted a larger *b* value from a smaller *a* value and, therefore, needed to borrow from a nonexistent source. If the operands are unsigned binary values and the *carryborrow* output is high, then the *n*-bit result will be incorrect. This condition would indicate that more than *n* bits are needed for the correct sum or that a larger number was subtracted from a smaller number. As we saw in Sections 6-3 and 6-4, when using signed numbers, the *carryborrow* output is disregarded and the *n*-bit answer will also be a signed number. Furthermore, we will need to check for overflow for a signed number operation.

### OUTCOME ASSESSMENT QUESTIONS

1. What keyword is used to assign a symbolic name to a fixed number?
2. In AHDL, where are constants defined? Where are they defined in VHDL?
3. Why are constants useful?
4. If the constant `max_val` has a value of 127, how will a compiler interpret the expression `max_val - 5`?

## SUMMARY

1. To represent signed numbers in binary, a sign bit is attached as the MSB. A + sign is a 0, and a - sign is a 1.
2. The 2's complement of a binary number is obtained by complementing each bit and then adding 1 to the result.
3. In the 2's-complement method of representing signed binary numbers, positive numbers are represented by a sign bit of 0 followed by the magnitude in its true binary form. Negative numbers are represented by a sign bit of 1 followed by the magnitude in 2's-complement form.
4. A signed binary number is negated (changed to a number of equal value but opposite sign) by taking the 2's complement of the number, including the sign bit.
5. Subtraction can be performed on signed binary numbers by negating (2's complementing) the subtrahend and adding it to the minuend.
6. In BCD addition, a special correction step is needed whenever the sum of a digit position exceeds 9 (1001).



- |                                   |                                 |
|-----------------------------------|---------------------------------|
| (e) $10011011 + 10011101$         | (k)* $101010 - 100101$          |
| (f) $1010.01 + 10.111$            | (l)* $1111.010 - 1000.001$      |
| (g) $10001111 + 01010001$         | (m) $10011 - 00110$             |
| (h) $11001100 + 00110111$         | (n) $11100010 - 01010001$       |
| (i) $110010100011 + 011101111001$ | (o) $100010.1001 - 001111.0010$ |
| (j)* $1010 - 0111$                | (p) $1011000110 - 1001110100$   |

**SECTION 6-2**

- B** 6-2. Represent each of the following signed decimal numbers in the 2's-complement system. Use a total of eight bits, including the sign bit.
- |           |           |          |          |
|-----------|-----------|----------|----------|
| (a)* +32  | (e)* +127 | (i) -1   | (m) +84  |
| (b)* -14  | (f)* -127 | (j) -128 | (n) +3   |
| (c)* +63  | (g)* +89  | (k) +169 | (o) -3   |
| (d)* -104 | (h)* -55  | (l) 0    | (p) -190 |
- B** 6-3. Each of the following numbers represents a signed decimal number in the 2's-complement system. Determine the decimal value in each case. (*Hint*: Use negation to convert negative numbers to positive.)
- |               |              |
|---------------|--------------|
| (a)* 01101    | (f) 10000000 |
| (b)* 11101    | (g) 11111111 |
| (c)* 01111011 | (h) 10000001 |
| (d)* 10011001 | (i) 01100011 |
| (e)* 01111111 | (j) 11011001 |
- 6-4. (a) What range of signed decimal values can be represented using 12 bits, including the sign bit?  
 (b) How many bits would be required to represent decimal numbers from -32,768 to +32,767?
- 6-5\* List, in order, all of the signed numbers that can be represented in five bits using the 2's-complement system.
- 6-6. Represent each of the following decimal values as an eight-bit signed binary value. Then negate each one.  
 (a)\* +73    (b)\* -12    (c) +15    (d) -1    (e) -128    (f) +127
- 6-7. (a)\* What is the range of unsigned decimal values that can be represented in 10 bits? What is the range of signed decimal values using the same number of bits?  
 (b) Repeat both problems using eight bits.

**SECTIONS 6-3 AND 6-4**

- 6-8. The reason why the sign-magnitude method for representing signed numbers is not used in most computers can readily be illustrated by performing the following.
- Represent +12 in eight bits using the sign-magnitude form.
  - Represent -12 in eight bits using the sign-magnitude form.
  - Add the two binary numbers and note that the sum does not look anything like zero.

- N** 6-9. Perform the following operations in the 2's-complement system. Use eight bits (including the sign bit) for each number. Check your results by converting the binary result back to decimal.
- |                             |                            |
|-----------------------------|----------------------------|
| (a)* Add +9 to +6.          | (g) Subtract +47 from +47. |
| (b)* Add +14 to -17.        | (h) Subtract -36 from -15. |
| (c)* Add +19 to -24.        | (i) Add +17 to -17.        |
| (d)* Add -48 to -80.        | (j) Subtract -17 from -17. |
| (e)* Subtract +16 from +17. | (k) Add +68 to +45         |
| (f) Subtract +21 from -13.  | (l) Subtract -50 from +77  |
- 6-10. Repeat Problem 6-9 for the following cases, and show that overflow occurs in each case.
- |                            |                            |
|----------------------------|----------------------------|
| (a) Add +37 to +95.        | (c) Add -37 to -95.        |
| (b) Subtract +37 from -95. | (d) Subtract -37 from +95. |

### SECTIONS 6-5 AND 6-6

- B, N** 6-11. Multiply the following pairs of binary numbers, and check your results by doing the multiplication in decimal.
- |                              |                          |
|------------------------------|--------------------------|
| (a)* $111 \times 101$        | (d) $.1101 \times .1011$ |
| (b)* $1011 \times 1011$      | (e) $1111 \times 1011$   |
| (c) $101.101 \times 110.010$ | (f) $10110 \times 111$   |
- B, N** 6-12. Perform the following divisions. Check your results by doing the division in decimal.
- |                         |                           |
|-------------------------|---------------------------|
| (a)* $1100 \div 100$    | (d) $10110.1101 \div 1.1$ |
| (b)* $111111 \div 1001$ | (e) $1100011 \div 1001$   |
| (c) $10111 \div 100$    | (f) $100111011 \div 1111$ |

### SECTIONS 6-7 AND 6-8

- B, N** 6-13. Add the following decimal numbers after converting each to its BCD code.
- |                  |                 |
|------------------|-----------------|
| (a)* $74 + 23$   | (e) $998 + 003$ |
| (b)* $58 + 37$   | (f) $623 + 599$ |
| (c)* $147 + 380$ | (g) $555 + 274$ |
| (d) $385 + 118$  | (h) $487 + 116$ |
- B, N** 6-14. Find the sum of each of the following pairs of hex numbers.
- |                    |                   |
|--------------------|-------------------|
| (a)* $3E91 + 2F93$ | (e) $FFF + 0FF$   |
| (b)* $91B + 6F2$   | (f) $D191 + AAAB$ |
| (c)* $ABC + DEF$   | (g) $5C74 + 22BA$ |
| (d) $2FFE + 0002$  | (h) $39F0 + 411F$ |
- B, N** 6-15. Perform the following subtractions on the pairs of hex numbers.
- |                    |                   |
|--------------------|-------------------|
| (a)* $3E91 - 2F93$ | (e) $F000 - EFFF$ |
| (b)* $91B - 6F2$   | (f) $2F00 - 4000$ |
| (c)* $0300 - 005A$ | (g) $9AE5 - C01D$ |
| (d) $0200 - 0003$  | (h) $4321 - F165$ |
- 6-16. The owner's manual for a small microcomputer states that the computer has usable memory locations at the following hex addresses: 0200

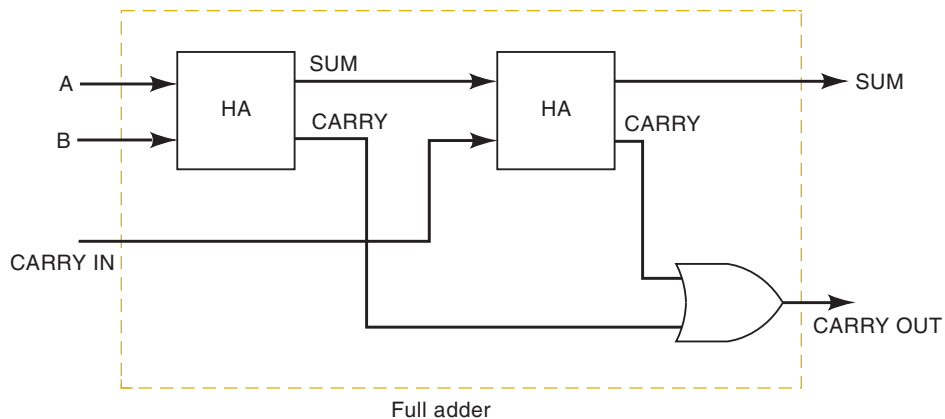
through 03FF, and 4000 through 7FD0. What is the total number of available memory locations?

- 6-17. (a)\* A certain eight-bit memory location holds the hex data 77. If this represents an *unsigned* number, what is its decimal value?  
 (b)\* If this represents a *signed* number, what is its decimal value?  
 (c) Repeat (a) and (b) if the data value is E5.

### SECTION 6-11

- 6-18. Convert the FA circuit of Figure 6-8 to all NAND gates.  
 6-19.\* Write the function table for a half adder (inputs  $A$  and  $B$ ; outputs SUM and CARRY). From the function table, design a logic circuit that will act as a half adder.  
 6-20. A full adder can be implemented in many different ways. Figure 6-27 shows how one may be constructed from two half adders. Construct a function table for this arrangement, and verify that it operates as a FA.

**FIGURE 6-27** Problem 6-20.



### SECTION 6-12

- 6-21.\* Refer to Figure 6-10. Determine the contents of the  $A$  register after the following sequence of operations:  $[A] = 0000$ ,  $[0100] \rightarrow [B]$ ,  $[S] \rightarrow [A]$ ,  $[1011] \rightarrow [B]$ ,  $[S] \rightarrow [A]$ .
- 6-22. Refer to Figure 6-10. Assume that each FF has  $t_{PLH} = t_{PHL} = 30$  ns and a setup time of 10 ns, and that each FA has a propagation delay of 40 ns. What is the minimum time allowed between the PGT of the LOAD pulse and the PGT of the TRANSFER pulse for proper operation?
- D** 6-23. In the adder and subtractor circuits discussed in this chapter, we gave no consideration to the possibility of *overflow*. Overflow occurs when the two numbers being added or subtracted produce a result that contains more bits than the capacity of the accumulator. For example, using four-bit registers, including a sign bit, numbers ranging from +7 to -8 (in 2's complement) can be stored. Therefore, if the result of an addition or subtraction exceeds +7 or -8, we would say that an overflow has occurred. When an overflow occurs, the results are useless because they cannot be stored correctly in the accumulator register. To illustrate, add +5 (0101) and +4 (0100), which results in 1001. This 1001 would be interpreted incorrectly as a negative number because there is a 1 in the sign-bit position.

In computers and calculators, there are usually circuits that are used to detect an overflow condition. There are several ways to do this. One method that can be used for the adder that operates in the 2's-complement system works as follows:

1. Examine the sign bits of the two numbers being added.
2. Examine the sign bit of the result.
3. Overflow occurs whenever the numbers being added are *both positive* and the sign bit of the result is 1 *or* when the numbers are *both negative* and the sign bit of the result is 0.

This method can be verified by trying several examples. Try the following cases: (1)  $5 + 4$ ; (2)  $-4 + (-6)$ ; (3)  $3 + 2$ . Cases 1 and 2 will produce an overflow, and case 3 will not. Thus, by examining the sign bits, one can design a logic circuit that will produce a 1 output whenever the overflow condition occurs. Design this overflow circuit for the adder of Figure 6-10.

- C, D** 6-24. Add the necessary logic circuitry to Figure 6-10 to accommodate the transfer of data from memory into the *A* register. The data values from memory are to enter the *A* register through its *D* inputs on the PGT of the *first* TRANSFER pulse; the data from the sum outputs of the FAs will be loaded into *A* on the PGT of the *second* TRANSFER. In other words, a LOAD pulse followed by two TRANSFER pulses is required to perform the complete sequence of loading the *B* register from memory, loading the *A* register from memory, and then transferring their sum into the *A* register. (*Hint*: Use a flip-flop *X* to control which source of data gets loaded into the *D* inputs of the accumulator.)

### SECTION 6-13

- C, D** 6-25.\* Design a look-ahead carry circuit for the adder of Figure 6-10 that generates the carry  $C_3$  to be fed to the FA of the MSB position based on the values of  $A_0, B_0, C_0, A_1, B_1, A_2,$  and  $B_2$ . In other words, derive an expression for  $C_3$  in terms of  $A_0, B_0, C_0, A_1, B_1, A_2,$  and  $B_2$ . (*Hint*: Begin by writing the expression for  $C_1$  in terms of  $A_0, B_0,$  and  $C_0$ . Then write the expression for  $C_2$  in terms of  $A_1, B_1,$  and  $C_1$ . Substitute the expression for  $C_1$  into the expression for  $C_2$ . Then write the expression for  $C_3$  in terms of  $A_2, B_2,$  and  $C_2$ . Substitute the expression for  $C_2$  into the expression for  $C_3$ . Simplify the final expression for  $C_3$  and put it in sum-of-products form. Implement the circuit.)

### SECTION 6-14

- N** 6-26. Show the logic levels at each input and output of Figure 6-11(b) when  $EC_{16}$  is added to  $43_{16}$ .

### SECTION 6-15

- 6-27. For the circuit of Figure 6-14, determine the sum outputs for the following cases.
- (a)\* *A* register = 0101 (+5), *B* register = 1110 (-2);  
SUB = 1, ADD = 0
  - (b) *A* register = 1100 (-4), *B* register = 1110 (-2);  
SUB = 0, ADD = 1
  - (c) Repeat (b) with ADD = SUB = 0.

- 6-28. For the circuit of Figure 6-14 determine the sum outputs for the following cases.
- (a) A register = 1101 (-3), B register = 0011 (+3);  
SUB = 1, ADD = 0.
- (b) A register = 1100 (-4), B register = 0010 (+2);  
SUB = 0, ADD = 1.
- (c) A register = 1011 (-5), B register = 0100 (+4);  
SUB = 1, ADD = 0.
- 6-29. For each of the calculations of Problem 6-27, determine if overflow has occurred.
- 6-30. For each of the calculations of Problem 6-28, determine if overflow has occurred.
- D** 6-31. Show how the gates of Figure 6-14 can be implemented using three 74HC00 chips.
- D** 6-32\* Modify the circuit of Figure 6-14 so that a single control input,  $X$ , is used in place of ADD and SUB. The circuit is to function as an adder when  $X = 0$ , and as a subtractor when  $X = 1$ . Then simplify each set of gates. (*Hint*: Note that now each set of gates is functioning as a controlled inverter.)

**SECTION 6-16**

- B** 6-33. Determine the  $F$ ,  $C_{N+4}$ , and  $OVR$  outputs for each of the following sets of inputs applied to a 74LS382.
- (a)\* [S] = 011, [A] = 0110, [B] = 0011,  $C_N = 0$
- (b) [S] = 001, [A] = 0110, [B] = 0011,  $C_N = 1$
- (c) [S] = 010, [A] = 0110, [B] = 0011,  $C_N = 1$
- D** 6-34. Show how the 74HC382 can be used to produce  $[F] = [\bar{A}]$ . (*Hint*: Recall that special property of an XOR gate.)
- 6-35. Determine the  $\Sigma$  outputs in Figure 6-16 for the following sets of inputs.
- (a)\* [S] = 110, [A] = 10101100, [B] = 00001111
- (b) [S] = 100, [A] = 11101110, [B] = 00110010
- C, D** 6-36. Add the necessary logic to Figure 6-16 to produce a single HIGH output whenever the binary number at A is exactly the same as the binary number at B. Apply the appropriate select input code (three codes can be used).

**SECTION 6-17**

- T** 6-37. Consider the circuit of Figure 6-10. Assume that the  $A_2$  output is stuck LOW. Follow the sequence of operations for adding two numbers, and determine the results that will appear in the A register after the second TRANSFER pulse for each of the following cases. Note that the numbers are given in decimal, and the first number is the one loaded into B by the first LOAD pulse.
- (a)\*  $2 + 3$
- (b)  $3 + 7$
- (c)  $7 + 3$
- (d)  $8 + 3$
- (e)  $9 + 3$



- T** 6-38. A technician breadboards the adder/subtractor of Figure 6-14. During testing, she finds that whenever an addition is performed, the result is 1 more than expected, and when a subtraction is performed, the result is 1 less than expected. What is the likely error that the technician made in connecting this circuit?
- 6-39\* Describe the symptoms that would occur at the following points in the circuit of Figure 6-14 if the ADD and SUB lines were shorted together.
- B[3..0] inputs of the 74LS283 IC
  - C<sub>0</sub> input of the 74LS283 IC
  - SUM ( $\Sigma$ ) [3..0] outputs
  - C<sub>4</sub>

### SECTION 6-18

- N** 6-40. Functionally simulate (with 15 test cases) the 8-bit adder given in:
- Figure 6-19(a)
  - Figure 6-19(b)
- N, D** 6-41. Design a mod-16 binary, up/down counter using LPM\_FF and LPM\_ADD\_SUB megafunctions. The count direction will be controlled by an input named UP\_DN. Functionally simulate your design to verify that it operates correctly.

### SECTION 6-19

Problems 6-42 through 6-47 deal with the same two arrays,  $a$  and  $b$ , which we will assume have been defined in an HDL source file and have the following values:  $[a] = [10010111]$ ,  $[b] = [00101100]$ . Output array  $[z]$  is also an eight-bit array. Answer Problems 6-42 through 6-47 based on this information. (Assume undefined bits in  $z$  are 0.)

- B, H** 6-42. Declare these data objects using your favorite HDL syntax.
- B, H** 6-43. Give the value of  $z$  for each expression (identical AHDL and VHDL expressions are given):
- $z[] = a[] \& b[];$   $z \leq a \text{ AND } b;$
  - $z[] = a[] \# b[];$   $z \leq a \text{ OR } b;$
  - $z[] = a[] \$ !b[];$   $z \leq a \text{ XOR NOT } b;$
  - $z[7..4] = a[3..0] \& b[3..0];$   
 $z(7 \text{ DOWNTO } 4) \leq a(3 \text{ DOWNTO } 0) \text{ AND } b(3 \text{ DOWNTO } 0);$
  - $z[7..1] = a[6..0]; z[0] = \text{GND};$   
 $z(7 \text{ DOWNTO } 1) \leq a(6 \text{ DOWNTO } 0); z(0) \leq '0';$
- 6-44. What is the value of each of the following:
- $a[3..0]$   $a(3 \text{ DOWNTO } 0)$
  - $b[0]$   $b(0)$
  - $a[7]$   $a(7)$
- B, H** 6-45. What is the value of each of the following?
- $a[5]$   $a(5)$
  - $b[2]$   $b(2)$
  - $b[7..1]$   $b(7 \text{ DOWNTO } 1)$
- H** 6-46\* Write one or more statements in HDL that will shift all the bits in  $[a]$  one position to the right. The LSB should move to the MSB position. The rotated data should end up in  $z[]$ .

- 6-47. Write one or more HDL statements that will take the upper nibble of  $b$  and place it in the lower nibble of  $z$ . The upper nibble of  $z$  should be zero.
- D, H** 6-48. Refer to Problem 6-23. Modify the code of Figure 6-23 or Figure 6-24 to add an overflow output.

### SECTION 6-20

- B, H** 6-49. Modify Figure 6-23 or Figure 6-24 to make it a 12-bit adder without using constants.
- B, H** 6-50. Modify Figure 6-23 or Figure 6-24 to make it a versatile  $n$ -bit adder module with a constant defining the number of bits.
- D, C, H** 6-51. Write an HDL file to create the equivalent of a 74382 ALU without using a built-in macrofunction.

### DRILL QUESTION

- 6-52. Define each of the following terms.
- |                           |                      |
|---------------------------|----------------------|
| (a) Full adder            | (f) Accumulator      |
| (b) 2's complement        | (g) Parallel adder   |
| (c) Arithmetic/logic unit | (h) Look-ahead carry |
| (d) Sign bit              | (i) Negation         |
| (e) Overflow              | (j) $B$ register     |

### MICROCOMPUTER APPLICATIONS

- C, D** 6-53.\* In a typical microprocessor ALU, the results of every arithmetic operation are usually (but not always) transferred to the accumulator register, as in Figures 6-10 and 6-14. In most microprocessor ALUs, the result of each arithmetic operation is also used to control the states of several special flip-flops called *flags*. These flags are used by the microprocessor when it is making decisions during the execution of certain types of instructions. The three most common flags are:

S (sign flag). This FF is always in the same state as the sign of the last result from the ALU.

Z (zero flag). This flag is set to 1 whenever the result from an ALU operation is exactly 0. Otherwise, it is cleared to 0.

C (carry flag). This FF is always in the same state as the carry from the MSB of the ALU.

Using the adder/subtractor of Figure 6-14 as the ALU, design the logic circuit that will implement these flags. The sum outputs and  $C_4$  output are to be used to control what state each flag will go to upon the occurrence of the TRANSFER pulse. For example, if the sum is exactly 0 (i.e., 0000), the Z flag should be set by the PGT of TRANSFER; otherwise, it should be cleared.

- 6-54.\* In working with microcomputers, it is often necessary to move binary numbers from an eight-bit register to a 16-bit register. Consider the numbers 01001001 and 10101110, which represent +73 and -82, respectively, in the 2's-complement system. Determine the 16-bit representations for these decimal numbers.
- 6-55. Compare the eight- and 16-bit representations for +73 from Problem 6-54. Then compare the two representations for -82. There is a general

rule that can be used to convert easily from eight-bit to 16-bit representations. Can you see what it is? It has something to do with the sign bit of the eight-bit number.

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

---

### SECTION 6-1

1. (a) 11101 (b) 101.111 (c) 10010000  
 2. (a) 011011 (b) 01010110 (c) 00111.0111

### SECTION 6-2

1. (a) 00001101 (b) 11111001 (c) 10000000 2. (a) -29 (b) -64  
 (c) +126 3. -2048 to +2047 4. Seven 5. -32768 6. (a) 10000  
 (b) 10000000 (c) 1000 7. Refer to text in the chapter.

### SECTION 6-3

1. True 2. (a)  $100010_2 = -30_{10}$  (b)  $000000_2 = 0_{10}$

### SECTION 6-4

1. (a)  $01111_2 = +15_{10}$  (b)  $11111_2 = -1_{10}$  2. By comparing the sign bit of the sum with the sign bits of the numbers being added

### SECTION 6-5

1. 1100010

### SECTION 6-7

1. The sum of at least one decimal digit position is greater than 1001 (9).  
 2. The correction factor is added to both the units and the tens digit positions.

### SECTION 6-8

1. 923 2. 3DB 3. 2F, 77EC, 6D

### SECTION 6-9

1. Accumulator register, B register, Arithmetic logic circuits. 2. The accumulator holds the input (augend) before the addition and holds the sum (augend + addend) after the addition. Successive additions produce a running total.

### SECTION 6-10

1. Three; two 2. (a)  $S_2 = 0, C_3 = 1$  (b)  $C_5 = 0$

### SECTION 6-11

1. Compare results to text Section 6-11.

### SECTION 6-12

1. One; four; four 2. 0100

### SECTION 6-13

1.  $16 \text{ adders} \times 2 \text{ ns} = 32 \text{ ns}$ . 2.  $F = 1/T = 1/32 \text{ ns} = 31.25 \text{ MHz}$  3. Use logic circuits to predict "look ahead" carries.
-

**SECTION 6-14**

1. Five chips    2. 240 ns    3. 1

**SECTION 6-15**

1. To add the 1 needed to complete the 2's-complement representation of the number in the *B* register    2. 0010    3. 1101    4. False; the 1's complement appears there.

**SECTION 6-16**

1.  $F = 1011; OVR = 0; C_{N+4} = 0$     2.  $F = 0111; OVR = 1; C_{N+4} = 1$     3.  $F = 1000$   
 4.  $\Sigma = 01101011; C_{N+4} = OVR = 0$     5.  $\Sigma = 11111111$     6. Eight

**SECTION 6-18**

1. Look at the data sheet for an MSI 74283 chip.  
 2. An HDL description of a standard IC that can be used from the library.

**SECTION 6-19**

1. (a) 0001 (b) 1011    2. (a) 1010 (b) 1011    3. toggles[3..0]  
 4. toggles(3 DOWNTO 0)    5. [X] = [1,1,1,1]    6. AHDL:  $xx[] = d[] \# g[]$ ;  
 VHDL:  $x \leq d \text{ OR } g$ ;    7. AHDL:  $xx[2..1] = d[3..2] \# g[1..0]$ ; VHDL:  
 $x(2 \text{ DOWNTO } 1) \leq d(3 \text{ DOWNTO } 2) \text{ XOR } g(1 \text{ DOWNTO } 0)$ ;

**SECTION 6-20**

1. Replace 8 with 4 and 9 with 5 in the bit array ranges.  
 2. Replace 255 with 15 and 511 with 31 in the integer ranges.

**SECTION 6-21**

1. CONSTANT    2. In AHDL, near the top of the source file. In VHDL, in a PACKAGE near the top of the source file.    3. They allow for global changes of the value of a symbol used throughout the code.    4. max\_val -5 represents the number 122.



# COUNTERS AND REGISTERS

## ■ OUTLINE

### Part 1

- 7-1 Asynchronous (Ripple) Counters
- 7-2 Propagation Delay in Ripple Counters
- 7-3 Synchronous (Parallel) Counters
- 7-4 Counters with MOD Numbers  $< 2^N$
- 7-5 Synchronous Down and Up/Down Counters
- 7-6 Presetable Counters
- 7-7 IC Synchronous Counters
- 7-8 Decoding a Counter
- 7-9 Analyzing Synchronous Counters
- 7-10 Synchronous Counter Design

- 7-11 Altera Library Functions for Counters
- 7-12 HDL Counters
- 7-13 Wiring HDL Modules Together
- 7-14 State Machines

### Part 2

- 7-15 Register Data Transfer
- 7-16 IC Registers
- 7-17 Shift-Register Counters
- 7-18 Troubleshooting
- 7-19 Megafunction Registers
- 7-20 HDL Registers
- 7-21 HDL Ring Counters
- 7-22 HDL One-Shots

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Describe the operation and characteristics of synchronous and asynchronous counters.
- Construct counters with MOD numbers less than  $2^N$ .
- Construct both up and down counters.
- Connect multistage counters.
- Analyze and evaluate various types of counters.
- Design arbitrary-sequence synchronous counters.
- Describe several schemes used to decode different types of counters.
- Describe counter circuits using different levels of abstraction in HDL.
- Construct shift-register counters.
- Explain the operation of various types of IC registers.
- Describe shift registers and shift-register counters using HDL.
- Apply troubleshooting techniques used for combinational logic systems to troubleshoot sequential logic systems.

## ■ INTRODUCTION

In Chapter 5, we saw how flip-flops could be connected to function as counters and registers. At that time we studied only the basic counter and register circuits. Digital systems employ many variations of these basic circuits using either standard IC chips that are rapidly becoming obsolete or the current technology of programmable logic devices and custom-integrated circuits. In this chapter, we will look in more detail at the underlying concepts and typical features of different types of counters and registers. Our discussion will range from how logic gates are used to control the flip-flops to create a specific counter or register functionality to the use of hardware description languages to accomplish the same. We will emphasize timing diagrams to illustrate the operation of counter and register circuits. Timing diagrams provide a powerful tool to graphically show the relationships between the signals in a digital system. Digital circuit simulators present their analysis results to us as timing diagrams. This information allows us to determine if the functionality and timing are correct for an application. Timing issues are also becoming ever more critical in dealing with high-speed digital systems. Many systems may be capable of operating at lower speeds but fail at higher frequencies. Being able to interpret timing diagram information is vital to an engineer or technician.

Because there are a great number of topics in this chapter, it has been divided into two parts. In **Part 1**, we will cover the principles of counter

operation, the various counter circuit arrangements, and representative IC counters. **Part 2** will present several types of IC registers, shift-register counters, and troubleshooting. Each part includes a section containing HDL descriptions of counters and registers. A separate list of important terms and summary are provided for each part. Problems for the entire chapter are given after Part 2.

As you progress through this chapter, you will find that you are constantly drawing on your understanding of the material we have covered in the preceding chapters. It is a good idea to go back and review previously covered material whenever you need to.

## PART 1

### 7-1 ASYNCHRONOUS (RIPPLE) COUNTERS

#### OUTCOMES

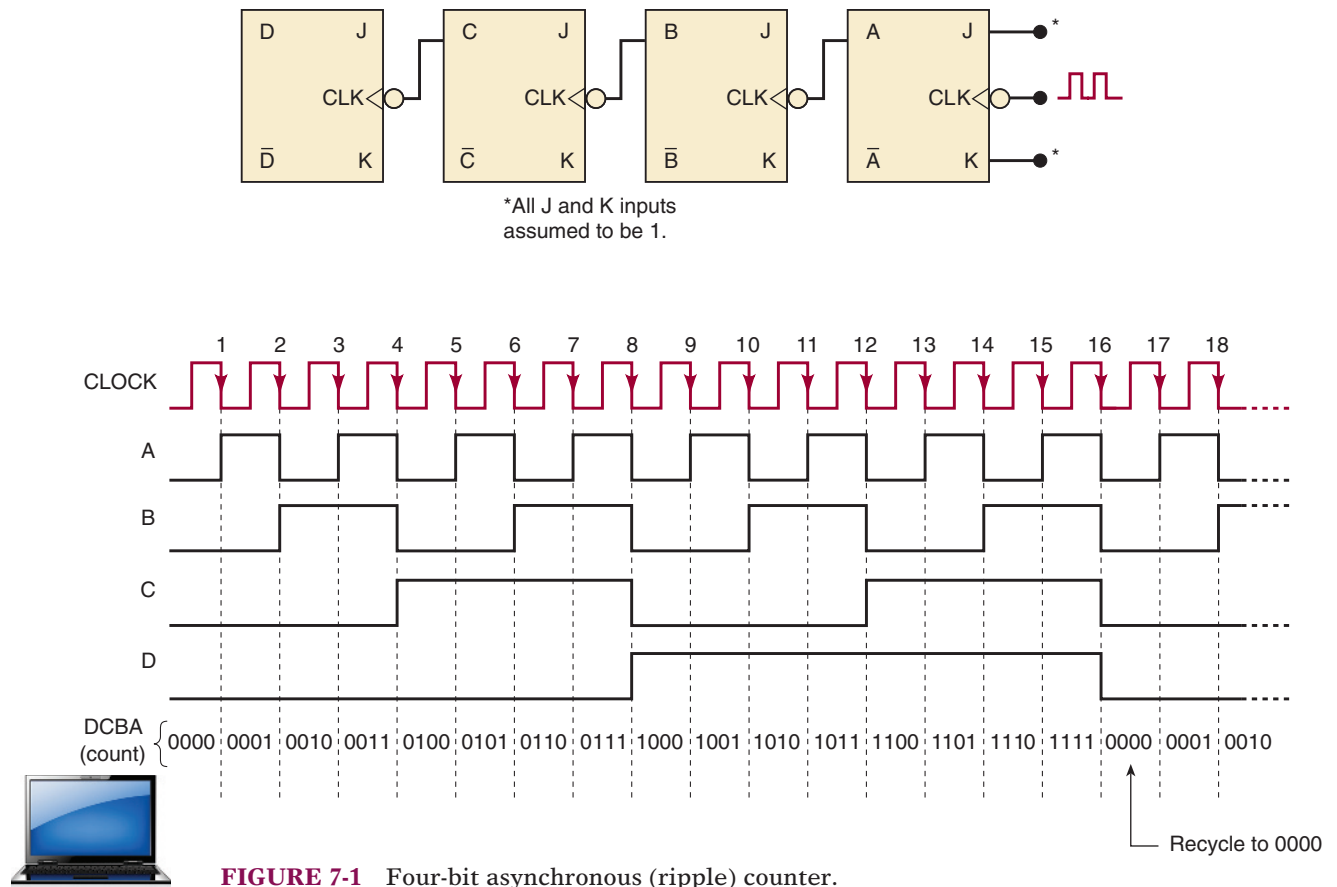
*Upon completion of this section, you will be able to:*

- Cascade flip-flops to form an asynchronous binary counter.
- Draw the timing diagram of a counter.
- Define the MOD number of any counter.
- Use a counter of MOD  $2^N$  to divide the clock frequency by powers of 2.
- Evaluate the effect on frequency and duty cycle of output waveforms if  $\text{Mod \#} < 2^N$ .

Figure 7-1 shows a four-bit binary counter circuit that is designed like the three-bit binary counter discussed in Section 5-18. Recall the following points concerning its operation:

1. The clock pulses are applied only to the *CLK* input of flip-flop A. Thus, flip-flop A will toggle (change to its opposite state) each time the clock pulses make a negative (HIGH-to-LOW) transition. Note that  $J = K = 1$  for all FFs.
2. The normal output of flip-flop A acts as the *CLK* input for flip-flop B, and so flip-flop B will toggle each time the *A* output goes from 1 to 0. Similarly, flip-flop C will toggle when output *B* goes from 1 to 0, and flip-flop D will toggle when output *C* goes from 1 to 0.
3. FF outputs *D*, *C*, *B*, and *A* represent a four-bit binary number, with *D* as the MSB. Let's assume that all FFs have been cleared to the 0 state (CLEAR inputs are not shown). The waveforms in Figure 7-1 show that a binary counting sequence from 0000 to 1111 is followed as clock pulses are continuously applied.
4. After the NGT of the 15th clock pulse has occurred, the counter FFs are in the 1111 condition. On the 16th NGT, flip-flop A goes from 1 to 0, which causes flip-flop B to go from 1 to 0, and so on, until the counter is in the 0000 state. In other words, the counter has gone through one complete cycle (0000 through 1111) and has *recycled* back to 0000. From this point, it will begin a new counting cycle as subsequent clock pulses are applied.

In this counter, each FF output drives the *CLK* input of the next FF. This type of counter arrangement is called an **asynchronous counter** because the



**FIGURE 7-1** Four-bit asynchronous (ripple) counter.

FFs do not change states in exact synchronism with the applied clock pulses; only flip-flop A responds to the clock pulses. FF B must wait for FF A to change states before it can toggle; FF C must wait for FF B, and so on. Thus, there is a delay between the responses of successive FFs. This delay is typically 5–20 ns per FF. In some cases, as we shall see, this delay can be troublesome. This type of counter is also often referred to as a **ripple counter** because of the way the FFs respond one after another in a kind of rippling effect. We will use the terms *asynchronous counter* and *ripple counter* interchangeably.

### Signal Flow

It is conventional in circuit schematics to draw the circuits (wherever possible) so that the signal flow is from left to right, with inputs on the left and outputs on the right. In this chapter, we will often break with this convention, especially in diagrams showing counters. For example, in Figure 7-1, the *CLK* inputs of each FF are on the right, the outputs are on the left, and the input clock signal is shown coming in from the right. We will use this arrangement because it makes the counter operation easier to understand and follow (because the order of the FFs is the same as the order of the bits in the binary number that the counter represents). In other words, FF A (which is the LSB) is the rightmost FF, and FF D (which is the MSB) is the leftmost FF. If we adhered to the conventional left-to-right signal flow, we would have to put FF A on the left and FF D on the right, which is opposite to their positions in the binary number that the counter represents. In some of the counter diagrams later in the chapter, we will employ the conventional left-to-right signal flow so that you will get used to seeing it.



**EXAMPLE 7-1**

The counter in Figure 7-1 starts off in the 0000 state, and then clock pulses are applied. Some time later the clock pulses are removed, and the counter FFs read 0011. How many clock pulses have occurred?

**Solution**

The apparent answer seems to be 3 because 0011 is the binary equivalent of 3. With the information given, however there is no way to tell whether or not the counter has recycled. This means that there could have been 19 clock pulses; the first 16 pulses bring the counter back to 0000, and the last 3 bring it to 0011. There could have been 35 pulses (two complete cycles and then three more), or 51 pulses, and so on.

**MOD Number**

The counter in Figure 7-1 has 16 distinctly different states (0000 through 1111). Thus, it is a *MOD-16 ripple counter*. Recall that the **MOD number** is generally equal to the number of states that the counter goes through in each complete cycle before it recycles back to its starting state. The MOD number can be increased simply by adding more FFs to the counter. That is,

$$\text{MOD number} = 2^N \quad (7-1)$$

where  $N$  is the number of FFs connected in the arrangement of Figure 7-1.

**EXAMPLE 7-2**

A counter is needed that will count the number of items passing on a conveyor belt. A photocell and light source combination is used to generate a single pulse each time an item crosses its path. The counter must be able to count as many as one thousand items. How many FFs are required?

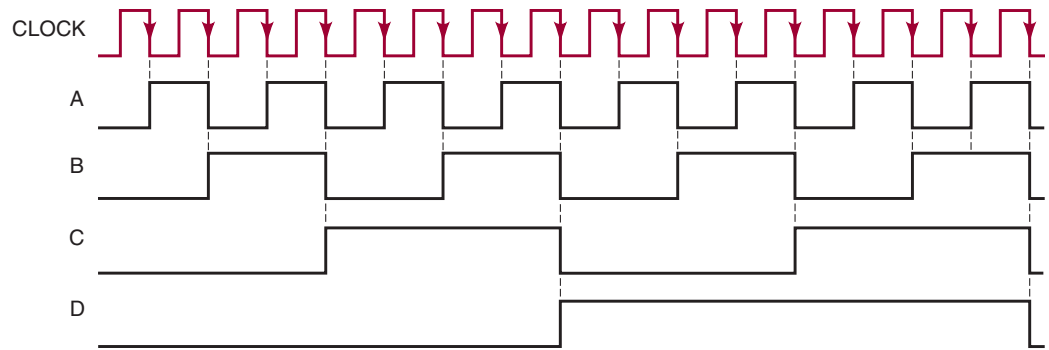
**Solution**

It is a simple matter to determine what value of  $N$  is needed so that  $2^N \geq 1000$ . Since  $2^9 = 512$ , 9 FFs will not be enough.  $2^{10} = 1024$ , so 10 FFs would produce a counter that could count as high as  $1111111111_2 = 1023_{10}$ . Therefore, we should use 10 FFs. We could use more than 10, but it would be a waste of FFs because any FF past the tenth one will not be needed.

**Frequency Division**

In Section 5-18, we saw that in the basic counter each FF provides an output waveform that is exactly *half* the frequency of the waveform at its *CLK* input. To illustrate, suppose that the clock signal in Figure 7-1 is 16 kHz. Figure 7-2 shows the FF output waveforms. The waveform at output *A* is an 8-kHz *square wave*, at output *B* it is 4 kHz, at output *C* it is 2 kHz, and at output *D* it is 1 kHz. Notice that the output of flip-flop *D* has a frequency equal to the original clock frequency divided by 16. In general,

**In any counter, the signal at the output of the last FF (i.e., the MSB) will have a frequency equal to the input clock frequency divided by the MOD number of the counter.**



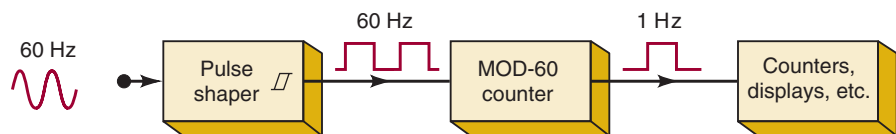
**FIGURE 7-2** Counter waveforms showing frequency division by 2 for each FF.

For example, in a MOD-16 counter, the output from the last FF will have a frequency of  $\frac{1}{16}$  of the input clock frequency. Thus, it can also be called a *divide-by-16 counter*. Likewise, a MOD-8 counter has an output frequency of  $\frac{1}{8}$  the input frequency; it is a *divide-by-8 counter*.

### EXAMPLE 7-3

The first step involved in building a digital clock is to take the 60-Hz signal and feed it into a Schmitt-trigger, pulse-shaping circuit\* to produce a square wave, as illustrated in Figure 7-3. The 60-Hz square wave is then put into a MOD-60 counter, which is used to divide the 60-Hz frequency by exactly 60 to produce a 1-Hz waveform. This 1-Hz waveform is fed to a series of counters, which then count seconds, minutes, hours, and so on. How many FFs are required for the MOD-60 counter?

**FIGURE 7-3** Example 7-3.



### Solution

There is no integer power of 2 that will equal 60. The closest is  $2^6 = 64$ . Thus, a counter using six FFs would act as a MOD-64 counter. Obviously, this will not satisfy the requirement. It seems that there is no solution using a counter of the type shown in Figure 7-1. This is partly true; in Section 7-4, we will see how to modify basic binary counters so that almost *any* MOD number can be obtained and we will not be limited to values of  $2^N$ .

### Duty Cycle

As we can see in Figure 7-2, on each NGT of *CLOCK*, the output of FF A will toggle. With a constant frequency clock signal applied, that means waveform A will be LOW for an amount of time equal to the period of *CLOCK* and then will be HIGH for the same length of time. The amount of time that the signal is HIGH is referred to as the pulse width,  $t_w$ . Flip-flop A produces a periodic output waveform since it will only have a single pulse occurring in each cycle of the repeating waveform. The period of waveform A is the

\*See Section 5-20.

sum of the LOW time and the HIGH time of that signal. Likewise, signal A is used to clock flip-flop B so that output B will be LOW or HIGH for a length of time equal to the period of output A. Flip-flops C and D will have the same action. The pulse width for each of the output signals in our binary counter is exactly half of the period of that waveform. Recall that the **duty cycle** of a periodic waveform is defined as the ratio between the pulse width and the period,  $T$ , of the waveform and is expressed as a percentage.

$$\text{Duty cycle} = \frac{t_w}{T} \times 100\% \quad (7-2)$$

Therefore, we can see that a binary counter, a counter whose  $\text{MOD} = 2^N$ , will always produce output signals that have a 50% duty cycle. As we shall see in later sections of this chapter, if the MOD number for a counter is less than  $2^N$ , then the duty cycle for some FF output signals will not be 50%. In fact, there may be some FF outputs that do not have a defined duty cycle at all because the waveforms do not have a simple periodic pattern. For a counter with a truncated count sequence (i.e.,  $\text{MOD} < 2^N$ ), it will be necessary to analyze the counter's operation to determine count sequences, output signal frequencies, and waveform duty cycles.

### OUTCOME ASSESSMENT QUESTIONS

1. *True or false:* In an asynchronous counter, all FFs change states at the same time.
2. Assume that the counter in Figure 7-1 is holding the count 0101. What will be the count after 27 clock pulses?
3. What would be the MOD number of the counter if three more FFs were added?

## 7-2 PROPAGATION DELAY IN RIPPLE COUNTERS

### OUTCOMES

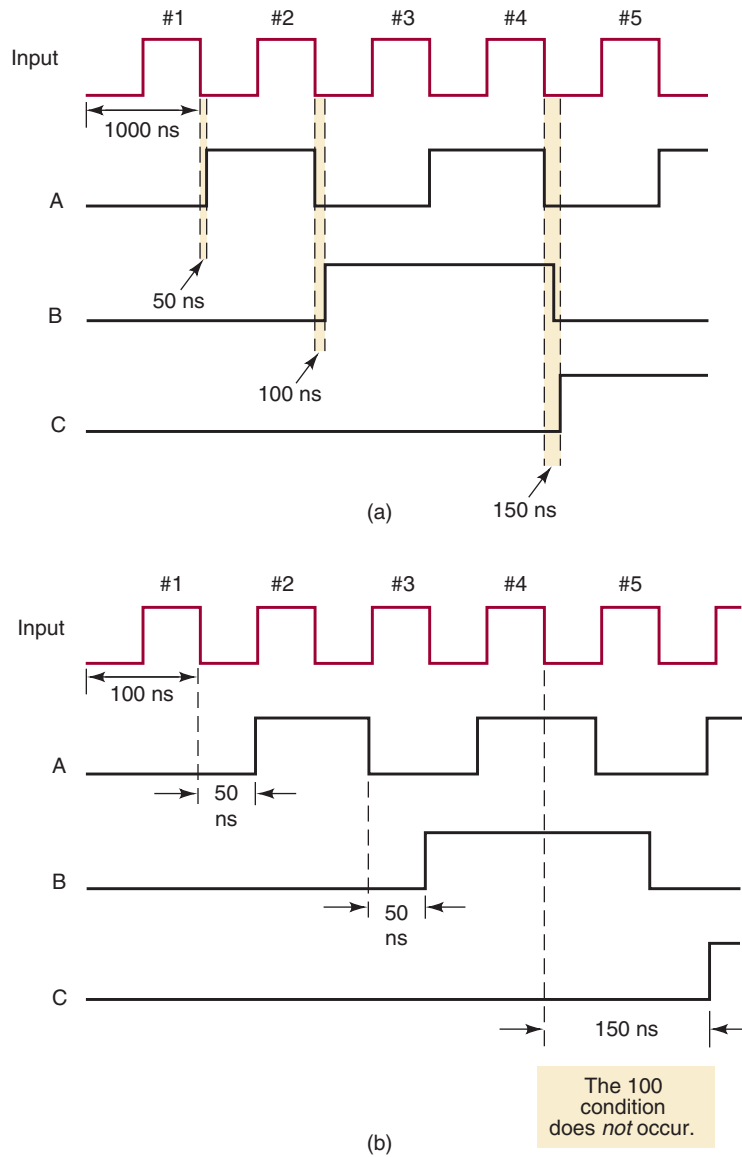
*Upon completion of this section, you will be able to:*

- Predict propagation delays in ripple counters.
- Identify spurious states between counts.
- Determine the maximum operating frequency based on the number of FFs and propagation delays.

Ripple counters are the simplest type of binary counters because they require the fewest components to produce a given counting operation. They do, however, have one major drawback, which is caused by their basic principle of operation: Each FF is triggered by the transition at the output of the preceding FF. Because of the inherent propagation delay time ( $t_{pd}$ ) of each FF, this means that the second FF will not respond until a time  $t_{pd}$  after the first FF receives an active clock transition; the third FF will not respond until a time equal to  $2 \times t_{pd}$  after that clock transition; and so on. In other words, the propagation delays of the FFs accumulate so that the  $N$ th FF cannot change states until a time equal to  $N \times t_{pd}$  after the clock transition occurs. This is illustrated in Figure 7-4, where the waveforms for a three-bit ripple counter are shown.

The first set of waveforms in Figure 7-4(a) shows a situation where an input pulse occurs every 1000 ns (the clock period  $T = 1000$  ns) and it is

**FIGURE 7-4** Waveforms of a three-bit ripple counter illustrating the effects of FF propagation delays for different input pulse frequencies.



assumed that each FF has a propagation delay of 50 ns ( $t_{pd} = 50$  ns). Notice that the A output toggles 50 ns after the NGT of each input pulse. Similarly, B toggles 50 ns after A goes from 1 to 0, and C toggles 50 ns after B goes from 1 to 0. As a result, when the fourth input NGT occurs, the C output goes HIGH after a delay of 150 ns. In this situation, the counter does operate properly in the sense that the FFs do eventually get to their correct states, representing the binary count. However, the situation worsens if the input pulses are applied at a much higher frequency.

The waveforms in Figure 7-4(b) show what happens if the input pulses occur once every 100 ns. Again, each FF output responds 50 ns after the 1-to-0 transition at its CLK input (note the change in the relative time scale). Of particular interest is the situation after the falling edge of the *fourth* input pulse, where the C output does not go HIGH until 150 ns later, which is the same time that the A output goes HIGH in response to the *fifth* input pulse. In other words, the condition  $C = 1, B = A = 0$  (count of 100) never appears because the input frequency is too high. This could cause a serious problem if this condition were supposed to be used to control some

other operation in a digital system. Problems such as this can be avoided if the period between input pulses is made longer than the total propagation delay of the counter. That is, for proper counter operation we need

$$T_{\text{clock}} \geq N \times t_{\text{pd}} \quad (7-3)$$

where  $N$  = the number of FFs. Stated in terms of input-clock frequency, the maximum frequency that can be used is given by

$$f_{\text{max}} = \frac{1}{N \times t_{\text{pd}}} \quad (7-4)$$

For example, suppose that a four-bit ripple counter is constructed using the 74LS112 J-K flip-flop. Table 5-2 shows that the 74LS112 has  $t_{\text{PLH}} = 16$  ns and  $t_{\text{PHL}} = 24$  ns as the propagation delays from  $CLK$  to  $Q$ . To calculate  $f_{\text{max}}$ , we will assume the “worst case”; that is, we will use  $t_{\text{pd}} = t_{\text{PHL}} = 24$  ns, so that

$$f_{\text{max}} = \frac{1}{4 \times 24 \text{ ns}} = 10.4 \text{ MHz}$$

Clearly, as the number of FFs in the counter increases, the total propagation delay increases and  $f_{\text{max}}$  decreases. For example, a ripple counter that uses six 74LS112 FFs will have

$$f_{\text{max}} = \frac{1}{6 \times 24 \text{ ns}} = 6.9 \text{ MHz}$$

Thus, asynchronous counters are not useful at very high frequencies, especially for counters with large numbers of bits. Another problem caused by propagation delays in asynchronous counters occurs when we try to electronically detect (*decode*) the counter’s output states. If you look closely at Figure 7-4(a), for a short period of time (50 ns in our example) right after state 011, you see that state 010 occurs before 100. This is obviously not the correct binary counting sequence, and while the human eye is much too slow to see this temporary state, our digital circuits will be fast enough to detect it. These erroneous count patterns can generate what are called **glitches** in the signals that are produced by digital systems using asynchronous counters. In spite of their simplicity, these problems limit the usefulness of asynchronous counters in digital applications.

#### OUTCOME ASSESSMENT QUESTIONS

1. Explain why a ripple counter’s maximum frequency limitation decreases as more FFs are added to the counter.
2. A certain J-K flip-flop has  $t_{\text{pd}} = 12$  ns. What is the largest MOD counter that can be constructed from these FFs and still operate up to 10 MHz?

### 7-3 SYNCHRONOUS (PARALLEL) COUNTERS

#### OUTCOMES

*Upon completion of this section, you will be able to:*

- Create sequential circuits that operate synchronously.
- Calculate the maximum frequency of a counter that is synchronous.

The problems encountered with ripple counters are caused by the accumulated FF propagation delays; stated another way, the FFs do not all change



## Circuit Operation

For this circuit to count properly, on a given NGT of the clock, only those FFs that are supposed to toggle on that NGT should have  $J = K = 1$  when that NGT occurs. Let's look at the counting sequence in Figure 7-5(b) to see what this means for each FF.

The counting sequence shows that flip-flop A must change states at each NGT. For this reason, its  $J$  and  $K$  inputs are permanently HIGH so that it will toggle on each NGT of the clock input.

The counting sequence shows that flip-flop B must change states on each NGT that occurs while  $A = 1$ . For example, when the count is 0001, the next NGT must toggle B to the 1 state; when the count is 0011, the next NGT must toggle B to the 0 state; and so on. This operation is accomplished by connecting output  $A$  to the  $J$  and  $K$  inputs of flip-flop B so that  $J = K = 1$  only when  $A = 1$ .

The counting sequence shows that flip-flop C must change states on each NGT that occurs while  $A = B = 1$ . For example, when the count is 0011, the next NGT must toggle C to the 1 state; when the count is 0111, the next NGT must toggle C to the 0 state; and so on. By connecting the logic signal  $AB$  to FF C's  $J$  and  $K$  inputs, this FF will toggle only when  $A = B = 1$ .

In a like manner, we can see that flip-flop D must toggle on each NGT that occurs while  $A = B = C = 1$ . When the count is 0111, the next NGT must toggle D to the 1 state; when the count is 1111, the next NGT must toggle D to the 0 state. By connecting the logic signal  $ABC$  to FF D's  $J$  and  $K$  inputs, this FF will toggle only when  $A = B = C = 1$ .

The basic principle for constructing a synchronous counter can therefore be stated as follows:

**Each FF should have its  $J$  and  $K$  inputs connected so that they are HIGH only when the outputs of all lower-order FFs are in the HIGH state.**

## Advantage of Synchronous Counters over Asynchronous

In a parallel counter, all of the FFs will change states simultaneously; that is, they are all synchronized to the NGTs of the input clock pulses. Thus, unlike the asynchronous counters, the propagation delays of the FFs do not add together to produce the overall delay. Instead, the total response time of a synchronous counter like the one in Figure 7-5 is the time it takes *one* FF to toggle plus the time for the new logic levels to propagate through a *single* AND gate to reach the  $J, K$  inputs. That is, for a synchronous counter,

$$\text{total delay} = \text{FF } t_{pd} + \text{AND gate } t_{pd} \quad (7-5)$$

This total delay is the same no matter how many FFs are in the counter, and it will generally be much lower than with an asynchronous counter with the same number of FFs. Thus, a synchronous counter can operate at a much higher input frequency. Of course, the circuitry of the synchronous counter is more complex than that of the asynchronous counter.

## Actual ICs

There are many synchronous IC counters in both the TTL and the CMOS logic families. Some of the most commonly used devices are:

- 74ALS160/162, 74HC160/162: synchronous decade counters
- 74ALS161/163, 74HC161/163: synchronous MOD-16 counters

**EXAMPLE 7-4**

- (a) Determine  $f_{\max}$  for the synchronous counter of Figure 7-5(a) if  $t_{pd}$  for each FF is 50 ns and  $t_{pd}$  for each AND gate is 20 ns. Compare this value with  $f_{\max}$  for a MOD-16 ripple counter.
- (b) What must be done to convert this counter to MOD-32?
- (c) Determine  $f_{\max}$  for the MOD-32 parallel counter.

**Solution**

- (a) In a synchronous counter, the total delay that must be allowed between input clock pulses is equal to FF  $t_{pd}$  + AND gate  $t_{pd}$ . Thus, the period  $T_{\text{clock}} \geq 50 + 20 = 70$  ns, and so the synchronous counter has a maximum frequency of

$$f_{\max} = \frac{1}{T} = \frac{1}{70 \text{ ns}} = 14.3 \text{ MHz (parallel counter)}$$

A MOD-16 ripple counter uses four FFs with  $t_{pd} = 50$  ns. From Eq. 7-3,  $T_{\text{clock}} \geq N \times t_{pd}$ . Thus,  $f_{\max}$  for the ripple counter is

$$f_{\max} = \frac{1}{T} = \frac{1}{4 \times 50 \text{ ns}} = 5 \text{ MHz (ripple counter)}$$

- (b) A fifth FF must be added because  $2^5 = 32$ . The  $CLK$  input of this FF is also tied to the input pulses. Its  $J$  and  $K$  inputs are fed by the output of a four-input AND gate whose inputs are  $A$ ,  $B$ ,  $C$ , and  $D$ .
- (c)  $f_{\max}$  is still determined as in (a) regardless of the number of FFs in the parallel counter. Thus,  $f_{\max}$  is still 14.3 MHz.

**OUTCOME ASSESSMENT QUESTIONS**

1. What is the advantage of a synchronous counter over an asynchronous counter? What is the disadvantage?
2. How many logic devices are required for a MOD-64 parallel counter?
3. What logic signal drives the  $J$ ,  $K$  inputs of the MSB flip-flop for the counter of question 2?

**7-4 COUNTERS WITH MOD NUMBERS  $< 2^N$** **OUTCOMES**

Upon completion of this section, you will be able to:

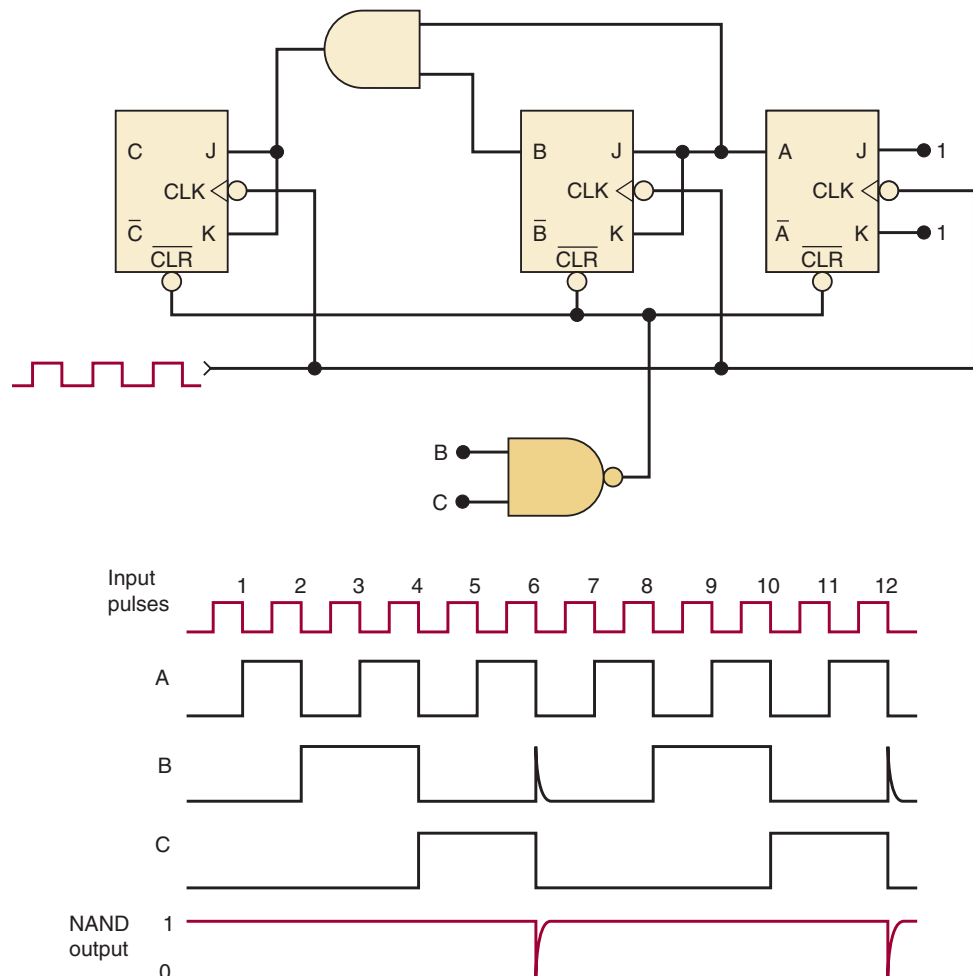
- Modify an  $n$ -bit binary counter to have a MOD # of any integer value less than  $2^N$ .
- Draw the timing of any counter including any temporary states as it recycles.
- Draw a state transition diagram of any counter.
- Determine if each Q output is periodic and if so, its period and frequency.



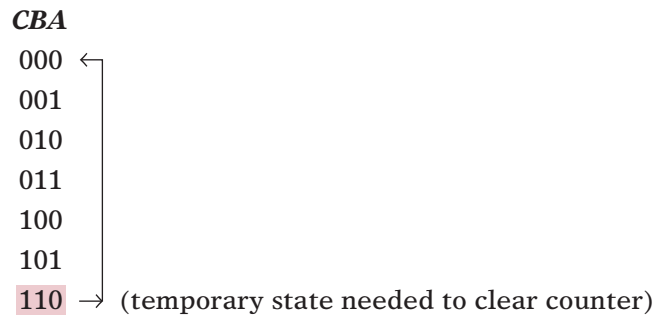
The basic synchronous counter of Figure 7-5 is limited to MOD numbers that are equal to  $2^N$ , where  $N$  is the number of FFs. This value is actually the maximum MOD number that can be obtained using  $N$  flip-flops. The basic counter can be modified to produce MOD numbers less than  $2^N$  by allowing the counter to *skip states* that are normally part of the counting sequence. A truncated count sequence can be produced in a number of different ways. One of the most common methods for doing this is illustrated in Figure 7-6, where a three-bit counter is shown. Disregarding the NAND gate for a moment, we can see that the counter is a MOD-8 binary counter that will count in sequence from 000 to 111. However, the presence of the NAND gate will alter this sequence as follows:

1. The NAND output is connected to the asynchronous  $\overline{CLR}$  inputs of each FF. As long as the NAND output is HIGH, it will have no effect on the counter. When it goes LOW, however, it will clear all of the FFs so that the counter immediately goes to the 000 state.
2. The inputs to the NAND gate are the outputs of flip-flops B and C, and so the NAND output will go LOW whenever  $B = C = 1$ . This condition will occur when the counter goes from the 101 state to the 110 state on the NGT of input pulse 6. The LOW at the NAND output will immediately (generally within a few nanoseconds) clear the counter to the 000 state. Once the FFs have been cleared, the NAND output goes back HIGH because the  $B = C = 1$  condition no longer exists.

**FIGURE 7-6** MOD-6 counter produced by clearing a MOD-8 counter when a count of six (110) occurs.



3. The counting sequence is, therefore



Although the counter does go to the 110 state, it remains there for only a few nanoseconds before it recycles to 000. Thus, we can essentially say that this counter counts from 000 (zero) to 101 (five) and then recycles to 000. It essentially skips 110 and 111 so that it goes through only six different states; thus, it is a MOD-6 counter. Temporary counter states, like the 110 state for this counter, are called **transient states**.

Notice that the waveform at the *B* output contains a *spike* or *glitch* caused by the momentary occurrence of the 110 state before clearing. This glitch is very narrow and so would not produce any visible indication on indicator LEDs or numerical displays. It could, however, cause a problem if the *B* output were being used to drive other circuitry outside the counter. It should also be noted that the *C* output has a frequency equal to one-sixth of the input frequency; in other words, this MOD-6 counter has divided the input frequency by *six*. The waveform at *C* is *not* a symmetrical square wave (50% duty cycle) because it is HIGH for only two clock cycles, whereas it is LOW for four cycles.

### State Transition Diagram

Figure 7-7(a) is the state transition diagram for the MOD-6 counter of Figure 7-6, showing how FFs C, B, and A change states as pulses are applied to the *CLK* input of flip-flop A. Recall that each circle represents one of the possible counter states and that the arrows indicate how one state changes to another in response to an input clock pulse.

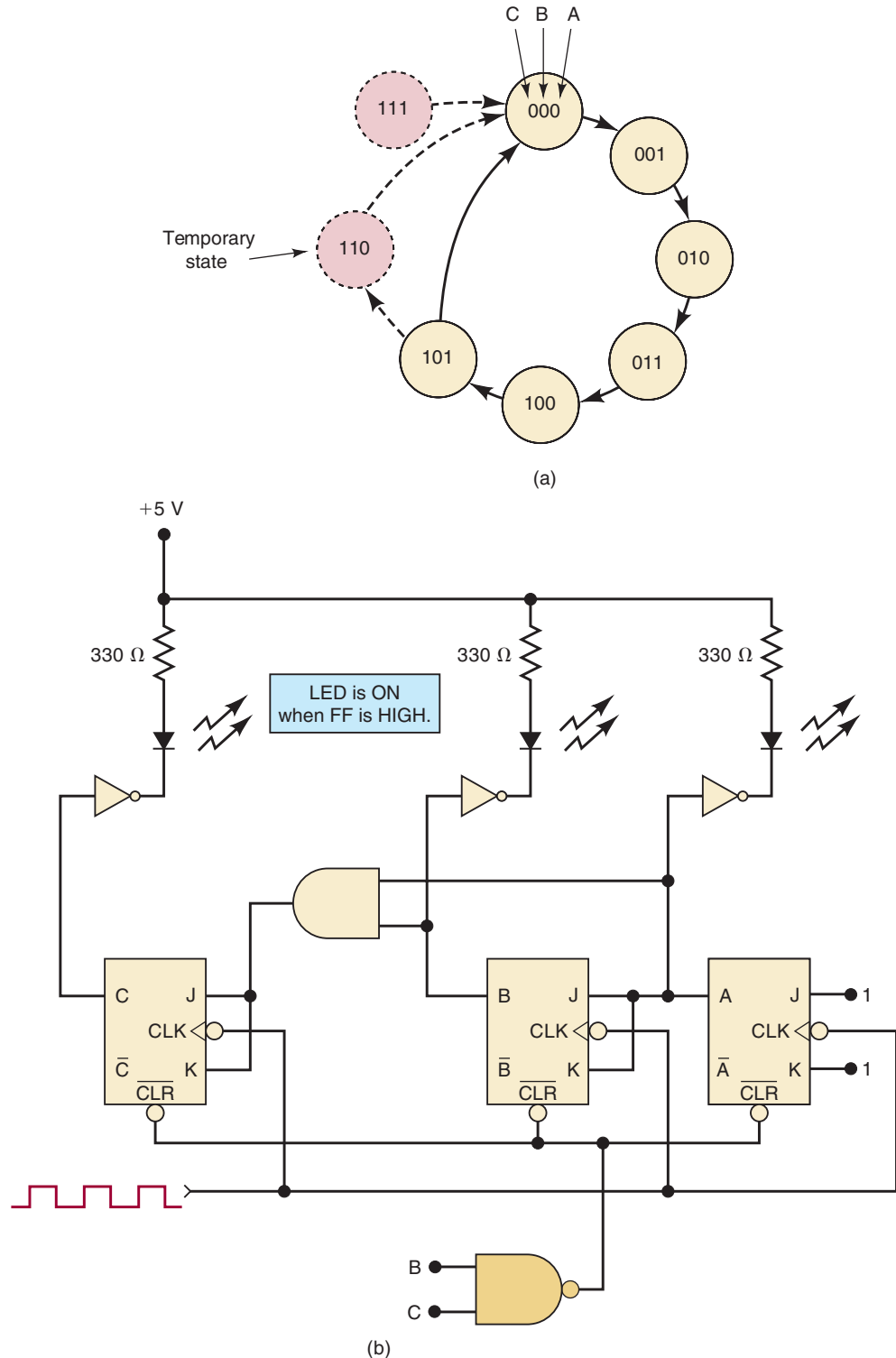
If we assume a starting count of 000, the diagram shows that the states of the counter change normally up until the count of 101. When the next clock pulse occurs, the counter temporarily goes to the 110 count before going to the stable 000 count. The dotted lines indicate the temporary nature of the 110 state. As stated earlier, the duration of this temporary state is so short that for most purposes we can consider that the counter goes directly from 101 to 000 (solid arrow).

Note that there is no arrow into the 111 state because the counter can never advance to that state. However, the 111 state can occur on power-up when the FFs come up in random states. If that happens, the 111 condition will produce a LOW at the NAND gate output and immediately clear the counter to 000. Thus, the 111 state is also a temporary condition that ends up at 000.

### Displaying Counter States

Sometimes during normal operation, and very often during testing, it is necessary to have a visible display of how a counter is changing states in response to the input pulses. We will take a detailed look at several ways of

**FIGURE 7-7** (a) State transition diagram for the MOD-6 counter of Figure 7-6. (b) LEDs are often used to display the states of a counter.



doing this later in the text. For now, Figure 7-7(b) shows one of the simplest methods using individual indicator LEDs for each FF output. Each FF output is connected to an INVERTER whose output provides the current path for the LED. For example, when output A is HIGH, the INVERTER output goes LOW and the LED turns ON. An LED that is turned ON indicates  $A = 1$ . When output A is LOW, the INVERTER output is HIGH and the LED turns OFF. When the LED is turned OFF, it indicates  $A = 0$ .

**EXAMPLE 7-5**

- (a) What will be the status of the LEDs when the counter in Figure 7-7(b) is holding the count of five?
- (b) What will the LEDs display as the counter is clocked by a 1-kHz input?
- (c) Will the 110 state be visible on the LEDs?

**Solution**

- (a) Because  $5_{10} = 101_2$ , the  $2^0$  (A) and  $2^2$  (C) LEDs will be ON, and the  $2^1$  (B) LED will be OFF.
- (b) At 1 kHz, the LEDs will be switching ON and OFF so rapidly that they will appear to the human eye to be ON all the time at about half the normal brightness.
- (c) No; the 110 state will persist for only a few nanoseconds as the counter recycles to 000.

**Changing the MOD Number**

The counter of Figures 7-6 and 7-7 is a MOD-6 counter because of the choice of inputs to the NAND gate. Any desired MOD number can be obtained by changing these inputs. For example, using a three-input NAND gate with inputs A, B, and C, the counter would function normally until the 111 condition was reached, at which point it would immediately reset to the 000 state. Ignoring the very temporary excursion into the 111 state, the counter would go from 000 through 110 and then recycle back to 000, resulting in a MOD-7 counter (seven states).

**EXAMPLE 7-6**

Determine the MOD number of the counter in Figure 7-8(a). Also determine the frequency at the D output.

**Solution**

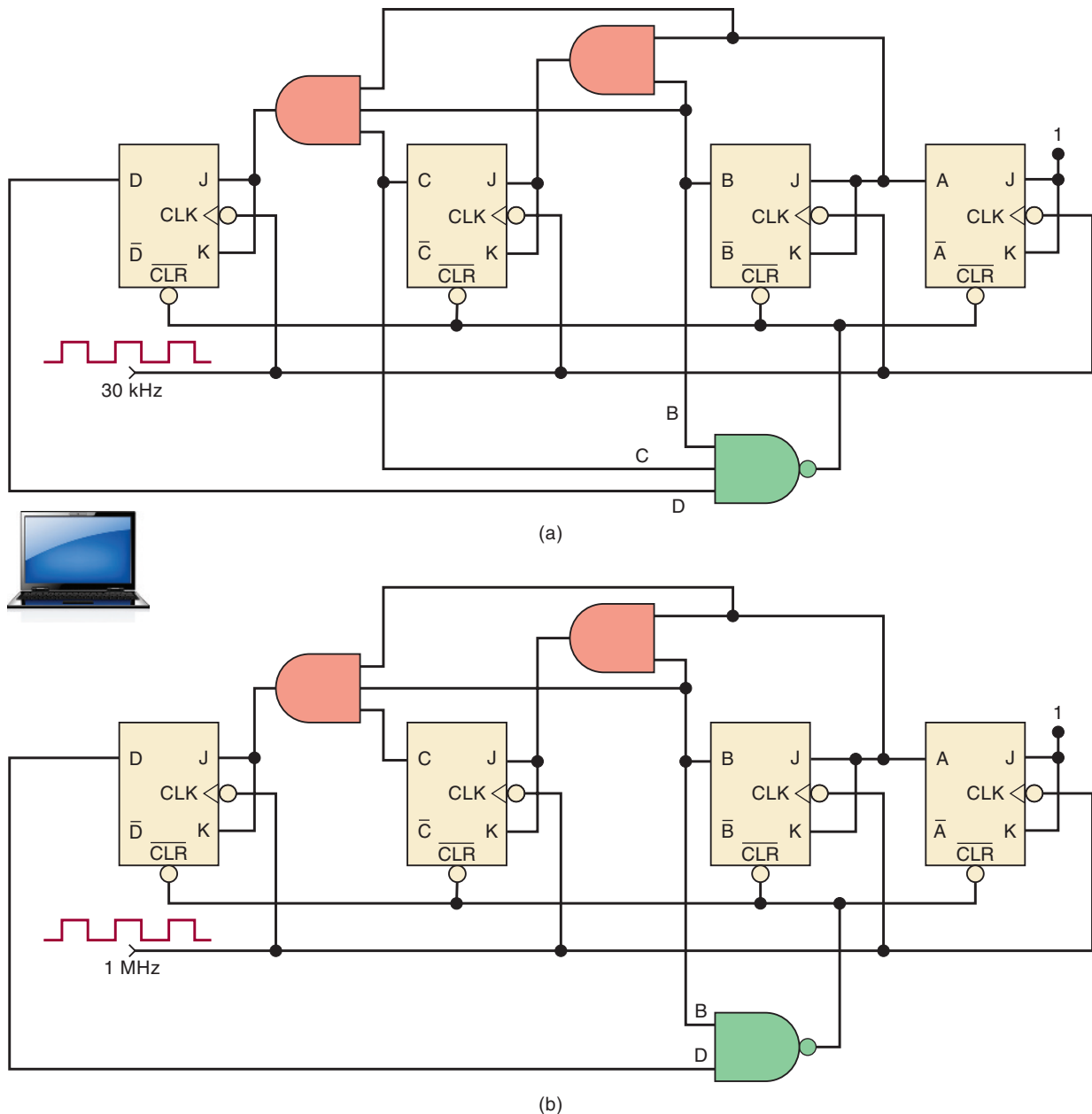
This is a four-bit counter, which would normally count from 0000 through 1111. The NAND inputs are D, C, and B, which means that the counter will immediately recycle to 0000 when the 1110 (decimal 14) count is reached. Thus, the counter actually has 14 stable states 0000 through 1101 and is therefore a MOD-14 counter. Because the input frequency is 30 kHz, the frequency at output D will be

$$\frac{30 \text{ kHz}}{14} = 2.14 \text{ kHz}$$

**General Procedure**

To construct a counter that starts counting from all 0s and has a MOD number of X:

1. Find the smallest number of FFs such that  $2^N \geq X$ , and connect them as a counter. If  $2^N = X$ , do not do steps 2 and 3.
2. Connect a NAND gate to the asynchronous CLEAR inputs of all the FFs.
3. Determine which FFs will be in the HIGH state at a count = X; then connect the normal outputs of these FFs to the NAND gate inputs.



**FIGURE 7-8** (a) MOD-14 counter; (b) MOD-10 (decade) counter.

### EXAMPLE 7-7

Construct a MOD-10 counter that will count from 0000 (zero) through 1001 (decimal 9).

#### Solution

$2^3 = 8$  and  $2^4 = 16$ ; thus, four FFs are required. Because the counter is to have stable operation up to the count of 1001, it must be reset to zero when the count of 1010 is reached. Therefore, FF outputs *D* and *B* must be connected as the NAND gate inputs. Figure 7-8(b) shows the arrangement.

## Decade Counters/BCD Counters

The MOD-10 counter of Example 7-7 is also referred to as a **decade counter**. In fact, a decade counter is any counter that has 10 distinct states, no matter what the sequence. A decade counter such as the one in Figure 7-8(b), which counts in sequence from 0000 (zero) through 1001 (decimal 9), is also commonly called a **BCD counter** because it uses only the 10 BCD code groups 0000, 0001, ..., 1000, and 1001. To reiterate, any MOD-10 counter is a decade counter; and any decade counter that counts in binary from 0000 to 1001 is a BCD counter.

Decade counters, especially the BCD type, find widespread use in applications where pulses or events are to be counted and the results displayed on some type of decimal numerical readout. We shall examine this later in more detail. A decade counter is also often used for dividing a pulse frequency *exactly* by 10. The input pulses are applied to the paralleled clock inputs, and the output pulses are taken from the output of flip-flop D, which has one-tenth the frequency of the input signal.

### EXAMPLE 7-8

In Example 7-3, a MOD-60 counter was needed to divide the 60-Hz line frequency down to 1 Hz. Construct an appropriate MOD-60 counter.

#### Solution

$2^5 = 32$  and  $2^6 = 64$ , and so we need six FFs, as shown in Figure 7-9. The counter is to be cleared when it reaches the count of 60 (111100). Thus, the outputs of flip-flops  $Q_5$ ,  $Q_4$ ,  $Q_3$ , and  $Q_2$  must be connected to the NAND gate. The output of flip-flop  $Q_5$  will have a frequency of 1 Hz.

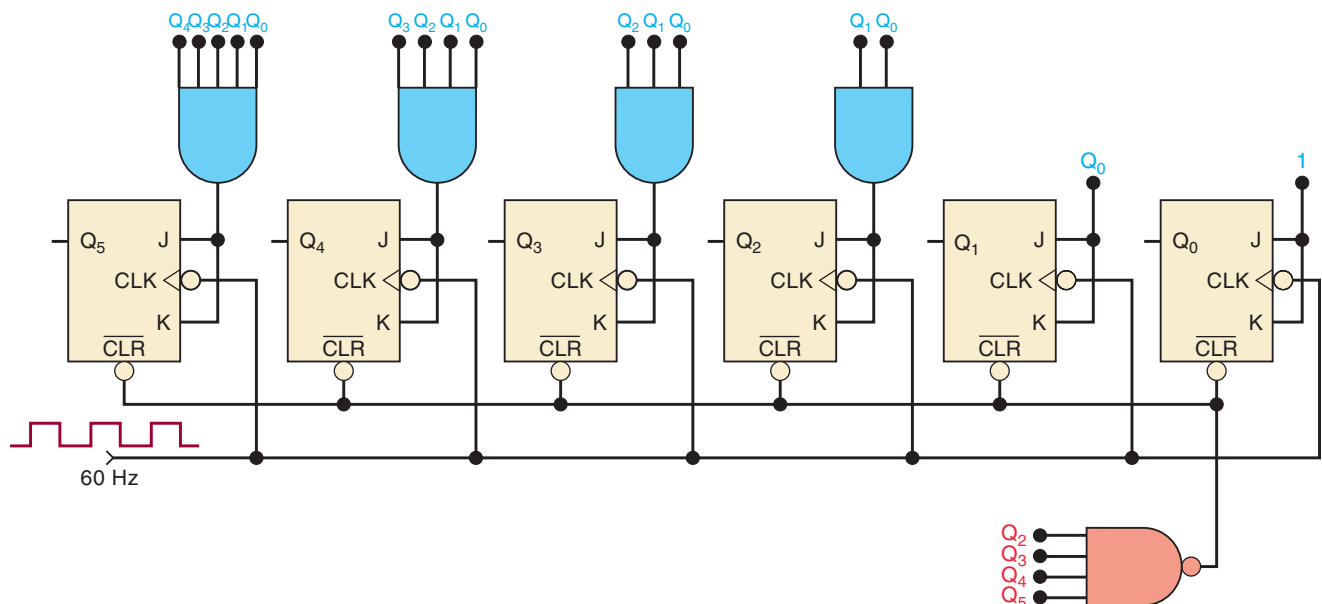


FIGURE 7-9 MOD-60 counter.

### OUTCOME ASSESSMENT QUESTIONS

1. What FF outputs should be connected to the clearing NAND gate to form a MOD-13 counter?
2. *True or false:* All BCD counters are decade counters.
3. What is the MSB output frequency of a decade counter that is clocked from a 50-kHz signal?

## 7-5 SYNCHRONOUS DOWN AND UP/DOWN COUNTERS

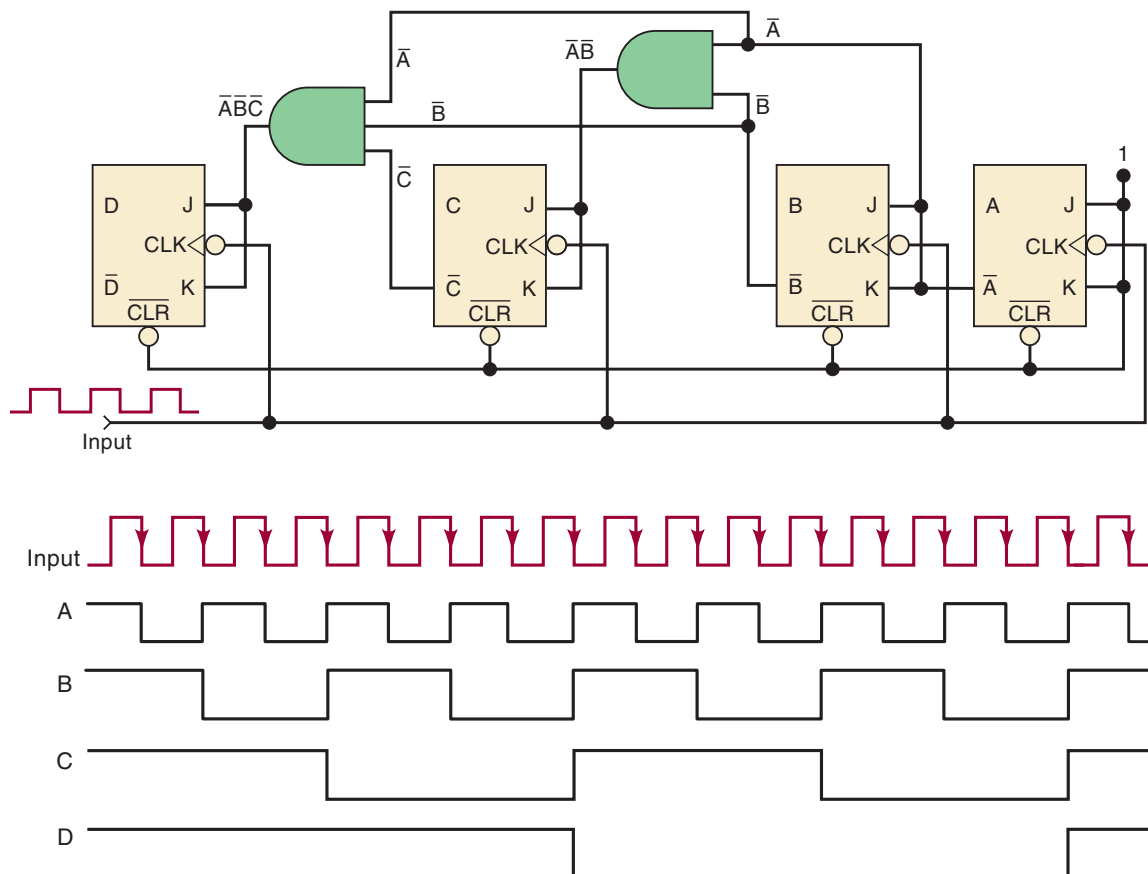
### OUTCOMES

Upon completion of this section, you will be able to:

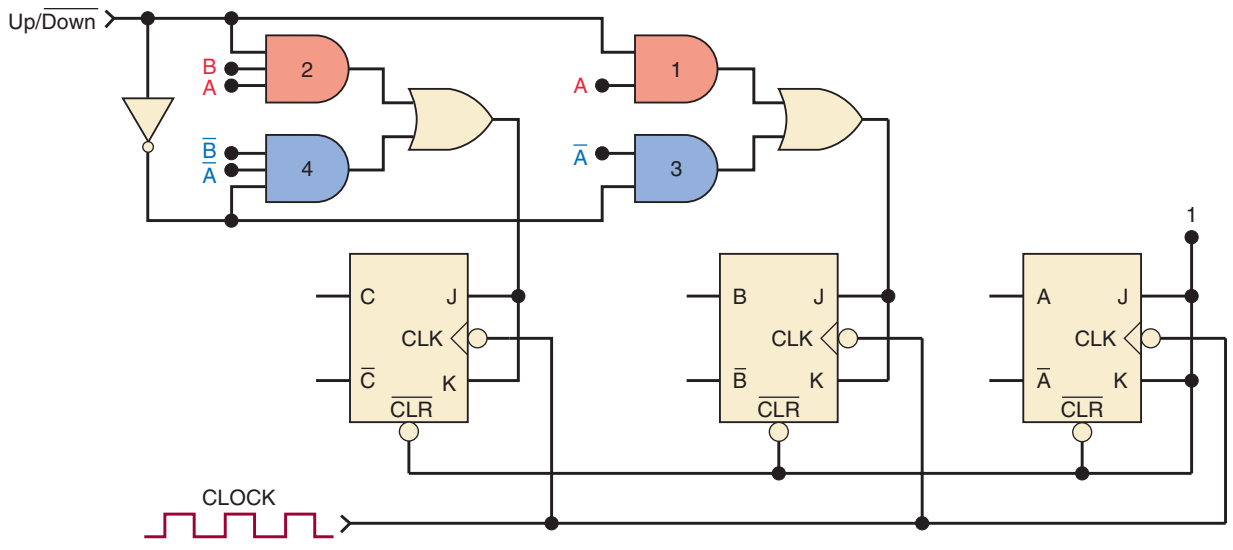
- Create counters that can count up or down.
- Use state transition diagrams to describe the operation of up/down counters.

In Section 7-3, we saw that using the output of lower-order FFs to control the toggling of each FF creates a synchronous **up counter**. A synchronous **down counter** is constructed in a similar manner except that we use the inverted FF outputs to control the higher-order  $J, K$  inputs. Comparing the synchronous, MOD-16, down counter in Figure 7-10 with the up counter in Figure 7-5 shows that we need only to substitute the corresponding inverted FF output in place of the  $A, B,$  and  $C$  outputs. For a down count sequence, the LSB FF  $A$  still needs to toggle with each NGT of the clock input signal. Flip-flop  $B$  must change states on the next NGT of the clock when  $A = 0$  ( $\bar{A} = 1$ ). Flip-flop  $C$  changes states when  $A = B = 0$  ( $\bar{A} \cdot \bar{B} = 1$ ), and flip-flop  $D$  changes states when  $A = B = C = 0$  ( $\bar{A} \cdot \bar{B} \cdot \bar{C} = 1$ ). This circuit configuration will produce the count sequence: 15, 14, 13, 12, ..., 3, 2, 1, 0, 15, 14, and so on, as shown in the timing diagram.

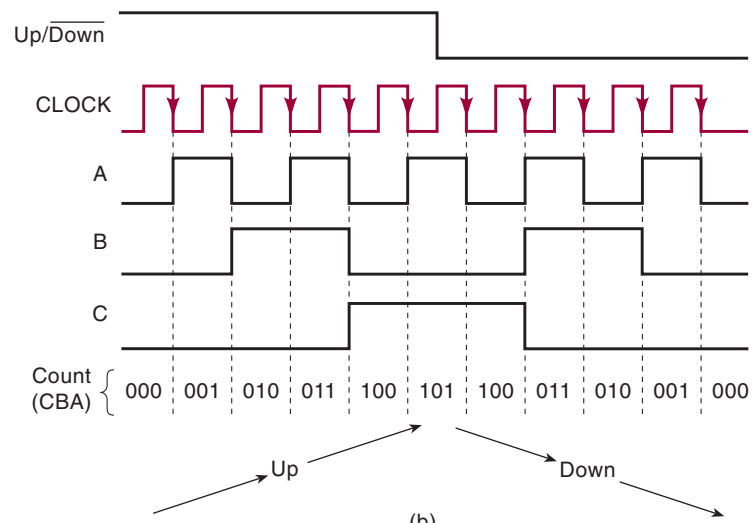
Figure 7-11(a) shows how to form a parallel **up/down counter**. The control input  $Up/\overline{Down}$  controls whether the normal FF outputs or the inverted FF outputs are fed to the  $J$  and  $K$  inputs of the successive FFs. When  $Up/\overline{Down}$



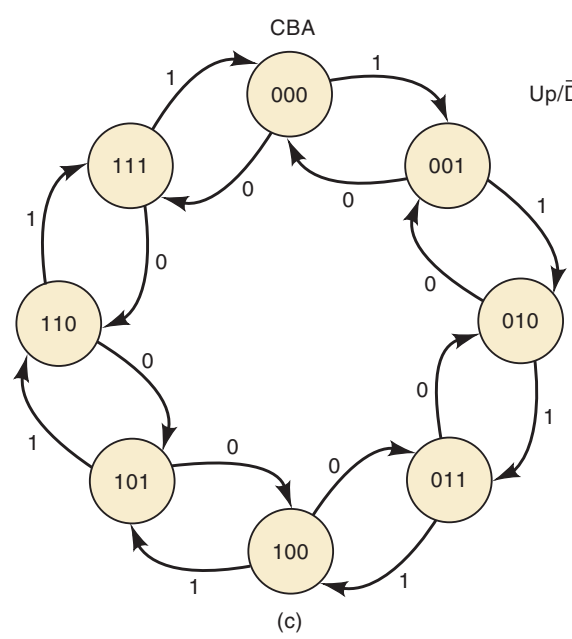
**FIGURE 7-10** Synchronous, MOD-16, down counter and output waveforms.



(a)



(b)



(c)



**FIGURE 7-11** MOD-8 synchronous up/down counter: (a) schematic; (b) sample timing diagram; (c) state transition diagram.



is held HIGH, AND gates 1 and 2 are enabled while AND gates 3 and 4 are disabled (note the inverter). This allows the  $A$  and  $B$  outputs through gates 1 and 2 to control the  $J$  and  $K$  inputs of FFs B and C. When  $Up/\overline{Down}$  is held LOW, AND gates 1 and 2 are disabled while AND gates 3 and 4 are enabled. This allows the inverted  $A$  and  $B$  outputs through gates 3 and 4 to control the  $J$  and  $K$  inputs of FFs B and C.

The waveforms in Figure 7-11(b) illustrate the operation of this counter. Notice that for the first five clock pulses,  $Up/\overline{Down} = 1$  and the counter counts up; for the last five pulses,  $Up/\overline{Down} = 0$ , and the counter counts down.

The state transition diagram for this counter is given in Figure 7-11(c). The arrows represent state transitions that occur on the NGT of the clock signal. Note that there are two arrows leaving each state's bubble. This is referred to as a **conditional transition**. The next state for this counter is, of course, dependent upon the logic level applied to the control input,  $Up/\overline{Down}$ . Each of the arrows must be labeled with the required input control logic level that produces the indicated transition. The name of the control signal is provided as a legend near the state transition diagram.

The nomenclature used for the control signal ( $Up/\overline{Down}$ ) was chosen to make it clear how it affects the counter. The count-up operation is active-HIGH; the count-down operation is active-LOW.

#### EXAMPLE 7-9

What problems might be caused if the  $Up/\overline{Down}$  signal in Figure 7-11(b) changes levels on the NGT of the clock?

#### Solution

The FFs might operate unpredictably because some of them would have their  $J$  and  $K$  inputs changing at about the same time that a NGT occurs at their  $CLK$  input. However, the effects of the change in the control signal must propagate through two gates before reaching the  $J, K$  inputs, so it is more likely that the FFs will respond predictably to the levels that are at  $J, K$  prior to the NGT of  $CLK$ .

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the difference between the counting sequence of an up counter and a down counter?
2. What circuit changes will convert a synchronous, binary up counter into a binary down counter?

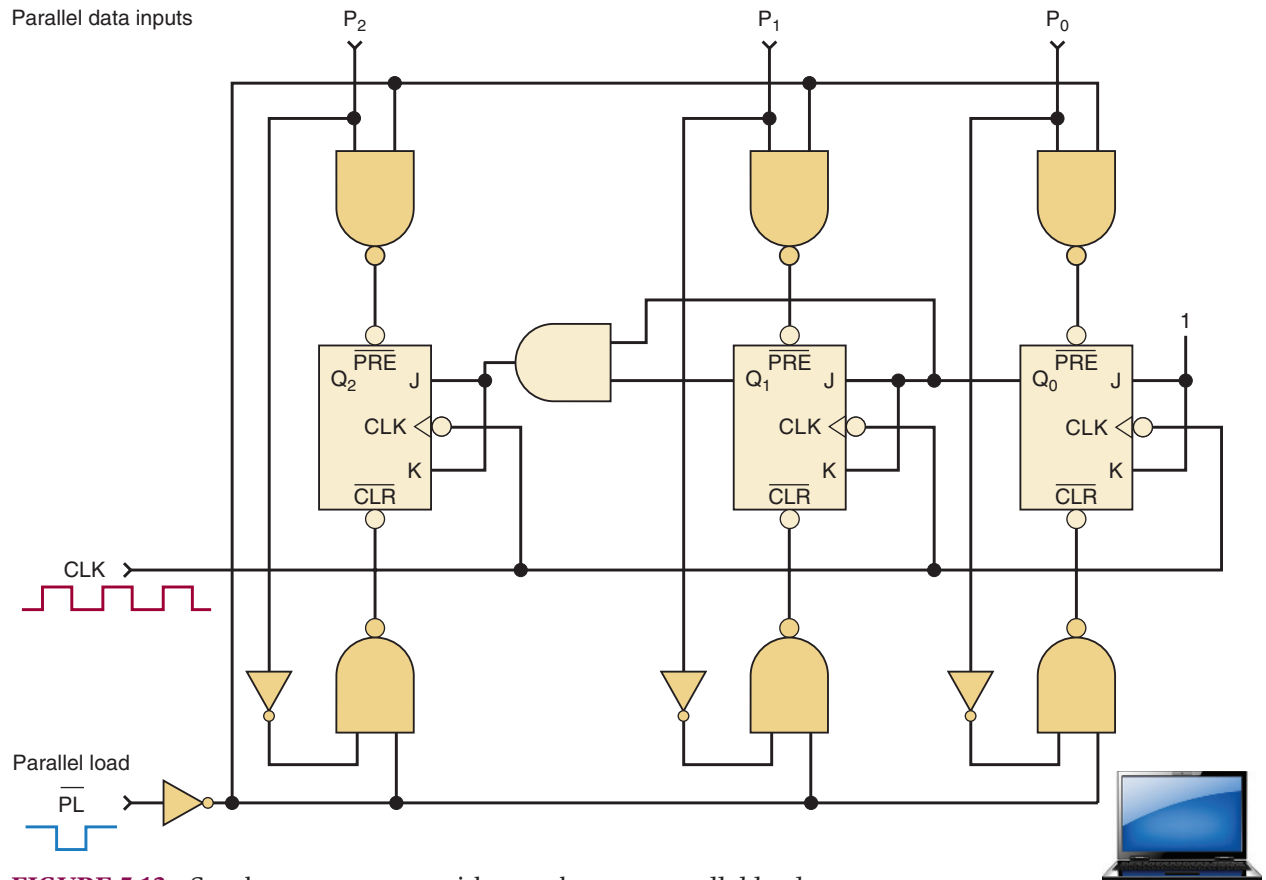
## 7-6 PRESETTABLE COUNTERS

### OUTCOMES

Upon completion of this section, you will be able to:

- Create a counter that can be initialized or “loaded” with any binary value of  $n$  bits.
- Distinguish between synchronous and asynchronous control in the loading process.

Many synchronous (parallel) counters that are available as ICs are designed to be **presettable**; in other words, they can be preset to any desired starting



**FIGURE 7-12** Synchronous counter with asynchronous parallel load.

count either asynchronously (independent of the clock signal) or synchronously (on the active transition of the clock signal). This presetting operation is also referred to as **parallel loading** the counter.

Figure 7-12 shows the logic circuit for a three-bit presettable parallel up counter. The  $J$ ,  $K$ , and  $CLK$  inputs are wired for operation as a parallel up counter. The asynchronous PRESET and CLEAR inputs are wired to perform asynchronous presetting. The counter is loaded with any desired count at any time by doing the following:

1. Apply the desired count to the parallel data inputs,  $P_2$ ,  $P_1$ , and  $P_0$ .
2. Apply a LOW pulse to the PARALLEL LOAD input,  $\overline{PL}$ .

This procedure will perform an asynchronous transfer of the  $P_2$ ,  $P_1$ , and  $P_0$  levels into flip-flops  $Q_2$ ,  $Q_1$ , and  $Q_0$ , respectively (Section 5-16). This *jam transfer* occurs independently of the  $J$ ,  $K$ , and  $CLK$  inputs. The effect of the  $CLK$  input will be disabled as long as  $\overline{PL}$  is in its active-LOW state because each FF will have one of its asynchronous inputs activated while  $\overline{PL} = 0$ . Once  $\overline{PL}$  returns HIGH, the FFs can respond to their  $CLK$  inputs and can resume the counting-up operation starting from the count that was loaded into the counter.

For example, let's say that  $P_2 = 1$ ,  $P_1 = 0$ , and  $P_0 = 1$ . While  $\overline{PL}$  is HIGH, these parallel data inputs have no effect. If clock pulses are present, the counter will perform the normal count-up operation. Now let's say that  $\overline{PL}$  is pulsed LOW when the counter is at the 010 count (i.e.,  $Q_2 = 0$ ,  $Q_1 = 1$ , and  $Q_0 = 0$ ). This LOW at  $\overline{PL}$  will produce LOWs at the  $\overline{CLR}$  input of  $Q_1$  and at the  $\overline{PRE}$  inputs of  $Q_2$  and  $Q_0$  so that the counter will go to the 101

count regardless of what is occurring at the CLK input. The count will hold at 101 until  $\overline{PL}$  is deactivated (returned HIGH); at that time the counter will resume counting up at each clock pulse from the count of 101.

This asynchronous presetting is used by several IC counters, such as the TTL 74ALS190, 74ALS191, 74ALS192, and 74ALS193 and the CMOS equivalents, 74HC190, 74HC191, 74HC192, and 74HC193.

### Synchronous Presetting

Many IC parallel counters use *synchronous presetting* whereby the counter is preset on the active transition of the same clock signal that is used for counting. The logic level on the parallel load control input determines if the counter is preset with the applied input data at the next active clock transition.

Examples of IC counters that use synchronous presetting include the TTL 74ALS160, 74ALS161, 74ALS162, and 74ALS163 and their CMOS equivalents, 74HC160, 74HC161, 74HC162, and 74HC163.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is meant when we say that a counter is presettable?
2. Describe the difference between asynchronous and synchronous presetting.

## 7-7 IC SYNCHRONOUS COUNTERS

### OUTCOMES

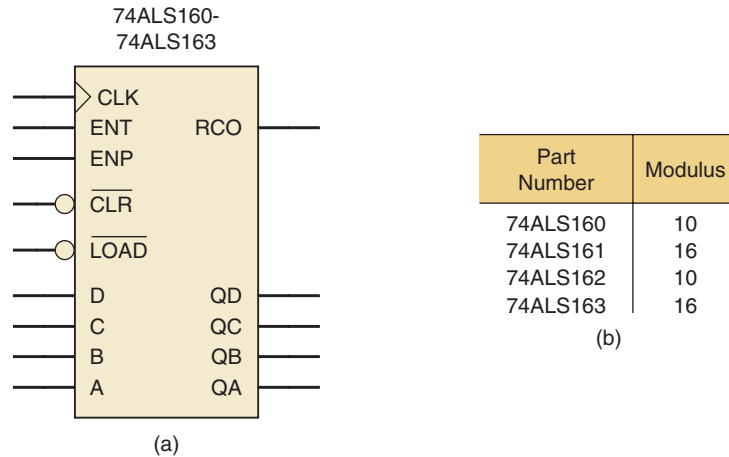
Upon completion of this section, you will be able to:

- Identify and describe common control features found on counters.
- Distinguish between synchronous and asynchronous control inputs.
- Analyze the operation of counter circuits and predict their timing, sequence, modulus, and state transition diagram.
- Cascade counter blocks synchronously.

### The 74ALS160-163/74HC160-163 Series

Figure 7-13 shows the logic symbol, modulus, and function table for the 74ALS160 through 74ALS163 series of IC counters (and the equivalent CMOS counterparts, 74HC160 through 74HC163). These recycling, four-bit counters have outputs labeled  $QD$ ,  $QC$ ,  $QB$ ,  $QA$ , where  $QA$  is the LSB and  $QD$  is the MSB. They are clocked by a PGT applied to  $CLK$ . Each of the four different part numbers has a different combination of two feature variations. As seen in Figure 7-13(b), two of the counters are MOD-10 counters (74ALS160 and 74ALS162), while the other two are MOD-16 binary counters (74ALS161 and 74ALS163). The other variation for these parts is in the operation of the clear function [as highlighted in Figure 7-13(c)]. The 74ALS160 and 74ALS161 each has an asynchronous clear input. This means that as soon as  $\overline{CLR}$  goes LOW ( $\overline{CLR}$  is active-LOW for all four parts), the counter's output will be reset to 0000. On the other hand, the 74ALS162 and 74ALS163 IC counters are synchronously cleared. For these counters to be synchronously cleared, the  $\overline{CLR}$  input must be LOW and a PGT must be applied to the clock input. The clear input has priority over all other functions for this series of IC counters. Clear will override all other control inputs, as indicated by the Xs in the Figure 7-13(c) function table.

**FIGURE 7-13** 74ALS160-74ALS163 series synchronous counters: (a) logic symbol; (b) modules; (c) function table.



**74ALS160-74ALS163 Function Table**

$\overline{\text{CLR}}$	$\overline{\text{LOAD}}$	ENP	ENT	CLK	Function	Part Numbers
L	X	X	X	X	Asynch. clear	74ALS160 & 74ALS161
L	X	X	X	↑	Synchr. clear	74ALS162 & 74ALS163
H	L	X	X	↑	Synchr. load	All
H	H	H	H	↑	Count up	All
H	H	L	X	X	No change	All
H	H	X	L	X	No change	All

(c)

The second priority function available in this series of IC counters is the parallel loading of data into the counter's flip-flops. To preset a data value, make the clear input inactive (HIGH), apply the desired four-bit value to the data input pins *D*, *C*, *B*, *A* (*A* is LSB and *D* is MSB), apply a LOW to the  $\overline{\text{LOAD}}$  input control, and then clock the chip with a PGT. The load function is therefore synchronous and has priority over counting, so it does not matter what logic levels are applied to *ENT* or *ENP*. To count from the preset state it will be necessary to disable the load (with a HIGH) and enable the count function. If the load function is inactive, it does not matter what is applied to the data input pins.

To enable counting, the lowest-priority function, both  $\overline{\text{CLR}}$  and  $\overline{\text{LOAD}}$  control inputs must be inactive. Additionally, there are two active-HIGH count enable controls, *ENT* and *ENP*. *ENT* and *ENP* are essentially ANDed together to control the count function. If either or both of the **count enable** controls is inactive (LOW), the counter will hold the current state. Therefore, to increment the count with each PGT on *CLK*, all four of the control inputs must be HIGH. When counting, the decade counters (74ALS160 and 74ALS162) will automatically recycle to 0000 after state 1001 (9) and the binary counters (74ALS161 and 74ALS163) will automatically recycle after 1111 (15).

This series of IC counter chips has one more output pin, *RCO*. The function of this active-HIGH output is to detect (*decode*) the last or terminal state of the counter. The terminal state for a decade counter is 1001 (9), while the terminal state for a MOD-16 counter is 1111 (15). *ENT*, the primary count enable input, also controls the operation of *RCO*. *ENT* must be HIGH for the counter to indicate with the *RCO* output that it has reached its terminal state. You will see that this feature is very useful in connecting two or more counter chips together in a multistage arrangement to create larger counters.

## EXAMPLE 7-10

Refer to Figure 7-14, where a 74HC163 has the input signals given in the timing diagram applied. The parallel data inputs are permanently connected as 1100. Assume the counter is initially in the 0000 state, and determine the counter output waveforms.

## Solution

Initially (at  $t_0$ ), the counter's FFs are all LOW. Since this is not the terminal state for the counter, output  $RCO$  will also be LOW. The first PGT on the  $CLK$  input occurs at  $t_1$  and, since all control inputs are HIGH, the counter will increment to 0001. The counter continues to count up with each PGT until  $t_2$ . The  $\overline{CLR}$  input is LOW for  $t_2$ . This will synchronously reset the counter to 0000 at  $t_2$ . After  $t_2$ , the  $\overline{CLR}$  input goes inactive (HIGH) so the counter will start counting up again from 0000 with each subsequent PGT. The  $\overline{LOAD}$

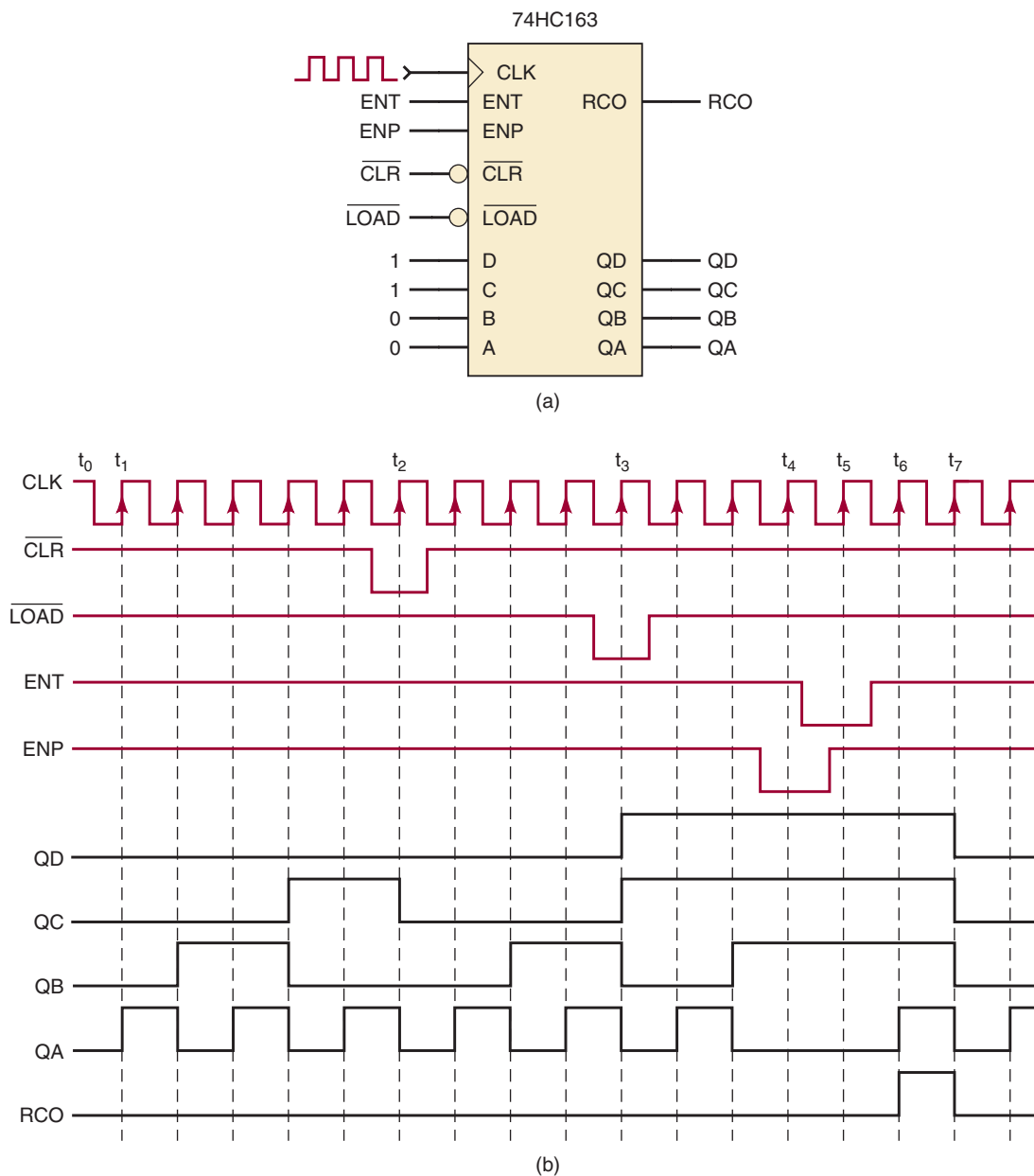
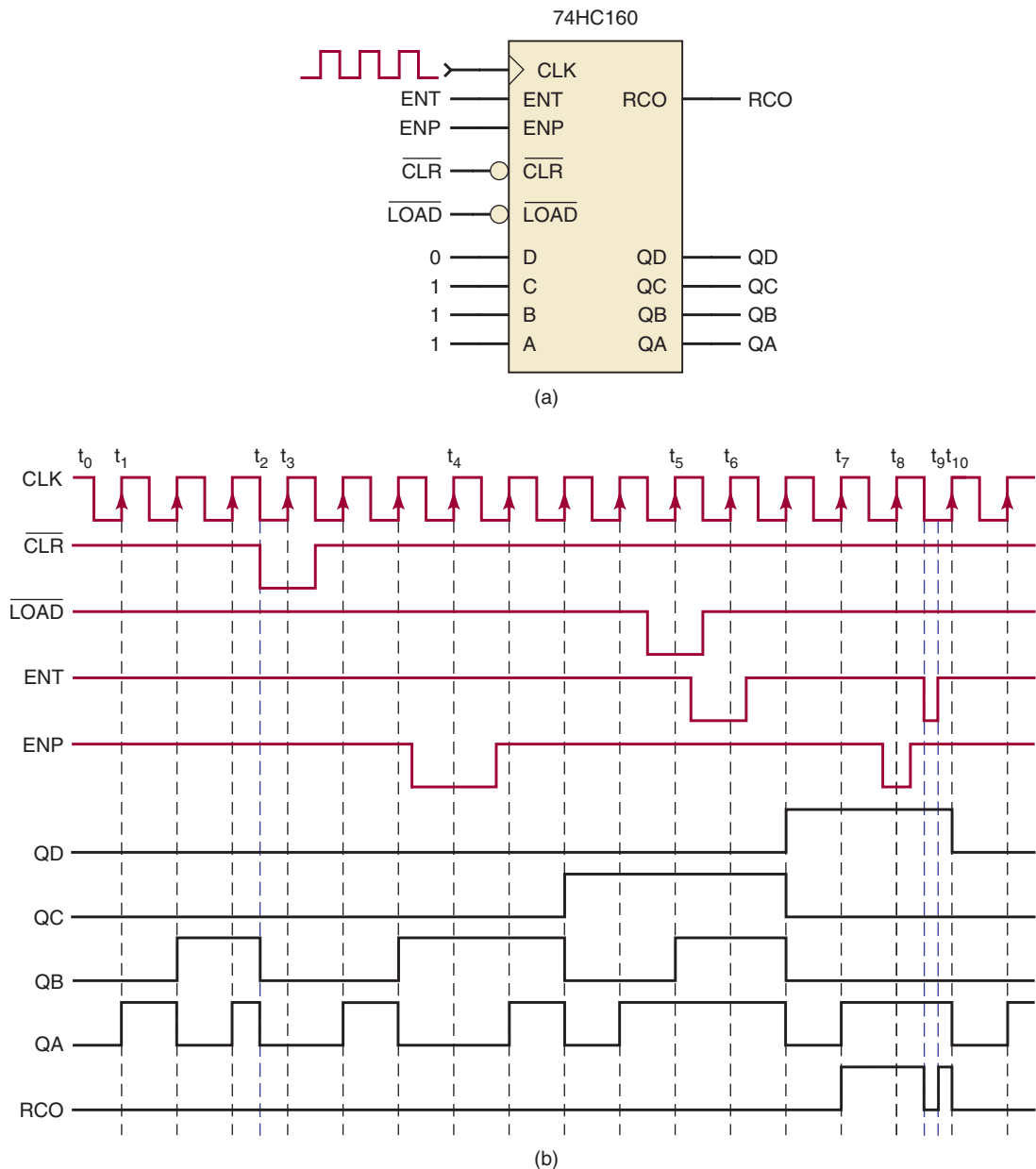


FIGURE 7-14 Example 7-10.

input is LOW for  $t_3$ . This will synchronously load the applied data value 1100 (12) into the counter at  $t_3$ . After  $t_3$ , the  $\overline{LOAD}$  input goes inactive (HIGH), so the counter will continue counting up from 1100 with each subsequent PGT until  $t_4$ . The counter output does not change at  $t_4$  or  $t_5$ , since either  $\overline{ENP}$  or  $\overline{ENT}$  (the count enable inputs) is LOW. This holds the count at 1110 (14). At  $t_6$ , the counter is enabled again and counts up to 1111 (15), its terminal state. As a result, the  $\overline{RCO}$  output now goes HIGH. At  $t_7$ , another PGT on  $\overline{CLK}$  will make the counter recycle to 0000 and  $\overline{RCO}$  returns to a LOW output.

**EXAMPLE 7-11**

Refer to Figure 7-15, where a 74HC160 has the input signals given in the timing diagram applied. The parallel data inputs are permanently connected as 0111. Assume the counter is initially in the 0000 state, and determine the counter output waveforms.



**FIGURE 7-15** Example 7-11.

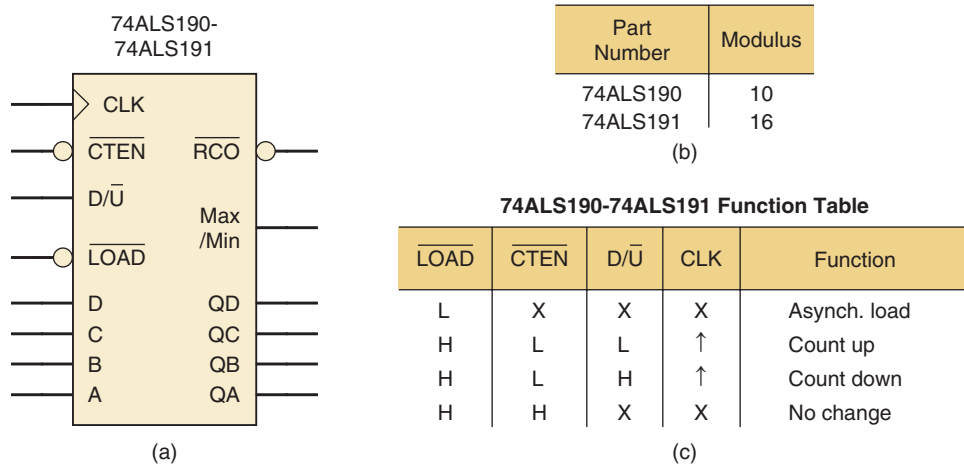
**Solution**

Initially (at  $t_0$ ) the counter's FFs are all LOW. Since this is not the terminal state for the BCD counter, output  $RCO$  will also be LOW. The first PGT on the  $CLK$  input occurs at  $t_1$  and, since all control inputs are HIGH, the counter will increment to 0001. The counter continues to count up with each PGT until  $t_2$ . The asynchronous  $\overline{CLR}$  input goes LOW at  $t_2$  and will immediately reset the counter to 0000 at that point. At  $t_3$ , the  $\overline{CLR}$  input is still active (LOW), so the PGT of the  $CLK$  input will be ignored and the counter will stay at 0000. Later the  $\overline{CLR}$  input goes inactive again and the counter will count up to 0001 and then to 0010. At  $t_4$ , the count enable  $ENP$  is LOW, so the count holds at 0010. For subsequent PGTs of the  $CLK$  input, the counter is enabled and counts up until  $t_5$ . The  $\overline{LOAD}$  input is LOW for  $t_5$ . This will synchronously load the applied data value 0111 (7) into the counter at  $t_5$ . At  $t_6$ , the count enable  $ENT$  is LOW, so the count holds at 0111. For the two subsequent PGTs after  $t_6$ , the counter will continue counting up since it is re-enabled. At  $t_7$ , the BCD counter reaches its terminal state 1001 (9) and the  $RCO$  output now goes HIGH. At  $t_8$ ,  $ENP$  is LOW and the counter stops counting (remaining at 1001). At  $t_9$ , while  $ENT$  is LOW, the  $RCO$  output will be disabled so that it returns to a LOW even though the counter is still at its terminal state (1001). Recall that only  $ENT$  controls the  $RCO$  output. When  $ENT$  returns HIGH during the counter's terminal state,  $RCO$  goes HIGH again. At  $t_{10}$  the counter is enabled, and it recycles to 0000 and then counts to 0001 on the last PGT.

**The 74ALS190-191/74HC190-191 Series**

Figure 7-16 shows the logic symbol, modulus, and function table for the 74ALS190 and 74ALS191 series of IC counters (and the equivalent CMOS counterparts, 74HC190 and 74HC191). These recycling, four-bit counters have outputs labeled  $QD$ ,  $QC$ ,  $QB$ ,  $QA$ , where  $QA$  is the LSB and  $QD$  is the MSB. They are clocked by a PGT applied to  $CLK$ . The only difference between the two part numbers is the counter's modulus. The 74ALS190 is a MOD-10 counter and the 74ALS191 is a MOD-16 binary counter. Both chips are up/down counters and have an asynchronous, active-LOW load input. This means that as soon as  $\overline{LOAD}$  goes LOW, the counter will be preset to the parallel data on the  $D$ ,  $C$ ,  $B$ ,  $A$  ( $A$  is LSB and  $D$  is MSB) input pins. If the

**FIGURE 7-16** 74ALS190-74ALS191 series synchronous counters: (a) logic symbol; (b) modulus; (c) function table.



load function is inactive, it does not matter what is applied to the data input pins. The load input has priority over the counting function.

To count, the  $\overline{LOAD}$  control input must be inactive (HIGH) and the count enable control  $\overline{CTEN}$  must be LOW. The count direction is controlled by the  $D/\overline{U}$  control input. If  $D/\overline{U}$  is LOW, the count is incremented with each PGT on  $CLK$ , while a HIGH on  $D/\overline{U}$  will decrement the count. Both counters automatically recycle in either count direction. The decade counter recycles to 0000 after state 1001 (9) when counting up or to 1001 after state 0000 when counting down. The binary counter will recycle to 0000 after 1111 (15) when counting up or to 1111 after state 0000 when counting down.

These counter chips have two more output pins,  $MAX/MIN$  and  $\overline{RCO}$ .  $MAX/MIN$  is an active-HIGH output that detects (decodes) the terminal state of the counter. Since they are up/down counters, the terminal state depends on the direction of the count. The terminal state (MIN) for either counter when counting down is 0000 (0). However when counting up, the terminal state (MAX) for a decade counter is 1001 (9), while the terminal state for a MOD-16 counter is 1111 (15). Note that  $MAX/MIN$  detects only one state in the count sequence—it just depends on whether it is counting up or down. The active-LOW  $\overline{RCO}$  output also detects the appropriate terminal state for the counter, but it is a bit more complicated. First, it is only enabled when  $\overline{CTEN}$  is LOW. Additionally,  $\overline{RCO}$  will only be LOW while the  $CLK$  input is also LOW. So essentially  $\overline{RCO}$  will mimic the  $CLK$  waveform only during the terminal state while the counter is enabled.

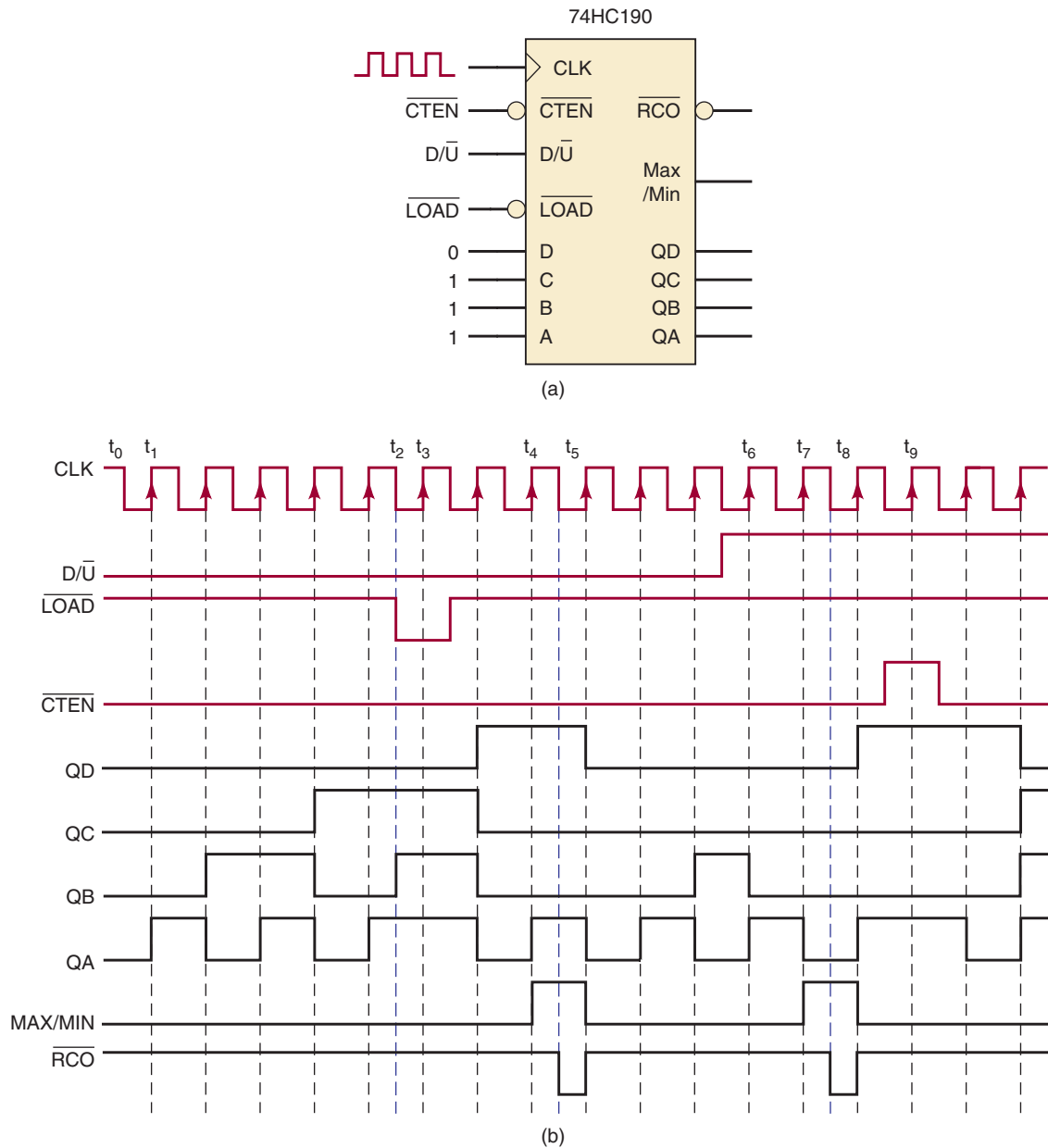
### EXAMPLE 7-12

Refer to Figure 7-17, where a 74HC190 has the input signals given in the timing diagram applied. The parallel data inputs are permanently connected as 0111. Assume the counter is initially in the 0000 state, and determine the counter output waveforms.

#### Solution

Initially (at  $t_0$ ), the counter's FFs are all LOW. Since the counter is enabled ( $\overline{CTEN} = 0$ ) and the count direction control  $D/\overline{U} = 0$ , the BCD counter will start counting up on the first PGT applied to  $CLK$  at  $t_1$  and continues to count up with each PGT until  $t_2$ , where the count has reached 0101. The asynchronous  $\overline{LOAD}$  input goes LOW at  $t_2$  and will immediately load 0111 into the counter at that point. At  $t_3$ , the  $\overline{LOAD}$  input is still active (LOW), so the PGT of the  $CLK$  input will be ignored and the counter will stay at 0111. Later the  $\overline{LOAD}$  input goes HIGH again and the counter will count up to 1000 at the next PGT. At  $t_4$ , the counter increments to 1001, which is the terminal state for a BCD up counter and the  $MAX/MIN$  output goes HIGH. During  $t_5$ , the counter is at its terminal state and the  $CLK$  input is LOW, so  $\overline{RCO}$  goes LOW. For subsequent PGTs of the  $CLK$  input, the counter recycles to 0000 and continues to count up until  $t_6$ . Just prior to  $t_6$ , the  $D/\overline{U}$  control changes to a HIGH. This will make the counter count down at  $t_6$  and again at  $t_7$ , where it will be at state 0000, which now is the terminal state since we are counting down, and  $MAX/MIN$  will output a HIGH. During  $t_8$ , when the  $CLK$  input goes LOW, the  $\overline{RCO}$  output again will be LOW. At  $t_9$ , the counter is disabled with  $\overline{CTEN} = 1$  and the counter holds at 1001. For the subsequent  $CLK$  pulses, the counter continues to count down.





**FIGURE 7-17** Example 7-12.

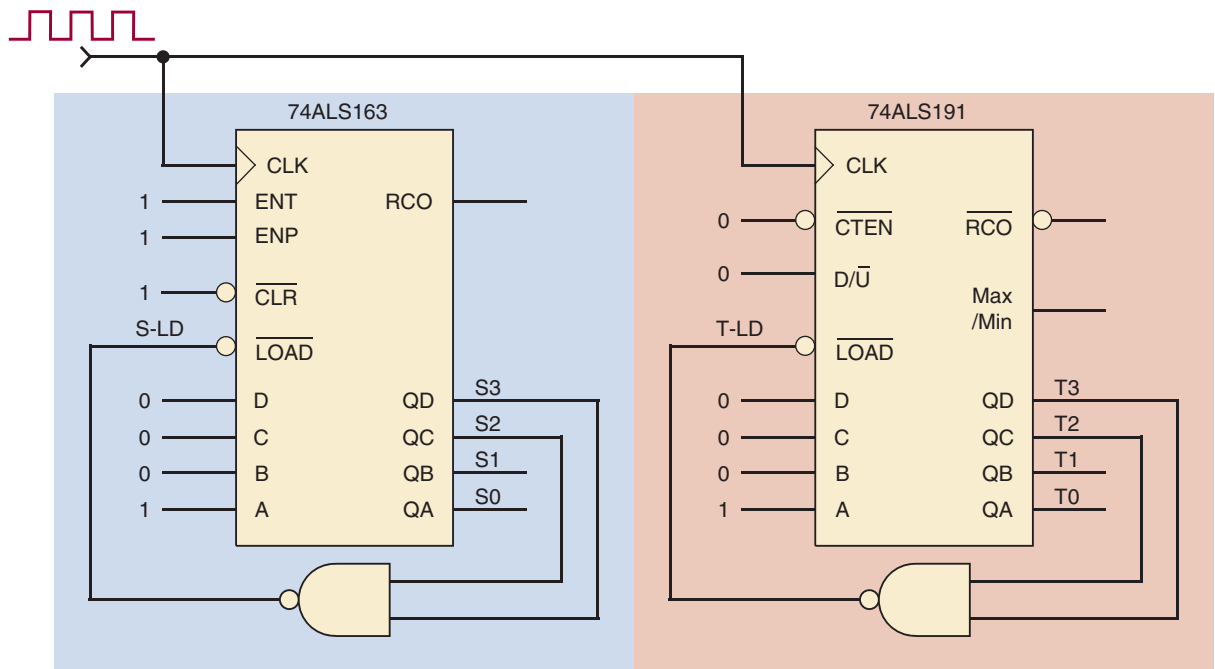
### EXAMPLE 7-13

Compare the operation of two counters, one with synchronous load and the other with asynchronous load. Refer to Figure 7-18(a), in which a 74ALS163 and a 74ALS191 have been wired in a similar fashion to count up in binary. Both chips are driven by the same clock signal and have their  $QD$  and  $QC$  outputs NANDed together to control the respective  $\overline{LOAD}$  input control. Assume that both counters are initially in the 0000 state.

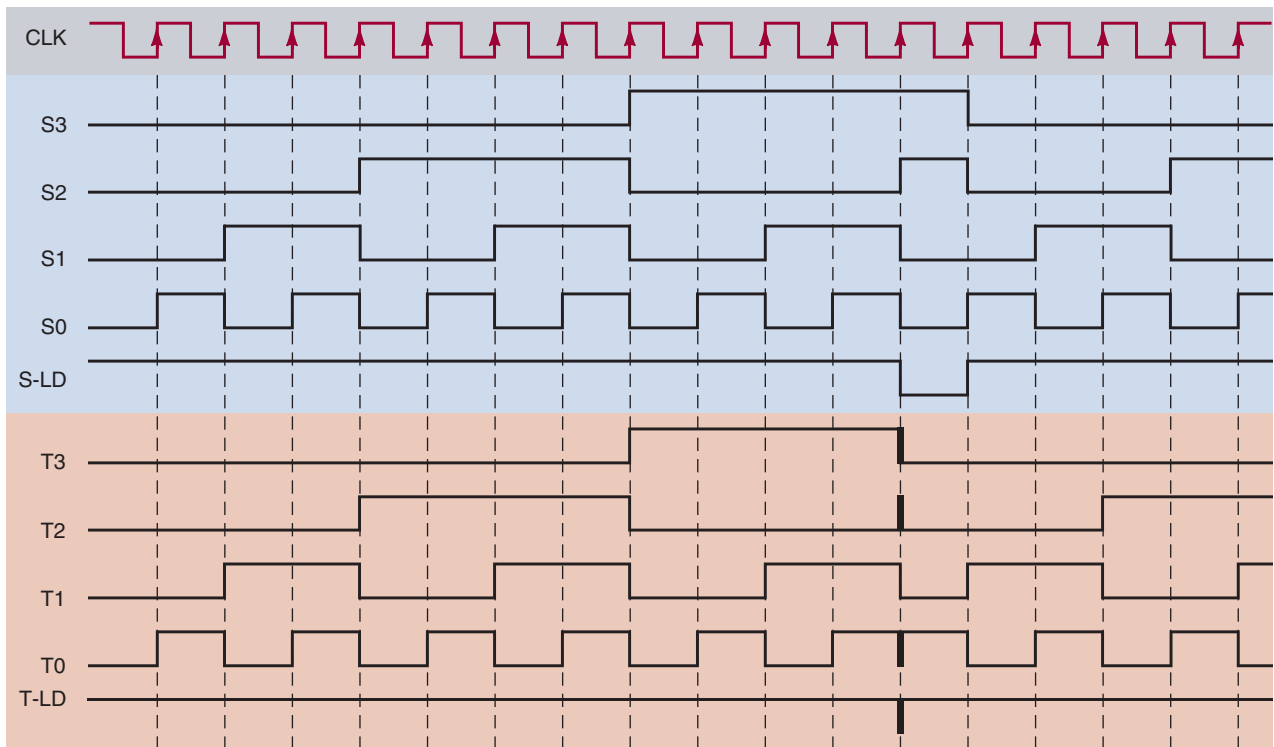
- Determine the output waveform for each counter.
- What is the recycling count sequence and modulus for each counter?
- Why do they have different count sequences?
- Draw the complete (include all 16 states) state transition diagram for each counter.

**Solution**

- (a) Starting at state 0000, each counter will count up until it reaches state 1100 (12) as shown in Figure 7-18(b). The output of each NAND gate will apply a LOW to the respective  $\overline{LOAD}$  input at that time. The 74ALS163 has a synchronous  $\overline{LOAD}$  and will wait until the next PGT on  $CLK$  to load

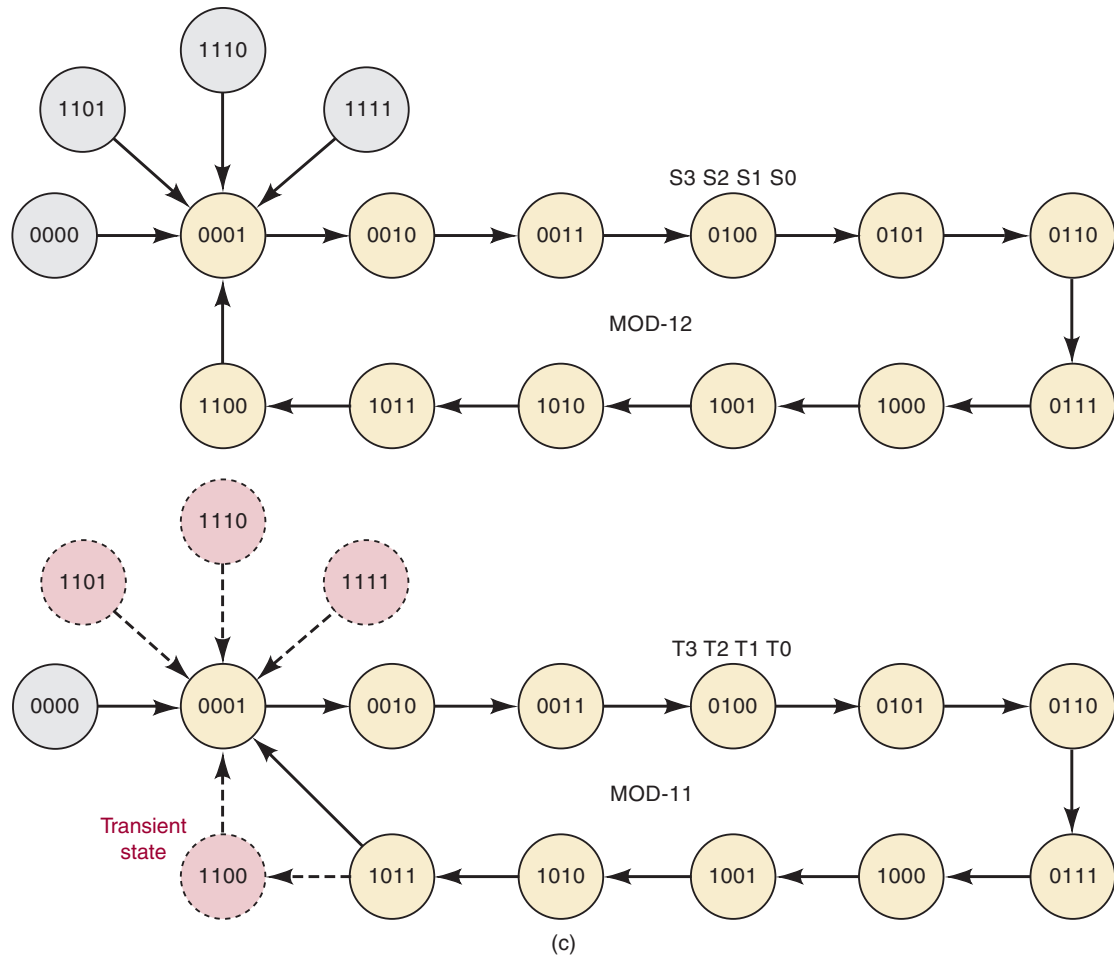


(a)



(b)

**FIGURE 7-18** Example 7-13.



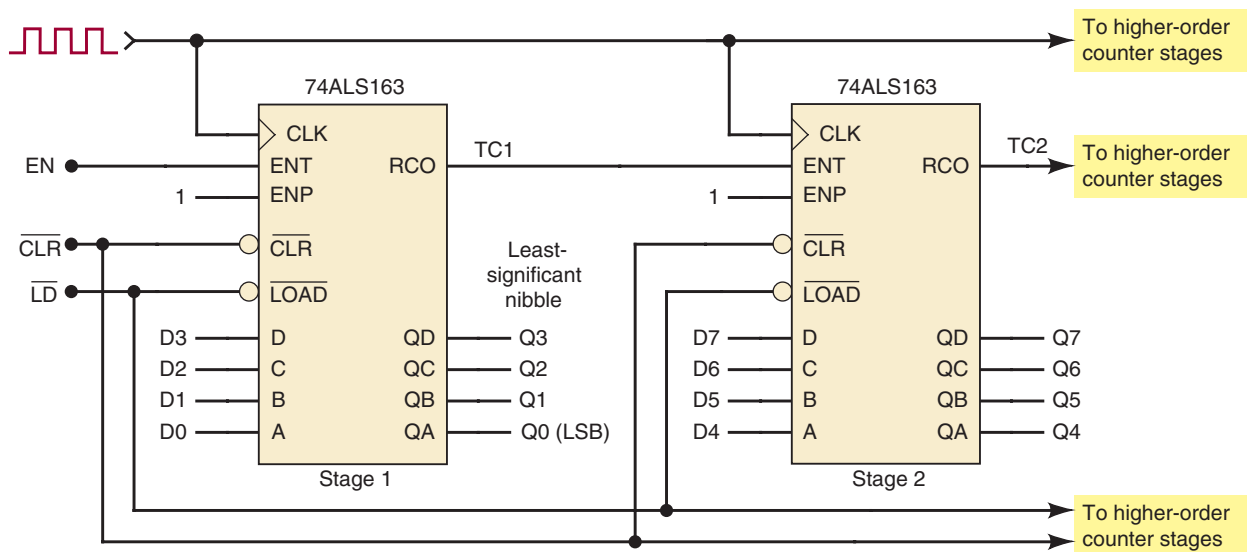
**FIGURE 7-18** Continued

the data input 0001 into the counter. The 74ALS191 has an asynchronous  $\overline{LOAD}$  and will immediately load the data input 0001 into the counter. This will make the 1100 state a temporary or transient state for the 74ALS191. The transient state will produce some spikes or glitches for some of the counter's outputs because of their rapid switching back and forth.

- (b) The 74ALS163 circuit has a recycling count sequence of 0001 through 1100 and is a MOD-12 counter. The 74ALS191 circuit has a recycling count sequence of 0001 through 1011 and is a MOD-11 counter. Transient states are not included in determining the modulus for a counter.
- (c) The counter circuits have different count sequences because one has a synchronous load and the other has an asynchronous load.
- (d) The state transition diagrams are shown in Figure 7-18(c). Both counters will count up until they reach state 1100, at which point, the NAND gate enables the  $\overline{LOAD}$  control. With the 74ALS163, the next state will be 0001 when the counter is clocked. The NAND gate treats the three other states (1101, 1110, and 1111) the same and will load 0001 on the next clock. Since the LOAD function for a 74ALS191 is asynchronous, each of the four states (11XX) that are detected by the NAND gate will immediately load 0001 into the counter. That will make each of those states transient when (or if) they occur. The transient conditions are shown as dotted lines in the state transition diagram. Note that state 0000 does not reoccur in the count sequence for either counter.

## Multistage Arrangement

Many standard IC counters have been designed to make it easy to connect multiple chips together to create circuits with a higher counting range. All of the counter chips presented in this section can be simply connected in a **multistage** or **cascading** arrangement. In Figure 7-19, two 74ALS163s are connected in a two-stage counter arrangement that produces a recycling, binary sequence from 0 to 255 for a maximum modulus of 256. Applying a LOW to the  $\overline{CLR}$  input will synchronously clear both counter stages, and applying a LOW to  $\overline{LD}$  will synchronously preset the eight-bit counter to the binary value on inputs  $D7, D6, D5, D4, D3, D2, D1, D0$  ( $D0 = \text{LSB}$ ). The block on the left (stage 1) is the low-order stage and provides the least-significant counter outputs  $Q3, Q2, Q1, Q0$  (with  $Q0 = \text{LSB}$ ). Stage 2 on the right provides the most-significant counter outputs  $Q7, Q6, Q5, Q4$  (with  $Q7 = \text{MSB}$ ).



**FIGURE 7-19** Two 74ALS163s connected in a two-stage arrangement to extend the maximum counting range.

$EN$ , the enable for the eight-bit counter, is connected to the  $ENT$  input on stage 1. Note that we must use the  $ENT$  input and not  $ENP$ , since only  $ENT$  controls the  $RCO$  output. Using  $ENT$  and  $RCO$  makes cascading very easy. Both counter blocks are clocked together synchronously, but the block on the right (stage 2) is disabled until the least-significant output nibble has reached its terminal state, which will be indicated by the  $TC1$  output. When  $Q3, Q2, Q1, Q0$  reaches 1111 and if  $EN$  is HIGH, then  $TC1$  will output a HIGH. This will allow both counter stages to count up one with the next PGT on the clock. Stage 1 will recycle back to 0000 and stage 2 will increment from its previous output state.  $TC1$  will return to a LOW, since stage 1 is no longer at its terminal state. With subsequent clock pulses, stage 1 will continue to count up if  $EN = 1$  until it again reaches 1111 and the process repeats. When the eight-bit counter reaches 11111111, it will recycle back to 00000000 on the next clock pulse.

Additional 74ALS163 counter chips can be cascaded in the same fashion.  $TC2$  would be connected to the  $ENT$  control on the next chip, and so on.  $TC2$  will be HIGH when  $Q7, Q6, Q5, Q4$  is equal to 1111 and  $TC1$  is HIGH, which in turn means that  $Q3, Q2, Q1, Q0$  is also equal to 1111 and  $EN$  is HIGH. This cascading technique works for all chips (TTL or CMOS families)

in this series, even for the BCD counters. The 74ALS190-191 (or 74HC190-191) series also can be cascaded similarly using the active-LOW  $\overline{CTEN}$  and  $\overline{RCO}$  pins. A multistage counter using 74ALS190-191 chips connected in this fashion can count up or down.

### OUTCOME ASSESSMENT QUESTIONS

1. Describe the function of the inputs  $\overline{LOAD}$  and  $D, C, B, A$ .
2. Describe the function of the  $\overline{CLR}$  input.
3. *True or false:* The 74HC161 cannot be preset while  $\overline{CLR}$  is active.
4. What logic levels must be present on the control inputs in order for the 74ALS162 to count pulses that appear on the  $CLK$ ?
5. What logic levels must be present on the control inputs in order for the 74HC190 to count down with pulses that appear on the  $CLK$ ?
6. What would be the maximum counting range for a four-stage counter made up of 74HC163 ICs? What is the maximum counting range for 74ALS190 ICs?

## 7-8 DECODING A COUNTER

### OUTCOMES

Upon completion of this section, you will be able to:

- Define the decoding operation.
- Decode any state of a counter.
- Provide output logic that is either active-HIGH or active-LOW.

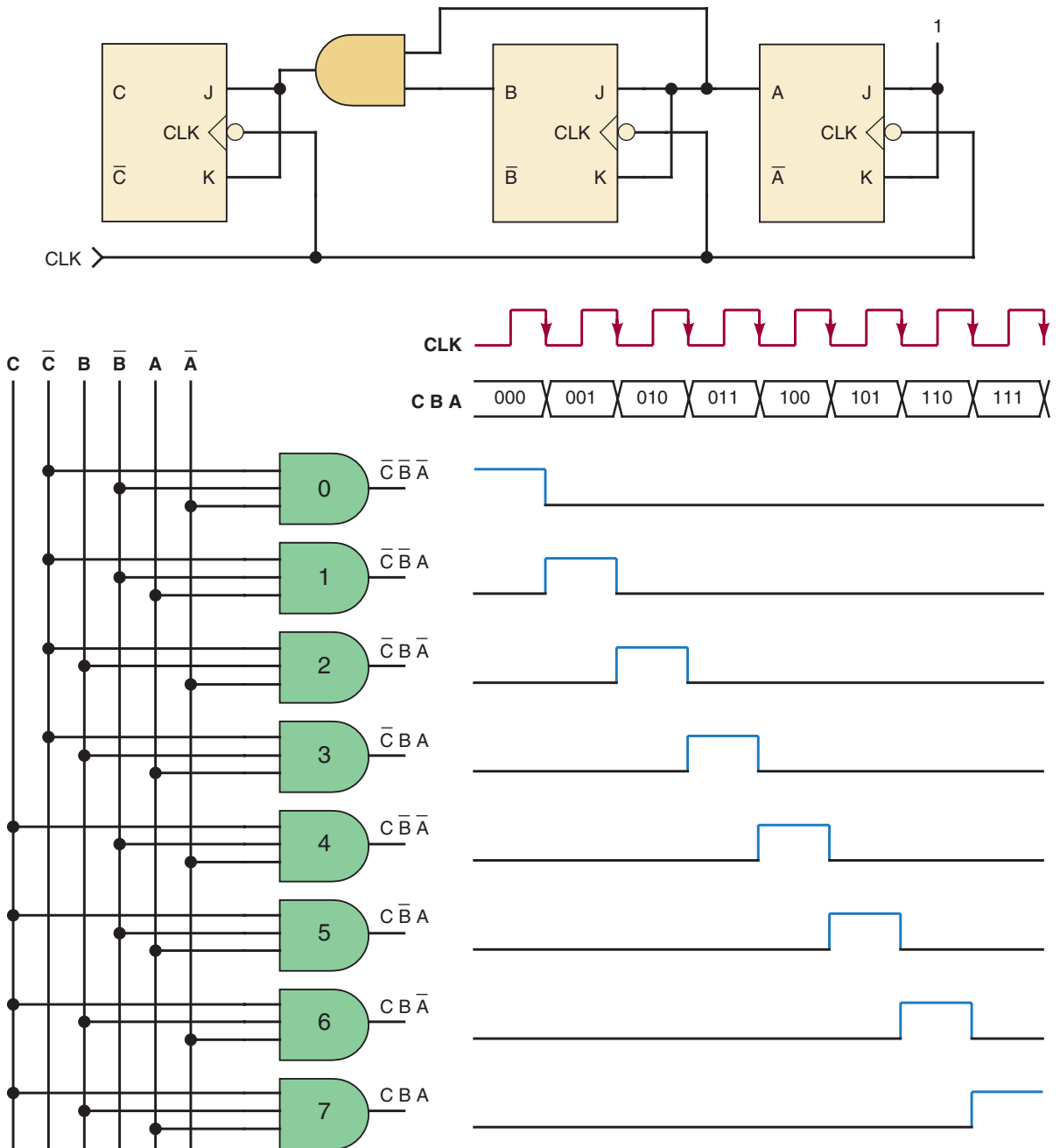
Digital counters are often used in applications where the count represented by the states of the FFs must somehow be determined or displayed. One of the simplest means for displaying the contents of a counter involves just connecting the output of each FF to a small indicator LED [see Figure 7-7(b)]. In this way the states of the FFs are visibly represented by the LEDs (ON = 1, OFF = 0), and the count can be mentally determined by **decoding** the binary states of the LEDs. For instance, suppose that this method is used for a BCD counter and the states of the LEDs are OFF-ON-ON-OFF, respectively. This would represent 0110, which we would mentally decode as decimal 6. Other combinations of LED states would represent the other possible counts.

The indicator LED method becomes inconvenient as the size (number of bits) of the counter increases because it is much harder to decode the displayed results mentally. For this reason, it is preferable to develop a means for *electronically* decoding the contents of a counter and displaying the results in a form that is immediately recognizable and requires no mental operations.

An even more important reason for electronic decoding of a counter occurs because of the many applications in which counters are used to control the timing or sequencing of operations *automatically* without human intervention. For example, a certain system operation might have to be initiated when a counter reaches the 101100 state (count of  $44_{10}$ ). A logic circuit can be used to decode for or detect when this particular count is present and then initiate the operation. Many operations may have to be controlled in this manner in a digital system. Clearly, human intervention in this process would be undesirable except in extremely slow systems.

### Active-HIGH Decoding

A MOD- $X$  counter has  $X$  different states; each state is a particular pattern of 0s and 1s stored in the counter FFs. A decoding network is a logic circuit that generates  $X$  different outputs, each of which detects (decodes) the presence of one particular state of the counter. The decoder outputs can be designed to produce either a HIGH or a LOW level when the detection occurs. An active-HIGH decoder produces HIGH outputs to indicate detection. Figure 7-20 shows the complete active-HIGH decoding logic for a MOD-8 counter. The decoder consists of eight three-input AND gates. Each AND gate produces a HIGH output for one particular state of the counter.



**FIGURE 7-20** Using AND gates to decode a MOD-8 counter.

For example, AND gate 0 has at its inputs the FF outputs  $\bar{C}$ ,  $\bar{B}$ , and  $\bar{A}$ . Thus, its output will be LOW at all times *except* when  $A = B = C = 0$ , that is, on the count of 000 (zero). Similarly, AND gate 5 has as its inputs the FF outputs  $C$ ,  $\bar{B}$ , and  $A$ , so that its output will go HIGH only when  $C = 1$ ,  $B = 0$ , and  $A = 1$ , that is, on the count of 101 (decimal 5). The rest of the AND gates perform in the same manner for the other possible counts. At any one time, only one AND gate output is HIGH: the one that is decoding for the particular count present in the counter. The waveforms in Figure 7-20 show this clearly.

The eight AND outputs can be used to control eight separate indicator LEDs, which represent the decimal numbers 0 through 7. Only one LED will be ON at a given time, indicating the proper count.

The AND gate decoder can be extended to counters with any number of states. The following example illustrates.

#### EXAMPLE 7-14

How many AND gates are required to decode completely all of the states of a MOD-32 binary counter? What are the inputs to the gate that decodes for the count of 21?

#### Solution

A MOD-32 counter has 32 possible states. One AND gate is needed to decode for each state; therefore, the decoder requires 32 AND gates. Because  $32 = 2^5$ , the counter contains five FFs. Thus, each gate will have five inputs, one from each FF. Decoding for the count of 21 (that is,  $10101_2$ ) requires AND gate inputs of  $E$ ,  $\bar{D}$ ,  $C$ ,  $\bar{B}$ , and  $A$ , where  $E$  is the MSB flip-flop output.

### Active-LOW Decoding

If NAND gates are used in place of AND gates, the decoder outputs produce a normally HIGH signal, which goes LOW only when the number being decoded occurs. Both types of decoders are used, depending on the type of circuits being driven by the decoder outputs.

#### EXAMPLE 7-15

Figure 7-21 shows a common situation in which a counter is used to generate a control waveform, which could be used to control devices such as a motor, solenoid valve, or heater. The MOD-16 counter cycles and recycles through its counting sequence. Each time it goes to the count of 8 (1000), the upper NAND gate will produce a LOW output, which sets flip-flop X to the 1 state. Flip-flop X stays HIGH until the counter reaches the count of 14 (1110), at which time the lower NAND gate decodes it and produces a LOW output to clear X to the 0 state. Thus, the X output is HIGH between the counts of 8 and 14 for each cycle of the counter.

### BCD Counter Decoding

A BCD counter has 10 states that can be decoded using the techniques described previously. BCD decoders provide 10 outputs corresponding to the decimal digits 0 through 9 and represented by the states of the counter FFs. These 10 outputs can be used to control 10 individual indicator LEDs for a visual display. More often, instead of using 10 separate LEDs, a single display device is used to display the decimal numbers 0 through 9. One class of decimal displays contains seven small segments made of a material (usually LEDs or liquid-crystal displays) that either emits light or reflects ambient

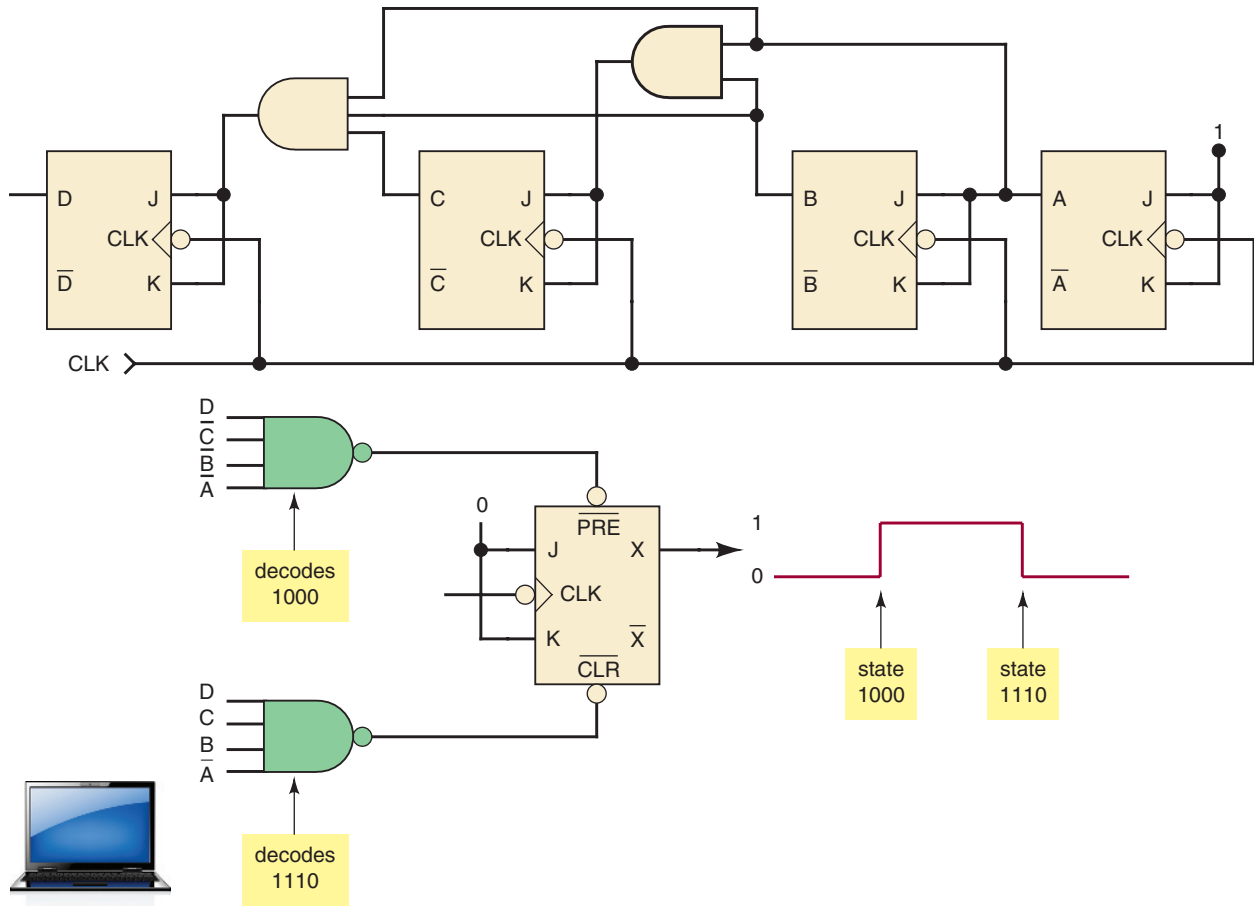
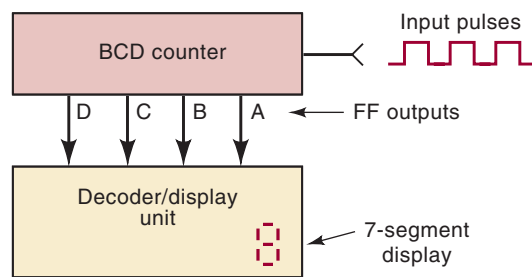


FIGURE 7-21 Example 7-15.

light. The BCD decoder outputs control which segments are illuminated in order to produce a pattern representing one of the decimal digits.

We will go into more detail concerning these types of decoders and displays in Chapter 9. However, because BCD counters and their associated decoders and displays are very commonplace, we will use the decoder/display unit (see Figure 7-22) to represent the complete circuitry used to display visually the contents of a BCD counter as a decimal digit.

FIGURE 7-22 BCD counters usually have their count displayed on a single display device.



**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. How many gates are needed to decode a six-bit counter fully?
2. Describe the decoding gate needed to produce a LOW output when a MOD-64 counter is at the count of 23.



## 7-9 ANALYZING SYNCHRONOUS COUNTERS

### OUTCOMES

Upon completion of this section, you will be able to:

- Create PRESENT/NEXT state tables for FF circuits.
- Use PRESENT/NEXT state tables to analyze an FF circuit.
- Draw a circuit timing diagram.
- Draw a circuit state transition diagram.

Synchronous counter circuits can be custom-designed to generate any desired count sequence. We can use just the synchronous inputs that are applied to the individual flip-flops to produce the counter's sequence. By not using asynchronous FF controls, such as the clears, to change the counter's sequence, we will never have to deal with transient states and possible glitches in output waveforms. The process of designing completely synchronous counters will be investigated in the next section. First, let's see how to analyze a counter design of this type by predicting the FF control inputs for each state of the counter. A **PRESENT state/NEXT state table** is a very useful tool in this analysis process. The first step is to write the logic expression for each FF control input. Next assume a PRESENT state for the counter and apply that combination of bits to the control logic expressions. The outputs from the control expressions will allow us to predict the commands to each FF and the resulting NEXT state for the counter after clocking. Repeat the analysis process until the entire count sequence is determined.

Figure 7-23 is a synchronous counter that has slightly different  $J$  and  $K$  inputs than we saw in Section 7-3 for a regular binary up counter. These minor changes to the control circuitry will cause the counter to produce a different count sequence. The control input expressions for this counter are:

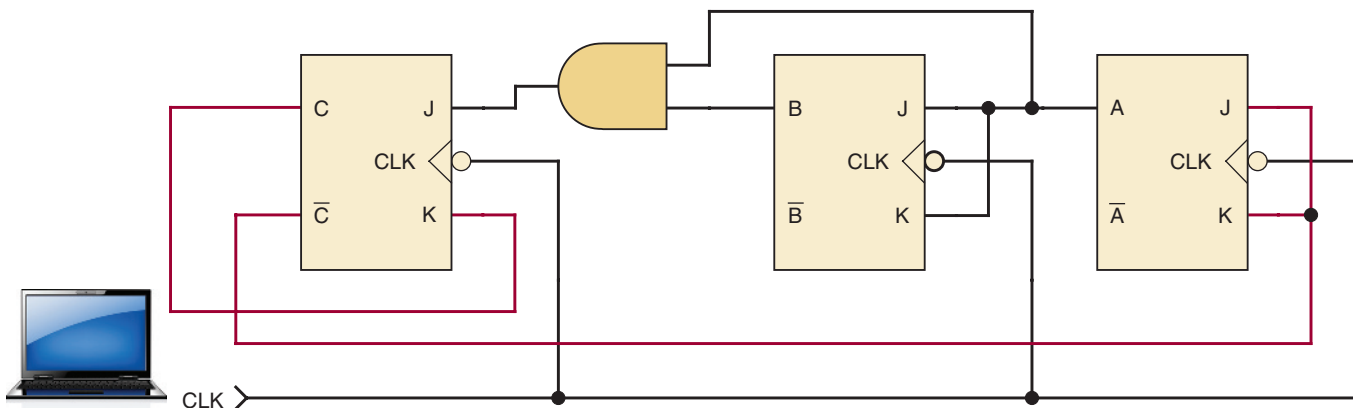
$$J_C = A \cdot B$$

$$K_C = C$$

$$J_B = K_B = A$$

$$J_A = K_A = \bar{C}$$

Let us assume that the PRESENT state for the counter is  $CBA = 000$ . Applying this combination to the control expressions above will yield  $J_C K_C = 00$ ,  $J_B K_B = 00$ , and  $J_A K_A = 11$ . These control inputs will tell FFs C and B to hold and FF A to toggle on the next NGT on  $CLK$ . Our predicted

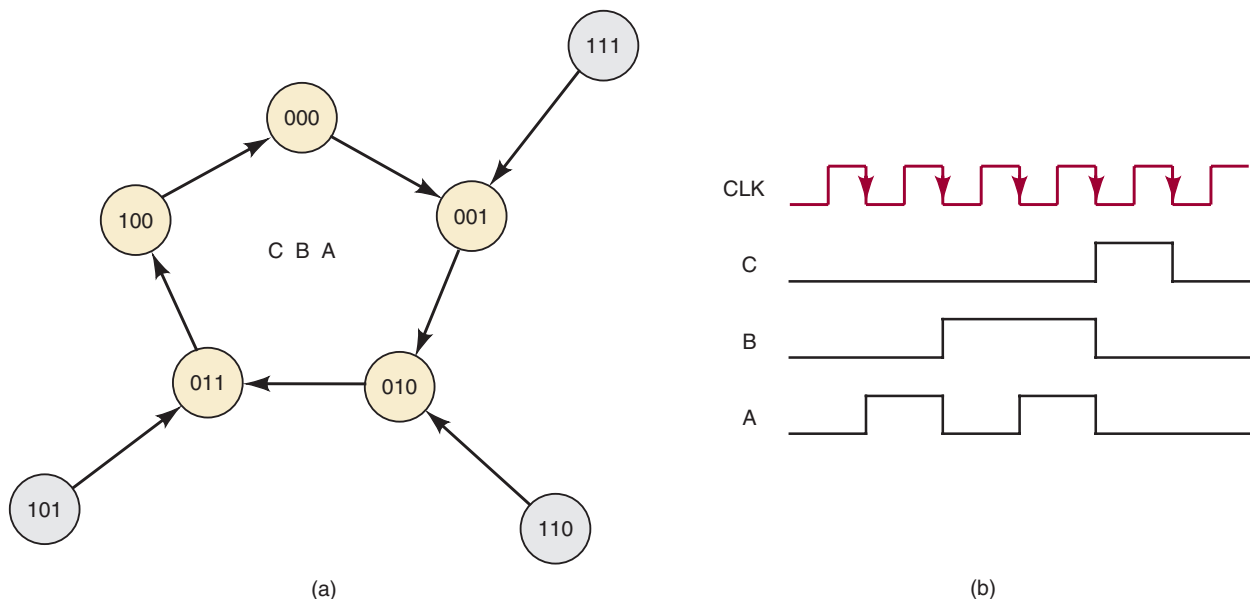


**FIGURE 7-23** Synchronous counter with different control inputs.

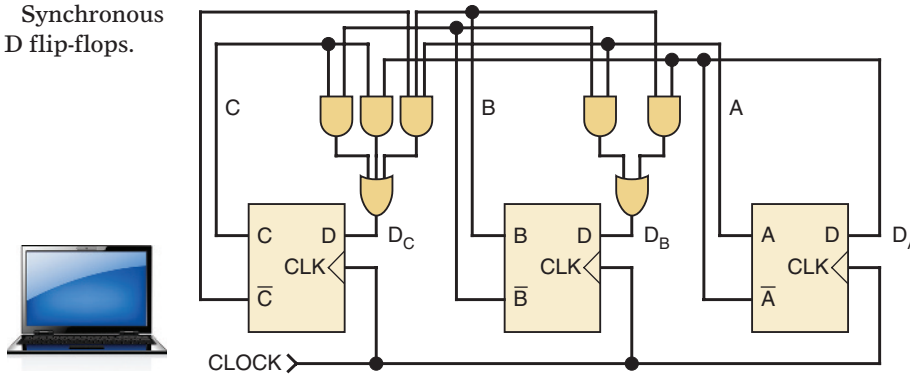
**TABLE 7-1** PRESENT state/NEXT state analysis table for Figure 7-23.

PRESENT State			Control Inputs						NEXT State		
C	B	A	$J_C$	$K_C$	$J_B$	$K_B$	$J_A$	$K_A$	C	B	A
0	0	0	0	0	0	0	1	1	0	0	1
0	0	1	0	0	1	1	1	1	0	1	0
0	1	0	0	0	0	0	1	1	0	1	1
0	1	1	1	0	1	1	1	1	1	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	1	0	1	1	1	0	0	0	1	1
1	1	0	0	1	0	0	0	0	0	1	0
1	1	1	1	1	1	1	0	0	0	0	1

NEXT state is 001 for  $CBA$ . This information has been entered in the first line of the PRESENT state/NEXT state table shown in Table 7-1. Next we can use the state 001 as our PRESENT state. Analyzing the control expressions with this new combination will now yield  $J_C K_C = 00$ ,  $J_B K_B = 11$ , and  $J_A K_A = 11$ , giving us a hold command for FF C and toggle commands for FFs B and A. This will produce a NEXT state of 010 for  $CBA$ , which we have listed on the second line of Table 7-1. Continuing with this process will result in a recycling count sequence of 000, 001, 010, 011, 100, 000. This would be a MOD-5 count sequence. We can also predict the NEXT states for the remaining three possible state combinations in the same way. By doing so, we can determine if the counter design is *self-correcting*. A **self-correcting counter** is one in which normally unused states will all somehow return to the normal count sequence. If any of these unused states cannot return to the normal sequence, the counter is said to be not self-correcting. Our NEXT-state predictions for all possible states have been entered into Table 7-1. The highlighted information indicates that this counter design happens to be self-correcting. The complete state transition diagram and timing diagram for this counter is shown in Figure 7-24.



**FIGURE 7-24** (a) State transition diagram and (b) timing diagram for synchronous counter in Figure 7-23.

**FIGURE 7-25** Synchronous counter using D flip-flops.

We can likewise analyze the operation of counter circuits that use D flip-flops to store the present state of the counter. The control circuitry for a D-type will typically be more complex than for an equivalent JK-type counter that produces the same count sequence, but we will also have half the number of synchronous inputs to control. Most PLDs utilize D flip-flops for their memory elements, so the analysis of this type of counter circuit will give us some insight into how counters are actually programmed inside a PLD.

A synchronous counter designed with D flip-flops is shown in Figure 7-25. The first step is to write the logic expressions for the  $D$  inputs:

$$\begin{aligned} D_C &= \overline{C\overline{B}} + \overline{C\overline{A}} + \overline{CBA} \\ D_B &= \overline{B}A + B\overline{A} \\ D_A &= \overline{A} \end{aligned}$$

Then we will determine the PRESENT state/NEXT state table for the counter circuit by assuming a state and applying that set of bit values to the input expressions given above. If we pick  $CBA = 000$  for the initial counter state, we will find that  $D_C = 0$ ,  $D_B = 0$ , and  $D_A = 1$ . With a PGT on CLOCK, the flip-flops will “load” in the value 001, which becomes the counter’s NEXT state. Using 001 as a PRESENT state will produce inputs of  $D_C = 0$ ,  $D_B = 1$ , and  $D_A = 0$  so that 010 will be the NEXT state, and so on. The completed PRESENT state/NEXT state table, shown in Table 7-2, indicates that this circuit is a recycling MOD-8 binary counter. By applying a little Boolean

**TABLE 7-2** Analysis table for Figure 7-25.

PRESENT State			Control Inputs			NEXT State		
C	B	A	$D_C$	$D_B$	$D_A$	C	B	A
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

algebra to the input expressions, we can see that there is actually a fairly simple circuit pattern in creating binary counters from D flip-flops:

$$\begin{aligned} D_C &= C\bar{B} + C\bar{A} + \bar{C}BA = C(\bar{B} + \bar{A}) + \bar{C}BA \\ &= C\bar{B}\bar{A} + \bar{C}(BA) = C \oplus (AB) \\ D_B &= \bar{B}A + B\bar{A} = B \oplus A \\ D_A &= \bar{A} \end{aligned}$$

It is important to note that the gating resources for most PLDs actually consist of sets of AND-OR circuit arrangements and the SOP logic expression more accurately describes the internal circuit implementation. However, we can see that the expressions have been greatly simplified by using the XOR function. This leads us to predict correctly that to create a MOD-16 binary counter with D flip-flops, we would need a fourth FF with:

$$D_D = D \oplus (ABC)$$

### OUTCOME ASSESSMENT QUESTIONS

1. Why is it desirable to avoid having asynchronous controls on counters?
2. What tool is useful in the analysis of synchronous counters?
3. What determines the count sequence for a counter circuit?
4. What counter characteristic is described by saying that it is self-correcting?

## 7-10 SYNCHRONOUS COUNTER DESIGN\*

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Follow a process to create counters that progress through any sequence of states.
- Create a PRESENT/NEXT/excitation table.
- Create a sequencer that will drive a stepper motor.

Many different counter arrangements are available as ICs—asynchronous, synchronous, and combined asynchronous/synchronous. Most of these count in a normal binary or BCD count sequence, although their counting sequences can be somewhat altered using the clearing or loading methods we demonstrated for the 74ALS160-163 and 74ALS190-191 series of ICs. There are situations, however, where a custom counter is required that follows a sequence that is not a regular binary count pattern, for example, 000, 010, 101, 001, 110, 000, . . .

Several methods exist for designing counters that follow arbitrary sequences. We will present the details for one common method that uses J-K flip-flops in a synchronous counter configuration. The same method can be used in designs with D flip-flops. The technique is one of several design procedures that are part of an area of digital circuit design called **sequential circuit design**, which is normally part of an advanced course.

### Basic Idea

In synchronous counters, all of the FFs are clocked at the same time. Before each clock pulse, the *J* and *K* input of each FF in the counter must be at the

\*This topic may be omitted without affecting the continuity of the remainder of the book.

correct level to ensure that the FF goes to the correct state. For example, consider the situation where state 101 for counter CBA is to be followed by state 011. When the next clock pulse occurs, the  $J$  and  $K$  inputs of the FFs must be at the correct levels that will cause flip-flop C to change from 1 to 0, flip-flop B from 0 to 1, and flip-flop A from 1 to 1 (i.e., no change).

The process of designing a synchronous counter thus becomes one of designing the logic circuits that *decode* the various states of the counter to supply the proper logic levels to each  $J$  and  $K$  input at the correct time. The inputs to these decoder circuits will come from the outputs of one or more of the FFs. To illustrate, for the synchronous counter of Figure 7-5, the AND gate that feeds the  $J$  and  $K$  inputs of flip-flop C decodes the states of flip-flops A and B. Likewise, the AND gate that feeds the  $J$  and  $K$  inputs of flip-flop D decodes the states of A, B, and C.

### J-K Excitation Table

Before we begin the process of designing the decoder circuits for each  $J$  and  $K$  input, we must first review the operation of the J-K flip-flop using a different approach, one called an *excitation table* (Table 7-3). The leftmost column of this table lists each possible FF output transition. The second and third columns list the FF's PRESENT state, symbolized as  $Q_n$ , and the NEXT state, symbolized as  $Q_{n+1}$ , for each transition. The last two columns list the  $J$  and  $K$  levels required to produce each transition. Let's examine each case.

**TABLE 7-3** J-K flip-flop excitation table.

Transition at FF Output	PRESENT State $Q_n$	NEXT State $Q_{n+1}$	$J$	$K$
0 → 0	0	0	0	$x$
0 → 1	0	1	1	$x$
1 → 0	1	0	$x$	1
1 → 1	1	1	$x$	0

**0 → 0 TRANSITION** The FF PRESENT state is at 0 and is to remain at 0 when a clock pulse is applied. From our understanding of how a J-K flip-flop works, this can happen when either  $J = K = 0$  (no-change condition) or  $J = 0$  and  $K = 1$  (clear condition). Thus,  $J$  must be at 0, but  $K$  can be at either level. The table indicates this with a “0” under  $J$  and an “ $x$ ” under  $K$ . Recall that “ $x$ ” means the don't-care condition.

**0 → 1 TRANSITION** The PRESENT state is 0 and is to change to a 1, which can happen when either  $J = 1$  and  $K = 0$  (set condition) or  $J = K = 1$  (toggle condition). Thus,  $J$  must be a 1, but  $K$  can be at either level for this transition to occur.

**1 → 0 TRANSITION** The PRESENT state is 1 and is to change to a 0, which can happen when either  $J = 0$  and  $K = 1$  or  $J = K = 1$ . Thus,  $K$  must be a 1, but  $J$  can be at either level.

**1 → 1 TRANSITION** The PRESENT state is a 1 and is to remain a 1, which can happen when either  $J = K = 0$  or  $J = 1$  and  $K = 0$ . Thus,  $K$  must be a 0 while  $J$  can be at either level.

The use of this **J-K excitation table** (Table 7-3) is a principal part of the synchronous counter design procedure.

**TABLE 7-4** The sequence of desired states.

C	B	A
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
0	0	0
0	0	1
etc.		

## Design Procedure

We will now go through a complete synchronous counter design procedure. Although we will do it for a specific counting sequence, the same steps can be followed for any desired sequence.

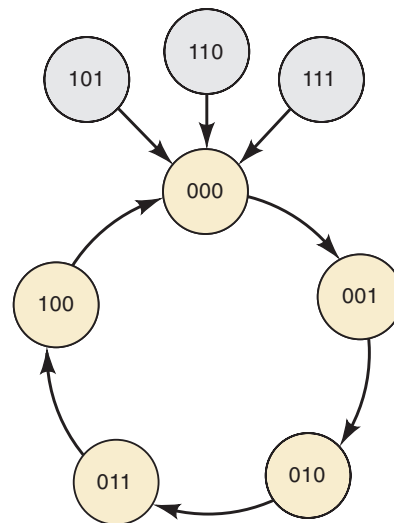
**Step 1.** Determine the desired number of bits (FFs) and the desired counting sequence.

For our example, we will design a three-bit counter that goes through the sequence shown in Table 7-4. Notice that this sequence does not include the 101, 110, and 111 states. We will refer to these states as *undesired states*.

**Step 2.** Draw the state transition diagram showing *all* possible states, including those that are not part of the desired counting sequence.

For our example, the state transition diagram appears as shown in Figure 7-26. The 000 through 100 states are connected in the expected sequence. We have also included a defined NEXT state for each of the undesired states. This was done in case the counter accidentally gets into one of these states upon power-up or due to noise. The circuit designer can choose to have each of these undesired states go to any state upon the application of the next clock pulse. Alternatively, the designer may choose not to define the counter's action for the undesired states at all. In other words, we may not care about the NEXT state for any undesired state. Using the latter "don't care" design approach will generally produce a simpler design but can be a potential problem in the application where this counter is to be used. For our design example, we will choose to have all undesired states go to the 000 state. This will make our design self-correcting but slightly different from the example MOD-5 counter that was analyzed in Section 7-9.

**FIGURE 7-26** State transition diagram for the synchronous counter design example.



**Step 3.** Use the state transition diagram to set up a table that lists *all* PRESENT states and their NEXT states.

For our example, the information is shown in Table 7-5. The left-hand portion of the table lists *every* possible state, even those that are not part of the sequence. We label these as the PRESENT states. The right-hand portion lists the NEXT state for each PRESENT state. These are obtained from the state transition diagram in Figure 7-26. For instance, line 1 shows that the PRESENT state of 000 has the NEXT state of 001, and line 5 shows that the PRESENT state of 100 has the NEXT state of 000. Lines 6, 7, and

8 show that the undesired PRESENT states 101, 110, and 111 all have the NEXT state of 000.

**TABLE 7-5** Complete PRESENT state/NEXT state table.

	PRESENT State			NEXT State		
	C	B	A	C	B	A
Line 1	0	0	0	0	0	1
2	0	0	1	0	1	0
3	0	1	0	0	1	1
4	0	1	1	1	0	0
5	1	0	0	0	0	0
6	1	0	1	0	0	0
7	1	1	0	0	0	0
8	1	1	1	0	0	0

**Step 4.** Add a column to this table for each  $J$  and  $K$  input. For each PRESENT state, indicate the levels required at each  $J$  and  $K$  input in order to produce the transition to the NEXT state.

Our design example uses three FFs—C, B, and A—and each one has a  $J$  and a  $K$  input. Therefore, we must add six new columns as shown in Table 7-6. This completed table is called the **circuit excitation table**. The six new columns are the  $J$  and  $K$  inputs of each FF. The entries under each  $J$  and  $K$  are obtained from Table 7-3, the J-K flip-flop excitation table that we developed earlier. We will demonstrate this for several of the cases, and you can verify the rest.

**TABLE 7-6** Circuit excitation table.

	PRESENT State			NEXT State			$J_C$	$K_C$	$J_B$	$K_B$	$J_A$	$K_A$
	C	B	A	C	B	A						
Line 1	0	0	0	0	0	1	0	x	0	x	1	x
2	0	0	1	0	1	0	0	x	1	x	x	1
3	0	1	0	0	1	1	0	x	x	0	1	x
4	0	1	1	1	0	0	1	x	x	1	x	1
5	1	0	0	0	0	0	x	1	0	x	0	x
6	1	0	1	0	0	0	x	1	0	x	x	1
7	1	1	0	0	0	0	x	1	x	1	0	x
8	1	1	1	0	0	0	x	1	x	1	x	1

Let's look at line 1 in Table 7-6. The PRESENT state of 000 is to go to the NEXT state of 001 on the occurrence of a clock pulse. For this state transition, the C flip-flop goes from 0 to 0. From the J-K excitation table, we see that  $J_C$  must be at 0 and  $K_C$  at "x" for this transition to occur. The B flip-flop also goes from 0 to 0, and so  $J_B = 0$  and  $K_B = x$ . The A flip-flop goes from 0 to 1. Also from Table 7-3, we see that  $J_A = 1$  and  $K_A = x$  for this transition.

In line 4 in Table 7-6, the PRESENT state of 011 has a NEXT state of 100. For this state transition, flip-flop C goes from 0 to 1, which requires  $J_C = 1$  and  $K_C = x$ . Flip-flops B and A are both going from 1 to 0. The J-K excitation table indicates that these two FFs need  $J = x$  and  $K = 1$  for this to occur.

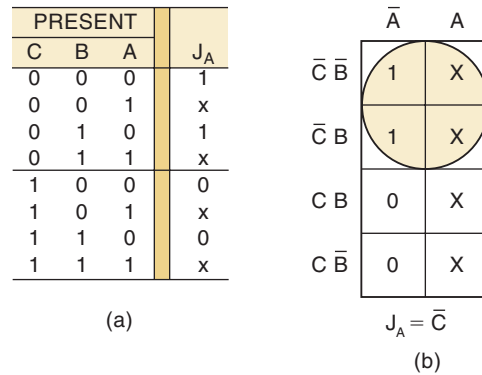
The required  $J$  and  $K$  levels for all other lines in Table 7-6 can be determined in the same manner.

**Step 5.** Design the logic circuits needed to generate the levels required at each  $J$  and  $K$  input.

Table 7-6, the circuit excitation table, lists six  $J, K$  inputs— $J_C, K_C, J_B, K_B, J_A,$  and  $K_A$ . We must consider each of these as an output from its own logic circuit with inputs from flip-flops C, B, and A. Then we must design the circuit for each one. Let's design the circuit for  $J_A$ .

To do this, we need to look at the PRESENT states of C, B, and A and the desired levels at  $J_A$  for each case. This information has been extracted from Table 7-6 and presented in Figure 7-27(a). This truth table shows the desired levels at  $J_A$  for each PRESENT state. Of course, for some of the cases,  $J_A$  is a don't-care. To develop the logic circuit for  $J_A$ , we must first determine its expression in terms of C, B, and A. We will do this by transferring the truth-table information to a three-variable Karnaugh map and performing the K-map simplification, as in Figure 7-27(b).

**FIGURE 7-27** (a) Portion of circuit excitation table showing  $J_A$  for each PRESENT state; (b) K map used to obtain the simplified expression for  $J_A$ .



There are only two 1s in this K map, and they can be looped to obtain the term  $\bar{A}\bar{C}$ , but if we use the don't-care conditions at  $A\bar{B}\bar{C}$  and  $ABC\bar{C}$  as 1s, we can loop a quad to obtain the simpler term  $\bar{C}$ . Thus, the final expression is

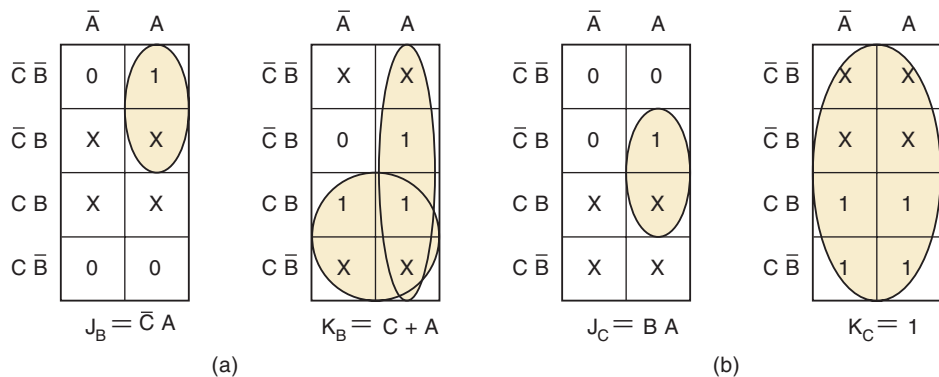
$$J_A = \bar{C}$$

Now let's consider  $K_A$ . We can follow the same steps as we did for  $J_A$ . However, a look at the entries under  $K_A$  in the circuit excitation table shows only 1s and don't-cares. If we change all the don't-cares to 1s, then  $K_A$  is always a 1. Thus, the final expression is

$$K_A = 1$$

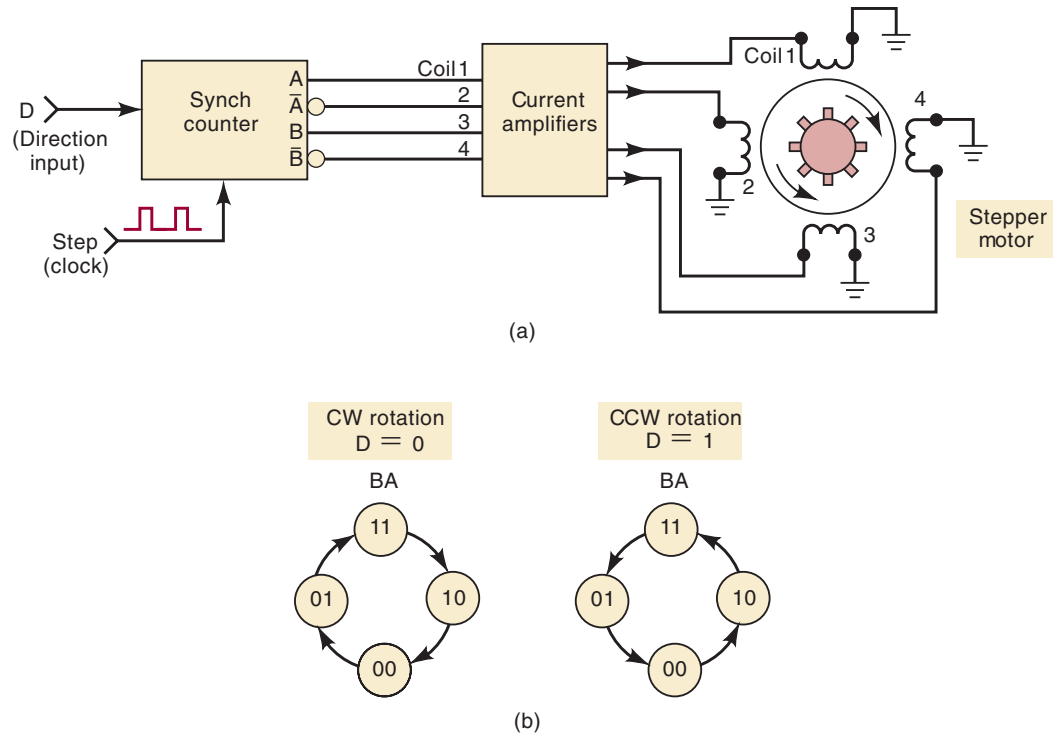
In a similar manner, we can derive the expressions for  $J_C, K_C, J_B,$  and  $K_B$ . The K maps for these expressions are given in Figure 7-28. You might want to confirm their correctness by checking them against the circuit excitation table.

**FIGURE 7-28** (a) K maps for the  $J_B$  and  $K_B$  logic circuits; (b) K maps for the  $J_C$  and  $K_C$  logic circuits.









**FIGURE 7-30** (a) A synchronous counter supplies the appropriate sequential outputs to drive a stepper motor; (b) state transition diagrams for both states of Direction input, *D*.

input, *D*, does not change in going from the PRESENT to the NEXT state because it is an independent input that is held HIGH or LOW as the counter goes through its sequence.

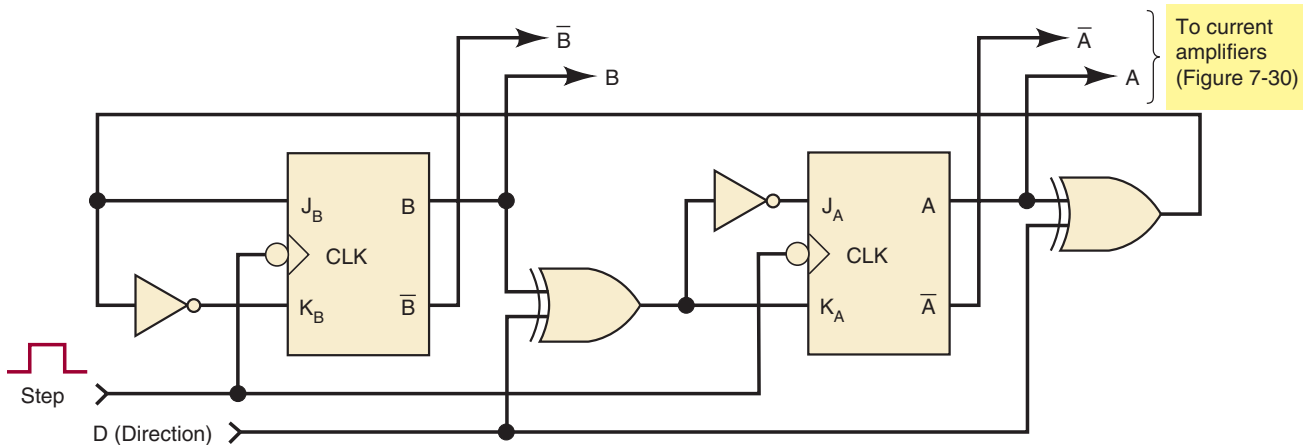
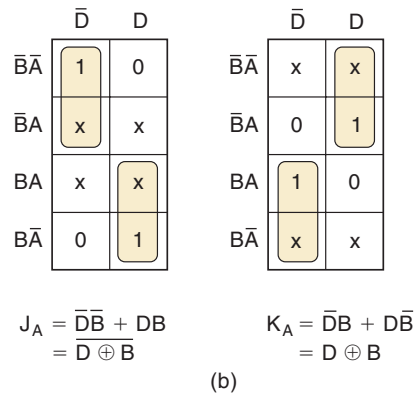
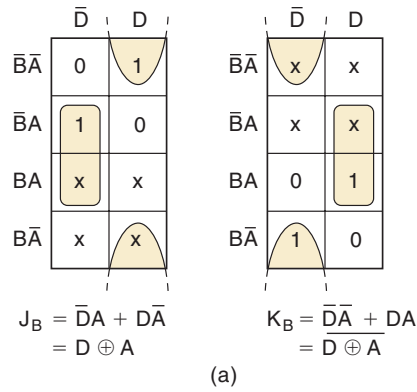
**TABLE 7-7** Excitation table for stepper driver.

PRESENT State			NEXT State		Control Inputs			
D	B	A	B	A	J <sub>B</sub>	K <sub>B</sub>	J <sub>A</sub>	K <sub>A</sub>
0	0	0	0	1	0	x	1	x
0	0	1	1	1	1	x	x	0
0	1	0	0	0	x	1	0	x
0	1	1	1	0	x	0	x	1
1	0	0	1	0	1	x	0	x
1	0	1	0	0	0	x	x	1
1	1	0	1	1	x	0	1	x
1	1	1	0	1	x	1	x	0

Step 5 of the design process is presented in Figure 7-31, where the information in Table 7-7 has been transferred to the K maps showing how each *J* and *K* signal is related to the PRESENT states of *D*, *B*, and *A*. Using the appropriate looping, the simplified logic expressions for each *J* and *K* signal are obtained.

The final step is shown in Figure 7-32, where the two-bit synchronous counter is implemented using the *J*, *K* expressions obtained from the K maps.

**FIGURE 7-31** (a) K maps for  $J_B$  and  $K_B$ ; (b) K maps for  $J_A$  and  $K_A$ .



**FIGURE 7-32** Synchronous counter implemented from the  $J, K$  equations.

### Synchronous Counter Design with D FF

We have provided a detailed procedure for designing synchronous counters using J-K flip-flops. Historically, J-K flip-flops have been used to implement counters because the logic circuits needed for the  $J$  and  $K$  inputs are usually simpler than the logic circuits needed to control an equivalent synchronous counter using D flip-flops. When designing counters that will be implemented in PLDs, where abundant gates are generally available, it makes sense to use D flip-flops instead of J-Ks. Let us now look at synchronous counter design using D FFs.

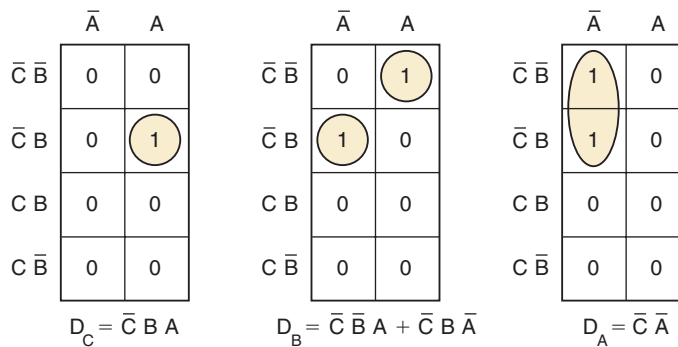
Designing counter circuits using D flip-flops is even easier than using J-K flip-flops. We will demonstrate by designing a D FF circuit that produces the same count sequence as is given in Figure 7-26. The first three steps for

synchronous D counter design are identical to the J-K technique. Step 4 for D FF design is trivial since the necessary D inputs are the same as the desired NEXT state as seen in Table 7-8. Step 5 is to generate the logic expressions from the PRESENT state/NEXT state table for the D inputs. The K maps and simplified expressions are given in Figure 7-33. Finally, for step 6, the counter can be implemented with the circuit shown in Figure 7-34.

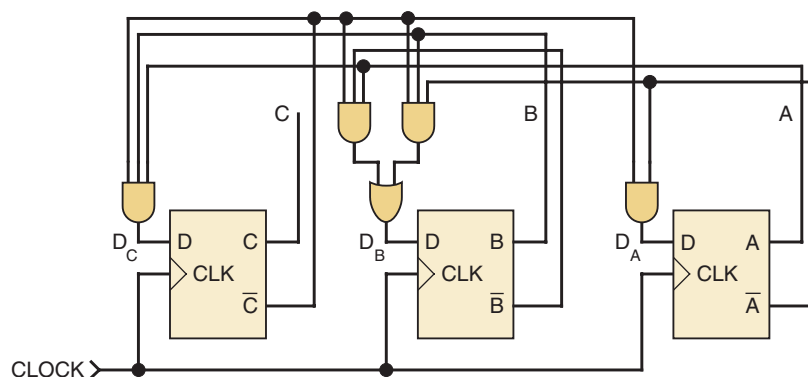
**TABLE 7-8** Excitation table using D Flip-Flops.

PRESENT State			NEXT State			Control Inputs		
C	B	A	C	B	A	D <sub>C</sub>	D <sub>B</sub>	D <sub>A</sub>
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0

**FIGURE 7-33** K maps and simplified logic expressions for MOD-5 flip-flop counter design.



**FIGURE 7-34** Circuit implementation of MOD-5 D flip-flop counter design.



### OUTCOME ASSESSMENT QUESTIONS

- List the six steps in the procedure for designing a synchronous counter.
- What information is contained in a PRESENT state/NEXT state table?
- What information is contained in the circuit excitation table?
- True or false:* The synchronous counter design procedure can be used for the following sequence: 0010, 0011, 0100, 0111, 1010, 1110, 1111, and repeat.

## 7-11 ALTERA LIBRARY FUNCTIONS FOR COUNTERS

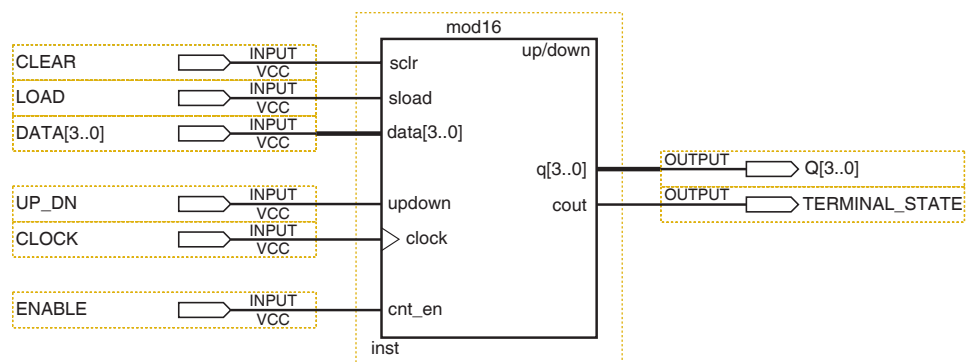
### OUTCOMES

Upon completion of this section, you will be able to:

- Locate Altera’s library of parameterized modules (LPMs).
- Specify parameters common to counters.
- Create modules to meet system needs.

We can use the Quartus Block Editor to program a PLD with any counter using flip-flops and gates such as those illustrated in earlier sections of this chapter. As we have seen in Chapters 5 and 6, Altera’s Quartus II software contains libraries of common digital building blocks. This would include functionally equivalent representations of “old-style” MSI counter chips such as the 74160-74163 and 74190-74191 series of MSI devices. These macrofunctions can be found in the maxplus2 library. This makes it very easy to create schematics like those in Figure 7-18(a) or 7-19. An even more versatile counter option is available with the megafunction **LPM\_COUNTER** (found in the Plug-Ins Arithmetic folder). The MegaWizard Manager makes designing counters quick and easy. All you need to do is select the desired features, number of bits, and modulus. A full-featured (but it does not use all available options) MOD-16, up/down counter is shown in Figure 7-35. The counter has an active-HIGH count enable control and will count up when  $UP\_DN = 1$  or down if  $UP\_DN = 0$ . The counter also can be synchronously cleared and parallel loaded with new data that is input on the port labeled  $DATA[3..0]$ . The carry-out (*cout*) will decode the counter’s terminal state of 15 when counting up or 0 when counting down. All of these features are automatically created by just telling the Wizard what is needed.

**FIGURE 7-35** Full-featured MOD-16 counter.



### EXAMPLE 7-16

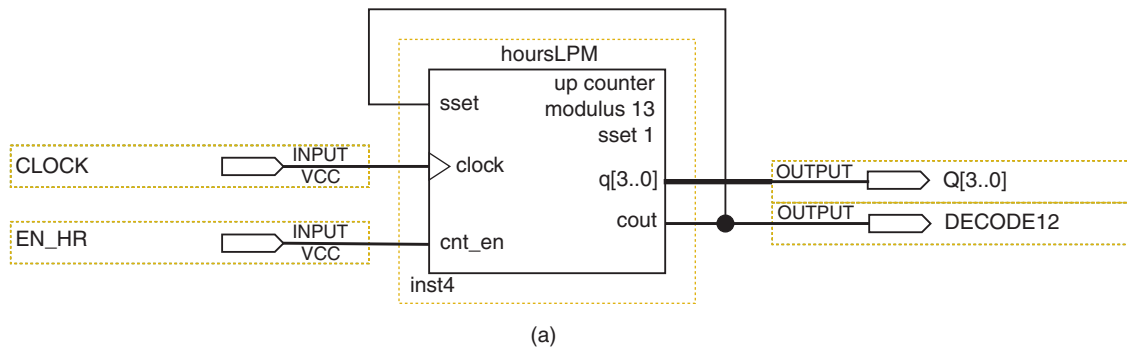
Design the hours and minutes counters for a digital clock. Use a binary counter for the hours and cascaded BCD counters for the minutes. Since the minutes block and the seconds block of a digital clock will each require MOD-60 counters, we will be able to use the same design for both sections of the clock. Provide enable inputs for each counter block so that they can be cascaded together synchronously.

### Solution

The hours counter LPM design is shown in Figure 7-36(a). The  $EN\_HR$  enable input will be controlled by the minutes counter block. When the

minutes counter reaches its terminal state of 59, the hours counter should be enabled and then both the hours counter and the minutes counter will be clocked simultaneously. The MegaWizard settings for the hours counter block are given in Figure 7-36(b). The desired recycling binary count sequence for the hours counter will be 1 to 12, so a MOD-12 counter will be needed. However, the modulus that was input to the Wizard is 13 since the LPM counter will automatically recycle to 0; we want the counter to count up to 12, which will be 13 states instead of 12. The counter is then forced to recycle back to 1 instead of 0 by controlling the *sset* input (and specifying a data value of 1) with the terminal state decoding output *DECODE12*. This decoder output can also potentially control an AM/PM flip-flop even though that was not specified for this project. Simulation results (note that an arbitrary time scale was used) for this design are given in Figure 7-36(c).

The minutes counter (Figure 7-37) is designed to provide a BCD count sequence for the MOD-60 counter by subdividing it into two LPM counter blocks.



**LPM parameter decisions:**

How wide should the 'q' output bus be? **4**

What should the counter direction be? Select UP only

Which type of counter do you want? Select Modulus with a count modulus of **13**

Do you want any optional additional ports? Select Count Enable and Carry-out

Do you want any optional inputs?

Synchronous inputs: Select Set => Set to **1** Asynchronous: none

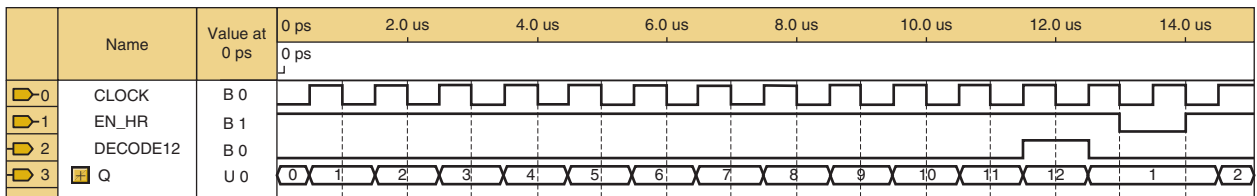
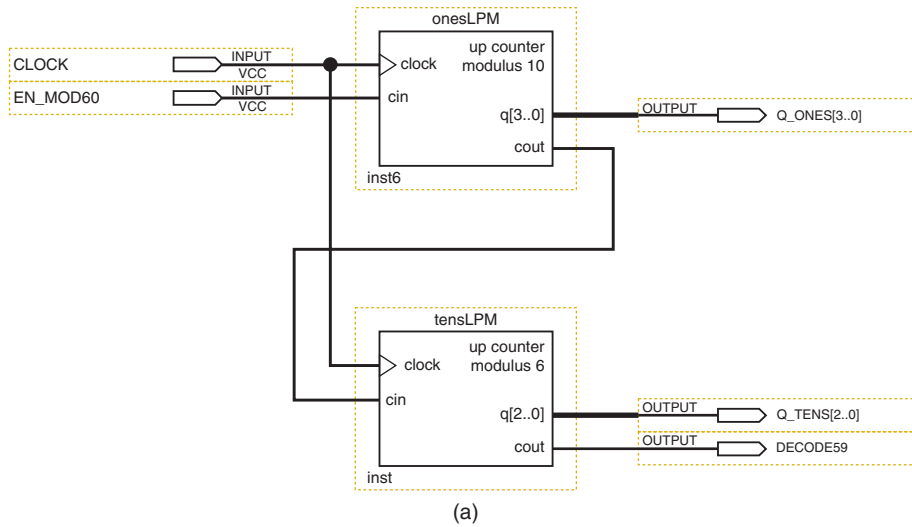


FIGURE 7-36 Digital clock hours counter: (a) block diagram; (b) MegaWizard settings; (c) simulation results.



**LPM parameter decisions:**

**Ones\_LPM**

Which type of counter do you want? Select Modulus with a count modulus of 10

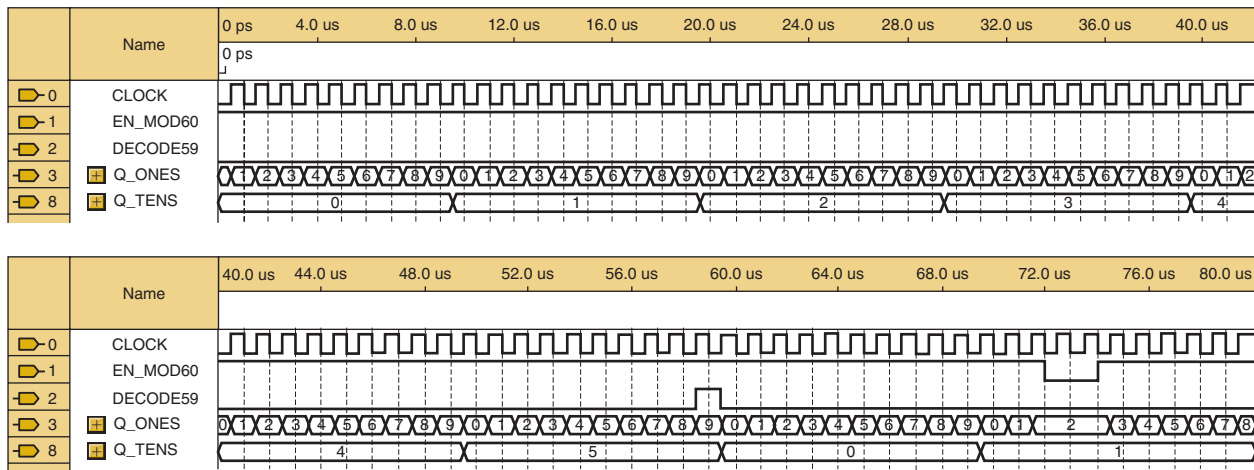
Do you want any optional additional ports? Select Carry-in and Carry-out

**Tens\_LPM**

Which type of counter do you want? Select Modulus with a count modulus of 6

Do you want any optional additional ports? Select Carry-in and Carry-out

(b)



(c)

**FIGURE 7-37** Digital clock minutes counter: (a) block diagram; (b) MegaWizard settings for onesLPM and tensLPM; (c) simulation results.

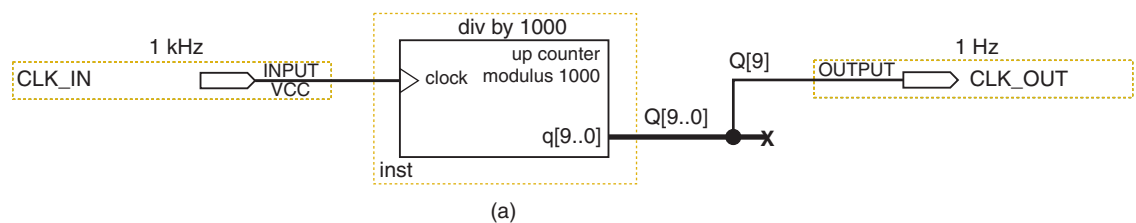
This will make it more convenient to interface to a digital display (see Chapter 9 for display circuit details). The *onesLPM* block is a MOD-10 counter that will output the least significant digit (LSD) for the minutes counter. The *tensLPM* block produces the most significant digit (MSD) with a MOD-6 count sequence. The carry-in (*cin*) enable has been specifically selected for these counters because it also enables the carry-out (*cout*) decoding of each counter's terminal state. This will make it very easy to cascade the two sub-blocks synchronously together. This is done by connecting *cout* from *onesLPM* to the *cin* on the *tensLPM*. By cascading the two counter blocks together in this fashion, the enable input *EN\_MOD60* will be able to control the entire minutes counter. This technique also allows the output port *DECODE59* to detect when *tensLPM* is at its terminal state of 5 AND *onesLPM* is at its terminal state of 9, or in other words, state 59 for the entire minutes counter. Functional simulation results (again an arbitrary time scale was used) for this design are given in Figure 7-37(c).

### EXAMPLE 7-17

Design the frequency divider circuit to obtain the correct clocking frequency to drive the MOD-60 seconds counter of a digital clock. The system clock frequency is 1 kHz.

#### Solution

The clock frequency for the seconds counter should be 1 Hz. Therefore, we will need to divide the 1-kHz signal by 1000 to produce the proper frequency. An LPM\_COUNTER block with a modulus of 1000 was created (see Figure 7-38). The MSB output will provide a frequency division factor equal to the counter's modulus. The frequency at  $Q[9]$  will be equal to the input frequency divided by 1000. The only output signal needed is  $Q[9]$ , so the output bus was split out to obtain just the one signal. Whenever a bus is split, the bus and signal splits must be labeled. We applied the specified



#### LPM parameter decisions:

How wide should the 'q' output bus be? 10

What should the counter direction be? Select UP only

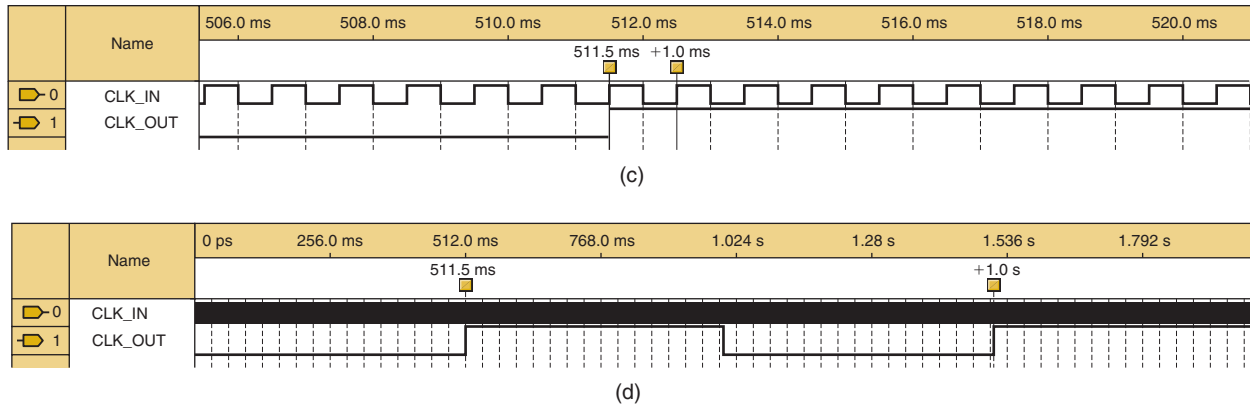
Which type of counter do you want? Select Modulus with a count modulus of 1000

Do you want any optional additional port? None

(b)

**FIGURE 7-38** Clock frequency divider: (a) block diagram; (b) MegaWizard settings.





**FIGURE 7-38** (Continued) Clock frequency divider: (c) measuring simulated input period with time bars; (d) measuring output period for simulation results.

clock frequency (1 kHz) for our simulation. The functional simulation results show that we will have the correct period of 1 second for our output signal if a 1-kHz clock is applied to the counter. Figure 7-38(c) is a zoom-in on the clock signal to show its period, which has been measured in Quartus using two time bars (note that the time mark is +1.0 ms after the first time bar). The period for the output signal is measured in Figure 7-38(d).

### OUTCOME ASSESSMENT QUESTIONS

1. Which Altera megafunction library folder contains LPM\_COUNTER?
2. How do you define the features and modulus for an LPM\_COUNTER?
3. Explain the difference between an asynchronous clear and a synchronous clear for a counter.
4. What is the function of *cout* for an LPM\_COUNTER?
5. The counting of an LPM\_COUNTER can be enabled or disabled using either *cnt\_en* or *cin*. What is the difference between these two controls?

## 7-12 HDL COUNTERS

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe counter operations using HDL.
- Use registers in AHDL.
- Use processes in VHDL.
- Describe synchronous and asynchronous operations using HDL.
- Determine precedence/priority of controls using HDL.

In Chapter 5, we studied flip-flops and the methods used with HDLs to represent flip-flop circuits. The last section in Chapter 5 illustrated how to connect FF components very much like you would wire integrated circuits to one another. By connecting the *Q* output of one FF to the clock input of the next FF, we found that a counter circuit can be created. Using an HDL to describe component connections is referred to as the structural level of

abstraction. It is obvious that constructing a complicated circuit using the structural methods would be very tedious and also very difficult to read and interpret. In this section, we will broaden our use of HDL to describe circuits using methods that are considered higher levels of abstraction. This term sounds intimidating, but it only means that there are much more concise and sensible ways to describe what we want a counter to do without worrying about all the details of how to wire flip-flop circuits to do it.

It is still vital that we understand the fundamental principles of flip-flop operation compared to combinational logic gates. As you recall, flip-flops have the following unique characteristics. The output is normally updated according to the condition of the synchronous control inputs when the *active edge* of the clock occurs, which means there is a logic state on the  $Q$  output before the clock edge (PRESENT state) and potentially a different state on the  $Q$  output after the clock edge (NEXT state). A flip-flop “remembers,” or holds its state between clocks, regardless of changes in the synchronous control inputs (e.g.,  $J$  and  $K$ ).

Counter circuits using HDL rely on this basic understanding of a circuit going through a sequence of states in response to the event of a clock edge. Ripple counters provide an easy circuit to analyze and understand. They are also much less complicated to build using flip-flops and logic gates than their synchronous counterparts. The problem with ripple counters is the combination of time delay and spurious temporary states that occur when the counter changes state. When we advance to the next level of abstraction and plan to use PLDs to implement our design, we are no longer focusing on wiring issues but rather on describing the circuit’s operation concisely. Consequently, the methods we use to describe counter circuits using HDL primarily use synchronous techniques, where all flip-flops update simultaneously in response to the same clock event. All the bits in a count sequence go from their PRESENT state to their prescribed NEXT state simultaneously, thereby preventing any intermediate, spurious states.

## State Transition Description Methods

The next method of describing circuits that we need to examine uses tables. This method is not concerned with connecting ports of components but rather with assigning values to objects like ports, signals, and variables. In other words, it describes how the output data relates to the input data throughout the circuit. We have already used this method in several of the introductory circuits in Chapters 3 and 4, in the form of truth tables. With sequential counter circuits, the equivalent of the truth table is the PRESENT state/NEXT state table, as we saw in the previous section. We can use the HDL essentially to describe the PRESENT state/NEXT state table and thus avoid the tedious details of generating the Boolean equations, as we did in Section 7-10 to design with standard logic devices.

---

## STATE DESCRIPTIONS IN AHDL

As an example of a simple counter circuit, we will implement the MOD-5 counter of Figure 7-26 in AHDL. The inputs and outputs are defined in the SUBDESIGN section of Figure 7-39, as always. In the VARIABLE section on line 7, we have declared (or instantiated) a three-bit array of DFF primitives that are given the instance name *count[ ]*. This array will be treated basically as a three-bit register in the design and we will essentially define what value should be stored for each NEXT state. Because this is a synchronous

```

1  SUBDESIGN fig7_39
2  (
3      clock          :INPUT;
4      qout[2..0]    :OUTPUT;
5  )
6  VARIABLE
7      count[2..0]   :DFF;          --create a 3-bit register
8  BEGIN
9      count[].clk = clock;        --connect all clocks in parallel
10
11         CASE count[].q IS
12 --             Present                Next
13 -----
14             WHEN 0    =>    count[].d = 1;
15             WHEN 1    =>    count[].d = 2;
16             WHEN 2    =>    count[].d = 3;
17             WHEN 3    =>    count[].d = 4;
18             WHEN 4    =>    count[].d = 0;
19             WHEN OTHERS =>    count[].d = 0;
20         END CASE;
21     qout[] = count[].q;          --assign register to output pins
22 END;

```

**FIGURE 7-39** AHDL MOD-5 counter.

counter, we need to tie all the DFF *clk* inputs to the SUBDESIGN's *clock* input. This is accomplished in AHDL by the following statement in the logic section:

```
count[ ].clk = clock;
```

The flip-flop primitives provided in AHDL have standard inputs and outputs that are referred to as “ports.” These ports are labeled by a standard port name that is attached to the instance name of the flip-flops. As seen in Table 5-3, the clock port name is *.clk*, a *D* input is named *.d*, and the FF's output has the name *.q*. To implement the PRESENT state/NEXT state table, a CASE construct is used. For each of the possible values of the register *count[]*, we determine the value that should be placed on the *D* inputs of the flip-flops, which will determine the NEXT state of the counter. The statement on line 21 assigns the value on *count[]* to the output pins. Without this line, the counter would be “buried” in the SUBDESIGN and would not be visible to the outside world.

An alternative design solution is given in Figure 7-40. There are two modifications from Figure 7-39. The first is seen on line 7, where the array name for the D flip-flops is now the same as the output port for the SUBDESIGN. This will automatically connect the flip-flop outputs to the SUBDESIGN outputs and eliminate the need to include an assignment statement like line 21 in the first solution. The second modification is the use of an AHDL TABLE instead of the CASE statement used in Figure 7-39. In line 11, the *.q* port on the *q[]* DFF array represents the PRESENT state side of the table, while the *.d* port for *q[]* represents the NEXT state that will be entered into the array's set of *D* inputs when a PGT is applied to *clock*.

```

1  SUBDESIGN fig7_40
2  (
3      clock      :INPUT;
4      q[2..0]    :OUTPUT;
5  )
6  VARIABLE
7      q[2..0]    :DFF;    -- create a 3-bit register
8  BEGIN
9      q[].clk = clock;    -- connect all clocks in parallel
10     TABLE
11         q[].q => q[].d;
12         0     => 1;
13         1     => 2;
14         2     => 3;
15         3     => 4;
16         4     => 0;
17         5     => 0;
18         6     => 0;
19         7     => 0;
20     END TABLE;
21 END;
```

**FIGURE 7-40** Another version of the MOD-5 counter described in Figure 7-26.

## STATE DESCRIPTIONS IN VHDL

As an example of a simple counter circuit, we will implement the MOD-5 counter of Figure 7-26 in VHDL. Our purpose in this example is to demonstrate a counter using a control structure similar to a PRESENT state/NEXT state table. Two key tasks must be accomplished in VHDL: detecting the desired clock edge, and assigning the proper NEXT state to the counter. Recall from our study of flip-flops that a PROCESS can be used to respond to a transition of an input signal. Also, we have learned that a CASE construct can evaluate an expression and, for any valid input value, assign a corresponding value to another signal. The code in Figure 7-41 uses a PROCESS and a CASE construct to implement this counter. The inputs and outputs are defined in the ENTITY declaration, as in the past.

When VHDL is used to describe a counter, we must find a way to “store” the state of the counter between clock pulses (i.e., the action of a flip-flop). This is done in one of two ways: using SIGNALs, or using VARIABLEs. We have used SIGNALs extensively in previous examples that operated concurrently. A SIGNAL in VHDL holds the last value that was assigned to it, very much like a flip-flop. Consequently, we can use a SIGNAL as the data object representing the counter value. This SIGNAL can then be used to connect the counter value to any other elements in the architecture description.

In this design, we have chosen to use a **VARIABLE** instead of a SIGNAL as the data object that stores the counter value. VARIABLEs are not exactly like SIGNALs because they are not used to connect various parts of the design. Instead, they are used as a local place to “store” a value. Variables are considered to be local data objects because they are recognized only within the PROCESS in which they are declared. On line 11 of Figure 7-41, the variable named *count* is declared within the PROCESS before BEGIN. Its type is the same as the output port *q*. The keyword PROCESS on line 10

```

1  ENTITY fig7_41 IS
2  PORT (
3      clock    :IN BIT;
4      q        :OUT BIT_VECTOR(2 DOWNTO 0)
5  );
6  END fig7_41 ;
7
8  ARCHITECTURE a OF fig7_41    IS
9  BEGIN
10     PROCESS (clock)          -- respond to clk input
11     VARIABLE count: BIT_VECTOR(2 DOWNTO 0);  -- create a 3-bit register
12     BEGIN
13         IF (clock = '1' AND clock'EVENT) THEN  -- rising edge trigger
14             CASE count IS
15 --             Present                Next
16 -----
17                 WHEN "000" =>    count := "001";
18                 WHEN "001" =>    count := "010";
19                 WHEN "010" =>    count := "011";
20                 WHEN "011" =>    count := "100";
21                 WHEN "100" =>    count := "000";
22                 WHEN OTHERS =>    count := "000";
23             END CASE;
24         END IF;
25         q <= count;          -- assign register to output pins
26     END PROCESS;
27 END a;

```

**FIGURE 7-41** VHDL MOD-5 counter.

is followed by the sensitivity list containing the input signal *clock*. Whenever *clock* changes state, the PROCESS is invoked, and the statements within the PROCESS will be evaluated to produce a result. A 'EVENT (read as “tick-event”) attribute will evaluate as TRUE if the signal preceding it has just changed states. Line 13 states that if *clock* has just changed states and right now it is '1', then we know it was a rising edge. To implement the PRESENT state/NEXT state table, a CASE construct is used. For each of the possible values of the variable *count*, we determine the NEXT state of the counter. Notice that the := operator is used to assign a value to a variable. Line 25 assigns the value stored in *count* to the output pins. Because *count* is a local variable, this assignment must be done before END PROCESS on line 26.

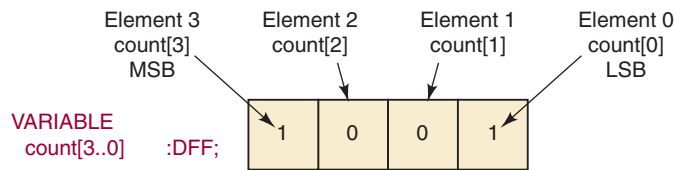
### Behavioral Description

The **behavioral level of abstraction** is a way to describe a circuit by describing its behavior in terms very similar to the way you might describe its operation in English. Think about the way a counter circuit's operation might be described by someone who knows nothing about flip-flops or logic gates. Perhaps that person's description would sound something like, “When the counter input changes from LOW to HIGH, the number on the output counts up by 1.” This level of description deals more with cause-and-effect relationships than with the path of data flow or wiring details. However, we cannot really use just any description in English to describe the circuit's behavior. The proper syntax must be used within the constraints of the HDL.

## AHDL

In AHDL, the first important step in this description method is to declare the counter output pins properly. They should be declared as a bit array, with indices decreasing left to right and with 0 as the least significant index in the array, as opposed to individual bits named a, b, c, d, and so on. In this way, the numeric value associated with the bit array's name is interpreted as a binary number upon which certain arithmetic operations can be performed. For example, the bit array *count* shown in Figure 7-42 might contain the bits 1001, as shown. The AHDL compiler interprets this bit pattern as having the value of 9 in decimal.

**FIGURE 7-42** The elements of a D register storing the number 9.



In order to create our MOD-5 counter in AHDL, we will need a three-bit register that will store the current counter state. This three-bit array, named *count*, is declared using D flip-flops on line 7 in Figure 7-43. Recall from Figure 7-40 that we could name the DFF array the same as the output port *q[2..0]* and thereby eliminate line 15, but we would also need to change *count[ ]* to *q[ ]* everywhere in the logic section. In other words, the statement on line 7 can be changed to

```
q[2..0] :DFF;
```

If this were done, all references to *count* thereafter would be changed to *q*. This can make the code shorter, but it does not demonstrate universal HDL concepts as clearly. In AHDL, all the clocks can be specified as being tied

```

1  SUBDESIGN fig7_43
2  (
3    clock      :INPUT;
4    q[2..0]    :OUTPUT;  -- declare 3-bit array of output bits
5  )
6  VARIABLE
7    count[2..0] :DFF;  -- declare a register of D flip flops.
8
9  BEGIN
10   count[].clk = clock;  -- connect all clocks to synchronous source
11   IF count[].q < 4 THEN  -- note; count[] is the same as count[].q
12     count[].d = count[].q + 1;  -- increment current value by one
13   ELSE count[].d = 0;  -- recycle to zero: force unused states to 0
14   END IF;
15   q[] = count[].q;  -- transfer register contents to outputs
16  END;
```

**FIGURE 7-43** Behavioral description of a counter in AHDL.

together and connected to a common clock source using the statement on line 10, `count[ ].clk = clock`. In this example, `count[ ].clk` refers to the clock input of each flip-flop in the array called `count`.

The behavioral description of this counter is very simple. The current state of the counter is evaluated (`count[ ].q`) on line 11, and if it is less than the highest desired count value, it uses the description `count[ ].d = count.q + 1` (line 12). This means that the current state of the *D* inputs must be equal to a value one count greater than the current state of the *Q* outputs. When the current state of the counter has reached the highest desired state (or higher), the *IF* statement test will be false, resulting in a NEXT state input value of zero (line 13), which recycles the counter. The last statement on line 15 simply connects the counter value to the output pins of the device.

## VHDL

In VHDL, the first important step in this description method is to declare properly the counter output port, as shown in Figure 7-44. The data type of the output port (line 3) must match the type of the counter variable (line 9), and it must be a type that allows arithmetic operations. Recall that VHDL treats `BIT_VECTORS` as just a string of bits, not as a binary numeric quantity. In order to recognize the signal as a numeric quantity, the data object must be typed as an `INTEGER`. The compiler looks at the `RANGE 0 TO 7` clause on line 3 and knows that the counter needs three bits. A similar declaration is needed for the register variable on line 9 that will actually be counting up. This is called `count`. The first statement after `BEGIN` in the `PROCESS` responds to the rising edge of the clock as in the previous examples. It then uses behavioral description methods to define the counter's response to the clock edge. If the counter has not reached its maximum (line 12), then it should be incremented (line 13). Otherwise (line 14), it should

```

1  ENTITY fig7_44 IS
2  PORT( clock:IN BIT;
3         q :OUT INTEGER RANGE 0 TO 7 );
4  END fig7_44;
5
6  ARCHITECTURE a OF fig7_44 IS
7  BEGIN
8      PROCESS (clock)
9          VARIABLE count: INTEGER RANGE 0 to 7; -- define a numeric VARIABLE
10         BEGIN
11             IF (clock = '1' AND clock'EVENT) THEN -- rising edge?
12                 IF count < 4 THEN -- less than max?
13                     count := count + 1; -- increment value
14                 ELSE -- must be at max or bigger
15                     count := 0; -- recycle to zero
16                 END IF;
17             END IF;
18             q <= count; -- transfer register contents to outputs
19         END PROCESS;
20     END a;

```

**FIGURE 7-44** Behavioral description of a counter in VHDL.

recycle the counter to zero (line 15). The last statement on line 18 simply connects the counter value to the output pins of the device.

### Simulation of Basic Counters

Simulation of any of our MOD-5 counter designs is pretty straightforward. The counters have only one input bit (*clock*) and three output bits (*q2 q1 q0*) to display in the simulation. The clock frequency has not been specified, so we can use any frequency that we wish for a functional simulation—although we probably should avoid a high-frequency clock unless we want to investigate the effects of propagation delays. About the only decision that we must make is to determine how many clock pulses to apply. Since the counter is a MOD-5 counter, we should apply at least five clock pulses to verify that the HDL design has the correct count sequence and that it recycles. The simulation will start with the initial state 000. We will not be able to test for any of the unused states because the HDL designs did not provide for a way to preset the counter to any of the unused states. Our simulation results for the HDL design of a MOD-5 counter are shown in Figure 7-45.

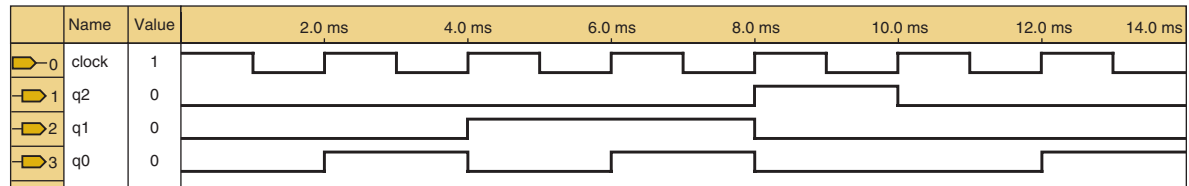


FIGURE 7-45 Simulation results for HDL design of MOD-5 counter.

### Full-Featured Counters in HDL

The examples we have chosen so far have been very basic counters. All they do is count up to four and then roll over to zero. The standard IC counters that we have examined have many other features that make them very useful for numerous digital applications. For example, consider the 74161 and the 74191 IC counters that were discussed in Section 7-7. These devices have combinations of various features including count enable, up/down counting, parallel loading (preset to any count), and clearing. In addition, these counters have been designed to easily cascade synchronously to create larger counters. In this section, we will explore the techniques that allow us to include these features in an HDL counter. We are going to create a counter that will combine more features than are found in either the 74161 or the 74191. We will use this example to demonstrate the methods of designing a counter with capabilities that specifically suit our needs. When we use HDLs to create digital designs, we are not limited to features that happen to be included with a certain IC.

Let's review the specifications for our more complex counter example. The recycling, MOD-16 binary counter is to change states on the rising edge of the clock input when the counter is enabled with a HIGH level. A direction control input will make the counter count up when it is LOW or count down when it is HIGH. The counter will have an active-HIGH, asynchronous clear to reset the counter immediately when the control input is activated. The counter can be synchronously loaded with a number on the data input pins when the load control is HIGH. The priority of the input control functions, from highest to lowest, will be clearing, loading, and counting. And finally, the counter will also include an active-HIGH output that will



detect the terminal state of the counter when the count function is enabled. Remember, the terminal state will be dependent on the count direction. As we will see, the correct operation of these features is determined by the way we write the HDL code, so we will have to pay very close attention to the details.

## AHDL FULL-FEATURED COUNTER

The code in Figure 7-46 implements all of the features we have discussed. This is a four-bit counter, but it can easily be expanded in size. Read through the inputs and outputs on lines 3 and 4 to make sure you understand what each one is supposed to do. If you do not, reread the previous paragraphs of this section. Line 7 defines a four-bit register of D flip-flops that will serve as the counter. It should be noted again here that this register could have been named the same as the output variable (*q*). The code is written with different names to distinguish between ports (inputs and outputs) of the circuit and the devices that are operating within the circuit. The clock input is connected to all the *clk* inputs of all the D flip-flops on line 10. All the active-LOW clear inputs (*clrn*) to the DFF primitive are connected to the complement of the *clear* input signal on line 11. This clears the flip-flops immediately when the *clear* input goes HIGH because the *prn* and *clrn* inputs to the DFF primitive are not dependent on the clock (i.e., they are asynchronous).

In order to make the load function synchronously, the *D* inputs to the flip-flops must be controlled so that the input data (*din*) is present on the *D* inputs when the load line is HIGH. This way, when the next active clock edge comes along, the data will be loaded into the counter. This action must happen regardless of whether the counter is enabled or not. Consequently,

```

1  SUBDESIGN fig7_46
2  (
3      clock, clear, load, cntenabl, down, din[3..0]      :INPUT;
4      q[3..0], term_ct :OUTPUT;  -- declare 4-bit array of output bits
5  )
6  VARIABLE
7      count[3..0] :DFF;          -- declare a register of D flip flops
8
9  BEGIN
10     count[].clk = clock;        -- connect all clocks to synch source
11     count[].clrn= !clear;      -- connect for asynch active HIGH clear
12     IF load THEN count[].d = din[]; -- synchronous load
13         ELSIF !cntenabl THEN count[].d = count[].q; -- hold count
14         ELSIF !down THEN count[].d = count[].q + 1; -- increment
15         ELSE count[].d = count[].q - 1;          -- decrement
16     END IF;
17     IF ((count[].q == 0) & down # (count[].q == 15) & !down) & cntenabl
18     THEN term_ct = VCC;        -- synchronous cascade output signal
19     ELSE term_ct = GND;
20     END IF;
21     q[] = count[].q;          -- transfer register contents to outputs
22 END;
```

FIGURE 7-46 Full-featured counter in AHDL.

the first conditional decision (IF) on line 12 evaluates the load input. Recall from Chapter 4 that the IF/ELSE decision structure gives precedence to the first condition that is found to be true because, once it finds a condition that is true, it does not go on to evaluate the conditions in subsequent ELSE clauses. In this case, it means that if the load line is activated, it does not matter whether the count is enabled, or it is trying to count up or down. It will do a parallel load on the next clock edge.

Assuming that the load line is not active, the ELSIF clause on line 13 is evaluated to see if the count is disabled. In AHDL, it is very important to realize that the *Q* output must be fed back to the *D* input so that, on the next clock edge, the register will hold its previous value. Forgetting to insert this clause results in the *D* inputs defaulting to zero, thus resetting the counter. If the counter is enabled, the ELSIF clause on line 14 is evaluated and either increments *count* (line 14) or decrements *count* (line 15). To summarize these decisions, first decide if it is time to load, next decide if the count should hold or change, then decide whether to count up or down.

The next function described is the detecting (or decoding) of the terminal count. Lines 17–20 decide whether the terminal count has been reached while counting up or down. The double equals (==) operator is the symbol that tests for equality between the expressions on each side of the operator. The counter state, which is the terminal state, depends on the counting direction. This is determined by ANDing the appropriate terminal state detection of 0 or 15 with the correct expression, *down* or *!down*. *Term\_ct* will output a HIGH if the correct state has been reached, otherwise it will be LOW. Line 21 will connect the output for *count* to the output pins for the SUBDESIGN.

One of the key concepts of using HDLs is that it is generally very easy to expand the size of a logic module. Let us look at the necessary changes to this AHDL design to increase the binary counter modulus to 256. Since  $2^8 = 256$ , we will need to increase the number of bits to eight. Only four modifications to Figure 7-46 will be required to make this change in counter modulus:

<i>Line #</i>	<i>Modification</i>
3	din[ <del>3</del> 7 . . 0]
4	q[ <del>3</del> 7 . . 0]
7	count[ <del>3</del> 7 . . 0]
17	(count[ ].q == <del>15</del> 255)

## VHDL FULL-FEATURED COUNTER

The code in Figure 7-47 implements all the features we have discussed. This is a four-bit counter, but it can easily be expanded in size. Read through the inputs and outputs on lines 2–5 to make sure you understand what each one is supposed to do. If you do not, reread the previous paragraphs of this section. The PROCESS statement on line 10 is the key to all clocked circuits described in VHDL, but it also plays an important role in determining whether the circuit responds synchronously or asynchronously to its inputs. We want this circuit to respond immediately to transitions on the *clock*, *clear*, and *down* inputs. With these signals in the sensitivity list, we assure that the code inside the PROCESS will be evaluated as soon as any of these inputs change states. The variable *count* is defined on line 11 as an INTEGER so it can be incremented and decremented easily. Variables are declared within the PROCESS and can be used within the PROCESS only.

```

1  ENTITY fig7_47 IS
2  PORT( clock, clear, load, cntenabl, down  :IN BIT;
3        din          :IN INTEGER RANGE 0 TO 15;
4        q            :OUT INTEGER RANGE 0 TO 15;
5        term_ct      :OUT BIT);
6  END fig7_47;
7
8  ARCHITECTURE a OF fig7_47 IS
9    BEGIN
10   PROCESS (clock, clear, down)
11     VARIABLE count :INTEGER RANGE 0 to 15;  -- define a numeric signal
12     BEGIN
13       IF clear = '1' THEN count := 0;      -- asynch clear
14       ELSIF (clock = '1' AND clock'EVENT) THEN -- rising edge?
15         IF load = '1' THEN count := din;   -- parallel load
16         ELSIF cntenabl = '1' THEN         -- enabled?
17           IF down = '0' THEN count := count + 1; -- increment
18           ELSE count := count - 1; -- decrement
19         END IF;
20       END IF;
21     END IF;
22     IF ((count = 0) AND (down = '1')) OR
23        ((count = 15) AND (down = '0')) AND cntenabl = '1'
24     THEN term_ct <= '1';
25     ELSE term_ct <= '0';
26     END IF;
27     q <= count;  -- transfer register contents to outputs
28   END PROCESS;
29 END a;

```

**FIGURE 7-47** Full-featured counter in VHDL.

The *clear* input is given precedence by evaluating it with the first IF statement on line 13. Recall from Chapter 4 that the IF/ELSE decision structure gives precedence to the first condition that is found to be true because it does not go on to evaluate the conditions in subsequent ELSE clauses. In this case, if the *clear* is active, the other conditions will not matter. The output will be zero. In order to make the *load* function operate synchronously, it must be evaluated after detecting the clock edge. The clock edge is detected on line 14, and the circuit checks immediately to see if *load* is active. If *load* is active, the *count* is loaded from *din*, regardless of whether or not the counter is enabled. Consequently, the conditional decision (IF) on line 15 evaluates the *load* input; only if it is inactive does it evaluate line 16 to see if the counter is enabled. If the counter is enabled, the *count* will be incremented or decremented (lines 17 and 18, respectively).

The next issue is detecting the terminal count. Lines 22–25 decide whether the maximum or minimum terminal count has been reached and drive the output to the appropriate level. The decision-making structure here is very important because we want to evaluate this situation, regardless of whether the decision-making process was invoked by *clock*, *clear*, or *down*. Notice that this decision is not another ELSE branch of the previous IF decisions but is evaluated for each signal in the sensitivity list *after* the clearing or counting has occurred. After all these decisions are made, *count*

should have the right value in the register, and line 27 effectively connects the register to the output pins.

One of the key concepts of using HDLs is that it is generally very easy to expand the size of a logic module. Let us look at the necessary changes to this VHDL design to increase the binary counter modulus to 256. Only four modifications to Figure 7-47 will be required to make this change in counter modulus:

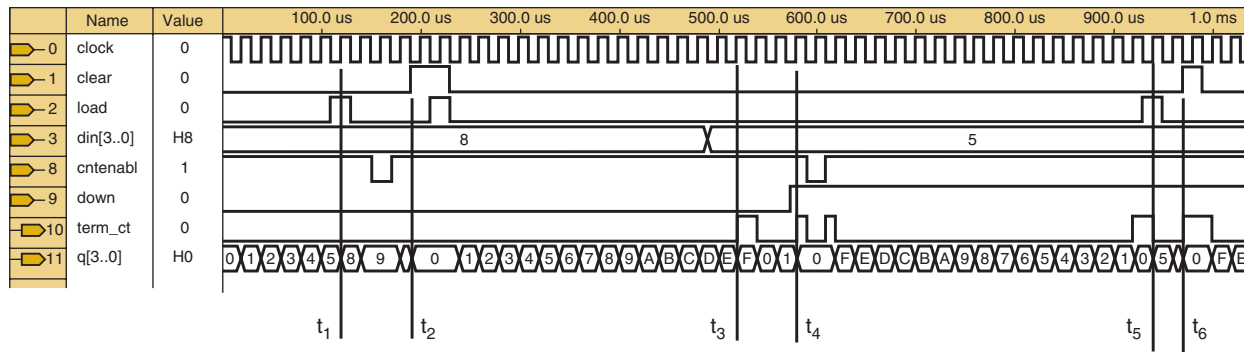
<i>Line #</i>	<i>Modification</i>
3	RANGE 0 TO <del>15</del> 255
4	RANGE 0 TO <del>15</del> 255
11	RANGE 0 TO <del>15</del> 255
23	(count = <del>15</del> 255)

### Simulation of Full-Featured Counter

Simulation of our full-featured counter design will require some planning to generate appropriate input waveforms. While it may not be necessary to exhaustively simulate every conceivable input combination, we do need to test enough of the possible input conditions to be convinced that it works properly. This is exactly what we should also do to test our prototype design on the bench. The counter has five different input signals (*clock*, *clear*, *load*, *cntenabl*, and *din*) and two different output signals (*q* and *term\_ct*) to display in our simulation. One of the input signals and one of the output signals actually is four bits wide. We will pick a convenient clock frequency since none has been specified for our functional simulation of the counter. We will need to provide enough clock pulses to allow us to look at several operational conditions. The simulation should test the functions of enabling and disabling the counter, counting up and counting down, clearing the counter, loading a value into the counter and counting from that value, and terminal count state detection.

There are some general simulation issues that we should consider in creating our input waveforms. The simulation will start with the initial output state at 0000. Therefore, it would be better to wait until the count has reached another state before applying a clear input so that we can see a change in the output. Likewise, loading in the same value as the counter's NEXT state does not really convince us that *load* is working correctly. Changing input control signals at the same time as the clocking edge occurs may create some setup time problems and produce questionable results. Asynchronous controls should be applied at a time other than the proper clocking edge to show clearly that the resultant circuit action is immediate and not dependent on the clock. In general, we should apply common sense in creating our input waveforms and consider what we are trying to verify with the simulation. Simulation will be valuable in the design process only if we apply appropriate input conditions and evaluate the results critically.

Some simulation results for the full-featured counter are shown in Figure 7-48. The four-bit input *din* and the four-bit output *q* are displayed in hexadecimal. The counter is initially enabled (*cntenabl* = 1) to count up (*down* = 0), and we see the output is incrementing 0, 1, 2, 3, 4, 5. At  $t_1$ , the counter synchronously (i.e., on the PGT of *clock*) responds to the HIGH applied to the *load* input. The counter is preset to the parallel data input (*din*) value of 8. This also shows that loading has priority over counting, since they are both active at the same time. After  $t_1$ , *load* is LOW again and the counter continues to count up from 8. A LOW input to *cntenabl* makes



**FIGURE 7-48** Simulation results for HDL design of full-featured counter.

the counter hold at state 9 for an extra clock cycle. The count is continued when *cntenabl* goes HIGH again until  $t_2$ , when the counter is asynchronously cleared. Notice the shortened time for the output state A due to the immediate clearing of the counter. We would have to zoom in to actually see that state A is displayed. We can also see that the clear function has the highest priority when all three controls, *clear*, *load*, and *cntenabl*, are simultaneously high. The count-up sequence continues and recycles to 0 after state F to verify that the counter is a MOD-16 binary counter. At  $t_3$ , the counter reaches its terminal state F when counting up, and *term\_ct* outputs a HIGH. At  $t_4$ , the counter starts counting down because *down* has been switched to a HIGH. Again, *term\_ct* outputs a HIGH since the counter is now at state 0, which is the terminal state when counting down. Notice that, by the action of *term\_ct*, the terminal state for the counter depends on its direction of counting, which is controlled by the input *down*. The count holds at state 0 for an extra clock period when *cntenabl* goes LOW. The output *term\_ct* is also disabled while *cntenabl* = 0. The down count sequence continues correctly when *cntenabl* again goes HIGH. At  $t_5$ , the counter synchronously loads the parallel data value 5. At  $t_6$ , the counter is asynchronously cleared. Again the priority of loading or clearing over a down count is verified at  $t_5$  and  $t_6$ . Did we verify that our design operates correctly in comparison to the specifications? We did a pretty good job, but there are a couple of test conditions that could also be added for completeness. Will the counter clear or load when the *cntenabl* is LOW? It appears that we neglected to verify those scenarios. As you can see, complex designs may require a lot of thought to verify their operation adequately by simulation or bench testing. Can you think of any other tests that we should make?

### OUTCOME ASSESSMENT QUESTIONS

1. What type of table is used to describe a counter's operation?
2. When designing a counter with D flip-flops, what is applied to the *D* inputs in order to drive it to the NEXT state on the next active clock edge?
3. How would you write the HDL description to trigger a storage device (flip-flop) on a falling edge instead of a rising edge of the clock?
4. Which method describes the circuit's operation using cause-and-effect relationships?
5. What is the difference between asynchronous clear and synchronous load?
6. How do you create an asynchronous clear function in an HDL?
7. How do you create functions priority in an HDL description of a counter?

## 7-13 WIRING HDL MODULES TOGETHER

### OUTCOMES

Upon completion of this section, you will be able to:

- Use hierarchical design techniques to describe systems.
- Combine blocks graphically.
- Combine blocks using VHDL.

In the previous section we looked at how to implement common counter features using an HDL. We should also investigate how we can connect these counter circuits to other digital modules to create larger systems. Designing large digital systems becomes much easier if the system is subdivided into smaller, more manageable modules that are then interconnected. This is the essence of the concept of **hierarchical design**, and we will readily see its benefits with example projects in Chapter 10. Let us now look at the basic techniques for wiring modules together.

### DECODING THE AHDL MOD-5 COUNTER

We looked briefly at the idea of decoding a counter in Section 7-8. You should recall that a decoding circuit detects a counter's state by the unique bit pattern for that state. Let's see how to connect a decoder circuit to the MOD-5 counter design in Figure 7-39 (or Figure 7-40). We will rename the counter SUBDESIGN mod5 to be a bit more descriptive in the block diagram for the overall circuit that we will draw later. Since the counter does not produce all eight possible states for a three-bit counter, our decoder design shown in Figure 7-49 will only decode the states that are used, 000 through 100. The three input bits ( $c = \text{MSB}$ ) declared on line 3 will be connected later to the MOD-5 counter's outputs. The five outputs for the decoder are named *state0* through *state4* on line 4. A CASE statement (lines 7–14) describes the behavior of the decoder by checking the  $c b a$  input combination to determine which one of the decoder outputs should be HIGH. When the  $c b a$  input is 000, only the *state0* output will be HIGH or, when  $c b a$  is 001, only the *state1* output will be HIGH, and so on. Any input value greater than 100, which is covered by OTHERS and actually should not occur in this application, will produce LOWs on all outputs.

```

1  SUBDESIGN decode5
2  (
3      c, b, a          : INPUT;
4      state[0..4]     : OUTPUT;
5  )
6  BEGIN
7      CASE (c,b,a) IS          -- decode binary value
8          WHEN B"000" => state[] = B"10000";
9          WHEN B"001" => state[] = B"01000";
10         WHEN B"010" => state[] = B"00100";
11         WHEN B"011" => state[] = B"00010";
12         WHEN B"100" => state[] = B"00001";
13         WHEN OTHERS => state[] = B"00000";
14     END CASE;
15 END;
```

FIGURE 7-49 AHDL MOD-5 counter decoder module.

We will instruct the Altera software to create symbols for our two design files, `mod5` (using a more descriptive name for any of the earlier design file choices) and `decode5`. This will allow us to draw a block diagram (see Figure 7-50) for our complete circuit that consists of these two modules, input and output ports, and the wiring between them. Each symbol is labeled with its respective SUBDESIGN name `mod5` or `decode5`. Notice that some of the wiring is drawn with heavier-weight lines. This is to represent a bus, which is a collection of signal lines. The lighter-weight lines are individual signals. The symbols created by Altera will automatically have ports drawn to indicate whether they represent individual signals or buses. This will be determined by the signal declarations in the SUBDESIGN section. Ports with group names will be drawn as buses. Since the counter output port is a bus but the decoder input ports are individual signals, it will be necessary to split the bus into individual signal lines to wire the two modules together. Whenever a bus is split, you must label both the group signal name of the bus and the individual signals that are being used. Our block diagram has a bus labeled `q[2..0]` and the corresponding individual signals `q[2]`, `q[1]`, and `q[0]`. The simulation results for this counter and decoder circuit are shown in Figure 7-51.

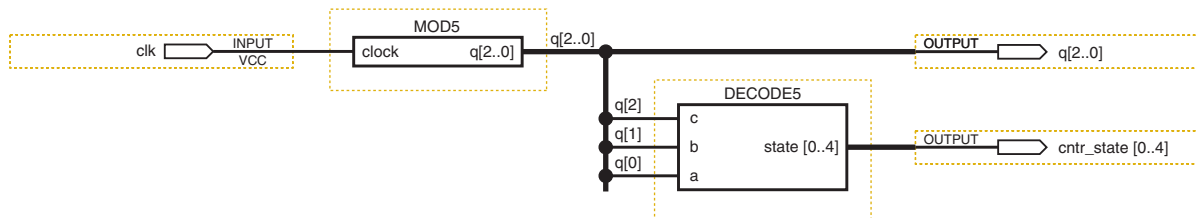


FIGURE 7-50 Block diagram design for the MOD-5 counter and decoder circuit.

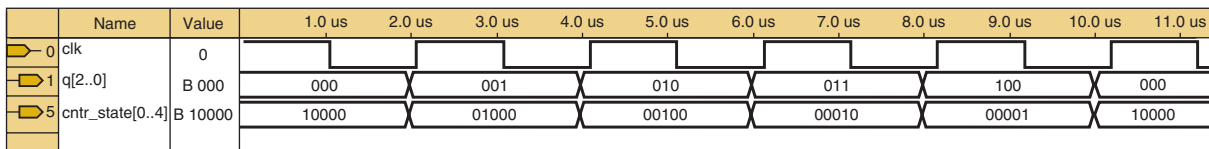


FIGURE 7-51 Simulation of MOD-5 counter and decoder circuit.

## DECODING THE VHDL MOD-5 COUNTER

We looked briefly at the idea of decoding a counter in Section 7-8. You should recall that a decoding circuit detects a counter's state by the unique bit pattern for that state. Let's see how to connect a decoder circuit to the MOD-5 counter design in Figure 7-41. We will rename the counter ENTITY `mod5` to make it easier to identify the module in our overall circuit. Since the counter does not produce all eight possible states for a three-bit counter, our decoder design shown in Figure 7-52 will only decode the states that are used, 000 through 100. The three input bits ( $c = \text{MSB}$ ) declared on line 3 will be connected later to the MOD-5 counter's outputs. The five outputs for the decoder are named `state`, a bit vector, on line 4. An internal bit vector signal named `input` is declared on line 9. Then line 11 combines the three input port bits ( $c b a$ ) together as a bit vector called `input`, which then can be evaluated by the CASE statement on lines 14–21. If any of the input bits changes logic level, the PROCESS will be invoked to determine the resultant output. The CASE statement describes the behavior of the decoder by checking the `input` combination (representing  $c b a$ ) to determine which one of the decoder outputs should be HIGH. When the `input` is 000, only the

```

1  ENTITY decode5  IS
2  PORT (
3      c, b, a    : IN BIT;
4      state     : OUT BIT_VECTOR (0 TO 4)
5  );
6  END decode5;
7
8  ARCHITECTURE a OF decode5 IS
9  SIGNAL  input   : BIT_VECTOR (2 DOWNT0 0);
10 BEGIN
11     input <= (c & b & a);    -- combine inputs into bit vector
12     PROCESS (c, b, a)
13     BEGIN
14         CASE input IS
15             WHEN "000" =>      state <= "10000";
16             WHEN "001" =>      state <= "01000";
17             WHEN "010" =>      state <= "00100";
18             WHEN "011" =>      state <= "00010";
19             WHEN "100" =>      state <= "00001";
20             WHEN OTHERS =>     state <= "00000";
21         END CASE;
22     END PROCESS;
23 END a;
```

**FIGURE 7-52** VHDL MOD-5 counter decoder module.

*state(0)* output will be HIGH; when *input* is 001, only the *state(1)* output will be HIGH; and so on. Any *input* value greater than 100, which is covered by OTHERS and actually should not occur in this application, will produce LOWs on all outputs.

Since we are using the Altera Quartus Development software, we can connect the two modules graphically. To do this, you will need to instruct the software to create symbols for our two design files, mod5 (using a more descriptive name for any of the earlier design file choices) and decode5. This will allow us to draw a block diagram (see Figure 7-50) for our complete circuit that consists of these two modules, input and output ports, and the wiring between them. Notice that some of the wiring is drawn with heavier-weight lines. This is to represent a bus, which is a collection of signal lines. The lighter-weight lines are individual signals. The symbols created by Altera will automatically have ports drawn to indicate whether they represent individual signals or buses. This will be determined by the data type declarations for each port of the ENTITY. BIT\_VECTOR ports will be drawn as buses and BIT type ports will be drawn as individual signal lines. Since the counter output port is a bus but the decoder input ports are individual signals, it will be necessary to split the bus into individual signal lines to wire the two modules together. Whenever a bus is split, you must label both the group signal name of the bus and the individual signals that are being used. Our block diagram has a bus labeled  $q[2..0]$  and the corresponding individual signals  $q[2]$ ,  $q[1]$ , and  $q[0]$ . The simulation results for this counter and decoder circuit are shown in Figure 7-51.

The standard VHDL technique (and an alternative with Altera's software) to connect design modules is to use VHDL to describe the connections between the modules in a text file. The desired modules are instantiated in a higher-level design file using COMPONENTs in which the module's PORTs are declared. The wiring connections for each instance where the module is utilized are listed in a PORT MAP. A VHDL file that connects the mod5 and decode5 modules together is shown in Figure 7-53. Even though  $q$  is an output port for our top-level design file, it is typed as a BUFFER on line 4 due



```

1  ENTITY mod5decoded1 IS
2  PORT (
3      clk          :IN BIT;
4      q            :BUFFER BIT_VECTOR (2 DOWNT0 0);
5      cntr_state   :OUT BIT_VECTOR (0 TO 4)
6  );
7  END mod5decoded1;
8
9  ARCHITECTURE toplevel OF mod5decoded1 IS
10 COMPONENT mod5
11     PORT (
12         clock     :IN BIT;
13         q         :OUT BIT_VECTOR (2 DOWNT0 0)
14     );
15 END COMPONENT;
16 COMPONENT decode5
17     PORT (
18         c, b, a   :IN BIT;
19         state     :OUT BIT_VECTOR (0 TO 4)
20     );
21 END COMPONENT;
22 BEGIN
23 counter:  mod5      PORT MAP (clock => clk, q => q);
24 decoder:  decode5  PORT MAP
25     (c => q(2), b => q(1), a => q(0), state => cntr_state);
26 END toplevel;

```

**FIGURE 7-53** Higher-level VHDL file to connect mod5 and decode5 together.

to the fact that it is necessary to “read” the bit vector array for an input to the *decode5* COMPONENT in its PORT MAP (line 25). VHDL does not permit output ports to be used as inputs. The BUFFER data type declaration provides a port that can be used for both input and output. The mod5 module is declared on lines 10–15 and the decode5 module is declared on lines 16–21. The mod5 and decode5 ENTITY/ARCHITECTURE descriptions may be included within the top-level design file, or instead they may be saved in the same folder as the top-level file as was done here. The PORT MAP for each instance of the modules is listed on lines 23 and 24–25. The word to the left of the colon is a unique label for each instance and the module name is on the right, then the keywords PORT MAP, and finally, in parentheses, are the named associations between the design signals and ports. The => operator indicates which module ports (on the left side) are connected to which higher-level system signals (on the right side). This circuit produces the simulation results shown in Figure 7-51.

### MOD-100 BCD Counter

We wish to design a recycling, MOD-100 BCD counter that has a synchronous clear. Creating a MOD-10 BCD counter module and synchronously cascading two of these modules together in a higher-level design file is the easiest way to do this. The clock inputs to the two MOD-10 modules will both be connected to the system clock to achieve synchronous cascading of the two counter modules. Remember, there are significant benefits to using synchronous counter design rather than asynchronous clocking techniques. Also, if we did not employ synchronous clocking, the synchronous clear would not work properly. Even though the design specifications did not require a count enable or terminal count detection for the MOD-100 counter, it will be necessary to include these features in our design. In order

to synchronously cascade two counters, the enable and decoding features will be needed. The count enable input causes the counter to ignore clock edges unless it is enabled. The terminal count output indicates that the counting sequence has reached its limit and will roll over on the next clock. To synchronously cascade counter stages together, the terminal count output is connected to the next higher-order stage's enable input. By using the count enable to also control the decoding of the terminal count, our MOD-10 module can be used to create even larger BCD counters.

## CASCADING AHDL BCD COUNTERS

Our MOD-10 BCD counter SUBDESIGN is shown in Figure 7-54. The terminal state for a BCD counter is 9. Lines 10–13 will detect this terminal state only when the counter is enabled with a HIGH. ANDing the *enable* control in the decoding function will allow more than two counter modules to be cascaded synchronously if necessary and makes our mod10 design more versatile. The *clear* function will operate synchronously in AHDL by including it in the IF statement as shown on lines 14–15. If *clear* is inactive, we check to see if the counter is enabled (line 16). If *enable* is HIGH, the counter checks, using a nested IF on lines 17–21, to see if the last state 9 has been reached. After state 9, the counter synchronously recycles to 0. Otherwise, the count will be incremented. If the counter is disabled, lines 22–23 will hold the current count value by feeding the current output back to the counter's input. This holding action will be necessary in the cascaded MOD-100 counter for the 10s digit to hold its current state while the 1s digit progresses through its count sequence. An appropriate design strategy would be for us to simulate this module to determine if it functions correctly before we use it in a more complex circuit application. From the simulation results for mod10,

```

1  SUBDESIGN  mod10
2  (
3      clock, enable, clear      :INPUT;
4      counter[3..0], tc        :OUTPUT;
5  )
6  VARIABLE
7      counter[3..0]             :DFF;
8  BEGIN
9      counter[].clk = clock;
10     IF counter[].q == 9 & enable == VCC THEN
11         tc = VCC;              -- detect terminal count
12     ELSE
13         tc = GND;
14     END IF;
15     IF clear THEN
16         counter[].d = B"0000";  -- synchronous clear
17     ELSIF enable THEN          -- clear has priority
18         IF counter[].q == 9 THEN -- check for last state
19             counter[].d = B"0000";
20         ELSE
21             counter[].d = counter[].q + 1;  -- increment
22         END IF;
23     ELSE                        -- hold count when disabled
24         counter[].d = counter[].q;
25     END IF;
26 END;
```

FIGURE 7-54 MOD-10 BCD counter in AHDL.

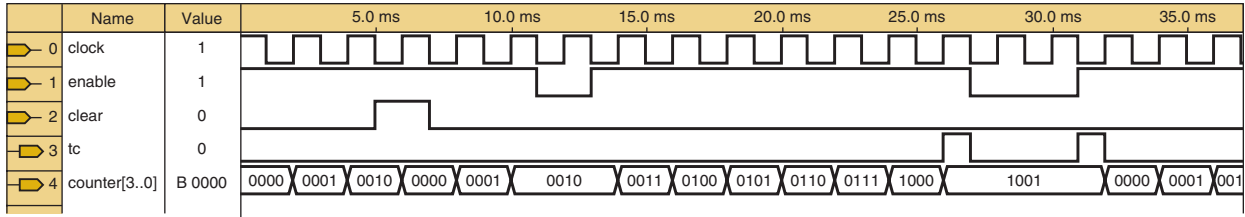


FIGURE 7-55 MOD-10 simulation results.

given in Figure 7-55, we see that the count sequence is correct, the *clear* is synchronous and has priority, and *enable* controls both the count function and the decoding output *tc*.

After creating a default symbol for our mod10 counter module, we can draw the block diagram for the MOD-100 BCD counter application. The input ports, output ports, and wiring have also been added to create the design in Figure 7-56. Notice that the counter outputs representing the 1s and 10s digits are drawn as buses. The mod10 modules are clocked synchronously. They are cascaded by using the terminal count output from the 1s digit to control the enable input on the 10s digit. The *en* input port controls the enabling/disabling of the entire MOD-100 counter circuit. The BCD counter design can be easily expanded with an additional mod10 stage by connecting the *tc* output to the next *enable* input for each digit needed. A sample of simulation results can be seen in Figure 7-57. The simulation shows that the MOD-100 counter has a correct BCD count sequence and can be synchronously cleared.

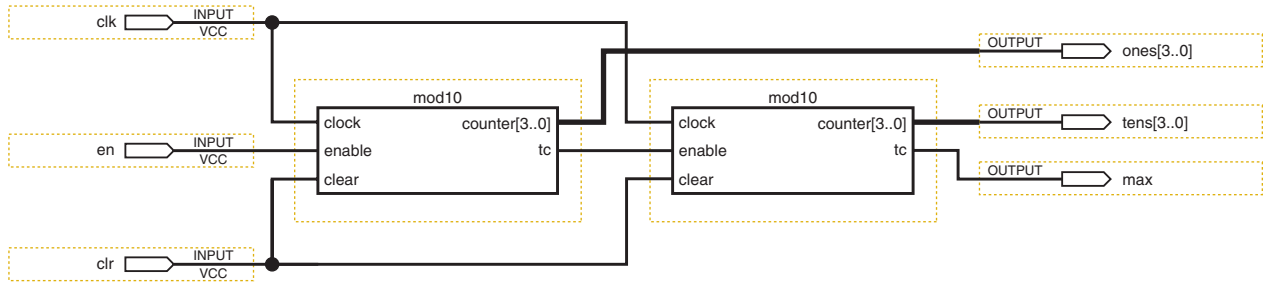


FIGURE 7-56 Block diagram design for a MOD-100 BCD counter.

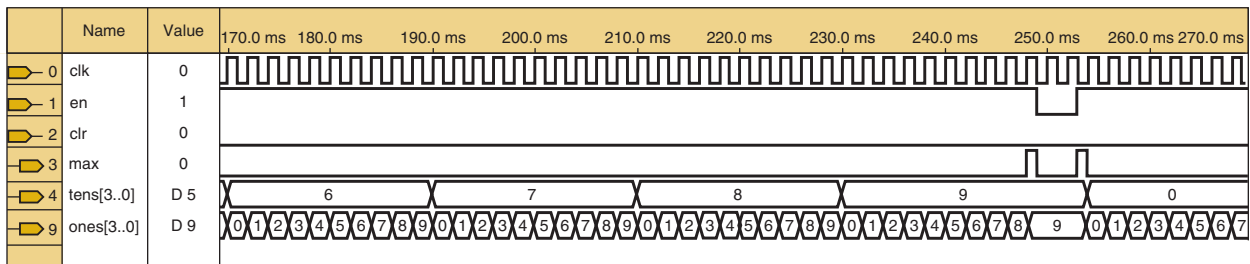
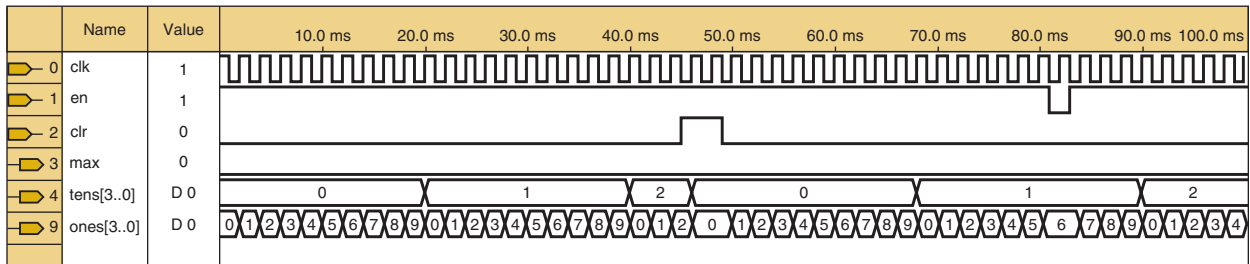


FIGURE 7-57 Simulation results for MOD-100 BCD counter design.

## CASCADING VHDL BCD COUNTERS

The ENTITY and ARCHITECTURE for our MOD-10 BCD counter is shown in lines 26–51 of Figure 7-58. The terminal state for a BCD counter is 9. Lines 38–40 will detect this terminal state only when the counter is enabled with a HIGH. ANDing the *enable* control in the decoding function will allow more than two counter modules to be cascaded synchronously if necessary

```

1  ENTITY mod100 IS
2  PORT (
3      clk, en, clr                :IN BIT;
4      ones                       :OUT INTEGER RANGE 0 TO 15;
5      tens                       :OUT INTEGER RANGE 0 TO 15;
6      max                       :OUT BIT
7  );
8  END mod100;
9  ARCHITECTURE toplevel OF mod100 IS
10 COMPONENT mod10
11     PORT (
12         clock, enable, clear    :IN BIT;
13         q                      :OUT INTEGER RANGE 0 TO 15;
14         tc                     :OUT BIT
15     );
16 END COMPONENT;
17 SIGNAL rco                    :BIT;
18 BEGIN
19 digit1: mod10 PORT MAP (clock => clk, enable => en,
20                       clear => clr, q => ones, tc => rco);
21 digit2: mod10 PORT MAP (clock => clk, enable => rco,
22                       clear => clr, q => tens, tc => max);
23 END toplevel;
24
25
26 ENTITY mod10 IS
27 PORT (
28     clock, enable, clear        :IN BIT;
29     q                          :OUT INTEGER RANGE 0 TO 15;
30     tc                         :OUT BIT
31 );
32 END mod10;
33 ARCHITECTURE lowerblk OF mod10 IS
34 BEGIN
35     PROCESS (clock, enable)
36         VARIABLE counter        :INTEGER RANGE 0 TO 15;
37     BEGIN
38         IF ((counter = 9) AND (enable = '1')) THEN tc <= '1';
39         ELSE tc <= '0';
40         END IF;
41         IF (clock'EVENT AND clock = '1') THEN
42             IF (clear = '1') THEN counter := 0;
43             ELSIF (enable = '1') THEN
44                 IF (counter = 9) THEN counter := 0;
45                 ELSE counter := counter + 1;
46             END IF;
47         END IF;
48     END IF;
49     q <= counter;
50 END PROCESS;
51 END lowerblk;

```

FIGURE 7-58 MOD-100 BCD counter in VHDL.

and makes our mod10 design more versatile. The *clear* function will be synchronous in VHDL by placing it in the nested IF statement (line 42) after the clock edge has been detected in line 41. If *clear* is inactive, we check to see if the counter is enabled (line 43). If *enable* is HIGH, the counter checks, using another nested IF on lines 44–46, to see if the last state 9 has been reached. After state 9, the counter synchronously recycles to 0. Otherwise, the count will be incremented. If the counter is disabled, VHDL will automatically hold the current count value. This holding action will be necessary in the cascaded MOD-100 counter for the 10s digit to hold its current state while the 1s digit progresses through its count sequence. An appropriate design strategy would be for us to simulate this module as a separate ENTITY to determine if it functions correctly before we use it in a more complex circuit application. Simulation results for the mod10 ENTITY, given in Figure 7-55, show that the count sequence is correct, the *clear* is synchronous and has priority, and *enable* controls both the count function and the decoding output.

We have two choices for implementing the MOD-100 counter. One technique is to represent the design graphically in a block diagram as seen in Figure 7-56. The mod10 counter modules, input ports, output ports, and wiring have also been added to create the MOD-100 counter. Notice that the counter outputs representing the 1s and 10s digits are drawn as buses. The mod10 modules are clocked synchronously. They are cascaded by using the terminal count output from the 1s digit to control the enable input on the 10s digit. The *en* input port controls the enabling/disabling of the entire MOD-100 counter circuit. The BCD counter design can be easily expanded with an additional mod10 stage by connecting the *tc* output to the next *enable* input for each digit needed. A sample of simulation results can be seen in Figure 7-57. The simulation shows that the MOD-100 counter has a correct BCD count sequence and can be synchronously cleared.

The second technique for creating the MOD-100 counter is to make the necessary connections between design modules by describing the circuit structure with VHDL. The listing for this system design file is given in Figure 7-58. The ENTITY/ARCHITECTURE description for the mod10 sub-block is contained within the overall mod100 design file (but could be in a separate file within this project's folder). The mod100 design file would be the top level for the hierarchical design of this system. It contains lower-level sub-blocks, which are actually two copies of the lower-level mod10 counter. The mod10 COMPONENT is declared in this higher-level design file (lines 10–16). The wiring connections for each instance where the module is utilized are listed in a PORT MAP. Since we need two instances of mod10, there is a PORT MAP for each instance (lines 19–20 and 21–22). Each instance must have a unique label (*digit1* or *digit2*) to distinguish them from each other. The PORT MAPs contain named associations between the lower-level module ports, given on the left, and the higher-level signals to which they are connected, given on the right. This circuit produces the same simulation results shown in Figure 7-57.

#### OUTCOME ASSESSMENT QUESTIONS

1. Describe how to connect HDL modules together to create a digital system.
2. What is a bus and how is it represented in a graphical block diagram design file in Altera?
3. What counter features must be included to synchronously cascade counter modules together?

## 7-14 STATE MACHINES

---

### OUTCOMES

Upon completion of this section, you will be able to:

- Define the Mealy and Moore models of state machines.
- Differentiate between counters and state machines and the methods to describe both.
- Describe Mealy and Moore models of state machines using HDL.

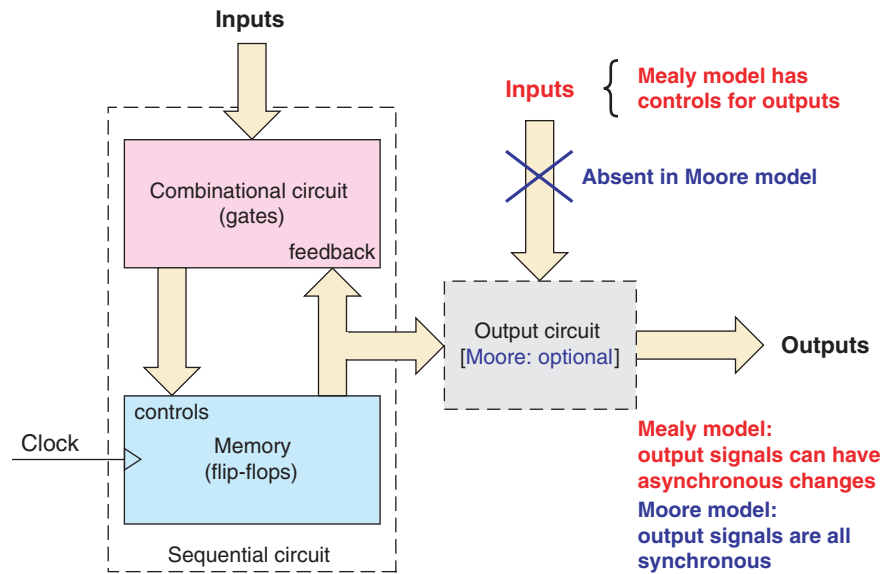
The term **state machine** refers to a circuit that sequences through a set of predetermined states controlled by a clock and other input signals. So the counter circuits we have been studying so far in this chapter are state machines. Generally, we use the term *counter* for sequential circuits that have a regular numeric count sequence. They may count up or count down, they may have a full  $2^N$  modulus or they may have a  $< 2^N$  modulus, or they may recycle or stop automatically at some predetermined state. A counter, as its name implies, is used to count things. The things that are counted are actually called clock pulses, but the pulses may represent many kinds of events. The pulses may be the cycles of a signal for frequency division or they may be seconds, minutes, and hours of a day for a digital clock. They may indicate that an item has moved down the conveyer in a factory or that a car has passed a particular spot on the highway.

The term *state machine* is more often used to describe other kinds of sequential circuits. They may have an irregular counting pattern like our stepper motor control circuit in Section 7-10. The objective for that design was to drive a stepper motor so that it would rotate in precise angular steps. The control circuit had to produce the required specific sequence of states for that movement, rather than count numerically. There are also many applications where we do not care about the specific binary value for each state because we will use appropriate decoding logic to identify specific states of interest and to generate desired output signals. The general distinction between the two terms is that a *counter* is commonly used to count events, while a *state machine* is commonly used to control events. The correct descriptive term depends on how we wish to use the sequential circuit.

The block diagram shown in Figure 7-59 may represent a state machine or a counter. In Section 7-10 we found out that the classic sequential circuit design process was to figure out how many flip-flops would be needed and then determine the necessary combinational circuit to produce the desired sequence. The output produced by a counter or a state machine may come directly from the flip-flop outputs or there may be some gating circuitry needed, as indicated in the block diagram. The two variations are described as either a **Mealy model** for a sequential circuit or a **Moore model**. In the Mealy model the output signals are also controlled by additional input signals, while the Moore model does not have any external controls for the generated output signals. The Moore output is a function only of the current flip-flop state. An example of a Moore-type design would be the decoded MOD-5 circuit in Section 7-13. On the other hand, the BCD counter design in the same section would be a Mealy-type design because of the external input (*enable*) that controls the terminal state decoding output (*tc*). One significant consequence of this subtle design variation is that Moore-type circuit outputs will be completely synchronous to the circuit's clock, while outputs produced by a Mealy-type circuit can change asynchronously. The enable input is not synchronized to the system clock in our MOD-10 design.

---

**FIGURE 7-59** Block diagram for counters and state machines.



HDLs, of course, can make state machines easy and intuitive to describe. As an oversimplified example that everyone can relate to, the following hardware description deals with four states through which a typical washing machine might progress. Although a real washing machine is more complex than this example, it will serve to demonstrate the techniques. This washing machine is idle until the start button is pressed, then it fills with water until the tub is full, then it runs the agitator until a timer expires, and finally it spins the tub until the water is spun out, at which time it goes back to idle. The point of this example focuses on the use of a set of named states for which no binary values are defined. The name of the counter variable is *wash*, which can be in any of the named states: *idle*, *fill*, *agitate*, or *spin*.

## SIMPLE AHDL STATE MACHINE

The AHDL code in Figure 7-60 shows the syntax for declaring a counter with named states on lines 6 and 7. The name of this counter is *cycle*. The keyword **MACHINE** is used in AHDL to define *cycle* as a state machine. The number of bits needed for this counter to produce the named states will be determined by the compiler. Notice that in line 7 the states are named, but the binary value for each state is also left for the compiler to determine. The designer does not need to worry about this level of detail. The CASE structure on lines 11–25 and the decoding logic that drives the outputs (lines 27–33) refer to the states by name. This makes the description easy to read and allows the compiler more freedom to minimize the circuitry. If the design requires the state machine also to be connected to an output port, then line 6 can be changed to:

```
cycle: MACHINE OF BITS (st[1..0])
```

and the output port *st[1..0]* can be added to the SUBDESIGN section. A second state machine option that is available is the ability for the designer

**FIGURE 7-60** State machine example using AHDL.

```

1  SUBDESIGN fig7_60
2  (  clock, start, full, timesup, dry      :INPUT;
3    water_valve, ag_mode, sp_mode      :OUTPUT;
4  )
5  VARIABLE
6  cycle:  MACHINE
7          WITH STATES (idle, fill, agitate, spin);
8  BEGIN
9  cycle.clk = clock;
10
11     CASE cycle IS
12         WHEN idle =>IF start THEN cycle = fill;
13             ELSE      cycle = idle;
14         END IF;
15     WHEN fill =>IF full THEN cycle = agitate;
16         ELSE      cycle = fill;
17     END IF;
18     WHEN agitate => IF timesup THEN cycle = spin;
19         ELSE      cycle = agitate;
20     END IF;
21     WHEN spin => IF dry THEN cycle = idle;
22         ELSE      cycle = spin;
23     END IF;
24     WHEN OTHERS => cycle = idle;
25 END CASE;
26
27     TABLE
28     cycle      => water_valve,      ag_mode,      sp_mode;
29     idle       => GND,              GND,          GND;
30     fill       => VCC,              GND,          GND;
31     agitate    => GND,              VCC,          GND;
32     spin       => GND,              GND,          VCC;
33 END TABLE;
34 END;
```

to define a binary value for each state. This can be accomplished in this example by changing line 7 to:

```
WITH STATES (idle = B"00", fill = B"01", agitate = B"11",
spin = B"10");
```

## SIMPLE VHDL STATE MACHINE

The VHDL code in Figure 7-61 shows the syntax for declaring a counter with named states. On line 6, a data object is declared named *state\_machine*. Notice the keyword **TYPE**. This is called an **enumerated type** in VHDL, in which the designer lists by symbolic names all possible values that a signal, variable, or port that is declared to be of that type is allowed to have. Notice also that on line 6, the states are named, but the binary value for each state is left for the compiler to determine. The designer does not need to worry



```

1  ENTITY fig7_61 IS
2  PORT (  clock, start, full, timesup, dry           :IN BIT;
3         water_valve, ag_mode, sp_mode             :OUT BIT);
4  END fig7_61;
5  ARCHITECTURE vhdl OF fig7_61 IS
6  TYPE state_machine IS (idle, fill, agitate, spin);
7  BEGIN
8  PROCESS (clock)
9  VARIABLE cycle           :state_machine;
10 BEGIN
11 IF (clock'EVENT AND clock = '1') THEN
12 CASE cycle IS
13 WHEN idle =>
14     IF start = '1' THEN      cycle := fill;
15     ELSE                    cycle := idle;
16     END IF;
17 WHEN fill =>
18     IF full = '1' THEN      cycle := agitate;
19     ELSE                    cycle := fill;
20     END IF;
21 WHEN agitate =>
22     IF timesup = '1' THEN   cycle := spin;
23     ELSE                    cycle := agitate;
24     END IF;
25 WHEN spin =>
26     IF dry = '1' THEN       cycle := idle;
27     ELSE                    cycle := spin;
28     END IF;
29 END CASE;
30 END IF;
31 CASE cycle IS
32 WHEN idle   => water_valve <= '0'; ag_mode <= '0'; sp_mode <= '0';
33 WHEN fill   => water_valve <= '1'; ag_mode <= '0'; sp_mode <= '0';
34 WHEN agitate => water_valve <= '0'; ag_mode <= '1'; sp_mode <= '0';
35 WHEN spin   => water_valve <= '0'; ag_mode <= '0'; sp_mode <= '1';
36 END CASE;
37 END PROCESS;
38 END vhdl;

```

**FIGURE 7-61** State machine example using VHDL.

about this level of detail. The CASE structure on lines 12–29 and the decoding logic that drives the outputs (lines 31–36) refer to the states by name. This makes the description easy to read and allows the compiler more freedom to minimize the circuitry.

## Simulation of State Machines

Using the simulator to verify our HDL designs produces the results given in Figure 7-62. The Altera simulator allows us to also simulate intermediate nodes in our design modules. The “buried” state machine named *cycle* has been included in the simulation in order to confirm that it operates correctly. Note that the results for *cycle* are given twice, since it will be displayed differently for the two HDLs. The simulator cannot actually show the simulations for both AHDL and VHDL together. The second buried node information has been merely copied and pasted for a composite figure here. In AHDL the machine state names are displayed, while in VHDL the compiler-assigned values for the enumerated state names are displayed instead.

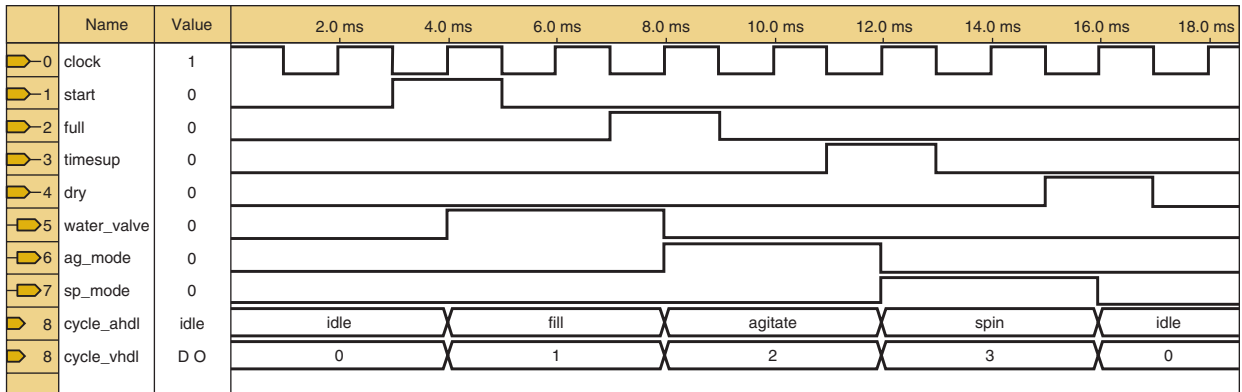


FIGURE 7-62 Simulation of washing machine HDL design example for a state machine.

### Traffic Light Controller State Machine

Let us investigate a state machine design that is a little more complicated, a traffic light controller. The block diagram is shown in Figure 7-63. Our simple controller is designed to control the flow of traffic at the intersection of a main road with a less busy side road. Traffic will flow uninterrupted on the main road with a green light, until a car is sensed on the side road (indicated by the input labeled *car*). After a time delay that is set by the five-bit binary input labeled *tmaingrn*, the main road light will change to yellow. The *tmaingrn* time delay ensures that the main road will receive a green light for at least this length of time during each cycling of the lights. The yellow light will last for a fixed amount of time that is set in the HDL design and then transition to red. When the main road light is red, the side road light turns to green. The side road light will be green for a time that is set by the five-bit binary input labeled *tsidegrn*. Again the yellow light will last for the same fixed length of time and then the side road will return to

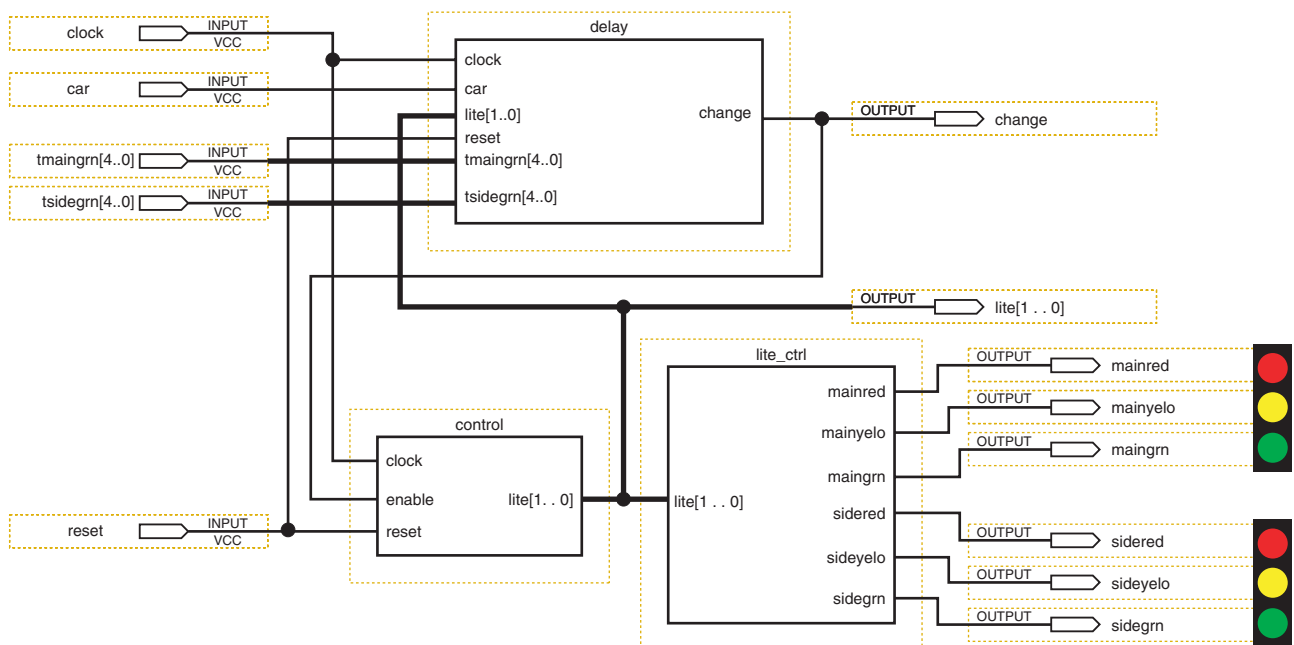


FIGURE 7-63 Traffic light controller.

a red light and the main road light will be green again. The delay module will control the time periods for each of the lights. The actual time delays will be the period of the system clock multiplied times the delay factor. The control module determines the state of the traffic controller. There are four light combinations—main-green/side-red, main-yellow/side-red, main-red/side-green, and main-red/side-yellow—so control will need four states. The traffic light states are translated into the proper on-off patterns for each of the six pairs of lights by the `lite_ctrl` module. The outputs labeled *change* and *lite* are provided for diagnostic purposes. *Reset* is used to initialize each of the two sequential circuits.

## AHDL TRAFFIC LIGHT CONTROLLER

The three design modules for our AHDL traffic light controller are listed together in Figure 7-64. They are actually three separate design files that are interconnected with the block diagram design shown in Figure 7-63. The delay module (lines 1–23) is basically a buried down counter (line 20) named *mach*, which waits at zero when the main road has a green light (*lite* = 0) until it is triggered by the car sensor (line 13) to load the delay factor *tmaingrn*–1 on line 14. Since the counter decrements all the way to zero, one is subtracted from each delay factor to make the delay counter’s modulus equal to the value of the delay factor. For example, if we wish to have a delay factor of 25, the counter must count from 24 down to 0. The actual length of time represented by the delay factors depends on the clock frequency. With a 1-Hz clock frequency, the period would be 1 s, and the delay factors would then be in seconds. Line 22 defines an output signal called *change* that detects when *mach* is equal to one. *Change* will be HIGH to indicate that the test condition is true, which in turn will enable the state machine in the control module to move to its next state (*lite* = 1) when clocked to indicate a yellow light on the main road. As the delay counter *mach* counts down and reaches zero, CASE determines that *lite* has a new value and the fixed time delay factor of 5 for a yellow light is loaded (actually loading one less than 5, as previously discussed) into *mach* (line 16) on the next clock. The count down continues from this new delay time, with *change* again enabling the control module to move to its next state (*lite* = 2) when *mach* is equal to 1, resulting in a green light for the side road. When *mach* again reaches zero, the time delay (*tsidegrn*–1) for a green light on the side road will be loaded into the down counter (line 17). When *change* again goes active, *lite* will advance to state 3 for a yellow light on the side road. *Mach* will recycle to the value 4 (5–1) on line 18 for the fixed time delay for a yellow light. When *change* goes active this time, the control module will return to the *lite* = 0 state (green light on main). When *mach* decrements to its terminal state (zero) this time, lines 13–15 will determine by the status of the *car* sensor input whether to wait for another car or to load in the delay factor for a green light on main (*tmaingrn*–1) to start the cycle over again. The main road will receive a green light for at least this length of time, even if there is a continuous stream of cars on the side road. It is obvious that we could make improvements to this design, but that, of course, would also complicate the design further.

The control module (lines 25–40) contains a state machine named *light* that will sequence through the four states for the traffic light combinations. The bits for the state machine are named and connected as an output port for this module (lines 27 and 29). The four states for *light* are named *mgrn*, *myel*, *sgrn*, and *syel* on line 30. Each state represents which road, main or

```

1  SUBDESIGN delay
2  ( clock, car, lite[1..0], reset      :INPUT;
3    tmaingrn[4..0], tsidegrn[4..0]   :INPUT;
4    change                            :OUTPUT; )
5  VARIABLE
6    mach[4..0]                        :DFF;
7  BEGIN
8    mach[].clk = clock;                -- with 1 Hz clock, times in seconds
9    mach[].clrn = reset;
10   IF mach[] == 0 THEN
11     CASE lite[] IS                  -- check state of light controller
12       WHEN 0 =>
13         IF !car THEN mach[].d = 0;   -- wait for car on side road
14         ELSE mach[].d = tmaingrn[] - 1; -- set time for main's green
15         END IF;
16       WHEN 1 => mach[].d = 5 - 1;     -- set time for main's yellow
17       WHEN 2 => mach[].d = tsidegrn[] - 1; -- set time for side's green
18       WHEN 3 => mach[].d = 5 - 1;     -- set time for side's yellow
19     END CASE;
20   ELSE mach[].d = mach[].q - 1;      -- decrement timer counter
21   END IF;
22   change = mach[] == 1;             -- change lights on control module
23 END;
24 -----
25 SUBDESIGN control
26 ( clock, enable, reset      :INPUT;
27   lite[1..0]                :OUTPUT; )
28 VARIABLE
29   light: MACHINE OF BITS (lite[1..0]) -- need 4 states for light combinations
30         WITH STATES (mgrn = B"00", myel = B"01", sgrn = B"10", syel = B"11");
31 BEGIN
32   light.clk = clock;
33   light.reset = !reset;            -- MACHINES have asynchronous, active-high reset
34   CASE light IS                  -- wait for enable to change light states
35     WHEN mgrn    => IF enable THEN light = myel; ELSE light = mgrn; END IF;
36     WHEN myel   => IF enable THEN light = sgrn; ELSE light = myel; END IF;
37     WHEN sgrn   => IF enable THEN light = syel; ELSE light = sgrn; END IF;
38     WHEN syel   => IF enable THEN light = mgrn; ELSE light = syel; END IF;
39   END CASE;
40 END;
41 -----
42 SUBDESIGN lite_ctrl
43 ( lite[1..0]                  :INPUT;
44   mainred, mainyelo, maingrn  :OUTPUT;
45   sidered, sideyelo, sidegrn  :OUTPUT; )
46 BEGIN
47   CASE lite[] IS                -- determine which lights to turn on
48     WHEN B"00" => maingrn = VCC; mainyelo = GND; mainred = GND;
49                   sidegrn = GND; sideyelo = GND; sidered = VCC;
50     WHEN B"01" => maingrn = GND; mainyelo = VCC; mainred = GND;
51                   sidegrn = GND; sideyelo = GND; sidered = VCC;
52     WHEN B"10" => maingrn = GND; mainyelo = GND; mainred = VCC;
53                   sidegrn = VCC; sideyelo = GND; sidered = GND;
54     WHEN B"11" => maingrn = GND; mainyelo = GND; mainred = VCC;
55                   sidegrn = GND; sideyelo = VCC; sidered = GND;
56   END CASE;
57 END;

```

**FIGURE 7-64** AHDL design files for traffic light controller.

side, is to receive a green or yellow light. The other road will have a red light. The values for each state of the control module have also been specified on line 30 so that we can identify them as inputs to the other two modules, *delay* and *lite\_ctrl*. The *enable* input is connected to the *change* output signal produced by the *delay* module. When enabled, the *light* state machine will advance to the next state when clocked as described by the CASE and nested IF statements on lines 34–39. Otherwise, *light* will hold at the current state.

The *lite\_ctrl* module (lines 42–57) inputs *lite[1..0]*, which represents the state of the *light* state machine from the control module, and will output the signals that will turn on the proper combinations of green, yellow, and red lights for the main and side roads. Each output from the *lite\_ctrl* module will actually be connected to lamp driver circuits to control the higher voltages and currents necessary for real lamps in a traffic light. The CASE statement on lines 47–55 determines which main road/side road light combination to turn on for each state of *light*. The function of the *lite\_ctrl* module is very much like a decoder. It essentially decodes each state combination of *lite* to turn on a green or yellow light for one road and a red light for the other road. A unique output combination is produced for each input state.

## VHDL TRAFFIC LIGHT CONTROLLER

The VHDL design for the traffic light controller is listed in Figure 7-65. The top level of the design is described structurally on lines 1–34. There are three COMPONENT modules to declare (lines 10–24). The PORT MAPs giving the wiring interconnects between each module and the top level design are listed on lines 26–33.

The *delay* module (lines 36–66) is basically a buried down counter (line 59) created with the integer variable *mach* that waits at zero when the main road has a green light (*lite* = “00”) until it is triggered by the car sensor (line 52) to load the delay factor *tmaingrn*–1 on line 53. Since the counter decrements all the way to zero, one is subtracted from each delay factor to make the delay counter’s modulus equal to the value of the delay factor. For example, if we wish to have a delay factor of 25, the counter must count from 24 down to 0. The actual length of time represented by the delay factors depends on the clock frequency. With a 1-Hz clock frequency, the period would be 1 s, and the delay factors would then be in seconds. Lines 62–64 define an output signal called *change* that detects when *mach* is equal to one. *Change* will be HIGH to indicate that the test condition is true, which in turn will enable the state machine in the control module to move to its next state (*lite* = “00”) when clocked to indicate a yellow light on the main road. When *mach* reaches zero now, CASE determines that *lite* has a new value and the fixed time delay factor of 5 for a yellow light is loaded (actually loading one less, as previously discussed) into *mach* (line 55) on the next clock. The count down continues from this new delay time, with *change* again enabling the control module to move to its next state (*lite* = “00”), resulting in a green light for the side road. When *mach* again reaches zero, the time delay (*tsidegrn*–1) for a green light on the side road will be loaded into the down counter (line 56). When *change* again goes active, *lite* will advance to “11” for a yellow light on the side road. *Mach* will recycle to the value 4 (5 – 1) on line 57 for the fixed time delay for a yellow light. When *change* goes active this time, the control module will return to *lite* = “00”

```

1  ENTITY traffic IS
2  PORT (   clock, car, reset           :IN BIT;
3          tmaingrn, tsidegrn         :IN INTEGER RANGE 0 TO 31;
4          lite                        :BUFFER INTEGER RANGE 0 TO 3;
5          change                      :BUFFER BIT;
6          mainred, mainyelo, maingrn  :OUT BIT;
7          sidered, sideyelo, sidegrn  :OUT BIT);
8  END traffic;
9  ARCHITECTURE toplevel OF traffic IS
10 COMPONENT delay
11     PORT ( clock, car, reset         :IN BIT;
12           lite                      :IN INTEGER RANGE 0 TO 3;
13           tmaingrn, tsidegrn       :IN INTEGER RANGE 0 TO 31;
14           change                   :OUT BIT);
15 END COMPONENT;
16 COMPONENT control
17     PORT ( clock, enable, reset      :IN BIT;
18           lite                      :OUT INTEGER RANGE 0 TO 3);
19 END COMPONENT;
20 COMPONENT lite_ctrl
21     PORT ( lite                     :IN INTEGER RANGE 0 TO 3;
22           mainred, mainyelo, maingrn :OUT BIT;
23           sidered, sideyelo, sidegrn :OUT BIT);
24 END COMPONENT;
25 BEGIN
26 module1: delay    PORT MAP (clock => clock, car => car, reset => reset,
27                          lite => lite, tmaingrn => tmaingrn, tsidegrn => tsidegrn,
28                          change => change);
29 module2: control  PORT MAP (clock => clock, enable => change, reset => reset,
30                          lite => lite);
31 module3: lite_ctrl PORT MAP (lite => lite, mainred => mainred, mainyelo => mainyelo,
32                          maingrn => maingrn, sidered => sidered, sideyelo => sideyelo,
33                          sidegrn => sidegrn);
34 END toplevel;
35 -----
36 ENTITY delay IS
37 PORT (   clock, car, reset           :IN BIT;
38         lite                        :IN BIT_VECTOR (1 DOWNT0 0);
39         tmaingrn, tsidegrn         :IN INTEGER RANGE 0 TO 31;
40         change                     :OUT BIT);
41 END delay;
42 ARCHITECTURE time OF delay IS
43 BEGIN
44     PROCESS (clock, reset)
45     VARIABLE mach                    :INTEGER RANGE 0 TO 31;
46     BEGIN
47     IF reset = '0' THEN mach := 0;
48     ELSIF (clock = '1' AND clock'EVENT) THEN -- with 1 Hz clock, times in seconds
49         IF mach = 0 THEN
50             CASE lite IS
51             WHEN "00"
52                 IF car = '0' THEN mach := 0; -- wait for car on side road
53                 ELSE mach := tmaingrn - 1; -- set time for main's green
54                 END IF;
55             WHEN "01" => mach := 5 - 1; -- set time for main's yellow
56             WHEN "10" => mach := tsidegrn - 1; -- set time for side's green
57             WHEN "11" => mach := 5 - 1; -- set time for side's yellow
58             END CASE;
59         ELSE mach := mach - 1; -- decrement timer counter
60         END IF;
61     END IF;

```

**FIGURE 7-65** VHDL design for traffic light controller.

```

62     IF mach = 1 THEN change <= '1';           -- change lights on control
63     ELSE change <= '0';
64     END IF;
65     END PROCESS;
66 END time;
67 -----
68 ENTITY control IS
69 PORT ( clock, enable, reset   :IN BIT;
70       lite                    :OUT BIT_VECTOR (1 DOWNTO 0));
71 END control;
72 ARCHITECTURE a OF control IS
73 TYPE enumerated IS (mgrn, myel, sgrn, syel); -- need 4 states for light combinations
74 BEGIN
75     PROCESS (clock, reset)
76     VARIABLE lights :enumerated;
77     BEGIN
78         IF reset = '0' THEN lights := mgrn;
79         ELSIF (clock = '1' AND clock'EVENT) THEN
80             IF enable = '1' THEN -- wait for enable to change light states
81                 CASE lights IS
82                     WHEN mgrn    => lights := myel;
83                     WHEN myel    => lights := sgrn;
84                     WHEN sgrn    => lights := syel;
85                     WHEN syel    => lights := mgrn;
86                 END CASE;
87             END IF;
88         END IF;
89         CASE lights IS -- patterns for light states
90             WHEN mgrn => lite <= "00";
91             WHEN myel => lite <= "01";
92             WHEN sgrn => lite <= "10";
93             WHEN syel => lite <= "11";
94         END CASE;
95     END PROCESS;
96 END a;
97 -----
98 ENTITY lite_ctrl IS
99 PORT ( lite                    :IN BIT_VECTOR (1 DOWNTO 0);
100      mainred, mainyelo, maingrn :OUT BIT;
101      sidered, sideyelo, sidegrn :OUT BIT);
102 END lite_ctrl;
103 ARCHITECTURE patterns OF lite_ctrl IS
104 BEGIN
105     PROCESS (lite)
106     BEGIN
107         CASE lite IS -- control state determines which lights to turn on/off
108             WHEN "00" => maingrn <= '1';    mainyelo <= '0';    mainred <= '0';
109                       sidegrn <= '0';    sideyelo <= '0';    sidered <= '1';
110             WHEN "01" => maingrn <= '0';    mainyelo <= '1';    mainred <= '0';
111                       sidegrn <= '0';    sideyelo <= '0';    sidered <= '1';
112             WHEN "10" => maingrn <= '0';    mainyelo <= '0';    mainred <= '1';
113                       sidegrn <= '1';    sideyelo <= '0';    sidered <= '0';
114             WHEN "11" => maingrn <= '0';    mainyelo <= '0';    mainred <= '1';
115                       sidegrn <= '0';    sideyelo <= '1';    sidered <= '0';
116         END CASE;
117     END PROCESS;
118 END patterns;

```

FIGURE 7-65 Continued

(green light on main). When *mach* decrements to its terminal state (zero) this time, lines 52–54 will determine by the status of the *car* sensor input whether to wait for another car or to load in the delay factor for a green light on main (*tmaingrn*–1) to start the cycle over again. The main road will receive a green light for at least this length of time, even if there is a continuous stream of cars on the side road. It is obvious that we could make improvements to this design, but that, of course, would also complicate the design further.

The control module (lines 68–96) contains a state machine named *lights* that will sequence through four enumerated states for the traffic light combinations. The four enumerated states for *lights* are *mgrn*, *myel*, *sgrn*, and *syel* (lines 73 and 76). Each state represents which road, main or side, is to receive a green or yellow light. The other road will have a red light. The *enable* input is connected to the *change* output signal produced by the delay module. When enabled, the *lights* state machine will advance to the next state when clocked, as described by the nested IF and CASE statements on lines 79–88. Otherwise, *lights* will hold at the current state. The bit patterns for output port *lite* have been specified for each state of *lights* with the CASE statement on lines 89–94 so that we can identify them as inputs to the other two modules, *delay* and *lite\_ctrl*.

The *lite\_ctrl* module (lines 98–118) inputs *lite*, which represents the state of the *lights* state machine from the control module, and will output the signals that will turn on the proper combinations of green, yellow, and red lights for the main and side roads. Each output from the *lite\_ctrl* module will actually be connected to lamp driver circuits to control the higher voltages and currents necessary for real lamps in a traffic light. The CASE statement on lines 107–116, invoked by the PROCESS when the *lite* input changes, determines which main road/side road light combination to turn on for each state of *lights*. The function of the *lite\_ctrl* module is very much like a decoder. It essentially decodes each state combination of *lite* to turn on a green or yellow light for one road and a red light for the other road. A unique output combination is produced for each input state.

## Choosing HDL Coding Techniques

By this time, you may be wondering why there are so many ways to describe logic circuits. If one way is easier than the others, why not just study that one? The answer, of course, is that each level of abstraction offers advantages over the others in certain cases. The structural method provides the most complete control over interconnections. The use of Boolean equations, truth tables, and PRESENT state/NEXT state tables allows us to describe the way data flows through the circuit using HDL. Finally, the behavioral method allows a more abstract description of the circuit's operation in terms of cause and effect. In practice, each source file may have portions that can be categorized under each level of abstraction. Choosing the right level when writing code is not an issue of right and wrong as much as it is an issue of style and preference.

There are also several ways to approach any task from a standpoint of choosing control structures. Should we use selected signal assignments or Boolean equations, IF/ELSE or CASE, sequential processes or concurrent statements, macrofunctions or megafunctions? Or should we write our own code? The answers to these questions ultimately define your personal strategy in solving the problem. Your preferences and the advantages you find in using one method over another will be established with practice and experience.



**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is the fundamental difference between a counter and a state machine?
2. What is the difference between describing a counter and describing a state machine in an HDL?
3. If the actual binary states for a state machine are not defined in the HDL code, how are they assigned?
4. What is the advantage of using state machine description?

**PART 1 SUMMARY**

1. In asynchronous (ripple) counters, the clock signal is applied to the LSB FF, and all other FFs are clocked by the output of the preceding FF.
2. A counter's MOD number is the number of stable states in its counting cycle; it is also the maximum frequency-division ratio.
3. The normal (maximum) MOD number of a counter is  $2^N$ . One way to modify a counter's MOD number is to add circuitry that will cause it to recycle before it reaches its normal last count.
4. Counters can be cascaded (chained together) to produce greater counting ranges and frequency-division ratios.
5. In a synchronous (parallel) counter, all of the FFs are simultaneously clocked from the input clock signal.
6. The maximum clock frequency for an asynchronous counter,  $f_{\max}$ , decreases as the number of bits increases. For a synchronous counter,  $f_{\max}$  remains the same, regardless of the number of bits.
7. A decade counter is any MOD-10 counter. A BCD counter is a decade counter that sequences through the 10 BCD codes (0–9).
8. A presettable counter can be loaded with any desired starting count.
9. An up/down counter can be commanded to count up or count down.
10. Logic gates can be used to decode (detect) any or all states of a counter.
11. The count sequence for a synchronous counter can be easily determined by using a PRESENT state/NEXT state table that lists all possible states, the flip-flop input control information, and the resulting NEXT states.
12. Synchronous counters with arbitrary counting sequences can be implemented by following a standard design procedure.
13. Counters with a specified modulus and features can be easily created with the LPM\_COUNTER megafunction using the MegaWizard Manager or with behavioral descriptions using an HDL.
14. All the features available on the various standard IC counter chips, such as asynchronous or synchronous loading or clearing, count enabling, and terminal count decoding, can be described using HDL. HDL counters can be easily modified for higher MOD numbers or changes in the active levels for controls.
15. Digital systems can be subdivided into smaller modules or blocks that can be interconnected as a hierarchical design.
16. State machines can be represented in HDL using descriptive names for each state rather than specifying a numeric sequence of states.

## PART 1 IMPORTANT TERMS

asynchronous (ripple) counter	conditional transition presettable counters	circuit excitation table
MOD number	parallel load	LPM_COUNTER VARIABLE
duty cycle	count enable	behavioral level of abstraction
glitches	multistage counters	hierarchical design state machine
synchronous (parallel) counters	cascading	Mealy model
transient state	decoding	Moore model
decade counter	PRESENT	MACHINE
BCD counter	state/NEXT state table	enumerated type
up counter	self-correcting counter	
down counter	sequential circuit design	
up/down counters	J-K excitation table	

## PART 2

### 7-15 REGISTER DATA TRANSFER

#### OUTCOME

*Upon completion of this section, you will be able to:*

- Categorize input/output modes for register transfer.

The various types of registers can be classified according to the manner in which data can be entered into the register for storage and the manner in which data are outputted from the register. The various classifications are listed below and illustrated in Figure 7-66.

1. Parallel in/parallel out (PIPO)
2. Serial in/serial out (SISO)
3. Parallel in/serial out (PISO)
4. Serial in/parallel out (SIPO)

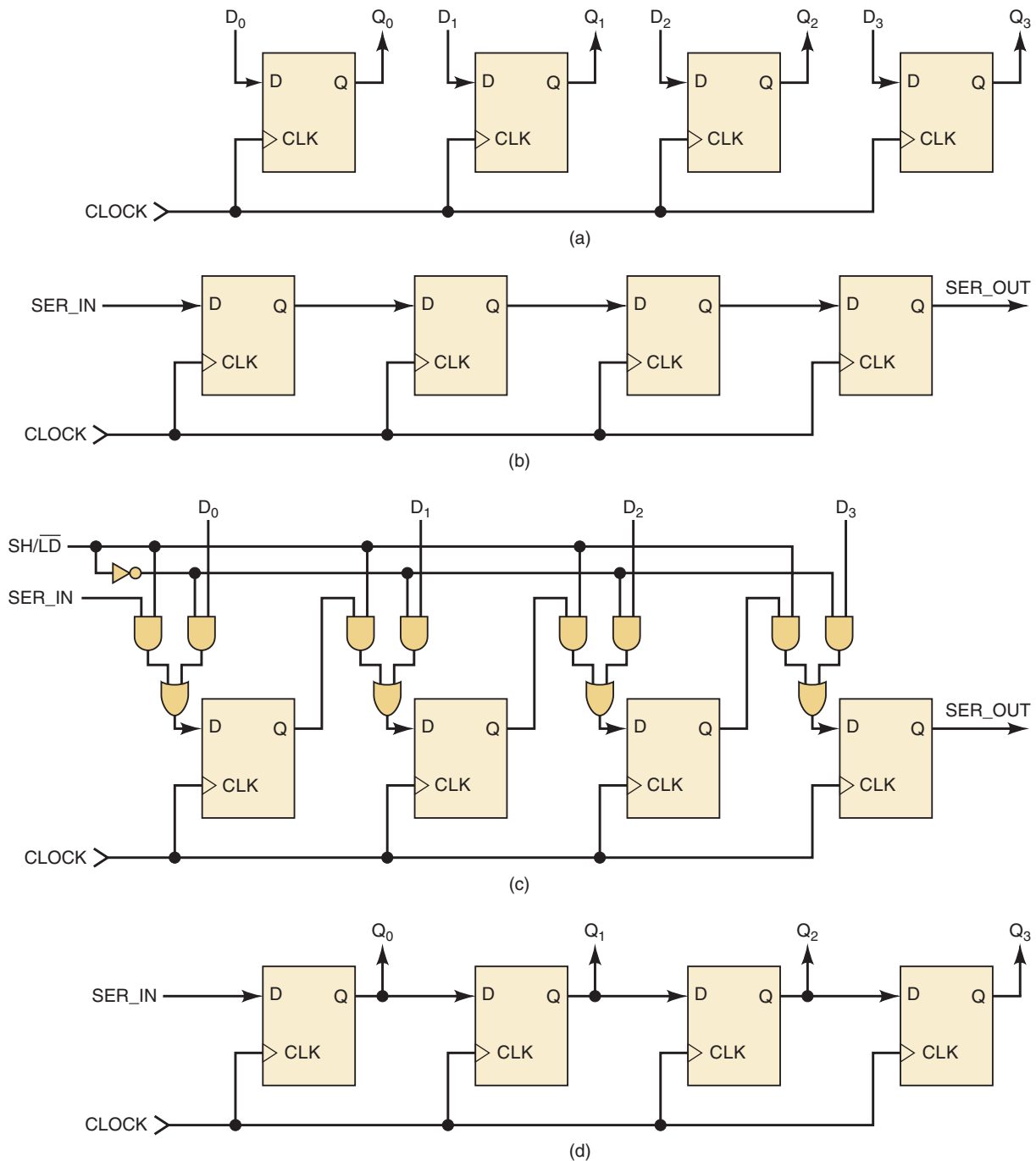
Serial data flow through a register is generally called shifting, and the data may be shifted either to the left or to the right. If the serial output data is fed back into the serial input of the same register, the operation is called a data rotate. Parallel inputting of data is often described as a register load. Many applications may have registers that are capable of a number of different types of data movement.

### 7-16 IC REGISTERS

#### OUTCOMES

*Upon completion of this section, you will be able to:*

- Define common controls for register operations.
- Predict the output for any combination of inputs.
- Represent register operations using timing diagrams.
- Decompose functional blocks for each register configuration to groups of flip-flops and control logic.

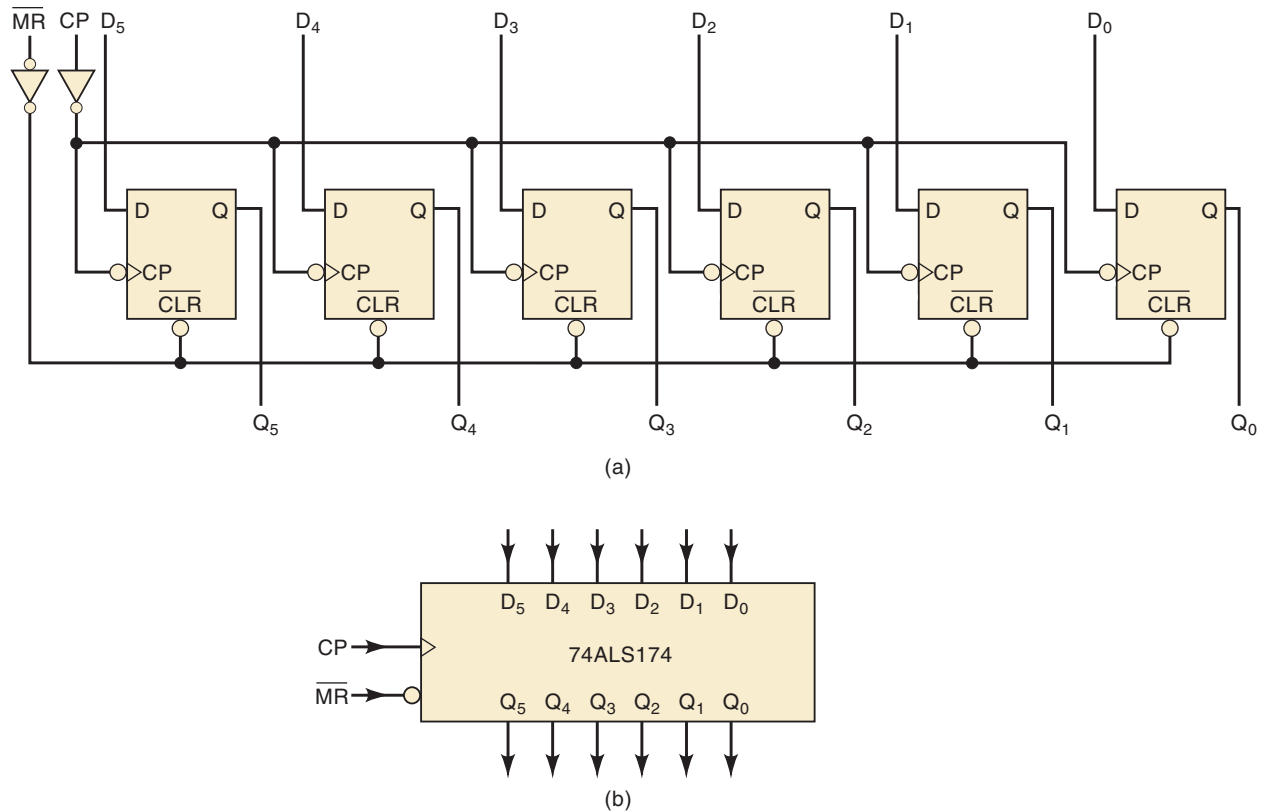


**FIGURE 7-66** Data transfer circuits: (a) PIPO; (b) SISO; (c) PISO; (d) SIPO.

Many standard IC register devices have been designed over the years. There are chips for each of the data flow classifications, and the chips can be cascaded together to create larger-sized registers. Usually, the logic designer could find exactly what is required for a given application. Let's examine a representative IC from each of the data flow categories.

### Parallel In/Parallel Out—The 74ALS174/74HC174

A group of flip-flops that can store multiple bits simultaneously and in which all bits of the stored binary value are directly available is referred to as a



**FIGURE 7-67** (a) Circuit diagram of the 74ALS174; (b) logic symbol.

**parallel in/parallel out register.** Figure 7-67(a) shows the logic diagram for the 74ALS174 (also the 74HC174), a six-bit register that has parallel inputs  $D_5$  through  $D_0$  and parallel outputs  $Q_5$  through  $Q_0$ . Parallel data are loaded into the register on the PGT of the clock input  $CP$ . A master reset input  $\overline{MR}$  can be used to reset asynchronously all of the register FFs to 0. The logic symbol for the 74ALS174 is shown in Figure 7-67(b). This symbol is used in circuit diagrams to represent the circuitry of Figure 7-67(a).

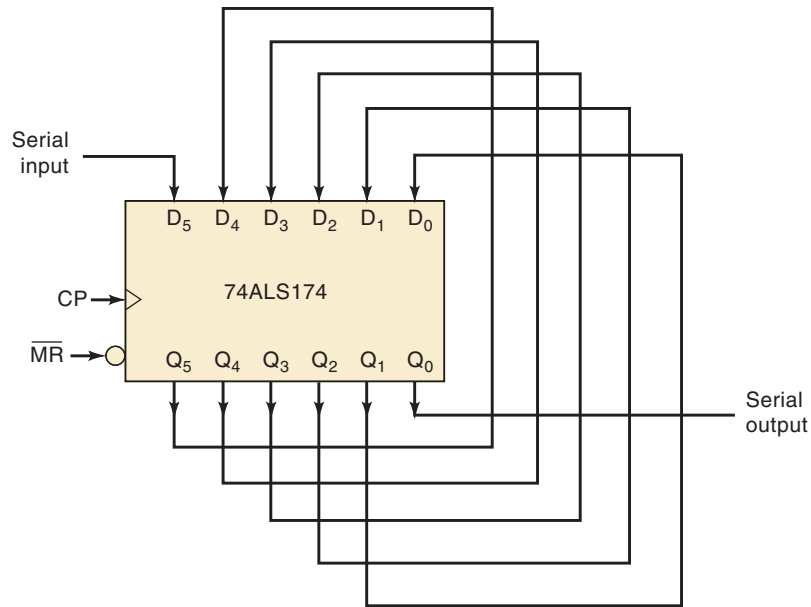
The 74ALS174 is normally used for synchronous parallel data transfer whereby the logic levels present at the  $D$  inputs are transferred to the corresponding  $Q$  outputs when a PGT occurs at the clock  $CP$ . This IC, however, can be wired for serial data transfer, as the following examples will show.

### EXAMPLE 7-18

Show how to connect the 74ALS174 so that it operates as a serial shift register with data shifting on each PGT of  $CP$  as follows: Serial input  $\rightarrow Q_5 \rightarrow Q_4 \rightarrow Q_3 \rightarrow Q_2 \rightarrow Q_1 \rightarrow Q_0$ . In other words, serial data will enter at  $D_5$  and will output at  $Q_0$ .

#### Solution

Looking at Figure 7-67(a), we can see that to connect the six FFs as a serial shift register, we have to connect the  $Q$  output of one to the  $D$  input of the next so that data is transferred in the required manner. Figure 7-68 shows how this is accomplished. Note that data shifts left to right, with input data applied at  $D_5$  and output data appearing at  $Q_0$ .



**FIGURE 7-68** Example 7-18: The 74ALS174 wired as a shift register.

### EXAMPLE 7-19

How would you connect two 74ALS174s to operate as a 12-bit shift register?

#### Solution

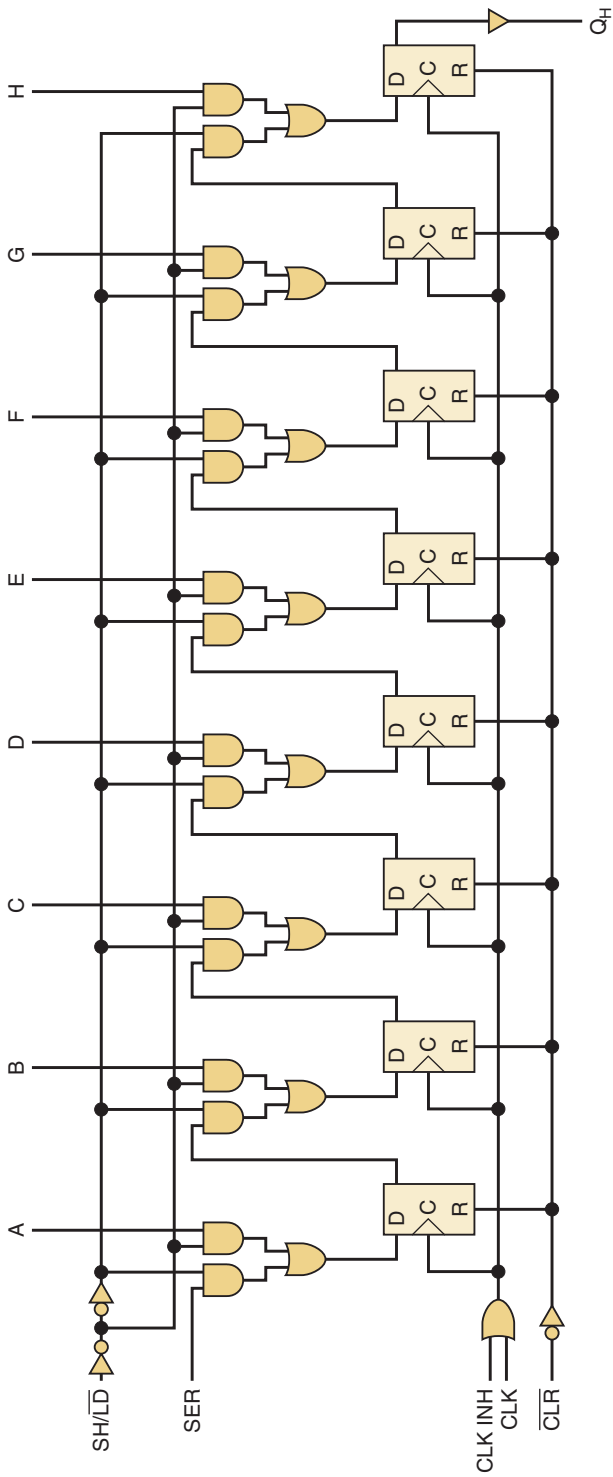
Connect a second 74ALS174 IC as a shift register, and connect  $Q_0$  from the first IC to  $D_5$  of the second IC. Connect the  $CP$  inputs of both ICs so that they will be clocked from the same signal. Also connect the  $MR$  inputs together if using the asynchronous reset.

### Serial In/Serial Out—The 74ALS166/74HC166

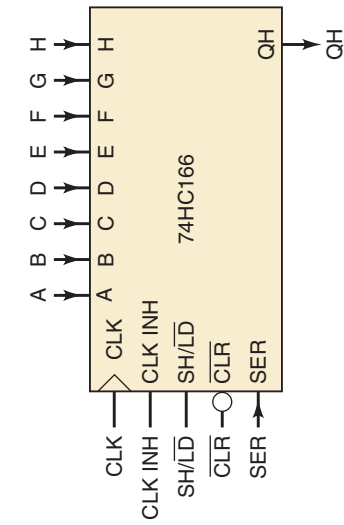
A **serial in/serial out** shift register will have data loaded into it one bit at a time. The data will move one bit at a time with each clock pulse through the set of flip-flops toward the other end of the register. With continued clocking, the data will exit the register one bit at a time in the same order as it was originally loaded. The 74HC166 (and also the 74ALS166) can be used as a serial in/serial out register. The logic diagram and schematic symbol for the 74HC166 is shown in Figure 7-69. It is an eight-bit shift register of which only FF QH is accessible. The serial data is input on  $SER$  and will be stored in FF QA. The serial output is obtained at the other end of the shift register on  $QH$ . As can be seen from the function table for this shift register in Figure 7-69(c), parallel data can also be synchronously loaded into it. If  $SH/\overline{LD} = 1$ , the register function will be serial shifting, while a LOW will instead parallel load data via the  $A$  through  $H$  inputs. The synchronous serial shifting and parallel loading functions can be inhibited (disabled) by applying a HIGH to the  $CLK\ INH$  control input. The register also has an active-LOW, asynchronous clear input ( $\overline{CLR}$ ).

### EXAMPLE 7-20

A shift register is often used as a way to delay a digital signal by an integral number of clock cycles. The digital signal is applied to the shift register's serial input and is shifted through the shift register by successive clock pulses until it reaches the end of the shift register, where it appears



(a)



(b)

INPUTS						OUTPUTS			
CLR	SH/LD	CLK INH	CLK	SER	PARALLEL A...H	INTERNAL		QH	
						QA	QB		
L	X	X	X	X	X	L	L	L	L
H	X	L	L	X	X	QA0	QB0	Qh0	h
H	L	L	↑	X	a...h	a	b	QAn	QAn
H	H	L	↑	H	X	H	L	L	L
H	H	L	↑	L	X	L	L	QA0	QA0
H	X	H	↑	X	X	QA0	QB0	Qh0	Qh0

(c)

**FIGURE 7-69** (a) Circuit diagram of the 74HC166; (b) logic symbol; (c) function table.

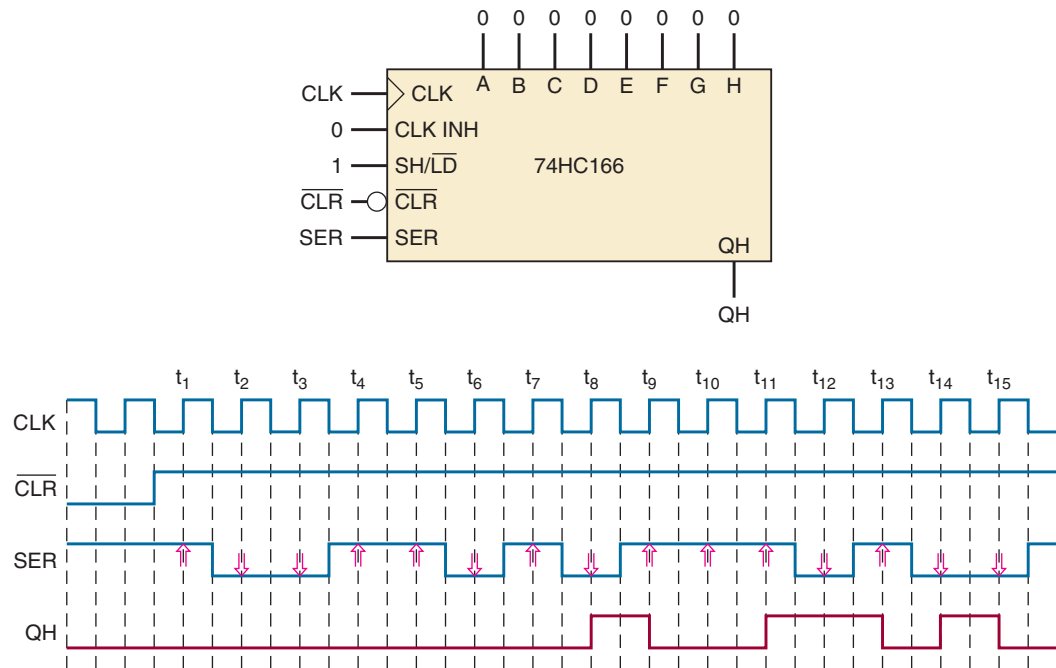


FIGURE 7-70 Example 7-20.

as the output signal. This method for delaying the effect of a digital signal is common in the digital communications field. For instance, the digital signal might be the digitized version of an audio signal that is to be delayed before it is transmitted. The input waveforms given in Figure 7-70 are applied to a 74HC166. Determine the resultant output waveform.

### Solution

$QH$  starts at a LOW, since all flip-flops are initially cleared by the LOW applied to the asynchronous  $\overline{CLR}$  input at the beginning of the timing diagram. At  $t_1$ , the shift register will input the current bit applied to  $SER$ . This will be stored in flip-flop QA. At  $t_2$ , the first bit will move to FF QB and a second bit on  $SER$  will be stored in QA. At  $t_3$ , the first bit will now move to FF QC and a third bit on  $SER$  will be stored in QA. The first data input bit will finally show up at the output  $QH$  at  $t_8$ . Each successive input bit on  $SER$  will follow at output  $QH$  delayed by eight clock cycles.

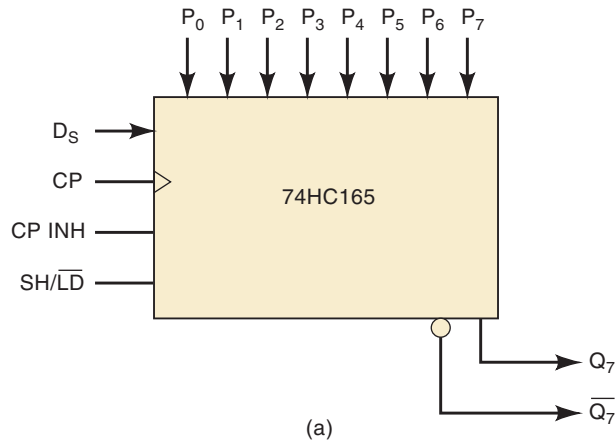
### Parallel In/Serial Out—The 74ALS165/74HC165

The logic symbol for the 74HC165 is shown in Figure 7-71(a). This IC is an eight-bit **parallel in/serial out** register. It actually has serial data entry via  $D_8$  and asynchronous parallel data entry via  $P_0$  through  $P_7$ . The register contains eight FFs— $Q_0$  through  $Q_7$ —internally connected as a shift register, but the only accessible FF outputs are  $Q_7$  and  $\overline{Q}_7$ .  $CP$  is the clock input used for the shifting operation. The clock inhibit input,  $CP\ INH$  is used to inhibit the effect of the  $CP$  input. The shift/load input,  $SH/\overline{LD}$ , controls which operation is taking place—shifting or parallel loading. The function table in Figure 7-71(b) shows how the various input combinations determine what operation, if any, is being performed. Parallel loading is asynchronous and serial shifting is synchronous. Note that the serial shifting function will always be synchronous, since the clock is required to ensure that the input data moves only one bit at a time with each appropriate clocking edge.

## EXAMPLE 7-21

Examine the 74HC165 function table and determine (a) the conditions necessary to load the register with parallel data; (b) the conditions necessary for the shifting operation.

**FIGURE 7-71** (a) Logic symbol for the 74HC165 parallel in/serial out register; (b) function table.



Function Table

Inputs			Operation
SH/ $\overline{LD}$	CP	CP INH	
L	X	X	Parallel load
H	H	X	No change
H	X	H	No change
H	$\mathcal{F}$	L	Shifting
H	L	$\mathcal{F}$	Shifting

H = high level  
L = low level  
X = immaterial  
 $\mathcal{F}$  = PGT

(b)

**Solution**

- (a) The first entry in the table shows that the  $SH/\overline{LD}$  input has to be LOW for the parallel load operation. When this input is LOW, the data present at the  $P$  inputs are *asynchronously* loaded into the register FFs, independent of the  $CP$  and the  $CP INH$  inputs. Of course, only the outputs from the last FF are externally available.
- (b) The shifting operation cannot take place unless the  $SH/\overline{LD}$  input is HIGH and a PGT occurs at  $CP$  while  $CP INH$  is LOW [see the fourth table entry in Figure 7-71(b)]. A HIGH at  $CP INH$  will inhibit the effect of any clock pulses. Note that the roles of the  $CP$  and  $CP INH$  inputs can be reversed, as indicated by the last table entry, because these two signals are ORed together inside the IC.

## EXAMPLE 7-22

Determine the output signal at  $Q_7$  if we connect a 74HC165 with  $D_S = 0$  and  $CP INH = 0$  and then apply the input waveforms given in Figure 7-72.  $P_0$ – $P_7$  represent the parallel data on  $P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7$ .

**Solution**

We have drawn the timing diagram for all eight FFs so that we could track their contents over time even though only  $Q_7$  will be accessible. The parallel load is asynchronous and will occur as soon as  $SH/\overline{LD}$  goes LOW. After



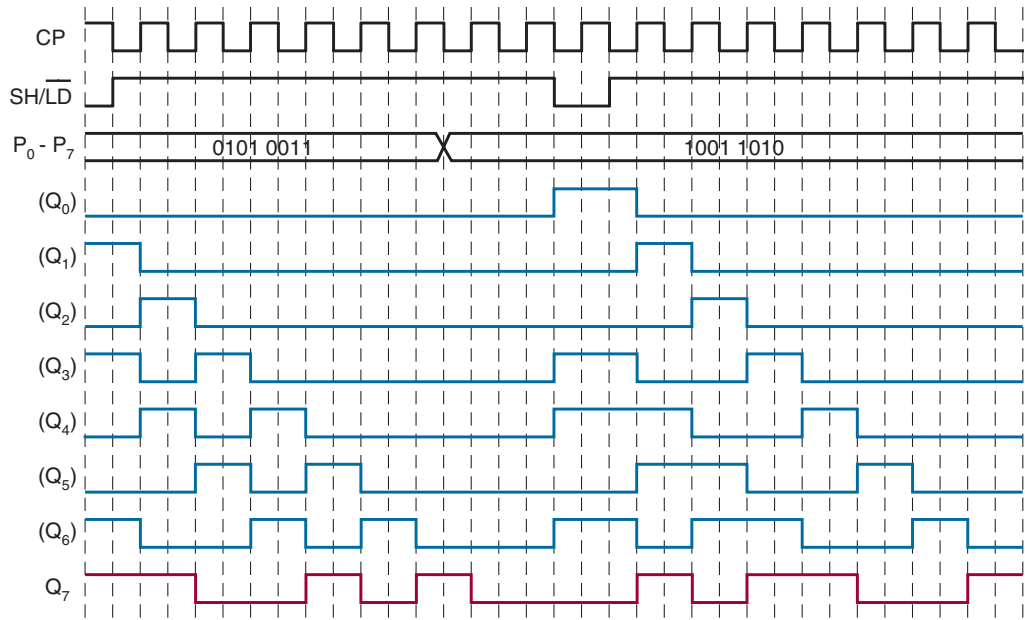


FIGURE 7-72 Example 7-22.

$SH/\overline{LD}$  returns to a HIGH, the data stored in the register will move one FF to the right (toward  $Q_7$ ) with each PGT on  $CP$ .

### Serial In/Parallel Out—The 74ALS164/74HC164

The logic diagram for the 74ALS164 is shown in Figure 7-73(a). It is an eight-bit serial in/parallel out shift register with each FF output externally accessible. Instead of a single serial input, an AND gate combines inputs  $A$  and  $B$  to produce the serial input to flip-flop  $Q_0$ .

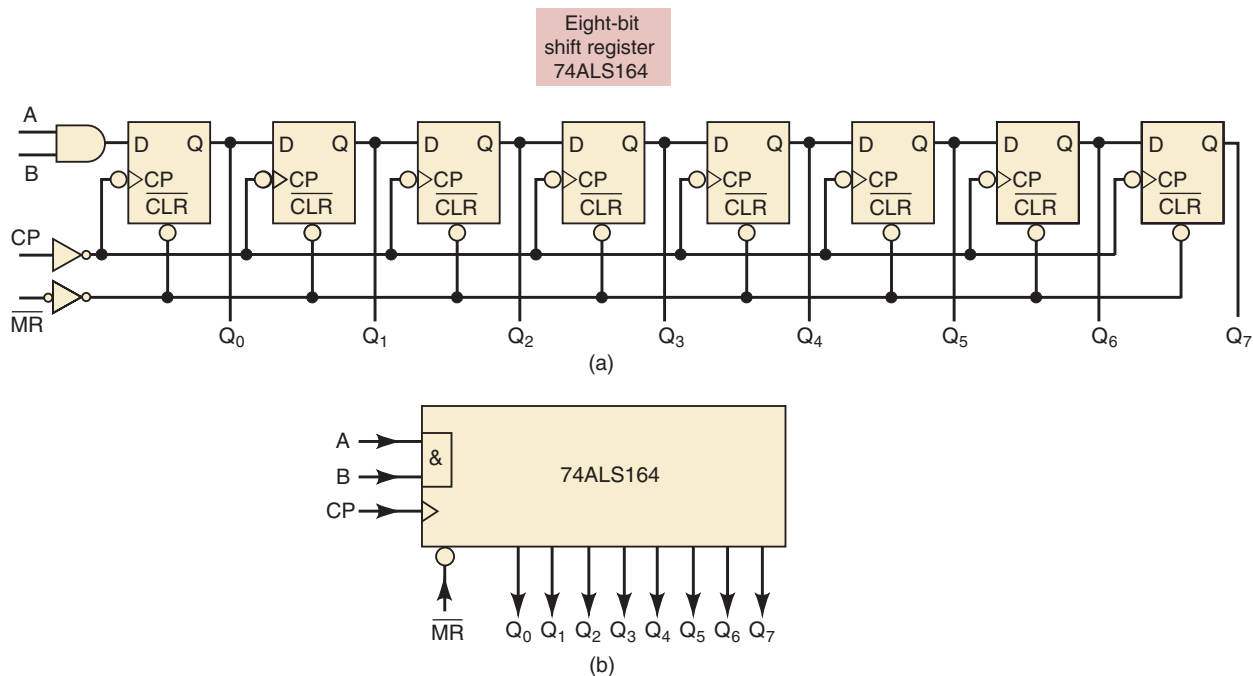


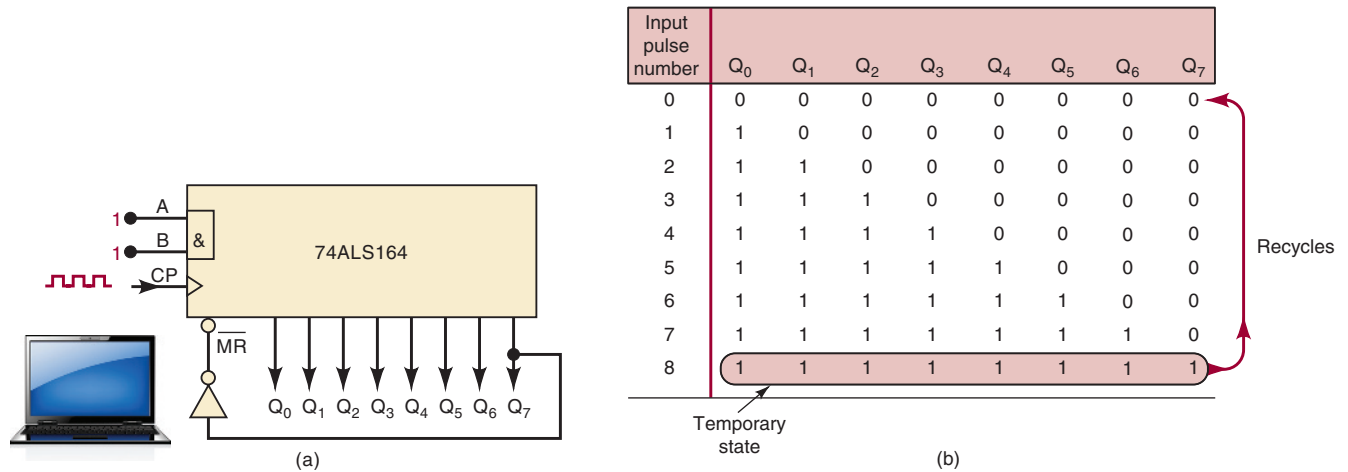
FIGURE 7-73 (a) Logic diagram for the 74ALS164; (b) logic symbol.

The shift operation occurs on the PGTs of the clock input  $CP$ . The  $\overline{MR}$  input provides asynchronous resetting of all FFs on a LOW level.

The logic symbol for the 74ALS164 is shown in Figure 7-73(b). Note that the & symbol is used inside the block to indicate that the  $A$  and  $B$  inputs are ANDed inside the IC and the result is applied to the  $D$  input of FF  $Q_0$ .

**EXAMPLE 7-23**

Assume that the initial contents of the 74ALS164 register in Figure 7-74(a) are 00000000. Determine the sequence of states as clock pulses are applied.



**FIGURE 7-74** Example 7-23.

**Solution**

The correct sequence is given in Figure 7-74(b). With  $A = B = 1$ , the serial input is 1, so that 1s will shift into the register on each PGT of  $CP$ . Because output  $Q_7$  is initially at 0, the  $\overline{MR}$  input is inactive.

On the eighth pulse, the register tries to go to the 11111111 state as the 1 from FF  $Q_6$  shifts into FF  $Q_7$ . This state occurs only momentarily because  $Q_7 = 1$  produces a LOW at  $\overline{MR}$  that immediately resets the register back to 00000000. The sequence is then repeated on the next eight clock pulses.

**OUTCOME ASSESSMENT QUESTIONS**

1. What kind of register can have a complete binary number loaded into it in one operation, and then have it shifted out one bit at a time?
2. *True or false:* A serial in/parallel out register can have all of its bits displayed at one time.
3. What type of register can have data entered into it only one bit at a time, but has all data bits available as outputs?
4. In what type of register do we store data one bit at a time and have access to only one output bit at a time?
5. How does the parallel data entry differ for the 74165 and the 74174?
6. How does the  $CP\ INH$  input of the 74ALS165 work?
7. In Figure 7-72 exchange the values on  $P_0$ – $P_7$  i.e. swap values 01010011 and 10011010 and redraw the output timing of each  $Q$ .

## 7-17 SHIFT-REGISTER COUNTERS

### OUTCOMES

Upon completion of this section, you will be able to:

- Define common terms, ring counter, and Johnson counter.
- State characteristics of ring and Johnson counters.

In Section 5-17, we saw how to connect FFs in a shift-register arrangement to transfer data left to right, or vice versa, one bit at a time (serially). Shift-register counters use *feedback*, which means that the output of the last FF in the register is connected back to the first FF in some way.

### Ring Counter

The simplest shift-register counter is essentially a **circulating shift register** connected so that the last FF shifts its value into the first FF. This arrangement is shown in Figure 7-75 using D-type FFs (J-K flip-flops can also be used). The FFs are connected so that information shifts from left to right and back around from  $Q_0$  to  $Q_3$ . In most instances, only a single 1 is in the register, and it is made to circulate around the register as long as clock pulses are applied. For this reason, it is called a **ring counter**.

The waveforms, sequence table, and state diagram in Figure 7-75 show the various states of the FFs as pulses are applied, assuming a starting state of  $Q_3 = 1$  and  $Q_2 = Q_1 = Q_0 = 0$ . After the first pulse, the 1 has shifted from  $Q_3$  to  $Q_2$  so that the counter is in the 0100 state. The second pulse produces the 0010 state, and the third pulse produces the 0001 state. On the *fourth* clock pulse, the 1 from  $Q_0$  is transferred to  $Q_3$ , resulting in the 1000 state, which is, of course, the initial state. Subsequent pulses cause the sequence to repeat.

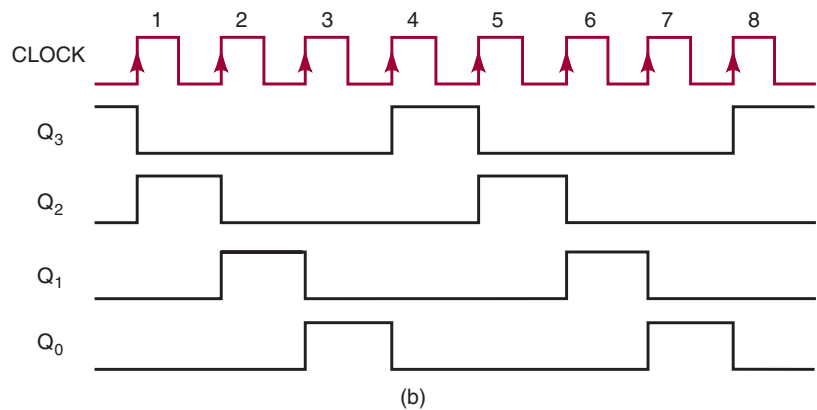
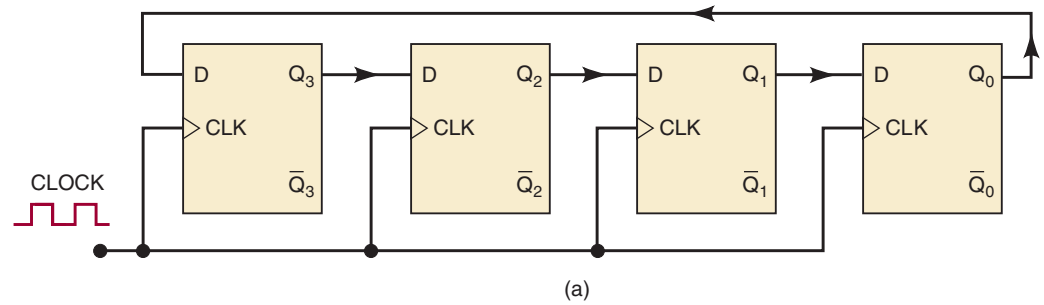
This counter functions as a MOD-4 counter because it has *four* distinct states before the sequence repeats. Although this circuit does not progress through the normal binary counting sequence, it is still a counter because each count corresponds to a unique set of FF states. Note that each FF output waveform has a frequency equal to one-fourth of the clock frequency because this is a MOD-4 ring counter.

Ring counters can be constructed for any desired MOD number; a MOD- $N$  ring counter uses  $N$  flip-flops connected in the arrangement of Figure 7-75. In general, a ring counter requires more FFs than a binary counter for the same MOD number; for example, a MOD-8 ring counter requires eight FFs, while a MOD-8 binary counter requires only three.

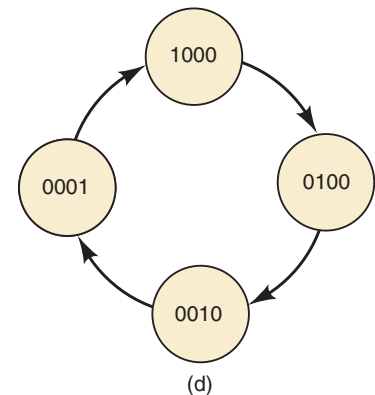
Despite the fact that it is less efficient in the use of FFs, a ring counter is still useful because it can be decoded without the use of decoding gates. The decoding signal for each state is obtained at the output of its corresponding FF. Compare the FF waveforms of the ring counter with the decoding waveforms in Figure 7-20. In some cases, a ring counter might be a better choice than a binary counter with its associated decoding gates. This is especially true in applications where the counter is being used to control the sequencing of operations in a system.

### Starting a Ring Counter

To operate properly, a ring counter must start off with only one FF in the 1 state and all the others in the 0 state. Because the starting states of the FFs will be unpredictable on power-up, the counter must be preset to the required starting state before clock pulses are applied. One way to do this is to apply a momentary pulse to the asynchronous  $\overline{PRE}$  input of one of the FFs (e.g.,  $Q_3$  in



Q <sub>3</sub>	Q <sub>2</sub>	Q <sub>1</sub>	Q <sub>0</sub>	CLOCK pulse
1	0	0	0	0
0	1	0	0	1
0	0	1	0	2
0	0	0	1	3
1	0	0	0	4
0	1	0	0	5
0	0	1	0	6
0	0	0	1	7
.	.	.	.	.
.	.	.	.	.



**FIGURE 7-75** (a) Four-bit ring counter; (b) waveforms; (c) sequence table; (d) state diagram.

Figure 7-75) and to the  $\overline{CLR}$  input of all other FFs. Another method is shown in Figure 7-76. On power-up, the capacitor will charge up relatively slowly toward  $+V_{CC}$ . The output of Schmitt-trigger INVERTER 1 will stay HIGH, and the output of INVERTER 2 will remain LOW until the capacitor voltage exceeds the positive-going threshold voltage ( $V_{T+}$ ) of the INVERTER 1 input (about 1.7 V). This will hold the  $\overline{PRE}$  input of  $Q_3$  and the  $\overline{CLR}$  inputs of  $Q_2$ ,  $Q_1$ , and  $Q_0$  in the LOW state long enough during power-up to ensure that the counter starts at 1000.

### Johnson Counter

The basic ring counter can be modified slightly to produce another type of shift-register counter, which will have somewhat different properties. The



that the *inverse* of the level stored in  $Q_0$  will be transferred to  $Q_2$  on the clock pulse.

The Johnson-counter operation is easy to analyze if we realize that on each positive clock-pulse transition, the level at  $Q_2$  shifts into  $Q_1$ , the level at  $Q_1$  shifts into  $Q_0$ , and the *inverse* of the level at  $Q_0$  shifts into  $Q_2$ . Using these ideas and assuming that all FFs are initially 0, the waveforms, sequence table, and state diagram of Figure 7-77 can be generated.

Examination of the waveforms and sequence table reveals the following important points:

1. This counter has six distinct states—000, 100, 110, 111, 011, and 001—before it repeats the sequence. Thus, it is a MOD-6 Johnson counter. Note that it does not count in a normal binary sequence.
2. The waveform of each FF is a square wave (50% duty cycle) at one-sixth the frequency of the clock. In addition, the FF waveforms are shifted by one clock period with respect to each other.

The MOD number of a Johnson counter will always be equal to *twice* the number of FFs. For example, if we connect five FFs in the arrangement of Figure 7-77, the result is a MOD-10 Johnson counter, where each FF output waveform is a square wave at one-tenth the clock frequency. Thus, it is possible to construct a MOD- $N$  counter (where  $N$  is an even number) by connecting  $N/2$  flip-flops in a Johnson-counter arrangement.

### Decoding a Johnson Counter

For a given MOD number, a Johnson counter requires only half the number of FFs that a ring counter requires. However, a Johnson counter requires decoding gates, whereas a ring counter does not. As in the binary counter, the Johnson counter uses one logic gate to decode for each count, but each gate requires only two inputs, regardless of the number of FFs in the counter. Figure 7-78 shows the decoding gates for the six states of the Johnson counter of Figure 7-77.

Notice that each decoding gate has only two inputs, even though there are three FFs in the counter, because for each count, two of the three FFs are in a unique combination of states. For example, the combination  $Q_2 = Q_0 = 0$  occurs only once in the counting sequence, at the count of 0. Thus, AND gate 0, with inputs  $\bar{Q}_2$  and  $\bar{Q}_0$ , can be used to decode for this count. This same characteristic is shared by all of the other states in the

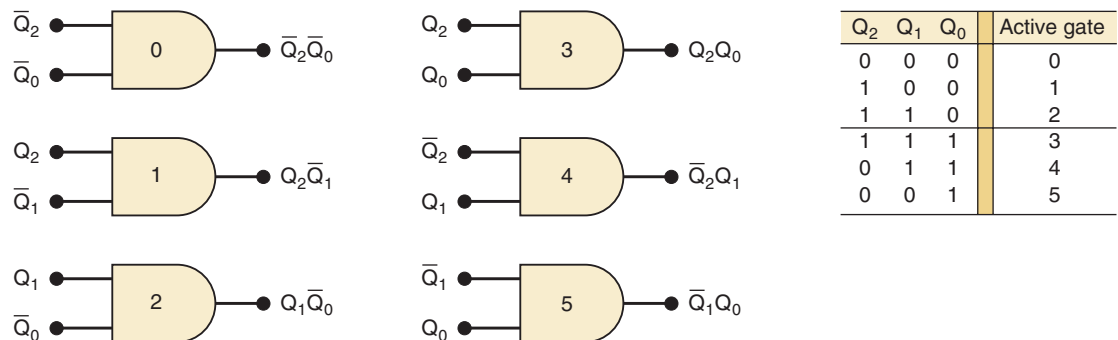


FIGURE 7-78 Decoding logic for a MOD-6 Johnson counter.

sequence, as the reader can verify. In fact, for *any size* Johnson counter, the decoding gates will have only two inputs.

Johnson counters represent a middle ground between ring counters and binary counters. A Johnson counter requires fewer FFs than a ring counter but generally more than a binary counter; it has more decoding circuitry than a ring counter but less than a binary counter. Thus, it sometimes represents a logical choice for certain applications.

### IC Shift-Register Counters

Very few ring counters or Johnson counters are available as ICs because it is relatively simple to take a shift-register IC and to wire it as either a ring counter or a Johnson counter. Some of the CMOS Johnson-counter ICs (74HC4017, 74HC4022) include the complete decoding circuitry on the same chip as the counter.

#### OUTCOME ASSESSMENT QUESTIONS

1. Which shift-register counter requires the most FFs for a given MOD number?
2. Which shift-register counter requires the most decoding circuitry?
3. How can a ring counter be converted to a Johnson counter?
4. *True or false:*
  - (a) The outputs of a ring counter are always square waves.
  - (b) The decoding circuitry for a Johnson counter is simpler than for a binary counter.
  - (c) Ring and Johnson counters are synchronous counters.
4. How many FFs are needed in a MOD-16 ring counter? How many are needed in a MOD-16 Johnson counter?

## 7-18 TROUBLESHOOTING

### OUTCOME

*Upon completion of this section, you will be able to:*

- Improve analytical skills for troubleshooting.

Flip-flops, counters, and registers are the major components in **sequential logic systems**. A sequential logic system, because of its storage devices, has the characteristic that its outputs and sequence of operations depend on both the present inputs and the inputs that occurred earlier. Even though sequential logic systems are generally more complex than combinational logic systems, the essential procedures for troubleshooting apply equally well to both types of systems. Sequential systems suffer from the same types of failures (open circuits, shorts, internal IC faults, and the like) as do combinational systems.

Many of the same steps used to isolate faults in a combinational system can be applied to sequential systems. One of the most effective troubleshooting techniques begins with the troubleshooter observing the system operation and, by analytical reasoning, determining the possible causes of the system malfunction. Then he or she uses available test instruments to

isolate the exact fault. The following examples will show the kinds of analytical reasoning that should be the initial step in troubleshooting sequential systems. After studying these examples, you should be ready to tackle the troubleshooting problems at the end of the chapter.

### EXAMPLE 7-24

Figure 7-79(a) shows a 74ALS161 wired as a MOD-12 counter, but it produces the count sequence given in Figure 7-79(b). Determine the cause of the incorrect circuit behavior.

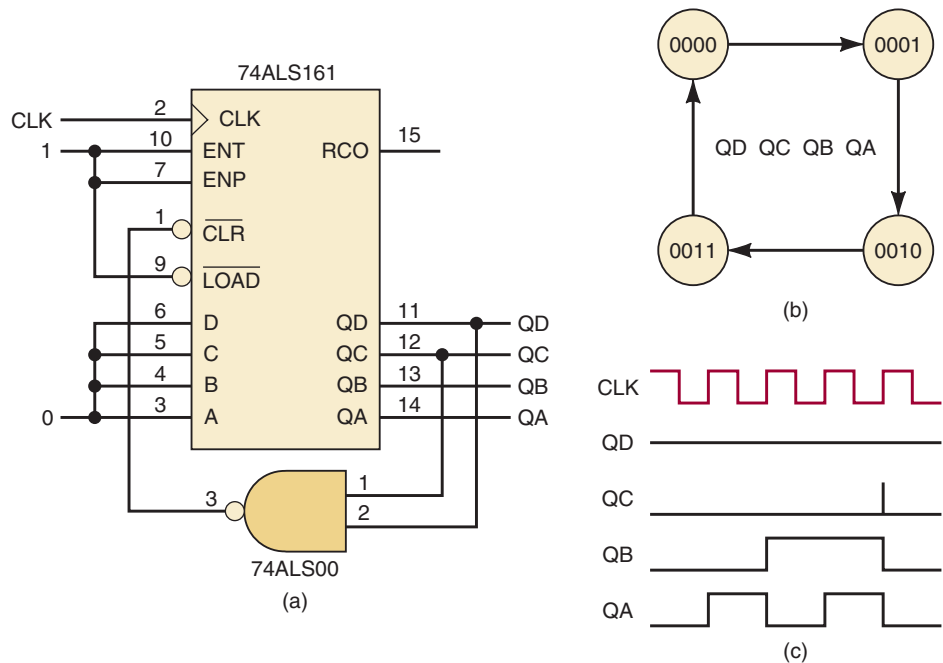


FIGURE 7-79 Example 7-24.

### Solution

Outputs  $QB$  and  $QA$  seem to be operating correctly but  $QC$  and  $QD$  stay LOW. Our first choice for the fault is that  $QC$  is shorted to ground, but an ohmmeter check does not confirm this. The 74ALS161 might have an internal fault that prevents it from counting above 0011. We try removing the 7400 NAND chip from its socket and shorting the  $\overline{CLR}$  pin to a HIGH. The counter now counts a regular MOD-16 sequence, so at least the counter's outputs seem to be ok. Next we decide to look at the  $\overline{CLR}$  pin with the NAND reconnected. Using a logic probe with its "pulse capture" turned on shows us that the  $\overline{CLR}$  pin is receiving pulses. Connecting a scope to the outputs, we see that the counter produces the waveforms shown in Figure 7-79(c). A glitch is observed on  $QC$  when the counter should be going to state 0100. That indicates that 0100 is a transient state when the transient state should actually be 1100. The  $QD$  connection to the NAND gate is now suspected, so we use the logic probe to check pin 2. There is no logic signal at all indicated on pin 2, which now leads us to the conclusion that the fault is an open between the  $QD$  output and pin 2 on the NAND. The NAND input is floating HIGH, causing the circuit to detect state 0100 instead of 1100 as it should be doing.



## EXAMPLE 7-25

A technician receives a “trouble ticket” for a circuit board that says the variable frequency divider operates “sometimes.” Sounds like a dreaded intermittent fault problem—often the hardest problems to find! His first thought is to send it back with the note “Use only when operating correctly!” but he decides to investigate further since he feels up to a good challenge today. The schematic for the circuit block is shown in Figure 7-80. The desired divide-by factor is applied to input  $f[7..0]$  in binary. The eight-bit counter counts down from this number until it reaches zero and then asynchronously loads in  $f[7..0]$  again, making zero a transient state. The resulting modulus will be equal to the value on  $f[7..0]$ . The output frequency signal is obtained by decoding state 00000001, making the frequency of  $out$  equal to the frequency of  $in$  divided by the binary value  $f[7..0]$ . In the application, the frequency of  $in$  is 100 kHz. Change  $f[7..0]$  and a new frequency will be output.

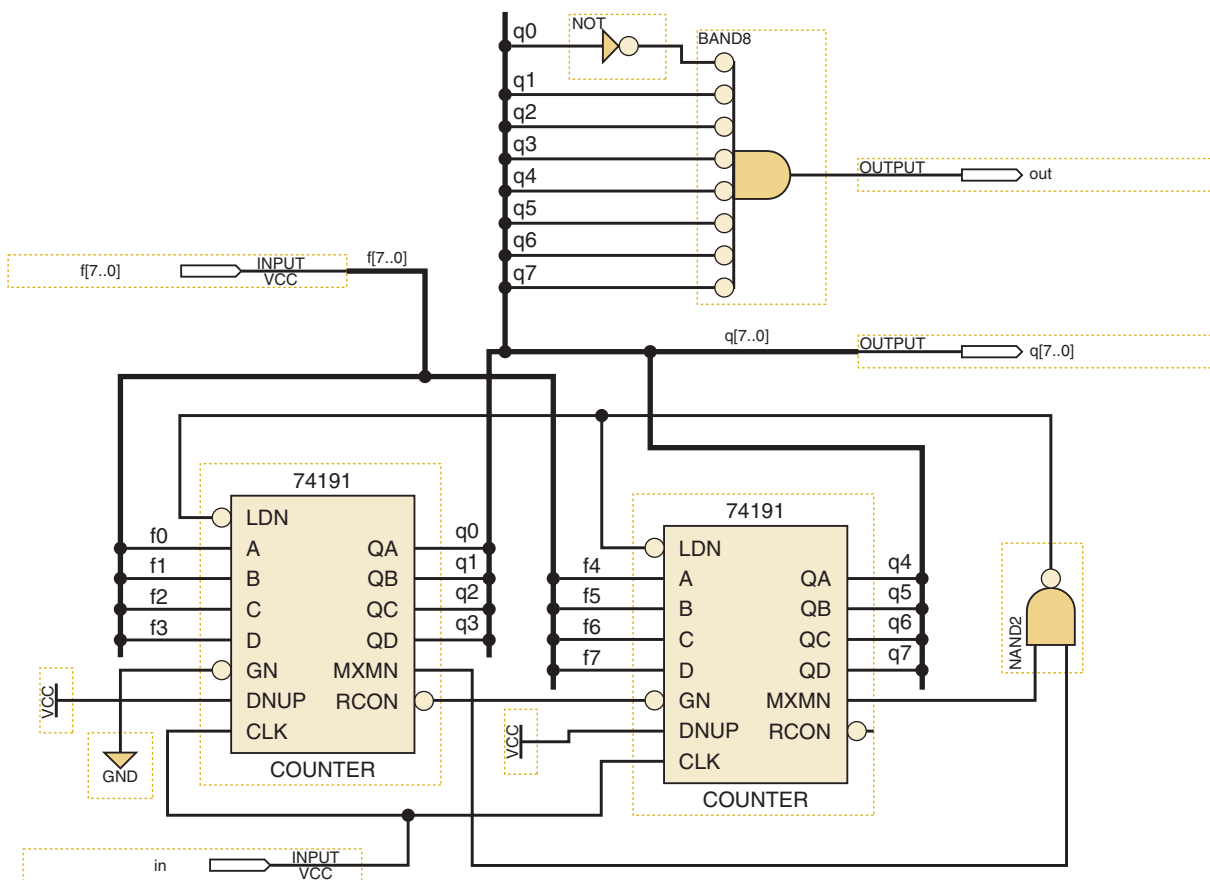


FIGURE 7-80 Example 7-25.

### Solution

The technician decides that he needs to obtain some test results to look at. He picks some easy divide-by factors to apply to  $f$  and records the results listed in Table 7-9.

He observes that the circuit produces correct results for some test cases but incorrect results for others. The problem does not seem to be

**TABLE 7-9** Measured output frequencies.

$f[ ]$ (decimal)	$f[ ]$ (binary)	Measured $f_{out}$	OK?
255	11111111	398.4 Hz	
240	11110000	416.7 Hz	✓
200	11001000	500.0 Hz	✓
100	01100100	1041.7 Hz	
50	00110010	2000.0 Hz	✓
25	00011001	4000.0 Hz	✓
15	00001111	9090.9 Hz	

intermittent after all. Instead, it appears to be dependent on the value for  $f$ . The technician decides to calculate the relationship between input and output frequencies for the three tests that failed and obtains the following:

$$100 \text{ kHz}/398.4 \text{ Hz} = 251$$

$$100 \text{ kHz}/1041.7 \text{ Hz} = 96$$

$$100 \text{ kHz}/9090.9 \text{ Hz} = 11$$

Each failure seems to be a divide-by factor that is four less than the value that was actually applied to the input. After looking again at the binary representation for  $f$ , he notes that every failure occurred when  $f_2 = 1$ . The weight for that bit, of course, is four. Eureka! That bit doesn't seem to be getting in—time for a logic-probe test on the  $f_2$  pin. Sure enough, the logic probe indicates the pin is LOW regardless of the value for  $f_2$ .

## 7-19 MEGAFUNCTION REGISTERS

### OUTCOMES

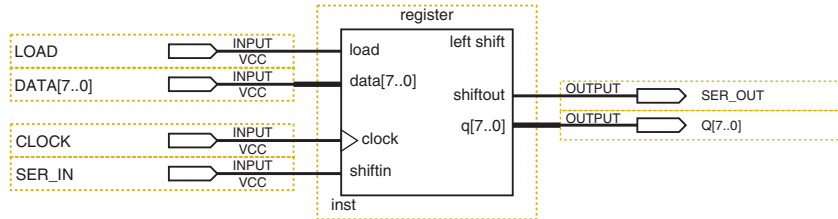
*Upon completion of this section, you will be able to:*

- Locate Altera's Library of Parameterized modules for common shift-register applications.
- Specify parameters to meet the needs of a system.

The Quartus II maxplus2 library also contains functionally equivalent versions of "old-style" MSI register chips such as the examples discussed in Section 7-16. A much easier schematic option for implementing registers in designs is available with the megafunction **LPM\_SHIFTREG** (found in the MegaWizard Manager's Plug-Ins Storage folder).

Figure 7-81(a) shows an example of an eight-bit multipurpose shift register created with the LPM\_SHIFTREG megafunction. All four categories of data movement can be accomplished with this register. Data may be input either serially or in parallel, and the data output is available either serially or in parallel. Serial shifting is to the left or toward  $Q_7$  with new data being input to  $Q_0$  with each PGT of  $CLOCK$ . SISO operation for this register is simulated in Figure 7-81(b). Serial input data is applied to  $SER\_IN$  and the serial output is available on  $SER\_OUT$  (or  $Q[7]$ ). Since it is an eight-bit register, it will require eight clock pulses to serially load eight new data bits into the register, after which the data would be available

for parallel output for SIPO register operation. Figure 7-81(c) illustrates PISO register operation with  $SER\_IN = 0$ , while Figure 7-81(d) serially inputs a 1. New parallel data is loaded synchronously when  $LOAD = 1$ . Since all register outputs ( $Q[7..0]$ ) are available, the register can also be used for PIPO register operation.



**LPM parameter decisions:**

How wide should the 'q' output bus be? **8**

Which direction do you want the register to shift? **Left**

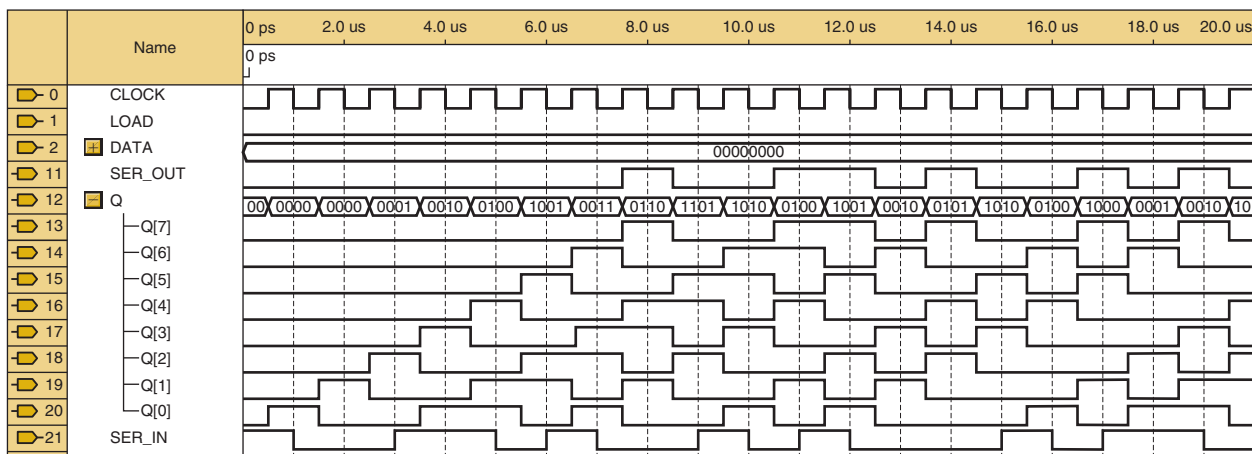
Which outputs do you want (select at least one)?

Select both: **Data output**  
**Serial shift data output**

Do you want any optional inputs?

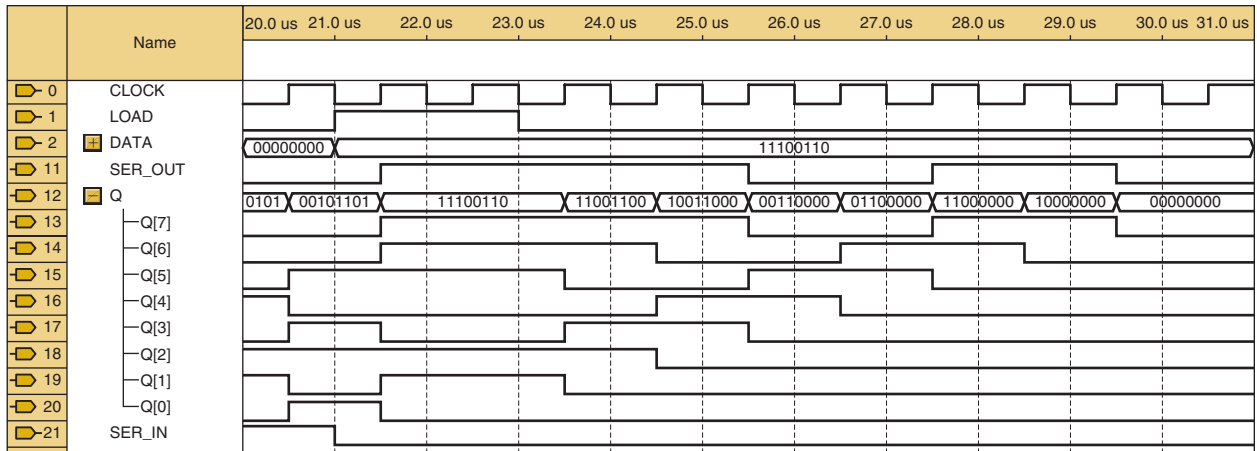
Select **Serial shift data input**  
**Parallel data input (load)**

(a)

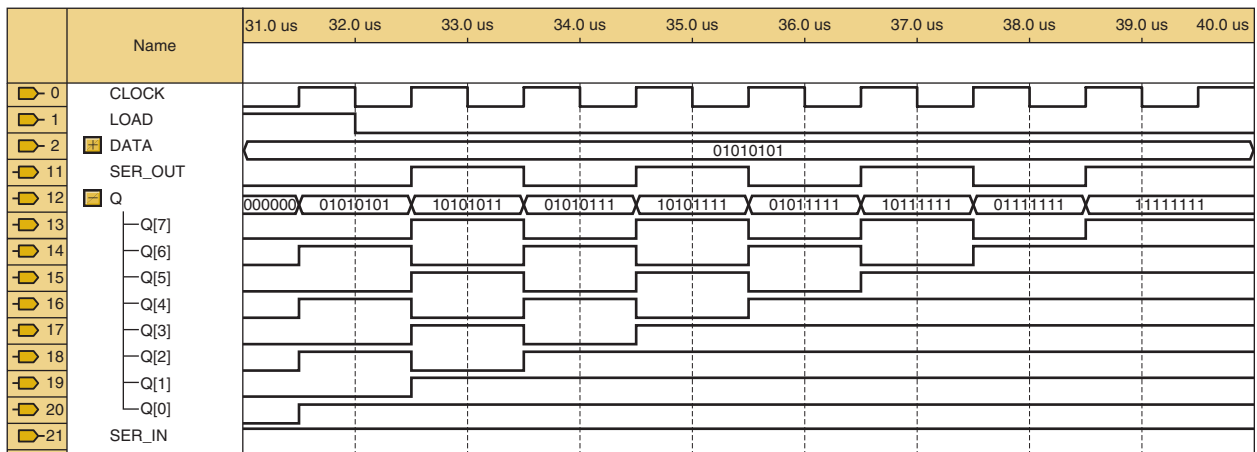


(b)

**FIGURE 7-81** Multipurpose shift register: (a) block diagram & MegaWizard settings; (b) functional simulation results.



(c)



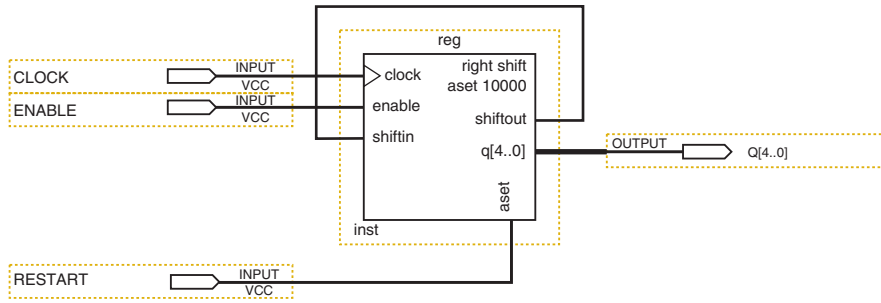
(d)

**FIGURE 7-81** (Continued) Multipurpose shift register: (c), (d) functional simulation results.**EXAMPLE 7-26**

Design a MOD-5 ring counter using LPM\_SHIFTREG. Use an asynchronous control to restart the ring counter at 10000 so that it will begin counting in the proper sequence and include an active-HIGH count enable control.

**Solution**

A MOD-5 ring counter will require a five-bit shift register with the serial output fed back into the serial input. The megafunction LPM\_SHIFTREG was used to implement the shift register shown in Figure 7-82(a). The *aset* (asynchronous set) control with a constant data input value of 10000 is used to restart the ring counter. The functional simulation results are shown in Figure 7-82(b).



**LPM parameter decisions:**

How wide should the 'q' output bus be? **5**

Which direction do you want the register to shift? **Right**

Which outputs do you want (select at least one)?

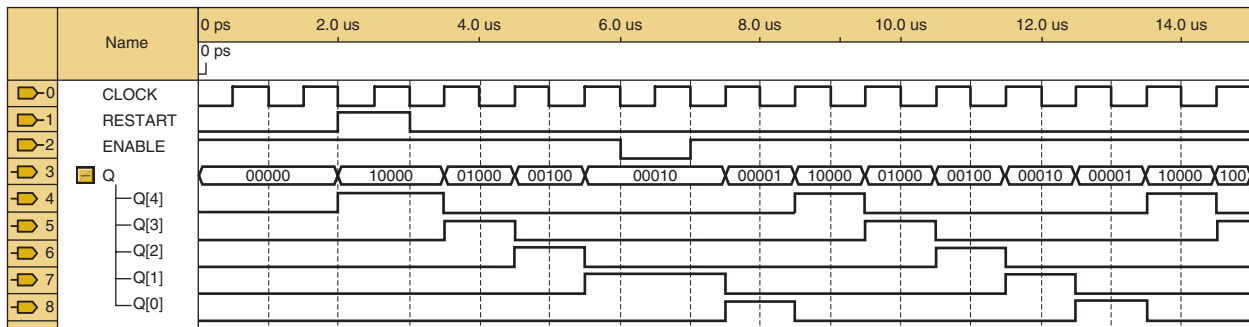
Select both: **Data output**  
**Serial shift data output**

Do you want any optional inputs?

Select **Clock Enable input**  
**Serial shift data input**

Asynchronous inputs: Select **Set to 10000**

(a)



(b)

**FIGURE 7-82** MOD-5 ring counter: (a) block diagram & MegaWizard settings; (b) simulation results.

**OUTCOME ASSESSMENT QUESTION**

1. Which classifications for data movement can be implemented by a shift register using an LPM\_COUNTER megafunction?

## 7-20 HDL REGISTERS

### OUTCOME

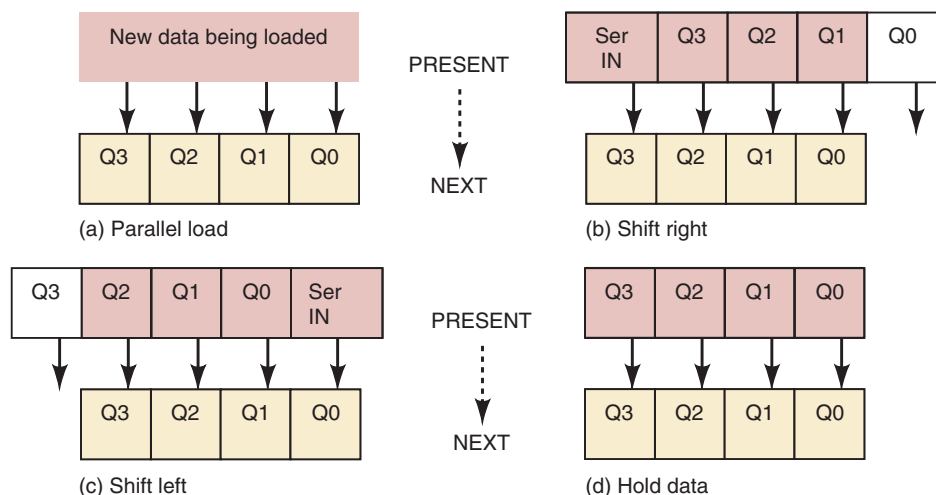
Upon completion of this section, you will be able to:

- Describe common shift-register operations and input/output transfer modes using HDL.

The various options of serial and parallel data transfer within registers were described in Section 7-15 and some example IC chips that perform these operations were examined in Section 7-16. The beauty of using HDL to describe a register is in the fact that a circuit can be given any of these options and as many bits as are needed by simply changing a few words.

HDL techniques use bit arrays to describe a register's data and to transfer that data in a parallel or serial format. To understand how data are shifted in HDL, consider the diagrams in Figure 7-83, which shows four flip-flops performing transfer operations of parallel load, shift right, shift left, and hold data. For all of these diagrams, the bits are transferred synchronously, which means that they all move simultaneously on a single clock edge. In Figure 7-83(a), the data that is to be parallel loaded into the register is presented to the  $D$  inputs, and on the next clock pulse, it will be transferred to the  $q$  outputs. Shifting data right means that each bit is transferred to the bit location to its immediate right, while a new bit is transferred in on the left end and the last bit on the right end is lost. This situation is depicted in Figure 7-83(b). Notice that the data set that we want in the NEXT state is made up of the new serial input and three of the four bits in the PRESENT state array. This data simply needs to move over and overwrite the four data bits of the register. The same operation occurs in Figure 7-83(c), but it is moving data to the left. The key to shifting the contents of the register to the right or left is to group the appropriate three PRESENT state data bits in correct order with the serial input bit so that these four bits can be loaded in parallel into the register. **Concatenation** (grouping together in a specific sequence) of the desired set of data bits can be used to describe the necessary data movement for serial shifting in either direction. The last possibility is called the hold data mode and is shown in Figure 7-83(d). It may seem unnecessary because registers (flip-flops) hold data by their very nature. We must consider, however, what must be done to a register in order to hold its value as it is clocked. The  $Q$  outputs must be tied back to the  $D$  inputs for each flip-flop so that the old data is reloaded on each clock. Let's look at some example HDL shift-register circuits.

**FIGURE 7-83** Data transfers made in shift registers: (a) parallel load; (b) shift right; (c) shift left; (d) hold data.



## AHDL SISO REGISTER

A four-bit serial in/serial out (SISO) register in AHDL is listed in Figure 7-84. An array of four D flip-flops is instantiated in line 7 and the serial output is obtained from the last FF  $q0$  (line 10). If the *shift* control is HIGH, *serial\_in* will be shifted into the register and the other bits will move to the right (lines 11–15). Concatenating *serial\_in* and FF output bits  $q3$ ,  $q2$ , and  $q1$  together in that order creates the proper shift-right data input bit pattern (line 12). If the *shift* control is LOW, the register will hold the current data (line 14). The simulation results are shown in Figure 7-85.

```

1  SUBDESIGN  fig7_84
2  (
3      clk, shift, serial_in      :INPUT;
4      serial_out                 :OUTPUT;
5  )
6  VARIABLE
7      q[3..0]                   :DFF;
8  BEGIN
9      q[].clk = clk;
10     serial_out = q0.q;          -- output last register bit
11     IF (shift == VCC) THEN
12         q[3..0].d = (serial_in, q[3..1].q); -- concatenates for shift
13     ELSE
14         q[3..0].d = (q[3..0].q);      -- hold data
15     END IF;
16 END;
```

FIGURE 7-84 Serial in/serial out register using AHDL.

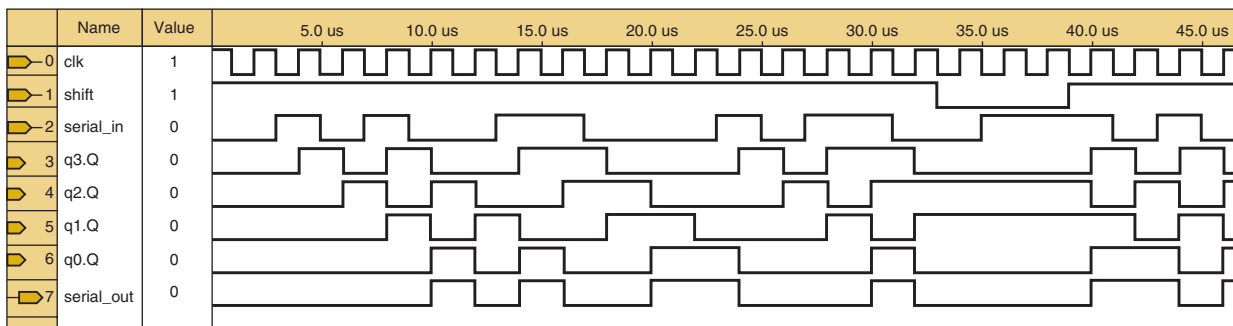


FIGURE 7-85 SISO register simulation.

## VHDL SISO REGISTER

A four-bit serial in/serial out (SISO) register in VHDL is listed in Figure 7-86. A register is created with the declaration of the variable  $q$  on line 8 and the serial output is obtained from the register's last bit or  $q(0)$  (line 10). If the *shift* control is HIGH, *serial\_in* will be shifted into the register and the other bits will move to the right (lines 12–14). Concatenating *serial\_in* and register bits  $q(3)$ ,  $q(2)$ , and  $q(1)$  together in that order creates the proper shift-right data input bit pattern (line 13). If the shift control is LOW, VHDL will assume that the variable stays the same and will therefore hold the current data. Simulation results are shown in Figure 7-85.

```

1  ENTITY  fig7_86  IS
2  PORT (      clk, shift, serial_in      :IN BIT;
3            serial_out                    :OUT BIT    );
4  END  fig7-86;
5  ARCHITECTURE  vhdl  OF  fig7-86  IS
6  BEGIN
7  PROCESS (clk)
8      VARIABLE  q      :BIT_VECTOR (3 DOWNT0 0);
9  BEGIN
10     serial_out <= q(0);                -- output last register bit
11     IF (clk'EVENT AND clk = '1') THEN
12         IF (shift = '1') THEN
13             q := (serial_in & q(3 DOWNT0 1)); -- concatenate for shift
14         END IF;                        -- otherwise, hold data
15     END IF;
16 END PROCESS;
17 END  vhdl;

```

FIGURE 7-86 Serial in/serial out register using VHDL.

### AHDL PISO REGISTER

A four-bit parallel in/serial out (PISO) register in AHDL is listed in Figure 7-87. The register named *q* is created on line 8 using four D FFs, and the serial output from *q0* is described on line 11. The register has separate parallel *load* and serial *shift* controls. The register's functions are defined in lines 12–15.

```

1  SUBDESIGN  fig7_87
2  (
3      clk, shift, load      :INPUT;
4      data[3..0]           :INPUT;
5      serial_out            :OUTPUT;
6  )
7  VARIABLE
8      q[3..0]              :DFF;
9  BEGIN
10     q[].clk = clk;
11     serial_out = q0.q;    -- output last register bit
12     IF (load == VCC) THEN q[3..0].d = data[3..0]; -- parallel load
13     ELSIF (shift == VCC) THEN q[3..0].d = (GND, q[3..1].q); -- shift
14     ELSE q[3..0].d = q[3..0].q; -- hold
15     END IF;
16 END;

```

FIGURE 7-87 Parallel in/serial out register using AHDL.

If *load* is HIGH, the external input *data[3..0]* will be synchronously loaded. *Load* has priority and must be LOW to serial-shift the register's contents on each PGT of *clk* when *shift* is HIGH. The pattern for shifting data right is created by concatenation on line 13. Note that a constant LOW will be the serial data input for a shift operation. If neither *load* nor *shift* is HIGH, the register will hold the current data value (line 14). Simulation results are shown in Figure 7-88.



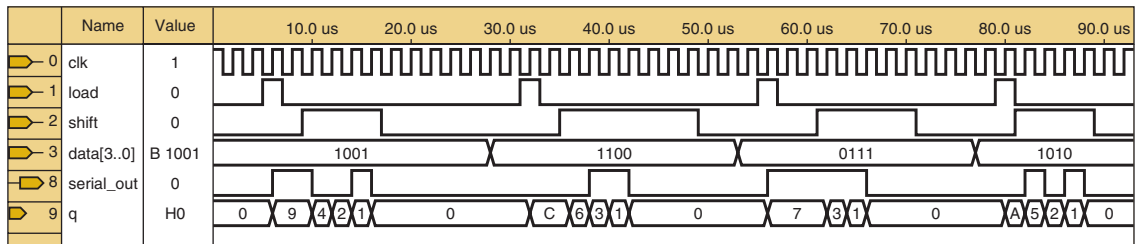


FIGURE 7-88 PISO register simulation.

## VHDL PISO REGISTER

A four-bit parallel in/serial out (PISO) register in VHDL is listed in Figure 7-89. The register is created with the variable declaration for  $q$  on line 11, and the serial output from  $q(0)$  is described on line 13. The register has separate parallel *load* and serial *shift* controls. The register's functions are defined in lines 14–18. If *load* is HIGH, the external input *data* will be synchronously loaded. *Load* has priority and must be LOW to serial-shift the register's contents on each PGT of *clk* when *shift* is HIGH. The pattern for shifting data right is created by concatenation on line 16. Note that a constant LOW will be the serial data input for a shift operation. If neither *load* nor *shift* is HIGH, the register will hold the current data value by VHDL's implied operation. Simulation results are shown in Figure 7-88.

```

1  ENTITY fig7_89 IS
2  PORT (
3      clk, shift, load      :IN BIT;
4      data                 :IN BIT_VECTOR (3 DOWNT0 0);
5      serial_out           :OUT BIT
6  );
7  END fig7_89;
8  ARCHITECTURE vhdl OF fig7_89 IS
9  BEGIN
10     PROCESS (clk)
11         VARIABLE q      :BIT_VECTOR (3 DOWNT0 0);
12         BEGIN
13             serial_out <= q(0);          -- output last register bit
14             IF (clk'EVENT AND clk = '1') THEN
15                 IF (load = '1') THEN q := data; -- parallel load
16                 ELSIF (shift = '1') THEN q := ('0' & q(3 DOWNT0 1)); -- shift
17                 END IF;                -- otherwise, hold
18             END IF;
19         END PROCESS;
20     END vhdl;

```

FIGURE 7-89 Parallel in/serial out register using VHDL.

### EXAMPLE 7-27

Suppose we want to design a universal four-bit shift register, using HDL, that has four synchronous modes of operation: Hold Data, Shift Left, Shift Right, and Parallel Load. Two input bits will select the operation that is to

be performed on each rising edge of the clock. To implement a shift register, we can use structural code to describe a string of flip-flops. Making the shift register versatile by allowing it to shift right or left or to parallel load would make this file quite long and thus hard to read and understand using structural methods. A much better approach is to use the more abstract and intuitive methods available in HDL to describe the circuit concisely. To do this, we must develop a strategy that will create the shifting action. The concept is very similar to the one presented in Example 7-18, where a D flip-flop register chip (74174) was wired to form a shift register. Rather than thinking of the shift register as a serial string of flip-flops, we consider it as a parallel register whose contents are being transferred in parallel to a set of bits that is offset by one bit position. Figure 7-83 demonstrates the concept of each transfer needed in this design.

### Solution

A very reasonable first step is to define a two-bit input named *mode* with which we can specify mode 0, 1, 2, or 3. The next challenge is deciding how to choose among the four operations using HDL. Several methods can work here. The CASE structure was chosen because it allows us to choose a different set of HDL statements for each and every possible mode value. There is no priority associated with checking for the existing mode settings or overlapping ranges of mode numbers, so we do not need the advantages of the IF/ELSE construct. The HDL solutions are given in Figures 7-90 and 7-91. The same inputs and outputs are defined in each approach: a clock, four bits of parallel load data, a single bit for the serial input to the register, two bits for the mode selection, and four output bits.

```

1  SUBDESIGN fig7_90
2  (
3      clock      :INPUT;
4      din[3..0]  :INPUT;  -- parallel data in
5      ser_in     :INPUT;  -- serial data in from Left or Right
6      mode[1..0] :INPUT;  -- MODE Select: 0=hold, 1=right, 2=left, 3=load
7      q[3..0]   :OUTPUT;
8  )
9  VARIABLE
10     ff[3..0]:DFF;      -- define register set
11  BEGIN
12     ff[].clk = clock;  -- synchronous clock
13     CASE mode[] IS
14         WHEN 0 => ff[].d      = ff[].q;      -- hold shift
15         WHEN 1 => ff[2..0].d  = ff[3..1].q;  -- shift right
16                 ff[3].d      = ser_in;     -- new data from left
17         WHEN 2 => ff[3..1].d  = ff[2..0].q;  -- shift left
18                 ff[0].d       = ser_in;     -- new data bit from right
19         WHEN 3 => ff[].d      = din[];      -- parallel load
20     END CASE;
21     q[] = ff[].q;      -- update outputs
22  END;
```

**FIGURE 7-90** AHDL universal shift register.

```

1  ENTITY fig7_91 IS
2  PORT (
3      clock      :IN BIT;
4      din        :IN BIT_VECTOR (3 DOWNTO 0);  -- parallel data in
5      ser_in     :IN BIT;                      -- serial data in L or R
6      mode       :IN INTEGER RANGE 0 TO 3;    -- 0=hold 1=rt 2=lt 3=load
7      q          :OUT BIT_VECTOR (3 DOWNTO 0));
8  END fig7_91;
9  ARCHITECTURE a OF fig7_91 IS
10 BEGIN
11     PROCESS (clock)                          -- respond to clock
12     VARIABLE ff :BIT_VECTOR (3 DOWNTO 0);
13     BEGIN
14         IF (clock'EVENT AND clock = '1') THEN
15             CASE mode IS
16                 WHEN 0 => ff := ff;           -- hold data
17                 WHEN 1 => ff(2 DOWNTO 0) := ff(3 DOWNTO 1); -- shift right
18                             ff(3) := ser_in;
19                 WHEN 2 => ff(3 DOWNTO 1) := ff(2 DOWNTO 0); -- shift left
20                             ff(0) := ser_in;
21                 WHEN 3 => ff := din;         -- parallel load
22             END CASE;
23         END IF;
24         q <= ff;                               -- update outputs
25     END PROCESS;
26 END a;

```

**FIGURE 7-91** VHDL universal shift register.

### AHDL SOLUTION

The AHDL solution of Figure 7-90 uses a register of D flip-flops declared by the name *ff* on line 10, representing the current state of the register. Because the flip-flops all need to be clocked at the same time (synchronously), all the clock inputs are assigned to *clock* on line 12. The CASE construct selects a different transfer configuration for each value of the *mode* inputs. Mode 0 (hold data) uses a direct parallel transfer from the current state to the same bit positions on the *D* inputs to produce the identical NEXT state. Mode 1 (shift right), which is described on lines 15 and 16, transfers bits 3, 2, and 1 to bit positions 2, 1, and 0, respectively, and loads bit 3 from the serial input. Mode 2 (shift left) performs a similar operation in the opposite direction (see lines 17 and 18). Mode 3 (parallel load) transfers the value on the parallel data inputs to become the NEXT state of the register. The code creates the circuitry that chooses one of these logical operations on the actual register, and the proper data is transferred to the output pins on the next clock. This code can be shortened by combining lines 15 and 16 into a single statement that concatenates the *ser\_in* with the three data bits and groups them as a set of four bits. The statement that can replace lines 15 and 16 is:

```
WHEN 1 => ff[ ].d = (ser_in, ff[3..1].q);
```

Lines 17 and 18 can also be replaced by:

```
WHEN 2 => ff[ ].d = (ff[2..0].q, ser_in);
```

## VHDL SOLUTION

The VHDL solution of Figure 7-91 defines an internal variable by the name *ff* on line 12, representing the current state of the register. Because all the transfer operations need to take place in response to a rising clock edge, a PROCESS is used, with *clock* specified in the sensitivity list. The CASE construct selects a different transfer configuration for each value of the *mode* inputs. Mode 0 (hold data) uses a direct parallel transfer from the current state to the same bit positions to produce the identical NEXT state. Mode 1 (shift right) transfers bits 3, 2, and 1 to bit positions 2, 1, and 0, respectively (line 17), and loads bit 3 from the serial input (line 18). Mode 2 (shift left) performs a similar operation in the opposite direction. Mode 3 (parallel load) transfers the value on the parallel data inputs to the NEXT state of the register. After choosing one of these operations on the actual register, the data is transferred to the output pins on line 24. This code can be shortened by combining lines 17 and 18 into a single statement that concatenates the *ser\_in* with the three data bits and groups them as a set of four bits. The statement that can replace lines 17 and 18 is:

```
WHEN 1 => ff := ser_in & ff(3 DOWNTO 1);
```

Lines 19 and 20 can also be replaced by:

```
WHEN 2 => ff := ff(2 DOWNTO 0) & ser_in;
```

### OUTCOME ASSESSMENT QUESTIONS

1. Write a HDL expression that can implement a shift left of an eight-bit array *reg[7..0]* with serial input *dat*.
2. Why is it necessary to reload the current data during the hold data mode on a shift register?

## 7-21 HDL RING COUNTERS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe the operation of ring counters using HDL.
- Assure that a ring counter will self-start using HDL.

In Section 7-17 we used a shift register to make a counter that circulates a single active logic level through all of its flip-flops. This was referred to as a ring counter. One characteristic of ring counters is that the modulus is equal to the number of flip-flops in the register and thus there are always many unused and invalid states. We have already discussed ways of describing counters using the CASE construct to specify PRESENT state and NEXT state transitions. In those examples, we took care of invalid states by including them under “others.” This method also works for ring counters. In this section, however, we look at a more intuitive way to describe shift counters.

These methods use the same techniques described in Section 7-20 in order to make the register shift one position on each clock. The main feature of this circuit design is the method of completing the “ring” by driving

the *ser\_in* line of the shift register. With a little planning, we should also be able to ensure that the counter eventually reaches the desired sequence, no matter what state it is in initially. For this example, we re-create the operation of the ring counter whose state diagram is shown in Figure 7-75(d). In order to make this counter self-start without using asynchronous inputs, we control the *ser\_in* line of the shift register using an IF/ELSE construct. Any time we detect that the upper three bits are all LOW, we assume the lowest order bit is HIGH, and on the next clock, we want to shift in a HIGH to *ser\_in*. For all other states (valid and invalid), we shift in a LOW. Regardless of the state to which the counter is initialized, it eventually fills with zeros; at which time, our logic shifts in a HIGH to start the ring sequence.

### AHDL RING COUNTER

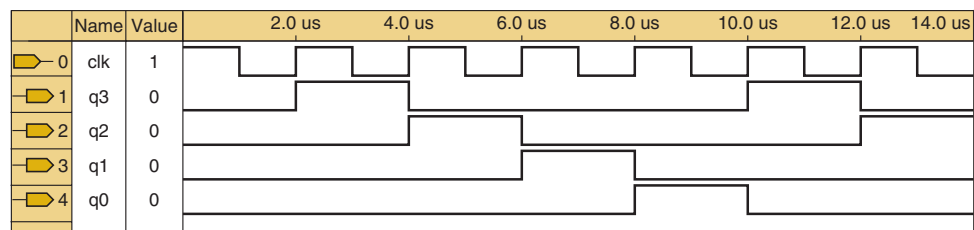
The AHDL code shown in Figure 7-92 should look familiar by now. Lines 11 and 12 control the serial input using the strategy we just described. Notice the use of the double equals (==) operator on line 11. This operator evaluates whether the expressions on each side are equal or not. Remember, the single equals (=) operator assigns (i.e., connects) one object to another. Line 14 implements the shift right action that we described previously. Simulation results are shown in Figure 7-93.

**FIGURE 7-92** AHDL four-bit ring counter.

```

1  SUBDESIGN fig7_92
2  (
3      clk          :INPUT;
4      q[3..0]     :OUTPUT;
5  )
6  VARIABLE
7      ff[3..0]    :DFF;
8      ser_in      :NODE;
9  BEGIN
10     ff[].clk = clk;
11     IF ff[3..1].q == B"000" THEN ser_in = VCC; -- self start
12     ELSE ser_in = GND;
13     END IF;
14     ff[3..0].d = (ser_in, ff[3..1].q);      -- shift right
15     q[] = ff[].q;
16 END;
```

**FIGURE 7-93** Simulation of HDL ring counter.



## VHDL RING COUNTER

The VHDL code shown in Figure 7-94 should look familiar by now. Lines 12 and 13 control the serial input using the strategy we just described. Line 16 implements the shift right action that we described previously. Simulation results are shown in Figure 7-93.

```

1  ENTITY fig7_94 IS
2  PORT (      clk      :IN BIT;
3           q          :OUT BIT_VECTOR (3 DOWNTO 0));
4  END fig7_94;
5
6  ARCHITECTURE vhdl OF fig7_94 IS
7  SIGNAL ser_in      :BIT;
8  BEGIN
9  PROCESS (clk)
10     VARIABLE ff      :BIT_VECTOR (3 DOWNTO 0);
11     BEGIN
12         IF (ff(3 DOWNTO 1) = "000") THEN ser_in <= '1';      -- self-start
13         ELSE ser_in <= '0';
14         END IF;
15         IF (clk'EVENT AND clk = '1') THEN
16             ff(3 DOWNTO 0) := (ser_in & ff(3 DOWNTO 1));    -- shift right
17         END IF;
18         q <= ff;
19     END PROCESS;
20 END vhdl;

```

FIGURE 7-94 VHDL four-bit ring counter.

### OUTCOME ASSESSMENT QUESTIONS

1. What does it mean for a ring counter to self-start?
2. Which lines of Figure 7-92 ensure that the ring counter self-starts?
3. Which lines of Figure 7-94 ensure that the ring counter self-starts?

## 7-22 HDL ONE-SHOTS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe the operation of monostable multivibrators (one-shots) using HDL.
- Differentiate operating characteristics for one-shots.
- Predict the output of any one-shot given the inputs.

Another important circuit that we have studied is the one-shot. We can apply the concept of a counter to implement a **digital one-shot** using HDL. Recall from Chapter 5 that one-shots are devices that produce a pulse of a pre-defined width every time the trigger input is activated. A *nonretriggerable* one-shot ignores the trigger input as long as the pulse output is still active.

A *retriggerable* one-shot starts a pulse in response to a trigger and restarts the internal pulse timer every time a subsequent trigger edge occurs before the pulse is complete. The first example we investigate is a nonretriggerable, HIGH-level-triggered digital one-shot. The one-shots that we studied in Chapter 5 used a resistor and capacitor as the internal pulse timing mechanism. In order to create a one-shot using HDL techniques, we use a four-bit counter to determine the width of the pulse. The inputs are a clock signal, trigger, clear, and pulse width value. The only output is the pulse out,  $Q$ . The idea is quite simple. Whenever a trigger is detected, make the pulse go HIGH and load a down-counter with a number from the pulse width inputs. The larger this number, the longer it will take to count down to zero. The advantage of this one-shot is that the pulse width can be adjusted easily by changing the value loaded into the counter. As you read the sections below, consider the following question: “What makes this circuit nonretriggerable and what makes it level-triggered?”

### SIMPLE AHDL ONE-SHOTS

A nonretriggerable, level-sensitive, one-shot description in AHDL is shown in Figure 7-95. A register of four flip-flops is created on line 8, and it serves as the counter that counts down during the pulse. The *clock* is connected in parallel to all the flip-flops on line 10. The reset function is implemented by connecting the *reset* control line directly to the asynchronous clear input of each flip-flop on line 11. After these assignments, the first condition that is tested is the trigger. If it is activated (HIGH) at any time while the count value is 0 (i.e., the previous pulse is done), then the delay value is loaded into the counter. On line 14, it tests to see if the pulse is done by checking to see if the counter is down to zero. If it is, then the counter should not roll over but rather stay at zero. If the count is not at zero, then it must be counting, so line 15 sets up the flip-flops to decrement on the next clock. Finally, line 17 generates the output pulse. This Boolean expression can be thought of as follows: “Make the pulse ( $Q$ ) HIGH when the *count* is anything other than zero.”

**FIGURE 7-95** AHDL nonretriggerable one-shot.

```

1  SUBDESIGN fig7_95
2  (
3      clock, trigger, reset      : INPUT;
4      delay[3..0]              : INPUT;
5      q                        : OUTPUT;
6  )
7  VARIABLE
8      count[3..0]              : DFF;
9  BEGIN
10     count[].clk = clock;
11     count[].clrn = reset;
12     IF trigger & count[].q == b"0000" THEN
13         count[].d = delay[];
14     ELSIF count[].q == B"0000" THEN count[].d = B"0000";
15     ELSE count[].d = count[].q - 1;
16     END IF;
17     q = count[].q != B"0000"; -- make output pulse
18 END;
```

## SIMPLE VHDL ONE-SHOTS

A nonretriggerable, level-sensitive, one-shot description in VHDL is shown in Figure 7-96. The inputs and outputs are shown on lines 3–5, as previously described. In the architecture description, a PROCESS is used (line 11) to respond to either of two inputs: the clock, or the reset. Within this PROCESS, a variable is used to represent the value on the counter. The input that should have overriding precedence is the *reset* signal. This is tested first (line 14) and if it is active, the *count* is cleared immediately. If the *reset* is not active, line 15 is evaluated and looks for a rising edge on the *clock*. Line 16 checks for the trigger. If it is activated at any time while the count value is 0 (i.e., the previous pulse is done), then the width value is loaded into the counter. On line 18, it tests to see if the pulse is done by checking to see if the counter is down to zero. If it is, then the counter should not roll over but rather stay at zero. If the *count* is not at zero, then it must be counting, so line 19 sets up the flip-flops to decrement on the next clock. Finally, lines 22 and 23 generate the output pulse. This Boolean expression can be thought of as follows: “Make the pulse (*q*) HIGH when the count is anything other than zero.”

```

1  ENTITY fig7_96 IS
2  PORT (
3      clock, trigger, reset    :IN BIT;
4      delay                   :IN INTEGER RANGE 0 TO 15;
5      q                       :OUT BIT
6  );
7  END fig 7_96;
8
9  ARCHITECTURE vhdl OF fig7_96 IS
10 BEGIN
11     PROCESS (clock, reset)
12     VARIABLE count           : INTEGER RANGE 0 TO 15;
13     BEGIN
14         IF reset = '0' THEN count := 0;
15         ELSIF (clock'EVENT AND clock = '1' ) THEN
16             IF trigger = '1' AND count = 0 THEN
17                 count := delay;                               -- load counter
18             ELSIF count = 0 THEN count := 0;
19             ELSE count := count - 1;
20             END IF;
21         END IF;
22         IF count /= 0 THEN q <= '1';
23         ELSE q <= '0';
24         END IF;
25     END PROCESS;
26 END vhdl;

```

**FIGURE 7-96** VHDL nonretriggerable one-shot.

### Nonretriggerable One-Shot Simulation

Now that we have reviewed the code that describes this one-shot, let's evaluate its performance. Converting a traditionally analog circuit to digital usually offers some advantages and some disadvantages. On a standard



one-shot chip, the output pulse starts immediately after the trigger. For the digital one-shot described here, the output pulse starts on the next clock edge and lasts as long as the counter is greater than zero. This situation is shown in Figure 7-97 within the first ms of the simulation. Notice that the trigger goes high almost 0.5 ms before the  $q$  out responds. If another trigger event occurs while it is counting down (like the one just before 3 ms), it is ignored. This is the nonretriggerable characteristic.

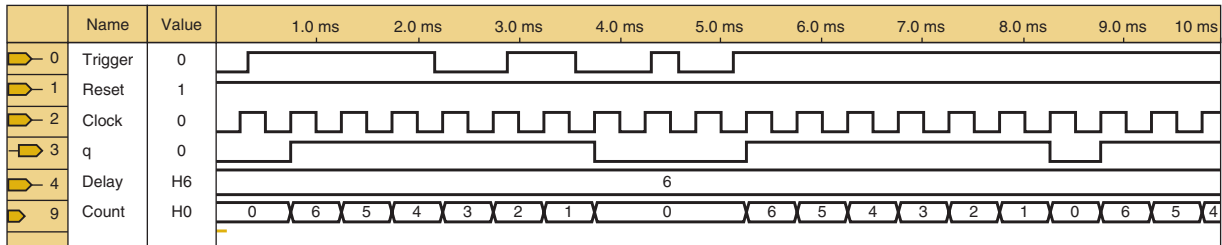


FIGURE 7-97 Simulation of the nonretriggerable one-shots.

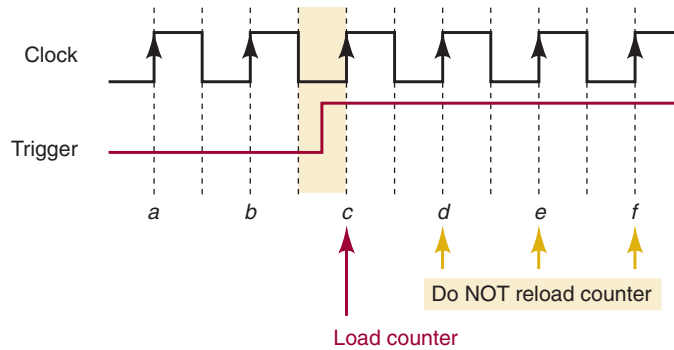
Another point to make for this digital one-shot is that the trigger pulse must be long enough to be seen as a HIGH on the rising clock edge. At about the 4.5-ms mark, a pulse occurs on the trigger input but goes LOW before the rising edge of the clock. This circuit does *not* respond to this input event. At just past 5 ms, the trigger goes HIGH and stays there. The pulse lasts exactly 6 ms, but because the trigger input remains HIGH, it responds with another output pulse one clock later. The reason for this situation is that this circuit is level-triggered rather than edge-triggered, like most of the conventional one-shot ICs.

### Retriggerable, Edge-Triggered One-Shots in HDL

Many applications of one-shots require the circuit to respond to an edge rather than a level. How can HDL code be used to make the circuit respond once to each positive transition on its trigger input? The technique described here is called *edge-trapping* and has been a very useful tool in programming microcontrollers for years. As we will see, it is equally useful for describing edge-triggering for a digital circuit using HDL. This section illustrates an example of a retriggerable one-shot while also demonstrating edge-trapping, which can be useful in many other situations.

The general operation of this retriggerable one-shot requires that it respond to a rising edge of the trigger input. As soon as the edge is detected, it should start timing the pulse. In the digital one-shot, this means that it loads the counter as soon as possible after the trigger edge and starts counting down toward zero. If another trigger event (rising edge) occurs before the pulse is terminated, the counter is immediately reloaded, and the pulse timing starts again from the beginning, thus sustaining the pulse. Activating the clear at any point should force the counter to zero and terminate the pulse. The minimum output pulse width is simply the number applied to the width input multiplied by the clock period.

The strategy behind edge-trapping for a one-shot is demonstrated in Figure 7-98. On each active clock edge are two important pieces of information that are needed. The first is the state of the *trigger* input *now* and the second is the state of the *trigger* input when the last active clock edge occurred. Start with point *a* on the diagram of Figure 7-98 and determine these two values, then move to point *b*, and so on. By completing this task, you should have concluded that, at point *c*, a unique result has been

**FIGURE 7-98** Detecting edges.

obtained. The *trigger* is HIGH now but it was LOW on the last active clock edge. This is the point where we have detected the *trigger* edge event.

In order to know what the *trigger* was on the last active clock edge, the system must remember the last value that *the trigger had at that point*. This is done by storing the value of the trigger bit in a flip-flop. Recall that we discussed a similar concept in Chapter 5 when we talked about using a flip-flop to detect a sequence. The code for a one-shot is written so that the counter is loaded only after a rising edge is detected on the *trigger* input.

## AHDL RETRIGGERABLE, EDGE-TRIGGERED ONE-SHOT

The first five lines of Figure 7-99 are identical to the previous nonretriggerable example. In AHDL, the only way to remember a value obtained in the past is to store the value on a flip-flop. This section uses a flip-flop named *trig\_was* (line 9) to store the value that was on the trigger on the last active

**FIGURE 7-99** AHDL retriggerable one-shot with edge trigger.

```

1  SUBDESIGN fig7_99
2  (
3      clock, trigger, reset  : INPUT;
4      delay[3..0]           : INPUT;
5      q                     : OUTPUT;
6  )
7  VARIABLE
8      count[3..0]          : DFF;
9      trig_was             : DFF;
10 BEGIN
11     count[].clk = clock;
12     count[].clrn = reset;
13     trig_was.clk = clock;
14     trig_was.d = trigger;
15
16     IF trigger & !trig_was.q THEN
17         count[].d = delay[];
18     ELSIF count[].q == B"0000" THEN count[].d = B"0000";
19     ELSE count[].d = count[].q - 1;
20     END IF;
21     q = count[].q != B"0000";
22 END;
```

clock edge. This flip-flop is simply connected so that the trigger is on its *D* input (line 14) and the clock is connected to its *clk* input (line 13). The *Q* output of *trig\_was* remembers the value of the *trigger* right up to the next clock edge. At this point, we use line 16 to evaluate if a triggering edge has occurred. If *trigger* is HIGH (now), but *trigger* was LOW (last clock), it is time to load the counter (line 17). Line 18 ensures that, once the *count* reaches zero, it will remain at zero until a new trigger comes along. If the decisions allow line 19 to be evaluated, it means that there is a value loaded into the counter and it is not zero, so it needs to be decremented. Finally, the output pulse is made HIGH any time a value other than 0000 is still on the counter, like we saw previously.

## VHDL RETRIGGERABLE, EDGE-TRIGGERED ONE-SHOT

The ENTITY description in Figure 7-100 is exactly like the previous non-retriggerable example. In fact, the only differences between this example and the one shown in Figure 7-96 have to do with the logic of the decision process. When we want to remember a value in VHDL, it must be stored in a VARIABLE. Recall that we can think of a PROCESS as a description of what happens each time a signal in the sensitivity list changes state. A VARIABLE retains the last value assigned to it between the times the

```

1  ENTITY fig7_100 IS
2  PORT ( clock, trigger, reset      : IN BIT;
3         delay                      : IN INTEGER RANGE 0 TO 15;
4         q                          : OUT BIT);
5  END fig7_100;
6
7  ARCHITECTURE vhdl OF fig7_100 IS
8  BEGIN
9      PROCESS (clock, reset)
10     VARIABLE count      : INTEGER RANGE 0 TO 15;
11     VARIABLE trig_was   : BIT;
12     BEGIN
13         IF reset = '0' THEN count := 0;
14         ELSIF (clock'EVENT AND clock = '1' ) THEN
15             IF trigger = '1' AND trig_was = '0' THEN
16                 count := delay;          -- load counter
17                 trig_was := '1';        -- "remember" edge detected
18             ELSIF count = 0 THEN count := 0; -- hold @ 0
19             ELSE count := count - 1;      -- decrement
20             END IF;
21             IF trigger = '0' THEN trig_was := '0';
22             END IF;
23         END IF;
24         IF count /= 0 THEN q <= '1';
25         ELSE q <= '0';
26         END IF;
27     END PROCESS;
28 END vhdl;

```

**FIGURE 7-100** VHDL retriggerable one-shot with edge trigger.

process is invoked. In this sense, it acts like a flip-flop. For the one-shot, we need to store a value that tells us what the trigger was on the last active clock edge. Line 11 declares a variable bit to serve this purpose. The first decision (line 13) is the overriding decision that checks and responds to the *reset* input. Notice that this is an asynchronous control because it is evaluated before the clock edge is detected on line 14. Line 14 determines that a rising clock edge has occurred, and then the main logic of this process is evaluated between lines 15 and 20.

When a clock edge occurs, one of three conditions exists:

1. A trigger edge has occurred and we must load the counter.
2. The counter is zero and we need to keep it at zero.
3. The counter is not zero and we need to count down by one.

Recall that it is very important to consider the order in which questions are asked and assignments are made in VHDL PROCESS statements because the *sequence* affects the operation of the circuit we are describing. The code that updates the *trig\_was* variable must occur after the evaluation of its previous condition. For this reason, the conditions necessary to detect a rising edge on *trigger* are evaluated on line 15. If an edge occurred, then the counter is loaded (line 16) and the variable is updated (line 17) to remember this for the next time. If a trigger edge has not occurred, the code either holds at zero (line 18) or counts down (line 19). Line 21 makes sure that, as soon as the trigger input goes LOW, the variable *trig\_was* remembers this by resetting. Finally, lines 24–25 are used to create the output pulse during the time the counter is not zero.

### Edge-Triggered Retriggerable One-Shot Simulation

The two improvements that were made in this one-shot over the last example are the edge-triggering and the retriggerable feature. Figure 7-101 evaluates the new performance features. Notice in the first ms of the timing diagram that a trigger edge is detected, but the response is not immediate. The output pulse goes high on the next clock edge. This is a drawback to the digital one-shot. The retriggerable feature is demonstrated at about the 2-ms mark. Notice that *trigger* goes high and on the next clock edge, the *count* starts again at 5, sustaining the output pulse. Also notice that even after the *q* output pulse is complete and the *trigger* is still HIGH, the one-shot does not fire another pulse because it is not level-triggered but rather rising edge-triggered. At the 6-ms mark, a short trigger pulse occurs but is ignored because it does not stay HIGH until the next clock. On the other hand, an even shorter trigger pulse occurring just after the 7-ms mark does fire the one-shot because it is present during the rising clock edge. The

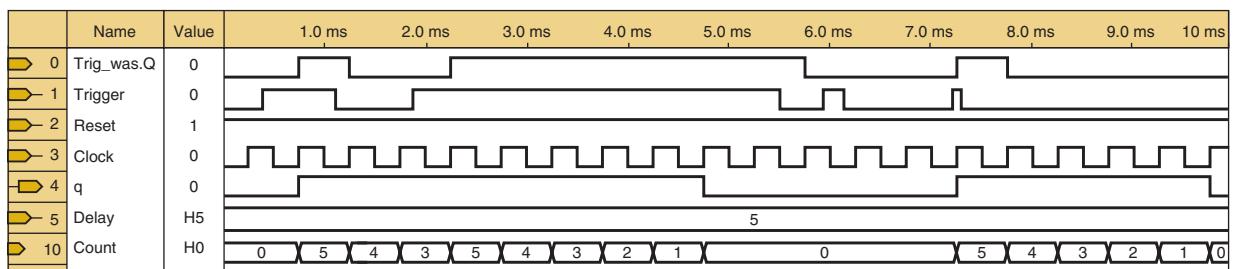


FIGURE 7-101 Simulation of the edge-triggered retriggerable one-shot.

resulting output pulse lasts exactly five clock cycles because no other triggers occur during this period.

To minimize the effects of delayed response to trigger edges and the possibility of missing trigger edges that are too short, this circuit can be improved quite simply. The clock frequency and the number of bits used to load the delay value can both be increased to provide the same range of pulse widths (with more precise control) while reducing the minimum trigger pulse width. In order to cure this problem completely, the one-shot must respond asynchronously to the trigger input. This is possible in both AHDL and VHDL, but it will always result in a pulse that fluctuates in width by up to one clock period.

### OUTCOME ASSESSMENT QUESTIONS

1. Which control input signal holds the highest priority for each of the one-shot descriptions?
2. Name two factors that determine how long a pulse from a digital one-shot will last.
3. For the one-shots shown in this section, are the counters loaded synchronously or asynchronously?
4. What is the advantage of loading a counter synchronously?
5. What is the advantage of loading the counter asynchronously?
6. What two pieces of information are necessary to detect an edge?

## PART 2 SUMMARY

1. Numerous IC registers are available and can be classified according to whether their inputs are parallel (all bits entered simultaneously), serial (one bit at a time), or both. Likewise, registers can have outputs that are parallel (all bits available simultaneously) or serial (one bit available at a time).
2. A sequential logic system uses FFs, counters, and registers, along with logic gates. Its outputs and sequence of operations depend on present and past inputs.
3. Troubleshooting a sequential logic system begins with observation of the system operation, followed by analytical reasoning to determine the possible causes of any malfunction, and finally test measurements to isolate the actual fault.
4. A ring counter is actually an  $N$ -bit shift register that recirculates a single 1 continuously, thereby acting as a MOD- $N$  counter. A Johnson counter is a modified ring counter that operates as MOD- $2N$  counter.
5. Shift registers can be implemented with HDL by writing custom descriptions of their operation.
6. The LPM\_SHIFTRREG megafunction can be used in schematics to implement shift registers for each of the data transfer options.
7. An understanding of bit arrays/bit vectors and their notation is very important in describing shift-register operations.
8. Shift-register counters such as Johnson and ring counters can be implemented easily in HDL. Decoding and self-starting features are easy to write into the description.

9. Digital one-shots are implemented with a counter loaded with a delay value when the trigger input is detected and counts down to zero. During the countdown time, the output pulse is held HIGH.
10. With strategic placement of the hardware description statements, HDL one-shots can be made edge- or level-triggered and retriggerable or nonretriggerable. They produce an output pulse that responds synchronously or asynchronously to the trigger.

## PART 2 IMPORTANT TERMS

---

parallel in/parallel out	circulating shift register	sequential logic system
serial in/serial out	ring counter	LPM_SHIFTREG
parallel in/serial out	Johnson counter (twisted	concatenation
serial in/parallel out	ring counter)	digital one-shot

## PROBLEMS

---

### PART 1

#### SECTION 7-1

- B** 7-1\* Add another flip-flop,  $E$ , to the counter of Figure 7-1. The clock signal is an 8-MHz square wave.
- (a) What will be the frequency at the  $E$  output? What will be the duty cycle of this signal?
  - (b) Repeat (a) if the clock signal has a 20% duty cycle.
  - (c) What will be the frequency at the  $C$  output?
  - (d) What is the MOD number of this counter?
- B** 7-2. Draw a binary counter that will convert a 64-kHz pulse signal into a 1-kHz square wave.
- B** 7-3\* Assume that a five-bit binary counter starts in the 00000 state. What will be the count after 144 input pulses?
- B** 7-4. A 10-bit ripple counter has a 256-kHz clock signal applied.
- (a) What is the MOD number of this counter?
  - (b) What will be the frequency at the MSB output?
  - (c) What will be the duty cycle of the MSB signal?
  - (d) Assume that the counter starts at zero. What will be the count in hexadecimal after 1000 input pulses?

#### SECTION 7-2

- 7-5\* A four-bit ripple counter is driven by a 20-MHz clock signal. Draw the waveforms at the output of each FF if each FF has  $t_{pd} = 20$  ns. Determine which counter states, if any, will not occur because of the propagation delays.
- 7-6. (a) What is the maximum clock frequency that can be used with the counter of Problem 7-5?
- (b) What would  $f_{max}$  be if the counter were expanded to six bits?

---

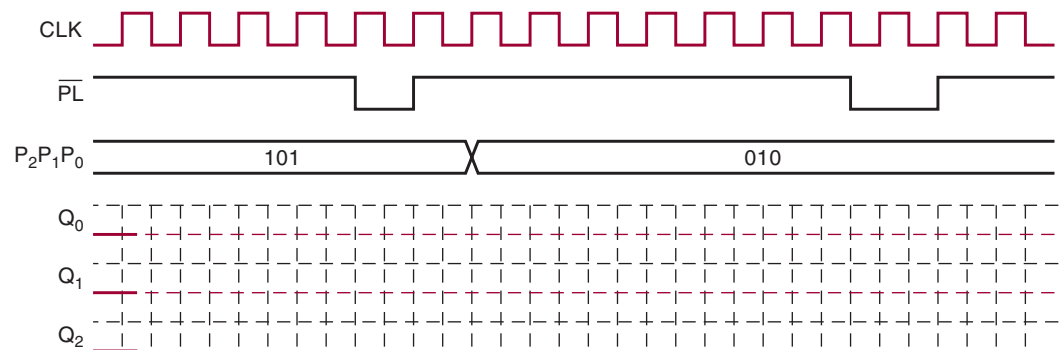
\* Answers to problems marked with an asterisk can be found in the back of the text.

## SECTIONS 7-3 AND 7-4

- B** 7-7\* (a) Draw the circuit diagram for a MOD-32 synchronous counter.  
 (b) Determine  $f_{\max}$  for this counter if each FF has  $t_{pd} = 20$  ns and each gate has  $t_{pd} = 10$  ns.
- B** 7-8. (a) Draw the circuit diagram for a MOD-64 synchronous counter.  
 (b) Determine  $f_{\max}$  for this counter if each FF has  $t_{pd} = 20$  ns and each gate has  $t_{pd} = 10$  ns.
- B, N** 7-9. The decade counter in Figure 7-8(b) has a 1-kHz clock applied.  
 (a) Draw the waveforms for each FF output, showing any glitches that may occur.  
 (b) Determine the frequency of the signal at the  $D$  output.  
 (c) If the counter is originally at state 1000, what state will the counter be at after 14 clock pulses are applied?  
 (d) If the counter is originally at state 0101, what state will the counter be at after 20 clock pulses are applied?
- B** 7-10. Repeat Problem 7-9 for the counter of Figure 7-8(a) with a 70-kHz clock.
- 7-11\* Change the inputs to the NAND gate of Figure 7-9 so that the counter divides the input frequency by 50.
- D** 7-12. Draw a synchronous counter that will output a 10-kHz signal when a 1-MHz clock is applied.

## SECTIONS 7-5 AND 7-6

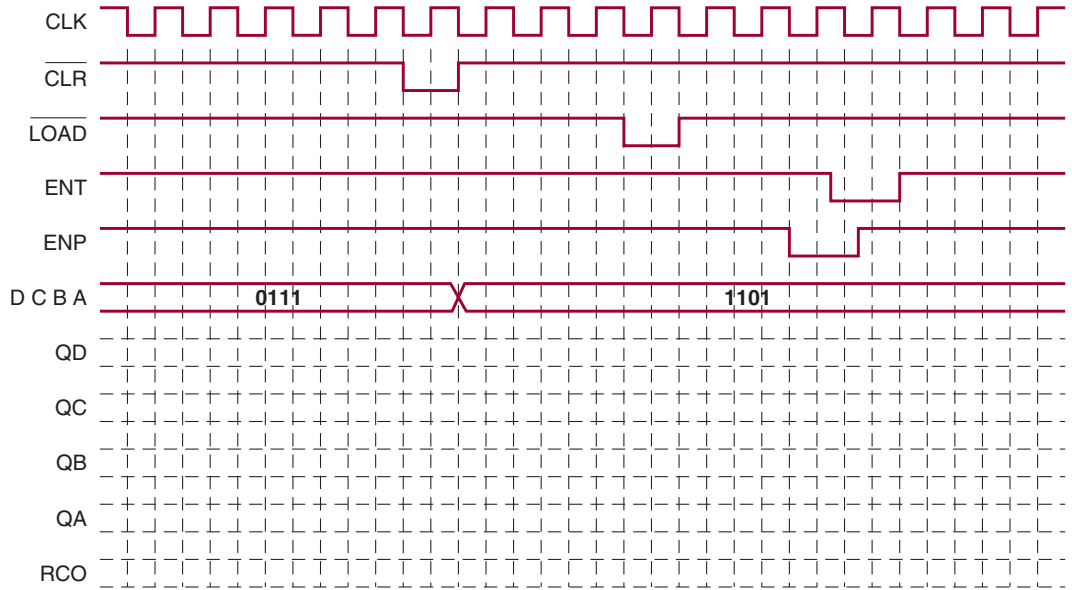
- B** 7-13\* Draw a synchronous, MOD-32, down counter.
- B** 7-14. Draw a synchronous, MOD-16, up/down counter. The count direction is controlled by  $dir$  ( $dir = 0$  to count up).
- C, T** 7-15\* Determine the count sequence of the up/down counter in Figure 7-11 if the INVERTER output were stuck HIGH. Assume the counter starts at 000.
- 7-16. Complete the timing diagram in Figure 7-102 for the presettable counter in Figure 7-12. Note that the initial condition for the counter is given in the timing diagram.



**FIGURE 7-102** Problem 7-16 timing diagram.

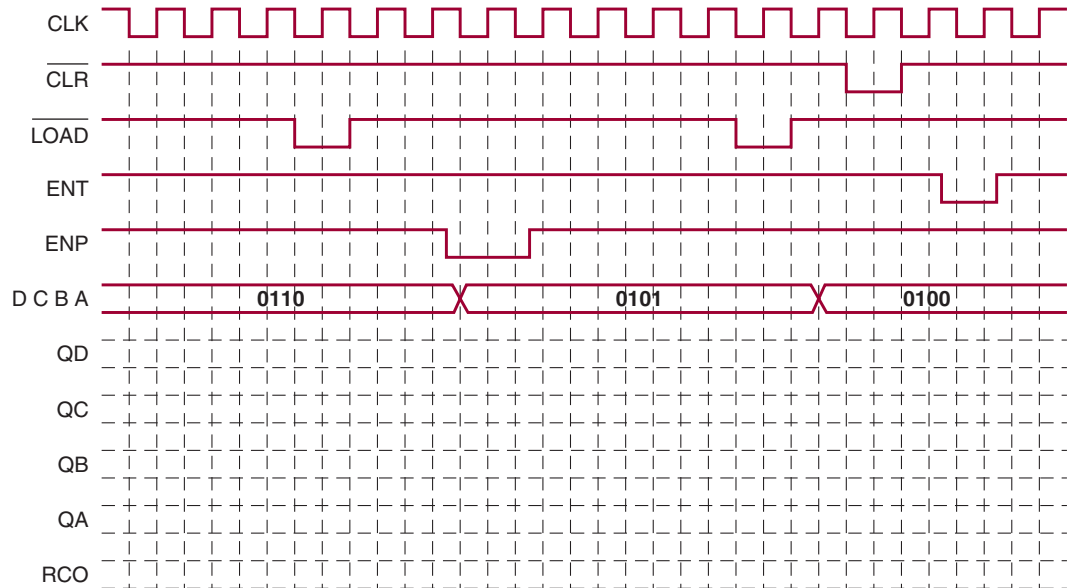
**SECTION 7-7**

7-17\* Complete the timing diagram in Figure 7-103 for a 74ALS161 with the indicated input waveforms applied. Assume the initial state is 0000.



**FIGURE 7-103** Problem 7-17 timing diagram.

7-18. Complete the timing diagram in Figure 7-104 for a 74ALS162 with the indicated input waveforms applied. Assume the initial state is 0000.



**FIGURE 7-104** Problem 7-18 timing diagram.



7-19\* Complete the timing diagram in Figure 7-105 for a 74ALS190 with the indicated input waveforms applied. The DCBA input is 0101.

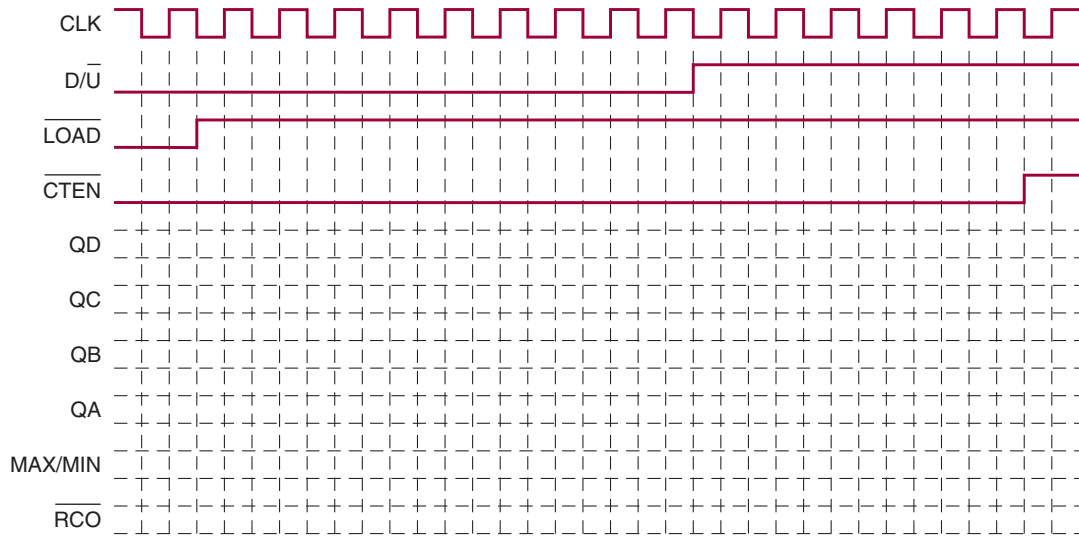


FIGURE 7-105 Problems 7-19 and 7-20 timing diagram.

7-20. Repeat Problem 7-19 for a 74ALS191 and a DCBA input of 1100.

- B** 7-21\* Refer to the IC counter circuit in Figure 7-106(a):
- Draw the state transition diagram for the counter's QD QC QB QA outputs.
  - Determine the counter's modulus.

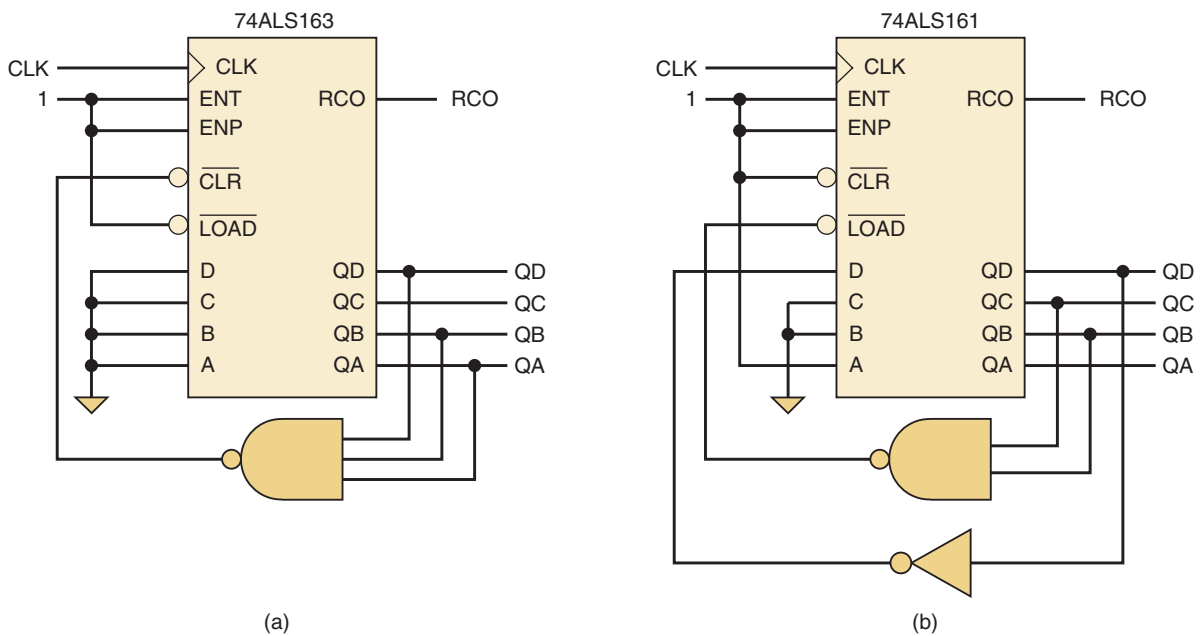
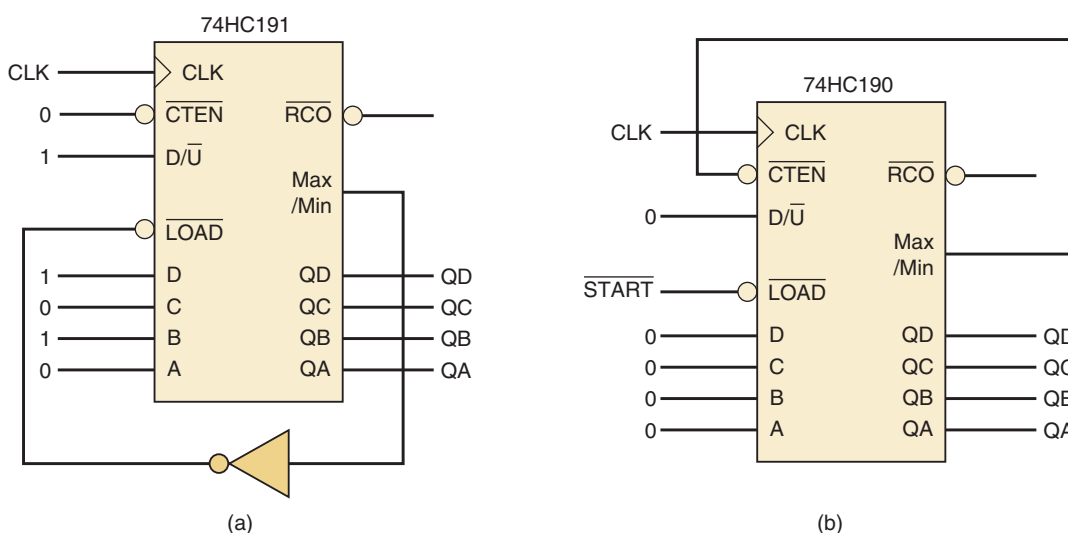


FIGURE 7-106 Problems 7-21 and 7-22

- (c) What is the relationship of the output frequency of the MSB to the input *CLK* frequency?
- (d) What is the duty cycle of the MSB output waveform?

- B** 7-22. Repeat Problem 7-21 for the IC counter circuit in Figure 7-106(b).
- B** 7-23\* Refer to the IC counter circuit in Figure 7-107(a).
  - (a) Draw the timing diagram for outputs *QD QC QB QA*.
  - (b) What is the counter's modulus?
  - (c) What is the count sequence? Does it count up or down?
  - (d) Can we produce the same modulus with a 74HC190? Can we produce the same count sequence with a 74HC190?

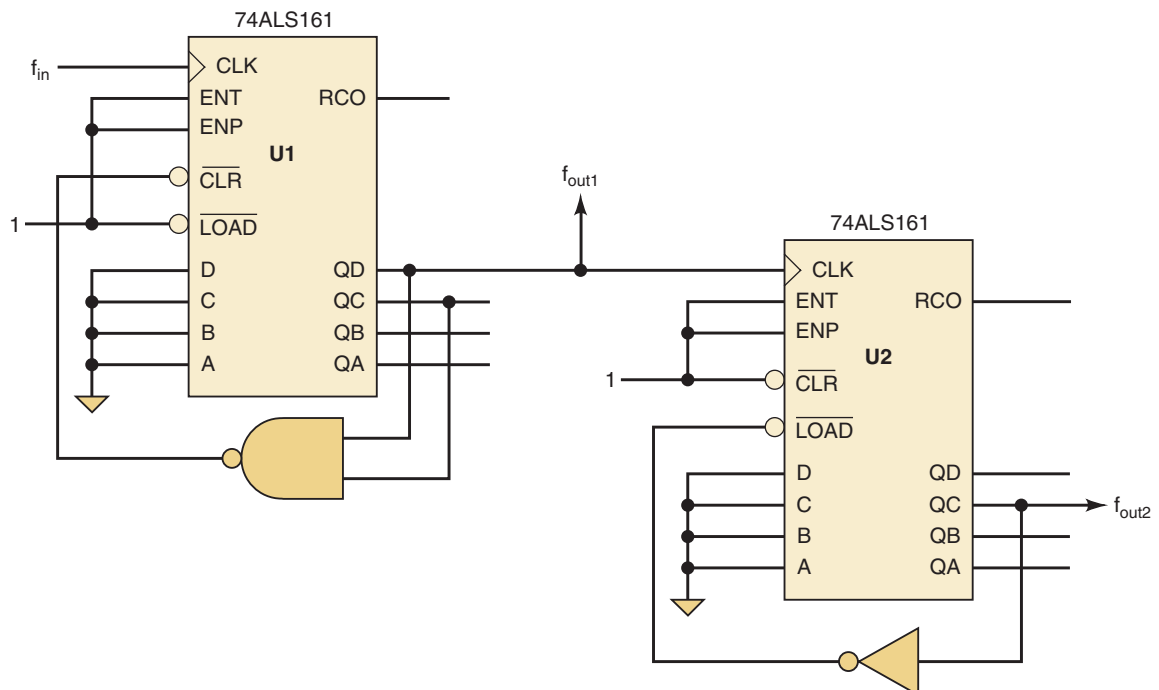


**FIGURE 7-107** Problems 7-23 and 7-24.

- 7-24. Refer to the IC counter circuit in Figure 7-107(b):
  - (a) Describe the counter's output on *QD QC QB QA* if  $\overline{START}$  is LOW.
  - (b) Describe the counter's output on *QD QC QB QA* if  $\overline{START}$  is momentarily pulsed LOW and then returns to a HIGH.
  - (c) What is the counter's modulus? Is this a recycling counter?
- D** 7-25\* Draw a schematic to create a recycling, MOD-6 counter that uses:
  - (a) the clear control on a 74ALS160
  - (b) the clear control on a 74ALS162
- D** 7-26. Draw a schematic to create a recycling, MOD-6 counter that produces the count sequence:
  - (a) 1, 2, 3, 4, 5, 6, and repeats with a 74ALS162
  - (b) 5, 4, 3, 2, 1, 0, and repeats with a 74ALS190
  - (c) 6, 5, 4, 3, 2, 1, and repeats with a 74ALS190
- D** 7-27\* Design a MOD-100, binary counter using either two 74HC161 or two 74HC163 chips and any necessary gates. The IC counter chips are to be synchronously cascaded together to produce the binary count sequence for 0 to 99. The MOD-100 is to have two control inputs,

an active-LOW count enable ( $\overline{EN}$ ) and an active-LOW, asynchronous clear ( $\overline{CLR}$ ). Label the counter outputs  $Q0$ ,  $Q1$ ,  $Q2$ , and so on, with  $Q0 = \text{LSB}$ . Which output is the MSB?

- D** 7-28. Design a MOD-100, BCD counter using either two 74HC160 or two 74HC162 chips and any necessary gates. The IC counter chips are to be synchronously cascaded together to produce the BCD count sequence for 0 to 99. The MOD-100 is to have two control inputs, an active-HIGH count enable ( $EN$ ) and an active-HIGH, synchronous load ( $LD$ ). Label the counter outputs  $Q0$ ,  $Q1$ ,  $Q2$ , and so on, with  $Q0 = \text{LSB}$ . Which set of outputs represents the 10s digit?
- B** 7-29\* With a 6-MHz clock input to a 74ALS163 that has all four control inputs HIGH, determine the output frequency and duty cycle for each of the *five* outputs (including  $RCO$ ).
- B** 7-30. With a 6-MHz clock input to a 74ALS162 that has all four control inputs HIGH, determine the output frequency and duty cycle for each of the following outputs:  $QA$ ,  $QC$ ,  $QD$ ,  $RCO$ . What is unusual about the waveform pattern that would be produced by the  $QB$  output? This pattern characteristic results in an undefined duty cycle.
- B** 7-31\* The frequency of  $f_{in}$  is 6 MHz in Figure 7-108. The two IC counter chips have been cascaded asynchronously so that the output frequency produced by counter U1 is the input frequency for counter U2. Determine the output frequency for  $f_{out1}$  and  $f_{out2}$ .



**FIGURE 7-108** Problem 7-31.

- B** 7-32. The frequency of  $f_{in}$  is 1.5 MHz in Figure 7-109. The two IC counter chips have been cascaded asynchronously so that the output frequency produced by counter U1 is the input frequency for counter U2. Determine the output frequency for  $f_{out1}$  and  $f_{out2}$ .

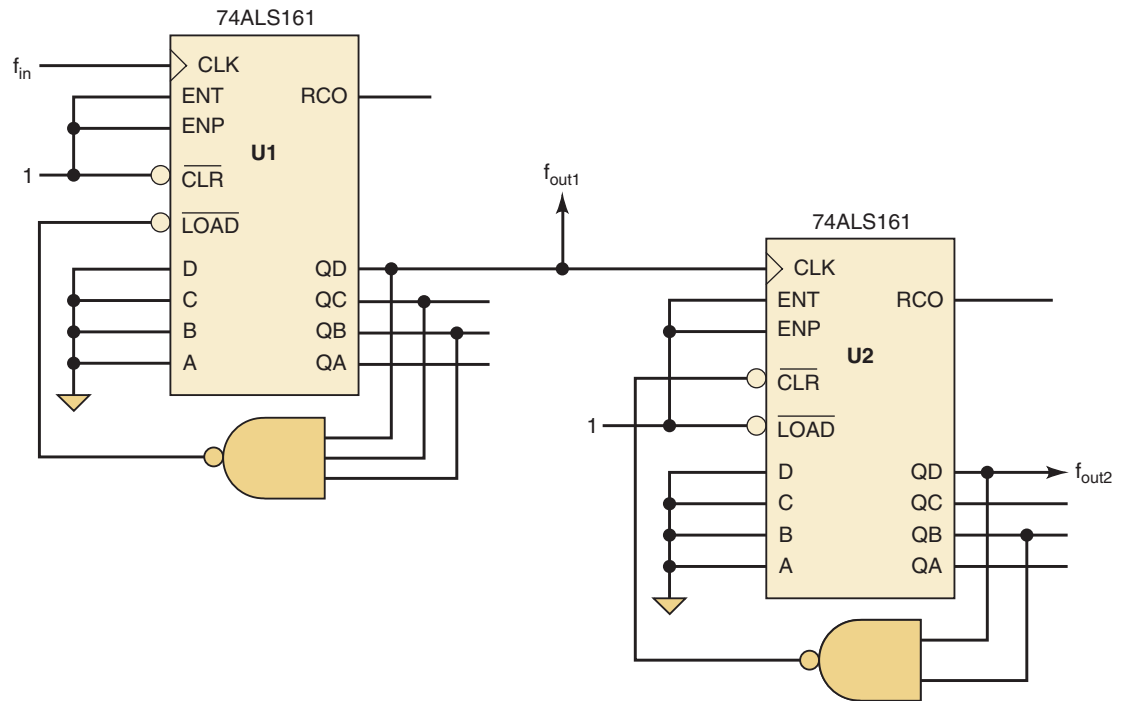


FIGURE 7-109 Problem 7-32.

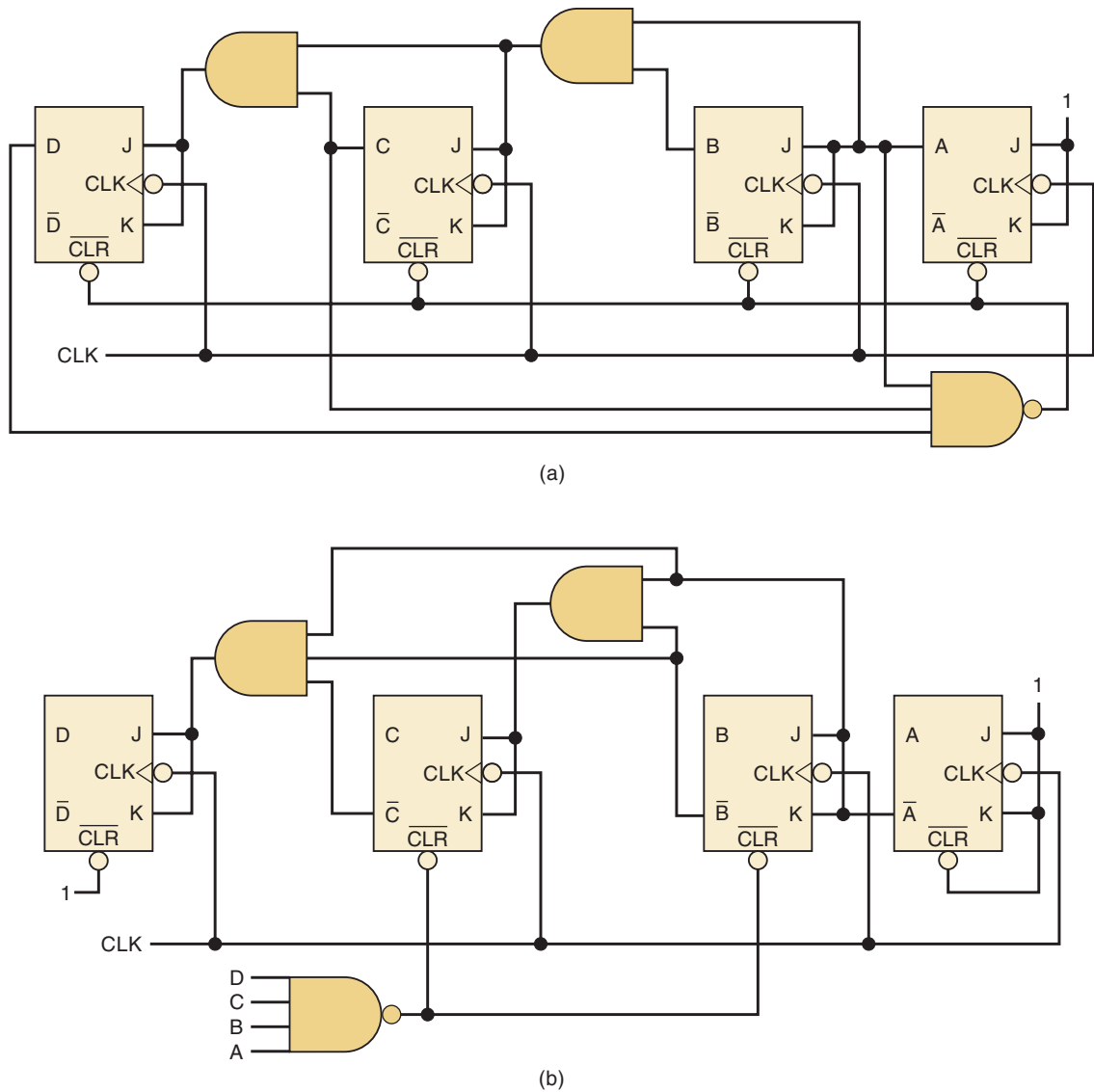
- D** 7-33\* Design a frequency divider circuit that will produce the following three output signal frequencies: 1.5 MHz, 150 kHz, and 100 kHz. Use 74HC162 and 74HC163 counter chips and any necessary gates. The input frequency is 12 MHz.
- D** 7-34. Design a frequency divider circuit that will produce the following three output signal frequencies: 1 MHz, 800 kHz, and 100 kHz. Use 74HC160 and 74HC161 counter chips and any necessary gates. The input frequency is 12 MHz.

**SECTION 7-8**

- B** 7-35\* Draw the gates necessary to decode all of the states of a MOD-16 counter using active-LOW outputs.
- B** 7-36. Draw the AND gates necessary to decode the 10 states of the BCD counter of Figure 7-8(b).

**SECTION 7-9**

- C** 7-37\* Analyze the synchronous counter in Figure 7-110(a). Draw its timing diagram and determine the counter's modulus.
- C** 7-38. Repeat Problem 7-37 for Figure 7-110(b).
- C** 7-39\* Analyze the synchronous counter in Figure 7-111(a). Draw its timing diagram and determine the counter's modulus.
- C** 7-40. Repeat Problem 7-39 for Figure 7-111(b).



**FIGURE 7-110** Problems 7-37 and 7-38.

- C** 7-41\* Analyze the synchronous counter in Figure 7-112(a). F is a control input. Draw its state transition diagram and determine the counter's modulus.
- C** 7-42. Analyze the synchronous counter in Figure 7-112(b). Draw its complete state transition diagram and determine the counter's modulus. Is the counter self-correcting?

### SECTION 7-10

- D** 7-43\*(a) Design a synchronous counter using J-K FFs that has the following sequence: 000, 010, 101, 110, and repeat. The undesired (unused) states 001, 011, 100, and 111 must always go to 000 on the next clock pulse.
- (b) Redesign the counter of part (a) without any requirement on the unused states; that is, their NEXT states can be don't cares. Compare with the design from (a).



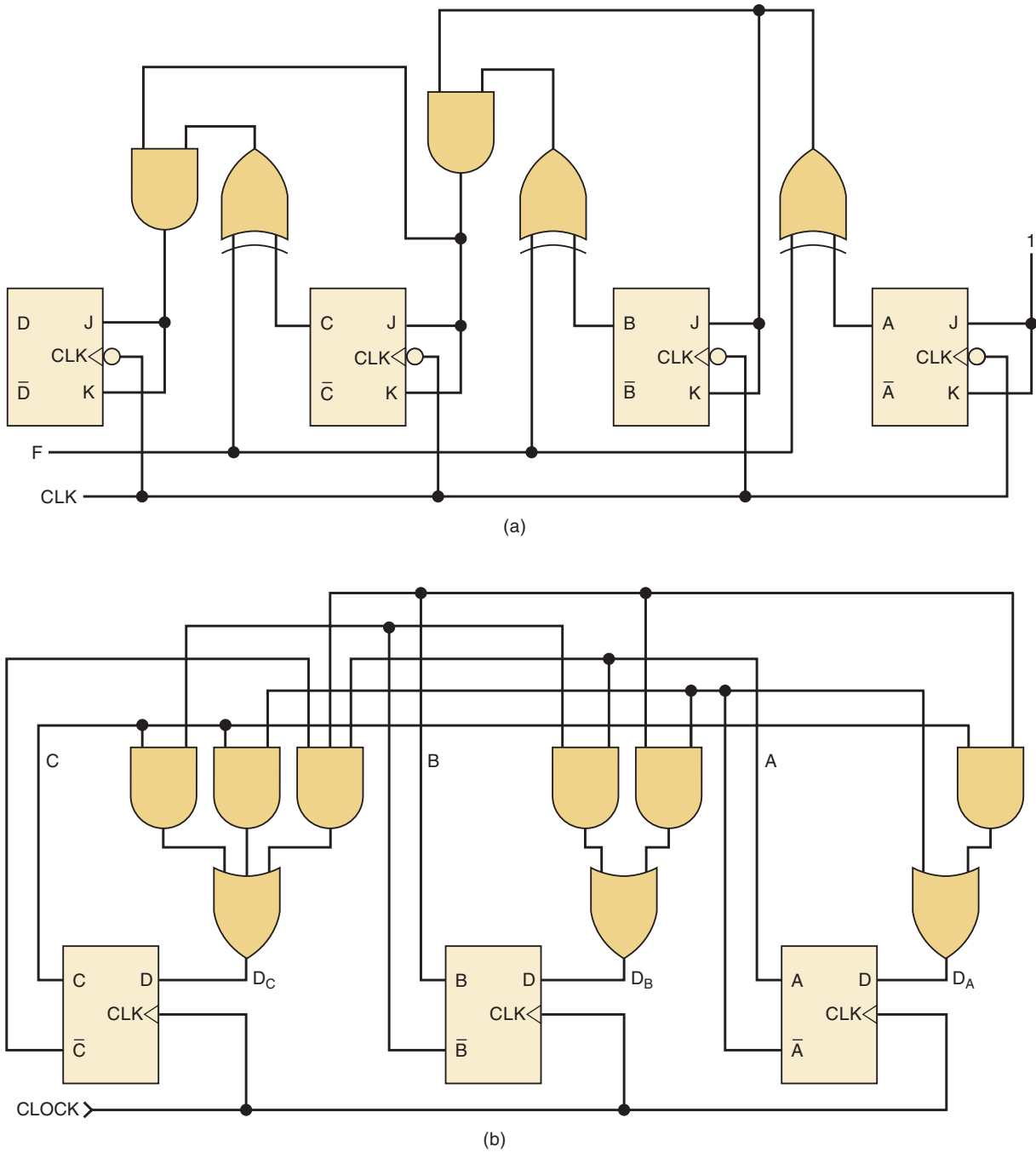


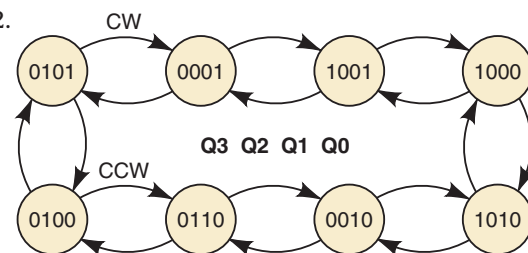
FIGURE 7-112 Problems 7-41 and 7-42.

**SECTIONS 7-11 AND 7-12**

- H, D, N** 7-49\* Design a recycling, MOD-13, up counter. The count sequence should be 0000 through 1100. Simulate (functional) the counter.
- (a) Use LPM\_COUNTER.
  - (b) Use an HDL.

- H, D, N** 7-50. Design a recycling, MOD-25, down counter. The count sequence should be 11000 through 00000. Simulate (functional) the counter.
- Use LPM\_COUNTER.
  - Use an HDL.
- H, D** 7-51\* Design a recycling, MOD-16 Gray code counter using an HDL. The counter should have an active-HIGH enable (*cnt*). Simulate the counter.
- H, D** 7-52. Design a bidirectional, half-step controller for a stepper motor using an HDL. The direction control input (*dir*) will produce a clockwise (CW) pattern when HIGH or counterclockwise when LOW. The sequence is given in Figure 7-113. Simulate the sequential circuit.

**FIGURE 7-113** Problem 7-52.



- H, D, N** 7-53\* Design a frequency divider circuit to output a 100-kHz signal. The input frequency is 5 MHz. Simulate (functional) the counter.
- Use LPM\_COUNTER.
  - Use an HDL.
- H, D, N** 7-54. Design a frequency divider circuit that will output either of two specified frequency signals. The output frequency is selected by the control input *fselect*. The divider will output a frequency of 5 kHz when *fselect* = 0 or 12 kHz when *fselect* = 1. The input frequency is 60 kHz. Simulate (functional) the counter.
- Use LPM\_COUNTER. Hints: Create a down counter that reloads the appropriate value (determined by *fselect*) after the terminal state is reached. You will also need one logic gate.
  - Use an HDL.
- H, B** 7-55\* Expand the full-featured HDL counter in Section 7-12 to a MOD-256 counter. Simulate the counter.
- H, B** 7-56. Expand the full-featured HDL counter in Section 7-12 to a MOD-1024 counter. Simulate the counter.
- H, D, N** 7-57\* Design a recycling, MOD-16, down counter. The counter should have the following controls (from lowest to highest priority): an active-LOW count enable (*en*), an active-HIGH synchronous clear (*clr*), and active-LOW synchronous load (*ld*). Decode the terminal count when enabled by *en*. Simulate (functional) the counter. Be sure to verify the decoder operation.
- Use LPM\_COUNTER. Use any necessary logic gates.
  - Use an HDL.
- H, D, N** 7-58. Design a recycling, MOD-10, up/down counter. The counter will count up when *up* = 1 and will count down when *up* = 0. The counter should also have the following controls (from lowest to highest



priority): an active-HIGH count enable (*enable*), active-HIGH synchronous load (*load*), and an active-LOW asynchronous clear (*clear*). Decode the terminal count when enabled by *enable*. Simulate (functional) the counter.

- (a) Use LPM\_COUNTER. Use any necessary logic gates.
- (b) Use an HDL.

### SECTION 7-13

- H** 7-59\* Create a MOD-1000 BCD counter by cascading together three of the HDL BCD counter modules (described in Section 7-13). Simulate the counter.
- H** 7-60. Create a MOD-256 binary counter by cascading together two of the full-featured, MOD-16, HDL counter modules (described in Section 7-12). Simulate the counter.
- H, D, N** 7-61\* Design a synchronous, MOD-50 BCD counter by cascading a MOD-10 and a MOD-5 counter together. The MOD-50 counter should have an active-HIGH count enable (*enable*) and an active-LOW synchronous clear (*clr*). Be sure to include the terminal count detection for the one's digit to cascade with the ten's digit. Simulate (functional) the counter.
  - (a) Use LPM\_COUNTER. Use any necessary logic gates.
  - (b) Use an HDL.
- H, D, N** 7-62. Design a synchronous, MOD-100, BCD down counter by cascading two MOD-10, down counter modules together. The MOD-100 counter should have a synchronous parallel load (*load*). Simulate (functional) the counter.
  - (a) Use LPM\_COUNTER.
  - (b) Use an HDL.

### SECTION 7-14

- H** 7-63\* Modify the HDL description in Figure 7-60 or Figure 7-61 to add a rinse sequence after the clothes are washed. The new state machine sequence should be *idle* → *wash\_fill* → *wash\_agitate* → *wash\_spin* → *rinse\_fill* → *rinse\_agitate* → *rinse\_spin* → *idle*. Use hot water to wash, and cold water to rinse (add output bits to control two water valves). Simulate the modified HDL design.
- H** 7-64. Simulate the HDL traffic light controller design presented in Section 7-14.

## PART 2

### SECTIONS 7-15 THROUGH 7-16

- B** 7-65\* A set of 74ALS174 registers is connected as shown in Figure 7-114. What type of data transfer is performed with each register? Determine the output of each register when the  $\overline{MR}$  is pulsed momentarily LOW and after each of the indicated clock pulses (CP#) in Table 7-10. How many clock pulses must be applied before data that are input on *I5–I0* are available at *Z5–Z0*?

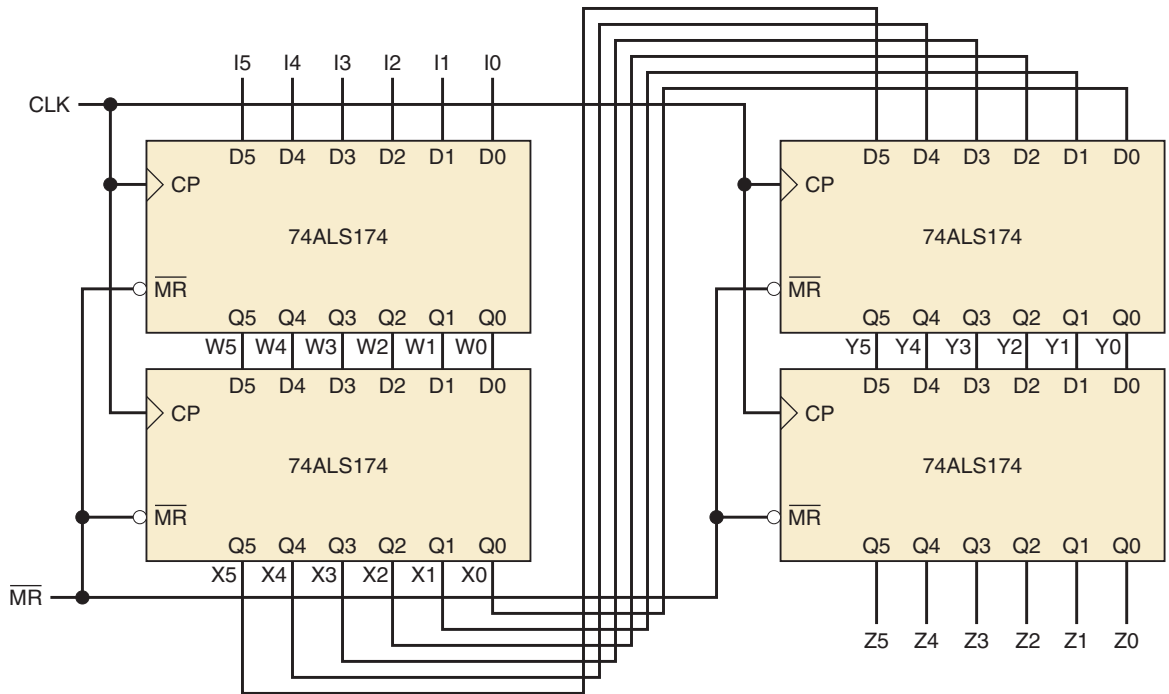


FIGURE 7-114 Problem 7-65.

TABLE 7-10

↑ CLK	$\overline{\text{MR}}$	I5-I10	W5-W0	X5-X0	Y5-Y0	Z5-Z0
X	0	101010				
CP1	1	101010				
CP2	1	010101				
CP3	1	000111				
CP4	1	111000				
CP5	1	011011				
CP6	1	001101				
CP7	1	000000				
CP8	1	000000				

- B** 7-66. Complete the timing diagram in Figure 7-115 for a 74HC174. How does the timing diagram show that the master reset is asynchronous?
- B** 7-67\* How many clock pulses will be needed to completely load eight bits of serial data into a 74ALS166? How does this relate to the number of flip-flops contained in the register?
- B** 7-68. Repeat Example 7-20 for the input waveforms given in Figure 7-116.
- 7-69\* Repeat Example 7-22 with  $D_5 = 1$  and the input waveforms given in Figure 7-117.
- 7-70. Apply the input waveforms given in Figure 7-118 to a 74ALS166 and determine the output produced.
- B** 7-71\* While examining the schematic for a piece of equipment, a technician or an engineer will often come across an IC that is unfamiliar. In such cases, it is often necessary to consult the manufacturer's data

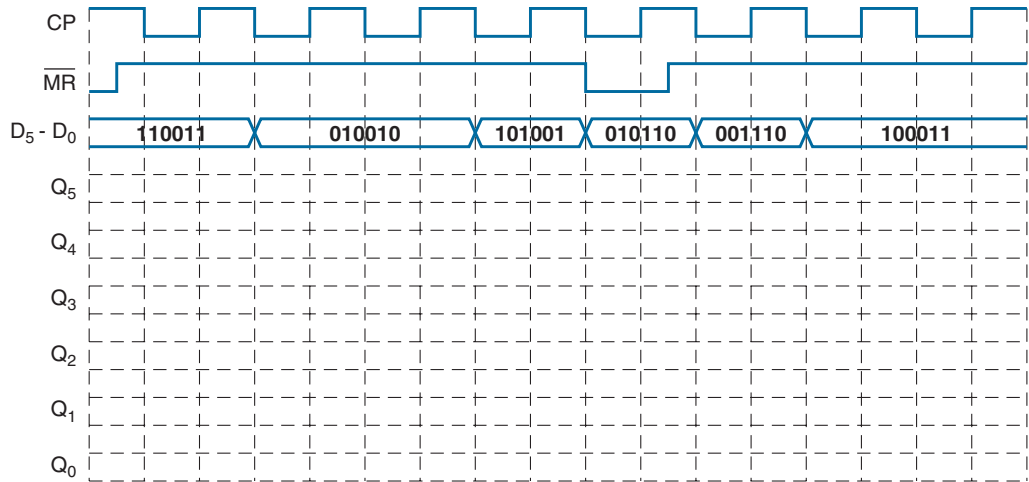


FIGURE 7-115 Problem 7-66.

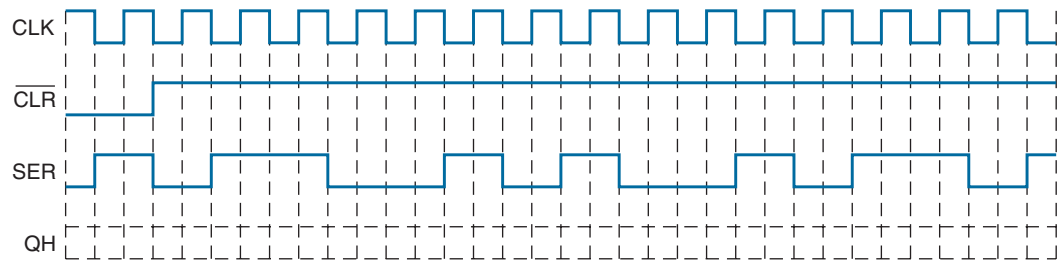


FIGURE 7-116 Problem 7-68.

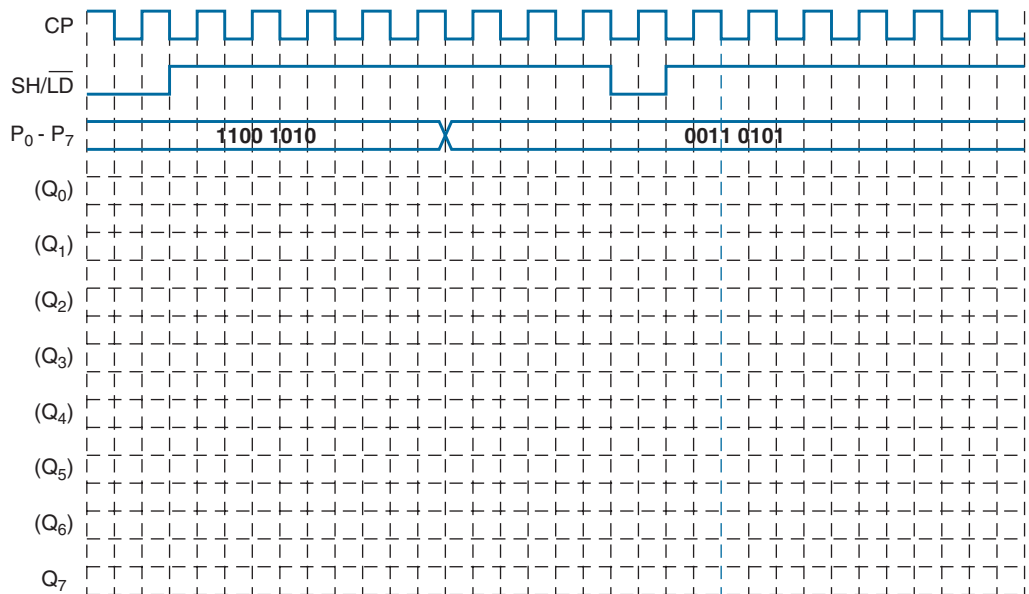
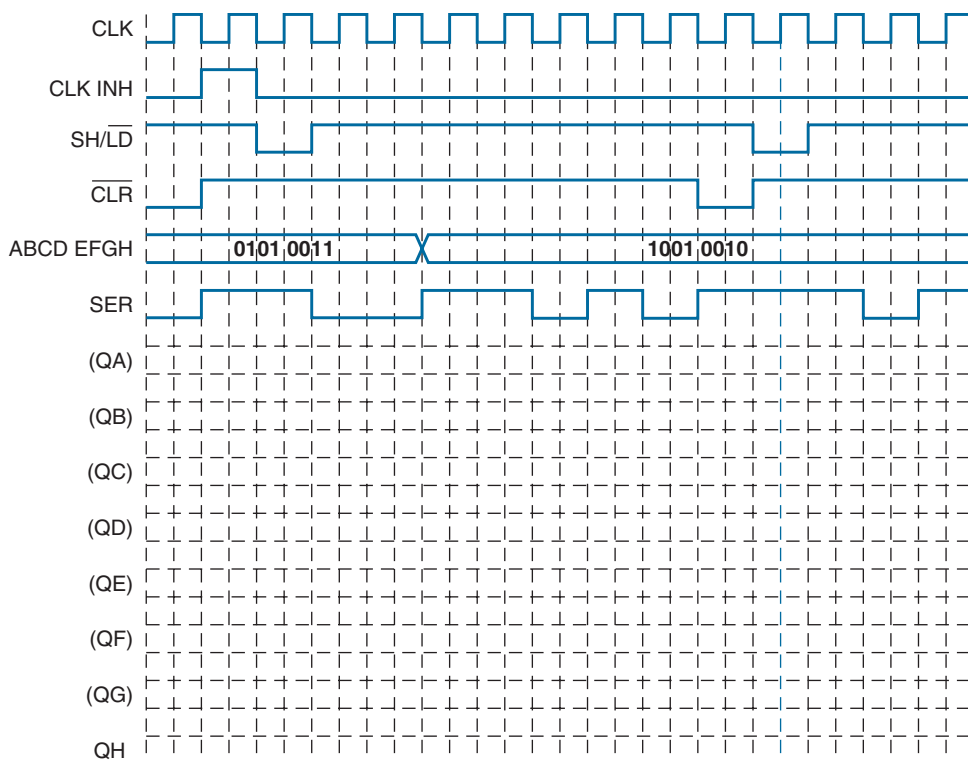


FIGURE 7-117 Problem 7-69.

FIGURE 7-118 Problem 7-70.



sheets for specifications on the device. Research the data sheet for the 74AS194 bidirectional universal shift register to answer the following questions:

- (a) Is the  $\overline{CLR}$  input asynchronous or synchronous?
- (b) *True or false:* When  $CLK$  is LOW, the  $S_0$  and  $S_1$  inputs have no effect on the register.
- (c) Assume the following conditions:

$$\begin{aligned}
 Q_A \ Q_B \ Q_C \ Q_D &= 1 \ 0 \ 1 \ 1 \\
 A \ B \ C \ D &= 0 \ 1 \ 1 \ 0 \\
 \overline{CLR} &= 1 \\
 S_R \ S_L &= 0 \\
 S_0 \ S_1 &= 1
 \end{aligned}$$

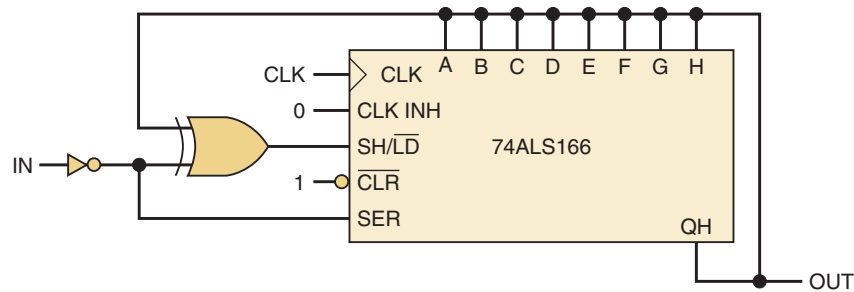
If  $S_0 = 0$  and  $S_1 = 1$ , determine the register outputs after one  $CLK$  pulse. After two  $CLK$  pulses. After three. After four.

- (d) Use the same conditions except  $S_0 = 1$  and  $S_1 = 0$  and repeat part (c).
- (e) Repeat part (c) with  $S_0 = 1$  and  $S_1 = 1$ .
- (f) Repeat part (c) with  $S_0 = 0$  and  $S_1 = 0$ .
- (g) Use the same conditions as in part (c), except assume that  $Q_A$  is connected to  $SL \ SER$ . What will be the register outputs after four  $CLK$  pulses?

**C** 7-72. Refer to Figure 7-119 to answer the following questions:

- (a) Which register function (load or shift) will be performed on the next clock if  $in = 1$  and  $out = 0$ ? What data value will be input when clocked?

FIGURE 7-119 Problem 7-72.



- Which register function (load or shift) will be performed on the next clock if  $in = 0$  and  $out = 1$ ? What data value will be input when clocked?
- Which register function (load or shift) will be performed on the next clock if  $in = 0$  and  $out = 0$ ? What data value will be input when clocked?
- Which register function (load or shift) will be performed on the next clock if  $in = 1$  and  $out = 1$ ? What data value will be input when clocked?
- What input condition will eventually (after several clock pulses) cause the output to switch states?
- To change the output logic level requires the new input condition to last for at least how many clock pulses?
- If the input signal changes levels and then goes back to its original logic level before the number of clock pulses specified in part (f), what happens to the output signal.
- Explain why this circuit can be used to debounce switches.

### SECTION 7-17

- 7-73\* Draw the diagram for a MOD-5 ring counter using J-K flip-flops. Make sure that the counter will start the proper count sequence when it is turned on.
- 7-74. Add one more J-K flip-flop to convert the MOD-5 ring counter in Problem 7-73 into a MOD-10 counter. Determine the sequence of states for this counter. This is an example of a decade counter that is not a BCD counter. Draw the decoding circuit for this counter.
- 7-75\* Draw the diagram for a MOD-10 Johnson counter using a 74HC164. Make sure that the counter will start the proper count sequence when it is turned on. Determine the count sequence for this counter and draw the decoding circuit needed to decode each of the 10 states. This is another example of a decade counter that is not a BCD counter.
- 7-76. The clock input to the Johnson counter in Problem 7-75 is 10 Hz. What is the frequency and duty cycle for each of the counter outputs?

### SECTION 7-18

- 7-77\* The MOD-10 counter in Figure 7-8(b) produces the count sequence 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, and repeats. Identify some possible fault conditions that might produce this result.

- T** 7-78. The MOD-10 counter in Figure 7-8(b) produces the count sequence 0000, 0101, 0010, 0111, 1000, 1101, 1010, 1111, and repeats. Identify some possible fault conditions that might produce this result.

### SECTIONS 7-19 AND 7-20

- N** 7-79\* Create an eight-bit SISO shift register. The serial input is called *ser* and the serial output is called *qout*. An active-LOW enable (*en*) controls the shift register. Simulate (functional) the design.
- (a) Use LPM\_SHIFTRREG. Use any necessary logic gates.  
(b) Use an HDL.
- N** 7-80. Create an eight-bit PIPO shift register. The data in is  $d[7..0]$  and the outputs are  $q[7..0]$ . An active-HIGH enable (*ld*) controls the shift register. Simulate (functional) the design.
- (a) Use LPM\_FF.  
(b) Use an HDL.
- N** 7-81\* Create an eight-bit PISO shift register. The data in is  $d[7..0]$  and the output is  $q0$ . The shift-register function is controlled by *sh\_ld* (*sh\_ld* = 0 to synchronously parallel load and *sh\_ld* = 1 to serial shift). The serial input while shifting should be a constant LOW. The register also should have an active-LOW asynchronous clear (*clrn*). Simulate (functional) the design.
- (a) Use LPM\_SHIFTRREG. Use any necessary logic gates.  
(b) Use an HDL.
- N** 7-82. Create an eight-bit SIPO shift register. The data in is *ser\_in* and the outputs are  $q[7..0]$ . The shift-register function is enabled by an active-HIGH control named *shift*. The shift register also has a higher priority active-HIGH synchronous clear (*clear*). Simulate (functional) the design.
- (a) Use LPM\_SHIFTRREG. Use any necessary logic gates.  
(b) Use an HDL.
- H** 7-83.\* Simulate the universal shift-register design from Example 7-27.
- H** 7-84. Create an eight-bit universal shift register by cascading two of the modules in Example 7-27. Simulate the design.

### SECTION 7-21

- H, D** 7-85\* Design a MOD-10, self-starting Johnson counter with an active-HIGH, asynchronous reset (*reset*) using an HDL. Simulate the design.
- H, D** 7-86. Sometimes a digital application may need a ring counter that recirculates a single zero instead of a single one. The ring counter would then have an active-LOW output instead of an active-HIGH. Design a MOD-8, self-starting ring counter with an active-LOW output using an HDL. The ring counter should also have an active-HIGH *hold* control to disable the counting. Simulate the design.

### SECTION 7-22

- H** 7-87\* Use Altera's simulator to test the nonretriggerable, level-sensitive, one-shot design example in either Figure 7-95 (AHDL) or 7-96

- (VHDL). Use a 1-kHz clock and create a 10-ms output pulse for the simulation. Verify that:
- The correct pulse width is created when triggered.
  - The output can be terminated early with the reset input.
  - The one-shot design is nonretriggerable and cannot be triggered again until it has timed out.
  - The trigger signal must last long enough for the clock to catch it.
  - The pulse width can be changed to a different value.
- H** 7-88. Modify the nonretriggerable, level-sensitive, one-shot design example from either Figure 7-95 (AHDL) or Figure 7-96 (VHDL) so that the one-shot is retriggerable but still level-sensitive. Simulate the design.

### DRILL QUESTION

- B** 7-89\* For each of the following statements, indicate the type(s) of counter being described.
- Each FF is clocked at the same time.
  - Each FF divides the frequency at its *CLK* input by 2.
  - The counting sequence is 111, 110, 101, 100, 011, 010, 001, 000.
  - The counter has 10 distinct states.
  - The total switching delay is the sum of the individual FFs' delays.
  - This counter requires no decoding logic.
  - The MOD number is always twice the number of FFs.
  - This counter divides the input frequency by its MOD number.
  - This counter can begin its counting sequence at any desired starting state.
  - This counter can count in any direction.
  - This counter can suffer from decoding glitches due to its propagation delays.
  - This counter only counts from 0 to 9.
  - This counter can be designed to count through arbitrary sequences by determining the logic circuit needed at each flip-flop's synchronous control inputs.

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

---

### PART 1

#### SECTION 7-1

1. False    2. 0000    3. 128

#### SECTION 7-2

1. Each FF adds its propagation delay to the total counter delay in response to a clock pulse.    2. MOD-256

#### SECTION 7-3

1. Can operate at higher clock frequencies and has more complex circuitry  
 2. Six FFs and four AND gates    3. ABCDE
-

**SECTION 7-4**

1.  $D$ ,  $C$ , and  $A$     2. True, because a BCD counter has 10 distinct states    3. 5 kHz

**SECTION 7-5**

1. In an up counter, the count is increased by 1 with each clock pulse; in a down counter, the count is decreased by 1 with each pulse.    2. Change connections to respective inverted outputs instead of  $Q$ s.

**SECTION 7-6**

1. It can be preset to any desired starting count.    2. Asynchronous presetting is independent of the clock input, while synchronous presetting occurs on the active edge of the clock signal.

**SECTION 7-7**

1.  $\overline{LOAD}$  is the control that enables the parallel loading of the data inputs  $D C B A$  ( $A = \text{LSB}$ ).    2.  $\overline{CLR}$  is the control that enables the resetting of the counter to 0000.    3. True    4. All control inputs ( $\overline{CLR}$ ,  $\overline{LOAD}$ ,  $ENT$ , and  $ENP$ ) on the 74162 must be HIGH.    5.  $\overline{LOAD} = 1$ ,  $\overline{CTEN} = 0$ , and  $D/\overline{U} = 1$  to count down.    6. 74HC163: 0 to 65,535; 74ALS190: 0 to 9999 or 9999 to 0.

**SECTION 7-8**

1. Sixty-four    2. A six-input NAND gate with inputs  $A$ ,  $B$ ,  $C$ ,  $\overline{D}$ ,  $E$ , and  $\overline{F}$ .

**SECTION 7-9**

1. We will not have to deal with transient states and possible glitches in output waveforms.    2. PRESENT state/NEXT state table    3. The gates control the count sequence.    4. Unused states all lead back to the count sequence of the counter.

**SECTION 7-10**

1. See text.    2. It associates every possible PRESENT state with its desired NEXT state.    3. It shows the necessary levels at each flip-flop's synchronous input to produce the counter's state transitions.    4. True

**SECTION 7-11**

1. Arithmetic    2. Use the MegaWizard Manager    3. An asynchronous clear will occur as soon (after a short propagation delay) as the control signal goes active while a synchronous clear will occur at the next clock edge after asserting the control.    4.  $Cout$  will automatically decode the last (or terminal) state in the count sequence.    5.  $Cin$  will also enable/disable the  $cout$  signal.

**SECTION 7-12**

1. PRESENT state/NEXT state tables    2. The desired NEXT state    3. AHDL:  
`ff[ ].clk = !clock`  
 VHDL:  
`IF (clock = '0' AND clock'EVENT) THEN`  
 4. Behavioral description    5. Asynchronous clear causes the counter to clear immediately. Synchronous load occurs on the next active clock edge.    6. AHDL: Use `.clrn` port on FFs; VHDL: Define clear function before checking for clock edge    7. By the order of evaluation in an IF statement.

**SECTION 7-13**

1. Both HDLs can use a block diagram to connect modules; VHDL can also use a text file that describes the connections between components.    2. A bus is a



collection of signal lines; it is represented graphically by a heavy-weight line  
 3. Count enable and terminal count decoding

### SECTION 7-14

1. A counter is commonly used to count events, while a state machine is commonly used to control events. 2. A state machine can be described using symbols to describe its states rather than actual binary states. 3. The compiler assigns the optimal values to minimize the circuitry. 4. The description is much easier to write and understand.

### PART 2

### SECTION 7-16

1. Parallel in/serial out 2. True 3. Serial in/parallel out 4. Serial in/serial out 5. The 74165 uses asynchronous parallel data transfer; the 74174 uses synchronous parallel data transfer. 6. A HIGH prevents shifting on CPs. 7. Compare to corresponding outputs in the figure.

### SECTION 7-17

1. Ring counter 2. Johnson counter 3. The inverted output of the last FF is connected to the input of the first FF. 4. (a)False (b) True (c) True 5. Sixteen; eight

### SECTION 7-19

1. PIPO, SISO, PISO, SIPO (all 4)

### SECTION 7-20

1. AHDL:

```
reg[].d = (reg[6..0], dat)
```

VHDL:

```
reg: = reg(6 DOWNT0 0) & dat
```

2. Because the register may continue to receive clock edges during hold

### SECTION 7-21

1. It can start in any state, but it will eventually reach the desired ring sequence. 2. Lines 11 and 12 3. Lines 12 and 13

### SECTION 7-22

1. The reset input 2. The clock frequency and the delay value loaded into the counter 3. Synchronously 4. The output pulse width is very consistent. 5. The output pulse responds to the trigger edge immediately. 6. The state of the trigger on the current clock edge and its state on the previous edge.

*This page intentionally left blank*



# INTEGRATED-CIRCUIT LOGIC FAMILIES

## ■ OUTLINE

- 8-1 Digital IC Terminology
- 8-2 The TTL Logic Family
- 8-3 TTL Data Sheets
- 8-4 TTL Series Characteristics
- 8-5 TTL Loading and Fan-Out
- 8-6 Other TTL Characteristics
- 8-7 MOS Technology
- 8-8 Complementary MOS Logic
- 8-9 CMOS Series Characteristics
- 8-10 Low-Voltage Technology
- 8-11 Open-Collector/Open-Drain Outputs
- 8-12 Tristate (Three-State) Logic Outputs
- 8-13 High-Speed Bus Interface Logic
- 8-14 CMOS Transmission Gate (Bilateral Switch)
- 8-15 IC Interfacing
- 8-16 Mixed-Voltage Interfacing
- 8-17 Analog Voltage Comparators
- 8-18 Troubleshooting
- 8-19 Characteristics of an FPGA

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Read and understand digital IC terminology as specified in manufacturers' data sheets.
- Compare the characteristics of standard TTL and the various TTL series.
- Determine the fan-out for a particular logic device.
- Use logic devices with open-collector outputs.
- Analyze circuits containing tristate devices.
- Compare the characteristics of the various CMOS series.
- Analyze circuits that use a CMOS bilateral switch to allow a digital system to control analog signals.
- Describe the major characteristics of and differences among TTL, ECL, MOS, and CMOS logic families.
- Describe the various considerations that are required when interfacing digital circuits from different logic families.
- Use voltage comparators to allow a digital system to be controlled by analog signals.
- Use a logic pulser and a logic probe as digital circuit troubleshooting tools.

## ■ INTRODUCTION

As we described in Chapter 4, digital IC technology has advanced rapidly from small-scale integration (SSI), with fewer than 12 gates per chip; through medium-scale integration (MSI), with 12 to 99 equivalent gates per chip; on to large-scale and very large scale integration (LSI and VLSI, respectively), which can have tens of thousands of gates per chip; and to ultra-large-scale integration (ULSI), with over 100,000 gates per chip, and giga-scale integration (GSI), with 1 million or more gates.

Most of the reasons that modern digital systems use integrated circuits are obvious. ICs pack a lot more circuitry in a small package, so that the overall size of almost any digital system is reduced. The cost is dramatically reduced because of the economies of mass-producing large volumes of similar devices. Some of the other advantages are not so apparent.

ICs have made digital systems more reliable by reducing the number of external interconnections from one device to another. Before we had ICs, every circuit connection was from one discrete component (transistor, diode, resistor, etc.) to another. Now most of the connections are internal to the ICs, where they are protected from poor soldering, breaks or shorts in connecting paths on a circuit board, and other physical problems. ICs have also drastically reduced the amount of electrical power needed to perform a given function because their miniature circuitry typically requires less power than their discrete counterparts. In addition to the

savings in power-supply costs, this reduction in power has also meant that a system does not require as much cooling.

There are some things that ICs cannot do. They cannot handle very large currents or voltages because the heat generated in such small spaces would cause temperatures to rise beyond acceptable limits. In addition, ICs cannot easily implement certain electrical devices such as inductors, transformers, and large capacitors. For these reasons, ICs are principally used to perform low-power circuit operations that are commonly called *information processing*. That is precisely the job of the digital logic circuits that we have been studying. The digital circuit will make decisions based on the input conditions that are present. When devices that require higher power levels must be controlled by a logic circuit, some type of interfacing circuit will be needed. The interfacing circuit will typically use discrete components or special power IC chips.

With the widespread use of ICs comes the necessity to know and understand the electrical and timing characteristics of the most common IC logic families. Remember that the various logic families differ in the major components that they use in their circuitry. TTL and ECL use *bipolar* transistors as their major circuit element; PMOS, NMOS, and CMOS use unipolar MOSFET transistors as their principal component. These various logic families have differing electrical characteristics that must be considered when designing digital systems. The electrical characteristics of a logic family are dependent upon both the transistor type and on the internal circuits of the chips. Numerous digital IC subfamilies have been developed over time to provide improvements in system power consumption and speed. We have seen a continuous (and continuing) evolution from high-power/low-speed devices to high-speed/low-power chips.

In this chapter, we will present the important characteristics of each of the IC families and their subfamilies. The most important point is understanding the nature of the input circuitry and output circuitry for each logic family. Once these are understood, you will be much better prepared to do analysis, troubleshooting, and some design of digital circuits that contain any combination of IC families. We will study the inner workings of devices in each family with the simplest circuitry that conveys the critical characteristics of all members of the family.

## 8-1 DIGITAL IC TERMINOLOGY

---

### OUTCOMES

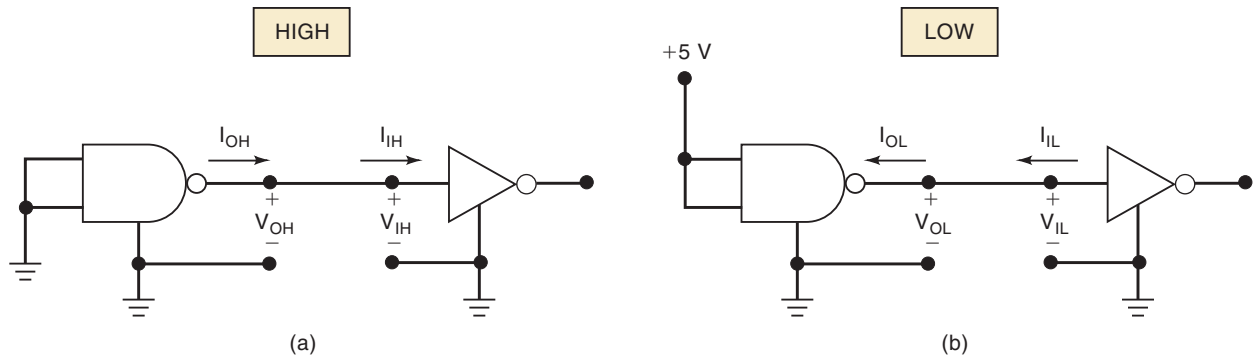
*Upon completion of this section, you will be able to:*

- Define, interpret, and measure commonly used parameters and parameter designations from data sheets.
- Look up and interpret performance information from data sheets.

Although there are many digital IC manufacturers, much of the nomenclature and terminology is fairly standardized. The most useful terms are defined and discussed below.

### Current and Voltage Parameters (See Figure 8-1)

- $V_{IH(\min)}$ —**High-Level Input Voltage**. The minimum voltage level required for a logical 1 at an *input*. Any voltage below this level will not be accepted as a HIGH by the logic circuit.
-



**FIGURE 8-1** Currents and voltages in the two logic states.

- **$V_{IL}(\text{max})$ —Low-Level Input Voltage.** The maximum voltage level required for a logic 0 at an *input*. Any voltage above this level will not be accepted as a LOW by the logic circuit.
- **$V_{OH}(\text{min})$ —High-Level Output Voltage.** The minimum voltage level at a logic-circuit *output* in the logical 1 state under defined load conditions.
- **$V_{OL}(\text{max})$ —Low-Level Output Voltage.** The maximum voltage level at a logic-circuit *output* in the logical 0 state under defined load conditions.
- **$I_{IH}$ —High-Level Input Current.** The current that flows into an input when a specified high-level voltage is applied to that input.
- **$I_{IL}$ —Low-Level Input Current.** The current that flows into an input when a specified low-level voltage is applied to that input.
- **$I_{OH}$ —High-Level Output Current.** The current that flows from an output in the logical 1 state under specified load conditions.
- **$I_{OL}$ —Low-Level Output Current.** The current that flows from an output in the logical 0 state under specified load conditions.

*Note:* The actual current directions may be opposite to those shown in Figure 8-1, depending on the logic family. All descriptions of current flow in this text refer to conventional current flow (from higher potential to lower potential). In keeping with the conventions of most data books, current flowing into a node or device is considered positive, and current flowing out of a node or device is considered negative.

### Fan-Out

In general, a logic-circuit output is required to drive several logic inputs. Sometimes all ICs in the digital system are from the same logic family, but many systems have a mix of various logic families. The **fan-out** (also called *loading factor*) is defined as the *maximum* number of logic inputs that an output can drive reliably. For example, a logic gate that is specified to have a fan-out of 10 can drive 10 logic inputs. If this number is exceeded, the output logic-level voltages cannot be guaranteed. Obviously, fan-out depends on the nature of the input devices that are connected to an output. Unless a different logic family is specified as the load device, fan-out is assumed to refer to load devices of the same family as the driving output.

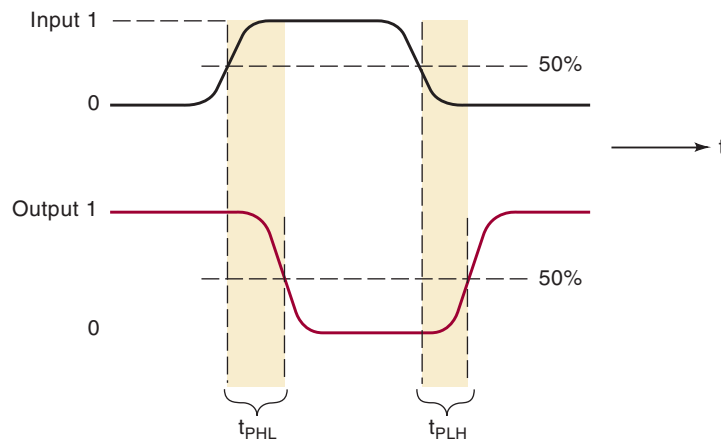
## Propagation Delays

A logic signal always experiences a delay in going through a circuit. The two propagation delay times are defined as follows:

- $t_{PLH}$ . Delay time in going from logical 0 to logical 1 state (LOW to HIGH)
- $t_{PHL}$ . Delay time in going from logical 1 to logical 0 state (HIGH to LOW)

Figure 8-2 illustrates these propagation delays for an INVERTER. Note that  $t_{PHL}$  is the delay in the output's response as it goes from HIGH to LOW. It is measured between the 50% points on the input and output transitions. The  $t_{PLH}$  value is the delay in the output's response as it goes from LOW to HIGH.

**FIGURE 8-2** Propagation delays.



In some logic circuits,  $t_{PHL}$  and  $t_{PLH}$  are not the same value, and both will vary depending on capacitive loading conditions. The values of propagation times are used as a measure of the relative speed of logic circuits. For example, a logic circuit with values of 10 ns is a faster logic circuit than one with values of 20 ns under specified load conditions.

## Power Requirements

Every IC requires a certain amount of electrical power to operate. This power is supplied by one or more power-supply voltages connected to the power pin(s) on the chip labeled  $V_{CC}$  (for TTL) or  $V_{DD}$  (for MOS devices).

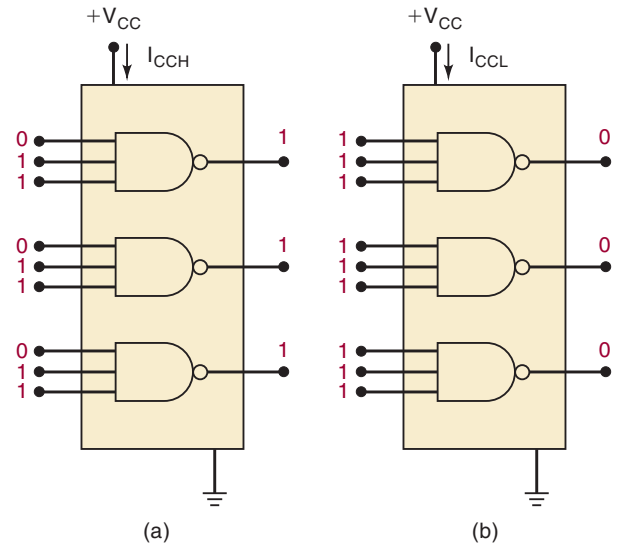
The amount of power that an IC requires is determined by the current,  $I_{CC}$  (or  $I_{DD}$ ), that it draws from the  $V_{CC}$  (or  $V_{DD}$ ) supply, and the actual power is the product  $I_{CC} \times V_{CC}$  (or  $I_{DD} \times V_{DD}$ ). For many ICs, the current drawn from the supply varies depending on the logic states of the circuits on the chip. For example, Figure 8-3(a) shows a NAND chip where *all* of the gate *outputs* are HIGH. The current drain on the  $V_{CC}$  supply for this case is called  $I_{CCH}$ . Likewise, Figure 8-3(b) shows the current when *all* of the gate *outputs* are LOW. This current is called  $I_{CCL}$ . The values are always measured with the outputs open circuit (no load) because the size of the load will also have an effect on  $I_{CCH}$ .

In some logic circuits,  $I_{CCH}$  and  $I_{CCL}$  will be different values. For these devices, the average current is computed based on the assumption that gate outputs are LOW half the time and HIGH half the time.

$$I_{CC(\text{avg})} = \frac{I_{CCH} + I_{CCL}}{2}$$

This equation can be rewritten to calculate average power dissipated:

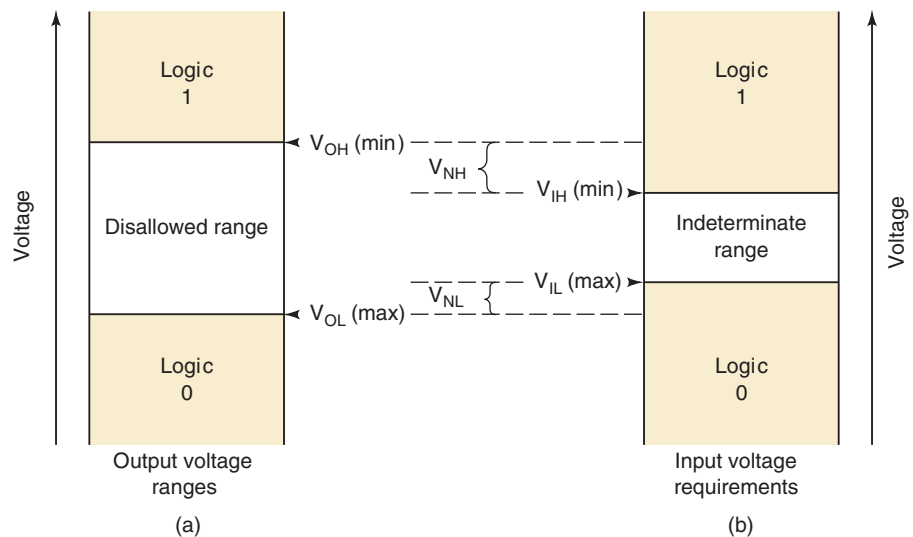
$$P_D(\text{avg}) = I_{CC(\text{avg})} \times V_{CC}$$

**FIGURE 8-3**  $I_{CCH}$  and  $I_{CCL}$ .

### Noise Immunity

Stray electric and magnetic fields can induce voltages on the connecting wires between logic circuits. These unwanted, spurious signals are called *noise* and can sometimes cause the voltage at the input to a logic circuit to drop below  $V_{IH}(\text{min})$  or rise above  $V_{IL}(\text{max})$ , which could produce unpredictable operation. The **noise immunity** of a logic circuit refers to the circuit's ability to tolerate noise without causing spurious changes in the output voltage. A quantitative measure of noise immunity is called **noise margin** and is illustrated in Figure 8-4.

Figure 8-4(a) is a diagram showing the range of voltages that can occur at a logic-circuit output. Any voltages greater than  $V_{OH}(\text{min})$  are considered a logic 1, and any voltages lower than  $V_{OL}(\text{max})$  are considered a logic 0. Voltages in the disallowed range should not appear at a logic-circuit output under normal conditions. Figure 8-4(b) shows the voltage requirements at a logic-circuit input. The logic circuit responds to any input greater than  $V_{IH}(\text{min})$  as a logic 1, and it responds to voltages lower than  $V_{IL}(\text{max})$  as a logic 0. Voltages in the indeterminate range produce an unpredictable response and should not be used.

**FIGURE 8-4** DC noise margins.



The *HIGH-state noise margin*  $V_{NH}$  is defined as

$$V_{NH} = V_{OH}(\min) - V_{IH}(\min) \quad (8-1)$$

and is illustrated in Figure 8-4.  $V_{NH}$  is the difference between the lowest possible HIGH output and the minimum input voltage required for a HIGH. When a HIGH logic output is driving a logic-circuit input, any negative noise spikes greater than  $V_{NH}$  appearing on the signal line can cause the voltage to drop into the indeterminate range, where unpredictable operation can occur.

The *LOW-state noise margin*  $V_{NL}$  is defined as

$$V_{NL} = V_{IL}(\max) - V_{OL}(\max) \quad (8-2)$$

and it is the difference between the largest possible LOW output and the maximum input voltage required for a LOW. When a LOW logic output is driving a logic input, any positive noise spikes greater than  $V_{NL}$  can cause the voltage to rise into the indeterminate range.

### EXAMPLE 8-1

The input/output voltage specifications for the standard TTL family are listed in Table 8-1. Use these values to determine the following.

- The maximum-amplitude noise spike that can be tolerated when a HIGH output is driving an input.
- The maximum-amplitude noise spike that can be tolerated when a LOW output is driving an input.

**TABLE 8-1** I/O voltage specifications.

Parameter	Min (V)	Typical (V)	Max (V)
$V_{OH}$	2.4	3.4	
$V_{OL}$		0.2	0.4
$V_{IH}$	2.0*		
$V_{IL}$			0.8*

\*Normally only the minimum  $V_{IH}$  and maximum  $V_{IL}$  values are given.

### Solution

- When an output is HIGH, it may be as low as  $V_{OH}(\min) = 2.4 \text{ V}$ . The minimum voltage that an input responds to as a HIGH is  $V_{IH}(\min) = 2.0 \text{ V}$ . A negative noise spike can drive the actual voltage below 2.0 V if its amplitude is greater than

$$\begin{aligned} V_{NH} &= V_{OH}(\min) - V_{IH}(\min) \\ &= 2.4 \text{ V} - 2.0 \text{ V} = 0.4 \text{ V} \end{aligned}$$

- When an output is LOW, it may be as high as  $V_{OL}(\max) = 0.4 \text{ V}$ . The maximum voltage that an input responds to as a LOW is  $V_{IL}(\max) = 0.8 \text{ V}$ . A positive noise spike can drive the actual voltage above the 0.8-V level if its amplitude is greater than

$$\begin{aligned} V_{NL} &= V_{IL}(\max) - V_{OL}(\max) \\ &= 0.8 \text{ V} - 0.4 \text{ V} = 0.4 \text{ V} \end{aligned}$$

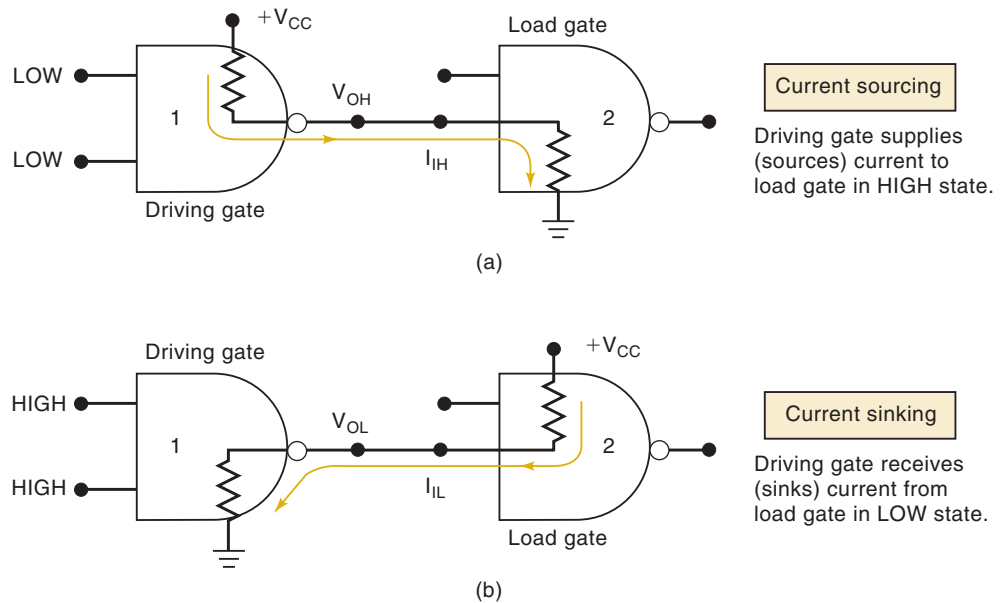
## Invalid Voltage Levels

For proper operation the input voltage levels to a logic circuit must be kept outside the indeterminate range shown in Figure 8-4(b); that is, they must be either lower than  $V_{IL}(\max)$  or higher than  $V_{IH}(\min)$ . For the standard TTL specifications given in Example 8-1, this means that the input voltage must be less than 0.8 V or greater than 2.0 V. An input voltage between 0.8 and 2.0 V is considered an *invalid* voltage that will produce an unpredictable output response, and so must be avoided. In normal operation, a logic input voltage will not fall into the invalid region because it comes from a logic output that is within the stated specifications. However, when this logic output is malfunctioning or is being overloaded (i.e., its fan-out is being exceeded), then its voltage may be in the invalid region. Invalid voltage levels in a digital circuit can also be caused by power-supply voltages that are outside the acceptable range. It is important to know the valid voltage ranges for the logic family being used so that invalid conditions can be recognized when testing or troubleshooting.

## Current-Sourcing and Current-Sinking Action

Logic families can be described according to how current flows between the output of one logic circuit and the input of another. Figure 8-5(a) illustrates **current-sourcing** action. When the output of gate 1 is in the HIGH state, it supplies a current  $I_{IH}$  to the input of gate 2, which acts essentially as a resistance to ground. Thus, the output of gate 1 is acting as a *source* of current for the gate 2 input. We can think of it as being like a faucet that acts as a *source* of water.

**FIGURE 8-5** Comparison of current-sourcing and current-sinking actions.



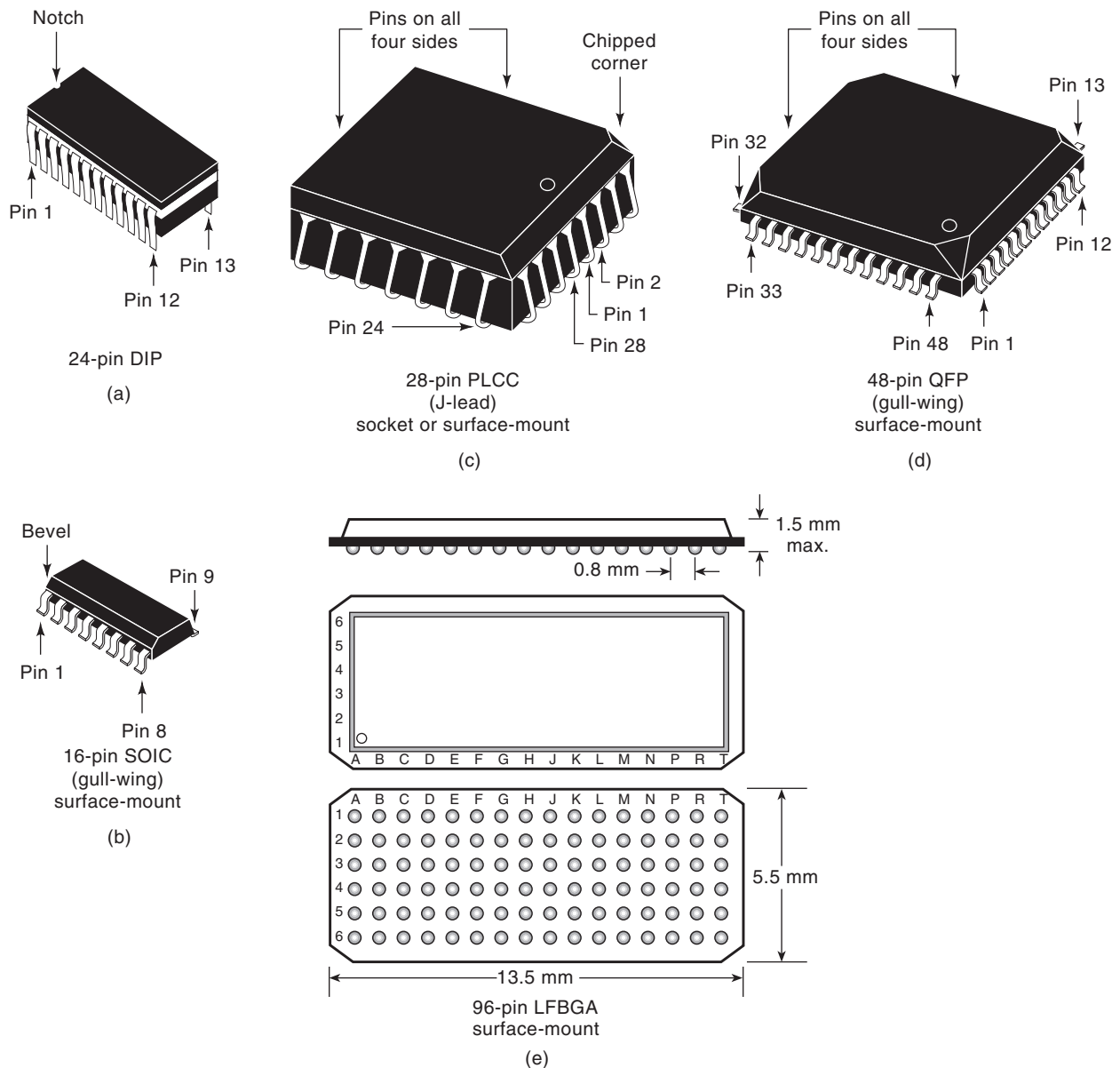
**Current-sinking** action is illustrated in Figure 8-5(b). Here the input circuitry of gate 2 is represented as a resistance tied to  $+V_{CC}$ , the positive terminal of a power supply. When the gate 1 output goes to its LOW state, current will flow in the direction shown from the input circuit of gate 2 back through the output resistance of gate 1 to ground. In other words, in the LOW state, the circuit output that drives the input of gate 2 must be able to *sink* a current,  $I_{IL}$ , coming from that input. We can think of this as acting like a *sink* into which water is flowing.

The distinction between current sourcing and current sinking is an important one, which will become more apparent as we examine the various logic families.

## IC Packages

Developments and advancements in integrated circuits continue at a rapid pace. The same is true of IC packaging. There are various types of packages, which differ in physical size, the environmental and power-consumption conditions under which the device can be operated reliably, and the way in which the IC package is mounted to the circuit board. Figure 8-6 shows seven representative IC packages.

The package in Figure 8-6(a) is the **DIP** (dual-in-line package), which has been around for a long time. Its pins (or leads) run down the two long sides of the rectangular package. The device shown is a 24-pin DIP. Note the



**FIGURE 8-6** Common IC packages.

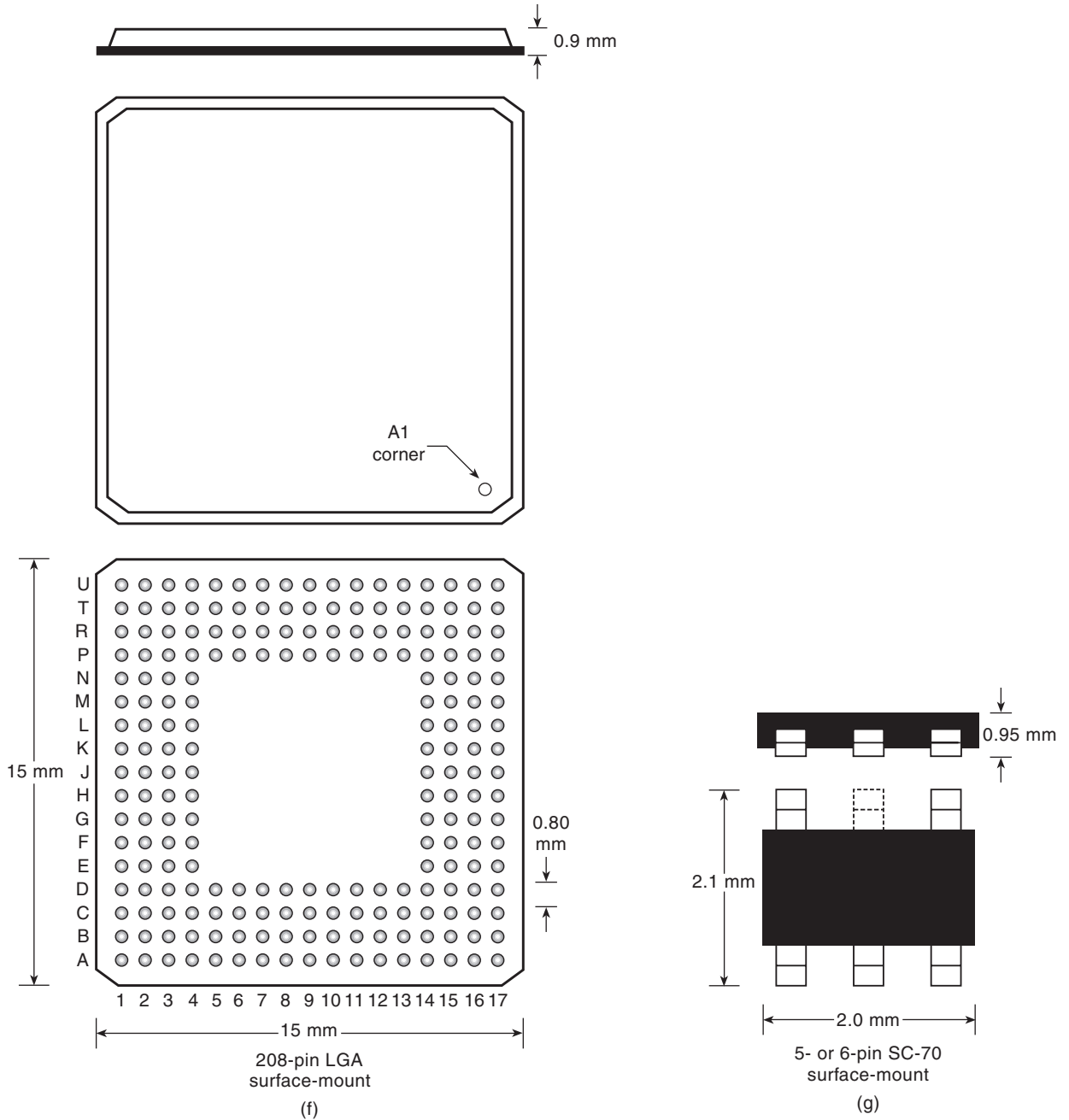


FIGURE 8-6 Continued

presence of the notch on one end, which is used to locate pin 1. Some DIPs use a small dot on the top surface of the package to locate pin 1. The leads extend straight out of the DIP package so that the IC can be plugged into an IC socket or inserted into holes drilled through a printed circuit board. The spacing between pins (**lead pitch**) is typically 100 mils (a mil is a thousandth of an inch).

Nearly all new circuit boards that are produced using automated manufacturing equipment have moved away from using DIP packages whose leads are inserted through holes in the board. Current manufacturing methods use **surface-mount technology (SMT)**, which places an IC onto

conductive pads on the surface of the board. They are held in place by a solder paste, and the entire board is heated to create a soldered connection. The precision of the placement machine allows for very tight lead spacing. The leads on these surface-mount packages are bent out from the plastic case, providing adequate surface area for the solder joint. The shape of these leads has resulted in the nickname of “gull-wing” package. Many different packages are available for surface-mount devices. Some of the most common packages used for logic ICs are shown in Figure 8-6. The oldest types of surface-mount packages are the various small-outline packages such as the small-outline integrated circuit (SOIC) shown in Figure 8-6(b). Table 8-2 gives the definition of each abbreviation along with its dimensions.

**TABLE 8-2** IC packages.

Abbreviation	Package Name	Height	Lead Pitch
DIP	Dual-in-line package	200 mils (5.1 mm)	100 mils (2.54 mm)
SOIC	Small outline integrated circuit	2.65 mm	50 mils (1.27 mm)
SSOP	Shrink small outline package	2.0 mm	0.65 mm
TSSOP	Thin shrink small outline package	1.1 mm	0.65 mm
TVSOP	Thin very small outline package	1.2 mm	0.4 mm
PLCC	Plastic leaded chip carrier	4.5 mm	1.27 mm
QFP	Quad flat pack	4.5 mm	0.635 mm
TQFP	Thin quad flat pack	1.6 mm	0.5 mm
LFBGA	Low-profile fine-pitch ball grid array	1.5 mm	0.8 mm
LGA	Land grid array	0.9 mm	0.8 mm

The need for more and more connections to a complex IC has resulted in another very popular package that has pins on all four sides of the chip. The PLCC has J-shaped leads that curl under the IC, as shown in Figure 8-6(c). These devices can be surface-mounted to a circuit board but can also be placed in a special PLCC socket. This is commonly used for components that are likely to need to be replaced for repair or upgrade, such as programmable logic devices or central processing units in computers. The QFP and TQFP packages have pins on all four sides in a gull-wing surface-mount package, as shown in Figure 8-6(d). The ball grid array (BGA) shown in Figure 8-6(e) is a surface-mount package that offers even more density. The pin grid array (PGA) is a similar package that is used when components must be in a socket to allow easy removal. The PGA has a long pin instead of a contact ball (BGA) at each position in the grid. The land grid array (LGA) package in Figure 8-6(f) is essentially a BGA package without the solder balls attached.

The proliferation of small, handheld consumer equipment such as digital video cameras, cellular phones, computers (PDAs), portable audio systems, and other devices has created a need for logic circuits in very small packages. Logic gates are available in individual surface-mount packages containing one, two, or three gates (1G, 2G, 3G, respectively). These devices may have as few as five or six pins (power, ground, two to three inputs, and an output), such as the example package shown in Figure 8-6(g), and take up less space than an individual letter on this page.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Define each of the following:  $V_{OH}$ ,  $V_{IL}$ ,  $I_{OL}$ ,  $I_{IH}$ ,  $t_{PLH}$ ,  $t_{PHL}$ ,  $I_{CCL}$ ,  $I_{CCH}$ .
2. *True or false:* If a logic circuit has a fan-out of 5, the circuit has five outputs.
3. *True or false:* The HIGH-stage noise margin is the difference between  $V_{IH}(\min)$  and  $V_{CC}$ .
4. Describe the difference between current sinking and current sourcing.
5. Which IC package can be plugged into sockets?
6. Which package has leads bent under the IC?
7. How do surface-mount packages differ from DIPs?
8. Will a standard TTL device work with an input level of 1.7 V?

## 8-2 THE TTL LOGIC FAMILY

### OUTCOMES

Upon completion of this section, you will be able to:

- Relate bipolar transistor technology and circuit configuration to operating characteristics for TTL.
- Relate logic function to circuit operation using bipolar transistor circuits.

At this writing, many small- to medium-scale ICs (SSI and MSI) can still be obtained in the standard **TTL** technology series that has been available for over 45 years. This original series of devices and their descendants in the TTL family have had a tremendous influence on the characteristics of all logic devices today. Successive generations of TTL logic were developed over about two decades with each generation providing gradual improvements in speed and power consumption. Even though the bipolar TTL family as a whole is on the decline, we will begin our discussion of logic ICs with the devices that shaped digital technology.

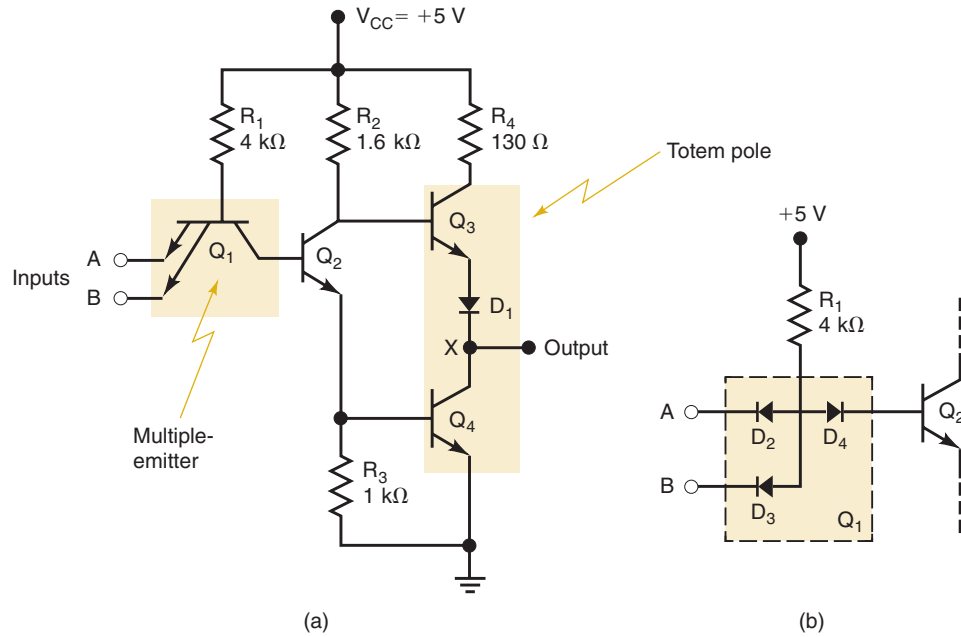
The basic TTL logic circuit is the NAND gate, shown in Figure 8-7(a). Even though the standard TTL family is nearly obsolete, we can learn a great deal about the more current family members by studying the original circuitry in its simplest form. The characteristics of TTL inputs come from the multiple-emitter (diode junction) configuration of transistor  $Q_1$ . Forward biasing either (or both) of these diode junctions will turn on  $Q_1$ . Only when all junctions are reverse biased will the transistor be off. This *multiple-emitter* input transistor can have up to eight emitters for an eight-input NAND gate.

Also note that on the output side of the circuit, transistors  $Q_3$  and  $Q_4$  are in a **totem-pole** arrangement. The totem pole is made up of two transistor switches,  $Q_3$  and  $Q_4$ . The job of  $Q_3$  is to connect  $V_{CC}$  to the output, making a logic HIGH. The job of  $Q_4$  is to connect the output to ground, making a logic LOW. As we will see shortly, in normal operation, either  $Q_3$  or  $Q_4$  will be conducting, depending on the logic state of the output.

### Circuit Operation—LOW State

Although this circuit looks extremely complex, we can simplify its analysis somewhat by using the diode equivalent of the multiple-emitter transistor  $Q_1$ , as shown in Figure 8-7(b). Diodes  $D_2$  and  $D_3$  represent the two E–B junctions of  $Q_1$ , and  $D_4$  is the collector-base (C–B) junction. In the following analysis, we will use this representation for  $Q_1$ .

**FIGURE 8-7** (a) Basic TTL NAND gate; (b) diode equivalent for  $Q_1$ .



First, let's consider the case where the output is LOW. Figure 8-8(a) shows this situation with inputs A and B both at +5 V. The +5 V at the cathodes of  $D_2$  and  $D_3$  will turn these diodes off, and they will conduct almost no current. The +5 V supply will push current through  $R_1$  and  $D_4$  into the base of  $Q_2$ , which turns on. Current from  $Q_2$ 's emitter will flow into the base of  $Q_4$  and turn  $Q_4$  on. At the same time, the flow of  $Q_2$  collector current produces a voltage drop across  $R_2$  that reduces  $Q_2$ 's collector voltage to a low value that is insufficient to turn  $Q_3$  on.

The voltage at  $Q_2$ 's collector is shown as approximately 0.8 V. This is because  $Q_2$ 's emitter is at 0.7 V relative to ground due to  $Q_4$ 's E–B forward voltage, and  $Q_2$ 's collector is at 0.1 V relative to its emitter due to  $V_{CE}(\text{sat})$ . This 0.8 V at  $Q_3$ 's base is not enough to forward-bias both  $Q_3$ 's E–B junction and diode  $D_1$ . In fact,  $D_1$  is needed to keep  $Q_3$  off in this situation.

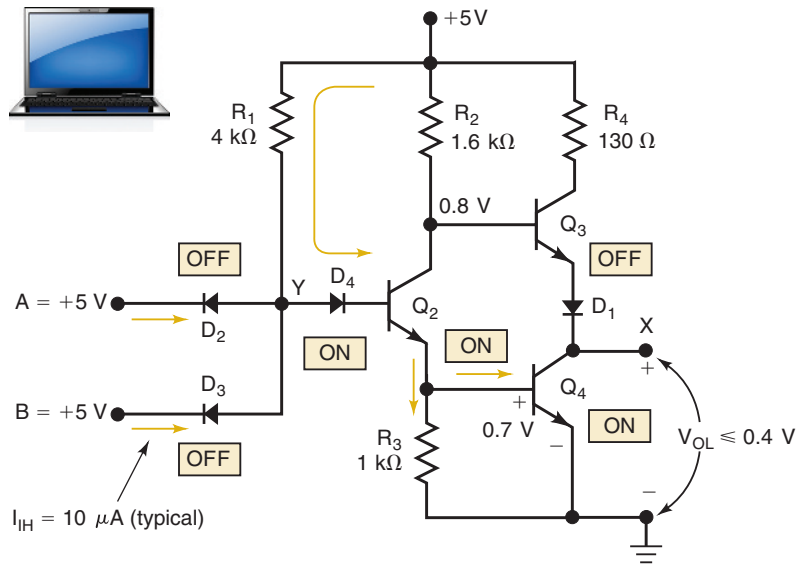
With  $Q_4$  on, the output terminal, X, will be at a very low voltage because  $Q_4$ 's ON-state resistance will be low (1 to 25  $\Omega$ ). Actually, the output voltage,  $V_{OL}$ , will depend on how much collector current  $Q_4$  conducts. With  $Q_3$  off, there is no current coming from the +5 V terminal through  $R_4$ . As we shall see,  $Q_4$ 's collector current will come from the TTL inputs that terminal X is connected to.

It is important to note that the HIGH inputs at A and B will have to supply only a very small diode leakage current. Typically, this current  $I_{IH}$  is only around 10  $\mu\text{A}$  at room temperature.

### Circuit Operation—HIGH State

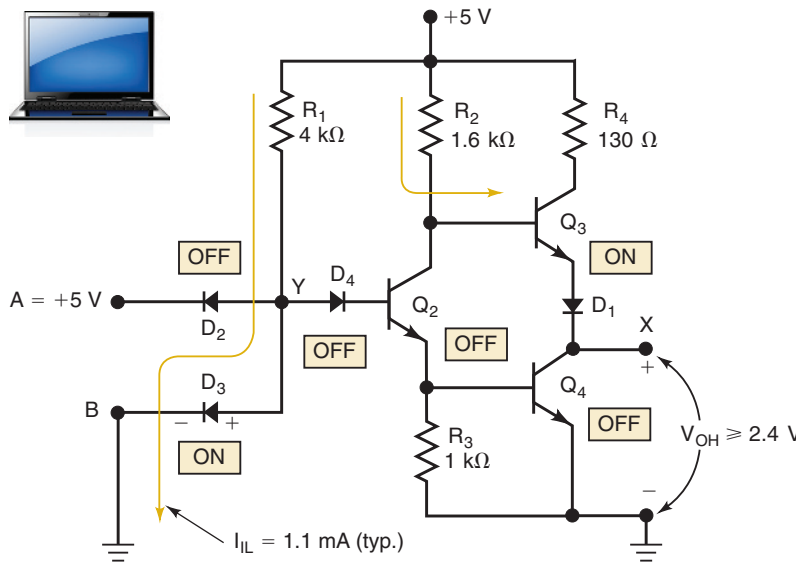
Figure 8-8(b) shows the situation where the circuit output is HIGH. This situation can be produced by connecting either or both inputs LOW. Here, input B is connected to ground. This will forward-bias  $D_3$  so that current will flow from the +5 V source terminal, through  $R_1$  and  $D_3$ , and through terminal B to ground. The forward voltage across  $D_3$  will hold point Y at approximately 0.7 V. This voltage is not enough to forward-bias  $D_4$  and the E–B junction of  $Q_2$  sufficiently for conduction.

With  $Q_2$  off, there is no base current for  $Q_4$ , and it turns off. Because there is no  $Q_2$  collector current, the voltage at  $Q_3$ 's base will be large enough



Input Conditions	Output Conditions
A and B are both HIGH ( $\geq 2\text{ V}$ )	$Q_3$ OFF
Input currents are very low $I_{IH} = 10\ \mu\text{A}$	$Q_4$ ON so that $V_X$ is LOW ( $\leq 0.4\text{ V}$ )

(a) LOW output



Input Conditions	Output Conditions
A or B or both are LOW ( $\leq 0.8\text{ V}$ )	$Q_4$ OFF
Current flows back to ground through LOW input terminal. $I_{IL} = 1.1\text{ mA}$	$Q_3$ acts as emitter-follower and $V_{OH} \geq 2.4\text{ V}$ , typically 3.6 V

(b) HIGH output

FIGURE 8-8 TTL NAND gate in its two output states.

to forward-bias  $Q_3$  and  $D_1$ , so that  $Q_3$  will conduct. Actually,  $Q_3$  acts as an emitter follower because output terminal X is essentially at its emitter. With no load connected from point X to ground,  $V_{OH}$  will be around 3.4 to 3.8 V because two 0.7-V diode drops (E-B of  $Q_3$ , and  $D_1$ ) subtract from the 5 V applied to  $Q_3$ 's base. This voltage will decrease under load because the load will draw emitter current from  $Q_3$ , which draws base current through  $R_2$ , thereby increasing the voltage drop across  $R_2$ .

It's important to note that there is a substantial current flowing back through input terminal B to ground when B is held LOW. This current,  $I_{IL}$ , is determined by the value of resistor  $R_1$ , which will vary from series to series. For standard TTL, it is about 1.1 mA. The LOW B input acts as a sink to ground for this current.



## Current-Sinking Action

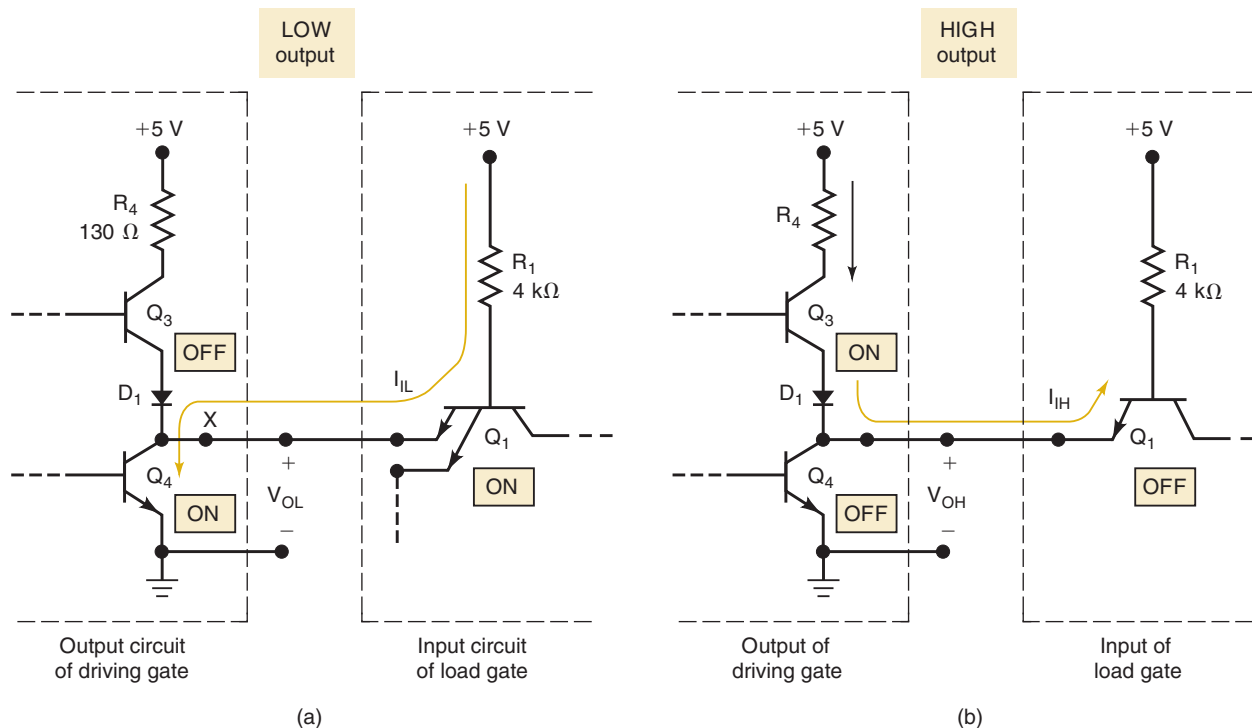
A TTL output acts as a current sink in the LOW state because it *receives* current from the input of the gate that it is driving. Figure 8-9 shows one TTL gate driving the input of another gate (the load) for both output voltage states. In the output LOW state situation depicted in Figure 8-9(a), transistor  $Q_4$  of the driving gate is on and essentially “shorts” point X to ground. This LOW voltage at X forward-biases the emitter–base junction of  $Q_1$ , and current flows, as shown, back through  $Q_4$ . Thus,  $Q_4$  is performing a current-sinking action that derives its current from the input current ( $I_{IL}$ ) of the load gate. We will often refer to  $Q_4$  as the **current-sinking transistor** or as the **pull-down transistor** because it brings the output voltage down to its LOW state.

## Current-Sourcing Action

A TTL output acts as a current source in the HIGH state. This is shown in Figure 8-9(b), where transistor  $Q_3$  is supplying the input current,  $I_{IH}$ , required by the  $Q_1$  transistor of the load gate. As stated above, this current is a small reverse-bias leakage current (typically  $10\ \mu\text{A}$ ). We will often refer to  $Q_3$  as the **current-sourcing transistor** or **pull-up transistor**. In some of the more modern TTL series, the pull-up circuit is made up of two transistors, rather than a transistor and diode.

## Totem-Pole Output Circuit

Several points should be mentioned concerning the totem-pole arrangement of the TTL output circuit, as shown in Figure 8-9, because it is not readily apparent why it is used. The same logic can be accomplished by eliminating



**FIGURE 8-9** (a) When the TTL output is in the LOW state,  $Q_4$  acts as a current sink, deriving its current from the load. (b) In the output HIGH state,  $Q_3$  acts as a current source, providing current to the load gate.

$Q_3$  and  $D_1$  and connecting the bottom of  $R_4$  to the collector of  $Q_4$ . But this arrangement would mean that  $Q_4$  would conduct a fairly heavy current in its saturation state ( $5\text{V}/130\ \Omega \approx 40\text{mA}$ ). With  $Q_3$  in the circuit, there will be no current through  $R_4$  in the output LOW state. This is important because it keeps the circuit power dissipation down.

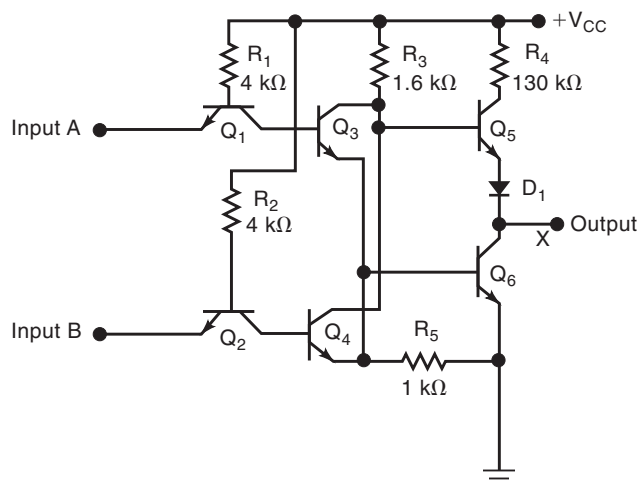
Another advantage of this arrangement occurs in the output HIGH state. Here  $Q_3$  is acting as an emitter follower with its associated low output impedance (typically  $10\ \Omega$ ). This low output impedance provides a short time constant for charging up any capacitive load on the output. This action (commonly called *active pull-up*) provides very fast rise-time waveforms at TTL outputs.

A disadvantage of the totem-pole output arrangement occurs during the transition from LOW to HIGH. Unfortunately,  $Q_4$  turns off more slowly than  $Q_3$  turns on, and so there is a period of a few nanoseconds during which both transistors are conducting and a relatively large current (30 to 40 mA) will be drawn from the 5-V supply. This can present a problem that will be examined later.

## TTL NOR Gate

Figure 8-10 shows the internal circuit for a TTL NOR gate. We will not go through a detailed analysis of this circuit, but it is important to note how it compares to the NAND circuit of Figure 8-8. On the input side, we can see that the NOR circuit *does not use a multiple-emitter* transistor; instead, each input is applied to the emitter of a separate transistor. On the output side, the NOR circuit uses the same totem-pole arrangement as the NAND circuit.

**FIGURE 8-10** TTL NOR gate circuit.



## Summary

All TTL circuits have a similar structure. NAND and AND gates use multiple-emitter transistor or multiple diode junction inputs; NOR and OR gates use separate input transistors. In either case, the input will be the cathode (N-region) of a P-N junction, so that a HIGH input voltage will turn off the junction and only a small leakage current ( $I_{IH}$ ) will flow. Conversely, a LOW input voltage turns on the junction, and a relatively large current ( $I_{IL}$ ) will flow back through the signal source. Most, but not all, TTL circuits will have some type of totem-pole output configuration. There are some exceptions that will be discussed later.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. *True or false:* A TTL output acts as a current sink in the LOW state.
2. In which TTL input state does the largest amount of input current flow?
3. State the advantages and disadvantages of a totem-pole output.
4. Which TTL transistor is the pull-up transistor in the NAND circuit?
5. Which TTL transistor is the pull-down transistor in the NOR circuit?
6. How does the TTL NOR circuit differ from the NAND circuit?

### 8-3 TTL DATA SHEETS

#### OUTCOME

*Upon completion of this chapter, you will be able to:*

- Use a data sheet to find operating characteristics of any logic device.

In 1964, Texas Instruments Corporation introduced the first line of standard TTL ICs. The 54/74 series, as it is called, has been one of the most widely used IC logic families. We will simply refer to it as the 74 series because the major difference between the 54 and 74 versions is that devices in the 54 series can operate over a wider range of temperatures and power-supply voltages. Many semiconductor manufacturers have produced TTL ICs. Fortunately, they all have used the same numbering system, so that the basic IC number is the same from one manufacturer to another. Each manufacturer, however, usually attaches its own special prefix to the IC number. For example, Texas Instruments uses the prefix SN, National Semiconductor uses DM, and Signetics uses S. Thus, depending on the manufacturer, you may see a quad NOR gate chip labeled as a DM7402, SN7402, S7402, or some other similar designation. The important part is the number 7402, which is the same for all manufacturers.

As we learned in Chapter 4, there are several series in the TTL family of logic devices (74, 74LS, 74S, etc.). The original standard series and its immediate descendants (74, 74LS, 74S) are no longer recommended by the manufacturers for use in new designs. In spite of this, enough demand in the market keeps them in production. An understanding of the characteristics that define the capabilities and limitations of any logic device is vital. This section will define those characteristics using the advanced low-power Schottky (ALS) series and help you understand a typical data sheet. Later we will discuss the other TTL series and compare their characteristics.

We can find all of the information we need on any IC by consulting the manufacturer's published data sheets for that particular IC family. These data sheets can be obtained from data books, CD ROMs, or the IC manufacturer's Internet web site. Figure 8-11 is the manufacturer's data sheet for the 74ALS00 NAND gate IC showing the recommended operating conditions, electrical characteristics, and switching characteristics. Most of the quantities discussed in the following paragraphs in this section can be found on this data sheet. As we discuss each quantity, you should refer to this data sheet to see where the information came from.

Recommended Operating Conditions		54ALS00			74ALS00			Unit
		Min	Nominal	Max	Min	Nominal	Max	
$V_{CC}$	Supply voltage	4.5	5	5.5	4.5	5	5.5	V
$V_{IH}$	High-level input voltage	2			2			V
$V_{IL}$	Low-level input voltage				0.8			V
$I_{OL}$	Low-level output current				4			8
$I_{OH}$	High-level output current				-0.4			mA
$T_A$	Operating free-air temperature	-55			125			0
					70			°C

Electrical Characteristics		54ALS00			74ALS00			Unit
Parameter	Test Conditions	Min	Typical	Max	Min	Typical	Max	
$V_{IK}$	$V_{CC} = 4.5\text{ V}$ $I_I = -18\text{ mA}$	-1.2			-1.5			V
$V_{OH}$	$V_{CC} = 4.5\text{ V}$ $I_{OH} = -0.4\text{ mA}$	$V_{CC} - 2$			$V_{CC} - 2$			V
$V_{OL}$	$V_{CC} = 4.5\text{ V}$	$I_{OL} = 4\text{ mA}$		0.25	0.4	0.25		0.4
		$I_{OL} = 8\text{ mA}$				0.35		0.5
$I_I$	$V_{CC} = 5.5\text{ V}$ $V_I = 7\text{ V}$	0.1			0.1			mA
$I_{IH}$	$V_{CC} = 5.5\text{ V}$ $V_I = 2.7\text{ V}$	20			20			μA
$I_{IL}$	$V_{CC} = 5.5\text{ V}$ $V_I = 0.4\text{ V}$	-0.1			-0.1			mA
$I_O$	$V_{CC} = 5.5\text{ V}$ $V_O = 2.25\text{ V}$	-20	-112		-30	-112		mA
$I_{CCH}$	$V_{CC} = 5.5\text{ V}$ $V_I = 0\text{ V}$	0.5		0.85	0.5		0.85	mA
$I_{CCL}$	$V_{CC} = 5.5\text{ V}$ $V_I = 4.5\text{ V}$	1.5		3	1.5		3	mA

Switching Characteristics		$V_{CC} = 4.5\text{ V to } 5.5\text{ V}$ $C_L = 50\text{ pF}$ $R_L = 500\Omega$ $T_A = \text{MIN to MAX}$				Unit	
Parameter	From (input)	To (output)	54ALS00		74ALS00		
			Min	Max	Min	Max	
$t_{PLH}$	A or B	Y	3	15	3	11	ns
$t_{PHL}$			2	9	2	8	ns

FIGURE 8-11 Typical data sheet information for a 74ALS00 NAND gate IC.

## Supply Voltage and Temperature Range

Both the 74ALS series and the 54ALS series use a nominal supply voltage ( $V_{CC}$ ) of 5 V, but can tolerate a supply variation of 4.5 to 5.5 V. The 74ALS series is designed to operate properly in ambient temperatures ranging from 0 to 70°C, while the 54ALS series can handle -55 to +125°C. Because of its greater tolerance of voltage and temperature variations, the 54ALS series is more expensive. It is employed only in applications where reliable operation must be maintained over an extreme range of conditions. Examples are military and space applications.

## Voltage Levels

The input and output logic voltage levels for the 74ALS series can be found on the data sheet of Figure 8-11. Table 8-3 presents them in summary form. The minimum and maximum values shown are for worst-case conditions of power supply, temperature, and loading conditions. Inspection of the

**TABLE 8-3** 74ALS series voltage levels.

	Minimum	Typical	Maximum
$V_{OL}$ (V)	—	0.35	0.5
$V_{OH}$ (V)	2.5	3.4	—
$V_{IL}$ (V)	—	—	0.8
$V_{IH}$ (V)	2.0	—	—

table reveals a guaranteed maximum logical 0 output  $V_{OL} = 0.5\text{ V}$ , which is 300 mV less than the logical 0 voltage needed at the input  $V_{IL} = 0.8\text{ V}$ . This means that the guaranteed LOW-state DC noise margin is 300 mV. That is,

$$V_{NL} = V_{IL}(\text{max}) - V_{OL}(\text{max}) = 0.8\text{ V} - 0.5\text{ V} = 0.3\text{ V} = 300\text{ mV}$$

Similarly, the logical 1 output  $V_{OH}$  is a guaranteed minimum of 2.5 V, which is 500 mV greater than the logical 1 voltage needed at the input,  $V_{IH} = 2.0\text{ V}$ . Thus, the HIGH-state DC noise margin is 500 mV.

$$V_{NH} = V_{OH}(\text{min}) - V_{IH}(\text{min}) = 2.5\text{ V} - 2.0\text{ V} = 0.5\text{ V} = 500\text{ mV}$$

Thus, the *guaranteed worst-case* DC noise margin for the 74ALS series is 300 mV.

## Maximum Voltage Ratings

The voltage values in Table 8-3 *do not include* the absolute maximum ratings beyond which the useful life of the IC may be impaired. The absolute maximum operating conditions are generally given at the top of a data sheet (not shown in Figure 8-11). The voltages applied to any input of this series IC must never exceed +7.0 V. A voltage greater than +7.0 V applied to an input emitter can cause reverse breakdown of the E–B junction of  $Q_1$ .

There is also a limit on the maximum *negative* voltage that can be applied to a TTL input. This limit,  $-0.5\text{ V}$ , is caused by the fact that most TTL circuits employ protective shunt diodes on each input. These diodes were purposely left out of our earlier analysis because they do not enter into the normal circuit operation. They are connected from each input to ground to limit the negative input voltage excursions that often occur when logic signals have excessive ringing. With these diodes, we should not apply more than  $-0.5\text{ V}$  to an input because the protective diodes would begin to conduct and draw substantial current, probably causing the diode to short out, resulting in a permanently faulty input.

## Power Dissipation

An ALS TTL NAND gate draws an average power of 2.4 mW. This is a result of  $I_{CCH} = 0.85\text{ mA}$  and  $I_{CCL} = 3\text{ mA}$ , which produces  $I_{CC}(\text{avg}) = 1.93\text{ mA}$  and  $P_D(\text{avg}) = 1.93\text{ mA} \times 5\text{ V} = 9.65\text{ mW}$ . This 9.65 mW is the total power required by all four gates on the chip. Thus, one NAND gate requires an average power of 2.4 mW.

## Propagation Delays

The data sheet gives minimum and maximum propagation delays. Assuming the typical value is midway, it gives a  $t_{PLH} = 7\text{ ns}$  and  $t_{PHL} = 5\text{ ns}$ . The typical *average* propagation delay  $t_{pd}(\text{avg}) = 6\text{ ns}$ .

**EXAMPLE 8-2**

Refer to the data sheet for the 74ALS00 quad two-input NAND IC in Figure 8-11. Determine the *maximum* average power dissipation and the *maximum* average propagation delay of a *single* gate.

**Solution**

Look under the electrical characteristics for the *maximum*  $I_{CCH}$  and  $I_{CCL}$  values. The values are 0.85 mA and 3 mA, respectively. The average  $I_{CC}$  is therefore 1.9 mA. The average power is obtained by multiplying by  $V_{CC}$ . The data sheet indicates that these  $I_{CC}$  values were obtained when  $V_{CC}$  was at its maximum value (5.5 V for the 74ALS series). Thus, we have

$$P_D(\text{avg}) = 1.9 \text{ mA} \times 5.5 \text{ V} = 10.45 \text{ mW}$$

as the power drawn by the *complete* IC. We can determine the power drain of one NAND gate by dividing this by 4:

$$P_D(\text{avg}) = 2.6 \text{ mW per gate}$$

Because this average power drain was calculated using the maximum current and voltage values, it is the maximum average power that a 74ALS00 NAND gate will draw under worst-case conditions. Designers often use worst-case values to ensure that their circuits will work under all conditions.

The maximum propagation delays for a 74ALS00 NAND gate are listed as

$$t_{PLH} = 11 \text{ ns} \quad t_{PHL} = 8 \text{ ns}$$

so that the maximum average propagation delay is

$$t_{pd}(\text{avg}) = \frac{11 + 8}{2} = 9.5 \text{ ns}$$

Again, this is a worst-case maximum possible average propagation delay.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is the maximum current that 74ALS00 can deliver to a load while maintaining a valid logic 1 on its output?
2. What is the highest valid voltage that a 74ALS00 will interpret as a logic 0 on its inputs?
3. How long (max) will it take for the output of a 74ALS00 to go HIGH after its input goes LOW?
4. What is the maximum current drawn from the power supply of a 74ALS00 IC if all of its outputs are LOW?

**8-4 TTL SERIES CHARACTERISTICS****OUTCOMES**

*Upon completion of this section, you will be able to:*

- Evaluate performance trade-offs between various technology series of the TTL family.
- Identify the best series to optimize a particular characteristic.

The standard 74 series of TTL has evolved into several other series. All of them offer a wide variety of gates and flip-flops in the small-scale integration (SSI) line, and counters, registers, multiplexers, decoders/encoders, and other logic functions in their medium-scale integration (MSI) line. The following TTL series—often called “subfamilies”—provide a wide range of speed and power capabilities.

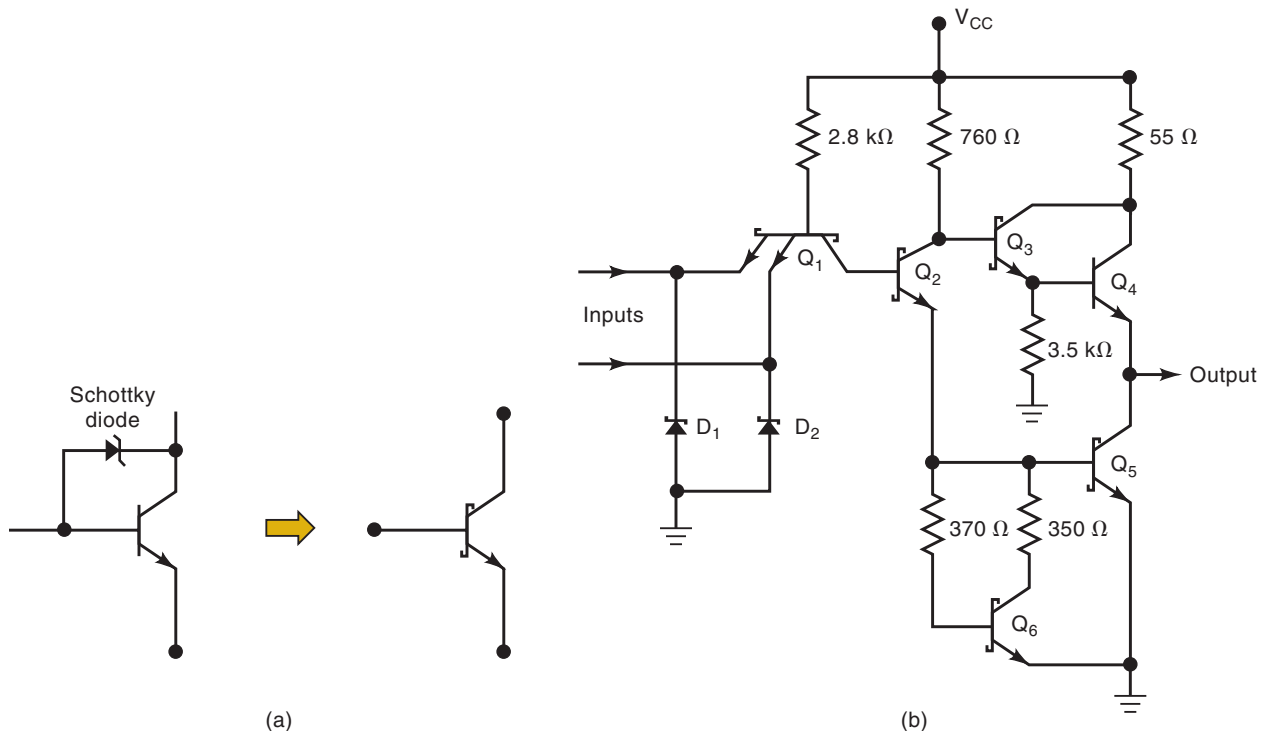
### Standard TTL, 74 Series

The original standard 74 series of TTL logic was described in Section 8-2. These devices are still available, but in most cases they are no longer a reasonable choice for new designs because other devices are now available that perform much better at a lower cost.

### Schottky TTL, 74S Series

The 7400 series operates using saturated switching in which many of the transistors, when conducting, will be in the saturated condition. This operation causes a storage-time delay,  $t_s$ , when the transistors switch from ON to OFF, and it limits the circuit's switching speed.

The 74S series reduces this storage-time delay by not allowing the transistor to go as deeply into saturation. It accomplishes this by using a Schottky barrier diode (SBD) connected between the base and the collector of each transistor, as shown in Figure 8-12(a). The SBD has a forward voltage of only 0.25 V. Thus, when the C–B junction becomes forward-biased at the onset of saturation, the SBD will conduct and divert some of the input current away from the base. This reduces the excess base current and decreases the storage-time delay at turn-off.



**FIGURE 8-12** (a) Schottky-clamped transistor; (b) basic NAND gate in S-TTL series.

As shown in Figure 8-12(a), the transistor/SBD combination is given a special symbol. This symbol is used for all of the transistors in the circuit diagram for the 74S00 NAND gate shown in Figure 8-12(b). This 74S00 NAND gate has an average propagation delay of only 3 ns, which is six times as fast as the 7400. Note the presence of shunt diodes  $D_1$  and  $D_2$  to limit negative input voltages.

Circuits in the 74S series also use smaller resistor values to help improve switching times. This increases the circuit average power dissipation to about 20 mW, about two times greater than the 74 series. The 74S circuits also use a Darlington pair ( $Q_3$  and  $Q_4$ ) to provide a shorter output rise time when switching from ON to OFF.

### Low-Power Schottky TTL, 74LS Series (LS-TTL)

The 74LS series is a lower-powered, slower-speed version of the 74S series. It uses the Schottky-clamped transistor, but with larger resistor values than the 74S series. The larger resistor values reduce the circuit power requirement, but at the expense of an increase in switching times. A NAND gate in the 74LS series will typically have an average propagation delay of 9.5 ns and an average power dissipation of 2 mW.

### Advanced Schottky TTL, 74AS Series (AS-TTL)

Innovations in integrated-circuit design led to the development of two improved TTL series: advanced Schottky (74AS) and advanced low-power Schottky (74ALS). The 74AS series provides a considerable improvement in speed over the 74S series at a much lower power requirement. The comparison is shown in Table 8-4 for a NAND gate in each series. This comparison clearly shows the advantage of the 74AS series. It is the fastest TTL series, and its power dissipation is significantly lower than that of the 74S series. The 74AS has other improvements, including lower input current requirements ( $I_{IL}$ ,  $I_{IH}$ ), that result in a greater fan-out than in the 74S series.

### Advanced Low-Power Schottky TTL, 74ALS Series

This series offers an improvement over the 74LS series in both speed and power dissipation, as the numbers in Table 8-5 illustrate. The 74ALS series has the lowest gate power dissipation of all the TTL series.

TABLE 8-4 Speed and Power for S and AS series.

	74S	74AS
Propagation delay	3 ns	1.7 ns
Power dissipation	20 mW	8 mW

TABLE 8-5 Speed and Power for LS and ALS series.

	74LS	74ALS
Propagation delay	9.5 ns	4 ns
Power dissipation	2 mW	1.2 mW

### 74F—Fast TTL

This series uses a new integrated-circuit fabrication technique to reduce interdevice capacitances and thus achieve reduced propagation delays. A typical NAND gate has an average propagation delay of 3 ns and a power



consumption of 6 mW. ICs in this series are designated with the letter F in their part number. For instance, the 74F04 is a hex-inverter chip.

### Comparison of TTL Series Characteristics

Table 8-6 gives the typical values for some of the more important characteristics of each of the TTL series. All of the performance ratings, except for the maximum clock rate, are for a NAND gate in each series. The maximum clock rate is specified as the maximum frequency that can be used to toggle a J-K flip-flop. This gives a useful measure of the frequency range over which each IC series can be operated.

**TABLE 8-6** Typical TTL series characteristics.

	74	74S	74LS	74AS	74ALS	74F
<b>Performance ratings</b>						
Propagation delay (ns)	9	3	9.5	1.7	4	3
Power dissipation (mW)	10	20	2	8	1.2	6
Max. clock rate (MHz)	35	125	45	200	70	100
Fan-out (same series)	10	20	20	40	20	33
<b>Voltage parameters</b>						
$V_{OH}(\text{min})$ (V)	2.4	2.7	2.7	2.5	2.5	2.5
$V_{OL}(\text{max})$ (V)	0.4	0.5	0.5	0.5	0.5	0.5
$V_{IH}(\text{min})$ (V)	2.0	2.0	2.0	2.0	2.0	2.0
$V_{IL}(\text{max})$ (V)	0.8	0.8	0.8	0.8	0.8	0.8

#### EXAMPLE 8-3

Use Table 8-6 to calculate the DC noise margins for a typical 74LS IC. How does this compare with the standard TTL noise margins?

#### Solution

**74LS**

$$\begin{aligned} V_{NH} &= V_{OH}(\text{min}) - V_{IH}(\text{min}) \\ &= 2.7\text{ V} - 2.0\text{ V} \\ &= 0.7\text{ V} \end{aligned}$$

$$\begin{aligned} V_{NL} &= V_{IL}(\text{max}) - V_{OL}(\text{max}) \\ &= 0.8\text{ V} - 0.5\text{ V} \\ &= 0.3\text{ V} \end{aligned}$$

**74**

$$\begin{aligned} V_{NH} &= 2.4\text{ V} - 2.0\text{ V} \\ &= 0.4\text{ V} \end{aligned}$$

$$\begin{aligned} V_{NL} &= 0.8\text{ V} - 0.4\text{ V} \\ &= 0.4\text{ V} \end{aligned}$$

#### EXAMPLE 8-4

Which TTL series can drive the most device inputs of the same series?

#### Solution

The 74AS series has the highest fan-out (40), which means that a 74AS00 NAND gate can drive 40 inputs of other 74AS devices. If we want to determine the number of inputs of a *different* TTL series that an output can drive, we will need to know the input and output currents of the two series. This will be dealt with in the next section.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

- Which TTL series is the best at high frequencies?
  - Which TTL series has the largest HIGH-state noise margin?
  - Which series has essentially become obsolete in new designs?
  - Which series uses a special diode to reduce switching time?
  - Which series would be best for a battery-powered circuit operating at 10 MHz?
- Assuming the same cost for each, why should you choose to use a 74ALS193 counter over a 74LS193 or a 74AS193 in a circuit operating from a 40-MHz clock?
- Identify the pull-up and pull-down transistors for the 74S circuit in Figure 8-12.

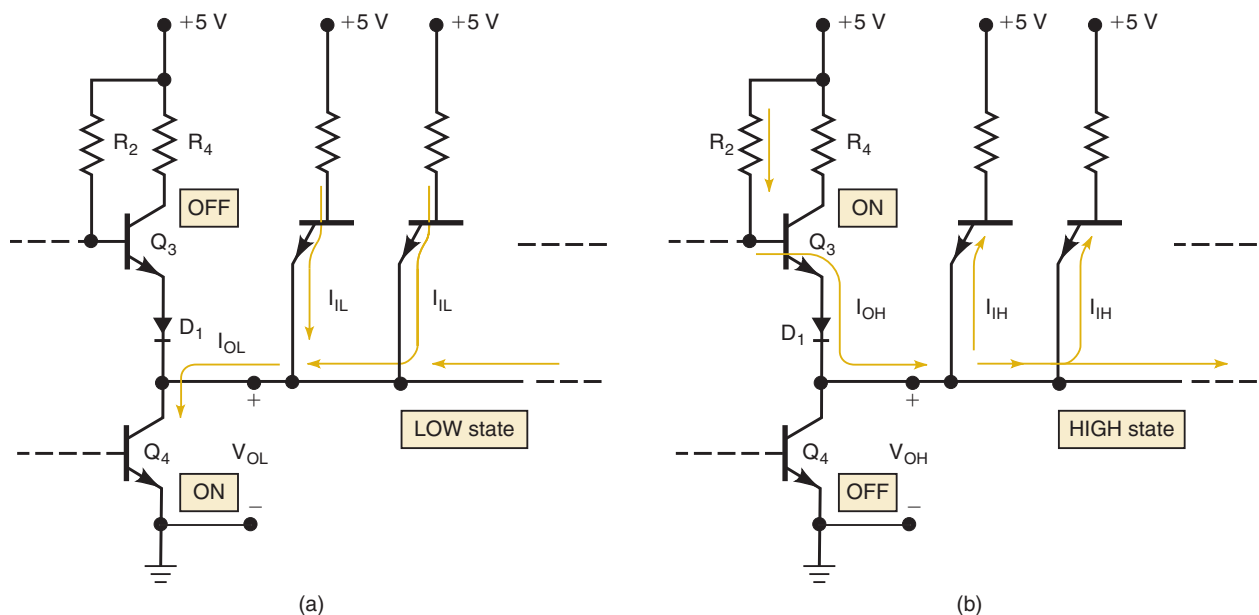
## 8-5 TTL LOADING AND FAN-OUT

### OUTCOMES

Upon completion of this section, you will be able to:

- Apply knowledge of internal circuitry to determine the limits of loading.
- Determine the number of inputs (of any series) that can be driven by an output.

It is important to understand what determines the fan-out or load drive capability of an IC output. Figure 8-13(a) shows a standard TTL output in the LOW state connected to drive several standard TTL inputs. Transistor  $Q_4$  is on and is acting as a current sink for an amount of current  $I_{OL}$  that is the sum of the  $I_{IL}$  currents from each input. In its ON state,  $Q_4$ 's collector-emitter resistance is very small, but it is not zero, and so the current  $I_{OL}$



**FIGURE 8-13** Currents when a TTL output is driving several inputs.

will produce a voltage drop  $V_{OL}$ . This voltage must not exceed the  $V_{OL}(\max)$  limit of the IC, which limits the maximum value of  $I_{OL}$  and thus the number of loads that can be driven.

To illustrate, suppose that the ICs are in the 74 series and each  $I_{IL}$  is 1.6 mA. From Table 8-6, we see that the 74 series has  $V_{OL}(\max) = 0.4$  V and  $V_{IL}(\max) = 0.8$  V. Let's suppose further that  $Q_4$  can sink up to 16 mA before its output voltage reaches  $V_{OL}(\max) = 0.4$  V. This means that it can sink the current from up to  $16 \text{ mA}/1.6 \text{ mA} = 10$  loads. If it is connected to more than 10 loads, its  $I_{OL}$  will increase and cause  $V_{OL}$  to increase above 0.4 V. This is usually undesirable because it reduces the noise margin at the IC inputs [remember,  $V_{NL} = V_{IL}(\max) - V_{OL}(\max)$ ]. In fact, if  $V_{OL}$  rises above  $V_{IL}(\max) = 0.8$  V, it will be in the indeterminate range.

A similar situation occurs in the HIGH state depicted in Figure 8-13(b). Here,  $Q_3$  is acting as an emitter follower that is sourcing (supplying) a total current  $I_{OH}$  that is the sum of the  $I_{IH}$  currents of the different TTL inputs. If too many loads are being driven, this current  $I_{OH}$  will become large enough to cause the voltage drops across  $R_2$ ,  $Q_3$ 's emitter-base junction, and  $D_1$  to bring  $V_{OH}$  below  $V_{OH}(\min)$ . This too is undesirable because it reduces the HIGH-state noise margin and could even cause  $V_{OH}$  to go into the indeterminate range.

What this all means is that a TTL output has a limit,  $I_{OL}(\max)$ , on how much current it can sink in the LOW state. It also has a limit,  $I_{OH}(\max)$ , on how much current it can source in the HIGH state. These output current limits must not be exceeded if the output voltage levels are to be maintained within their specified ranges.

### Determining the Fan-Out

To determine how many different inputs an IC output can drive, you need to know the current drive capability of the output [i.e.,  $I_{OL}(\max)$  and  $I_{OH}(\max)$ ] and the current requirements of each input (i.e.,  $I_{IL}$  and  $I_{IH}$ ). This information is always presented in some form on the manufacturer's IC data sheet or Web page. The following examples will illustrate one type of situation.

#### EXAMPLE 8-5

How many 74ALS00 NAND gate inputs can be driven by a 74ALS00 NAND gate output?

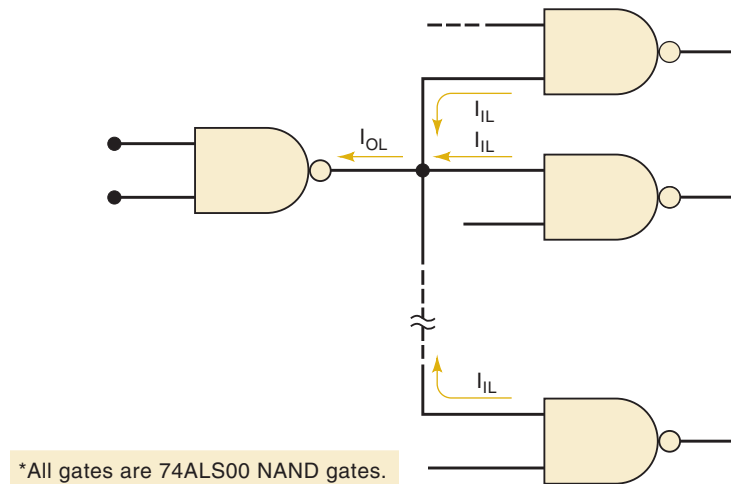
#### Solution

We will consider the LOW state first as depicted in Figure 8-14. Refer to the 74ALS00 data sheet in Figure 8-11 and find

$$\begin{aligned} I_{OL}(\max) &= 8 \text{ mA} \\ I_{IL}(\max) &= 0.1 \text{ mA} \end{aligned}$$

This says that a 74ALS00 output can sink a maximum of 8 mA and that each 74ALS00 input will source a maximum of 0.1 mA back through the driving gate's output. Thus, the number of inputs that can be driven in the LOW state is obtained as

$$\begin{aligned} \text{fan-out (LOW)} &= \frac{I_{OL}(\max)}{I_{IL}(\max)} \\ &= \frac{8 \text{ mA}}{0.1 \text{ mA}} \\ &= 80 \end{aligned}$$

**FIGURE 8-14** Example 8-5.

(Note: The entry for  $I_{IL}$  is actually  $-0.1$  mA. The negative sign is used to indicate that this current flows *out of* the input terminal; we can ignore this sign for our purposes here.) The HIGH state is analyzed in the same manner. Refer to the data sheet to find values for  $I_{OH}$  and  $I_{IH}$ , ignoring any negative signs.

$$I_{OH}(\text{max}) = 0.4 \text{ mA} = 400 \mu\text{A}$$

$$I_{IH}(\text{max}) = 20 \mu\text{A}$$

Thus, the number of inputs that can be driven in the HIGH state is

$$\begin{aligned} \text{fan-out (HIGH)} &= \frac{I_{OH}(\text{max})}{I_{IH}(\text{max})} \\ &= \frac{400 \mu\text{A}}{20 \mu\text{A}} \\ &= 20 \end{aligned}$$

If fan-out (LOW) and fan-out (HIGH) are not the same, as will sometimes occur, the fan-out is chosen as the smaller of the two. Thus, the 74ALS00 NAND gate can drive up to 20 other 74ALS00 NAND gates.

**EXAMPLE 8-6**

Refer to the data in Table 8-7 and determine how many 74AS20 NAND gates can be driven by the output of another 74AS20.

**Solution**

Table 8-7 gives the following values for the 74AS series:

$$I_{OH}(\text{max}) = 2 \text{ mA}$$

$$I_{OL}(\text{max}) = 20 \text{ mA}$$

$$I_{IH}(\text{max}) = 20 \mu\text{A}$$

$$I_{IL}(\text{max}) = 0.5 \text{ mA}$$

Considering the HIGH state first, we have

$$\text{fan-out (HIGH)} = \frac{2 \text{ mA}}{20 \mu\text{A}} = 100$$

For the LOW state, we have

$$\text{fan-out (LOW)} = \frac{20 \text{ mA}}{0.5 \text{ mA}} = 40$$

In this case, the overall fan-out is chosen to be 40 because it is the lower of the two values. Thus, one 74AS20 can drive 40 other 74AS20 inputs.

In older equipment, most of the logic ICs were often chosen from the same logic family. Later digital systems were much more likely to be a combination of various logic families. Consequently, loading and fan-out calculations are not as straightforward as they once were. A good method for determining the loading of any digital output is as follows:

- Step 1.** Add the  $I_{IH}$  for all inputs connected to an output. This sum must be less than the output's  $I_{OH}$  specification.
- Step 2.** Add the  $I_{IL}$  for all inputs connected to an output. This sum must be less than the output's  $I_{OL}$  specification.

Table 8-7 shows the limiting specifications for input and output currents in simple logic gates of the various TTL families. Notice that some of the current values are given as negative numbers. This convention is used to show the direction of current flow. Positive values indicate current flowing into the specified node, whether it is an input or an output. Negative values indicate current flowing out of the specified node. Consequently, all  $I_{OH}$  values are negative as current flows out of the output (sourcing current), and all  $I_{OL}$  values are positive as load current flows into the output pin on its way to ground (sinking current). Likewise,  $I_{IH}$  is positive, while  $I_{IL}$  is negative. When calculating loading and fan-out as described above, you should ignore these signs.

**TABLE 8-7** Current ratings of TTL series logic gates.\*

TTL Series	Outputs		Inputs	
	$I_{OH}$	$I_{OL}$	$I_{IH}$	$I_{IL}$
74	-0.4 mA	16 mA	40 $\mu\text{A}$	-1.6 mA
74S	-1 mA	20 mA	50 $\mu\text{A}$	-2 mA
74LS	-0.4 mA	8 mA	20 $\mu\text{A}$	-0.4 mA
74AS	-2 mA	20 mA	20 $\mu\text{A}$	-0.5 mA
74ALS	-0.4 mA	8 mA	20 $\mu\text{A}$	-0.1 mA
74F	-1 mA	20 mA	20 $\mu\text{A}$	-0.6 mA

\*Some devices may have different input or output current ratings. Always consult the data sheet.

**EXAMPLE 8-7**

A 74ALS00 NAND gate output is driving three 74S gate inputs and one 7406 input. Using data from Table 8-7, determine if there is a loading problem.

**Solution**

1. Add all of the  $I_{IH}$  values:

$$\begin{aligned} & 3 \cdot (I_{IH} \text{ for } 74S) + 1 \cdot (I_{IH} \text{ for } 74) \\ \text{Total} &= 3 \cdot (50 \mu\text{A}) + 1 \cdot (40 \mu\text{A}) = 190 \mu\text{A} \end{aligned}$$

The  $I_{OH}$  for the 74ALS output is  $400 \mu\text{A}$  (max), which is greater than the sum of the loads ( $190 \mu\text{A}$ ). This poses no problem when the output is HIGH.

2. Add all of the  $I_{IL}$  values:

$$\begin{aligned} & 3 \cdot (I_{IL} \text{ for } 74S) + 1 \cdot (I_{IL} \text{ for } 74) \\ \text{Total} &= 3 \cdot (2 \text{ mA}) + 1 \cdot (1.6 \text{ mA}) = 7.6 \text{ mA} \end{aligned}$$

The  $I_{OH}$  for the 74ALS output is  $8 \text{ mA}$  (max), which is greater than the sum of the loads ( $7.6 \text{ mA}$ ). This poses no problem when the output is LOW.

**EXAMPLE 8-8**

The 74ALS00 NAND gate output in Example 8-7 needs to be used to drive some 74ALS inputs in addition to the load inputs described in Example 8-7. How many additional 74ALS inputs could the output drive without being overloaded?

**Solution**

From the calculations of Example 8-7, only in the LOW state are we close to being overloaded. A 74ALS input has an  $I_{IL}$  of  $0.1 \text{ mA}$ . The maximum sink current ( $I_{OL}$ ) is  $8 \text{ mA}$ , and the load current is  $7.6 \text{ mA}$  (as calculated in Example 8-7). The additional current that the output can sink is found by

$$\begin{aligned} \text{Additional current} &= I_{OL\text{max}} - \text{sum of loads}(I_{IL}) \\ &= 8 \text{ mA} - 7.6 \text{ mA} = 0.4 \text{ mA} \end{aligned}$$

This output can drive up to four more 74ALS inputs that have an  $I_{IL}$  of  $0.1 \text{ mA}$ .

**EXAMPLE 8-9**

The output of a 74AS04 inverter is providing the CLEAR signal to a parallel register made up of 74AS74A D flip-flops. What is the maximum number of FF  $\overline{CLR}$  inputs that this gate can drive?

**Solution**

The input specifications for flip-flop inputs are not always the same as those for a logic gate input in the same family. Refer to the 74AS74A data sheet at [www.ti.com](http://www.ti.com). The clock and  $D$  inputs are similar to the gate inputs in Table 8-7. However, the  $\overline{PRE}$  and  $\overline{CLR}$  inputs have specifications of  $I_{IH} = 40 \mu\text{A}$  and  $I_{IL} = 1.8 \text{ mA}$ . The 74AS04 has specifications of  $I_{OH} = 2 \text{ mA}$  and  $I_{OL} = 20 \text{ mA}$ .

$$\begin{aligned} \text{Maximum number of inputs (HIGH)} &= 2 \text{ mA} / 40 \mu\text{A} = 50 \\ \text{Maximum number of inputs (LOW)} &= 20 \text{ mA} / 1.8 \text{ mA} = 11.11 \end{aligned}$$

We must limit the fan-out to 11  $\overline{CLR}$  inputs.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What factors determine the  $I_{OL}(\max)$  rating of a device?
2. How many 7407 inputs can a 74AS chip drive?
3. What can happen if a TTL output is connected to more gate inputs than it is rated to handle?
4. How many 74S112 *CP* inputs can be driven by a 74LS04 output? By a 74F00 output?

## 8-6 OTHER TTL CHARACTERISTICS

### OUTCOMES

Upon completion of this section, you will be able to:

- Identify common characteristics of TTL technology ICs.
- Use knowledge of internal circuitry to intelligently handle common design issues.
- Apply design techniques required by TTL technology to assure reliability.

Several other characteristics of TTL logic must be understood if one is to use TTL intelligently in a digital-system application.

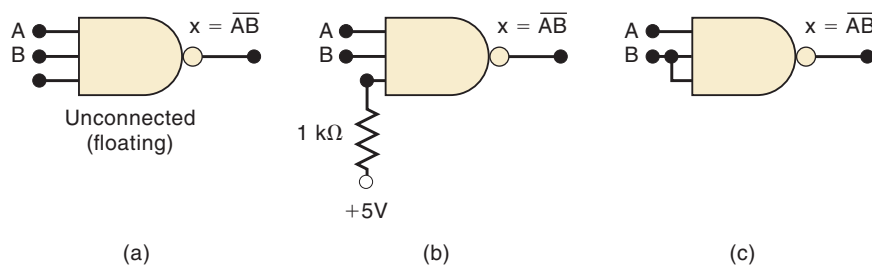
### Unconnected Inputs (Floating)

Any input to a TTL circuit that is left disconnected (open) acts exactly like a logical 1 applied to that input because in either case the emitter-base junction or diode at the input will not be forward-biased. This means that on *any* TTL IC, *all* of the inputs are 1s if they are not connected to some logic signal or to ground. When an input is left unconnected, it is said to be **floating**.

### Unused Inputs

Frequently, not all of the inputs on a TTL IC are being used in a particular application. A common example is when not all the inputs to a logic gate are needed for the required logic function. For example, suppose that we needed the logic operation  $\overline{AB}$  and we were using a chip that had a three-input NAND gate. The possible ways of accomplishing this are shown in Figure 8-15.

**FIGURE 8-15** Three ways to handle unused logic inputs.



In Figure 8-15(a), the unused input is left disconnected, which means that it acts as a logical 1. The NAND gate output is therefore  $x = \bar{A} \cdot \bar{B} \cdot \bar{1} = \bar{A} \cdot \bar{B}$ , which is the desired result. Although the logic is correct, it is highly undesirable to leave an input disconnected because it will act like an antenna, which is liable to pick up stray radiated signals that could cause the gate to operate improperly. A better technique is shown in Figure 8-15(b). Here, the unused input is connected to +5 V through a 1-k $\Omega$  resistor, so that the logic level is a 1. The 1-k $\Omega$  resistor is simply for current protection of the emitter–base junctions of the gate inputs in case of spikes on the power-supply line. This same technique can be used for AND gates because a 1 on an unused input will not affect the output. As many as 30 unused inputs can share the same 1-k $\Omega$  resistor tied to  $V_{CC}$ .

A third possibility is shown in Figure 8-15(c), where the unused input is tied to a used input. This is satisfactory provided that the circuit driving input  $B$  is not going to have its fan-out exceeded. This technique can be used for *any* type of gate. For OR gates and NOR gates, the unused inputs cannot be left disconnected or tied to +5 V because this would produce a constant-output logic level (1 for OR, 0 for NOR) regardless of the other inputs. Instead, for these gates, the unused inputs must either be connected to ground (0 V) for a logic 0 or be tied to a used input, as in Figure 8-15(c).

### Tied-Together Inputs

When two (or more) TTL inputs on the same gate are connected together to form a common input, as in Figure 8-15(c), the common input will generally represent a load that is the sum of the load current rating of each individual input. The only exception is for NAND and AND gates. For these gates, the LOW-state input load *will be the same as a single input* no matter how many inputs are tied together.

To illustrate, assume that each input of the three-input NAND gate in Figure 8-15(c) is rated at 0.5 mA for  $I_{IL}$  and 20  $\mu\text{A}$  for  $I_{IH}$ . The common input  $B$  will therefore represent an input load of 40  $\mu\text{A}$  in the HIGH state but only 0.5 mA in the LOW state. The same would be true if this were an AND gate. If it were an OR or a NOR gate, the common  $B$  input would present an input load 40  $\mu\text{A}$  in the HIGH state and 1 mA in the LOW state.

The reason for this characteristic can be found by looking back at the circuit diagram of the TTL NAND gate in Figure 8-8(b). The current  $I_{IL}$  is limited by the resistance  $R_1$ . Even if inputs  $A$  and  $B$  were tied together and grounded, this current would not change; it would merely divide and flow through the parallel paths provided by diodes  $D_2$  and  $D_3$ . The situation is different for OR and NOR gates because they do not use multiple-emitter transistors but rather have a separate input transistor for each input, as we saw in Figure 8-10.

#### EXAMPLE 8-10

Determine the load that the X output is driving in Figure 8-16. Assume that each gate is a 74LS series device with  $I_{IH} = 20 \mu\text{A}$  and  $I_{IL} = 0.4 \text{ mA}$ .

#### Solution

The loading on the output of gate 1 is equivalent to six 74LS input loads in the HIGH state but only five 74LS input loads in the LOW state because the NAND gate represents only a single input load in the LOW state.



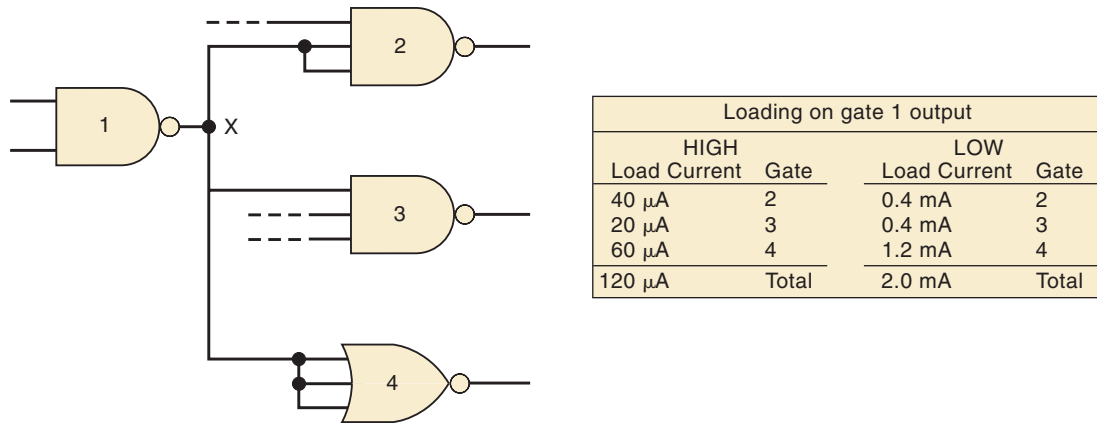


FIGURE 8-16 Example 8-10.

### Biasing TTL Inputs Low

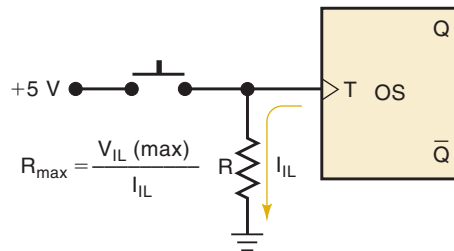
Occasionally, the situation arises where a TTL input must be held normally LOW and then caused to go HIGH by the actuation of a mechanical switch. This situation is illustrated in Figure 8-17 for the input to a one-shot. This OS triggers on a positive transition that occurs when the switch is momentarily closed. The resistor  $R$  serves to keep the  $T$  input LOW while the switch is open. Care must be taken to keep the value of  $R$  low enough so that the voltage developed across it by the current  $I_{IL}$  that flows out of the OS input to ground will not exceed  $V_{IL}(\text{max})$ . Thus, the largest value of  $R$  is given by

$$I_{IL} \times R_{\text{max}} = V_{IL}(\text{max})$$

$$R_{\text{max}} = \frac{V_{IL}(\text{max})}{I_{IL}} \tag{8-3}$$

$R$  must be kept below this value to ensure that the OS input will be at an acceptable LOW level while the switch is open. The minimum value of  $R$  is determined by the current drain on the 5-V supply when the switch is closed. In practice, this current drain should be minimized by keeping  $R$  just slightly below  $R_{\text{max}}$ .

FIGURE 8-17



**EXAMPLE 8-11**

Determine an acceptable value for  $R$  if the OS is a 74LS TTL IC with an  $I_{IL}$  input rating of 0.4 mA.

**Solution**

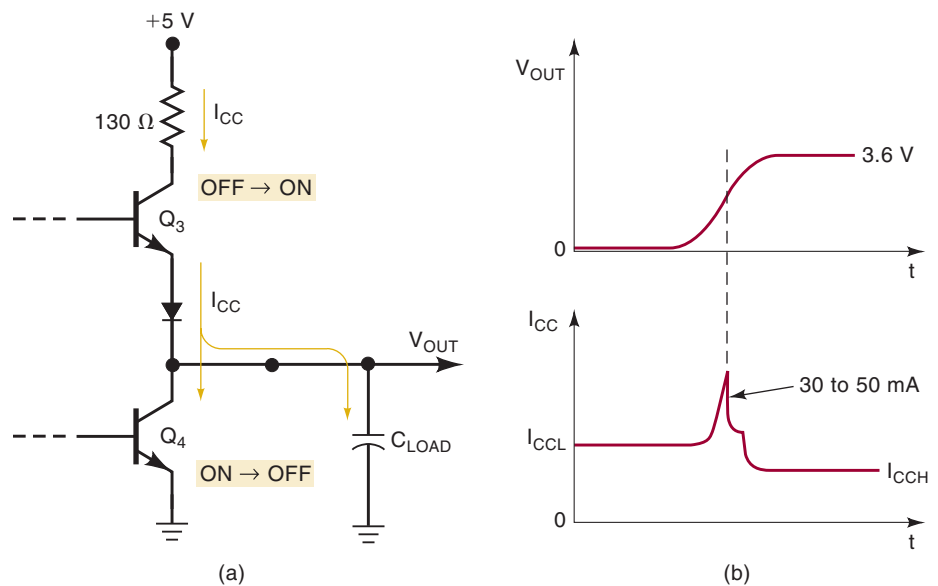
The value of  $I_{IL}$  will be a maximum of 0.4 mA. This maximum value should be used to calculate  $R_{\text{max}}$ . From Table 8-6,  $V_{IL}(\text{max}) = 0.8 \text{ V}$  for the 74LS series. Thus, we have

$$R_{\max} = \frac{0.8 \text{ V}}{0.4 \text{ mA}} = 2000 \ \Omega$$

A good choice here would be  $R = 1.8 \text{ k}\Omega$ , a standard resistor value.

## Current Transients

TTL logic circuits suffer from internally generated current transients or spikes because of the totem-pole output structure. When the output is switching from the LOW state to the HIGH state (see Figure 8-18), the two output transistors are changing states:  $Q_3$  OFF to ON, and  $Q_4$  ON to OFF. Because  $Q_4$  is changing from the saturated condition, it will take longer than  $Q_3$  to switch states. Thus, there is a short interval of time (about 2 ns) during the switching transition when both transistors are conducting and a



**FIGURE 8-18** A large current spike is drawn from  $V_{CC}$  when a totem-pole output switches from LOW to HIGH.

relatively large surge of current (30 to 50 mA) is drawn from the +5V supply. The duration of this current transient is extended by the effects of any load capacitance on the circuit output. This capacitance consists of stray wiring capacitance and the input capacitance of any load circuits and must be charged up to the HIGH-state output voltage. This overall effect can be summarized as follows:

**Whenever a totem-pole TTL output goes from LOW to HIGH, a high-amplitude current spike is drawn from the  $V_{CC}$  supply.**

In a complex digital circuit or system, there may be many TTL outputs switching states at the same time, each one drawing a narrow spike of current from the power supply. The accumulative effect of all of these current spikes will be to produce a voltage spike on the common  $V_{CC}$  line, mostly due to the distributed inductance on the supply line [remember:  $V = L(di/dt)$  for inductance, and  $di/dt$  is very large for a 2-ns current spike]. This voltage spike can cause serious malfunctions during switching transitions unless

some type of filtering is used. The most common technique uses small radio-frequency capacitors connected from  $V_{CC}$  to GROUND essentially to “short out” these high-frequency spikes. This is called **power-supply decoupling**.

It is standard practice to connect a 0.01- $\mu\text{F}$  or 0.1- $\mu\text{F}$  low-inductance, ceramic disk capacitor between  $V_{CC}$  and ground near each TTL IC on a circuit board. The capacitor leads are kept very short to minimize series inductance.

In addition, it is standard practice to connect a single large capacitor (2 to 20  $\mu\text{F}$ ) between  $V_{CC}$  and ground on each board to filter out relatively low-frequency variations in  $V_{CC}$  caused by the large changes in  $I_{CC}$  levels as outputs switch states.

### OUTCOME ASSESSMENT QUESTIONS

1. What will be the logic output of a TTL NAND gate that has all of its inputs unconnected?
2. What are two acceptable ways to handle unused inputs to an AND gate?
3. Repeat question 2 for a NOR gate.
4. *True or false:* When NAND gate inputs are tied together, they are always treated as a single load on the signal source.
5. What is power-supply decoupling? Why is it used?

## 8-7 MOS TECHNOLOGY

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Relate MOS transistor technology and circuit configuration to operating characteristics.
- Relate logic function to circuit operation using MOS transistor circuits.
- Estimate MOS electrical characteristics for both ON and OFF conditions.

MOS (metal-oxide-semiconductor) technology derives its name from the basic MOS structure of a metal electrode over an oxide insulator over a semiconductor substrate. The transistors of MOS technology are field-effect transistors called **MOSFETs**. This means that the electric *field* on the *metal* electrode side of the *oxide* insulator has an *effect* on the resistance of the substrate. Most of the MOS digital ICs are constructed entirely of MOSFETs and no other components.

The chief advantages of the MOSFET are that it is relatively simple and inexpensive to fabricate, it is small, and it consumes very little power. The fabrication of MOS ICs is approximately one-third as complex as the fabrication of bipolar ICs (TTL, ECL, etc.). In addition, MOS devices occupy much less space on a chip than do bipolar transistors. More important, MOS digital ICs normally do not use the IC resistor elements that take up so much of the chip area of bipolar ICs.

All of this means that MOS ICs can accommodate a much larger number of circuit elements on a single chip than bipolar ICs. This advantage is illustrated by the fact that MOS ICs have dominated bipolar ICs in the area of large-scale integration (LSI, VLSI). The high packing density of MOS ICs makes them especially well suited for complex ICs such as microprocessor and memory chips. Improvements in MOS IC technology have led to devices that are faster than 74, 74LS, and 74ALS TTL with comparable current drive

characteristics. Consequently, MOS devices (specifically CMOS) have also become dominant in the SSI and MSI market. The 74AS TTL family is as fast as the best CMOS devices, but at the price of much greater power dissipation.

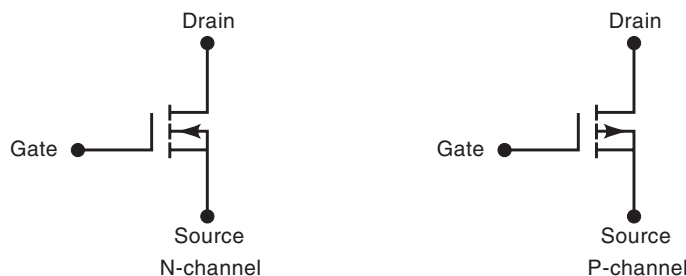
The principal disadvantage of MOS devices is their susceptibility to static-electricity damage. Although this can be minimized by proper handling procedures, TTL is still more durable for laboratory experimentation. Consequently, you are likely to see TTL devices used in education as long as they are available.

## The MOSFET

There are presently two general types of MOSFETs: *depletion* and *enhancement*. MOS digital ICs use enhancement MOSFETs exclusively, and so only this type will be considered in the following discussion. Furthermore, we will concern ourselves only with the operation of these MOSFETs as on/off switches.

Figure 8-19 shows the schematic symbols for the N-channel and P-channel enhancement MOSFETs, where the direction of the arrow indicates either P- or N-channel. The symbols show a broken line between the *source* and the *drain* to indicate that there is *normally* no conducting channel between these electrodes. The symbol also shows a separation between the *gate* and the other terminals to indicate the very high resistance (typically around  $10^{12} \Omega$ ) of the oxide layer between the gate and the channel, which is formed in the substrate.

**FIGURE 8-19** Schematic symbols for enhancement MOSFETs.



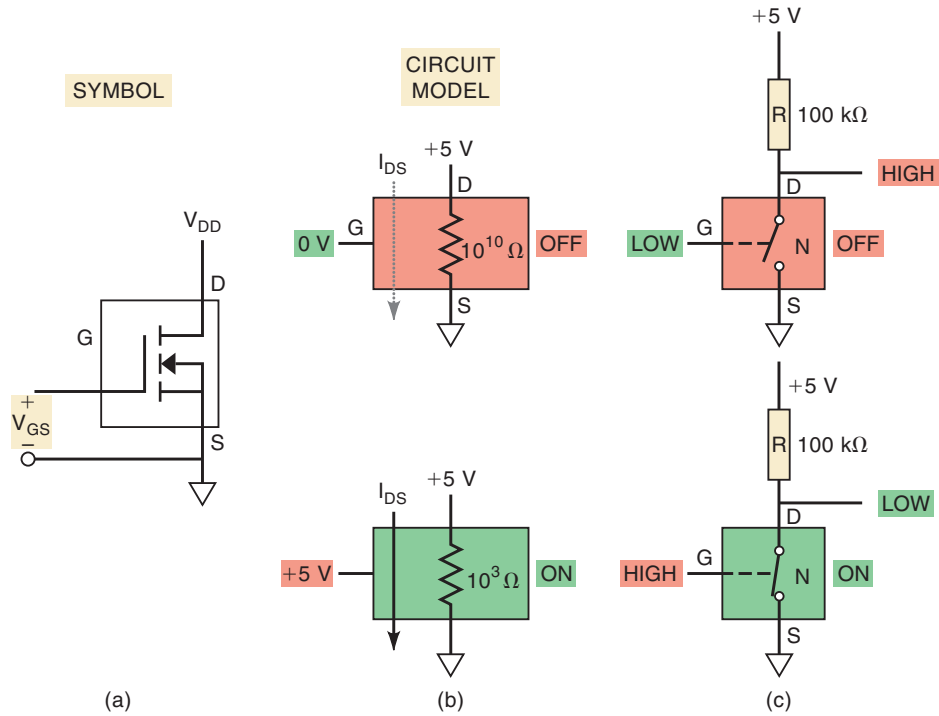
## Basic MOSFET Switch

Figure 8-20 shows the switching operation of an N-channel MOSFET, the basic element in a family of devices known as **N-MOS**. For the N-channel device, the drain is always biased positive relative to the source. The gate-to-source voltage  $V_{GS}$  is the input voltage, which is used to control the resistance between drain and source (i.e., the channel resistance) and therefore determines whether the device is on or off.

When  $V_{GS} = 0\text{ V}$ , there is no conductive channel between source and drain, and the device is off, as shown in Figure 8-20(b). Typically, the channel resistance in this OFF state is  $10^{10} \Omega$ , which for most purposes is an *open circuit*. The MOSFET will remain off as long as  $V_{GS}$  is zero or negative. As  $V_{GS}$  is made positive (gate positive relative to source), a threshold voltage ( $V_T$ ) is reached, at which point a conductive channel begins to form between source and drain. Typically  $V_T = +1.5\text{ V}$  for an N-MOSFET, and so any  $V_{GS} \geq 1.5\text{ V}$  will cause the MOSFET to conduct. Generally, a value of  $V_{GS}$  much larger than  $V_T$  is used to turn on the MOSFET more completely. As shown in Figure 8-20(b), when  $V_{GS} = +5\text{ V}$ , the channel resistance between source and drain has dropped to a value of  $R_{ON} = 1000 \Omega$ .

In essence, then, the N-MOS will switch from a very high resistance to a low resistance as the gate voltage switches from a LOW voltage to a

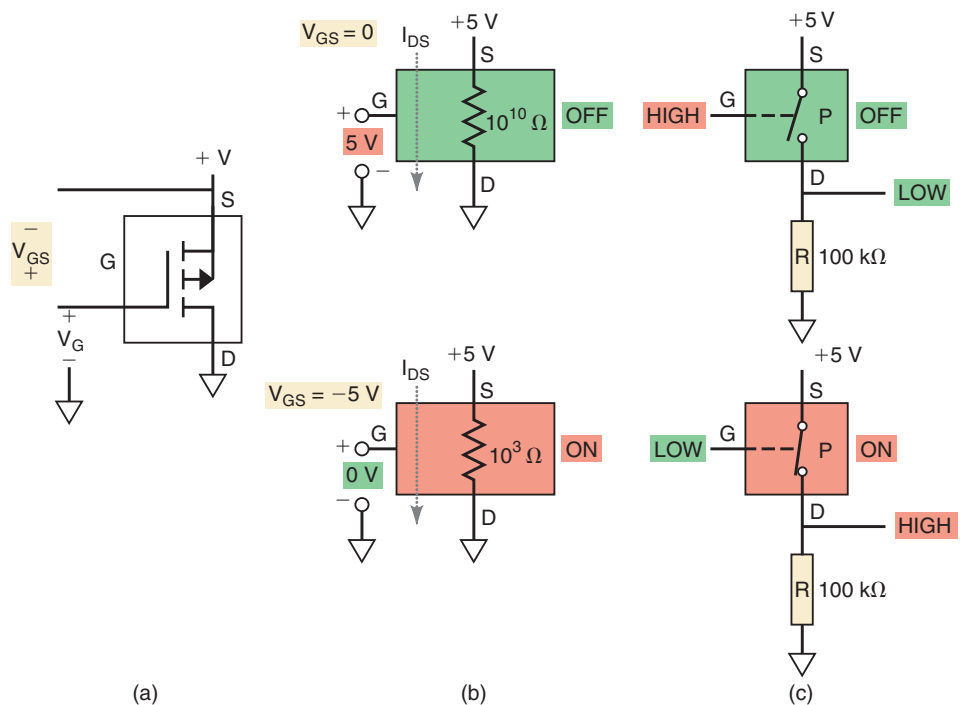
**FIGURE 8-20** N-channel MOSFET used as a switch: (a) symbol; (b) circuit model; (c) N-MOS inverter operation.



HIGH voltage. It is helpful simply to think of the MOSFET as a switch that is either opened or closed between source and drain. Figure 8-20(c) shows how an inverter can be formed using one N-MOS transistor as a switch. The first N-MOS logic devices were built using this approach. The drawback to this circuit, as with TTL, is that when the transistor is ON, there will always be current flowing from the supply to ground, producing heat.

The P-channel MOSFET, or **P-MOS**, shown in Figure 8-21(a) operates in exactly the same manner as the N-channel except that it uses voltages of

**FIGURE 8-21** P-channel MOSFET used as a switch: (a) symbol; (b) circuit model for OFF and ON; (c) P-MOS inverter circuit.



opposite polarity. For P-MOSFETs, the drain is connected to the lower side of the circuit so that it is biased with a more negative voltage relative to the source. To turn the P-MOSFET ON, a voltage *lower* than the source by  $V_T$  must be applied to the gate, meaning the voltage at the gate, relative to the source, must be negative.

Figure 8-21(b) shows that when the gate is at 5 V with respect to ground (the same voltage as applied to the source), the transistor is OFF and has a very high resistance from drain to source. When the gate is at 0 V (relative to ground), then the gate-to-source voltage  $V_{GS} = -5$  V and it turns the transistor ON, lowering its resistance from drain to source. The circuit of Figure 8-20(c) shows the switching action of an inverter using P-MOS logic.

Table 8-8 summarizes the P- and N-channel switching characteristics.

**TABLE 8-8** MOSFET characteristics.

	Drain-to-Source Bias	Gate-to-Source Voltage ( $V_{GS}$ ) Needed for Conduction	$R_{ON}$ ( $\Omega$ )	$R_{OFF}$ ( $\Omega$ )
P-channel	Negative	Typically more negative than $-1.5$ V	1000 (typical)	$10^{10}$
N-channel	Positive	Typically more positive than $+1.5$ V	1000 (typical)	$10^{10}$

### OUTCOME ASSESSMENT QUESTIONS

- Which value below is closest to the drain to source resistance of a MOSFET that is turned ON?  
1 ohm, 1 kilo-ohm, 1 mega-ohm, 1 giga-ohm
- Which value below is closest to the drain to source resistance of a MOSFET that is turned OFF?  
1 ohm, 1 kilo-ohm, 1 mega-ohm, 1 giga-ohm
- Which fundamental electrical component does the gate of a MOSFET represent?

## 8-8 COMPLEMENTARY MOS LOGIC

### OUTCOMES

*Upon completion of this section, you will be able to:*

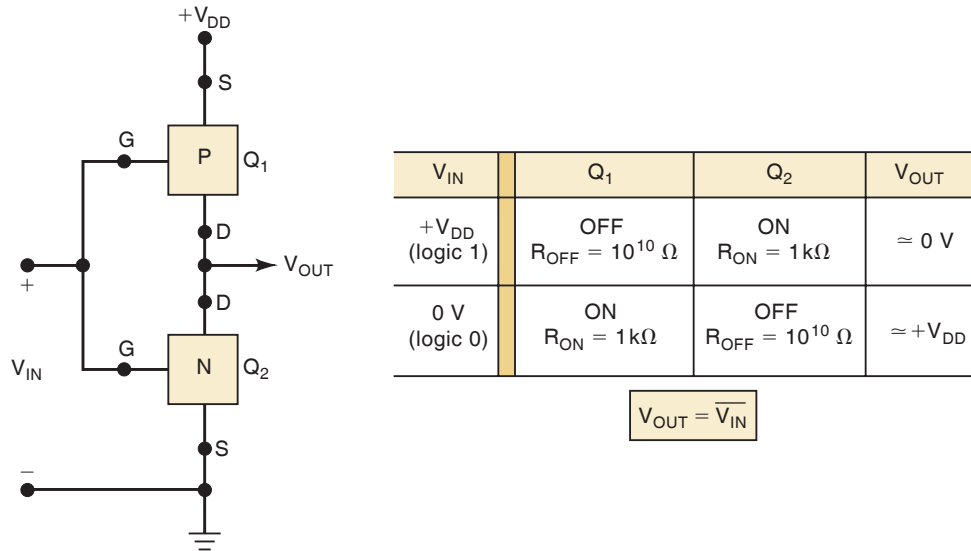
- Relate MOS transistor technology and circuit configuration to operating characteristics for CMOS.
- Relate logic function to circuit operation using CMOS technology circuits.

P-MOS and N-MOS logic circuits use fewer components and are much simpler to manufacture than TTL circuits. As a result, they began to dominate the LSI and VLSI markets in the 1970s and 1980s. During this era, a new technology began to emerge that used both P-MOS transistors (as high-side switches) and N-MOS transistors (as low-side switches) in the same logic circuit. This is referred to as complementary MOS, or **CMOS**, technology. CMOS logic circuits are not quite as simple and easy to manufacture as P-MOS or N-MOS, but they are faster, use much less power, and are the dominant technology in the market today.

### CMOS Inverter

Figure 8-22 shows the circuitry for the basic CMOS INVERTER. For this diagram and those that follow, the standard symbols for the MOSFETs have been replaced by blocks labeled P and N to denote a P-MOS and an N-MOS, respectively. This is done simply for convenience in analyzing the circuits. The CMOS INVERTER has two MOSFETs in series so that the P-channel device has its source connected to  $+V_{DD}$  (a positive voltage), and the N-channel device has its source connected to ground.\* The gates of the two devices are connected together as a common input. The drains of the two devices are connected together as the common output.

**FIGURE 8-22** Basic CMOS INVERTER.



The logic levels for CMOS are essentially  $+V_{DD}$  for logical 1 and 0 V for logical 0. Consider, first, the case where  $V_{IN} = +V_{DD}$ . In this situation, the gate of  $Q_1$  (P-channel) is at 0 V relative to the source of  $Q_1$ . Thus,  $Q_1$  will be in the OFF state with  $R_{OFF} \approx 10^{10} \Omega$ . The gate of  $Q_2$  (N-channel) will be at  $+V_{DD}$  relative to its source. Thus,  $Q_2$  will be on with typically  $R_{ON} = 1 \text{ k}\Omega$ . The voltage divider between  $Q_1$ 's  $R_{OFF}$  and  $Q_2$ 's  $R_{ON}$  will produce  $V_{OUT} \approx 0 \text{ V}$ .

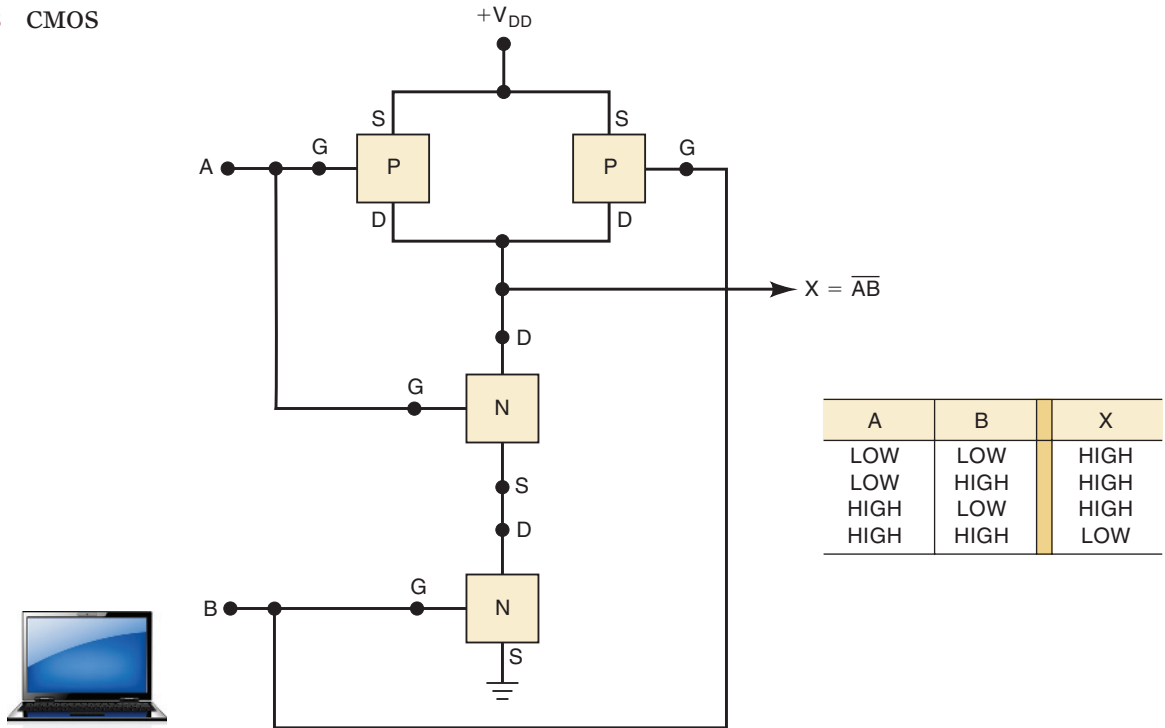
Next, consider the case where  $V_{IN} = 0 \text{ V}$ .  $Q_1$  now has its gate at a negative potential relative to its source, while  $Q_2$  has  $V_{GS} = 0 \text{ V}$ . Thus,  $Q_1$  will be on with  $R_{ON} = 1 \text{ k}\Omega$ , and  $Q_2$  will be off with  $R_{OFF} = 10^{10} \Omega$ , producing a  $V_{OUT}$  of approximately  $+V_{DD}$ . These two operating states are summarized in the table in Figure 8-22, showing that the circuit does act as a logic INVERTER.

### CMOS NAND Gate

Other logic functions can be constructed by modifying the basic INVERTER. Figure 8-23 shows a NAND gate formed by adding a parallel P-channel MOSFET and a series N-channel MOSFET to the basic INVERTER. To analyze this circuit, it helps to realize that a 0-V input turns on its corresponding P-MOS and turns off its corresponding N-MOS, and vice versa, for a  $+V_{DD}$  input. Thus, you can see that the only time a LOW output will occur is when inputs A and B are both HIGH ( $+V_{DD}$ ) to turn on both N-MOSFETs, thereby providing a low resistance from the output terminal to ground. For all other input conditions, at least one P-MOS will be on while at least one N-MOS will be off. This produces a HIGH output.

\*Most manufacturers label this terminal  $V_{SS}$ .

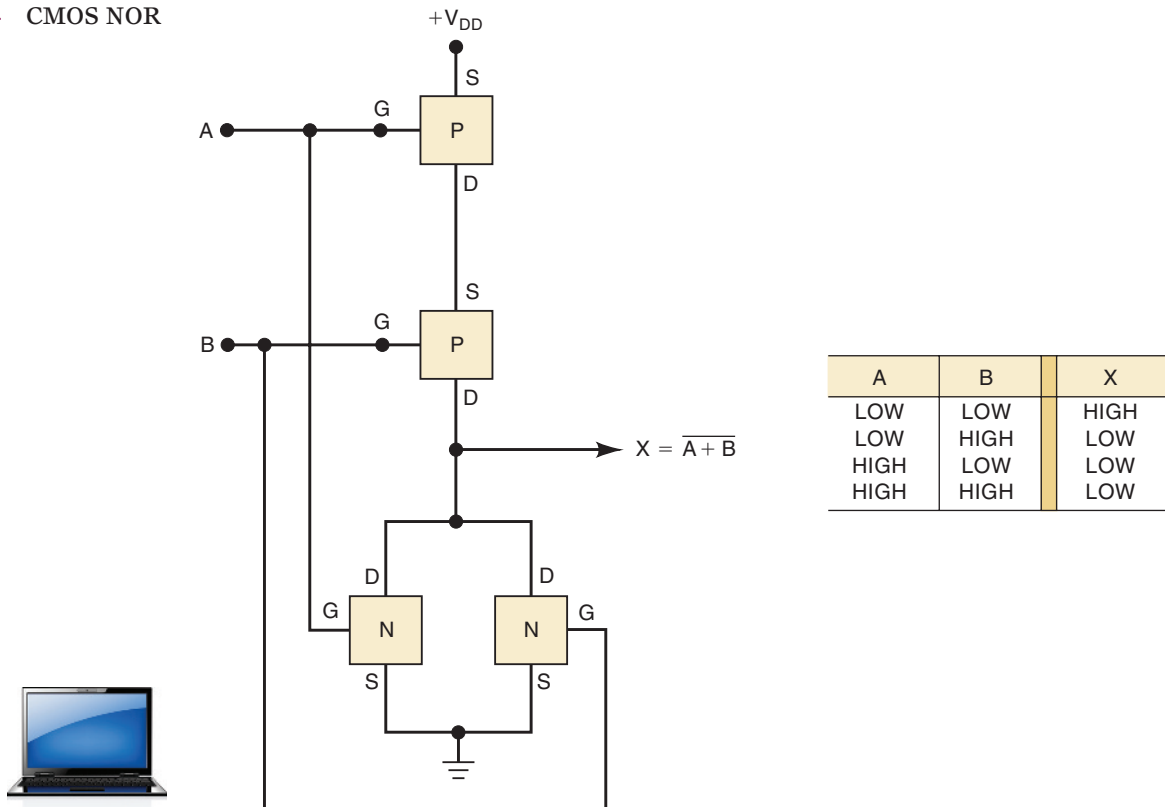
**FIGURE 8-23** CMOS NAND gate.



**CMOS NOR Gate**

A CMOS NOR gate is formed by adding a series P-MOS and a parallel N-MOS to the basic INVERTER, as shown in Figure 8-24. Once again, this circuit can be analyzed by realizing that a LOW at any input turns on its corresponding

**FIGURE 8-24** CMOS NOR gate.





P-MOS and turns off its corresponding N-MOS, and vice versa, for a HIGH input. It is left to the reader to verify that this circuit operates as a NOR gate.

CMOS AND and OR gates can be formed by combining NANDs and NORs with INVERTERS.

### CMOS SET-RESET FF

Two CMOS NOR gates or NAND gates can be cross-coupled to form a simple SET-RESET latch. Additional gating circuitry is used to convert the basic SET-RESET latch to clocked D and J-K flip-flops.

#### OUTCOME ASSESSMENT QUESTIONS

1. How does CMOS internal circuitry differ from N-MOS?
2. How many P-channel MOSFETs are in a CMOS INVERTER?
3. How many MOSFETs are in a three-input CMOS NAND gate?

## 8-9 CMOS SERIES CHARACTERISTICS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Assess compatibility of CMOS series with TTL series parts.
- Identify CMOS series and recognize their performance characteristics.
- Identify characteristics common to CMOS technology logic circuits.

CMOS ICs provide not only all of the same logic functions that are available in TTL but also several special-purpose functions not provided by TTL. Several different CMOS series have been developed over time because manufacturers have sought to improve performance characteristics. Before we look at the various CMOS series, it will be helpful to define a few terms that are used when ICs from different families or series are to be used together or as replacements for one another.

- **Pin-compatible.** Two ICs are pin-compatible when their pin configurations are the same. For example, pin 7 on both ICs is GROUND, pin 1 on both is an input to the first INVERTER, and so on.
- **Functionally equivalent.** Two ICs are functionally equivalent when the logic functions they perform are exactly the same. For example, both contain four two-input NAND gates, or both contain six D flip-flops with positive-edge clock triggering.
- **Electrically compatible.** Two ICs are electrically compatible when they can be connected directly to each other without taking any special measures to ensure proper operation.

### 4000/14000 Series

The oldest CMOS series is the 4000 series first introduced by RCA, and its functionally equivalent 14000 series from Motorola. Devices in the 4000/14000 series have very low power dissipation and can operate over a wide range of power-supply voltages (3 to 15 V). They are very slow compared to TTL and other CMOS series and have very low output current

capabilities. They are not pin-compatible or electrically compatible with any TTL series. The 4000/14000 series devices are rarely used in new designs except when a special-purpose IC is available that is not available in other series.

### 74HC/HCT (High-Speed CMOS)

The 74HC series has a 10-fold increase in switching speed, comparable to that of the 74LS devices, and a much higher output current capability than the first 7400 CMOS series ICs. 74HC/HCT ICs are pin-compatible with and functionally equivalent to TTL ICs with the same device number. 74HCT devices are electrically compatible with TTL, but 74HC devices are not. This means, for example, that a 74HCT04 hex-INVERTER chip can replace a 74LS04 chip, and vice versa. It also means that a 74HCT IC can be connected directly to any TTL IC.

### 74AC/ACT (Advanced CMOS)

This series is often referred to as ACL for advanced CMOS logic. The series is functionally equivalent to the various TTL series but is *not* pin-compatible with TTL because the pin placements on 74AC or 74ACT chips have been chosen to improve noise immunity so that the device inputs are less sensitive to signal changes occurring on other IC pins. 74AC devices are not electrically compatible with TTL; 74ACT devices can be connected directly to TTL. ACL offers advantages over the HC series in noise immunity, propagation delay, and maximum clock speed.

Device numbering for this series differs slightly from TTL, 74C, and 74HC/HCT numbering. It uses a five-digit device number beginning with the digits 11. The following examples illustrate:

$$\begin{array}{l} 74AC11 \quad 004 \quad \equiv \quad 74HC \quad 04 \\ 74ACT11 \quad 293 \quad \equiv \quad 74HCT \quad 293 \end{array}$$

### 74AHC/AHCT (Advanced High-Speed CMOS)

This series of CMOS devices offers a natural migration path from the HC series to faster, lower-power, low-drive applications. The devices in this series are three times faster and can be used as direct replacements for HC series devices. They offer similar noise immunity to HC without the overshoot/undershoot problems often associated with higher drive characteristics required for comparable speed.

### BiCMOS 5-V Logic

Several IC manufacturers have developed logic series that combine the best features of bipolar and CMOS logic—called BiCMOS logic. The low-power characteristics of CMOS and the high-speed characteristics of bipolar circuits are integrated to produce an extremely low-power, high-speed logic family. BiCMOS ICs are not available in most SSI and MSI functions, but are limited to functions that are used in microprocessor and bus interfacing applications such as latches, buffers, drivers, and transceivers. The 74BCT (BiCMOS bus-interface technology) series offers 75% reduction in power consumption over the 74F family while maintaining similar speed and drive characteristics. Parts in this series are pin-compatible with industry standard TTL parts and operate on standard 5-V logic levels. The 74ABT (advanced BiCMOS

technology) series is the second generation of BiCMOS bus-interface devices. Details of bus interface logic will be presented in Section 8-13.

### Power-Supply Voltage

The 4000/14000 series and 74C series devices operate with  $V_{DD}$  values ranging from 3 to 15 V, which makes them very versatile. They can be used in low-voltage battery-operated circuits, in standard 5-V circuits, and in circuits where a higher supply voltage is used to attain the noise margins required for operation in a high-noise environment. The 74HC/HCT, 74AC/ACT, and 74AHC/AHCT series operate over a much narrower range of supply voltages, typically between 2 and 6 V.

Logic series that are designed to operate at lower voltages (e.g., 2.5 or 3.3 V) are also available. Whenever devices that use different power-supply voltages are interconnected in the same digital system, special measures must be taken. The low-voltage devices and the special interfacing techniques will be covered in Section 8-10.

### Logic Voltage Levels

The input and output voltage levels will be different for the different CMOS series. Table 8-9 lists these voltage values for the various CMOS series as well as those for the TTL series. The values listed in the table assume that all devices are operating from a supply voltage of 5 V and that all device outputs are driving inputs of the same logic family.

**TABLE 8-9** Input/output voltage levels (in volts) with  $V_{DD} = V_{CC} = +5\text{V}$ .

Parameter	CMOS							TTL			
	4000B	74HC	74HCT	74AC	74ACT	74AHC	74AHCT	74	74LS	74AS	74ALS
$V_{IH}(\text{min})$	3.5	3.5	2.0	3.5	2.0	3.85	2.0	2.0	2.0	2.0	2.0
$V_{IL}(\text{max})$	1.5	1.0	0.8	1.5	0.8	1.65	0.8	0.8	0.8	0.8	0.8
$V_{OH}(\text{min})$	4.95	4.9	4.9	4.9	4.9	4.4	3.15	2.4	2.7	2.7	2.5
$V_{OL}(\text{max})$	0.05	0.1	0.1	0.1	0.1	0.44	0.1	0.4	0.5	0.5	0.5
$V_{NH}$	1.45	1.4	2.9	1.4	2.9	0.55	1.15	0.4	0.7	0.7	0.7
$V_{NL}$	1.45	0.9	0.7	1.4	0.7	1.21	0.7	0.4	0.3	0.3	0.4

Examination of this table discloses some important points. First, note that  $V_{OL}$  for the CMOS devices is very close to 0 V, and  $V_{OH}$  is very close to 5 V. The reason why is that the CMOS outputs do not have to source or sink any significant amount of current when they are driving CMOS inputs with their extremely high input resistance ( $10^{12} \Omega$ ). Also note that, except for 74HCT and 74ACT, the required input voltage levels are greater for CMOS than for TTL. Recall that 74HCT and 74ACT are designed to be electrically compatible with TTL, so they must be able to accept the same input voltage levels as TTL.

### Noise Margins

The noise margins for each series are also given in Table 8-9. They are calculated using

$$V_{NH} = V_{OH}(\text{min}) - V_{IH}(\text{min})$$

$$V_{NL} = V_{IL}(\text{max}) - V_{OL}(\text{max})$$

Note that, in general, the CMOS devices have greater noise margins than TTL. The difference would be even greater if the CMOS devices were operated at a supply voltage greater than 5 V.

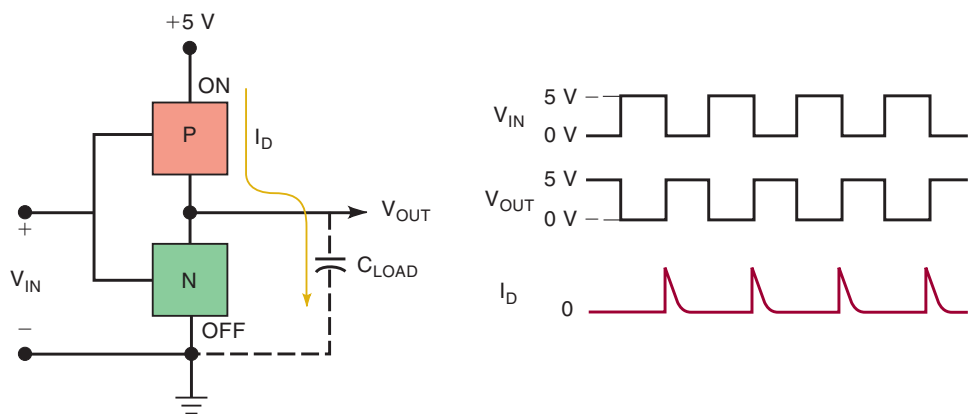
## Power Dissipation

When a CMOS logic circuit is in a static state (not changing), its power dissipation is extremely low. We can see the reason by examining each of the circuits shown in Figures 8-22 to 8-24. Note that, regardless of the state of the output, there is always a very high resistance between the  $V_{DD}$  terminal and ground because there is always an off MOSFET in the current path. This results in a typical CMOS DC power dissipation of only 2.5 nW per gate when  $V_{DD} = 5$  V; even at  $V_{DD} = 10$  V, this power increases to only 10 nW. With these values for  $P_D$ , it is easy to see why CMOS is ideally suited for applications using battery power or battery backup power.

## $P_D$ Increases with Frequency

The power dissipation of a CMOS IC will be very low as long as it is in a DC condition. Unfortunately,  $P_D$  will increase in proportion to the frequency at which the circuits are switching states. For example, a CMOS NAND gate that has  $P_D = 10$  nW under DC conditions will have  $P_D = 0.1$  mW at a frequency of 100 kpps, and 1 mW at 1 MHz. The reason for this dependence on frequency is illustrated in Figure 8-25.

**FIGURE 8-25** Current spikes are drawn from the  $V_{DD}$  supply each time the output switches from LOW to HIGH. This is due mainly to the charging current of the load capacitance.



Each time a CMOS output switches from LOW to HIGH, a transient charging current must be supplied to the load capacitance. This capacitance consists of the combined input capacitances of any loads being driven and the device's own output capacitance. These narrow spikes of current are supplied by  $V_{DD}$  and can have a typical amplitude of 5 mA and a duration of 20 to 30 ns. Clearly, as the switching frequency increases, there will be more of these current spikes occurring per second, and the average current drawn from  $V_{DD}$  will increase. Even with very low capacitive loads, there is a brief point in the transition from LOW to HIGH or HIGH to LOW when the two output transistors are partially turned on. This effectively lowers the resistance from the supply to ground, causing a current spike as well.

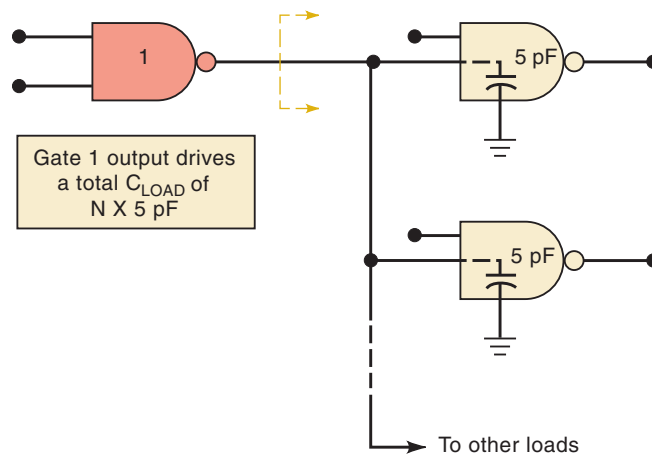
Thus, at higher frequencies, CMOS begins to lose some of its advantage over other logic families. As a general rule, a CMOS gate will have the same average  $P_D$  as a 74LS gate at frequencies near 2 to 3 MHz. Above these frequencies, TTL power also increases with frequency because of the current

required to reverse the charge on the load capacitance. For MSI chips, the situation is somewhat more complex than stated here, and a logic designer must do a detailed analysis to determine whether or not CMOS has a power-dissipation advantage at a particular frequency of operation.

### Fan-Out

Like N-MOS and P-MOS, CMOS inputs have an extremely large resistance ( $10^{12} \Omega$ ) that draws essentially no current from the signal source. Each CMOS input, however, typically presents a 5-pF load to ground. This input capacitance limits the number of CMOS inputs that one CMOS output can drive (see Figure 8-26). The CMOS output must charge and discharge the parallel combination of all of the input capacitances, so that the output switching time will be increased in proportion to the number of loads being driven. Typically, each CMOS load increases the driving circuit's propagation delay by 3 ns. For example, NAND gate 1 in Figure 8-26 might have a  $t_{PLH}$  of 25 ns if it were driving no loads; this would increase to  $25 \text{ ns} + 20(3 \text{ ns}) = 85 \text{ ns}$  if it were driving *twenty* loads.

**FIGURE 8-26** Each CMOS input adds to the total load capacitance seen by the driving gate's output.



Thus, CMOS fan-out depends on the permissible maximum propagation delay. Typically, CMOS outputs are limited to a fan-out of 50 for low-frequency operation ( $\leq 1 \text{ MHz}$ ). Of course, for higher-frequency operation, the fan-out would have to be less.

### Switching Speed

Although CMOS, like N-MOS and P-MOS, must drive relatively large load capacitances, its switching speed is somewhat faster because of its low output resistance in each state. An N-MOS output must charge the load capacitance through a relatively large ( $100\text{-k}\Omega$ ) resistance. In the CMOS circuit, the output resistance in the HIGH state is the  $R_{ON}$  of the P-MOSFET, which is typically  $1 \text{ k}\Omega$  or less. This allows more rapid charging of load capacitance.

A 4000 series NAND gate will typically have an average  $t_{pd}$  of 50 ns at  $V_{DD} = 5 \text{ V}$ , and 25 ns at  $V_{DD} = 10 \text{ V}$ . The reason for the improvement in  $t_{pd}$  as  $V_{DD}$  is increased is that the  $R_{ON}$  on the MOSFETs decreases significantly at higher supply voltages. Thus, it appears that  $V_{DD}$  should be made as large as possible for operation at higher frequencies. However, the larger  $V_{DD}$  will result in increased power dissipation.

A typical NAND gate in the 74HC or 74HCT series has an average  $t_{pd}$  of around 8 ns when operated at  $V_{DD} = 5$  V. A 74AC/ACT NAND gate has an average  $t_{pd}$  of around 4.7 ns. A 74AHC NAND gate has an average  $t_{pd}$  of around 4.3 ns.

## Unused Inputs

**CMOS inputs should never be left disconnected. All CMOS inputs must be tied either to a fixed voltage level (0 V or  $V_{DD}$ ) or to another input.**

This rule applies even to the inputs of extra unused logic gates on a chip. An unconnected CMOS input is susceptible to noise and static charges that could easily bias both the P-channel and the N-channel MOSFETs in the conductive state, resulting in increased power dissipation and possible overheating.

## Static Sensitivity

All electronic devices, to varying degrees, are sensitive to damage by static electricity. The human body is a great storehouse of electrostatic charges. For example, when you walk across a carpet, a static charge of over 30,000 V can be built up on your body. If you then touch an electronic device, some of this sizable charge can be transferred to the device. The MOS logic families (and all MOSFETs) are especially susceptible to static-charge damage. All of this potential difference (static charge) applied across the thin oxide layer overcomes its dielectric insulation capability. When it breaks down, the resulting flow of current (discharge) is like a lightning strike, blowing a hole in the oxide layer and permanently damaging the device.

**Electrostatic discharge (ESD)** causes damage to a great deal of electronic equipment annually, and equipment manufacturers have devoted considerable attention to developing special handling procedures for all electronic devices and circuits. Even though most modern ICs have on-chip resistor–diode networks to protect inputs and outputs from the effects of ESD, the following precautions are used by most engineering labs, production facilities, and field service departments:

1. Connect the chassis of all test instruments, soldering-iron tips, and your workbench (if metal) to earth ground (i.e., the round prong in the 120-VAC plug). This prevents the buildup of static charge on these devices that could be transferred to any circuit board or IC that they come in contact with.
  2. Connect yourself to earth ground with a special wrist strap. This will allow potentially dangerous charges from your body to be discharged to ground. The wrist strap contains a 1-M $\Omega$  resistor that limits current to a nonlethal value should you accidentally touch a “live” voltage while working with the equipment.
  3. Keep ICs (especially MOS) in conductive foam or aluminum foil. This will keep all IC pins shorted together so that no dangerous voltages can be developed between any two pins.
  4. Avoid touching IC pins, and insert the IC into the circuit immediately after removing it from the protective carrier.
-

5. Place shorting straps across the edge connectors of PC boards when the boards are being carried or transported. Avoid touching the edge connectors. Store PC boards in conductive plastic or metallic envelopes.
6. Do not leave any unused IC inputs unconnected because open inputs tend to pick up stray static charges.

### Latch-Up

Because of the unavoidable existence of *parasitic* (unwanted) PNP and NPN transistors embedded in the substrate of CMOS ICs, a condition known as **latch-up** can occur under certain circumstances. If these parasitic transistors on a CMOS chip are triggered into conduction, they will latch-up (stay ON permanently), and a large current may flow and destroy the IC. Most modern CMOS ICs are designed with protection circuitry that helps prevent latch-up, but it can still occur when the device's maximum voltage ratings are exceeded. Latch-up can be triggered by high-voltage spikes or ringing at the device inputs and outputs. Clamping diodes can be connected externally to protect against such transients, especially when the ICs are used in industrial environments where high-voltage and/or high-current load switching takes place (motor controllers, relays, etc.). A well-regulated power supply will minimize spikes on the  $V_{DD}$  line; if the supply also has current limiting, it will limit current should latch-up occur. Modern CMOS fabrication techniques have greatly reduced ICs' susceptibility to latch-up.

#### OUTCOME ASSESSMENT QUESTIONS

1. Which CMOS series is pin-compatible with TTL?
2. Which CMOS series is electrically compatible with TTL?
3. Which CMOS series is functionally equivalent to TTL?
4. What logic family combines the best features of CMOS and bipolar logic?
5. What factors determine CMOS fan-out?
6. What precautions should be taken when handling CMOS ICs?
7. Which IC family (CMOS, TTL) is best suited for battery-powered applications?
8. *True or false:*
  - (a) CMOS power drain increases with operating frequency.
  - (b) Unused CMOS inputs can be left unconnected.
  - (c) TTL is better suited than CMOS for operation in high-noise environments.
  - (d) CMOS switching speed increases with operating frequency.
  - (e) CMOS switching speed increases with supply voltage.
  - (f) The latch-up condition is an advantage of CMOS over TTL.

## 8-10 LOW-VOLTAGE TECHNOLOGY

### OUTCOME

Upon completion of this chapter, you will be able to:

- Identify series of logic devices and technologies that operate at lower voltage levels.

IC manufacturers are continually looking for ways to put semiconductor devices (diodes, resistors, transistors, etc.) closer together on a chip, that is, to increase the chip density. This higher density has at least two major benefits. First, it allows more circuits to be packed onto the chip; second, with the circuits closer together, the time for signals to propagate from one circuit to another will decrease, thereby improving overall circuit operating speed. There are also drawbacks to higher chip density. When circuits are placed closer together, the insulating material that isolates one circuit from another is narrower. This decreases the amount of voltage that the device can withstand before dielectric breakdown occurs. Increasing the chip density increases the overall chip power dissipation, which can raise the chip temperature above the maximum level allowed for reliable operation.

These drawbacks can be neutralized by operating the chip at lower voltage levels, thereby reducing power dissipation. Several series of logic on the market operate on 3.3 V. The newer series are optimized to run on 2.5 V. This low-voltage technology may very well signal the beginning of a gradual transition in the digital equipment field that will eventually find all digital ICs operating from a new low-voltage standard.

Low-voltage devices are currently designed for applications ranging from electronic games to engineering workstations. The newer CPUs are 2.5-V devices, and 3.3-V dynamic RAM chips are used in memory modules for personal computers.

Several low-voltage logic series are currently available. It is not possible to cover all of the families and series from all manufacturers, so we will describe those currently offered by Texas Instruments.

## CMOS Family

- The *74LVC (Low-Voltage CMOS)* series contains the widest assortment of the familiar SSI gates and MSI functions of the 5-V families, along with many bus-interface devices such as buffers, latches, drivers, and so on. This series can handle 5-V logic levels on its inputs, so it can convert from 5-V systems to 3-V systems. As long as the current drive is kept low enough to keep the output voltage within acceptable limits, the 74LVC can also drive 5-V TTL inputs. The  $V_{IH}$  input requirements of 5-V CMOS parts such as the 74HC/AHC do not allow LVC devices to drive them.
  - The *74ALVC (Advanced Low-Voltage CMOS)* series currently offers the highest performance. The devices in this series are intended primarily for bus-interface applications that use 3.3-V logic only.
  - The *74LV (Low-Voltage)* series offers CMOS technology and many of the common SSI gates and MSI logic functions, along with some popular octal buffers, latches, and flip-flops. It is intended to operate only with other 3.3-V devices.
  - The *74AVC (Advanced Very-Low-Voltage CMOS)* series has been introduced with tomorrow's systems in mind. It is optimized for 2.5-V systems, but it can operate on supplies as low as 1.2 V or as high as 3.3 V. This broad range of supply voltage makes it useful in mixed-voltage systems. It has propagation delays of less than 2 ns, which rivals 74AS bipolar devices. It has many of the bus interface features of the BiCMOS series that will make it useful in future generations of low-voltage workstations, PCs, networks, and telecommunications equipment.
  - The *74AUC (Advanced Ultra-Low-Voltage CMOS)* series is optimized to operate at 1.8-V logic levels.
-



- The *74AUP (Advanced Ultra-low Power)* series is the industries lowest-power logic series and is used in battery-operated portable applications.
- The *74CBT (Cross Bar Technology)* series offers high-speed bus-interface circuits that can switch quickly when enabled and not load the bus when they are disabled.
- The *74CBTLV (Cross Bar Technology Low Voltage)* is the 3.3-V complement to the 74CBT series.
- The *74GTLP (Gunning Transceiver Logic Plus)* series is made for high-speed parallel backplane applications. This series will be covered in a later section.
- The *74SSTV (Stub Series Terminated Logic)* is useful in the high-speed advanced-memory systems of today's computers.
- The *TS Switch (TI Signal Switch)* series is made for mixed-signal applications and offers some analog and digital switching and multiplexing solutions.
- The *74TVC (Translation Voltage Clamp)* series is used to protect the inputs and outputs of sensitive devices from voltage overshoot on the bus lines.

### BiCMOS Family

- The *74LVLT (Low-Voltage BiCMOS Technology)* contains BiCMOS parts that are intended for 8- and 16-bit bus-interface applications. As with the LVC series, the inputs can handle 5-V logic levels and serve as a 5-V to 3-V translator. Because the output levels [ $V_{OH(min)}$  and  $V_{OL(max)}$ ] are equivalent to TTL levels, they are fully electrically compatible with TTL. Table 8-10 compares the various features.

**TABLE 8-10** Low-voltage series characteristics.

	LV	ALVC	AVC	ALVT	ALB
$V_{CC}$ (recommended)	2.7–3.6	2.3–3.6	1.65–3.6	2.3–2.7	3–3.6
$t_{pd}$ (ns)	18	3	1.9	3.5	2
$V_{IH}$ (V)	2 to $V_{CC} + 0.5$	2.0 to 4.6	1.2 to 4.6	2 to 7	2.2 to 4.6
$V_{IL}$ (V)	0.8	0.8	0.7	0.8	0.6
$I_{OH}$ (mA)	6	12	8	32	25
$I_{OL}$ (mA)	6	12	8	32	25

- The *74ALVT (Advanced Low-Voltage BiCMOS Technology)* series is an improvement over the LVT series. It offers 3.3-V or 2.5-V operation at 3 ns and is pin-compatible with existing ABT and LVT series. It is also intended for bus-interface applications.
- The *74ALB (Advanced Low-Voltage BiCMOS)* series is designed for 3.3-V bus-interface applications. It provides 25 mA output drive and propagation delays of only 2.2 ns.
- The *74VME (VERSA Module Eurocard)* series is designed to operate with the standard VME bus technology.

Digital technicians and engineers can no longer assume that every IC in a digital circuit, system, or piece of equipment is operating at 5 V, and they

must be prepared to deal with the necessary interfacing considerations in mixed-voltage systems. The interfacing skills you learn in this chapter will allow you to accomplish this, regardless of what develops as low-voltage systems become more common.

The continued advancement of mobile devices has demanded low-voltage/low-power circuitry. Today there is still a mix of 5 V, 3.3 V, 2.5 V, 1.8 V, and even 1.5 V technology parts on the market.

### OUTCOME ASSESSMENT QUESTIONS

1. What are the two advantages of higher-density ICs?
2. What are the drawbacks?
3. What is the minimum HIGH voltage at a 74LVT input?
4. Which low-voltage series can work only with other low-voltage series ICs?
5. Which low-voltage series is fully electrically compatible with TTL?

## 8-11 OPEN-COLLECTOR/OPEN-DRAIN OUTPUTS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Define open-collector/open-drain in terms of internal circuitry.
- Apply open-collector/open-drain output circuits to design needs.
- Identify circuits that have open-collector/open-drain outputs.

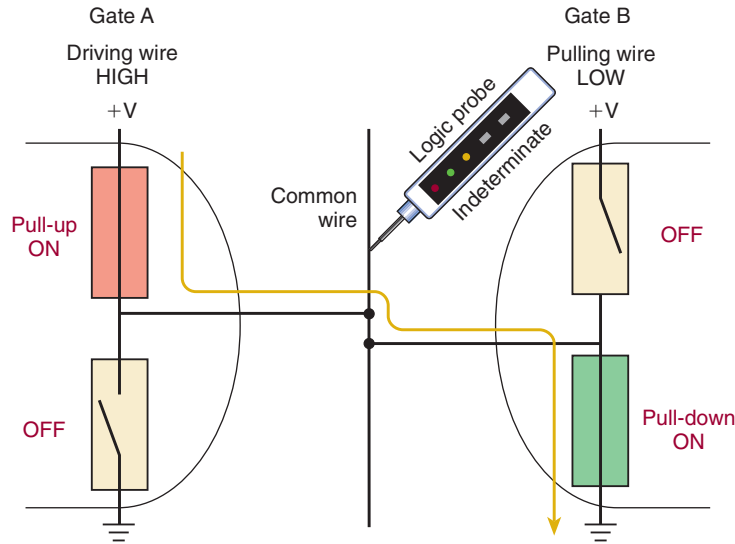
Several digital devices must sometimes share the use of a single wire in order to transmit a signal to some destination device, very much like several neighbors sharing the same street. This means that several devices must have their outputs connected to the same wire, which essentially connects them all to each other. For all of the logic devices we have considered so far, this presents a problem. Each output has two states, HIGH and LOW. When one output is HIGH while the other is LOW and when they are connected together, we have a HIGH/LOW conflict. Which one will win? Just like arm wrestling, the stronger of the two wins. In this case, the transistor circuit whose output transistor has the lowest “ON” resistance will pull the output voltage in its direction.

Figure 8-27 shows a generic block diagram of two logic devices with their outputs connected to a common wire. If the two logic devices were CMOS, then the ON resistance of the pull-up circuit that outputs the HIGH would be approximately the same as the ON resistance of the pull-down circuit that outputs the LOW. The voltage on the common wire will be about half the supply voltage. This voltage is in the indeterminate range for most CMOS series and is unacceptable for driving a CMOS input. Furthermore, the current through the two conducting MOSFETs will be much greater than normal, especially at higher values of  $V_{DD}$ , and it can damage the ICs.

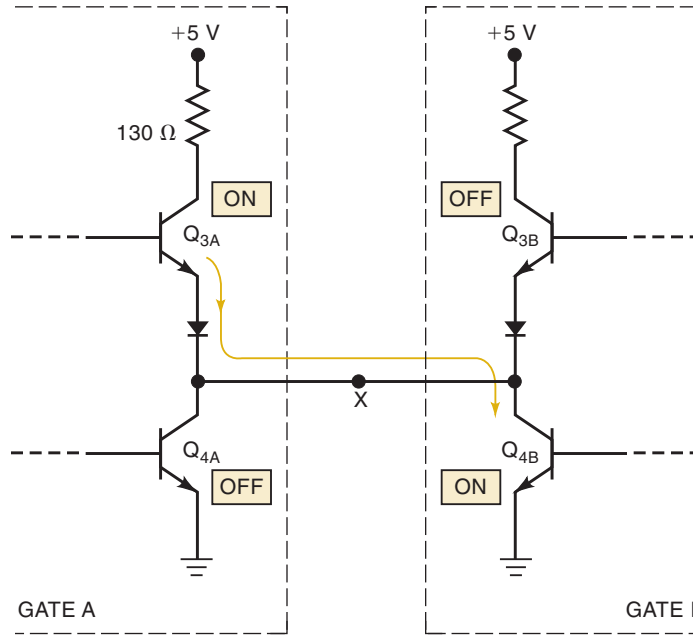
**Conventional CMOS outputs should never be connected together.**

If the two devices were TTL totem-pole outputs, as shown in Figure 8-28, a similar situation would occur but with different results because of the difference in output circuitry. Suppose that gate *A* output is in the HIGH state ( $Q_{3A}$  ON,  $Q_{4A}$  OFF) and the gate *B* output is in the LOW state ( $Q_{3B}$  OFF,  $Q_{4B}$  ON).

**FIGURE 8-27** Two outputs contending for control of a wire.



**FIGURE 8-28** Totem-pole outputs tied together can produce harmful current through  $Q_4$ .



In this situation,  $Q_{4B}$  is a very low resistance load on  $Q_{3A}$  and will draw a current that is far greater than it is rated to handle. This current might not damage  $Q_{3A}$  or  $Q_{4B}$  immediately, but over a period of time it can cause overheating and deterioration in performance and eventual device failure.

Another problem caused by this relatively high current flowing through  $Q_{4B}$  is that it will produce a larger voltage drop across the transistor collector emitter, making  $V_{OL}$  of between 0.5 and 1 V. This is greater than the allowable  $V_{OL(max)}$ . For these reasons:

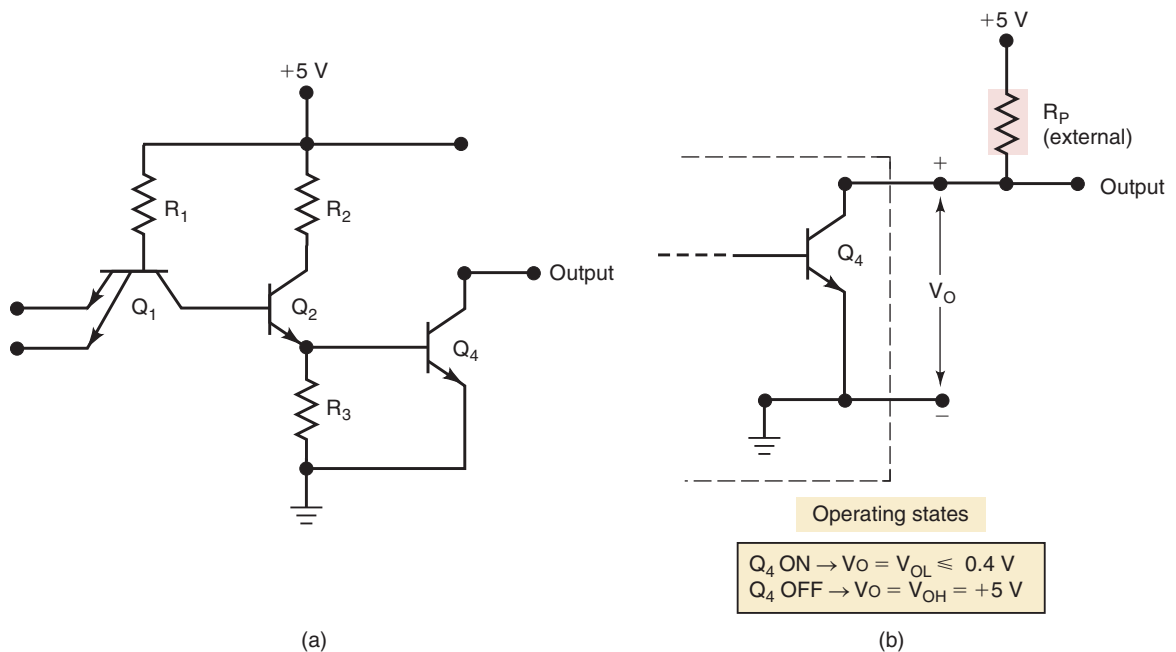
**TTL totem-pole outputs should never be tied together.**

### Open-Collector/Open-Drain Outputs

One solution to the problem of sharing a common wire among gates is to remove the active pull-up transistor from each gate's output circuit. In this way, none of the gates will ever try to assert a logic HIGH. TTL outputs

that have been modified in this way are called **open-collector outputs**. CMOS output circuits that have been modified in this way are called **open-drain outputs**. The output is taken at the drain of the N-channel pull-down MOSFET, which is an open circuit (i.e., not connected to any other circuitry).

The TTL equivalent is called an open-collector output because the collector of the bottom transistor in the totem pole is connected directly to the output pin and nowhere else, as shown in Figure 8-29(a). The open-collector structure eliminates the pull-up transistor  $Q_3$ ,  $D_1$ , and  $R_4$ . In the output LOW state,  $Q_4$  is ON (has base current and is essentially a short between collector and emitter); in the output HIGH state,  $Q_4$  is OFF (has no base current and is essentially an open between collector and emitter). Because this circuit has no internal way to pull the output HIGH, the circuit designer must connect an external pull-up resistor  $R_P$  to the output, as shown in Figure 8-29(b).

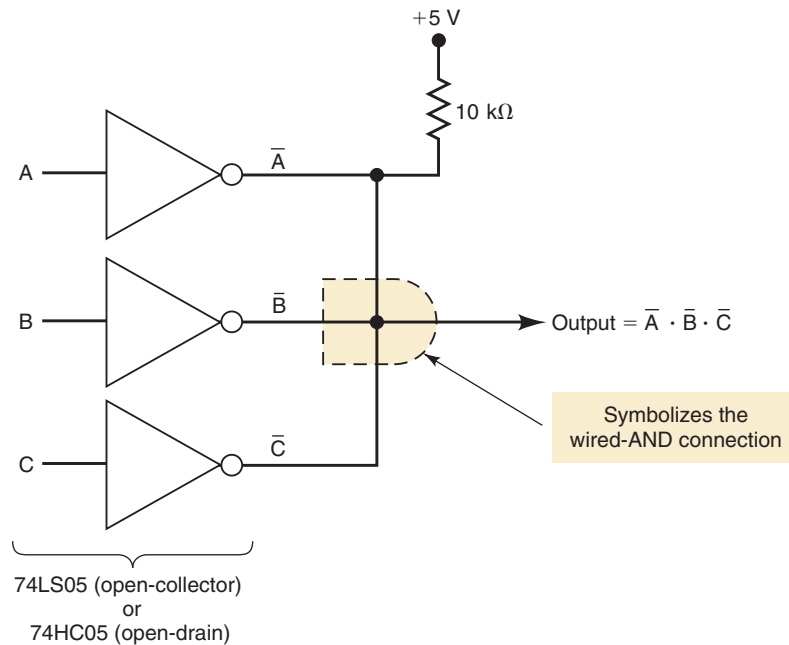


**FIGURE 8-29** (a) Open-collector TTL circuit; (b) with external pull-up resistor.

When  $Q_4$  is ON, it pulls the output voltage down to a LOW. When  $Q_4$  is OFF,  $R_P$  pulls the output of the gate HIGH. Note that without the pull-up resistor, the output voltage would be indeterminate (floating). The value of the resistor  $R_P$  is usually chosen to be  $10 \text{ k}\Omega$ . This value is small enough so that, in the HIGH state, the voltage dropped across it due to load current will not lower the output voltage below the minimum  $V_{OH}$ . It is large enough so that, in the LOW state, it will limit the current through  $Q_4$  to a value below  $I_{OL}(\text{max})$ .

When several open-collector or open-drain gates share a common connection, as shown in Figure 8-30, the common wire is HIGH by default due to the pull-up resistor. When any one (or more) of the gate outputs pulls it LOW, the  $5 \text{ V}$  are dropped across  $R_P$  and the common connection is in the LOW state. Because the common output is HIGH only when all the outputs are in the HIGH state, connecting the outputs in this way essentially implements the logic AND function. This is called a **wired-AND** connection. This is shown symbolically by the dotted AND gate symbol. There is no actual AND gate there. *A wired-AND can be implemented only with open-collector TTL and open-drain CMOS logic devices.*

**FIGURE 8-30** Wired-AND operation using open-collector gates.



To summarize, the open-collector/open-drain circuits cannot actively make their outputs HIGH; they can only pull them LOW. This feature can be used to allow several devices to share the same wire for transmitting a logic level to another device or to combine the outputs of the devices effectively in a logic AND function. As we mentioned before, the purpose of the active pull-up transistor in the output circuit of conventional gates is to charge up the load capacitance rapidly and allow for fast switching. Open-collector and open-drain devices have a much slower switching speed from LOW to HIGH and consequently are not used in high-speed applications.

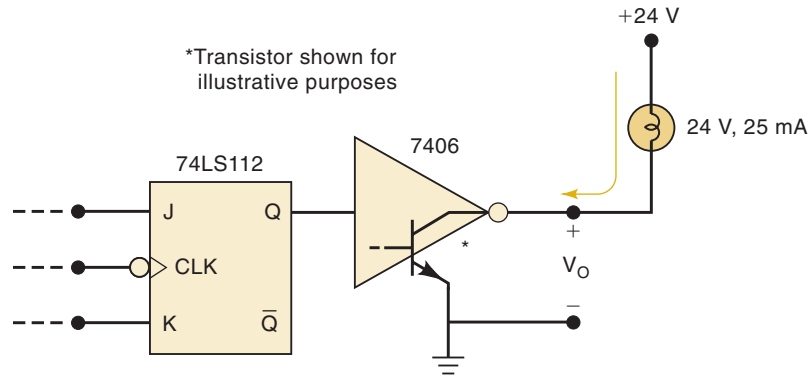
### Open-Collector/Open-Drain Buffer/Drivers

The applications of open-collector/drain outputs that we have described were more prevalent in the early days of logic circuits than they are today. A more common use of these circuits now is as a **buffer/driver**. A buffer or a driver is a logic circuit that is designed to have a greater output current and/or voltage capability than an ordinary logic circuit. They allow a weaker output circuit to drive a heavy load. Open-collector/drain circuits offer some unique flexibility as buffer/drivers.

Due to their high  $I_{OL}$  and  $V_{OH}$  specifications, the 7406 and 7407 are the only standard TTL devices that are still being recommended for new designs. The 7406 is an open-collector buffer/driver IC that contains six INVERTERS with open-collector outputs that can sink up to 40 mA in the LOW state. In addition, the 7406 can handle output voltages up to 30 V in the HIGH state. This means that the output can be connected to a load that operates on a voltage greater than 5 V. This is illustrated in Figure 8-31, where a 7406 is used as a buffer between a 74LS112 flip-flop and an incandescent indicator lamp that is rated at 24 V, 25 mA. The 7406 controls the lamp's ON/OFF status to indicate the state of FF output  $Q$ . Note that the lamp is powered from +24 V, and it acts as the pull-up resistor for the open-collector output.

When  $Q = 1$ , the 7406 output goes LOW, its output transistor sinks the 25 mA of lamp current supplied by the 24-V source, and the lamp is on. When  $Q = 0$ , the 7406 output transistor turns off; there is no path for

**FIGURE 8-31** An open-collector buffer/driver drives a high-current, high-voltage load.



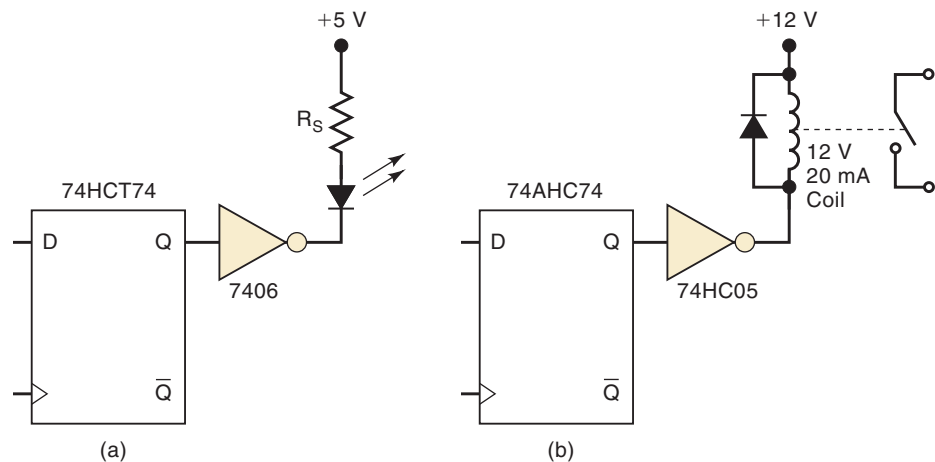
current, and the lamp turns off. In this state, the full 24 V will appear across the OFF output transistor so that  $V_{OH} = 24\text{ V}$ , which is lower than the 7406 maximum  $V_{OH}$  rating.

Open-collector outputs are often used to drive indicator LEDs, as shown in Figure 8-32(a). The resistor is used to limit the current to a safe value. When the INVERTER output is LOW, its output transistor provides a low-resistance path to ground for the LED current, so that the LED is on. When the INVERTER output is HIGH, its output transistor is off, and there is no path for LED current; in this state, the LED is off.

The 7407 is an open-collector, noninverting buffer with the same voltage and current ratings as a 7406.

The 74HC05 is an open-drain hex inverter with 25 mA current sink capability. Figure 8-32(b) shows a way to interface a 74AHC74 D-FF to a control relay. A control relay is an electromagnetic switch. The contacts close magnetically when the rated current flows through the coil. The 74HC05 can handle the relay's relatively high voltage and current so that the 74AHC74 output can turn the relay on and off.

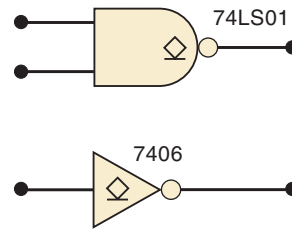
**FIGURE 8-32** (a) An open-collector output can be used to drive an LED indicator; (b) an open-drain CMOS output.



### IEEE/ANSI Symbol for Open-Collector/Drain Outputs

IEEE/ANSI symbology uses a distinctive notation to identify open-collector/drain outputs. Figure 8-33 shows the standard IEEE/ANSI designation for an open-collector/drain output. It is an underlined diamond. Although we will not normally use the complete IEEE/ANSI symbology in this book, we will use this underlined diamond to indicate open-collector and open-drain outputs.

**FIGURE 8-33** IEEE/ANSI notation for open-collector and open-drain outputs.



### OUTCOME ASSESSMENT QUESTIONS

1. When does a HIGH/LOW conflict occur?
2. Why shouldn't totem-pole outputs be tied together?
3. How do open-collector outputs differ from totem-pole outputs?
4. Why do open-collector outputs need a pull-up resistor?
5. What is the logic expression for the wired-AND connection of six 7406 outputs?
6. Why are open-collector outputs generally slower than totem-pole outputs?
7. What is the IEEE/ANSI symbol for open-collector outputs?

## 8-12 TRISTATE (THREE-STATE) LOGIC OUTPUTS

### OUTCOMES

Upon completion of this section, you will be able to:

- Define tristate logic in terms of internal circuitry.
- Apply tristate output circuits to design needs.
- Identify circuits that have tristate logic outputs.

The **tristate** configuration is a third type of output circuitry used in TTL and CMOS families. It takes advantage of the high-speed operation of the pull-up/pull-down output arrangement, while allowing outputs to be connected together to share a common wire. It is called tristate because it allows three possible output states: HIGH, LOW, and high-impedance (Hi-Z). The Hi-Z state is a condition in which both the pull-up and the pull-down transistors are turned OFF so that the output terminal is a high impedance to both ground and the power supply +V. Figure 8-34 illustrates these three states for a simple inverter circuit.

Devices with tristate outputs have an *enable* input. It is often labeled *E* for enable or *OE* for output enable. When  $OE = 1$ , as shown in Figure 8-34(a) and (b), the circuit operates as a normal INVERTER because the HIGH logic level at *OE* enables the output. The output will be either HIGH or LOW, depending on the input level. When  $OE = 0$ , as shown in Figure 8-34(c), the circuit's output is *disabled*. It goes into its Hi-Z state with both transistors in the nonconducting state. In this state, the output terminal is essentially an open circuit (not connected to anything).

### Advantage of Tristate

The outputs of tristate ICs can be connected together (share the use of a common wire) without sacrificing switching speed because a tristate output, when enabled, operates as a totem pole for TTL or an active pull-up/pull-down

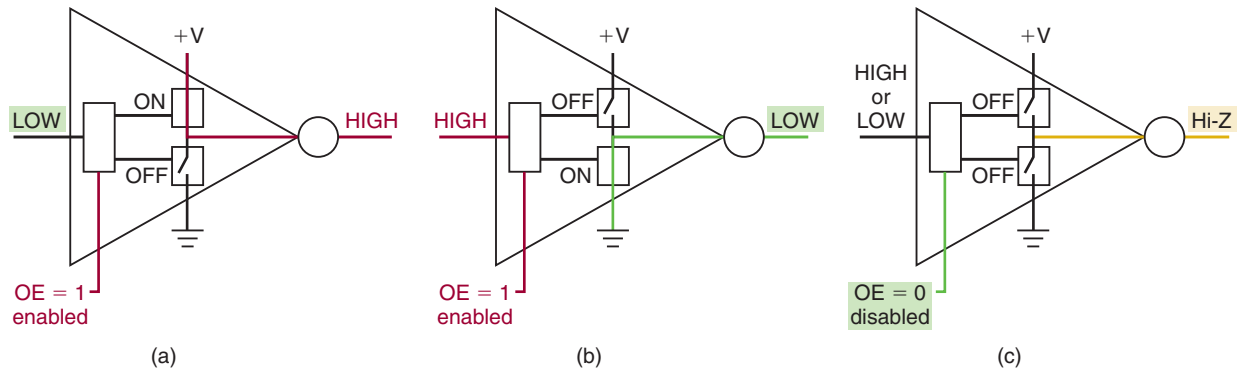


FIGURE 8-34 Three output conditions of tristate.

CMOS output with its associated low-impedance, high-speed characteristics. It is important to realize, however, that when tristate outputs are connected together, only one of them should be enabled at one time. Otherwise, two active outputs could fight for control of the common wire, as we discussed earlier, causing damaging currents to flow and producing invalid logic levels.

In our discussion of open-collector/open-drain and tristate circuits, we have referred to cases when the outputs of several devices must share a single wire to transmit information to another device. The shared wire is referred to as a bus wire. An entire bus is made up of several wires that are used to carry digital information between two or more devices that share the use of the bus.

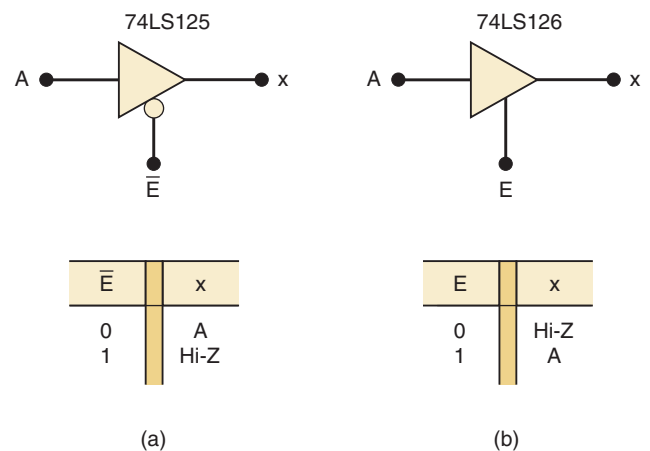
### Tristate Buffers

A *tristate buffer* is a circuit used to control the passage of a logic signal from input to output. Some tristate buffers also invert the signal as it goes through. The circuits in Figure 8-34 can be called *inverting tristate buffers*.

Two commonly used tristate buffer ICs are the 74LS125 and the 74LS126. Both contain four *noninverting* tristate buffers like those shown in Figure 8-35. The 74LS125 and 74LS126 differ only in the active state of their ENABLE inputs. The 74LS125 allows the input signal  $A$  to reach the output when  $\bar{E} = 0$ , while the 74LS126 passes the input when  $E = 1$ .

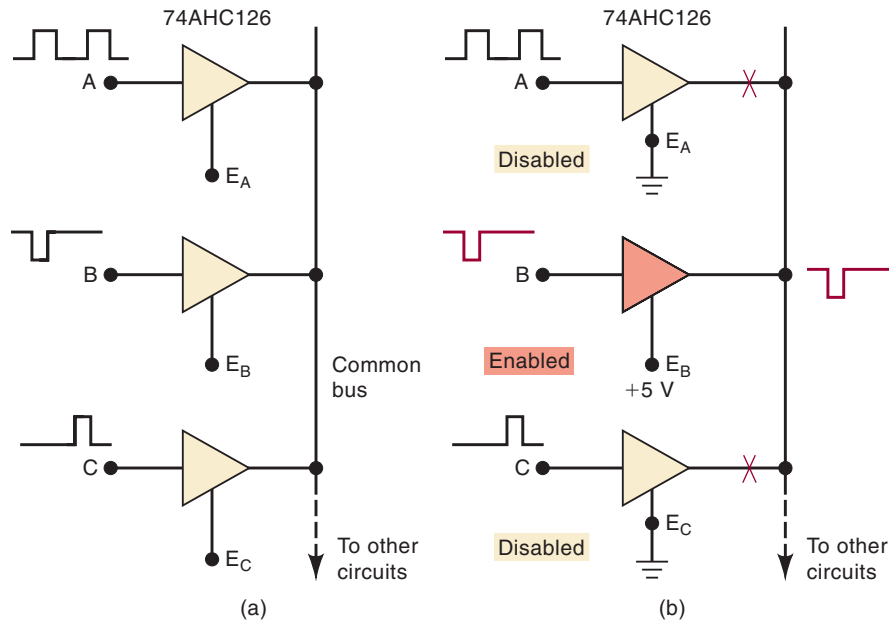
Tristate buffers have many applications in circuits where several signals are connected to common lines (buses). We will examine some of these

FIGURE 8-35 Tristate noninverting buffers.





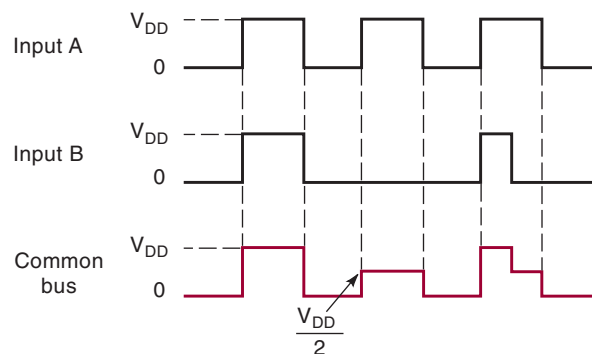
**FIGURE 8-36** (a) Tristate buffers used to connect several signals to a common bus; (b) conditions for transmitting  $B$  to the bus.



applications in Chapter 9, but we can get the basic idea from Figure 8-36(a). Here, we have three logic signals  $A$ ,  $B$ , and  $C$  connected to a common bus line through 74AHC126 tristate buffers. This arrangement permits us to transmit any one of these signals over the bus line to other circuits by enabling the appropriate buffer.

For example, consider the situation in Figure 8-36(b), where  $E_B = 1$  and  $E_A = E_C = 0$ . This disables the upper and lower buffers so that their outputs are in the Hi-Z state and are essentially disconnected from the bus. This is symbolized by the X's on the diagram. The middle buffer is enabled so that its input,  $B$ , is passed through to its output and onto the bus, from which it is routed to other circuits connected to the bus. When tristate outputs are connected together as in Figure 8-36, it is important to remember that no more than one output should be enabled at one time. Otherwise, two or more active totem-pole outputs would be connected, which could produce damaging currents. Even if damage did not occur, this situation would produce a signal on the bus that is a combination of more than one signal. This is commonly referred to as **bus contention**. Figure 8-37 shows the effect of enabling outputs  $A$  and  $B$  simultaneously. In Figure 8-36, when inputs  $A$  and  $B$  are in opposite states, they contend for control of the bus. The resulting voltage on the bus is an invalid logic state. In tristate bus systems, the designer must make sure that the enable signals do not allow bus contention to occur.

**FIGURE 8-37** If two enabled CMOS outputs are connected together, the bus will be at approximately  $V_{DD}/2$  when the outputs are trying to be different.



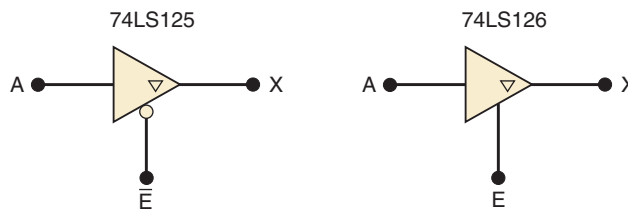
## Tristate ICs

In addition to tristate buffers, many ICs are designed with tristate outputs. For example, the 74LS374 is an octal D-type FF register IC with tristate outputs. This means that it is an eight-bit register made up of D-type FFs whose outputs are connected to tristate buffers. This type of register can be connected to common bus lines along with the outputs from other, similar devices to allow efficient transfer of data over the bus. We examine this *tristate data bus* arrangement in Chapter 9. Other types of logic devices that are available with tristate outputs include decoders, multiplexers, analog-to-digital converters, memory chips, and microprocessors.

## IEEE/ANSI Symbol for Tristate Outputs

The traditional logic symbology has no special notation for tristate outputs. Figure 8-38 shows the notation used in the IEEE/ANSI symbology to indicate a tristate output. It is a triangle that points downward. Although it is not part of the traditional symbology, we will use this triangle to designate tristate outputs throughout the remainder of the book.

**FIGURE 8-38** IEEE/ANSI notation for tristate outputs.



### OUTCOME ASSESSMENT QUESTIONS

1. What are the three possible states of a tristate output?
2. What is the state of a tristate output when it is disabled?
3. What is bus contention?
4. What conditions are necessary to transmit signal *C* onto the bus in Figure 8-37?
5. What is the IEEE/ANSI designation for tristate outputs?

## 8-13 HIGH-SPEED BUS INTERFACE LOGIC

### OUTCOMES

Upon completion of this section, you will be able to:

- Relate analog circuit issues (R, L, C) to logic circuits at very high frequencies.
- Identify technology that can operate at very high frequencies.

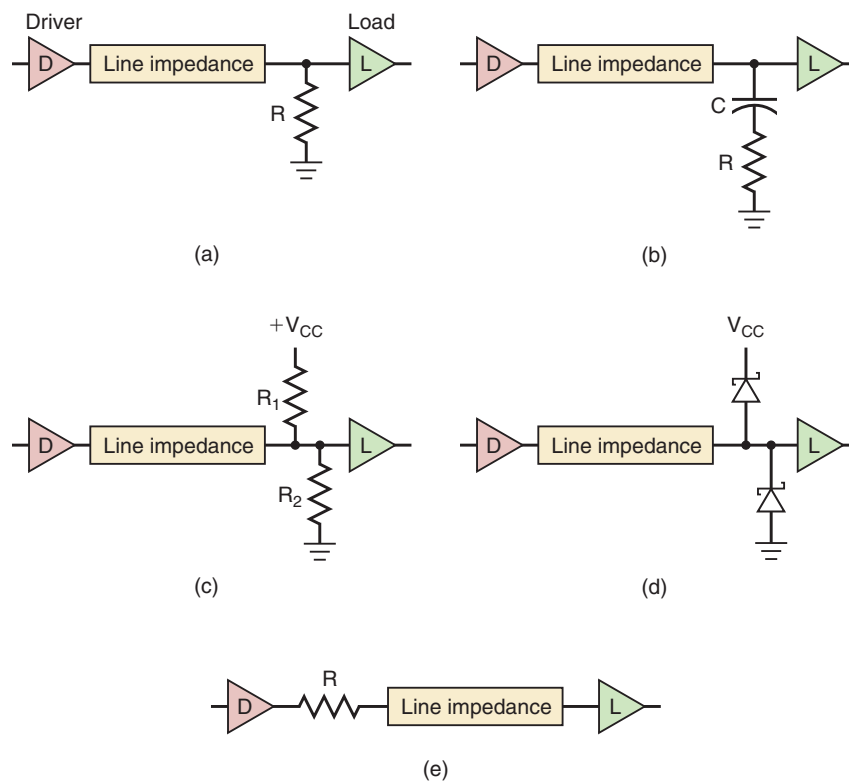
Many digital systems use a shared bus to transfer digital signals and data between the various components of the system. As you can see from our discussion of CMOS technology development, systems are getting faster and faster. Many of the newer high-speed logic series are designed specifically to interface to a tristate bus system. The components in these series are primarily tristate buffers, bidirectional transceivers, latches, and high-current line drivers.

A significant distance often physically separates the components in these systems. If this distance is more than about 4 inches, the bus wires between them need to be viewed as a transmission line. Although transmission line theory could fill up a whole book and is beyond the scope of this text, the general idea is simple enough. Wires have inductance ( $L$ ), capacitance ( $C$ ), and resistance ( $R$ ), which means that for changing signals (AC), they have a characteristic impedance that can affect a signal placed on one end and distort it by the time it reaches the other end. At the high speeds we are discussing, the travel time down the wire and the effects of reflected waves (like echoes) and ringing become real concerns. There are several ways to combat the problems associated with transmission lines. In order to prevent reflected pulse waves, the end of the bus must be terminated with a resistance that is equal to the line impedance (about  $50\ \Omega$ ), as shown in Figure 8-39(a). This method is not feasible because too much current is required to maintain logic level voltages across such a low resistance. Another technique uses a capacitor to block the DC current when the line is not changing, but effectively appears as just a resistor to the rising or falling pulse. This method is shown in Figure 8-39(b).

Using a voltage divider, as in Figure 8-39(c), with resistances larger than the line, impedance helps reduce reflections, but with hundreds of individual bus lines, it obviously makes a heavy load on the system power supply. The diode termination shown in Figure 8-39(d) simply clips off or clamps the overshoot/undershoot of the ringing caused by the reactive  $LC$  nature of the line. Series termination at the source, as shown in Figure 8-39(e), slows down the switching speed, which reduces the frequency limits of the bus but substantially improves the reliability of the bus signals.

As you can see, none of these methods are ideal. IC manufacturers are designing new series of logic circuits that overcome many of these problems. Texas Instruments' bus interface logic series offers new output circuits that

**FIGURE 8-39** Bus termination techniques.



dynamically lower the output impedance during signal transition to provide fast transition times, then raises the impedance during the steady state (like a series termination) to damp any ringing and reduce reflections on the bus line. The GTLP (Gunning Transceiver Logic Plus) series of bus interface devices is specially designed to drive the relatively long buses that connect modules of a large digital system. The backplane refers to the board (or cabling system) that the modules of a system plug into. All of the interconnecting signals travel between modules on conducting paths on the backplane.

Another major player in the high-speed bus-interface arena is known as **low-voltage differential signaling (LVDS)**. It uses two wires for each signal, and differential signaling means it responds to the difference between the two wires. Unwanted noise signals are usually present on both lines, and have no effect on the difference between the two. To represent the two logic states, LVDS uses a low voltage swing but switches polarity to clearly distinguish a 1 from a 0.

#### OUTCOME ASSESSMENT QUESTIONS

1. How close together do components need to be to ignore “transmission line” effects?
2. What three characteristics of real wires add up to distort signals that move through them?
3. What is the purpose of bus terminations?

## 8-14 CMOS TRANSMISSION GATE (BILATERAL SWITCH)

### OUTCOMES

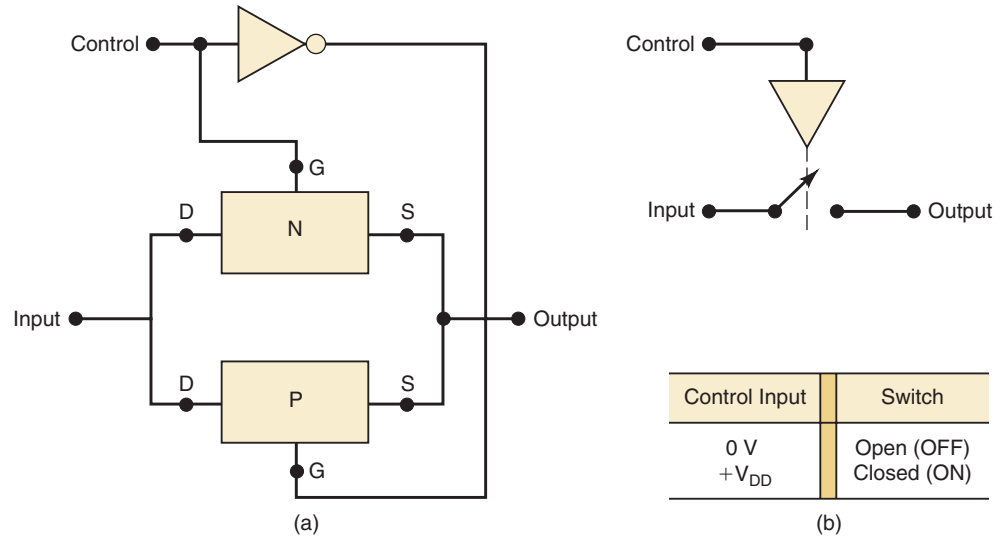
*Upon completion of this section, you will be able to:*

- Recognize and apply a CMOS bilateral switch.
- Explain the operation of a CMOS bilateral switch.

A special CMOS circuit that has no TTL or ECL counterpart is the **transmission gate** or **bilateral switch**, which acts essentially as a single-pole, single-throw switch controlled by an input logic level. This transmission gate passes signals in both directions and is useful for digital and analog applications.

Figure 8-40(a) is the basic arrangement for the bilateral switch. It consists of a P-MOSFET and an N-MOSFET in parallel so that both polarities of input voltage can be switched. The CONTROL input and its inverse are used to turn the switch on (closed) and off (open). When the CONTROL is HIGH, both MOSFETs are turned on and the switch is closed. When CONTROL is LOW, both MOSFETs are turned off and the switch is open. Ideally, this circuit operates like an electromechanical relay. In practice, however, it is not a perfect short circuit when the switch is closed; the switch resistance  $R_{ON}$  is typically  $200\ \Omega$ . In the open state, the switch resistance is very large, typically  $10^{12}\ \Omega$ , which for most purposes is an open circuit. The symbol in Figure 8-40(b) is used to represent the bilateral switch.

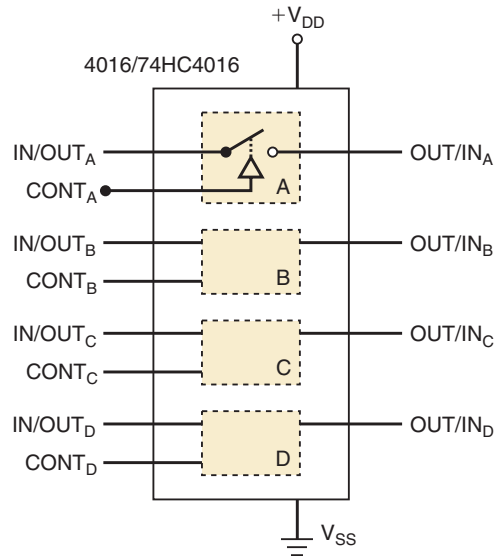
This circuit is called a *bilateral* switch because the input and output terminals can be interchanged. The signals applied to the switch input can be either digital or analog signals, provided that they stay within the limits of 0 to  $V_{DD}$  volts.



**FIGURE 8-40** CMOS bilateral switch (transmission gate).

Figure 8-41(a) shows the traditional logic diagram for a 4016 quad bilateral switch IC, which is also available in the 74HC series as a 74HC4016. The IC contains four bilateral switches that operate as described above. Each switch is independently controlled by its own control input. For example, the ON/OFF status of the top switch is controlled by input CONT<sub>A</sub>. Because the switches are bidirectional, either switch terminal can serve as input or output, as the labeling indicates.

**FIGURE 8-41** The 4016/74HC4016 quad bilateral switch.



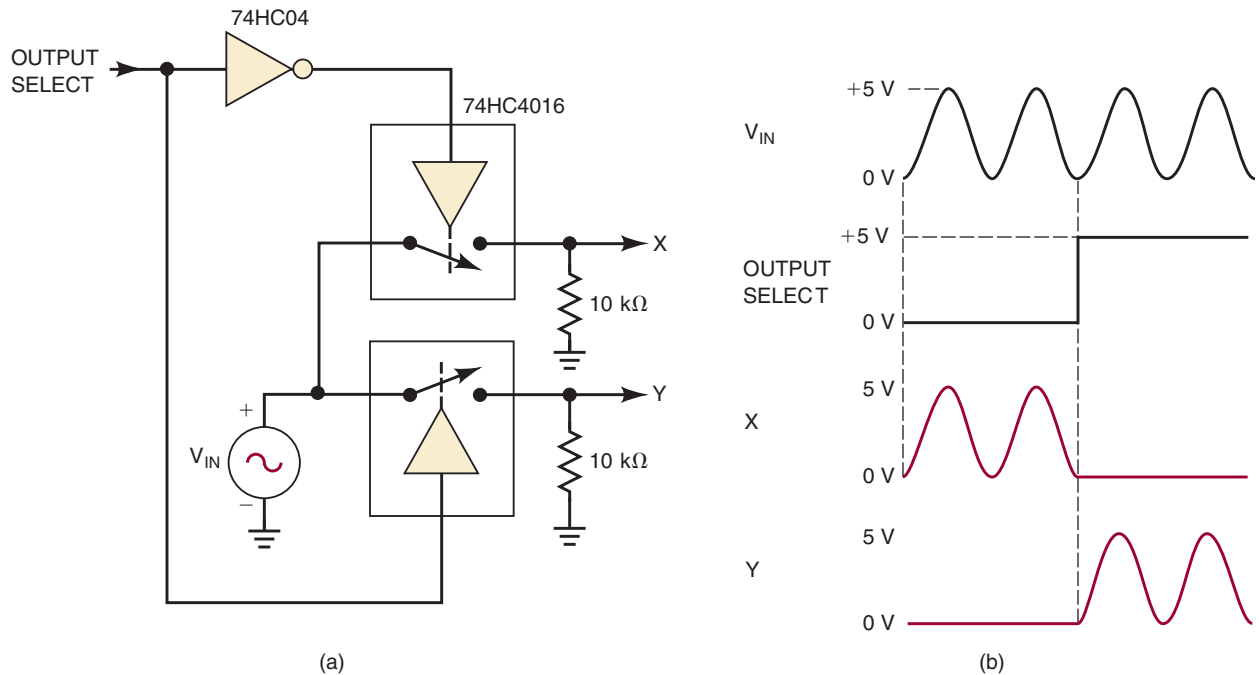
**EXAMPLE 8-12**

Describe the operation of the circuit in Figure 8-42.

**Solution**

Here, two of the bilateral switches are connected so that a common analog input signal can be switched to either output X or output Y, depending on the logic state of the OUTPUT SELECT input. When OUTPUT SELECT is LOW, the upper switch is closed and the lower one is open so that V<sub>IN</sub> is

connected to output X. When OUTPUT SELECT is HIGH, the upper switch is open and the lower one is closed so that  $V_{IN}$  is connected to output Y. Figure 8-42(b) shows some typical waveforms. Note that for proper operation,  $V_{IN}$  must be within the range 0 V to  $+V_{DD}$ .



**FIGURE 8-42** Example 8-12: 74HC4016 bilateral switches used to switch an analog signal to two different outputs.

The 4016/74HC4016 bilateral switch can switch only input voltages that lie between 0 V and  $V_{DD}$ , and so it could not be used for signals that were both positive and negative relative to ground. The 4316 and 74HC4316 ICs are quad bilateral switches that can switch *bipolar* analog signals. These devices have a second power-supply terminal called  $V_{EE}$ , which can be made negative with respect to ground. This permits input signals that can range from  $V_{EE}$  to  $V_{DD}$ . For example, with  $V_{EE} = -5$  V and  $V_{DD} = +5$  V, the analog input signal can be anywhere from  $-5$  V to  $+5$  V.

### OUTCOME ASSESSMENT QUESTIONS

1. Describe the operation of a CMOS bilateral switch.
2. *True or false:* There is no TTL bilateral switch.

## 8-15 IC INTERFACING

### OUTCOME

Upon completion of this chapter, you will be able to:

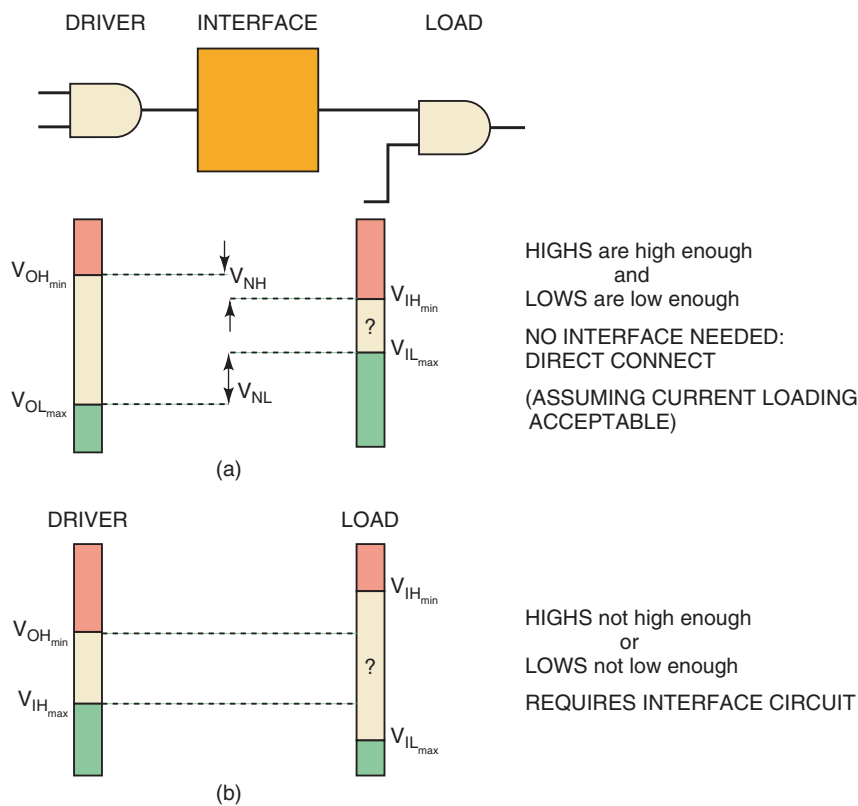
- Identify and overcome the challenges of interfacing various logic families with their diverse characteristics.

**Interfacing** means connecting the output(s) of one circuit or system to the input(s) of another circuit or system that has different electrical

characteristics. Often a direct connection cannot be made because of the difference in the electrical characteristics of the *driver* circuit that is providing the output signal and the *load* circuit that is receiving the signal. An interface circuit is connected between the driver and the load as shown in Figure 8-43. Its function is to take the driver output signal and condition it so that it is compatible with the requirements of the load. In digital systems this is pretty simple, because each device is either on or off. The interface must ensure that when the driver outputs a HIGH, the load receives a signal it perceives to be HIGH; *and*, when the driver outputs a LOW, the load receives a signal it perceives to be LOW.

The simplest and most desirable interface circuit between a driver and a load is a direct connection. Of course, devices that are in the same series are designed to interface directly with each other. Today, however, many systems involve mixed families, mixed voltages, and mixed series. In these systems the challenge is to make sure that the driver is able to consistently activate the load in both the LOW and HIGH states.

**FIGURE 8-43** Interfacing logic ICs: (a) no interface needed; (b) requires interface.



For any case such as shown in Figure 8-43(a), where  $V_{OH}$  of the driver is *enough greater* than the  $V_{OH}(\min)$  of the load and  $V_{OL}$  of the driver is *enough less* than the  $V_{IL}(\max)$  of the load, there is no need for an interface circuit other than a direct connection. “How much *greater*?” and “How much *less*?” are questions related to how much noise is expected in the system. Recall that the noise margins ( $V_{NH}$  and  $V_{NL}$ ) are measures of this difference between output and input characteristics. (Refer back to Figure 8-4.) The minimum acceptable noise margin for any system is a judgment call that must be made by the system designer. Whenever the  $V_{NH}$  or  $V_{NL}$  is determined to be too small (or even negative), an interface circuit is necessary in order to ensure that the driver and load can work together. This situation is depicted in Figure 8-43(b). To summarize, logic devices will be

voltage-compatible, and no interface will be necessary under the following circumstances:

<b>Driver</b>	<b>Load</b>
$V_{OH}(\min)$	$> V_{IH}(\min) + V_{NH}$
$V_{OL}(\max) + V_{NL}$	$< V_{IL}(\max)$

We should also note that, especially when using older families, the current (as opposed to voltage) characteristics of the driver and load must also match. The  $I_{OH}$  of the driver must be able to source enough current to supply the necessary  $I_{IH}$  of the load, and the  $I_{OL}$  of the driver must be able to sink enough current to accommodate the  $I_{IL}$  from the load. This topic was covered in Section 8-5 when we discussed fan-out. Most modern logic devices have high enough output drive and low enough input current to make loading a rare problem. However, this is very important when interfacing to external input/output devices such as motors, lights, or heaters. To summarize current loading requirements:

<b>Driver</b>	<b>Load</b>
$I_{OH}(\max)$	$> I_{IH}(\text{total})$
$I_{OL}(\max)$	$> I_{IL}(\text{total})$

Table 8-11 lists some nominal values for a number of different families and series of digital devices. Within each family there will be exceptions to these listed values, and so in practice it is important that you look up the data sheet values for the specific ICs you are working with. For the sake of convenience we will use these values in the examples that follow.

**TABLE 8-11** Input/output currents for standard devices with a supply voltage of 5 V.

Parameter	CMOS				TTL				
	4000B	74HC/HCT	74AC/ACT	74AHC/AHCT	74	74LS	74AS	74ALS	74F
$I_{IH}(\max)$	1 $\mu\text{A}$	1 $\mu\text{A}$	1 $\mu\text{A}$	1 $\mu\text{A}$	40 A	20 $\mu\text{A}$	20 $\mu\text{A}$	20 $\mu\text{A}$	20 $\mu\text{A}$
$I_{IL}(\max)$	1 $\mu\text{A}$	1 $\mu\text{A}$	1 $\mu\text{A}$	1 $\mu\text{A}$	1.6 mA	0.4 mA	0.5 mA	100 $\mu\text{A}$	0.6 mA
$I_{OH}(\max)$	0.4 mA	4 mA	24 mA	8 mA	0.4 mA	0.4 mA	2 mA	400 mA	1.0 mA
$I_{OL}(\max)$	0.4 mA	4 mA	24 mA	8 mA	16 mA	8 mA	20 mA	8 mA	20 mA

### Interfacing 5-V TTL and CMOS

When interfacing different types of ICs, we must check that the driving device can meet the current and voltage requirements of the load device. Examination of Table 8-11 indicates that the input current values for CMOS are extremely low compared to the output current capabilities of any TTL series. Thus, TTL has no problem meeting the CMOS input current requirements.

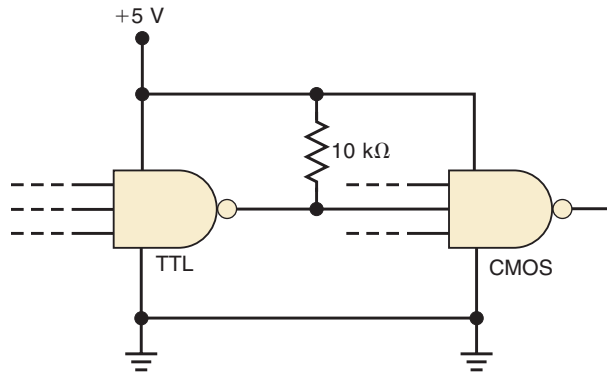
There is a problem, however, when we compare the TTL output voltages with the CMOS input voltage requirements. Table 8-9 showed that  $V_{OH}(\min)$  of every TTL series is too low when compared with the  $V_{IH}(\min)$  requirement of the 4000B, 74HC, and 74AC series. For these situations, something must be done to raise the TTL output voltage to an acceptable level for CMOS.

The most common solution to this interface problem is shown in Figure 8-44, where the TTL output is connected to +5V with a pull-up resistor. The presence of the pull-up resistor causes the TTL output to rise to approximately 5 V in the HIGH state, thereby providing an adequate CMOS input voltage level. This pull-up resistor is not required if the



CMOS device is a 74HCT or a 74ACT because these series are designed to accept TTL outputs directly, as Table 8-9 shows.

**FIGURE 8-44** External pull-up resistor is used when TTL drives CMOS.

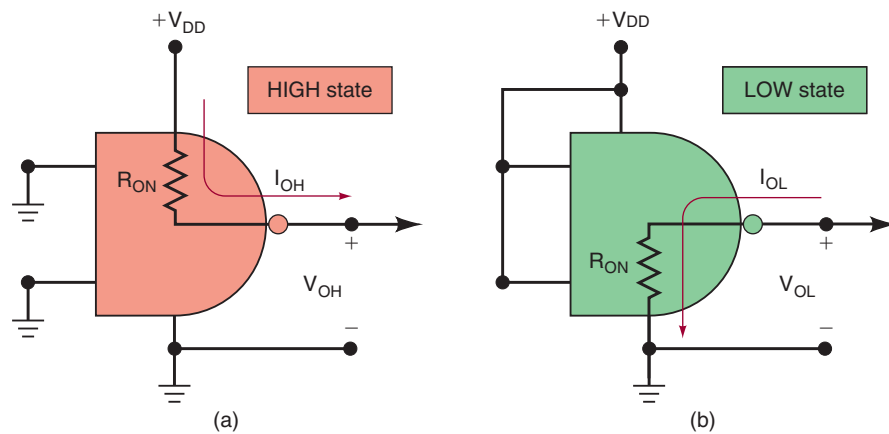


### CMOS Driving TTL

Before we consider the problem of interfacing CMOS outputs to TTL inputs, it will be helpful to review the CMOS output characteristics for the two logic states. Figure 8-45(a) shows the equivalent output circuit in the HIGH state. The  $R_{ON}$  of the P-MOSFET connects the output terminal to  $V_{DD}$  (remember, the N-MOSFET is off). Thus, the CMOS output circuit acts like a  $V_{DD}$  source with a source resistance of  $R_{ON}$ . The value of  $R_{ON}$  typically ranges from 100 to 1000  $\Omega$ .

Figure 8-45(b) shows the equivalent output circuit in the LOW state. The  $R_{ON}$  of the N-MOSFET connects the output terminal to ground (remember, the P-MOSFET is off). Thus, the CMOS output acts as a low resistance to ground; that is, it acts as a current sink.

**FIGURE 8-45** Equivalent CMOS output circuits for both logic states.



### CMOS Driving TTL in the HIGH State

Table 8-9 showed that CMOS outputs can easily supply enough voltage ( $V_{OH}$ ) to satisfy the TTL input requirement in the HIGH state ( $V_{IH}$ ). Table 8-11 shows that CMOS outputs can supply more than enough current ( $I_{OH}$ ) to meet the TTL input current requirements ( $I_{IH}$ ). Thus, no special consideration is needed for the HIGH state.

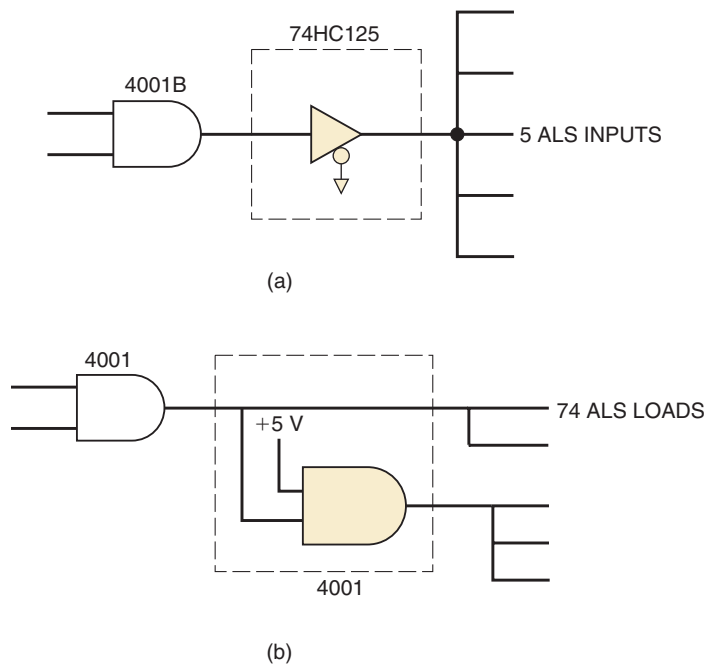
### CMOS Driving TTL in the LOW State

Table 8-11 shows that TTL inputs have a relatively high input current in the LOW state, ranging from 100  $\mu\text{A}$  to 1.6 mA. The 74HC and 74HCT series can sink up to 4 mA, and so they would have no trouble driving a

single TTL load of any series. The 4000B series, however, is much more limited. Its low  $I_{OL}$  capability is not sufficient to drive even one input of the 74 or 74AS series. The 74AHC series has output drive comparable to that of the 74LS series.

For the situation in which a driver cannot supply enough current to the load, the interface solution is to select a buffer that has input specifications that are compatible with the driver and enough output drive current to supply the load. Figure 8-46(a) shows an example of this situation. The 4001B's maximum output current is not enough to drive five ALS inputs. It is able to drive the 74HC125 input, which in turn can drive the other inputs. Another possible solution is shown in Figure 8-49(b), where the load is divided up among multiple 4001 series parts such that no output needs to drive more than three loads.

**FIGURE 8-46** (a) Using an HC series interface IC. (b) Using a similar gate to share the load.



### EXAMPLE 8-13

A 74HC output is driving three 7406 inputs. Is this a good design?

#### Solution

NO! The 74HC00 can sink 4 mA, but the 7406 input  $I_{IL}$  is 1.6 mA. Total load current when LOW is  $1.6 \text{ mA} \times 3 = 4.8 \text{ mA}$ . . . Too much load current.

### EXAMPLE 8-14

A 4001B output is driving three 74LS inputs. Is this a well-designed circuit?

#### Solution

NO! The 4001B can sink 0.4 mA, but each 74LS input accounts for  $0.4 \text{ mA} \times 3 = 1.2 \text{ mA}$ . . . Too much load current.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What must be done to interface a standard TTL output to a 74AC or a 74HC input? Assume  $V_{DD} = +5\text{ V}$ .
2. What is usually the problem with CMOS driving TTL?

## 8-16 MIXED-VOLTAGE INTERFACING

### OUTCOME

*Upon completion of this chapter, you will be able to:*

- Effectively interface logic devices that operate on different logic levels and power supplies.

As we discussed in Section 8-10, many new logic devices operate on less than 5 V. In many situations, these devices need to communicate with each other. In this section we will look specifically at how to interface logic devices that operate on different voltage standards.

### Low-Voltage Outputs Driving High-Voltage Loads

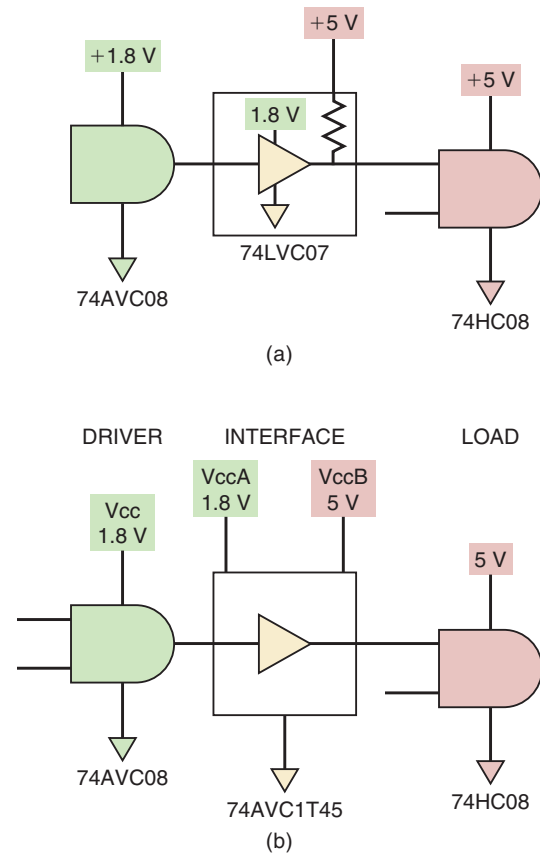
In some situations, the  $V_{OH}$  of the driver is just slightly lower than the load requires to recognize it as a HIGH. This situation was discussed when interfacing TTL outputs to 5-V CMOS inputs. The only interface component needed was a 10-k $\Omega$  pull-up resistor, which will cause the TTL output voltage to be boosted above the 3.3-V level when the output is HIGH.

When there is a need for a more substantial shift in voltage because the driver and load operate on different power-supply voltages, a **voltage-level translator** interface circuit is required. An example of this is a low-voltage (1.8-V) CMOS device driving a 5-V CMOS input. The driver can put out a maximum of only 1.8 V as a HIGH, and the load gate requires 3.33 V for a HIGH. We need an interface that can accept 1.8-V logic levels and translate them to 5-V CMOS levels. The simplest way to accomplish this is with a buffer that has an open-drain, such as the 74LVC07 shown in Figure 8-47(a). Notice that the pull-up resistor is connected to the 5-V supply, while the power supply for the interface buffer is 1.8 V. Another solution is to utilize a dual-supply-level translator circuit such as the 74AVC1T45, as shown in Figure 8-47(b). This device uses two different power-supply voltages, one for the inputs and the other for the outputs, and translates between the two levels.

### High-Voltage Outputs Driving Low-Voltage Loads

When logic circuits that operate on a higher voltage supply must drive other logic circuits that operate on a lower voltage supply, the output voltage of the driver often exceeds the safe limits that the load gate can handle. In these situations, a dual-supply-voltage-level translator can be used just as it was in Figure 8-47(b). Another common solution to this problem is to interface them using a buffer from a series that can withstand the higher input voltage. Figure 8-48 demonstrates this with a 5-V CMOS part driving a 1.8-V AUC series input. The highest voltage the AUC input (load gate) can handle is 3.6 V. However, a 74LVC07A can handle up to 5.5 V on its input without

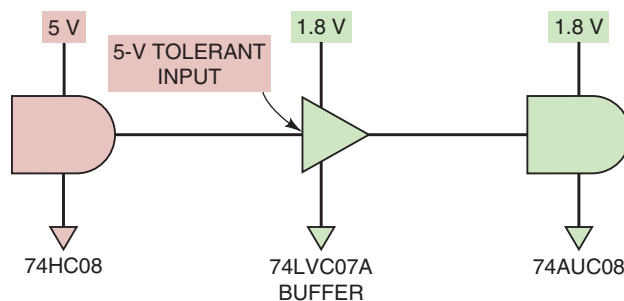
**FIGURE 8-47** (a) Using an open-drain with pullup to high voltage. (b) Using a voltage-level translator.



damaging it, even though it is operating on 1.8 V. Figure 8-48 shows how we can use the higher voltage tolerance of the 74LVC07A to translate a 5-V logic level down to a 1.8-V logic level.

At this point you may be wondering, “Why in the world would anyone choose to use such an assortment of incompatible parts?” The answer lies in considering larger systems and trying to balance performance and cost. In a computer system, for example, you may have a 2.5-V CPU, a 3.3-V memory module, and a 5-V hard drive controller all working on the same mother board. The low-voltage components may be necessary to obtain the desired performance, but the 5-V hard drive may be the least expensive or the only type available. The driver and load devices may not be standard logic gates at all, but may be a VLSI component in our system. Using the data sheet for those devices, we must look up the output characteristics and interface them using the techniques we have shown. As logic standards continue to evolve, it is important that we can make systems work using any of the diverse components available to us.

**FIGURE 8-48** (a) A low-voltage series with 5-V tolerant inputs as an interface.



**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is the function of an *interface* circuit?
2. *True or false:* All CMOS outputs can drive TTL in the HIGH state.
3. *True or false:* Any CMOS output can drive any single TTL input.
4. Which CMOS series can TTL drive without a pull-up resistor?
5. How many 7400 inputs can be driven from a 74HCT00 output?

## 8-17 ANALOG VOLTAGE COMPARATORS

### OUTCOMES

Upon completion of this section, you will be able to:

- Define the operation of an analog voltage comparator.
- Interface an analog voltage comparator to any logic family/series input.

Another very useful device for interfacing to digital systems is the **analog voltage comparator**. It is especially useful in systems that contain both analog voltages as well as digital components. An analog voltage comparator compares two voltages. If the voltage on the (+) input is greater than the voltage on the (−) input, the output is HIGH. If the voltage on the (−) input is greater than the voltage on the (+) input, the output is LOW. The inputs to a comparator can be thought of as analog inputs, but the output is digital because it will always be either HIGH or LOW. For this reason, the comparator is often referred to as a one-bit analog-to-digital (A/D) converter. We will examine A/D converters in detail in Chapter 11.

An LM339 is an analog linear IC that contains four voltage comparators. The output of each comparator is an open-collector transistor just like an open-collector TTL output.  $V_{CC}$  can range from 2 to 36 V but is usually set slightly greater than the analog input voltages that are being compared. A pull-up resistor must be connected from the output to the same supply that the digital circuits use (normally 5 V).

### EXAMPLE 8-15

Suppose that an incubator must have an emergency alarm to warn if the temperature exceeds a dangerous level. The temperature-measuring device is an LM34 that puts out a voltage directly proportional to the temperature. The output voltage goes up 10 mV per degree F. The digital system alarm must sound when the temperature exceeds 100°F. Design a circuit to interface the temperature sensor to the digital circuit.

#### Solution

We need to compare the voltage from the sensor with a fixed threshold voltage. First, we must calculate the proper threshold voltage. We want the comparator output to go HIGH when the temperature exceeds 100°F. The voltage out of the LM34 at 100°F will be

$$100^{\circ}\text{F} \cdot 10 \text{ mV}/^{\circ}\text{F} = 1.0 \text{ V}$$

This means that we must set the (–) input pin of the comparator to 1.0 V and connect the LM34 to the (+) input. In order to create a 1.0-V reference voltage, we can use a voltage-divider circuit and choose a bias current of 100  $\mu\text{A}$ . The LM339 input current will be relatively negligible because it will draw less than 1  $\mu\text{A}$ . This means that  $R_1 + R_2$  must total 10 k $\Omega$ . In this example, we can operate everything from a +5 V power supply. Figure 8-49 shows the complete circuit. The calculations are as follows:

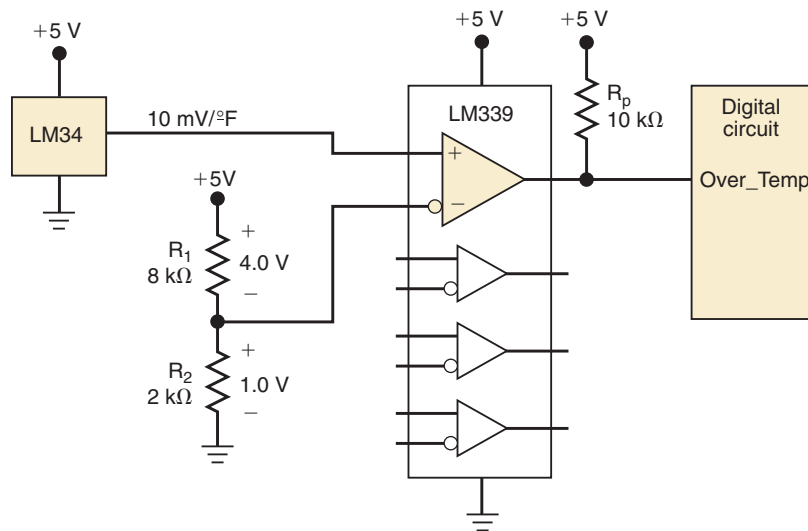
$$V_{R2} = V_{CC} \cdot \frac{R_2}{R_1 + R_2}$$

$$R_2 = V_{R2} \cdot (R_1 + R_2) / V_{CC}$$

$$= 1.0 \text{ V}(10 \text{ k}\Omega) / 5 \text{ V} = 2 \text{ k}\Omega$$

$$R_1 = 10 \text{ k}\Omega - R_2 = 10 \text{ k}\Omega - 2 \text{ k}\Omega = 8 \text{ k}\Omega$$

**FIGURE 8-49** A temperature-limit detector using an LM339 analog voltage comparator.



### OUTCOME ASSESSMENT QUESTIONS

1. What causes the output of a comparator to go to the HIGH logic state?
2. What causes the output of a comparator to go to the LOW logic state?
3. Is an LM339 output more similar to a TTL totem-pole or an open-collector output?

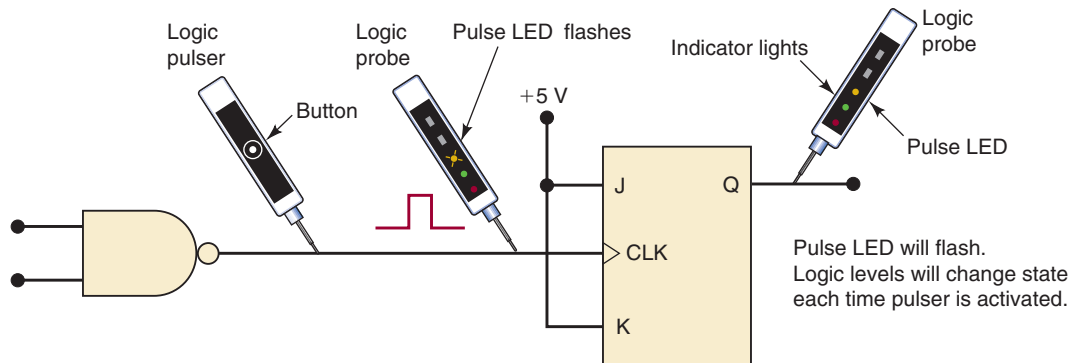
## 8-18 TROUBLESHOOTING

### OUTCOME

Upon completion of this chapter, you will be able to:

- Use a logic pulser and logic probe to find faults in a digital circuit.

A **logic pulser** is a testing and troubleshooting tool that generates a short-duration pulse when manually actuated, usually by pressing a push button. The logic pulser shown in Figure 8-50 has a needle-shaped tip that is touched to the circuit node that is to be pulsed. The logic pulser is designed



**FIGURE 8-50** A logic pulser can inject a pulse at any node that is not shorted directly to ground or  $V_{CC}$ .

so that it senses the existing voltage level at the node and produces a voltage pulse in the opposite direction. In other words, if the node is LOW, the logic pulser produces a narrow positive-going pulse; if the node is HIGH, it produces a narrow negative-going pulse.

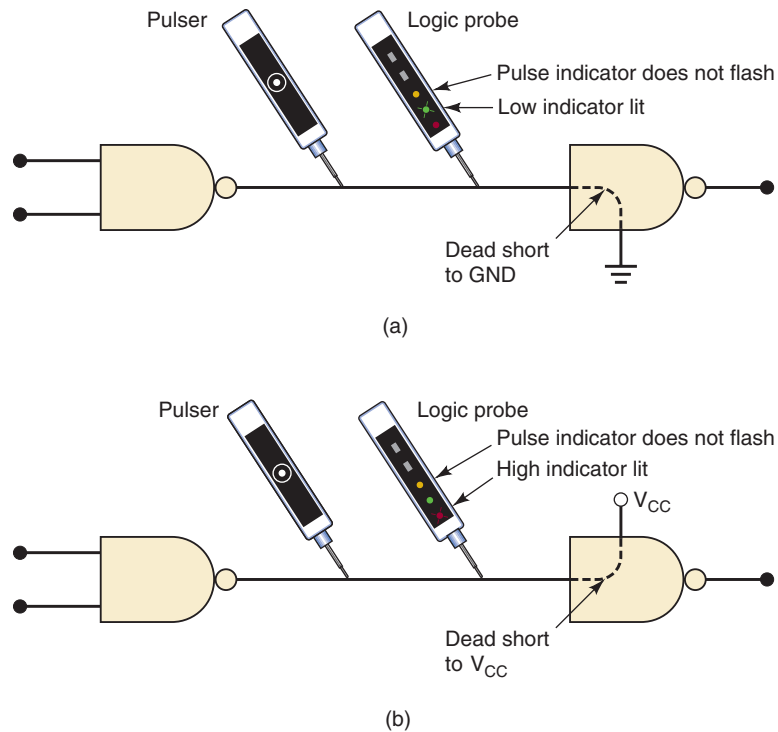
The logic pulser is used to change the logic level at a circuit node momentarily, even though the output of another device may be connected to that same node. In Figure 8-50, the logic pulser is contacting the output of the NAND gate. The logic pulser has a very low output impedance (typically  $2\ \Omega$  or less), so that it can overcome the NAND gate's output and can change the voltage at the node. The logic pulser, however, cannot produce a voltage pulse at a node that is shorted directly to ground or  $V_{CC}$  (e.g., through a solder bridge).

### Using a Logic Pulser and Probe to Test a Circuit

A logic pulser can be used to inject a pulse or a series of pulses manually into a circuit in order to test the circuit's response. A logic probe is almost always used to monitor the circuit's response to the logic pulser. In Figure 8-50, the J-K flip-flop's toggle operation is being tested by applying pulses from the logic pulser and monitoring  $Q$  with the logic probe. This logic pulser/logic probe combination is very useful for checking the operation of a logic device while it is wired into a circuit. Note that the logic pulser is applied to the circuit node *without* disconnecting the output of the NAND gate that is driving that node. When the probe is placed on the same node as the pulser, the logic level indicators appear to remain unchanged (LOW in this example), but the yellow pulse indicator flashes once each time the pulser button is pressed. When the probe is placed on the  $Q$  output, the pulse LED flashes once (indicating a transition), and the logic level indicators change state each time the pulser button is pushed.

### Finding Shorted Nodes

The logic pulser and logic probe can be used to check for nodes that are shorted directly to ground or  $V_{CC}$ , as shown in Figure 8-51. When you touch a logic pulser and a logic probe to the same node and press the logic pulser button, the logic probe should indicate the occurrence of a pulse at the node. If the probe indicates a constant LOW and the pulse LED does not flash, the node is shorted to ground, as shown in Figure 8-51(a). If the probe indicates a constant HIGH and the pulse LED does not flash, the node is shorted to  $V_{CC}$  as shown in Figure 8-51(b).



**FIGURE 8-51** A logic pulser and a logic probe can be used to trace shorted nodes.

### OUTCOME ASSESSMENT QUESTIONS

1. What is the function of a logic pulser?
2. *True or false:* A logic pulser will produce a voltage pulse at any node.
3. *True or false:* A logic pulser can force a node LOW or HIGH for extended periods of time.
4. How does a logic probe respond to the logic pulser?

## 8-19 CHARACTERISTICS OF AN FPGA

Use features available in FPGAs to interface to a wide variety of digital circuit technologies. We have shown in earlier chapters how to implement digital circuits using PLDs. These modern-day marvels of digital design flexibility use CMOS technology, but they can also provide a number of options in their electrical characteristics. Let's take a look at the electrical and timing characteristics for an example PLD family. We have selected the Altera Cyclone™ II, a popular family of devices for industry and education, for our discussion. These devices belong to a subcategory of PLD devices referred to as field programmable gate arrays (FPGAs). We will present more information about different PLD categories and architectures in Chapter 13.

### Power-Supply Voltage

Two different power-supply voltages must be applied to a Cyclone II chip. One voltage supply,  $V_{CCINT}$ , provides power for the internal logic of the chip. The nominal value is 1.2 V for  $V_{CCINT}$ . A separate supply voltage,  $V_{CCIO}$ , will be applied to power the input and output buffers of the Cyclone chips. The value of  $V_{CCIO}$  will be dependent upon the desired output logic voltage



levels. These devices will operate at 3.3-V, 2.5-V, 1.8-V, or 1.5-V levels by applying the corresponding supply voltage to  $V_{CCIO}$ .

### Logic Voltage Levels

While in the past there were usually only one or two standard interfaces for I/O within a system, today it is common to see three or more standard interfaces. This trend is driven by several factors including backward-compatible designs, complexity and size of newer systems, and different requirements for different subsystems within the system. Single-ended interfaces are very common for I/O signals that switch at less than 300 MHz. Single-ended signals, requiring only a single trace on the printed circuit board and having voltage levels that are measured with respect to a common ground, are inexpensive and very easy to use. On the other hand, differential interfaces, which use two signal pathways forming a current loop with current flow in one direction or the other, operate with lower voltage signals and can switch at higher frequencies. Differential signaling also exhibits common-mode noise rejection, which offers better noise immunity for circuit designs. The major disadvantage of using differential interfaces is the additional cost of needing two chip pins and two matched traces on the printed circuit board for each signal.

Cyclone devices support a variety of input/output standards that give designers the needed flexibility in designing their digital systems. Some of the common single-ended, general-purpose I/O standards, along with their input and output voltage parameters, are listed in Table 8-12. Cyclone devices can also be programmed for several other single-ended interface options. In addition, the Cyclone family supports a number of differential I/O standards that can provide improved noise immunity, lower electromagnetic interference (EMI) generation, and reduced power consumption.

**TABLE 8-12** Altera Cyclone II characteristics using general-purpose I/O standards.

Parameter	I/O Standard				
	3.3-V LVTTTL	3.3-V LVCMOS	2.5-V LVTTTL & LVCMOS	1.8-V LVTTTL & LVCMOS	1.5-V LVTTTL & LVCMOS
$V_{IL}(\text{max})$ (V)	0.8	0.8	0.7	$0.35 \times V_{CCIO}$	$0.35 \times V_{CCIO}$
$V_{IH}(\text{min})$ (V)	1.7	1.7	1.7	$0.65 \times V_{CCIO}$	$0.65 \times V_{CCIO}$
$V_{OL}(\text{max})$ (V)	0.45	0.2	0.4	0.45	$0.25 \times V_{CCIO}$
$V_{OH}(\text{min})$ (V)	2.4	$V_{CCIO} - 0.2$	2.0	$V_{CCIO} - 0.45$	$0.75 \times V_{CCIO}$

### Power Dissipation

The Cyclone II devices use CMOS technology and, therefore, the chip's power consumption will be low. Like other CMOS devices, the amount of power will be dependent upon the voltage level, frequencies, and loads for the I/O signals. A Cyclone II device can be configured for an infinite number of different circuit designs, so it is not possible to simply state the amount of power dissipation for any Cyclone device.

The Quartus II software has two tools to estimate the amount the power usage for an application. The PowerPlay Early Power Estimator is typically used during the early stages of design to get a magnitude estimate of the

device power. The PowerPlay Power Analyzer is used later in the design process, often with sample test vectors, to get a more accurate power consumption estimate. In both cases, these will be power estimates, but it does give the designer an idea of the amount of power consumption for the target FPGA device.

## Maximum Input Voltage and Output Current Ratings

The maximum DC input signal voltage is 4.6 V. Each output pin on a Cyclone II device can sink up to 40 mA or source up to 25 mA.

## Switching Speed

Cyclone II chips are available in three different speed grades, –6 (dash six), –7, and –8, with –6 being the fastest version. The speed of an application will be dependent upon the application and how it is implemented in the programmable device. Table 8-13 compares the maximum clocking frequency for LPM-implementations of 16-bit and 64-bit binary counters when using each of the three speed grades for Cyclone II devices.

**TABLE 8-13** Altera Cyclone II comparison of counter performance.

Application	–6 Speed grade	–7 Speed grade	–8 Speed grade
16-bit counter	401.6 MHz	349.4 MHz	310.65 MHz
64-bit counter	157.15 MHz	137.98 MHz	126.27 MHz

## SUMMARY

1. All digital logic devices are similar in nature but very much different regarding the details of their characteristics. An understanding of the terms used to describe these characteristics is important and allows us to compare and contrast the performance of devices. By understanding the capabilities and limitations of each type of device, we can intelligently combine devices to take advantage of each device's strengths in building reliable digital systems.
2. The TTL family of logic devices has been in use for over 45 years. The circuitry uses bipolar transistors. This family offers many SSI logic gates, and MSI devices. Numerous series of similarly numbered devices have been developed because advances in technology have offered improved characteristics.
3. When you are connecting devices together, it is vital to know how many inputs a given output can drive without compromising reliability. This is referred to as *fan-out*.
4. Open-collector and open-drain outputs can be wired together to implement a wired-AND function. Tristate outputs can be wired together to allow numerous devices to share a common data path known as a *bus*. In such a case, only one device is allowed to assert a logic level on the bus (i.e., drive the bus) at any one time.
5. The *fastest* logic devices are from a family that uses emitter-coupled logic (ECL). This technology also uses bipolar transistors but is not as widely used as TTL due to inconvenient input/output characteristics.

6. MOSFET transistors can also be used to implement logic functions. The main advantage of MOS logic is lower power and greater packing density.
7. The use of complementary MOSFETs has produced CMOS logic families. CMOS technology has captured the market due to its very low power and competitive speed.
8. The ongoing need to reduce power and size has led to several series of devices that operate on 3.3 V and 2.5 V.
9. Logic devices that use various technologies cannot always be directly connected together and operate reliably. The voltage and current characteristics of inputs and outputs must be considered and precautions taken to ensure proper operation.
10. CMOS technology allows a digital system to control analog switches called *transmission gates*. These devices can pass or block an analog signal, depending on the digital logic level that controls it.
11. Analog voltage comparators offer another bridge between analog signals and digital systems. These devices compare analog voltages and output a digital logic level based on which voltage is greater. They allow an analog system to control a digital system.
12. Many FPGAs use CMOS technology that supports a variety of input/output standards and are available in different speed grades.

## IMPORTANT TERMS

---

fan-out	current-sourcing transistor (pull-up transistor)	buffer/driver
noise immunity	floating inputs	tristate
noise margin	power-supply decoupling	bus contention
current sourcing	MOSFETs	low-voltage differential signaling (LVDS)
current sinking	N-MOS	transmission gate (bilateral switch)
DIP	P-MOS	interfacing
lead pitch	CMOS	voltage-level translator
surface-mount technology (SMT)	electrostatic discharge (ESD)	analog voltage comparator
TTL	latch-up	logic pulser
totem pole	open-collector output	
current-sinking transistor (pull-down transistor)	wired-AND	

## PROBLEMS

---

### SECTIONS 8-1 TO 8-3

- B** 8-1.\* Two different logic circuits have the characteristics shown in Table 8-14.
- (a) Which circuit has the best LOW-state DC noise immunity? The best HIGH-state DC noise immunity?
  - (b) Which circuit can operate at higher frequencies?
  - (c) Which circuit draws the most supply current?

---

\* Answers to problems marked with an asterisk can be found in the back of the text.

TABLE 8-14

	Circuit A	Circuit B
$V_{\text{supply}}$ (V)	6	5
$V_{\text{IH(min)}}$ (V)	1.6	1.8
$V_{\text{IL(max)}}$ (V)	0.9	0.7
$V_{\text{OH(min)}}$ (V)	2.2	2.5
$V_{\text{OL(max)}}$ (V)	0.4	0.3
$t_{\text{PLH}}$ (ns)	10	18
$t_{\text{PHL}}$ (ns)	8	14
$P_{\text{D}}$ (mW)	16	10

- B** 8-2. Look up the IC data sheets, and use *maximum* values to determine  $P_{\text{D(ave)}}$  and  $t_{\text{pd(ave)}}$  for one gate on each of the following TTL ICs. (See Example 8-2 in Section 8-3.)
- (a)\* 7432
  - (b)\* 74S32
  - (c) 74LS20
  - (d) 74ALS20
  - (e) 74AS20

8-3. A certain logic family has the following voltage parameters:

$$\begin{aligned} V_{\text{IH(min)}} &= 3.5 \text{ V} & V_{\text{IL(max)}} &= 1.0 \text{ V} \\ V_{\text{OH(min)}} &= 4.9 \text{ V} & V_{\text{OL(max)}} &= 0.1 \text{ V} \end{aligned}$$

- (a)\* What is the largest positive-going noise spike that can be tolerated?
- (b) What is the largest negative-going noise spike that can be tolerated?

### DRILL QUESTION

- B** 8-4.\* For each statement, indicate the term or parameter being described.
- (a) Current at an input when a logic 1 is applied to that input
  - (b) Current drawn from the  $V_{\text{CC}}$  source when all outputs are LOW
  - (c) Time required for an output to switch from the 1 to the 0 state
  - (d) The size of the voltage spike that can be tolerated on a HIGH input without causing indeterminate operation
  - (e) An IC package that does not require holes to be drilled in the printed circuit board
  - (f) When a LOW output receives current from the input of the circuit it is driving
  - (g) Number of different inputs that an output can safely drive
  - (h) Arrangement of output transistors in a standard TTL circuit
  - (i) Another term that describes pull-down transistor  $Q_4$
  - (j) Range of  $V_{\text{CC}}$  values allowed for TTL
  - (k)  $V_{\text{OH(min)}}$  and  $V_{\text{IH(min)}}$  for the 74ALS series
  - (l)  $V_{\text{IL(max)}}$  and  $V_{\text{OL(max)}}$  for the 74ALS series
  - (m) When a HIGH output supplies current to a load

## SECTION 8-4

- 8-5.\* (a) From Table 8-6, determine the noise margins when a 74LS device is driving a 74ALS input.
- (b) Repeat part (a) for a 74ALS driving a 74LS.
- (c) What will be the overall noise margin of a logic circuit that uses 74LS and 74ALS circuits in combination?
- (d) A certain logic circuit has  $V_{\text{IL}}(\text{max}) = 450 \text{ mV}$ . Which TTL series can be used with this circuit?

## SECTIONS 8-5 AND 8-6

**B** 8-6. DRILL QUESTION

- (a) Define *fan-out*.
- (b)\* In which type of gates do tied-together inputs always count as a single input load in the LOW state?
- (c)\* Define “floating” inputs.
- (d) What causes current spikes in TTL? What undesirable effect can they produce? What can be done to reduce this effect?
- (e) When a TTL output drives a TTL input, where does  $I_{\text{OL}}$  come from? Where does  $I_{\text{OH}}$  go?

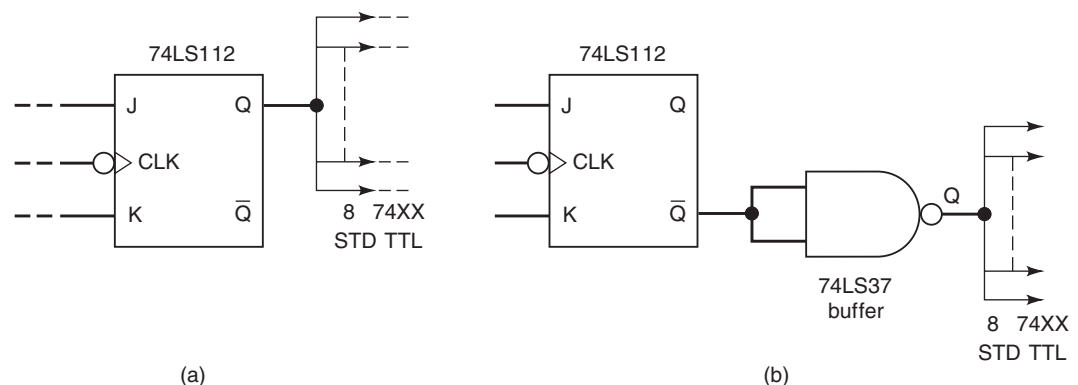
8-7. Use Table 8-11 to find the fan-out for interfacing the first logic family to drive the second.

- (a)\* 74AS to 74AS
- (b)\* 74F to 74F
- (c) 74AHC to 74AS
- (d) 74HC to 74ALS

**B** 8-8. Refer to the data sheet for the 74LS112 J-K flip-flop.

- (a)\* Determine the HIGH and LOW load current at the  $J$  and  $K$  inputs.
- (b) Determine the HIGH and LOW load current at the clock and clear inputs.
- (c) How many 74LS112 clock inputs can the output of one 74LS112 drive?

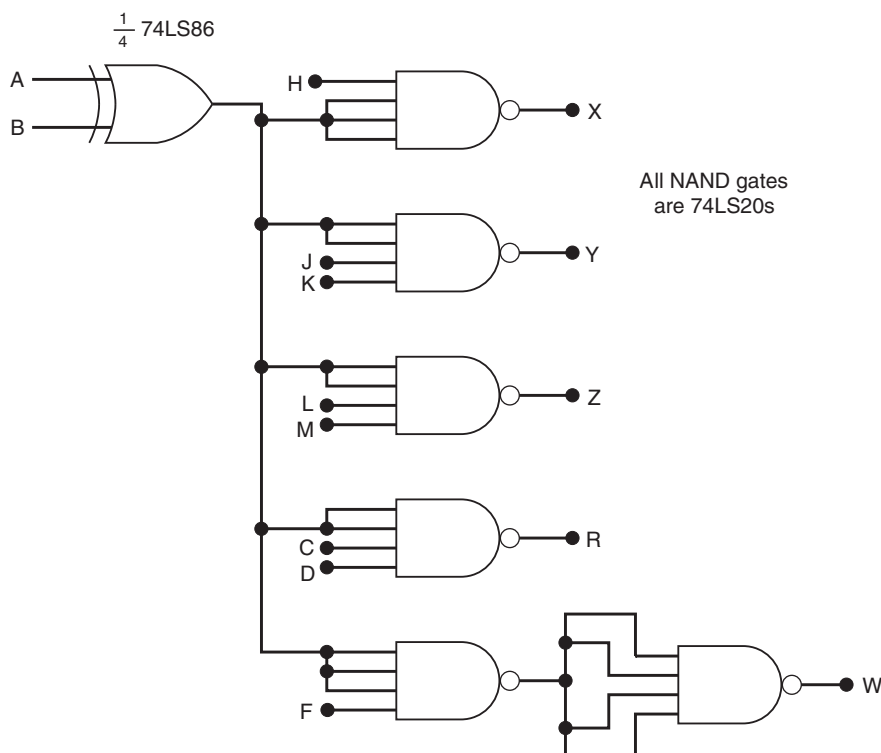
**B** 8-9.\* Figure 8-52(a) shows a 74LS112 J-K flip-flop whose output is required to drive a total of eight standard TTL inputs. Because this exceeds the fan-out of the 74LS112, a buffer of some type is needed. Figure 8-52(b) shows one possibility using one of the NAND gates from the 74LS37



**FIGURE 8-52** Problems 8-9 and 8-10.

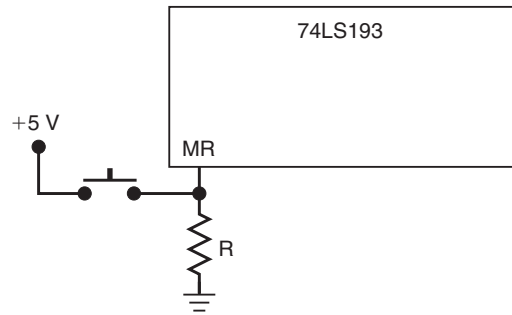
quad NAND buffer, which has a much higher fan-out than the 74LS112. Note that  $\bar{Q}$  is used because the NAND is acting as an INVERTER. Refer to the data sheet for the 74LS37.

- (a) Determine its maximum fan-out to standard TTL.
  - (b) Determine its maximum sink current in the LOW state.
- D** 8-10. Buffer gates are generally more expensive than ordinary gates, and sometimes there are unused ordinary gates available that can be used to solve a loading problem such as that in Figure 8-52(a). Show how 74LS00 NAND gates can be used to solve this problem.
- B** 8-11.\* Refer to the logic diagram of Figure 8-53, where the 74LS86 exclusive-OR output is driving several 74LS20 inputs. Determine whether the fan-out of the 74LS86 is being exceeded, and explain. Repeat using all 74AS devices. Use Table 8-7.

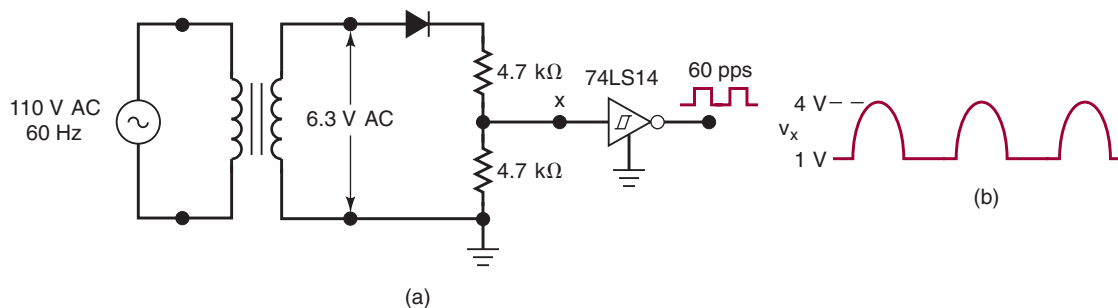


**FIGURE 8-53** Problems 8-11 and 8-13.

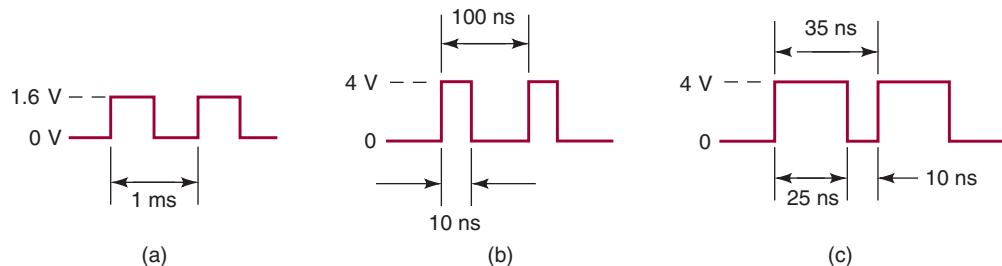
- B** 8-12. How long does it take for the output of a typical 74LS04 to change states in response to a positive-going transition at its input?
- C** 8-13.\* For the circuit of Figure 8-53, determine the longest time it will take for a change in the A input to be felt at output W. Use all worst-case conditions and maximum values of gate propagation delays. (*Hint:* Remember that NAND gates are inverting gates.) Repeat using all 74ALS devices.
- 8-14. (a)\* Figure 8-54 shows a 74LS193 counter with its active-HIGH master reset input activated by a push-button switch. Resistor R is used to hold MR LOW while the switch is open. What is the maximum value that can be used for R?
- (b) Repeat part (a) for the 74ALS193.

**FIGURE 8-54** Problem 8-14.

- C, T** 8-15. Figure 8-55(a) shows a circuit used to convert a 60-Hz sine wave to a 60-pps signal that can reliably trigger FFs and counters. This type of circuit might be used in a digital clock.
- (a) Explain the circuit operation.
- (b)\* A technician is testing this circuit and observes that the 74LS14 output stays LOW. He checks the waveform at the INVERTER input, and it appears as shown in Figure 8-55(b). Thinking that the INVERTER is faulty, he replaces the chip and observes the same results. What do you think is causing the problem, and how can it be fixed? (*Hint: Examine the  $v_x$  waveform carefully.*)

**FIGURE 8-55** Problem 8-15.

- T** 8-16. For each waveform in Figure 8-56, determine *why* it will *not* reliably trigger a 74LS112 flip-flop at its CLK input.

**FIGURE 8-56** Problem 8-16.

- T** 8-17. A technician breadboards a logic circuit for testing. As she tests the circuit's operation, she finds that many of the FFs and counters are triggering erratically. Like any good technician, she checks the  $V_{CC}$  line with a DC meter and reads 4.97 V, which is acceptable for TTL. She then checks all circuit wiring and replaces each IC one by one, but the problem persists. Finally she decides to observe  $V_{CC}$  on the

scope and sees the waveform shown in Figure 8-57. What is the probable cause of the noise on  $V_{CC}$ ? What did the technician forget to include when she breadboarded the circuit?



### SECTIONS 8-7 TO 8-10

- B** 8-18. Which type of MOSFET is turned on by placing
- 5 V on the gate and 0 V on the source?
  - 0 V on the gate and 5 V on the source?
- B** 8-19.\* Which of the following are advantages that CMOS generally has over TTL?
- Greater packing density
  - Higher speed
  - Greater fan-out
  - Lower output impedance
  - Simpler fabrication process
  - More suited for LSI
  - Lower  $P_D$  (below 1 MHz)
  - Transistors as only circuit element
  - Lower input capacitance
  - Less susceptible to ESD
- 8-20. Which of the following operating conditions will probably result in the lowest average  $P_D$  for a CMOS logic system? Explain.
- $V_{DD} = 5\text{ V}$ , switching frequency  $f_{\max} = 1\text{ MHz}$
  - $V_{DD} = 5\text{ V}$ ,  $f_{\max} = 10\text{ kHz}$
  - $V_{DD} = 10\text{ V}$ ,  $f_{\max} = 10\text{ kHz}$
- C** 8-21.\* The output of each INVERTER on a 74LS04 IC is driving two 74HCT08 inputs. The input to each INVERTER is LOW over 99% of the time. What is the maximum power that the 74LS04 chip is dissipating?
- 8-22. Use the values from Table 8-9 to calculate the HIGH-state noise margin when a 74HC gate drives a standard 74LS input.
- 8-23. What will cause latch-up in a CMOS IC? What might happen in this condition? What precautions should be taken to prevent latch-up?
- 8-24. Refer to the data sheet for the 74HC20 NAND gate IC. Use maximum values to calculate  $P_D(\text{avg})$  and  $t_{pd}(\text{avg})$ . Compare with the values calculated in Problem 8-2 for TTL.

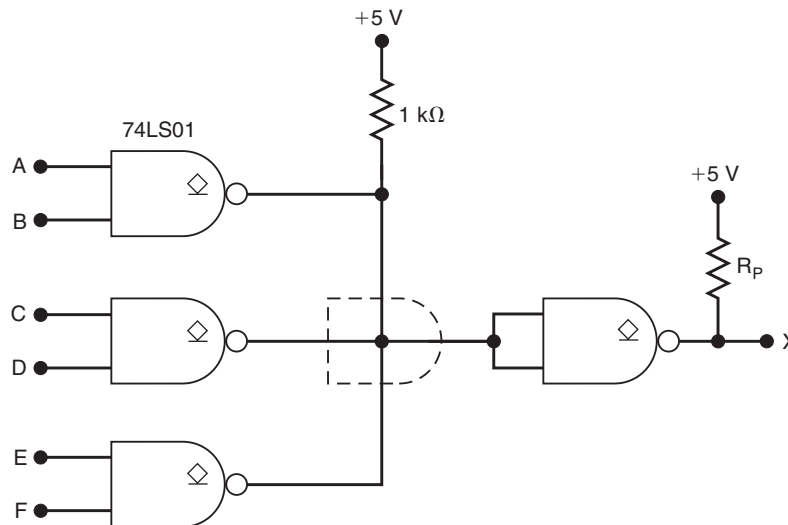
### SECTIONS 8-11 AND 8-12

- B** 8-25. **DRILL QUESTION**
- Define wired-AND.
  - What is a pull-up resistor? Why is it used?
  - What types of TTL outputs can safely be tied together?
  - What is bus contention?

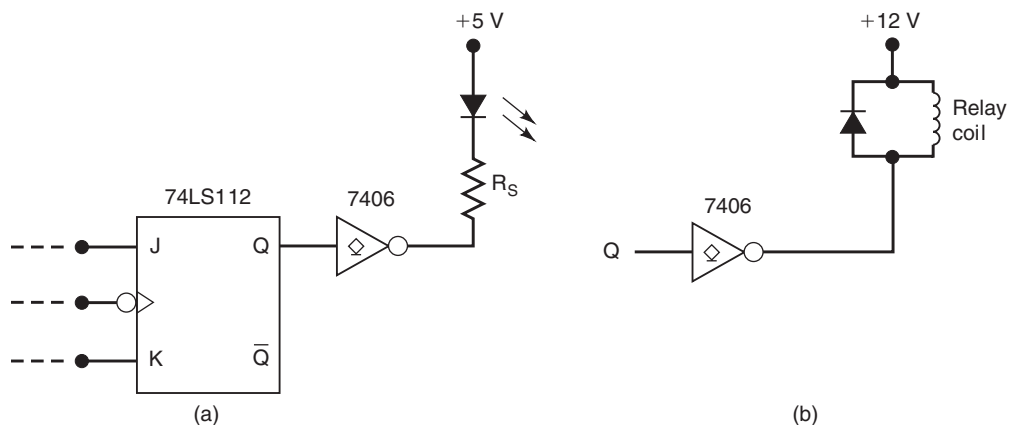


- D** 8-26. The 74LS09 TTL IC is a quad two-input AND with open-collector outputs. Show how 74LS09s can be used to implement the operation  $x = A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot G \cdot H \cdot I \cdot J \cdot K \cdot M$ .
- B** 8-27.\* Determine the logic expression for output X in Figure 8-58.

**FIGURE 8-58** Problem 8-27.



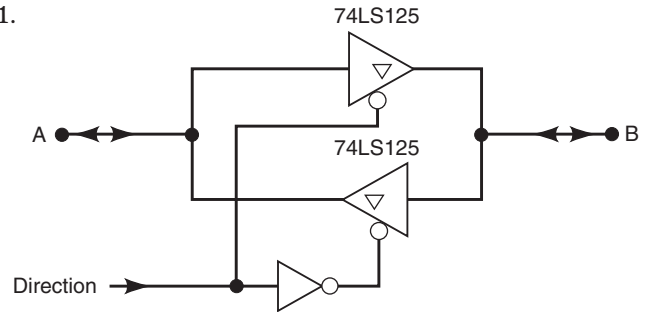
- C** 8-28. Which of the following would be most likely to destroy a TTL totem-pole output while it is trying to switch from HIGH to LOW?
- Tying the output to +5 V
  - Tying the output to ground
  - Applying an input of 7 V
  - Tying the output to another TTL totem-pole output
- D** 8-29.\* Figure 8-59(a) shows a 7406 open-collector inverting buffer used to control the ON/OFF status of an LED to indicate the state of FF output  $Q$ . The LED's nominal specification is  $V_F = 2.4$  V at  $I_F = 20$  mA, and  $I_F(\text{max}) = 30$  mA.
- What voltage will appear at the 7406 output when  $Q = 0$ ?
  - Choose an appropriate value for the series resistor for proper operation.



**FIGURE 8-59** Problems 8-29 and 8-30.

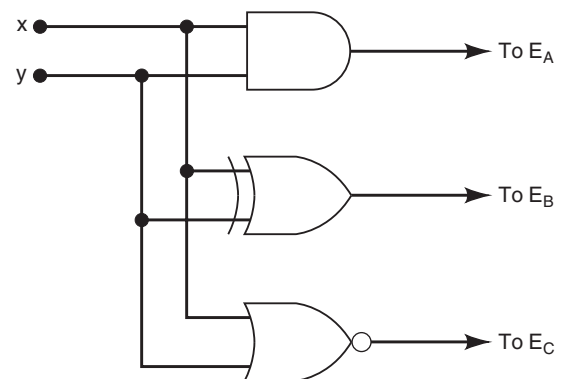
- 8-30. In Figure 8-62(b), the 7406 output is used to switch current to a relay.
- (a)\* What voltage will be at the 7406 output when  $Q = 0$ ?
  - (b)\* What is the largest current relay that can be used?
  - (c) How can we modify this circuit to use a 7407?
- 8-31. Figure 8-60 shows how two tristate buffers can be used to construct a *bidirectional transceiver* that allows digital data to be transmitted in either direction ( $A$  to  $B$ , or  $B$  to  $A$ ). Describe the circuit operation for the two states of the DIRECTION input.

FIGURE 8-60 Problem 8-31.



- 8-32. The circuit of Figure 8-61 is used to provide the enable inputs for the circuit of Figure 8-37.
- (a) Determine which of the data inputs ( $A$ ,  $B$ , or  $C$ ) will appear on the bus for each combination of inputs  $x$  and  $y$ .
  - (b) Explain why the circuit will not work if the NOR is changed to an XNOR.

FIGURE 8-61 Problem 8-32.



- 8-33.\* What type of counter circuit from Chapter 7 could control the enables in Figure 8-37 so that only one buffer is on at any time, and the buffers are enabled sequentially?
- D** 8-34. A Ping ultrasonic ranging module can measure distance based on the time it takes for sound to travel to an object and for the echo to return to the module. The user must provide a positive logic pulse to generate the burst of ultrasound (bang). Immediately after the bang, the module output goes HIGH until the echo is detected and it goes back LOW. The time HIGH for the output pulse is proportional to the distance from the object that caused the echo. However, the Ping module has only a single pin that serves as both input and output. Use tristate buffers and any logic gates necessary to allow a single active-HIGH push button input to pulse the signal pin and immediately after this pulse is finished to allow the signal pin to drive an output.

## SECTION 8-13

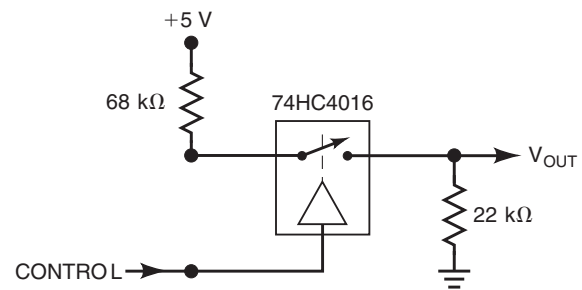
## 8-35. DRILL QUESTION

- How is resistance present in all transmission lines?
- How is inductance present in all transmission lines?
- How is capacitance present in all transmission lines?
- The combined effects of the R, L, and C components of a transmission line is referred to as the line's \_\_\_\_\_?
- Reflected waves and ringing are minimized by matching the line impedance with the \_\_\_\_\_ impedance.
- The circuits placed at the end of a transmission line to minimize adverse effects are called line \_\_\_\_\_.
- What does the acronym LVDS stand for?
- How are the two logic levels differentiated in LVDS?

## SECTION 8-14

- 8-36.\* Determine the approximate values of  $V_{OUT}$  for both states of the CONTROL input in Figure 8-62.

FIGURE 8-62 Problem 8-36.



- 8-37.\* Determine the waveform at output X in Figure 8-63 for the given input waveforms. Assume that  $R_{ON} \approx 200 \Omega$  for the bilateral switch.

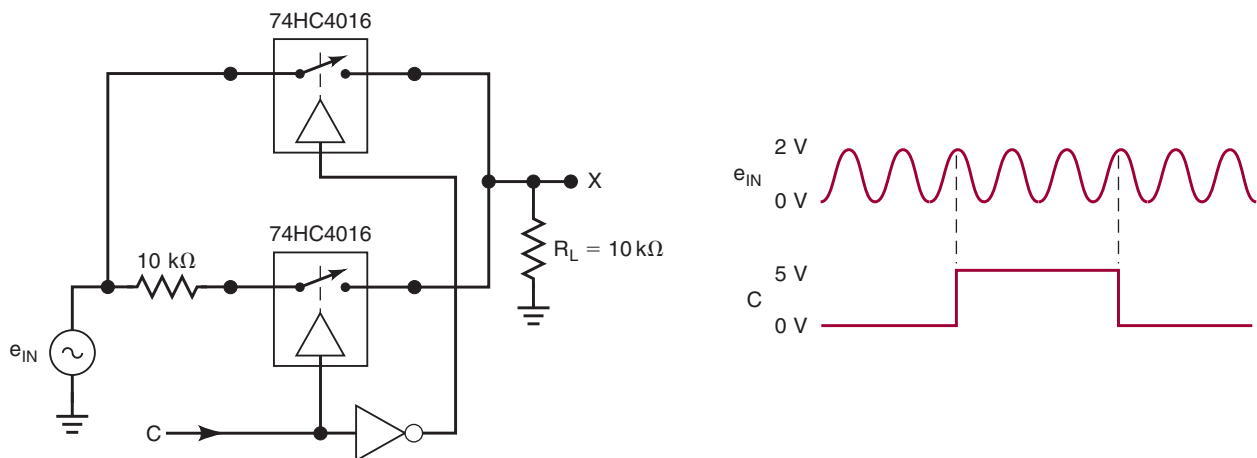
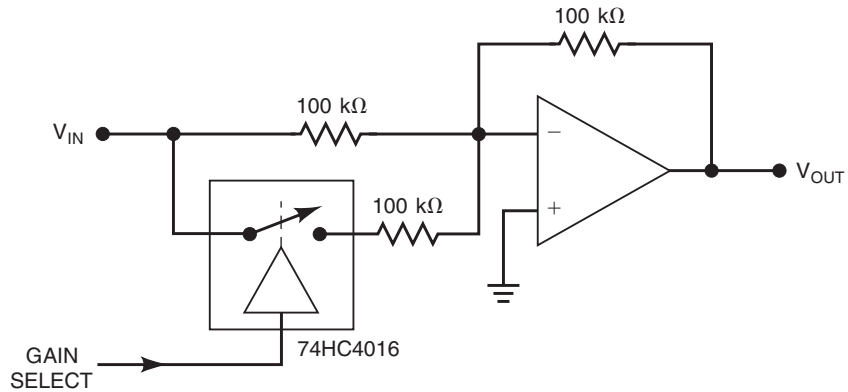


FIGURE 8-63 Problem 8-37.

- D, C** 8-38.\* Determine the gain of the op-amp circuit of Figure 8-64 for the two states of the GAIN SELECT input. This circuit shows the basic principle of digitally controlled signal amplification.

FIGURE 8-64 Problem 8-38.



SECTION 8-15

B 8-39.\* DRILL QUESTION

- (a) Which CMOS series can have its inputs driven directly from a TTL output?
- (b) What is the function of a level translator? When is it used?
- (c) Why is a buffer required between some CMOS outputs and TTL inputs?
- (d) *True or false:* Most CMOS outputs have trouble supplying the TTL HIGH-state input current.

T 8-40. Refer to Figure 8-65(a), where a 74LS TTL output, *Q*, is driving a CMOS INVERTER operating at  $V_{DD} = 10\text{ V}$ . The waveforms at *Q* and *X* appear as shown in Figure 8-65(b). Which of the following is a possible reason why *X* stays HIGH?

- (a) The 10-V supply is faulty.
- (b) The pull-up resistor is too large.

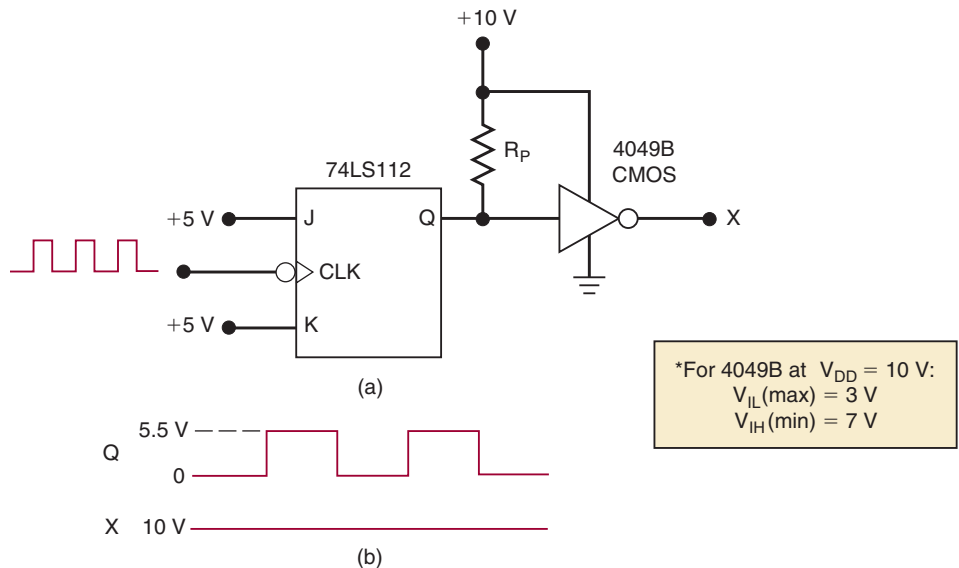
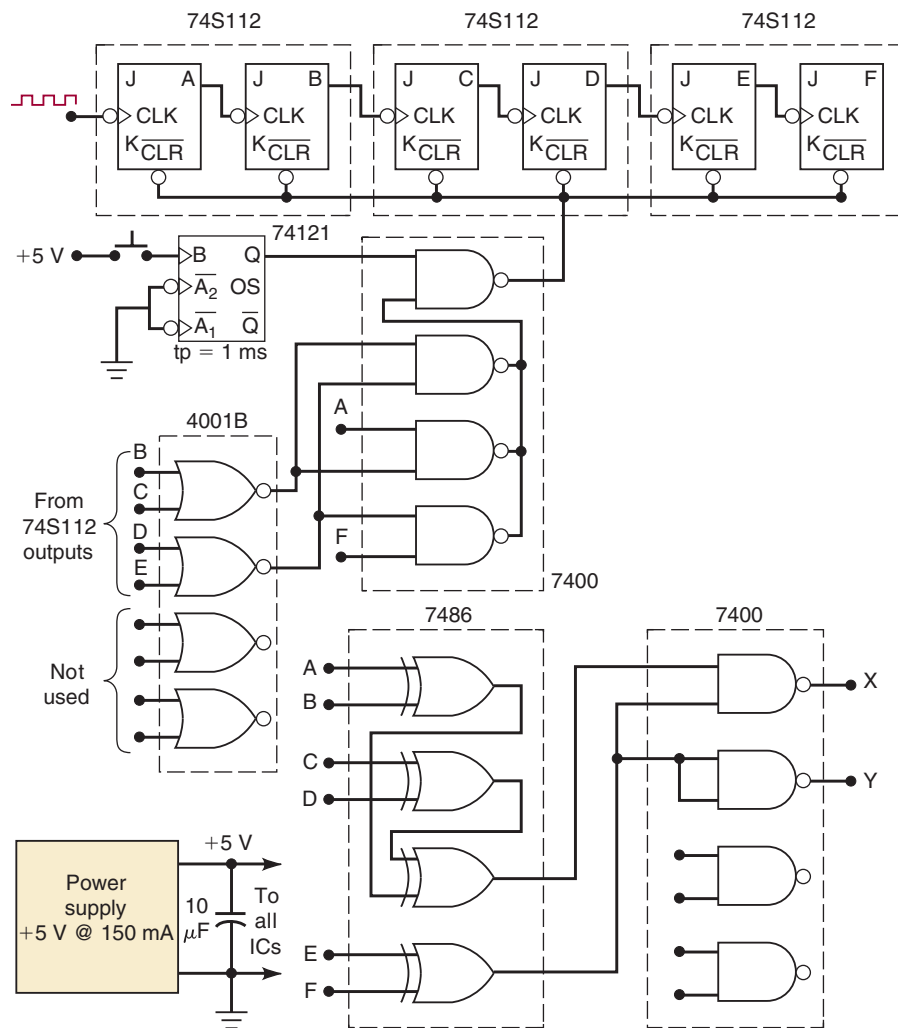


FIGURE 8-65 Problem 8-40.

- (c) The 74LS112 output breaks down at well below 10 V and maintains a 5.5-V level in the HIGH state. This is in the indeterminate range for the CMOS input.
- (d) The CMOS input is loading down the TTL output.
- 8-41. (a)\* Use Table 8-11 to determine how many 74AS inputs can typically be driven by a 4000B output.
- (b) Repeat part (a) for a 74HC output.
- T** 8-42. Figure 8-66 is a logic circuit that was poorly designed. It contains at least eight instances where the characteristics of the ICs have not been properly taken into account. Find as many of these as you can.



**FIGURE 8-66** Problems 8-42 and 8-43.

- T** 8-43. Repeat Problem 8-42 with the following changes in the circuit:
- Each TTL IC is replaced with its 74LS equivalent.
  - The 4001B is replaced with a 74HCT02.
- 8-44.\* Use Table 8-11 to explain why the circuit of Figure 8-67 will not work as it is. How can the problem be corrected?

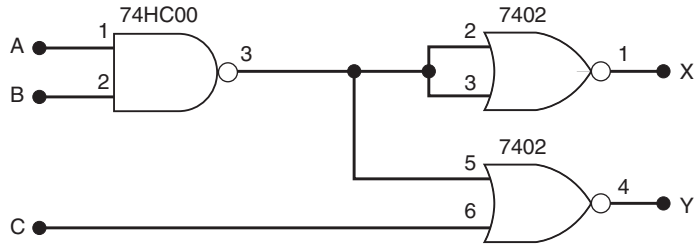


FIGURE 8-67 Problem 8-44.

SECTION 8-17

- D 8-45. The gas tank on your car has a fuel-level sending unit that works like a potentiometer. A float moves up and down with the gasoline level, changing the variable resistor setting and producing a voltage proportional to the gas level. A full tank produces 12 V, and an empty tank produces 0 V. Design a circuit using an LM339 that turns on the “Fuel Low” indicator lamp when the voltage level from the sending unit gets below 0.5 V.
- D 8-46.\* The over-temperature comparator circuit in Figure 8-49 is modified by replacing the LM34 temperature sensor with an LM35 that outputs 10 mV per degree Celsius. The alarm must still be activated (HIGH) when the temperature is over 100°F, which is equal to approximately 38°C. Recalculate the values of  $R_1$  and  $R_2$  to complete the modification.

SECTION 8-18

- T 8-47. The circuit in Figure 8-68 uses a 74HC05 IC that contains six open-drain INVERTERS. The INVERTERS are connected in a wired-AND arrangement. The output of the NAND gate is always HIGH,

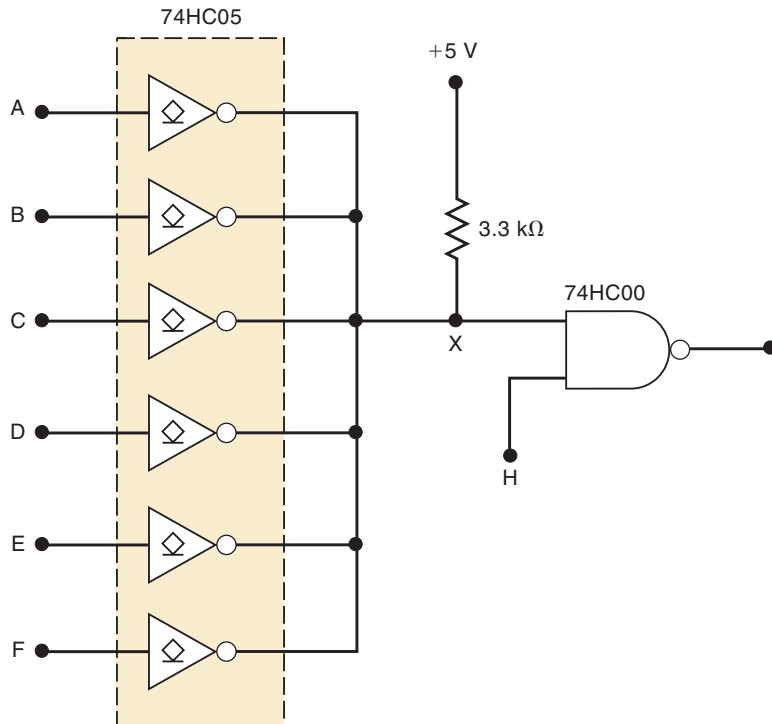


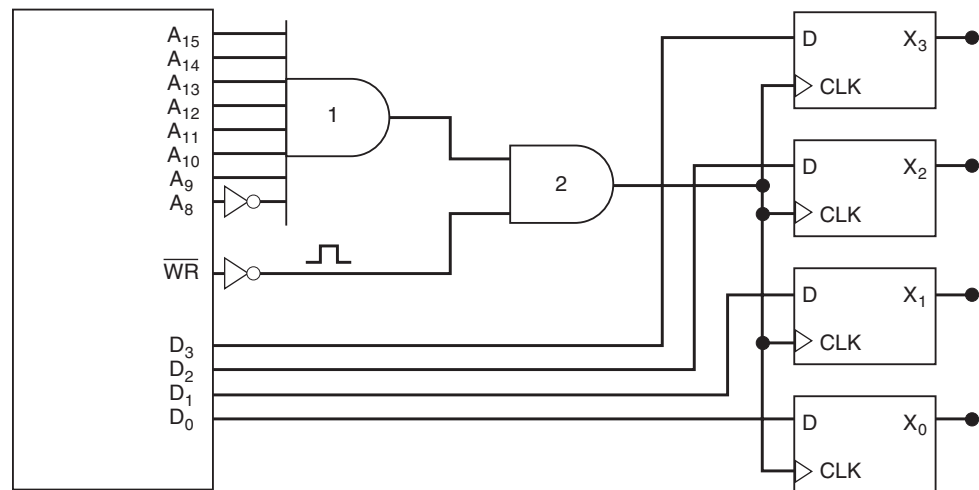
FIGURE 8-68 Problem 8-47.

regardless of the inputs A–H. Describe a procedure that uses a logic probe and pulser to isolate this fault.

- T** 8-48. The circuit of Figure 8-50 has a solder bridge to ground somewhere between the output of the NAND gate and the input of the FF. Describe a procedure for a test that could be performed to indicate that the fault is on the circuit board and probably not in either the NAND or the FF ICs.
- T** 8-49.\* In Figure 8-44, a logic probe indicates that the lower end of the pull-up resistor is stuck in the LOW state. Which of the following is the possible fault?
- The TTL gate's current-sourcing transistor is open.
  - The TTL gate's current-sinking transistor has a collector–emitter short.
  - There is a break in the connection from  $R_p$  to the CMOS gate.

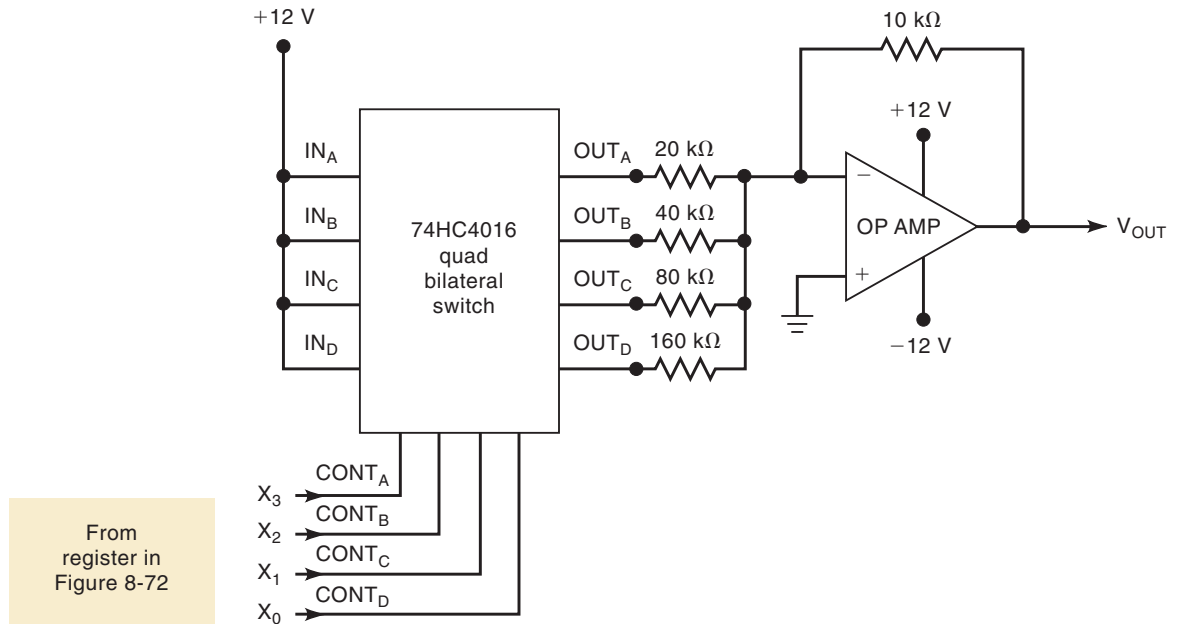
### MICROCOMPUTER APPLICATION

- C** 8-50.\* In Chapter 5, we saw how a microprocessor (MPU), under software control, transfers data to an external register. The circuit diagram is repeated in Figure 8-69. Once the data are in the register, they are stored there and used for whatever purpose they are needed. Sometimes, each individual bit in the register has a unique function. For example, in automobile computers, each bit could represent the status of a different physical variable being monitored by the MPU. One bit might indicate when the engine temperature is too high. Another bit might signal when oil pressure is too low. In other applications, the bits in the register are used to produce an analog output that can be used to drive devices requiring analog inputs that have many different voltage levels. Sketch the waveform from the DAC.



**FIGURE 8-69** Problems 8-50.

Figure 8-70 shows how we can use the MPU to generate an analog voltage by taking the register data from Figure 8-69 and using them to control the inputs to a summing amplifier. Assume that the MPU is



**FIGURE 8-70** Problem 8-50.

executing a program that is transferring a new set of data to the register every  $10\ \mu\text{s}$  according to Table 8-15. Sketch the resulting waveform at  $V_{\text{OUT}}$ .

**TABLE 8-15** Problem 8-50.

Time ( $\mu\text{s}$ )	MPU Data
0	0000
10	0010
20	0100
30	0111
40	1010
50	1110
60	1111
70	1111
80	1110
90	1100
100	1000

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

### SECTION 8-1

- See text.
- False
- False;  $V_{\text{NH}}$  is the difference between  $V_{\text{OH(min)}}$  and  $V_{\text{IH(min)}}$ .
- Current sinking: an output actually receives (sinks) current from the input of the circuit it is driving. Current sourcing: an output supplies (sources) current to the circuit it is driving.
- DIP
- J-lead
- Its leads are bent.
- No



**SECTION 8-2**

1. True
2. LOW
3. Fast switching times, low power dissipation; large current spike during switching from LOW to HIGH
4.  $Q_3$
5.  $Q_6$
6. No multiple-emitter transistor

**SECTION 8-3**

1.  $I_{OHmax} = 0.4 \text{ mA}$
2.  $V_{ILmax} = 0.8 \text{ V}$
3.  $t_{PLHmax} = 11 \text{ ns}$
4.  $I_{CCLmax} = 3 \text{ mA}$

**SECTION 8-4**

1. (a) 74AS (b) 74S, 74LS (c) standard 74 (d) 74S, 74LS, 74AS, 74ALS (e) 74ALS
2. All three can operate at 40 MHz, but the 74ALS193 will use less power.
3.  $Q_4$ ,  $Q_5$ , respectively

**SECTION 8-5**

1.  $Q_4$ 's ON-state resistance and  $V_{OL(max)}$
2. 12
3. Its output voltages may not remain in the allowed logic 0 and 1 ranges.
4. Two; five

**SECTION 8-6**

1. LOW
2. Connect to  $+V_{CC}$  through a 1-k $\Omega$  resistor; connect to another used input
3. Connect to ground; connect to another used input
4. False; only in the LOW state
5. Connecting small RF capacitors between  $V_{CC}$  and ground near each TTL IC to filter out voltage spikes caused by rapid current changes during output transitions from LOW to HIGH

**SECTION 8-7**

1. 1 kilo-ohm.
2. 1 Giga-ohm.
3. Capacitor.

**SECTION 8-8**

1. CMOS uses both P- and N-channel MOSFETs.
2. One
3. Six

**SECTION 8-9**

1. 74C, HC, HCT, AHC, AHCT
2. 74ACT, HCT, AHCT
3. 74C, HC/HCT, AC/ACT, AHC/AHCT
4. BiCMOS
5. Maximum permissible propagation delay; input capacitance of each load
6. See text.
7. CMOS
8. (a) True (b) False (c) False (d) False (e) True (f) False

**SECTION 8-10**

1. More circuits on chip; higher operating speed
2. Can't handle higher voltages: increased power dissipation can overheat the chip.
3. Same as standard TTL: 2.0 V
4. 74ALVC, 74LV
5. 74LVT

**SECTION 8-11**

1. When two or more circuit outputs are connected together
2. Damaging current can flow;  $V_{OL}$  exceeds  $V_{OL(max)}$ .
3. Current-sinking transistor  $Q_4$ 's collector is unconnected (there is no  $Q_3$ ).
4. To produce a  $V_{OH}$  level
5.  $\overline{A} \overline{B} \overline{C} \overline{D} \overline{E} \overline{F}$
6. No active pull-up transistor
7. See Figure 8-33.

**SECTION 8-12**

1. HIGH, LOW, Hi-Z
2. Hi-Z
3. When two or more tristate outputs tied to a common bus are enabled at the same time
4.  $E_A = E_B = 0$ ,  $E_C = 1$
5. See Figure 8-38.

**SECTION 8-13**

1. Less than 4 inches    2. Resistance, capacitance, inductance    3. To reduce reflections and reactive ringing on the line.

**SECTION 8-14**

1. The logical level at the control input controls the open/closed status of a bidirectional switch that can pass analog signals in either direction.    2. True

**SECTION 8-15**

1. A pull-up resistor must be connected to +5 V at the TTL output.    2. CMOS  $I_{OH}$  or  $I_{OL}$  may be too low.

**SECTION 8-16**

1. It takes the output from a driver circuit and conditions it so that it is compatible with the input requirements of the load.    2. True    3. False; for example, the 4000B series cannot sink  $I_{IL}$  of a 74 or a 74AS device.    4. 74HCT and ACT  
5. Two

**SECTION 8-17**

1.  $V^{(+)} > V^{(-)}$     2.  $V^{(-)} > V^{(+)}$     3. Open-collector

**SECTION 8-18**

1. It injects a voltage pulse of selected polarity at a node that is not shorted to  $V_{CC}$  or ground.    2. False    3. False    4. The pulse LED flashes once each time the pulser is activated.

---



## MSI LOGIC CIRCUITS

### ■ OUTLINE

- |      |                                       |      |   |
|------|---------------------------------------|------|---|
| 9-1  | Decoders                              | 9-12 | Data Busing                             |
| 9-2  | BCD-to-7-Segment Decoder/<br>Drivers  | 9-13 | The 74ALS173/HC173<br>Tristate Register |
| 9-3  | Liquid-Crystal Displays               | 9-14 | Data Bus Operation                      |
| 9-4  | Encoders                              | 9-15 | Decoders Using HDL                      |
| 9-5  | Troubleshooting                       | 9-16 | The HDL 7-Segment<br>Decoder/Driver     |
| 9-6  | Multiplexers<br>(Data Selectors)      | 9-17 | Encoders Using HDL                      |
| 9-7  | Multiplexer Applications              | 9-18 | HDL Multiplexers and<br>Demultiplexers  |
| 9-8  | Demultiplexers<br>(Data Distributors) | 9-19 | HDL Magnitude<br>Comparators            |
| 9-9  | More Troubleshooting                  | 9-20 | HDL Code Converters                     |
| 9-10 | Magnitude Comparator                  |      |   |
| 9-11 | Code Converters                       |      |   |

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Analyze and use decoders and encoders in various types of circuit applications.
- Compare the advantages and disadvantages of LEDs and LCDs.
- Utilize the observation/analysis technique to troubleshoot digital circuits.
- Analyze the operation of multiplexers and demultiplexers in circuit applications.
- Compare two binary numbers by using the magnitude comparator circuit.
- Describe the function and operation of code converters.
- Cite the precautions that must be considered when connecting digital circuits using the data bus concept.
- Use HDL to implement the equivalent of MSI logic circuits.

## ■ INTRODUCTION

Digital systems obtain binary-coded data and information that are continuously being operated on in some manner. Some of the operations include: (1) *decoding and encoding*, (2) *multiplexing*, (3) *demultiplexing*, (4) *comparison*, (5) *code conversion*, and (6) *data busing*. All of these operations and others have been facilitated in the past by the availability of numerous ICs in the MSI (medium-scale-integration) category.

In this chapter, we will study many of the common functional building blocks. For each type of block, we will start with a brief discussion of its basic operating principle and then introduce specific examples. We then show how they can be used alone or in combination with other functional blocks in various applications.

### 9-1 DECODERS

#### OUTCOMES

*Upon completion of this section, you will be able to:*

- Define the term *decoder*.
- State the nature of the inputs and outputs of a decoder.
- State the role of an “enable” input to a decoder.

A **decoder** is a logic circuit that accepts a set of inputs that represents a binary number and activates only the output that corresponds to that input number. In other words, a decoder circuit looks at its inputs, determines which binary number is present there, and activates the one output that

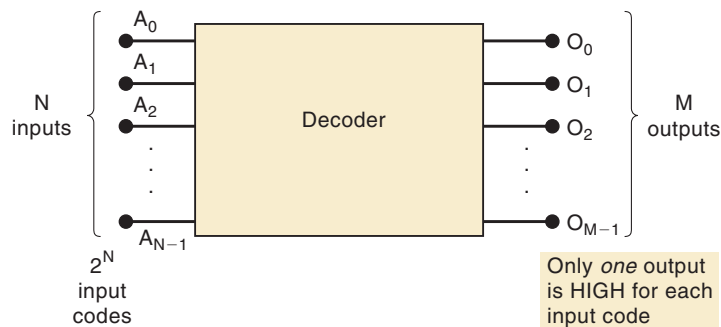
corresponds to that number; all other outputs remain inactive. The diagram for a general decoder is shown in Figure 9-1 with  $N$  inputs and  $M$  outputs. Because each of the  $N$  inputs can be 0 or 1, there are  $2^N$  possible input combinations or codes. For each of these input combinations, only one of the  $M$  outputs will be active (HIGH); all the other outputs are LOW. Many decoders are designed to produce active-LOW outputs, where only the selected output is LOW while all others are HIGH. This situation is indicated by the presence of small circles (bubbles) on the output lines in the decoder diagram.

Some decoders do not utilize all of the  $2^N$  possible input codes but only certain ones. For example, a BCD-to-decimal decoder has a four-bit input code and *ten* output lines that correspond to the *ten* BCD code groups 0000 through 1001. Decoders of this type are often designed so that if any of the unused codes are applied to the input, *none* of the outputs will be activated.

In Chapter 7, we saw how decoders are used in conjunction with counters to detect the various states of the counter. In that application, the FFs in the counter provided the binary code inputs for the decoder. The same basic decoder circuitry is used no matter where the inputs come from. Figure 9-2 shows the circuitry for a decoder with three inputs and  $2^3 = 8$  outputs. It uses all AND gates, and so the outputs are active-HIGH. Note that for a given input code, the only output that is active (HIGH) is the one corresponding to the decimal equivalent of the binary input code (e.g., output  $O_6$  goes HIGH only when  $CBA = 110_2 = 6_{10}$ ).

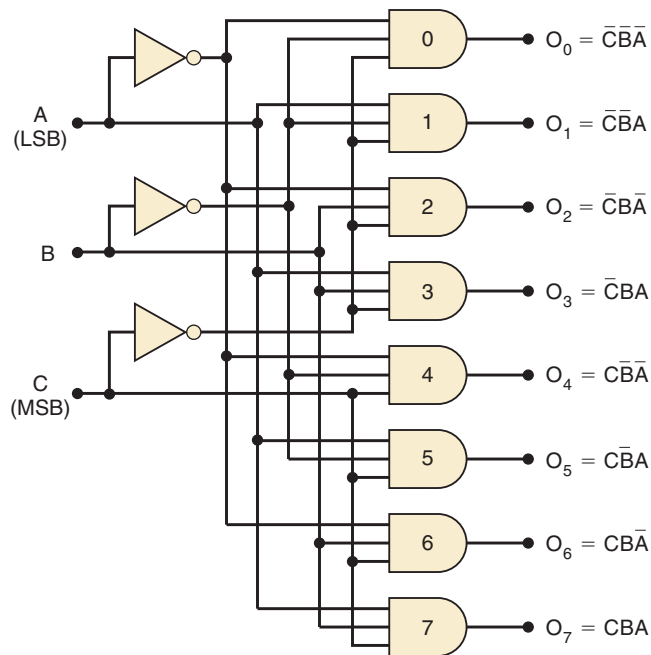
This decoder can be referred to in several ways. It can be called a *3-line-to-8-line decoder* because it has three input lines and eight output lines. It can also be called a *binary-to-octal decoder* or *converter* because it takes a three-bit binary input code and activates one of the eight (octal) outputs corresponding to that code. It is also referred to as a *1-of-8 decoder* because only one of the eight outputs is activated at one time.

**FIGURE 9-1** General decoder diagram.



## ENABLE Inputs

Some decoders have one or more enable inputs that are used to control the operation of the decoder. For example, refer to the decoder in Figure 9-2 and visualize having a common *ENABLE* line connected to a fourth input of each gate. With this *ENABLE* line held HIGH, the decoder will function normally, and the *A, B, C* input code will determine which output is HIGH. With *ENABLE* held LOW, however, *all* of the outputs will be forced to the LOW state regardless of the levels at the *A, B, C* inputs. Thus, the decoder is enabled only if *ENABLE* is HIGH.



C	B	A	$O_7$	$O_6$	$O_5$	$O_4$	$O_3$	$O_2$	$O_1$	$O_0$
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

**FIGURE 9-2** A 3-line-to-8-line (or 1-of-8) decoder.

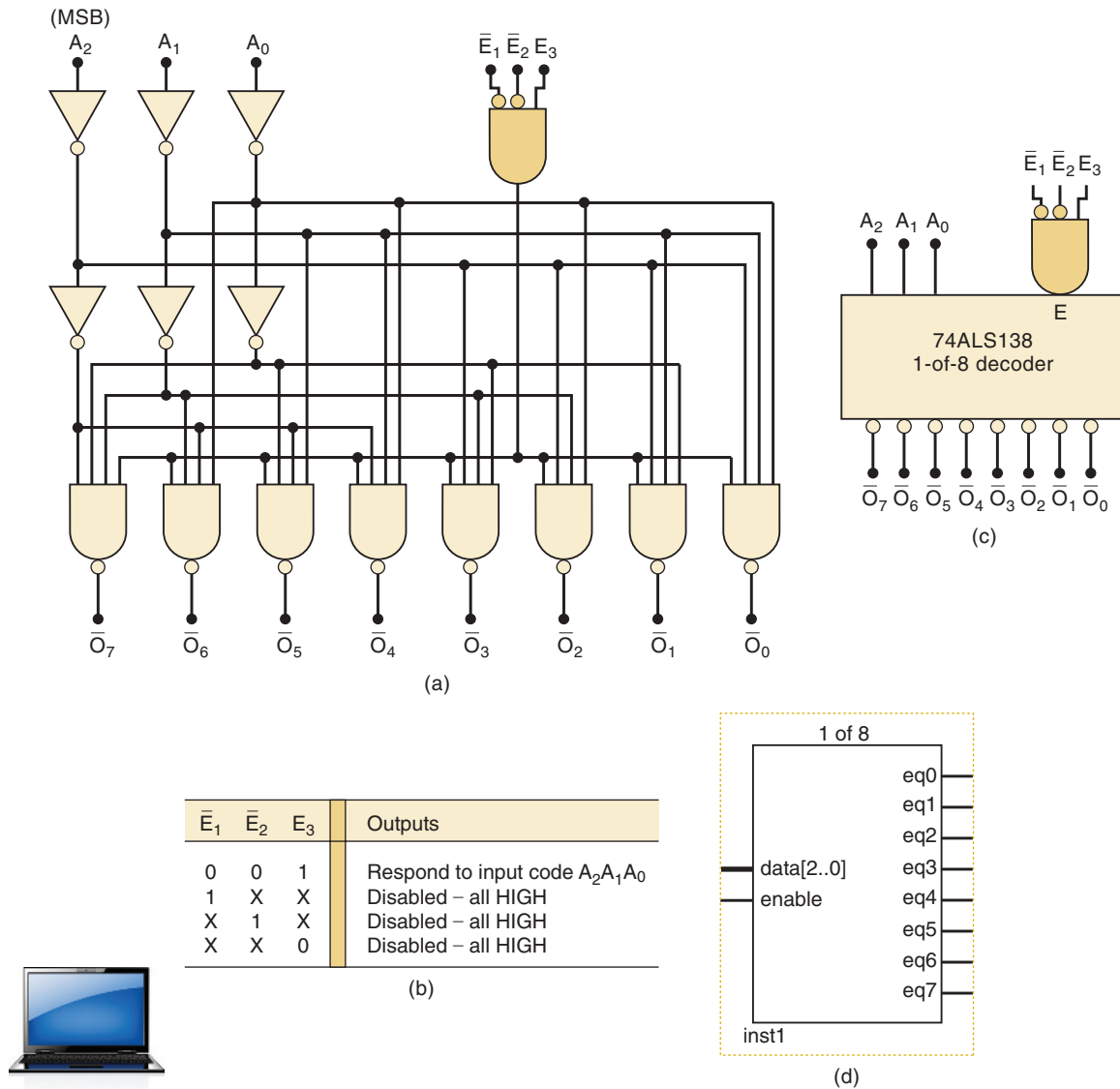
Figure 9-3(a) shows the logic diagram for the 74ALS138 decoder. By examining this diagram carefully, we can determine exactly how this decoder functions. First, notice that it has NAND gate outputs, so its outputs are active-LOW. Another indication is the labeling of the outputs as  $\overline{O}_7$ ,  $\overline{O}_6$ ,  $\overline{O}_5$ , and so on; the overbar indicates active-LOW outputs.

The input code is applied at  $A_2$ ,  $A_1$ , and  $A_0$ , where  $A_2$  is the MSB. With three inputs and eight outputs, this is a 3-to-8 decoder or, equivalently, a 1-of-8 decoder.

Inputs  $\overline{E}_1$ ,  $\overline{E}_2$ , and  $E_3$  are separate enable inputs that are combined in the AND gate. In order to enable the output NAND gates to respond to the input code at  $A_2A_1A_0$ , this AND gate output must be HIGH. This will occur only when  $\overline{E}_1 = \overline{E}_2 = 0$  and  $E_3 = 1$ . In other words,  $\overline{E}_1$  and  $\overline{E}_2$  are active-LOW,  $E_3$  is active-HIGH, and all three must be in their active states to activate the decoder outputs. If one or more of the enable inputs is in its inactive state, the AND output will be LOW, which will force all NAND outputs to their inactive HIGH state regardless of the input code. This operation is summarized in the truth table in Figure 9-3(b). Recall that  $x$  represents the don't-care condition.

The logic symbol for the 74ALS138 is shown in Figure 9-3(c). Note how the active-LOW outputs are represented and how the enable inputs are represented. Even though the enable AND gate is shown as external to the decoder block, it is part of the IC's internal circuitry. The 74HC138 is the high-speed CMOS version of this decoder.

Using Quartus II software and the MegaWizard Plug-In Manager, as described in earlier chapters, we can create functional blocks like decoders. The wizard allows you to choose things such as how many inputs you want, the presence of an *ENABLE* input, and which decoded outputs should be included in the functional block. Figure 9-3(d) shows a megafunction block for a decoder similar to the 74138. Two differences are that this megafunction only has one *ENABLE* input (active-HIGH) and it has active-HIGH outputs.



**FIGURE 9-3** (a) Logic diagram for the 74ALS138 decoder; (b) truth table; (c) logic symbol; (d) a Quartus II megafunction.

### EXAMPLE 9-1

Indicate the states of the 74ALS138 outputs for each of the following sets of inputs.

- (a)  $E_3 = \bar{E}_2 = 1, \bar{E}_1 = 0, A_2 = A_1 = 1, A_0 = 0$   
 (b)  $E_3 = 1, \bar{E}_2 = \bar{E}_1 = 0, A_2 = 0, A_1 = A_0 = 1$

#### Solution

- (a) With  $\bar{E}_2 = 1$ , the decoder is disabled and all of its outputs will be in their inactive HIGH state. This can be determined from the truth table or by following the input levels through the circuit logic.  
 (b) All of the enable inputs are activated, so the decoding portion is enabled. It will decode the input code  $011_2 = 3_{10}$  to activate output  $\bar{O}_3$ . Thus,  $\bar{O}_3$  will be LOW and all other outputs will be HIGH.

## EXAMPLE 9-2

Figure 9-4(a) shows how four 74ALS138s and an INVERTER can be arranged to function as a 1-of-32 decoder. The decoders are labeled  $Z_1$  to  $Z_4$  for easy reference, and the eight outputs from each one are combined into 32 outputs.  $Z_1$ 's outputs are  $\bar{O}_0$  to  $\bar{O}_7$ ;  $Z_2$ 's outputs  $\bar{O}_0$  to  $\bar{O}_7$  are renamed  $\bar{O}_8$  to  $\bar{O}_{15}$ , respectively;  $Z_3$ 's outputs are renamed  $\bar{O}_{16}$  to  $\bar{O}_{23}$ ; and  $Z_4$ 's are renamed  $\bar{O}_{24}$  to  $\bar{O}_{31}$ . A five-bit input code  $A_4A_3A_2A_1A_0$  will activate only one of these 32 outputs for each of the 32 possible input codes.

- Which output will be activated for  $A_4A_3A_2A_1A_0 = 01101$ ?
- What range of input codes will activate the  $Z_4$  chip?
- Create a megafunction circuit in Quartus that will implement a 1-of-32 decoder with active-HIGH outputs.

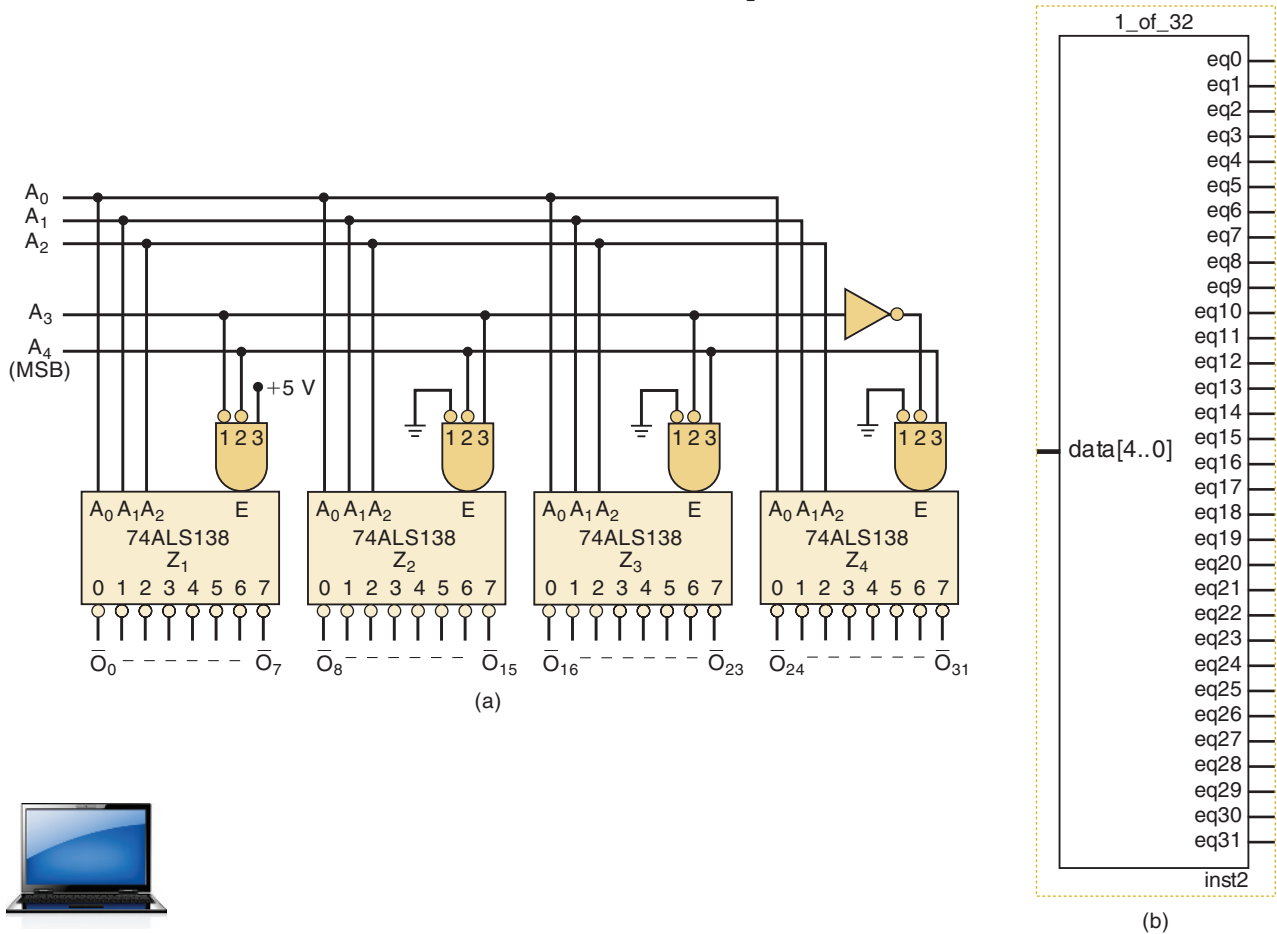


FIGURE 9-4 (a) Four 74ALS138s forming a 1-of-32 decoder; (b) a 1-of-32 decoder megafunction.

## Solution

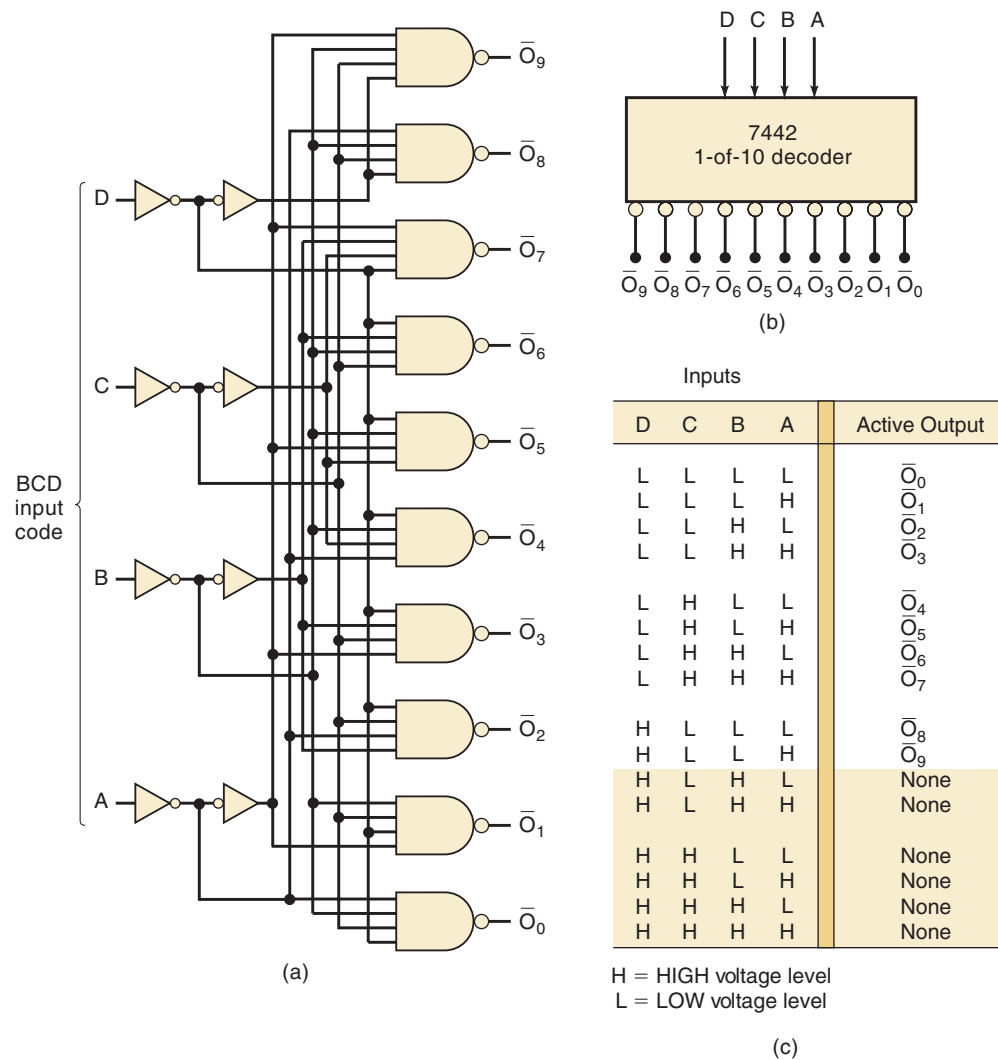
- The five-bit code has two distinct portions. The  $A_4$  and  $A_3$  bits determine which one of the decoder chips  $Z_1$  to  $Z_4$  will be enabled, while  $A_2A_1A_0$  determine which output of the enabled chip will be activated. With  $A_4A_3 = 01$ , only  $Z_2$  has all of its enable inputs activated. Thus,  $Z_2$  responds to the  $A_2A_1A_0 = 101$  code and activates its  $\bar{O}_5$  output, which has been renamed  $\bar{O}_{13}$ . Thus, the input code 01101, which is the binary equivalent of decimal 13, will cause output  $\bar{O}_{13}$  to go LOW, while all others stay HIGH.



- (b) To enable  $Z_4$ , both  $A_4$  and  $A_3$  must be HIGH. Thus, all input codes ranging from 11000 ( $24_{10}$ ) to 11111 ( $31_{10}$ ) will activate  $Z_4$ . This corresponds to outputs  $\bar{O}_{24}$  to  $\bar{O}_{31}$ .
- (c) Use the MegaWizard Plug-in Manager to create the decoder in Figure 9-4(b).

### BCD-to-Decimal Decoders

Figure 9-5(a) shows the logic diagram for a 7442 **BCD-to-decimal decoder**. It is also available as a 74LS42 and a 74HC42. Each output goes LOW only when its corresponding BCD input is applied. For example,  $\bar{O}_5$  will go LOW only when inputs  $DCBA = 0101$ ;  $\bar{O}_8$  will go LOW only when  $DCBA = 1000$ . For input combinations that are invalid for BCD, none of the outputs will be activated. This decoder can also be referred to as a *4-to-10 decoder* or a *1-of-10 decoder*. The logic symbol and the truth table for the 7442 are also shown in the figure. Note that this decoder does not have an enable input. In Problem 9-7(a), we will see how the 7442 can be used as a 3-line-to-8-line decoder, with the  $D$  input used as an *ENABLE* input.



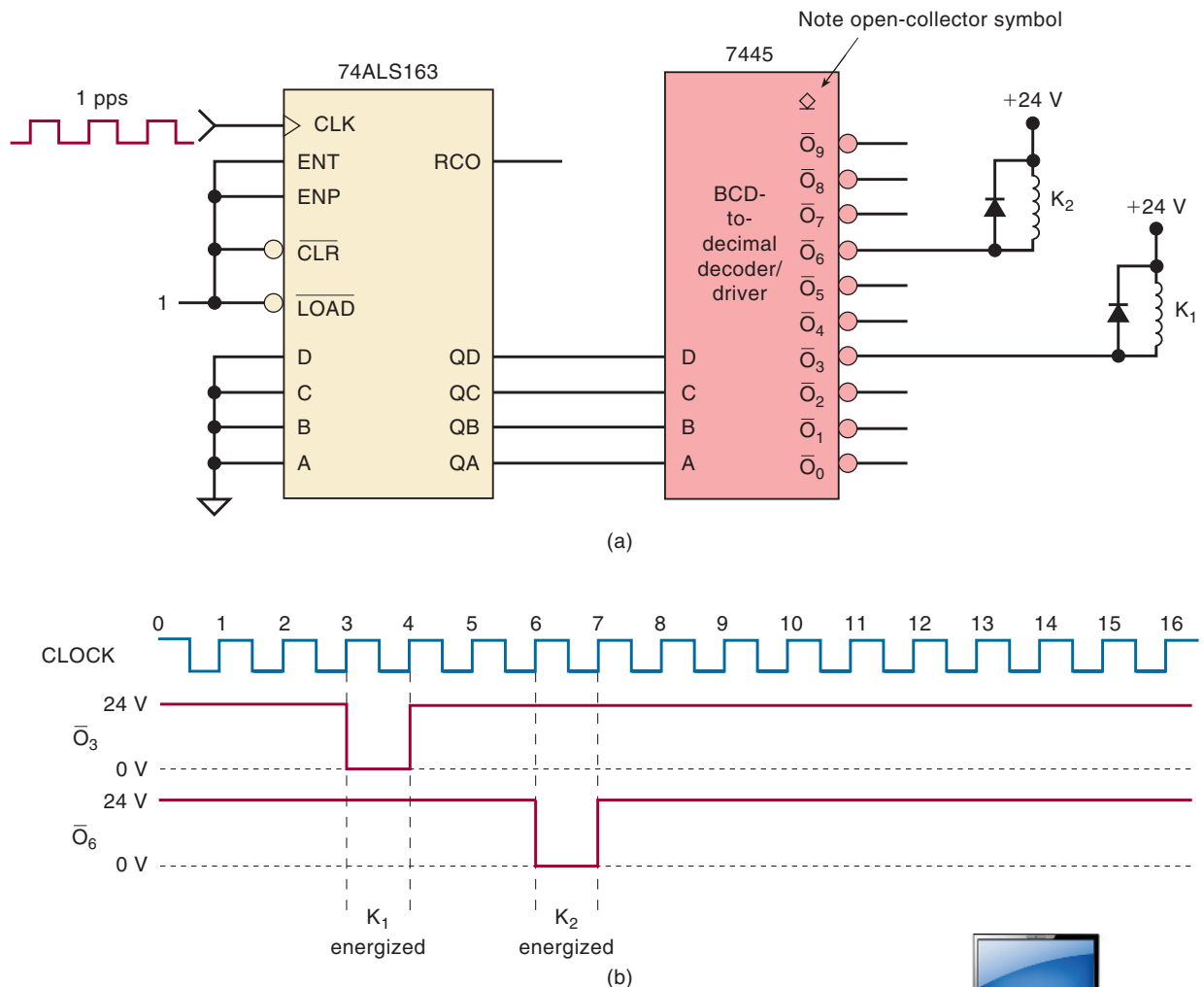
**FIGURE 9-5** (a) Logic diagram for the 7442 BCD-to-decimal decoder; (b) logic symbol; (c) truth table.

## BCD-to-Decimal Decoder/Driver

The TTL 7445 is a BCD-to-decimal decoder/driver. The term *driver* is added to its description because this IC has open-collector outputs that can operate at higher current and voltage limits than a normal TTL output. The 7445's outputs can sink up to 80 mA in the LOW state, and they can be pulled up to 30 V in the HIGH state. This makes them suitable for directly driving loads such as indicator LEDs or lamps, relays, or DC motors.

## Decoder Applications

Decoders are used whenever an output or a group of outputs is to be activated only on the occurrence of a specific combination of input levels. These input levels are often provided by the outputs of a counter or a register. When the decoder inputs come from a counter that is being continually pulsed, the decoder outputs will be activated sequentially, and they can be used as timing or sequencing signals to turn devices on or off at specific times. An example of this operation is shown in Figure 9-6 using the 74ALS163 counter and the 7445 decoder/driver described above.



**FIGURE 9-6** Example 9-3: counter/decoder combination used to provide timing and sequencing operations.



**EXAMPLE 9-3**

Describe the operation of the circuit in Figure 9-6(a).

**Solution**

The counter is being pulsed by a 1 pulse per second (pps) signal so that it will sequence through the binary counts at the rate of 1 count/s. The counter FF outputs are connected as the inputs to the decoder. The 7445 open-collector outputs  $\bar{O}_3$  and  $\bar{O}_6$  are used to switch relays  $K_1$  and  $K_2$  on and off. For instance, when  $\bar{O}_3$  is in its inactive HIGH state, its output transistor will be off (nonconducting) so that no current can flow through relay  $K_1$  and it will be deenergized. When  $\bar{O}_3$  is in its active-LOW state, its output transistor is on and acts as a current sink for current through  $K_1$  so that  $K_1$  is energized. Note that the relays operate from +24 V. Also note the presence of the diodes across the relay coils; these protect the decoder's output transistors from the large "inductive kick" voltage that would be produced when coil current is stopped abruptly.

The timing diagram in Figure 9-6(b) shows the sequence of events. If we assume that the counter is in the 0000 state at time 0, then both outputs  $\bar{O}_3$  and  $\bar{O}_6$  are initially in the inactive HIGH state, where their output transistors are off and both relays are deenergized. As clock pulses are applied, the counter will be incremented once per second. On the NGT of the third pulse (time 3), the counter will go to the 0011 (3) state. This will activate decoder output  $\bar{O}_3$  and thereby energize  $K_1$ . On the NGT of the fourth pulse, the counter goes to the 0100 (4) state. This will deactivate  $\bar{O}_3$  and deenergize relay  $K_1$ .

Similarly, at time 6, the counter will go to the 0110 (6) state; this will make  $\bar{O}_6 = 0$  and energize  $K_2$ . At time 7, the counter goes to 0111 (7) and deactivates  $\bar{O}_6$  to deenergize  $K_2$ .

The counter will continue counting as pulses are applied. After 16 pulses, the sequence just described will start over.

Decoders are widely used in the memory system of a computer where they respond to the address code generated by the central processor to activate a particular memory location. Each memory IC contains many registers that can store binary numbers (data). Each register needs to have its own unique address to distinguish it from all the other registers. A decoder is built into the memory IC's circuitry and allows a particular storage register to be activated when a unique combination of inputs (i.e., its address) is applied. In a system, there are usually several memory ICs combined to make up the entire storage capacity. A decoder is used to select a memory chip in response to a range of addresses by decoding the most significant bits of the system address and enabling (selecting) a particular chip. We will examine this application in Problem 9-63, and we will study it in much more depth when we read about memories in Chapter 12.

In more complicated memory systems, the memory chips are arranged in multiple banks that must be selected individually or simultaneously, depending on whether the microprocessor wants one or more bytes at a time. This means that under certain circumstances, more than one output of the decoder must be activated. For systems such as this, a programmable logic device is often used to implement the decoder because a simple 1-of-8 decoder alone is not sufficient. Programmable logic devices can be used easily for custom decoding applications.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Can more than one decoder output be activated at one time?
2. What is the function of a decoder's enable input(s)?
3. How does the 7445 differ from the 7442?
4. The 74154 is a 4-to-16 decoder with two active-LOW enable inputs. How many pins (including power and ground) does this IC have?

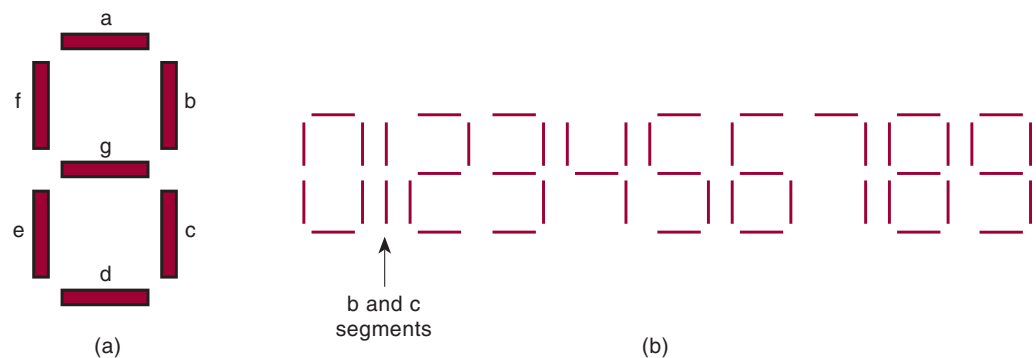
## 9-2 BCD-TO-7-SEGMENT DECODER/DRIVERS

### OUTCOMES

Upon completion of this section, you will be able to:

- Apply the decoding concept to driving individual LEDs arranged to form a decimal numeral.
- Distinguish between common anode and common cathode displays.
- Properly interface a decoder/driver to an LED display.

Most digital equipment has some means for displaying information in a form that can be understood readily by the user or operator. This information is often numerical data but can also be alphanumeric (numbers and letters). One of the simplest and most popular methods for displaying numerical digits uses a 7-segment configuration [Figure 9-7(a)] to form the decimal characters 0 through 9 and sometimes the hex characters A through F. One common arrangement uses light-emitting diodes (LEDs) for each segment. By controlling the current through each LED, some segments will be turned on and emit light while others will be turned off so that the desired character pattern will be generated. Figure 9-7(b) shows the segment patterns that are used to display the various digits. For example, to display a “6,” the segments *a*, *c*, *d*, *e*, *f*, and *g* are made bright while segment *b* is not lit and appears blank.



**FIGURE 9-7** (a) 7-segment arrangement; (b) active segments for each digit.

A **BCD-to-7-segment decoder/driver** is used to take a four-bit BCD input and provide the outputs that will pass current through the appropriate segments to display the decimal digit. The logic for this decoder is more complicated than the logic of decoders that we have looked at previously because each output is activated for more than one combination of inputs. For example, the *e* segment must be activated for any of the digits 0, 2, 6, and 8, which means whenever any of the codes 0000, 0010, 0110, or 1000 occurs.

**FIGURE 9-8** (a) BCD-to-7-segment decoder/driver driving a common-anode 7-segment LED display; (b) segment patterns for all possible input codes.

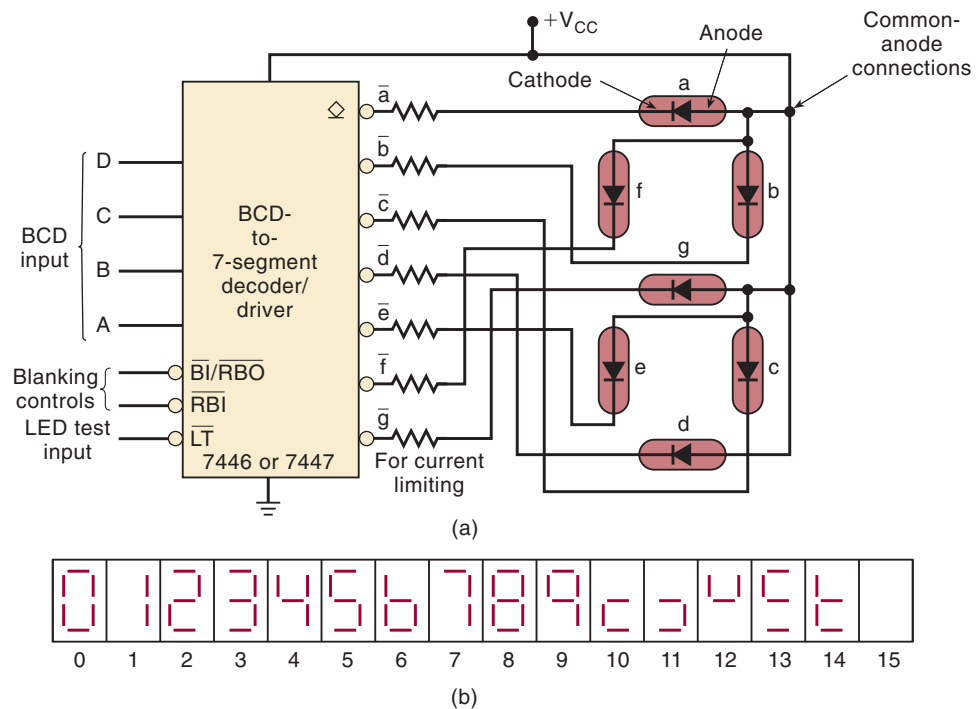


Figure 9-8(a) shows a BCD-to-7-segment decoder/driver (TTL 7446 or 7447) being used to drive a 7-segment LED readout. Each segment consists of an LED (light-emitting diode). Diodes are solid-state devices that allow current to flow through them in one direction, but block the flow in the other direction. Whenever the anode of an LED is more positive than the cathode by approximately 2 V, the LED will light up. The anodes of the LEDs are all tied to  $V_{CC}$  (+5 V). The cathodes of the LEDs are connected through current-limiting resistors to the appropriate outputs of the decoder/driver. The decoder/driver has active-LOW outputs that are open-collector driver transistors and can sink a fairly large current because LED readouts may require 10 to 40 mA per segment, depending on their type and size.

To illustrate the operation of this circuit, let us suppose that the BCD input is  $D = 0$ ,  $C = 1$ ,  $B = 0$ ,  $A = 1$ , which is BCD for 5. With these inputs, the decoder/driver outputs  $\bar{a}$ ,  $\bar{f}$ ,  $\bar{g}$ ,  $\bar{c}$ , and  $\bar{d}$  will be driven LOW (connected to ground), allowing current to flow through the  $a$ ,  $f$ ,  $g$ ,  $c$ , and  $d$  LED segments and thereby displaying the numeral 5. The  $\bar{b}$  and  $\bar{e}$  outputs will be HIGH (open), so that LED segments  $b$  and  $e$  cannot conduct.

The 7446/47 decoder/drivers are designed to activate specific segments even for non-BCD input codes (greater than 1001). Figure 9-8(b) shows the activated segment patterns for all possible input codes from 0000 to 1111. Note that an input code of 1111 (15) will blank out all the segments.

Seven-segment decoder/drivers such as the 7446/47 are exceptions to the rule that decoder circuits activate only one output for each combination of inputs. Rather, they activate a unique pattern of outputs for each combination of inputs.

### Common-Anode Versus Common-Cathode LED Displays

The LED display used in Figure 9-8 is a **common-anode** type because the anodes of all of the segments are tied together to  $V_{CC}$ . Another type of 7-segment LED display uses a **common-cathode** arrangement where the

cathodes of all of the segments are tied together and connected to ground. This type of display must be driven by a BCD-to-7-segment decoder/driver with active-HIGH outputs that apply a HIGH voltage to the anodes of those segments that are to be activated. Because each segment requires 10 to 20 mA of current to light it, TTL and CMOS devices are normally not used to drive the common-cathode display directly. Recall from Chapter 8 that TTL and CMOS outputs are not able to source large amounts of current. A transistor interface circuit is often used between decoder chips and the common-cathode display.

**EXAMPLE 9-4**

Each segment of a typical 7-segment LED display is rated to operate at 10 mA at 2.7 V for normal brightness. Calculate the value of the current-limiting resistor needed to produce approximately 10 mA per segment.

**Solution**

Referring to Figure 9-8(a), we can see that the series resistor must have a voltage drop equal to the difference between  $V_{CC} = 5\text{ V}$  and the segment voltage of 2.7 V. This 2.3 V across the resistor must produce a current of about 10 mA. Thus, we have

$$R_S = \frac{2.3\text{ V}}{10\text{ mA}} = 230\ \Omega$$

A standard resistor value close to this can be used. A 220- $\Omega$  resistor would be a good choice.

**OUTCOME ASSESSMENT QUESTIONS**

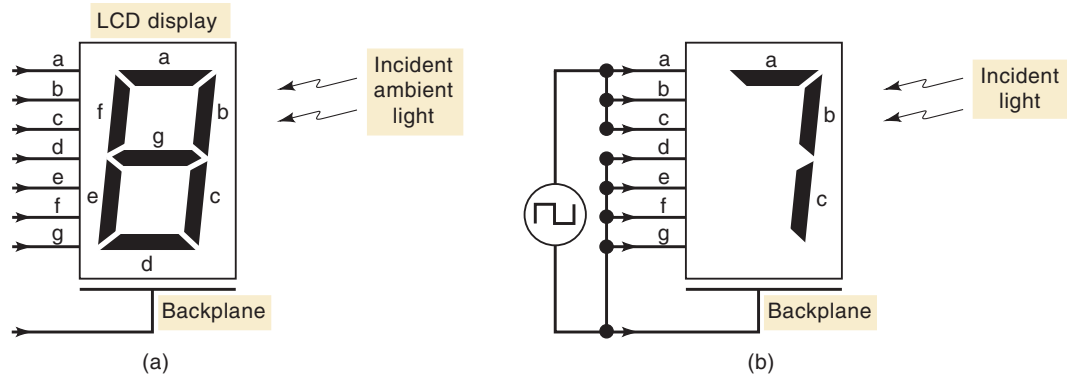
1. Which LED segments will be on for a decoder/driver input of 1001?
2. *True or false:* More than one output of a BCD-to-7-segment decoder/driver can be active at one time.

**9-3 LIQUID-CRYSTAL DISPLAYS****OUTCOMES**

*Upon completion of this section, you will be able to:*

- Describe the operation of an LCD.
- Distinguish between reflective and backlit LCDs.
- Define terms associated with LCD display technology.
- Describe the principles of operation of color LCD technology.

An LED display generates or emits light energy as current is passed through the individual segments. A liquid-crystal display (**LCD**) controls the reflection of available light. The available light may simply be ambient (surrounding) light such as sunlight or normal room lighting; *reflective* LCDs use ambient light. Or the available light might be provided by a small light source that is part of the display unit; *backlit* LCDs use this method. In any case, LCDs have gained wide acceptance because of their very low power consumption compared to LEDs, especially in battery-operated equipment such as calculators, digital watches, and portable electronic measuring



**FIGURE 9-9** Liquid-crystal display: (a) basic arrangement; (b) applying a voltage between the segment and the backplane turns ON the segment. Zero voltage turns the segment OFF.

instruments. LEDs have the advantage of a much brighter display that, unlike reflective LCDs, is easily visible in dark or poorly lit areas.

Basically, LCDs operate from a low-voltage (typically 3 to 15 V rms), low-frequency (25 to 60 Hz) AC signal and draw very little current. They are often arranged as 7-segment displays for numerical readouts as shown in Figure 9-9(a). The AC voltage needed to turn on a segment is applied between the segment and the **backplane**, which is common to all segments. The segment and the backplane form a capacitor that draws very little current as long as the AC frequency is kept low. It is generally not lower than 25 Hz because this would produce visible flicker.

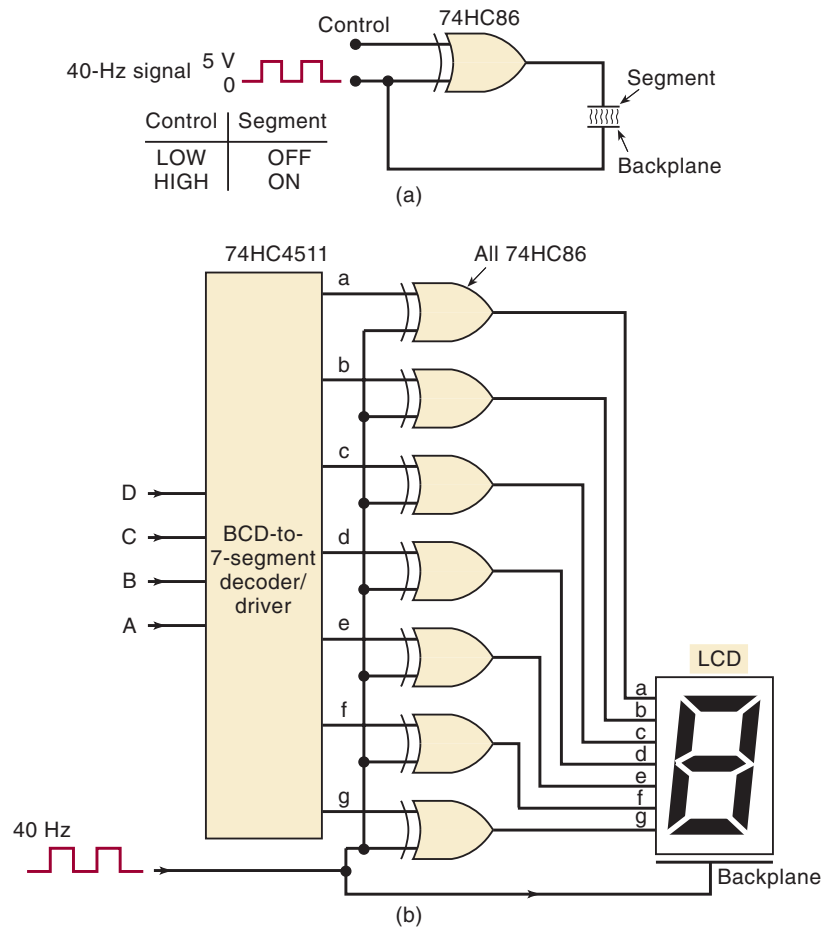
An admittedly simplified explanation of how an LCD operates goes something like this. When there is no difference in voltage between a segment and the backplane, the segment is said to be *nonactivated* (OFF). Segments *d*, *e*, *f*, and *g* in Figure 9-9(b) are OFF and will reflect incident light so that they appear invisible against their background. When an appropriate AC voltage is applied between a segment and the backplane, the segment is activated (ON). Segments *a*, *b*, and *c* in Figure 9-9(b) are ON and will not reflect the incident light, and thus they appear dark against their background.

### Driving an LCD

An LCD segment will turn ON when an AC voltage is applied between the segment and the backplane and will turn OFF when there is no voltage between the two. Rather than generating an AC signal, it is common practice to produce the required AC voltage by applying out-of-phase square waves to the segment and the backplane. This is illustrated in Figure 9-10(a) for one segment. A 40-Hz square wave is applied to the backplane and also to the input of a CMOS 74HC86 XOR. The other input to the XOR is a CONTROL input that will control whether the segment is ON or OFF.

When the CONTROL input is LOW, the XOR output will be exactly the same as the 40-Hz square wave, so that the signals applied to the segment and the backplane are equal. Because there is no difference in voltage, the segment will be OFF. When the CONTROL input is HIGH, the XOR output will be the INVERSE of the 40-Hz square wave, so that the signal applied to the segment is out of phase with the signal applied to the backplane. As a result, the segment voltage will alternately be at +5 V and at -5 V relative to the backplane. This AC voltage will turn ON the segment.

**FIGURE 9-10** (a) Method for driving an LCD segment; (b) driving a 7-segment display.



This same idea can be extended to a complete 7-segment LCD display, as shown in Figure 9-10(b). Here, the CMOS 74HC4511 BCD-to-7-segment decoder/driver supplies the CONTROL signals to each of seven XOR for the seven segments. The 74HC4511 has active-HIGH outputs because a HIGH is required to turn on a segment. The decoder/driver and XOR gates of Figure 9-10(b) are available on a single chip. The CMOS 74HC4543 is one such device. It takes the BCD input code and provides the outputs to drive the LCD segments directly.

In general, CMOS devices are used to drive LCDs for two reasons: (1) they require much less power than TTL and are more suited to the battery-operated applications where LCDs are used; (2) the TTL LOW-state voltage is not exactly 0 V and can be as much as 0.4 V. This will produce a DC component of voltage between the segment and the backplane that considerably shortens the life of an LCD.

## Types of LCDs

Liquid crystals are available as multidigit 7-segment decimal numeric displays. They come in many sizes and with many special characters such as colons (:) for clock displays, + and - indicators for digital voltmeters, decimal points for calculators, and battery-low indicators because many LCD devices are battery-powered. These displays must be driven by a decoder/driver chip such as the 74HC4543.

A more complicated but readily available LCD display is the alpha-numeric LCD module. These modules are available from many companies



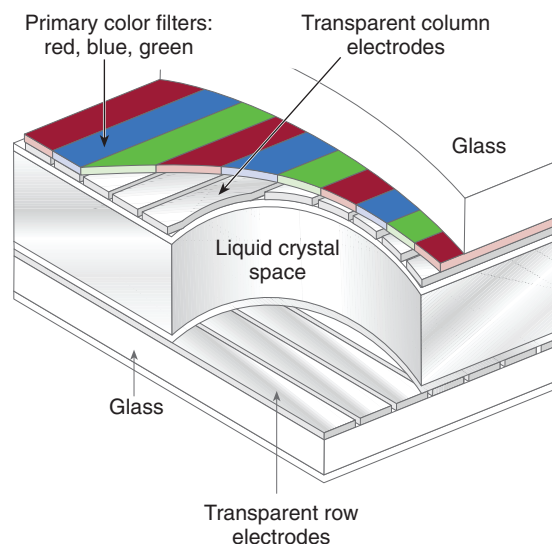
in numerous formats such as 1-line-by-16-characters up to 4-lines-by-40-characters. The interface to these modules has been standardized so that an LCD module from any manufacturer will use the same signals and data format. The module includes some VLSI chips that make this device simple to use. Eight data lines are used to send the ASCII code for whatever you wish to display. These data lines also carry special control codes to the LCD command register. Three other inputs (Register Select, Read/Write, and Enable) are used to control the location, direction, and timing of the data transfer. As characters are sent to the module, it stores them in its own memory and types them across the display screen.

Other LCD modules allow the user to create a graphical display by controlling individual dots on the screen called **pixels**. Larger LCD panels can be scanned at a high rate, producing high-quality video motion pictures. In these displays, the control lines are arranged in a grid of rows and columns. At the intersection of each row and column is a pixel that acts like a “window” or “shutter” that can be electronically opened and closed to control the amount of light that is transmitted through the cell. The voltage from a row to a column determines the brightness of each pixel. In a laptop computer, a binary number for each pixel is stored in the “video” memory. These numbers are converted to voltages that are applied to the display.

Each pixel on a color display is actually made up of three subpixels. These subpixels control the light that passes through a red, green, or blue filter to produce the color of each pixel. On a 640-by-480 LCD screen there would be  $640 \times 3$  connections for columns and 480 connections for rows, for a total of 2400 connections to the LCD. Obviously, the driver circuitry for such a device is a very complicated VLSI circuit.

The advances in technology for LCD displays have increased the speed at which the pixels can be turned on and off. The older screens are called Twisted Nematic (TN) or Super Twisted Nematic (STN). These devices are referred to as passive LCDs. Instead of using a uniform backplane like the 7-segment LCD displays, they have conducting parallel lines manufactured onto two pieces of glass. The two glass sheets are used to sandwich the liquid crystal material with the conducting lines at  $90^\circ$ , forming a grid of rows and columns, as shown in Figure 9-11. The intersection of each row and column

**FIGURE 9-11** A passive matrix LCD panel.



forms a pixel. The actual switching of the current on and off is done in the driver IC that is connected to the rows and columns of the display. Passive matrix displays are rather slow at turning off. This limits the rate at which objects can move on the screen without leaving a shadow trail behind them.

The newer displays are called active matrix TFT LCDs. The active matrix means that an active element on the display is used to switch the pixels on and off. The active component is a thin film transistor (TFT) that is manufactured directly onto one piece of glass. The other piece of glass has a uniform coating to form a backplane. The control lines for these transistors run in rows and columns between the pixels. The technology that allows these transistors to be manufactured in a matrix on a thin film the size of a laptop computer screen has made these displays possible. They provide a much faster-response, higher-resolution display. The use of polysilicon technology allows the driver circuits to be integrated into the display unit, reducing connection problems and requiring very little perimeter space around the LCD.

Other display technologies include vacuum fluorescent, gas discharge plasma, and electroluminescence. The optical physics for each of these displays varies, but the means of controlling all of them is the same. A digital system must activate a row and a column of a matrix in order to control the amount of light at the pixel located at the row/column intersection.

#### OUTCOME ASSESSMENT QUESTIONS

1. Indicate which of the following statements refer to LCD displays and which refer to LED displays.
  - (a) Emit light
  - (b) Reflect ambient light
  - (c) Are best for low-power applications
  - (d) Require an AC voltage
  - (e) Use a 7-segment arrangement to produce digits
  - (f) Require current-limiting resistors
2. What form of data is sent to each of the following?
  - (a) A 7-segment LCD display with a decoder/driver
  - (b) An alphanumeric LCD module
  - (c) An LCD computer display

## 9-4 ENCODERS

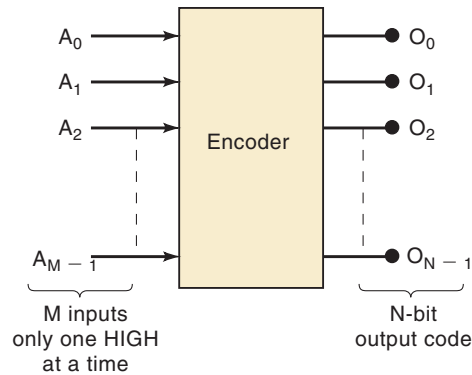
### OUTCOMES

*Upon completion of this section, you will be able to:*

- Define the term *encoder*.
- State the nature of the inputs and outputs of an encoder.
- Describe common applications of encoders.

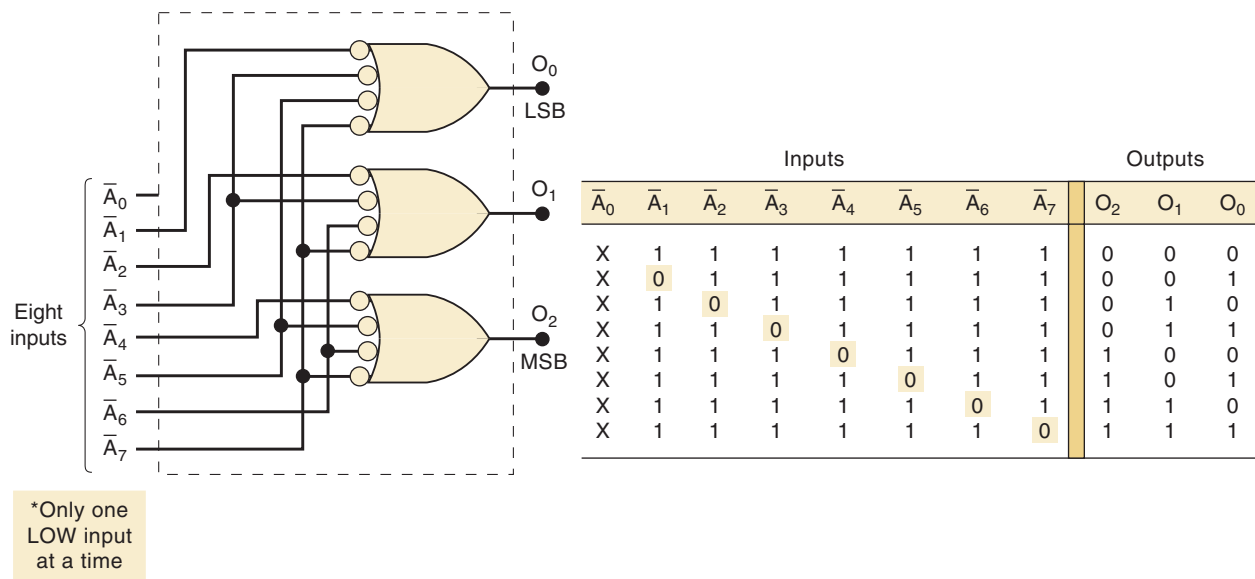
Most decoders accept an input code and produce a HIGH (or a LOW) at *one and only one* output line. In other words, we can say that a decoder identifies, recognizes, or detects a particular code. The opposite of this decoding process is called **encoding** and is performed by a logic circuit called an **encoder**. An encoder has a number of input lines, only one of which is activated at a

**FIGURE 9-12** General encoder diagram.



given time, and produces an  $N$ -bit output code, depending on which input is activated. Figure 9-12 is the general diagram for an encoder with  $M$  inputs and  $N$  outputs. Here, the inputs are active-HIGH, which means that they are normally LOW.

We saw that a *binary-to-octal decoder (3-line-to-8-line decoder)* accepts a three-bit input code and activates one of eight output lines corresponding to that code. An *octal-to-binary encoder (8-line-to-3-line encoder)* performs the opposite function: it accepts eight input lines and produces a three-bit output code corresponding to the activated input. Figure 9-13 shows the logic circuit and the truth table for an octal-to-binary encoder with active-LOW inputs.



**FIGURE 9-13** Logic circuit for an octal-to-binary (8-line-to-3-line) encoder. For proper operation, only one input should be active at one time.

By following through the logic, you can verify that a LOW at any single input will produce the output binary code corresponding to that input. For instance, a LOW at  $\bar{A}_3$  (while all other inputs are HIGH) will produce  $O_2 = 0, O_1 = 1$ , and  $O_0 = 1$ , which is the binary code for 3. Notice that  $\bar{A}_0$  is not connected to the logic gates because the encoder outputs will normally be at 000 when none of the inputs  $\bar{A}_1$  to  $\bar{A}_7$  is LOW.

## EXAMPLE 9-5

Determine the outputs of the encoder in Figure 9-13 when  $\bar{A}_3$  and  $\bar{A}_5$  are simultaneously LOW.

**Solution**

Following through the logic gates, we see that the LOWs at these two inputs will produce HIGHs at each output, in other words, the binary code 111. Clearly, this is not the code for either activated input.

**Priority Encoders**

This last example identifies a drawback of the simple encoder circuit of Figure 9-13 when more than one input is activated at one time. A modified version of this circuit, called a **priority encoder**, includes the necessary logic to ensure that when two or more inputs are activated, the output code will correspond to the highest-numbered input. For example, when both  $\bar{A}_3$  and  $\bar{A}_5$  are LOW, the output code will be 101 (5). Similarly, when  $\bar{A}_6$ ,  $\bar{A}_2$ , and  $\bar{A}_0$  are all LOW, the output code is 110 (6). The 74148, 74LS148, and 74HC148 are all octal-to-binary priority encoders.

**74147 Decimal-to-BCD Priority Encoder**

Figure 9-14 shows the logic symbol and the truth table for the 74147 (74LS147, 74HC147), which functions as a decimal-to-BCD priority encoder. It has nine active-LOW inputs representing the decimal digits 1 through 9, and it produces the *inverted* BCD code corresponding to the highest-numbered activated input.

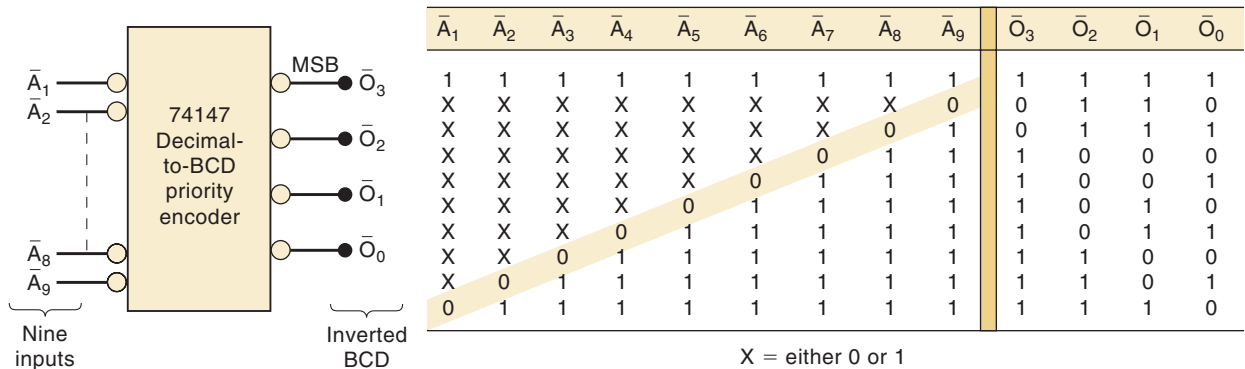


FIGURE 9-14 74147 decimal-to-BCD priority encoder.



Let's examine the truth table to see how this IC works. The first line in the table shows all inputs in their inactive HIGH state. For this condition, the outputs are 1111, which is the inverse of 0000, the BCD code for 0. The second line in the table indicates that a LOW at  $\bar{A}_9$ , regardless of the states of the other inputs, will produce an output code of 0110, which is the inverse of 1001, the BCD code for 9. The third line shows that a LOW at  $\bar{A}_8$ , provided that  $\bar{A}_9$  is HIGH, will produce an output code of 0111, the inverse of 1000, the BCD code for 8. In a similar manner, the remaining lines in the table show that a LOW at any input, provided that all higher-numbered inputs are HIGH, will produce the inverse of the BCD code for that input.

The 74147 outputs will normally be HIGH when none of the inputs are activated. This corresponds to the decimal 0 input condition. There is no  $\bar{A}_0$  input because the encoder assumes the decimal 0 input state when all other inputs are HIGH. The 74147 inverted BCD outputs can be converted to normal BCD by putting each one through an INVERTER.

### EXAMPLE 9-6

Determine the states of the outputs in Figure 9-14 when  $\bar{A}_5$ ,  $\bar{A}_7$ , and  $\bar{A}_3$  are LOW and all other inputs are HIGH.

### Solution

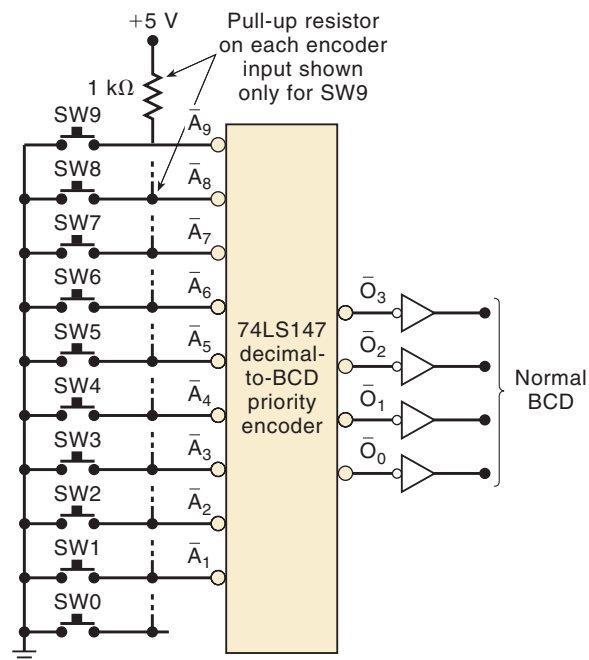
The truth table shows that when  $\bar{A}_7$  is LOW, the levels at  $\bar{A}_5$  and  $\bar{A}_3$  do not matter. Thus, the outputs will each be 1000, the inverse of 0111 (7).

## Switch Encoder

Figure 9-15 shows how a 74147 can be used as a *switch encoder*. The 10 switches might be the keyboard switches on a calculator representing digits 0 through 9. The switches are of the normally open type, so that the encoder inputs are all normally HIGH and the BCD output is 0000 (note the INVERTERS). When a digit key is depressed, the circuit will produce the BCD code for that digit. Because the 74LS147 is a priority encoder, simultaneous key depressions will produce the BCD code for the higher-numbered key.

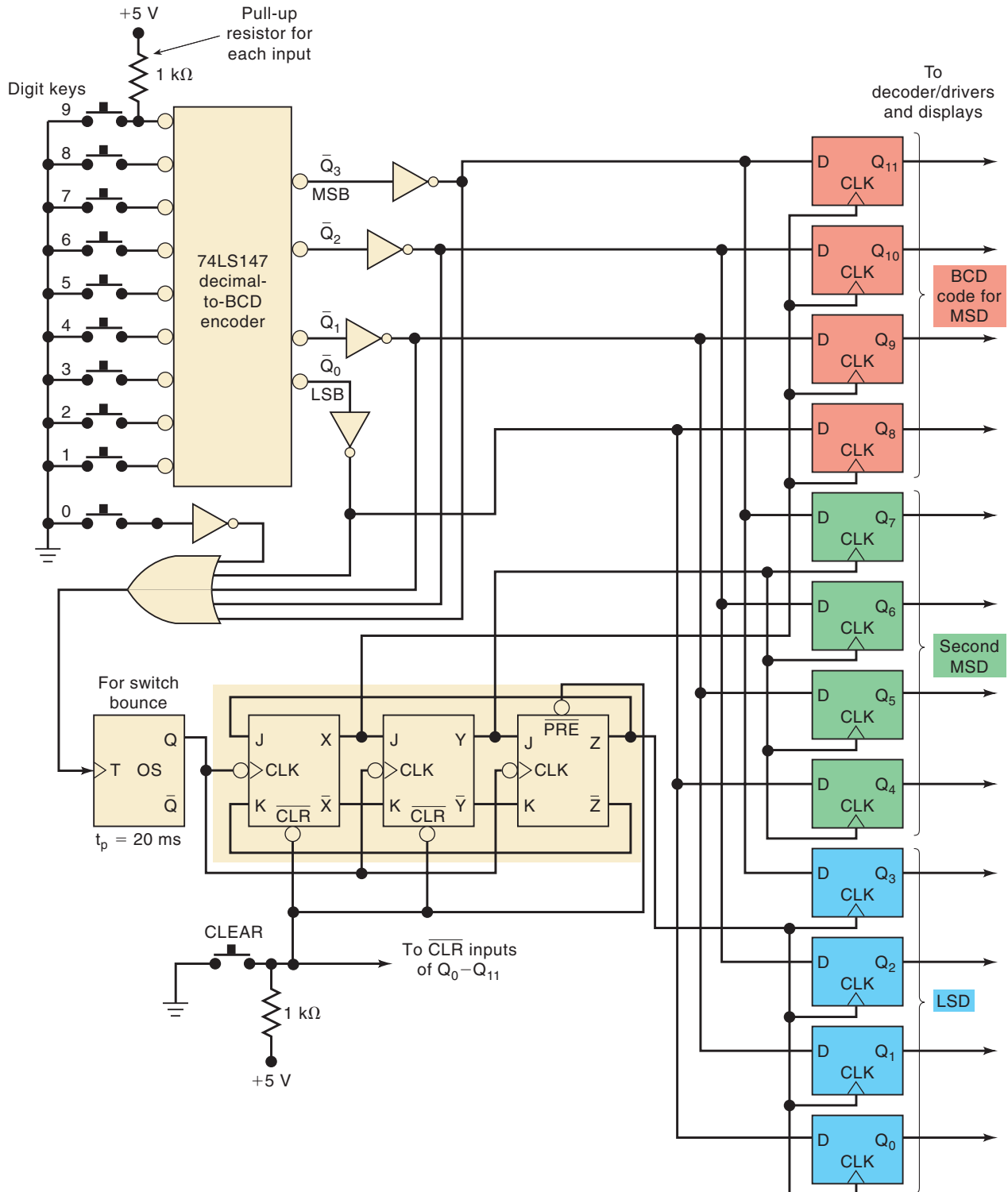
The switch encoder of Figure 9-15 can be used whenever BCD data must be entered manually into a digital system. A prime example would be in an electronic calculator, where the operator depresses several keyboard switches in succession to enter a decimal number. In a simple, basic calculator, the BCD code for each decimal digit is entered into a four-bit storage register. In other words, when the first key is depressed, the BCD code for that digit is sent to a four-bit FF register; when the second switch is depressed, the BCD code for that digit is sent to *another* four-bit FF register, and so on. Thus, a calculator

**FIGURE 9-15** Decimal-to-BCD switch encoder.



that can handle eight digits will have eight four-bit registers to store the BCD codes for these digits. Each four-bit register drives a decoder/driver and a numerical display so that the eight-digit number can be displayed.

The operation described above can be accomplished with the circuit in Figure 9-16. This circuit will take three decimal digits entered from the



**FIGURE 9-16** Circuit for keyboard entry of three-digit number into storage registers.

keyboard in sequence, encode them in BCD, and store the BCD in three FF output registers. The 12 D-type flip-flops  $Q_0$  to  $Q_{11}$  are used to receive and store the BCD codes for the digits.  $Q_8$  to  $Q_{11}$  store the BCD code for the most significant digit (MSD), which is the first one entered on the keyboard.  $Q_4$  to  $Q_7$  store the second entered digit, and  $Q_0$  to  $Q_3$  store the third entered digit. Flip-flops X, Y, and Z form a ring counter (Chapter 7) that controls the transfer of data from the encoder outputs to the appropriate output register. The OR gate produces a HIGH output any time one of the keys is depressed. This output may be affected by switch contact bounce, which would produce several pulses before settling down to the HIGH state. The OS is used to neutralize the switch bounce by triggering on the first positive transition from the OR gate and remaining HIGH for 20 ms, well past the time duration of the switch bounce. The OS output clocks the ring counter.

The circuit operation is described as follows for the case where the decimal number 309 is being entered:

1. The CLEAR key is depressed. This clears all storage D flip-flops  $Q_0$  to  $Q_{11}$  to 0. It also clears flip-flops X and Y and presets flip-flop Z to 1, so that the ring counter begins in the 001 state.
2. The CLEAR key is released and the “3” key is depressed. The encoder outputs 1100 are inverted to produce 0011, the BCD code for 3. These binary values are sent to the  $D$  inputs of the three four-bit output registers.
3. The OR output goes HIGH (because two of its inputs are HIGH) and triggers the OS output  $Q = 1$  for 20 ms. After 20 ms,  $Q$  returns LOW and clocks the ring counter to the 100 state (output X goes HIGH). The positive transition at output X is fed to the  $CLK$  inputs of the D flip-flops  $Q_8$  to  $Q_{11}$ , so that the encoder outputs are transferred to these FFs. That is, outputs  $Q_{11} = 0$ ,  $Q_{10} = 0$ ,  $Q_9 = 1$ , and  $Q_8 = 1$ . Note that flip-flops  $Q_0$  to  $Q_7$  are not affected because their  $CLK$  inputs have not received a positive transition.
4. The “3” key is released and the OR gate output returns LOW. The “0” key is then depressed. This produces the BCD code of 0000, which is fed to the inputs of the three registers.
5. The OR output goes HIGH in response to the “0” key (note the INVERTER) and triggers the OS for 20 ms. After 20 ms, the ring counter shifts to the 010 state (output Y goes HIGH). The positive transition at Y is fed to the  $CLK$  inputs of flip-flops  $Q_4$  to  $Q_7$  and transfers the 0000 to these FFs. Note that flip-flops  $Q_0$  to  $Q_3$  and  $Q_8$  to  $Q_{11}$  are not affected by the Y transition.
6. The “0” key is released and the OR output returns LOW. The “9” key is depressed, producing BCD outputs 1001, which are fed to the storage registers.
7. The OR output goes HIGH again, triggering the OS, which in turn clocks the ring counter to the 001 state (output Z goes HIGH). The positive transition at Z is fed to the  $CLK$  inputs of flip-flops  $Q_0$  to  $Q_3$  and transfers the 1001 into these FFs. The other storage FFs are unaffected.
8. At this point, the storage register contains 001100001001, beginning with FF  $Q_{11}$ . This is the BCD code of 309. These register outputs feed decoder/drivers that drive appropriate displays for indicating the decimal digits 309.
9. The storage FF outputs are also fed to other circuits in the system. In a calculator, for example, these outputs would be sent to the arithmetic section to be processed.

TABLE 9-1 74ALS148 function table.

$\overline{EI}$	Inputs								Outputs			$\overline{GS}$	$\overline{EO}$
	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{6}$	$\overline{7}$	$\overline{A}_2$	$\overline{A}_1$	$\overline{A}_0$		
H	x	x	x	x	x	x	x	x	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	x	x	x	x	x	x	x	L	L	L	L	L	H
L	x	x	x	x	x	x	L	H	L	L	H	L	H
L	x	x	x	x	L	H	H	H	L	H	L	L	H
L	x	x	x	L	H	H	H	H	L	H	H	L	H
L	x	x	L	H	H	H	H	H	H	L	H	L	H
L	x	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

Several problems at the end of the chapter will deal with some other aspects of this circuit, including troubleshooting exercises.

The 74ALS148 is slightly more sophisticated than the '147. It has eight inputs that are encoded into a three-bit binary number. This IC also provides three control pins as indicated in Table 9-1. The Enable Input ( $\overline{EI}$ ) and Enable Output ( $\overline{EO}$ ) can be used to cascade two IC's producing a hexadecimal-to-binary encoder. The  $\overline{EI}$  pin must be LOW in order for any output pin to go LOW, and the  $\overline{EO}$  pin will only go LOW when none of the eight inputs is active and the  $\overline{EI}$  is active. The  $\overline{GS}$  output is used to indicate when at least one of the eight inputs is activated. It should be noted that the outputs  $A_2$  through  $A_0$  are inverted, just as in the 74147.

### OUTCOME ASSESSMENT QUESTIONS

1. How does an encoder differ from a decoder?
2. How does a priority encoder differ from an ordinary encoder?
3. What will the outputs be in Figure 9-15 when SW6, SW5, and SW2 are all closed?
4. Describe the functions of each of the following parts of the keyboard entry circuit of Figure 9-16.
  - (a) OR gate
  - (b) 74147 encoder
  - (c) One-shot
  - (d) Flip-flops X, Y, Z
  - (e) Flip-flops  $Q_0$  to  $Q_{11}$
5. What is the purpose of each control input and output on a 74148 encoder?

## 9-5 TROUBLESHOOTING

### OUTCOME

Upon completion of this section, you will be able to:

- Expand troubleshooting techniques and skills.

As circuits and systems become more complex, the number of possible causes of failure obviously increases. Whereas the procedure for fault



isolation and correction remains essentially the same, the application of the **observation/analysis** process is more important for complex circuits because it helps the troubleshooter narrow the location of the fault to a small area of the circuit. This reduces to a reasonable amount the testing steps and resulting data that must be analyzed. By understanding the circuit operation, observing the symptoms of the failure, and reasoning through the operation, the troubleshooter can often predict the possible faults before ever picking up a logic probe or an oscilloscope. This observation/analysis process is one that inexperienced troubleshooters are hesitant to apply, probably because of the great variety and capabilities of modern test equipment available to them. It is easy to become overly reliant on these tools while not adequately utilizing the human brain's reasoning and analytical skills.

The following examples illustrate how the observation/analysis process can be applied. Many of the end-of-chapter troubleshooting problems will provide you with the opportunity to develop your skill at applying this process.

Another vital strategy in troubleshooting is known as **divide-and-conquer**. It is used to identify the location of the problem after observation/analysis has generated several possibilities. A less efficient method would be to investigate each possible cause, one by one. The divide-and-conquer method finds a point in the circuit that can be tested, thereby dividing the total possible number of causes in half. In simple systems, this may seem unnecessary, but as complexity increases, the total number of possible causes also increases. If there are eight possible causes, then a test should be performed that eliminates four of them. The next test should eliminate two more, and the third test should identify the problem.

### EXAMPLE 9-7

A technician tests the circuit of Figure 9-4 by using a set of switches to apply the input code at  $A_4$  through  $A_0$ . She runs through each possible input code and checks the corresponding decoder output to see if it is activated. She observes that all of the odd-numbered outputs respond correctly, but all of the even-numbered outputs fail to respond when their code is applied. What are the most probable faults?

#### Solution

In a situation where so many outputs are failing, it is unreasonable to expect that each of these outputs has a fault. It is much more likely that some faulty input condition is causing the output failures. What do all of the even-numbered outputs have in common? The input codes for several of them are listed in Table 9-2.

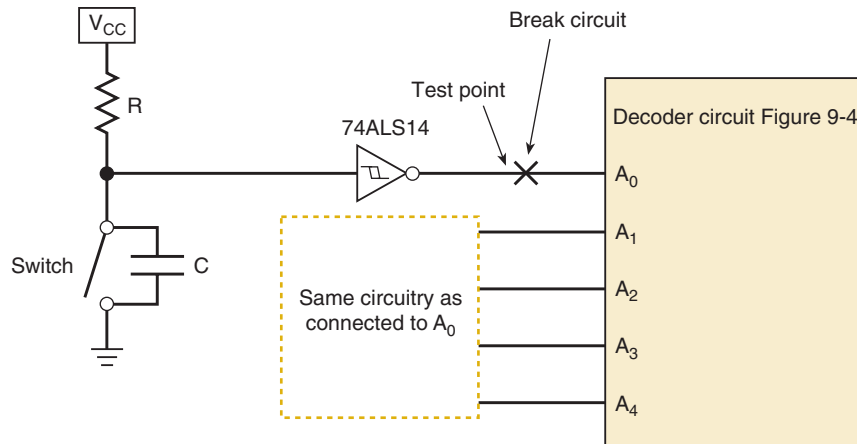
Clearly, each even-numbered output requires an input code with an  $A_0 = 0$  in order to be activated. Thus, the most probable faults would be those that prevent  $A_0$  from going LOW. These include:

1. A faulty switch connected to the  $A_0$  input
2. A break in the path between the switch and the  $A_0$  line
3. An external short from the  $A_0$  line to  $V_{CC}$
4. An internal short to  $V_{CC}$  at the  $A_0$  inputs of any one of the decoder chips

**TABLE 9-2** Data for Example 9-7.

Output	Input Code
$\bar{O}_0$	00000
$\bar{O}_4$	00100
$\bar{O}_{14}$	01110
$\bar{O}_{18}$	10010

**FIGURE 9-17**  
Troubleshooting circuitry  
in Example 9-7.



Through observation and analysis, the technician has identified several possible causes. Potential causes 1 and 2 are in the switches generating the address. Causes 3 and 4 are in the decoder circuit itself. The circuit can be divided by opening the connection between the least significant switch and the  $A_0$  input, as shown in Figure 9-17. A logic probe can be used to see if the switch can generate a LOW as well as a HIGH. Regardless of the outcome, two of the four possible causes have been eliminated.

Thus, the fault is narrowed to a specific area of the circuit. The exact fault can be traced with the testing and measurement techniques that we are already familiar with.

### EXAMPLE 9-8

A technician wires the outputs from a BCD counter to the inputs of the decoder/driver of Figure 9-8. He applies pulses to the counter at a very slow rate and observes the LED display, which is shown below, as the counter counts up from 0000 to 1001. Examine this observed sequence carefully and try to predict the most probable fault.

COUNT	0	1	2	3	4	5	6	7	8	9
Observed display	0	1	9	3	4	5	6	7	8	7
Expected display	0	1	2	3	4	5	6	7	8	9

### Solution

Comparing the observed display with the expected display for each count, we see several important points:

- For those counts where the observed display is incorrect, the observed display is not one of the segment patterns that correspond to counts greater than 1001.
- This rules out a faulty counter or faulty wiring from the counter to the decoder/driver.
- The correct segment patterns (0, 1, 3, 6, 7, and 8) have the common property that segments *e* and *f* are either both on or both off.

- The incorrect segment patterns have the common property that segments  $e$  and  $f$  are in opposite states, and if we interchange the states of these two segments, the correct pattern is obtained.

Giving some thought to these points should lead us to conclude that the technician has probably “crossed” the connections to the  $e$  and  $f$  segments.

### OUTCOME ASSESSMENT QUESTIONS

1. Name the two troubleshooting techniques described in this section.
2. In the first technique, describe what is observed and what is being analyzed.
3. In the second technique, describe what is being divided.

## 9-6 MULTIPLEXERS (DATA SELECTORS)

### OUTCOMES

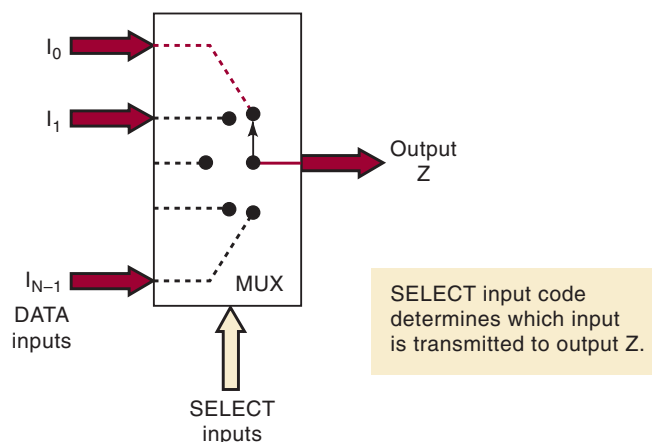
Upon completion of this section, you will be able to:

- Define the term *multiplexer*.
- State the nature of the inputs and outputs of a multiplexer.
- State the role of an “enable” input to a multiplexer.

A modern home stereo system may have a switch that selects music from one of four sources: an MP-3 player, a television tuner, a radio tuner, or audio from a DVD. The switch selects one of the electronic signals from one of these four sources and sends it to the power amplifier and speakers. In simple terms, this is what a **multiplexer (MUX)** does: it selects one of several input signals and passes it on to the output.

A *digital multiplexer* or *data selector* is a logic circuit that accepts several digital data inputs and selects one of them at any given time to pass on to the output. The routing of the desired data input to the output is controlled by **SELECT** inputs (often referred to as **ADDRESS** inputs). Figure 9-18 shows

**FIGURE 9-18** Functional diagram of a digital multiplexer (MUX).



the functional diagram of a general digital multiplexer. The inputs and outputs are drawn as wide arrows rather than lines; this indicates that they may actually be more than one signal line.

The multiplexer acts like a digitally controlled multiposition switch where the digital code applied to the SELECT inputs controls which data inputs will be switched to the output. For example, output  $Z$  will equal data input  $I_0$  for some particular SELECT input code,  $Z$  will equal  $I_1$  for another particular SELECT input code, and so on. Stated another way, a multiplexer selects 1 out of  $N$  input data sources and transmits the selected data to a single output channel. This is called **multiplexing**.

### Basic Two-Input Multiplexer

Figure 9-19 shows the logic circuitry for a two-input multiplexer with data inputs  $I_0$  and  $I_1$  and SELECT input  $S$ . The logic level applied to the  $S$  input determines which AND gate is enabled so that its data input passes through the OR gate to output  $Z$ . Looking at it another way, the Boolean expression for the output is

$$Z = I_0\bar{S} + I_1S$$

With  $S = 0$ , this expression becomes

$$\begin{aligned} Z &= I_0 \cdot 1 + I_1 \cdot 0 && \text{[gate 2 enabled]} \\ &= I_0 \end{aligned}$$

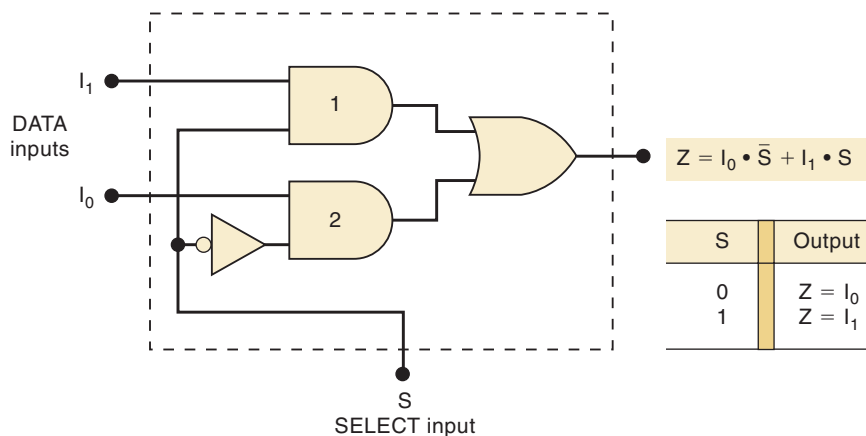
which indicates that  $Z$  will be identical to input signal  $I_0$ , which in turn can be a fixed logic level or a time-varying logic signal. With  $S = 1$ , the expression becomes

$$Z = I_0 \cdot 0 + I_1 \cdot 1 = I_1 \quad \text{[gate 1 enabled]}$$

showing that output  $Z$  will be identical to input signal  $I_1$ .

An example of where a two-input MUX could be used is in a digital system that uses two different MASTER CLOCK signals: a high-speed clock (say, 10 MHz) in one mode and a slow-speed clock (say, 4.77 MHz) for the other. Using the circuit of Figure 9-19, the 10-MHz clock would be tied to  $I_0$ , and the 4.77-MHz clock would be tied to  $I_1$ . A signal from the system's

**FIGURE 9-19** Two-input multiplexer.



control logic section would drive the SELECT input to control which clock signal appears at output Z for routing to the other parts of the circuit.

### Four-Input Multiplexer

The same basic idea can be used to form the four-input multiplexer shown in Figure 9-20(a). Here, four inputs are selectively transmitted to the output according to the four possible combinations of the  $S_1S_0$  select inputs. Each data input is gated with a different combination of select input levels.  $I_0$  is gated with  $\bar{S}_1\bar{S}_0$  so that  $I_0$  will pass through its AND gate to output Z only when  $S_1 = 0$  and  $S_0 = 0$ . The table in the figure gives the outputs for the other three input-select codes.

Another circuit that performs exactly the same function is shown in Figure 9-20(b). This approach uses tristate buffers to select one of the signals. The decoder ensures that only one buffer can be enabled at any time.  $S_1$  and  $S_0$  are used to specify which of the input signals is allowed to pass through its buffer and arrive at the output.

Two-, four-, eight-, and 16-input multiplexers are readily available in the TTL and CMOS logic families. These basic ICs can be combined for multiplexing a larger number of inputs.

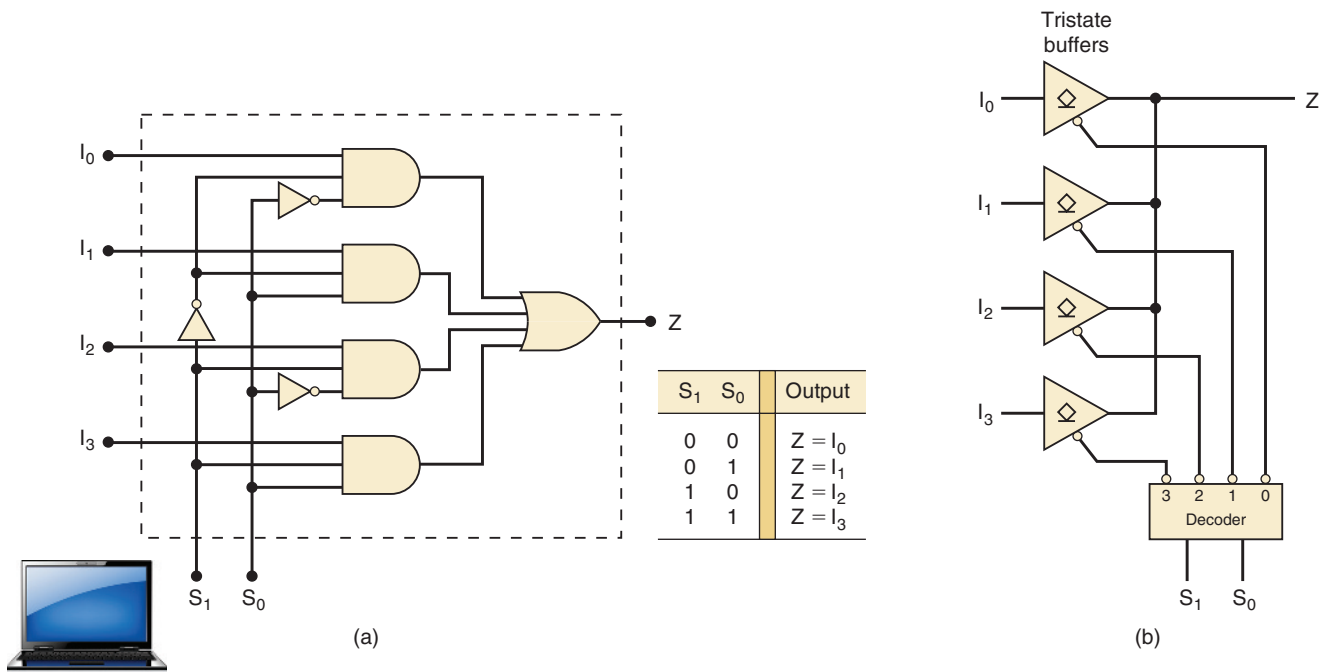
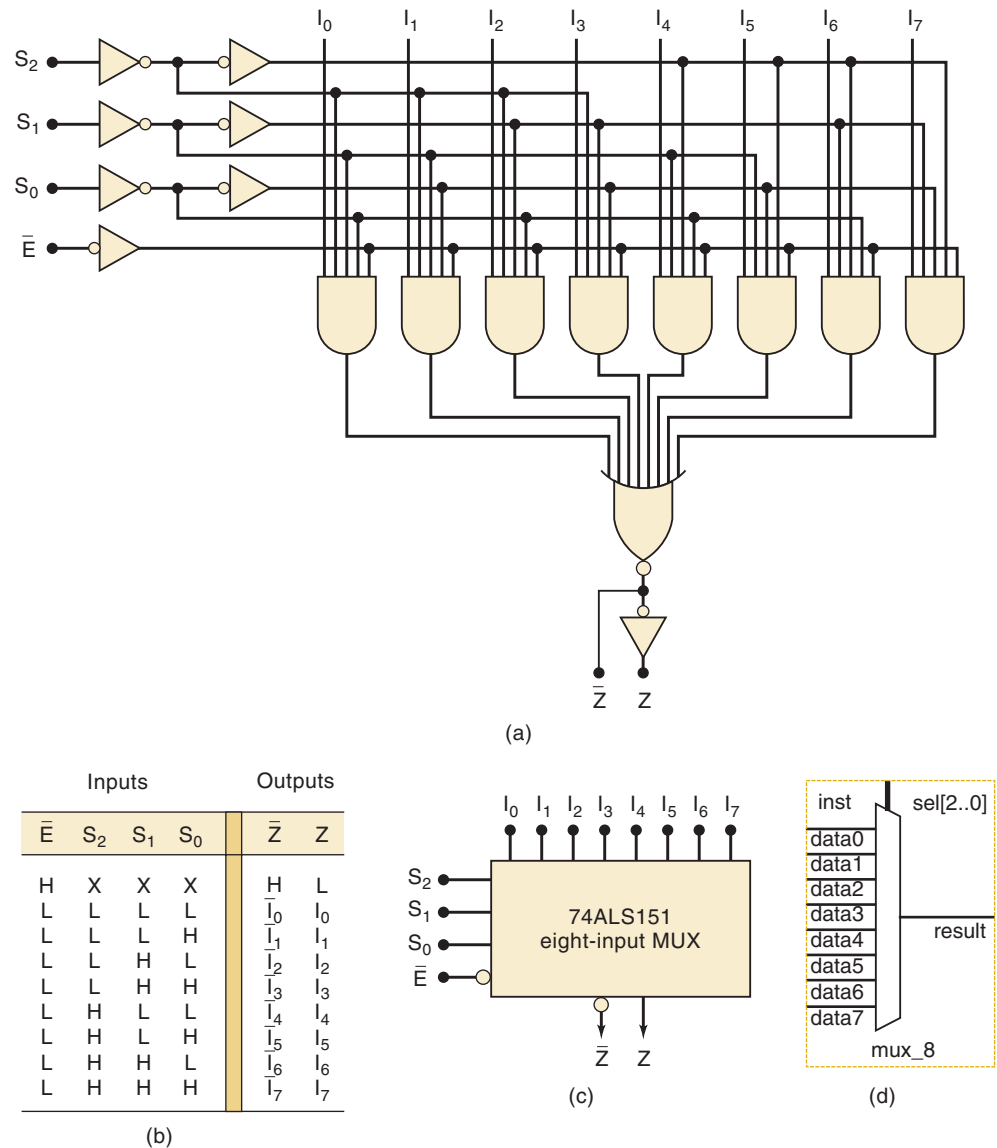


FIGURE 9-20 Four-input multiplexer: (a) using sum-of-products logic; (b) using tristate buffers.

### Eight-Input Multiplexer

Figure 9-21(a) shows the logic diagram for the 74ALS151 (74HC151) eight-input multiplexer. This multiplexer has an enable input  $\bar{E}$  and provides both the normal and the inverted outputs. When  $\bar{E} = 0$ , the select inputs  $S_2S_1S_0$  will select one data input (from  $I_0$  through  $I_7$ ) for passage to output Z. When  $\bar{E} = 1$ , the multiplexer is disabled so that  $Z = 0$  regardless of the select input code. This operation is summarized in Figure 9-21(b), and the 74151 logic symbol is shown in Figure 9-21(c). The symbol for an equivalent Altera megafunction is shown in Figure 9-21(d).



**FIGURE 9-21** (a) Logic diagram for the 74ALS151 multiplexer; (b) truth table; (c) logic symbol; (d) a similar MUX megafunction.

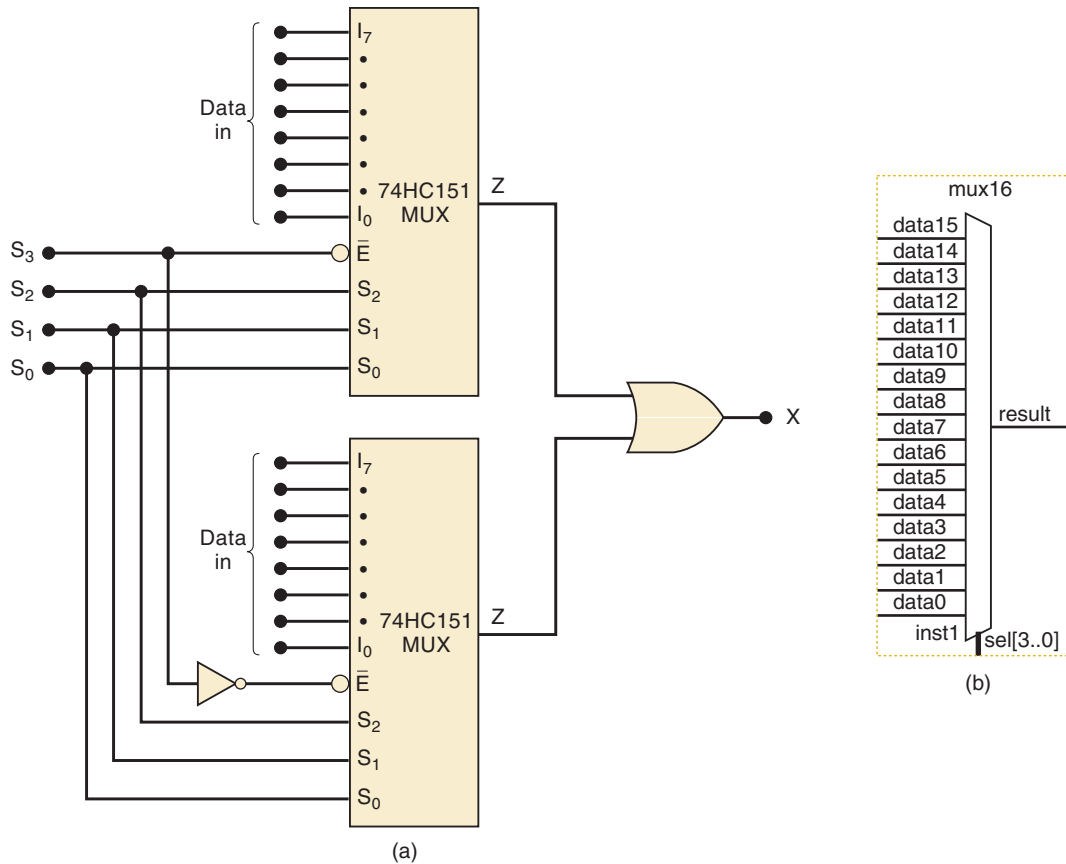
### EXAMPLE 9-9

The circuit in Figure 9-22(a) uses two 74HC151s, an INVERTER, and an OR gate. Describe this circuit's operation.

#### Solution

This circuit has a total of 16 data inputs, eight applied to each multiplexer. The two multiplexer outputs are combined in the OR gate to produce a single output  $X$ . The circuit functions as a 16-input multiplexer. The four select inputs  $S_3S_2S_1S_0$  will select one of the 16 inputs to pass through to  $X$ .

The  $S_3$  input determines which multiplexer is enabled. When  $S_3 = 0$ , the top multiplexer is enabled, and the  $S_2S_1S_0$  inputs determine which of its data inputs will appear at its output and pass through the OR gate to  $X$ . When  $S_3 = 1$ , the bottom multiplexer is enabled, and the  $S_2S_1S_0$  inputs select one of its data inputs for passage to output  $X$ .



**FIGURE 9-22** (a) Example 9-9: two 74HC151s combined to form a 16-input multiplexer; (b) a similar megafuction.

Figure 9-22(b) shows that the same functionality can be obtained by simply specifying more inputs for an Altera megafuction, rather than combining smaller modular blocks.

### Quad Two-Input MUX (74ALS157/HC157)

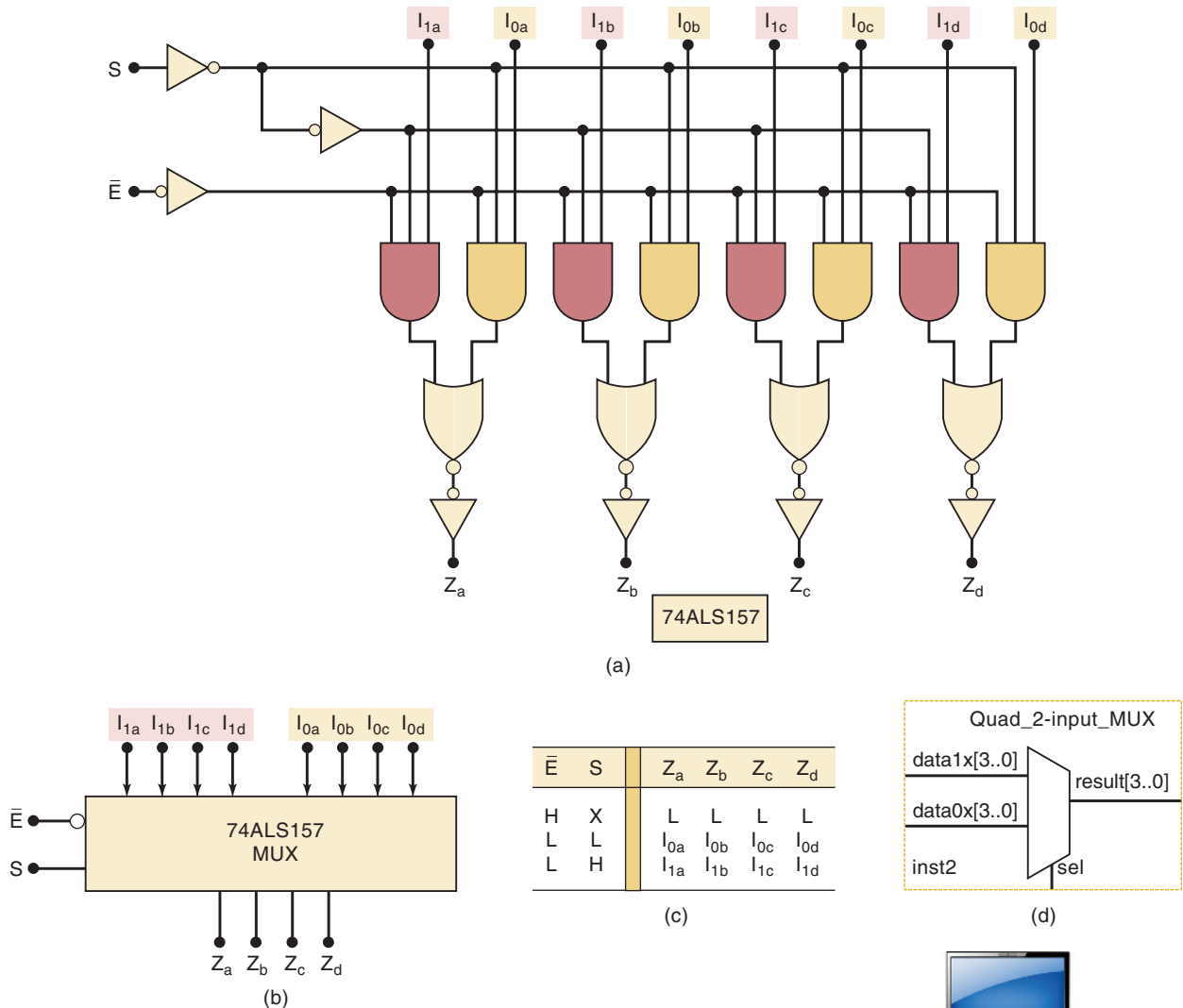
The 74ALS157 is a very useful multiplexer IC that contains four two-input multiplexers like the one in Figure 9-19. The logic diagram for the 74ALS157 is shown in Figure 9-23(a). Note the manner in which the data inputs and outputs are labeled. Subscripts *a*, *b*, *c*, and *d* represent the four bits of a binary number.  $I_1$  and  $I_0$  represent the two input numbers and  $Z$  represents the four-bit output number. Figure 9-23(b) shows the logic symbol and (c) is a function table that shows which signal is connected to the output based on  $S$ . Megafuctions can easily be created as shown in Figure 9-23(d).

**EXAMPLE 9-10**

Determine the input conditions required for each  $Z$  output in Figure 9-23 to take on the logic level of its corresponding  $I_0$  input. Repeat for  $I_1$ .

**Solution**

First of all, the enable input must be active; that is,  $\bar{E} = 0$ . In order for  $Z_a$  to equal  $I_{0a}$ , the select input must be LOW. These same conditions will produce  $Z_b = I_{0b}$ ,  $Z_c = I_{0c}$ , and  $Z_d = I_{0d}$ .



**FIGURE 9-23** (a) Logic diagram for the 74ALS157 multiplexer; (b) logic symbol; (c) truth table; (d) a MUX megafunction with two input channels of four bits.



With  $\bar{E} = 0$  and  $S = 1$ , the  $Z$  outputs will follow the set of  $I_1$  inputs; that is,  $Z_a = I_{1a}$ ,  $Z_b = I_{1b}$ ,  $Z_c = I_{1c}$ , and  $Z_d = I_{1d}$ .

All of the outputs will be disabled (LOW) when  $\bar{E} = 1$ .

It is helpful to think of this multiplexer as being a simple two-input multiplexer, but one in which each input is four lines and the output is four lines. The four output lines switch back and forth between the two sets of four input lines under the control of the select input.

### OUTCOME ASSESSMENT QUESTIONS

1. What is the function of a multiplexer's select inputs?
2. A certain multiplexer can switch one of 32 data inputs to its output. How many different inputs does this MUX have?



## 9-7 MULTIPLEXER APPLICATIONS

### OUTCOMES

Upon completion of this section, you will be able to:

- Identify common applications of multiplexers.
- Choose the appropriate configuration of a MUX for a given application.

Multiplexer circuits find numerous and varied applications in digital systems of all types. These applications include data selection, data routing, operation sequencing, parallel-to-serial conversion, waveform generation, and logic-function generation. We shall look at some of these applications here and several more in the problems at the end of the chapter.

### Data Routing

Multiplexers can route data from one of several sources to one destination. One typical application uses 74ALS157 multiplexers to select and display the contents of either of two BCD counters using a *single* set of decoder/drivers and LED displays. The circuit arrangement is shown in Figure 9-24.

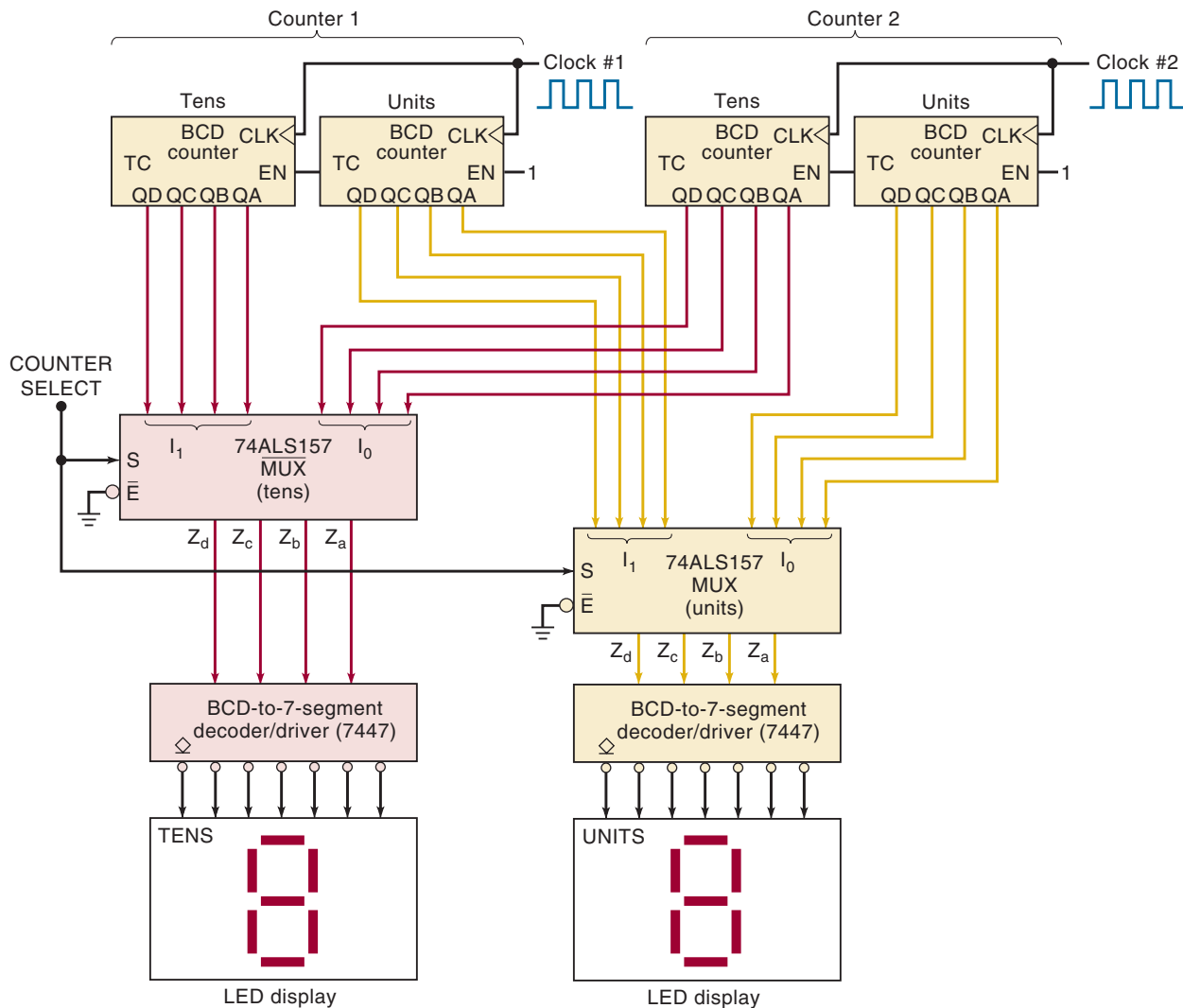


FIGURE 9-24 System for displaying two multidigit BCD counters one at a time.

Each counter consists of two cascaded BCD stages, and each one is driven by its own clock signal. When the COUNTER SELECT line is HIGH, the outputs of counter 1 will be allowed to pass through the multiplexers to the decoder/drivers to be displayed on the LED readouts. When COUNTER SELECT = 0, the outputs of counter 2 will pass through the multiplexers to the displays. In this way, the decimal contents of one counter or the other will be displayed under the control of the COUNTER SELECT input. A common situation where this might be used is in a digital watch. The digital watch circuitry contains many counters and registers that keep track of seconds, minutes, hours, days, months, alarm settings, and so on. A multiplexing scheme such as this one allows different data to be displayed on the limited number of decimal readouts.

The purpose of the multiplexing technique, as it is used here, is to *time-share* the decoder/drivers and display circuits between the two counters rather than have a separate set of decoder/drivers and displays for each counter. This results in a significant saving in the number of wiring connections, especially when more BCD stages are added to each counter. Even more important, it represents a significant decrease in power consumption because decoder/drivers and LED readouts typically draw relatively large amounts of current from the  $V_{CC}$  supply. Of course, this technique has the limitation that only one counter's contents can be displayed at a time. However, in many applications, this limitation is not a drawback. A mechanical switching arrangement could have been used to perform the function of switching first one counter and then the other to the decoder/drivers and displays, but the number of required switch contacts, the complexity of wiring, and the physical size could all be disadvantages over the completely logic method of Figure 9-24.

### Parallel-to-Serial Conversion

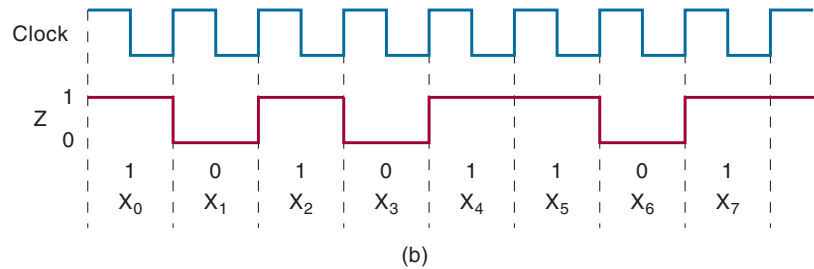
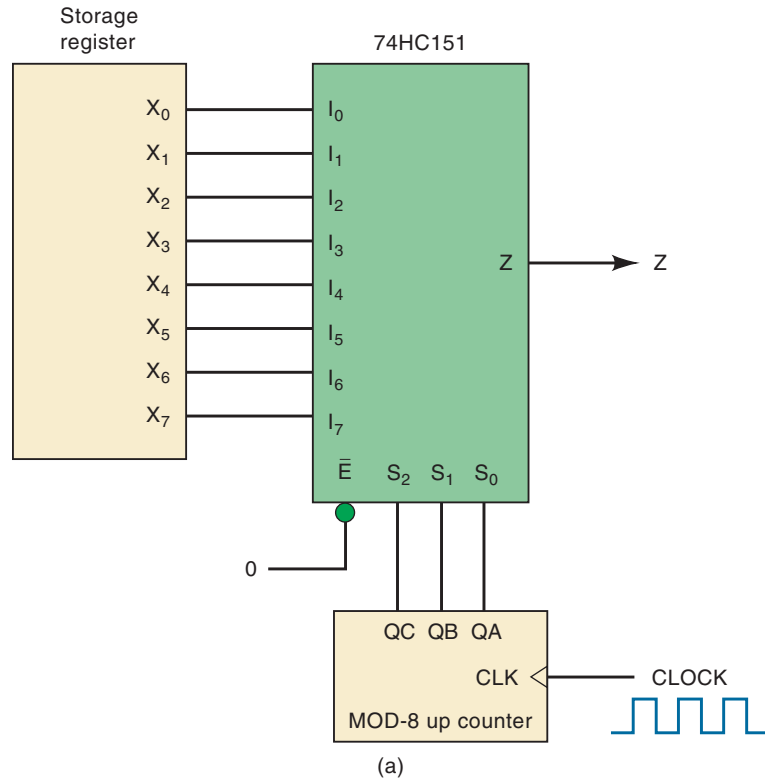
Many digital systems process binary data in parallel form (all bits simultaneously) because it is faster. When data are to be transmitted over relatively long distances, however, the parallel arrangement is undesirable because it requires a large number of transmission lines. For this reason, binary data or information in parallel form is often converted to serial form before being transmitted to a remote destination. One method for performing this **parallel-to-serial conversion** uses a multiplexer, as illustrated in Figure 9-25.

The data are present in parallel form at the outputs of the  $X$  register and are fed to the eight-input multiplexer. A three-bit (MOD-8) counter is used to provide the select code bits  $S_2S_1S_0$  so that they cycle through from 000 to 111 as clock pulses are applied. In this way, the output of the multiplexer will be  $X_0$  during the first clock period,  $X_1$  during the second clock period, and so on. The output  $Z$  is a waveform that is a serial representation of the parallel input data. The waveforms in the figure are for the case where  $X_7X_6X_5X_4X_3X_2X_1X_0 = 10110101$ . This conversion process takes a total of eight clock cycles. Note that  $X_0$  (the LSB) is transmitted first and the  $X_7$  (MSB) is transmitted last.

### Operation Sequencing

The circuit of Figure 9-26 uses an eight-input multiplexer as part of a control sequencer that steps through seven steps, each of which actuates some portion of the physical process being controlled. This could be, for example, a process that mixes two liquid ingredients and then cooks the mixture.

**FIGURE 9-25** (a) Parallel-to-serial converter; (b) waveforms for  $X_7X_6X_5X_4X_3X_2X_1X_0 = 10110101$ .



The circuit also uses a 3-line-to-8-line decoder and a MOD-8 binary counter. The operation is described as follows.

1. Initially the counter is reset to the 000 state. The counter outputs are fed to the select inputs of the multiplexer and to the inputs of the decoder. Thus, the decoder output  $\bar{O}_0 = 0$  and the others are all 1, so that all the ACTUATOR inputs of the process are LOW. The SENSOR outputs of the process all start out LOW. The multiplexer output  $\bar{Z} = \bar{I}_0 = 1$  because the S inputs are 000.
2. The START pulse initiates the sequencing operation by setting flip-flop  $Q_0$  HIGH, bringing the counter to the 001 state. This causes decoder output  $\bar{O}_1$  to go LOW, thereby activating actuator 1, which is the first step in the process (opening fill valve 1).
3. Some time later, SENSOR output 1 goes HIGH, indicating the completion of the first step (the float switch indicates that the tank is full). This HIGH is now present at the  $I_1$  input of the multiplexer. It is inverted and reaches the  $\bar{Z}$  output because the select code from the counter is 001.

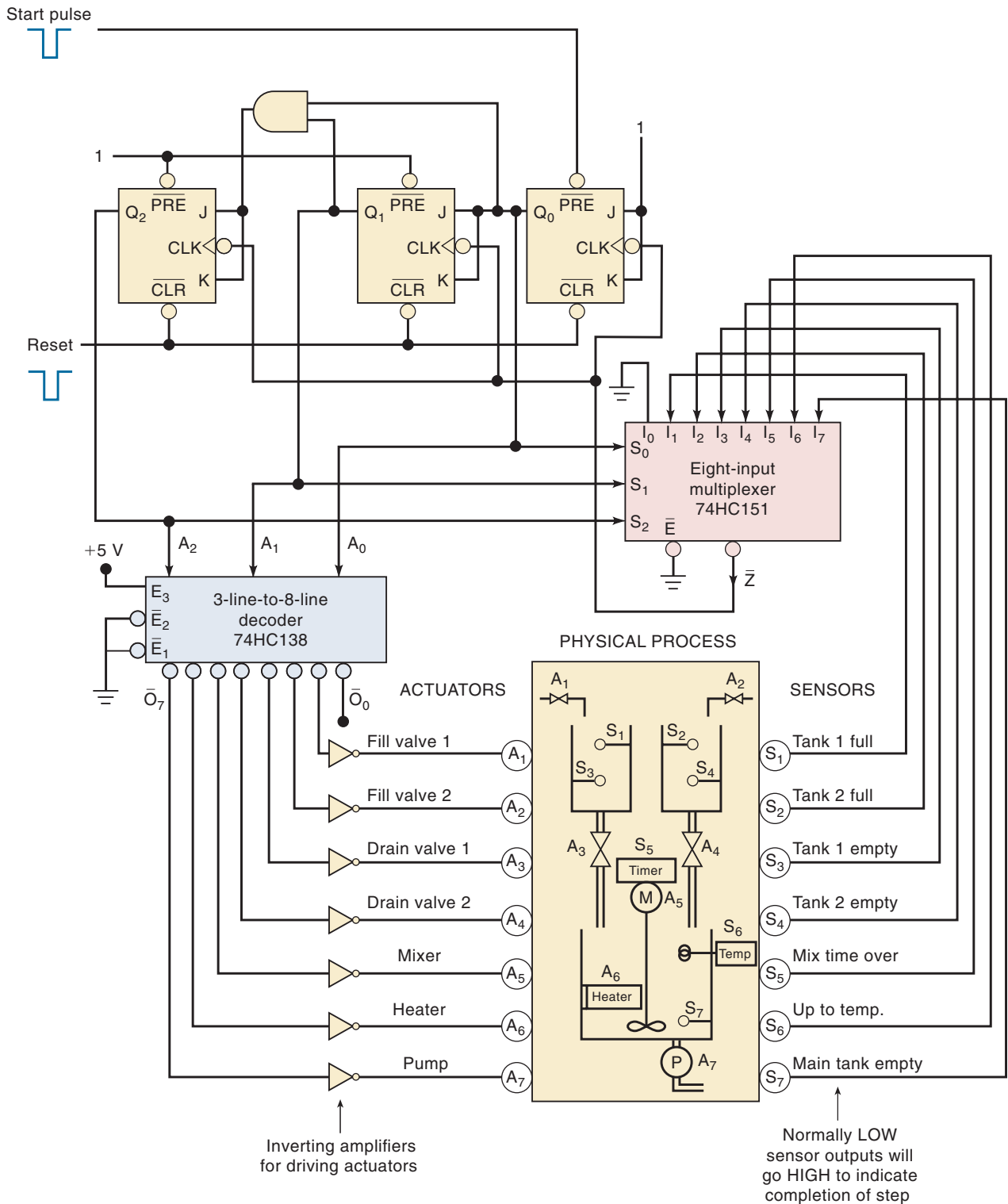


FIGURE 9-26 Seven-step control sequencer.

4. The LOW transition at output  $\bar{Z}$  is fed to the  $CLK$  of flip-flop  $Q_0$ . This negative transition advances the counter to the 010 state.
5. Decoder output  $\bar{O}_2$  now goes LOW, activating actuator 2, which is the second step in the process (opening fill valve 2).  $Z$  now equals  $I_2$  (the select code is 010). Because SENSOR output 2 is still LOW,  $Z$  will go HIGH.

6. When the second process step is complete, SENSOR output 2 goes HIGH, producing a LOW at  $\bar{Z}$  and advancing the counter to 011.
7. This same action is repeated for each of the other steps. When the seventh step is completed, SENSOR output 7 goes HIGH, causing the counter to go from 111 to 000, where it will remain until another START pulse reinitiates the sequence.

### Logic Function Generation

Multiplexers can be used to implement logic functions directly from a truth table without the need for simplification. When a multiplexer is used for this purpose, the select inputs are used as the logic variables, and each data input is connected permanently HIGH or LOW as necessary to satisfy the truth table.

Figure 9-27 illustrates how an eight-input multiplexer can be used to implement the logic circuit that satisfies the given truth table. The input variables  $A, B, C$  are connected to  $S_0, S_1, S_2$ , respectively, so that the levels on these inputs determine which data input appears at output  $Z$ . According to the truth table,  $Z$  is supposed to be LOW when  $CBA = 000$ . Thus, multiplexer input  $I_0$  should be connected LOW. Likewise,  $Z$  is supposed to be LOW for  $CBA = 011, 100, 101,$  and  $110$ , so that inputs  $I_3, I_4, I_5,$  and  $I_6$  should also be connected LOW. The other sets of  $CBA$  conditions must produce  $Z = 1$ , and so multiplexer inputs  $I_1, I_2,$  and  $I_7$  are connected permanently HIGH.

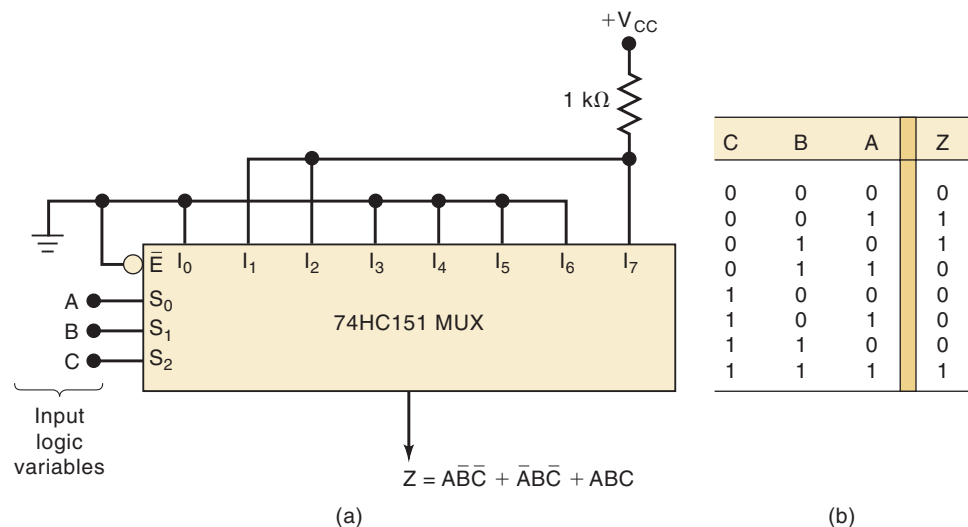
It is easy to see that any three-variable truth table can be implemented with this eight-input multiplexer. This method of implementation is often more efficient than using separate logic gates. For example, if we can write the sum-of-products expression for the truth table in Figure 9-27, we have

$$Z = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + ABC$$

This *cannot* be simplified either algebraically or by K mapping, and so its gate implementation would require three INVERTERS and four NAND gates, for a total of three ICs.

There is an even more efficient method for using multiplexers to implement logic functions. This method will allow the logic designer to use a multiplexer with three select inputs (e.g., a 74HC151) to implement a *four-variable* logic function. We will present this method in Problem 9-37.

**FIGURE 9-27** Multiplexer used to implement a logic function described by the truth table.



The most important concept to be gained from using a MUX to implement a sum-of-products expression is the fact that the logic function can be very easily changed by simply changing the 1s and 0s on the MUX inputs. In other words, a MUX can very easily be used as a programmable logic device (PLD). Many PLDs use this strategy in hardware blocks that are generally referred to as look-up tables (LUTs). We will discuss look-up tables in more detail in Chapters 12 and 13.

### OUTCOME ASSESSMENT QUESTIONS

1. What are some of the major applications of multiplexers?
2. *True or false:* When a multiplexer is used to implement a logic function, the logic variables are applied to the multiplexer's data inputs.
3. What type of circuit provides the select inputs when a MUX is used as a parallel-to-serial converter?

## 9-8 DEMULPLEXERS (DATA DISTRIBUTORS)

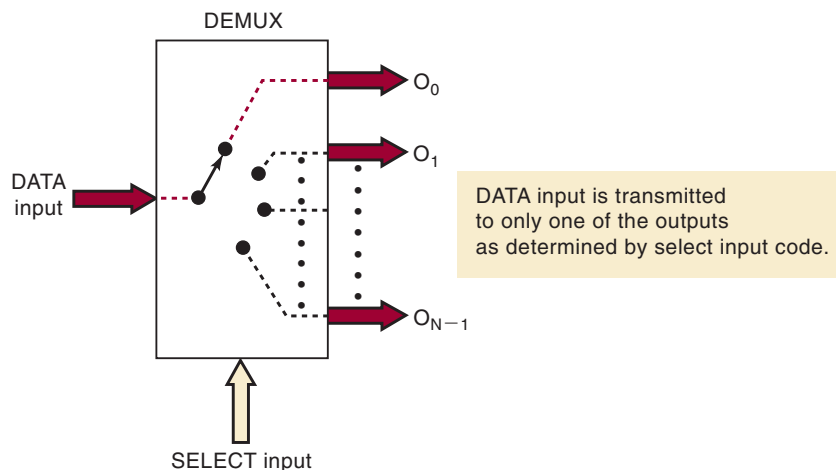
### OUTCOMES

Upon completion of this section, you will be able to:

- Define the term *demultiplexer*.
- State the nature of the inputs and outputs of a demultiplexer.
- State the role of an “enable” input to a demultiplexer.

A multiplexer takes several inputs and transmits *one* of them to the output. A **demultiplexer (DEMUX)** performs the reverse operation: it takes a single input and distributes it over several outputs. Figure 9-28 shows the functional diagram for a digital demultiplexer. The large arrows for inputs and outputs can represent one or more lines. The select input code determines to which output the DATA input will be transmitted. In other words, the demultiplexer takes one input data source and selectively distributes it to 1 of  $N$  output channels just like a multiposition switch.

**FIGURE 9-28** General demultiplexer.

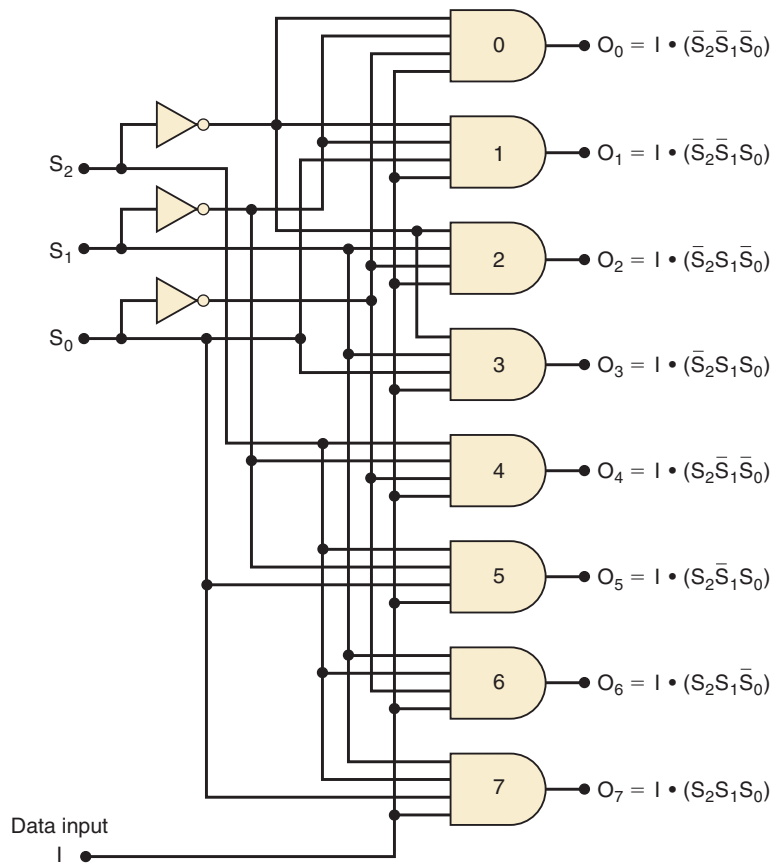


### 1-Line-to-8-Line Demultiplexer

Figure 9-29 shows the logic diagram for a demultiplexer that distributes one input line to eight output lines. The single data input line  $I$  is connected to all eight AND gates, but only one of these gates will be enabled by the SELECT input lines. For example, with  $S_2S_1S_0 = 000$ , only AND gate 0 will be enabled, and data input  $I$  will appear at output  $O_0$ . Other SELECT codes cause input  $I$  to reach the other outputs. The truth table summarizes the operation.

The demultiplexer circuit of Figure 9-29 is very similar to the 3-line-to-8-line decoder circuit in Figure 9-2 except that a fourth input ( $I$ ) has been added to each gate. It was pointed out earlier that many IC decoders have an ENABLE input, which is an extra input added to the decoder gates. This

**FIGURE 9-29** A 1-line-to-8-line demultiplexer.



Select Code			Outputs							
$S_2$	$S_1$	$S_0$	$O_7$	$O_6$	$O_5$	$O_4$	$O_3$	$O_2$	$O_1$	$O_0$
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

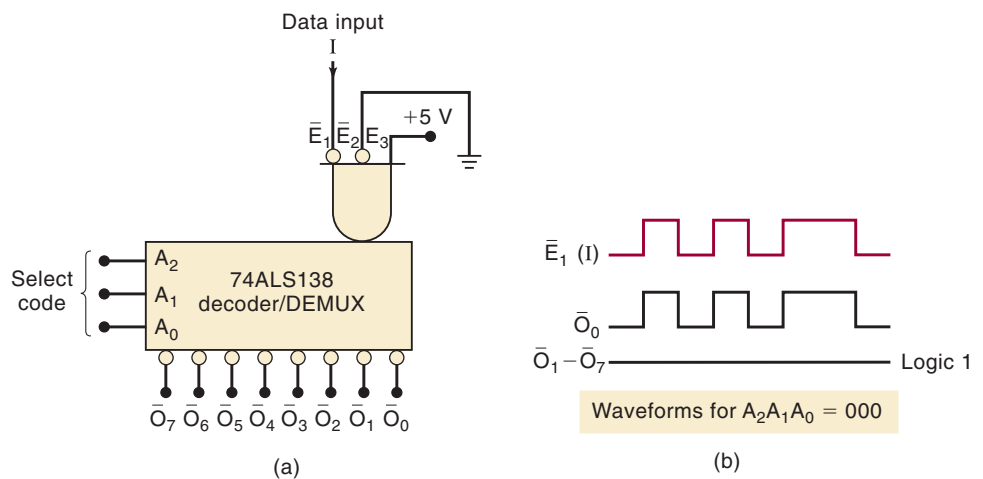
Note:  $I$  is the data input

type of decoder chip can therefore be used as a demultiplexer, with the binary code inputs (e.g.,  $A$ ,  $B$ ,  $C$  in Figure 9-2) serving as the SELECT inputs and the ENABLE input serving as the data input  $I$ . For this reason, IC manufacturers often call this type of device a *decoder/demultiplexer*, and it can be used for either function.

We saw earlier how the 74ALS138 is used as a 1-of-8 decoder. Figure 9-30 shows how it can be used as a demultiplexer. The enable input  $\bar{E}_1$  is used as the data input  $I$ , while the other two enable inputs are held in their active states. The  $A_2A_1A_0$  inputs are used as the select code. To illustrate the operation, let's assume that the select inputs are 000. With this input code, the only output that can be activated is  $\bar{O}_0$ , while all other outputs are HIGH.  $\bar{O}_0$  will go LOW only if  $\bar{E}_1$  goes LOW and will be HIGH if  $\bar{E}_1$  goes HIGH. In other words,  $\bar{O}_0$  will follow the signal on  $\bar{E}_1$  (i.e., the data input,  $I$ ) while all other outputs stay HIGH. In a similar manner, a different select code applied to  $A_2A_1A_0$  will cause the corresponding output to follow the data input,  $I$ .

Figure 9-30(b) shows typical waveforms for the case where  $A_2A_1A_0 = 000$  selects output  $\bar{O}_0$ . For this case, the data signal applied to  $\bar{E}_1$  will be transmitted to  $\bar{O}_0$ , and all other outputs will remain in their inactive HIGH state.

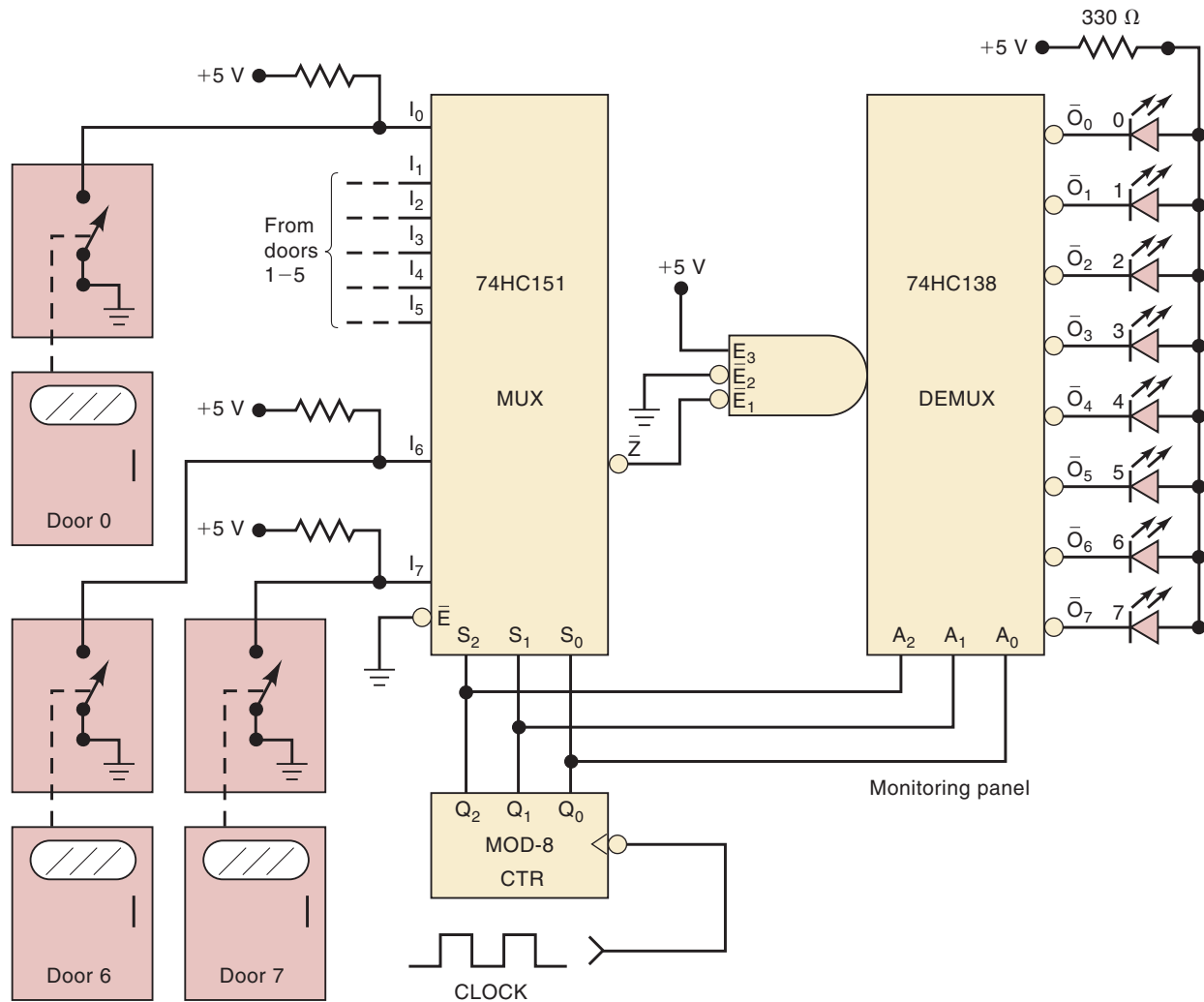
**FIGURE 9-30** (a) The 74ALS138 decoder can function as a demultiplexer with  $\bar{E}_1$  used as the data input; (b) typical waveforms for a select code of  $A_2A_1A_0 = 000$  show that  $\bar{O}_0$  is identical to the data input  $I$  on  $\bar{E}_1$ .



## Security Monitoring System

Consider the case of a security monitoring system in an industrial plant where the open/closed status of many access doors is to be monitored. Each door controls the state of a switch, and it is necessary to display the state of each switch on LEDs that are mounted on a remote monitoring panel at the security guard's station. One way to do this would be to run a separate signal from each door switch to an LED on the monitoring panel. This setup would require running many wires over a long distance. A better approach that would reduce the amount of wiring to the monitoring panel uses a multiplexer/demultiplexer combination. Figure 9-31 shows a system that can handle eight doors, but the basic idea can be expanded to any number.





**FIGURE 9-31** Security monitoring system.

### EXAMPLE 9-11

Examine Figure 9-31 carefully and describe the complete operation.

#### Solution

The eight door switches are the data inputs to the MUX; they produce a HIGH when a door is open and a LOW when it is closed. The MOD-8 counter provides the select inputs to the MUX and also to the DEMUX on the remote monitoring panel. Each DEMUX output is connected to an indicator LED that will be on when the output is LOW. Clock pulses applied to the counter will cause the select inputs to sequence through all of the possible states 000 through 111. At each number of the counter, the switch status for the door of the same number will be inverted by the MUX and passed to output  $\bar{Z}$ . From there, it is transmitted to the DEMUX input, which passes it through to the output corresponding to the same number.

For example, let's say that the counter is at the count of 110 (6). While the counter is in this state, let's say that door 6 is closed. The LOW at  $I_6$  will pass through the MUX and be inverted to produce a HIGH at  $\bar{Z}$ . This HIGH will be passed through the DEMUX to output  $\bar{O}_6$  so that LED 6 will be off, indicating that door 6 is closed. Now let's say that door 6 is open. A LOW will appear at

$\bar{Z}$  and  $\bar{O}_6$  so that LED 6 will be on to signal that door 6 is open. Of course, all other LEDs will be off during this time because  $\bar{O}_6$  is the only active output.

As the counter is clocked through its eight states 000 through 111, the LEDs will sequentially indicate the status of the eight doors. If all the doors are closed, none of the LEDs will be on even when the corresponding DEMUX output is selected. If a door is open, its LED will turn on only during the time interval that the counter is at the appropriate count; it will be off at all other counts. Thus, the LED will be flashing on and off if its door is open. The flashing rate can be adjusted by changing the frequency of the clock.

Note that there are only four signal lines going from the “door-sensing” circuitry to the remote monitoring panel: the  $\bar{Z}$  output and the three select lines. This is a saving of four lines when compared with the alternative of having one line per door. The MUX/DEMUX combination is used to transmit the status of each door to its LED one at a time (serially) instead of all at once (parallel).

### Synchronous Data Transmission System

Figures 9-32 and 9-33 show the logic diagrams for a synchronous data transmission system that is used to transmit four, four-bit words serially from a transmitter to a remote receiver. To operate this system, four data words are parallel-loaded into the input registers of the transmitter block and the

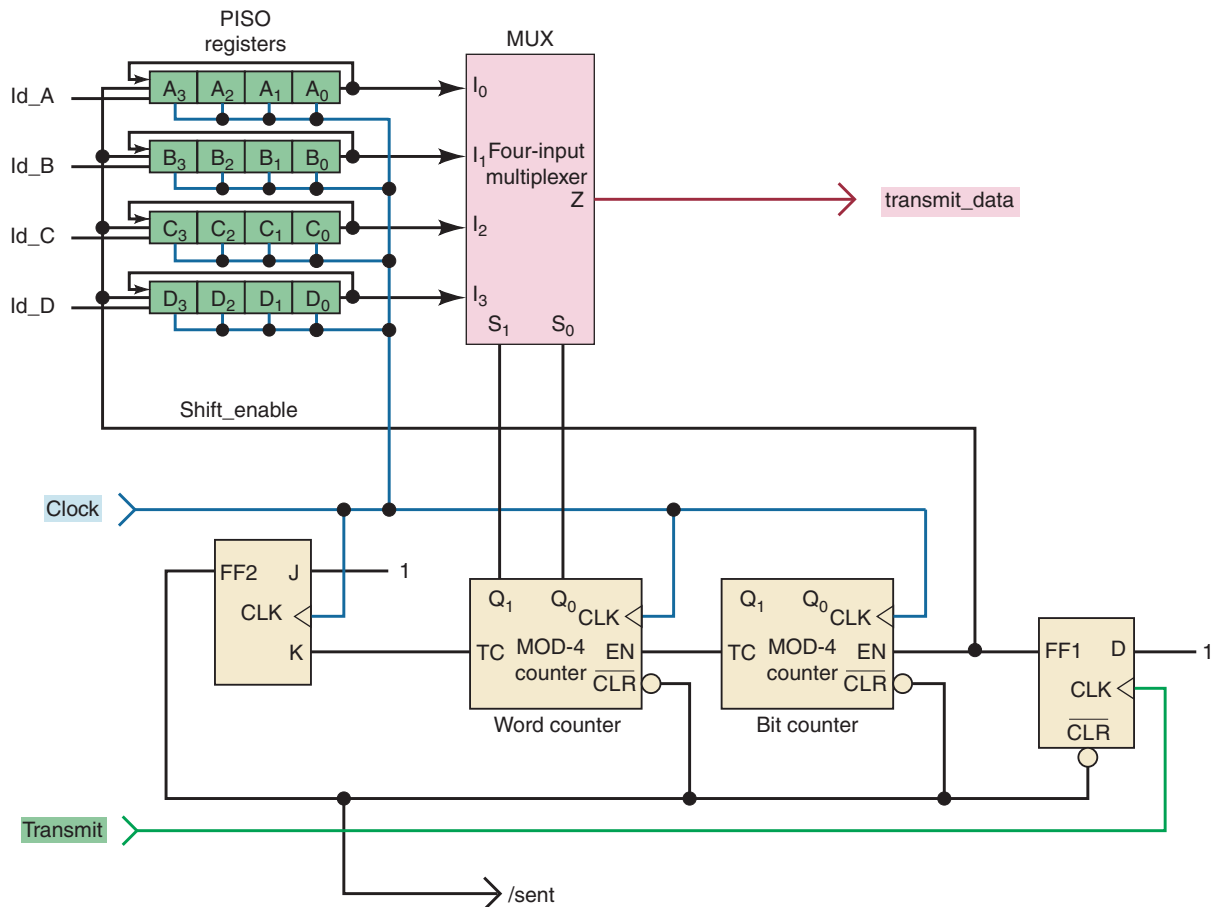
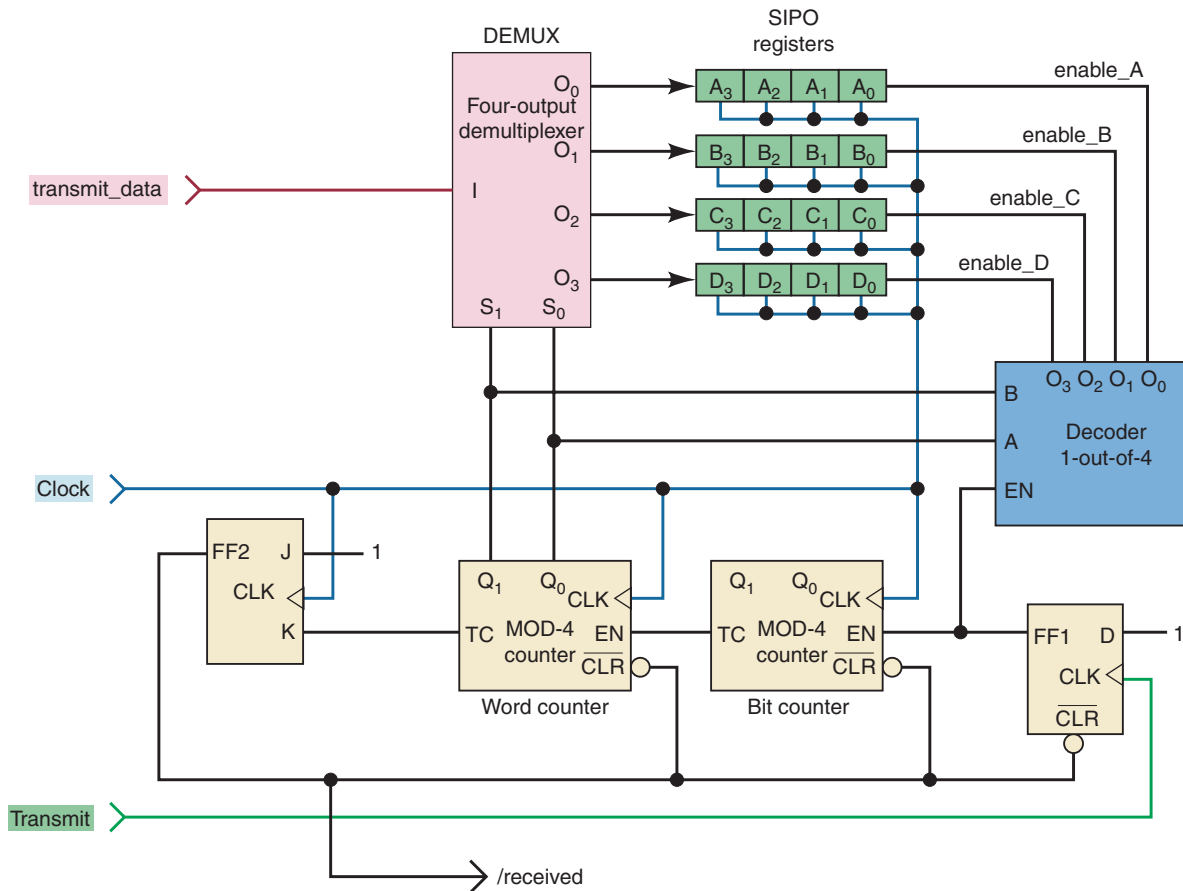


FIGURE 9-32 Transmitter block in synchronous data transmission system.



**FIGURE 9-33** Receiver block in synchronous data transmission system.

transmit signal is activated. The 16 data bits are then sent over a single data line, one bit at a time, reassembled by the receiver, and stored in output registers. Let's look at the transmitter details in Figure 9-32 first. The *clock* input is a high-frequency, constantly running, periodic clock signal that synchronizes all activities in the system. The four-bit data words are stored individually (synchronously) in the PISO registers when enabled by the appropriate *ld<sub>x</sub>* input. For simplicity, the parallel data inputs to the PISO registers are not shown in the diagram. These input registers are designed to shift the data to the right and also recirculate the LSB (rightmost bit) to the MSB (leftmost bit). With this arrangement the bits are all shifted to the serial output and also end up back in their proper locations after four clock pulses.

**TRANSMITTER OPERATION** Initially, let's assume that all the flip-flops and the two MOD-4 counters in Figure 9-32 are all cleared. On the next PGT of clock, FF2 is SET, removing the asynchronous clear command from the counters and FF1. When the *transmit* signal goes HIGH, FF1 is SET, putting all the shift registers in the shift mode. The MUX selects input 0 (register A) because the MOD-4 Word counter is at 0. At this point the LSB of register A is on the *transmit\_data* line. The next three clock pulses (counted by the Bit counter) shift the other bits of register A to the serial output. As a result, the *transmit\_data* line outputs each of the register A bits, one at a time from the least to the most significant. On the fourth PGT, the Bit counter rolls over to zero, the Word counter increments to 1, all of the shift registers have recirculated their data back to the original position, and the MUX now selects the LSB data from register B to output on the *transmit\_data* line. The next three clocks shift

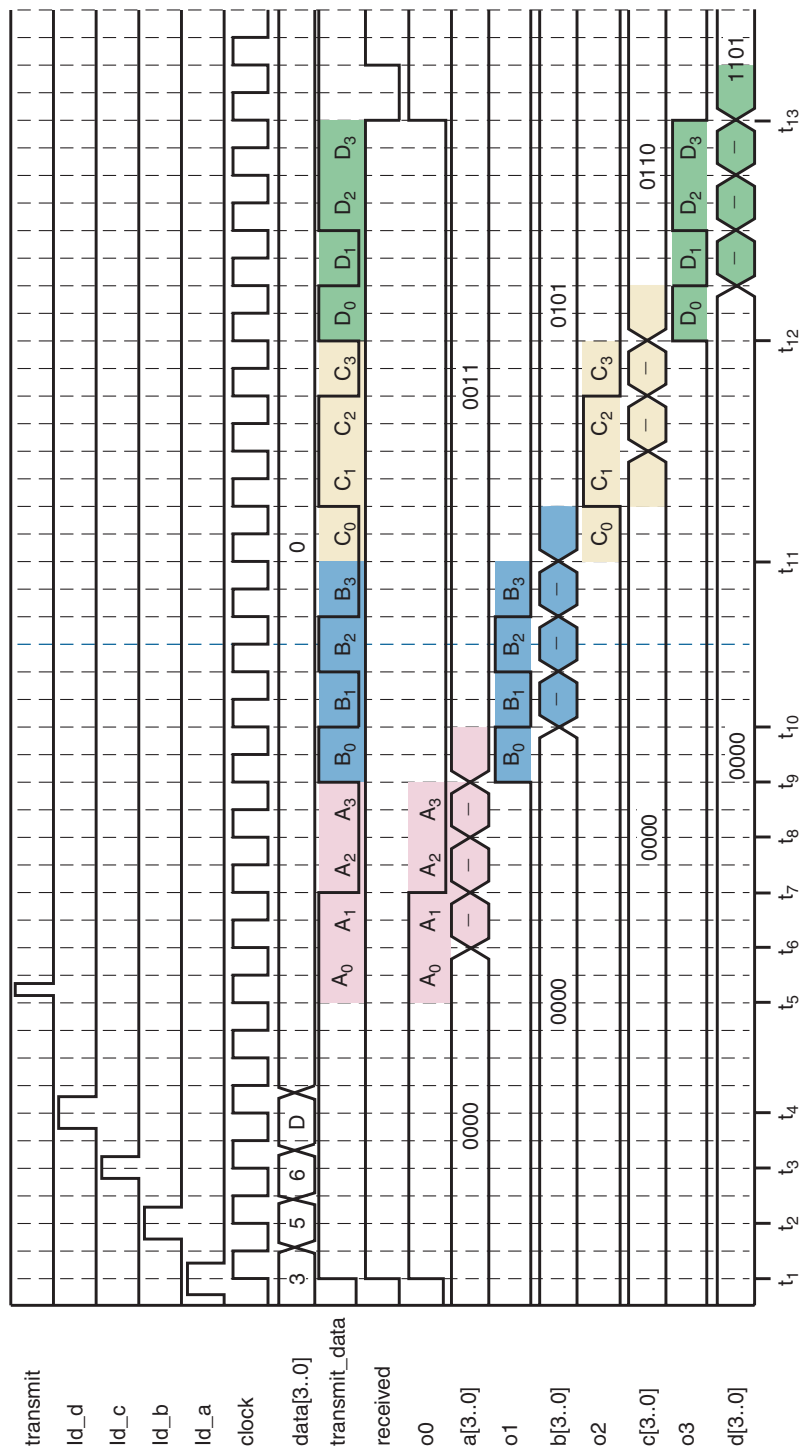
out the contents of register B, followed by registers C and D. On the 16th PGT, FF2 toggles to a zero state, resetting all the counters and disabling any further counting by also clearing FF1. The next PGT sets FF2 again, and the system is waiting for new data to be loaded and the next *transmit* signal.

**RECEIVER OPERATION** The receiver circuit shown in Figure 9-33 is very similar in operation to the transmitter. Notice that all flip-flops, counters, and registers use the same clock as the transmitter. The receiver uses a DEMUX to distribute the serial data to the appropriate SIPO register and a decoder to enable one register at a time. Let's begin analyzing this circuit with all counters and flip-flops at zero. The next *clock* sets FF2, removing the asynchronous clear command from the counters and FF1. When the *transmit* line goes HIGH, FF1 is SET, enabling the Bit counter, Word counter, and also the decoder. With the Word counter at zero, the decoder enables register A and the DEMUX connects the serial data line (which currently contains the LSB of transmit register A) to the serial data input of receive register A. The next PGT shifts the least significant data bit into register A and advances the Bit counter. The next three PGTs shift the next three data bits into register A, the Bit counter rolls over to zero, the Word counter increments to 1, and the decoder and DEMUX switch to register B. After the 16th PGT, all four registers contain the proper data, FF2 has toggled to a zero state, FF1 is cleared and disables the decoder, which disables all the SIPO registers. On the next PGT, FF2 is set and the system is waiting for the next transmission of data.

**SYSTEM TIMING** The timing diagram in Figure 9-34 shows the parallel data that is loaded into the transmitter, the serial data stream, and the distribution and storage of the four data values in the receiver registers. At times  $t_1$ – $t_4$ , the binary data values (shown as hex 3, 5, 6, and D) are loaded into transmit registers A, B, C, and D, respectively. The system is idle until the *transmit* line goes HIGH at  $t_5$ . At this point the LSB from register A ( $A_0$ ) is already on the *transmit\_data* line. Also notice that at  $t_5$ – $t_8$ , the data on output  $O_0$  of the DEMUX is identical to the *transmit\_data* line. This shows that the DEMUX has distributed the *transmit\_data* to shift register A. At  $t_6$ , the PGT of the clock shifts  $A_0$  into the MSB of receive register A, all transmit data registers (not shown in the timing) are shifted, and data bit  $A_1$  appears on the *transmit\_data* line. At times  $t_7$ ,  $t_8$ , and  $t_9$ , the other three bits are shifted into register A such that after  $t_9$ , receive register A contains the data bits that were stored in transmit register A. The diagram shows that the DEMUX has switched to distribute data to register B because the DEMUX output  $O_1$  is now identical to *transmit\_data* from  $t_9$  through  $t_{11}$ . Starting at  $t_{10}$ , the data are shifted into receive register B, which at  $t_{11}$  contains the value that was originally stored in transmit register B. Register C and register D are sent and stored from  $t_{11}$  to  $t_{12}$  and from  $t_{12}$  to  $t_{13}$ , respectively.

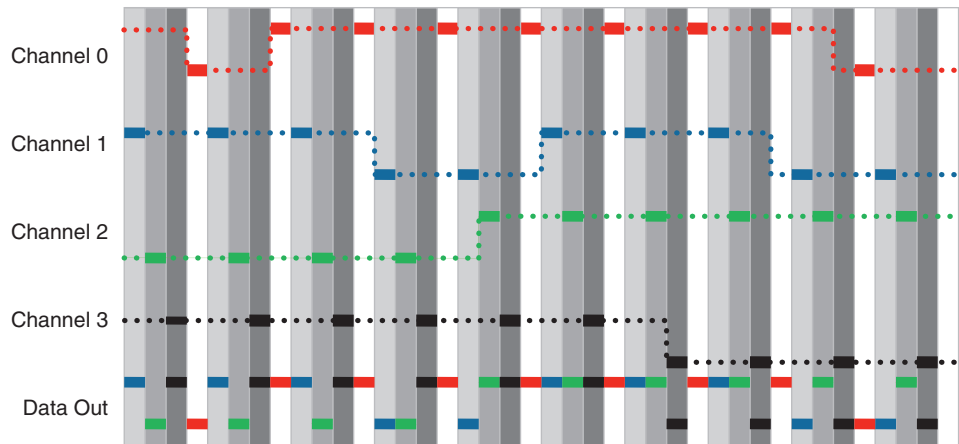
## Time Division Multiplexing

There are many instances where multiple independent digital signals must be transported over long distances using the same data pathway or transmission medium such as a wire, fiber optic cable, or wireless radio frequency. The signals must “take turns” using the pathway. It is very clear that these applications call for multiplexing and demultiplexing. This sounds very simple, but if the signals are “taking turns” travelling over the pathway, how can they all be arriving at the same time on the receiving end? As a humorous example, think about four people sharing a single two-way radio, each trying to converse with one of four people on the other end. While one couple is



**FIGURE 9-34** Timing diagram for one complete transmission cycle.

**FIGURE 9-35** Four channels of digital waveforms being multiplexed onto a single pathway.



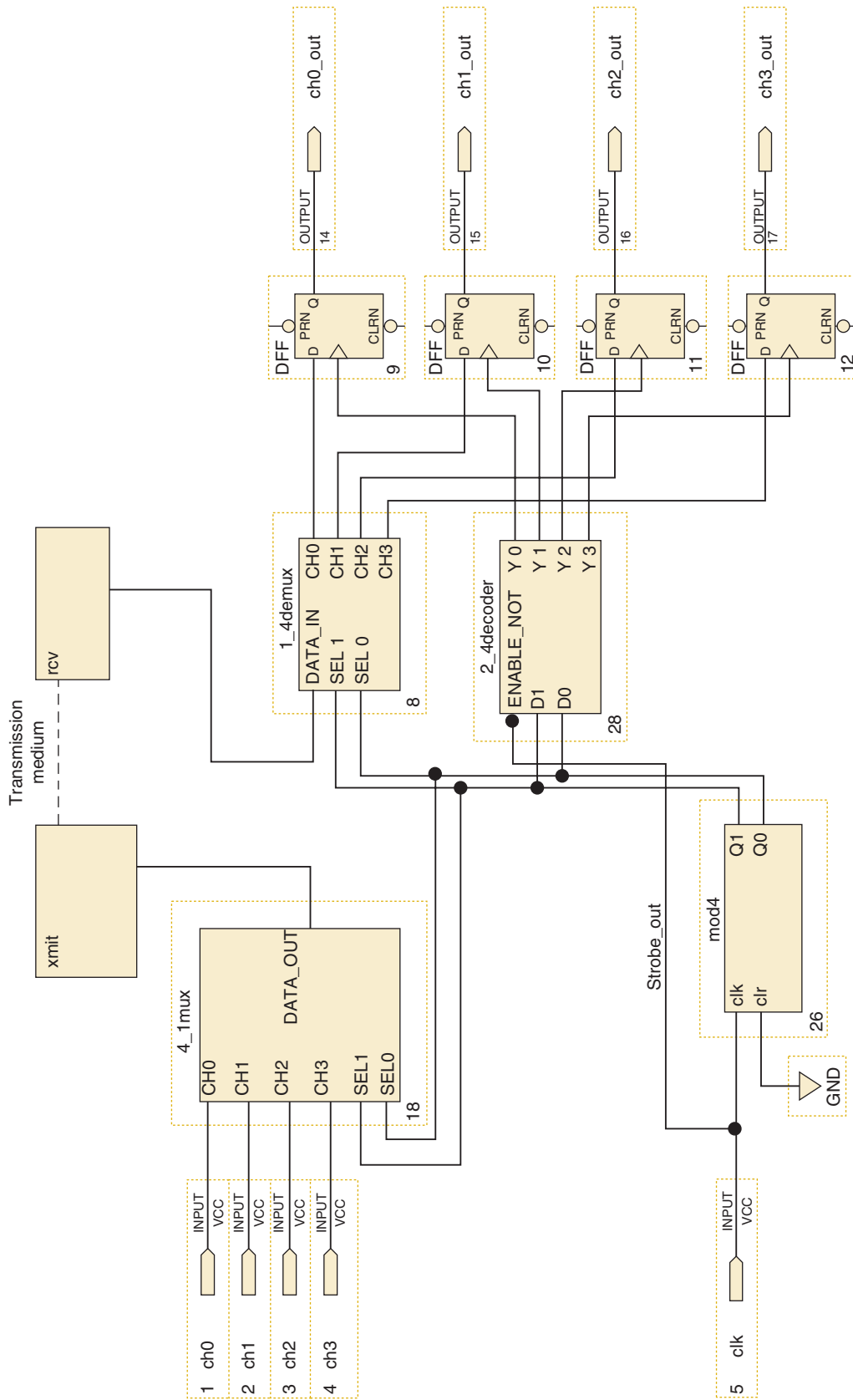
talking, the other three are completely idle, waiting for their next turn. If our modern telephone system worked this way, would people be satisfied with “taking turns”? In fact, our telephone system does allow us to have many conversations sharing a single fiber optic cable that carries those conversations over many miles. If at any point in time there is only a 1 or 0 on the fiber, how do all the conversations continue at the same time? This section describes the strategy that solves this problem called **time division multiplexing (TDM)**.

Time division multiplexing (TDM) allows each signal to use the common pathway for only a very short period of time. By switching the MUX/DEMUX fast enough, only a very tiny piece (i.e., a time slice) of each digital waveform will be using the data path at any time. Each piece will be either a 1 or 0 and it will take many of these tiny pieces to make up one complete cycle of the digital signal as shown in Figure 9-35.

The block diagram of a system to perform TDM on four channels of data is shown in Figure 9-36. The counter block cycles through the channel numbers from 0–3. For simplicity, this same count value is available at the receiving end to be used by the DEMUX and decoder.

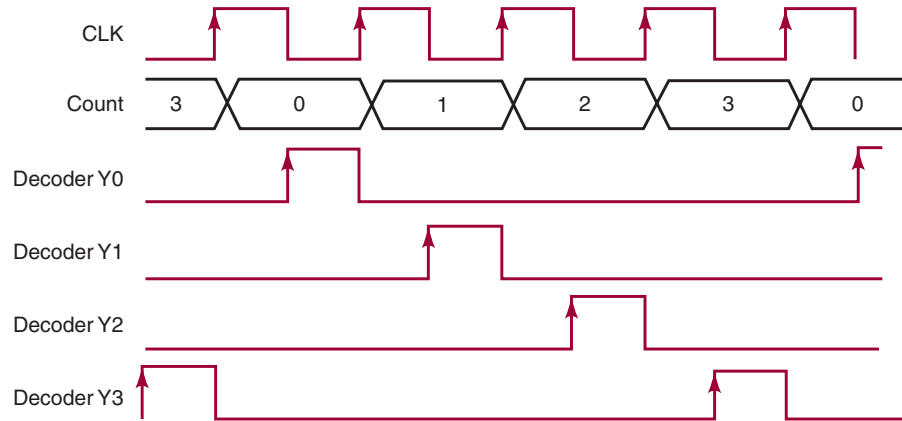
Each little piece of the multiplexed signals is distributed to its proper destination by the demultiplexer at the end of the wire. As each tiny piece comes out of the demultiplexer on the channel currently selected, it must somehow stay at its current state (1 or 0) while all the other channels take their turn on the wire. In Figure 9-35, this interval is represented by the dotted lines between bits. The value is held by storing the little bit of data in a D flip-flop until that channel’s turn comes around again. In order to store the data on the D flip-flop, we must provide a clock edge while the data is stable. In other words, the clock edge must occur between the time the data bit first arrives and well before it goes away (when the DEMUX switches again). The role of the decoder is to deliver a rising clock edge to the appropriate flip-flop at the appropriate time. Notice in Figure 9-37 how the rising edges of the decoder Y outputs are in the center of the counter states. This is accomplished by activating the enable of the decoder (active-LOW) with the low half-cycle of the clock, which begins half-way between the positive edges of the clock that increment the channel number.

The most important aspect of a TDM system is the speed at which the channels are switched and the signals are sampled. For this system, the clock speed must be fast enough to assure an absolute minimum of four clock cycles during the shortest interval of constant data. Even at this clock speed the output waveforms will be slightly distorted from the input waveform because the transitions are delayed until the next sampling of that channel. In general, the higher the clock frequency, the more similar the output waveforms will be to the input.



**FIGURE 9-36** A time slice MUX/DEMUX system.

**FIGURE 9-37** Timing for the decoder outputs to capture valid data with the DEMUX.



This is a simplified TDM system intended to demonstrate the concepts. A real system would need to generate separate clock signals, one at the transmitter and another at the receiver. The clocks and the counters on each end need to be synchronized. As you study more advanced topics in digital communication, you will see how the basic building blocks we have studied can be used to solve problems such as this.

### OUTCOME ASSESSMENT QUESTIONS

1. Explain the difference between a DEMUX and a MUX.
2. *True or false:* The circuit for a DEMUX is basically the same as for a decoder.
3. For the system of Figure 9-31, what will the security guard see on the monitoring panel when all of the doors are open?

## 9-9 MORE TROUBLESHOOTING

### OUTCOME

Upon completion of this section, you will be able to:

- Improve troubleshooting techniques and skills.

Here are three more examples to illustrate the observation/reasoning process that is such an important initial step when troubleshooting. For each case, try to determine the circuit fault before looking at the solution.

### EXAMPLE 9-12

Consider the circuit of Figure 9-24. A test performed on this circuit yields the result shown in Table 9-3. What is the probable circuit fault?

**TABLE 9-3**

		Actual Count	Displayed Count
Case 1	Counter 1	25	25
	Counter 2	37	35
Case 2	Counter 1	49	49
	Counter 2	72	79
Case 3	Counter 1	96	96
	Counter 2	14	16



**Solution**

In each of the test cases, the display of counter 1 matches the counter's actual count. This indicates that the  $I_1$  inputs, all MUX outputs, and both displays are probably working correctly. On the other hand, each test case shows that counter 2's *tens* digit is displayed correctly but its *units* digit is displayed incorrectly. This could mean that there is a fault somewhere between the output of the units section of counter 2 and the  $I_0$  inputs of the units MUX. We should compare the bit patterns of the actual and displayed values of the units for counter 2 (Table 9-4). The idea is to look for things such as a bit that does not change (stuck LOW or HIGH) or two bits that are reversed (crossed connections). The data in Table 9-4 reveal no obvious pattern.

**TABLE 9-4**

	Actual Units	Displayed Units
Case 1	0111 (7)	0101 (5)
Case 2	0010 (2)	1001 (9)
Case 3	0100 (4)	0110 (6)

If we take another look at the recorded test results, we see that the displayed units digit of counter 2 is always the same as the units digit of counter 1. This symptom is probably the result of a constant logic HIGH at the select input ( $S$ ) of the units MUX because that would continually pass the units digit of counter 1 to the units MUX output. This constant HIGH at the select input is most likely caused by an open path somewhere between the select input of the tens MUX and the select input of the units MUX. It could not be caused by a short to  $V_{CC}$  because that would also keep the select input of the tens MUX at a constant HIGH, and we know that the tens MUX is working.

**EXAMPLE 9-13**

The security monitoring system of Figure 9-31 is tested and the results are recorded in Table 9-5. What are the possible faults that could produce these results?

**TABLE 9-5**

Condition	LEDs
All doors closed	All LEDs off
Door 0 open	LED 4 flashing
Door 1 open	LED 5 flashing
Door 2 open	LED 6 flashing
Door 3 open	LED 7 flashing
Door 4 open	LED 4 flashing
Door 5 open	LED 5 flashing
Door 6 open	LED 6 flashing
Door 7 open	LED 7 flashing

**Solution**

Again, the data should be reviewed to see if there is some pattern that could help to narrow down the search for the fault to a small area of the circuit. The data in Table 9-5 reveal that the correct LEDs flash for open doors 4 through 7.

They also show that for open doors 0 through 3, the number of the flashing LED is *four* more than the number of the door, and LEDs 0 through 3 are always off. This is most probably caused by a constant logic HIGH at  $A_2$ , the MSB of the select input of the DEMUX, because this would always make the select code 4 or greater, and it would add 4 to the select codes 0 through 3.

Thus, we have two possibilities:  $A_2$  is somehow shorted to  $V_{CC}$ , or there is an open connection at  $A_2$ . A little thought will eliminate the first choice as a possibility because this would also mean that  $S_2$  of the MUX would also be stuck HIGH. If that were so, then the status of doors 0 through 3 would not get through the MUX and into the DEMUX. We know that this is not true because the data show that when any of these doors is open, it affects one of the DEMUX outputs.

#### EXAMPLE 9-14

An extremely important principle of troubleshooting, called *divide-and-conquer*, was introduced in Section 9-5. It is really not about military strategy, but rather describes the most efficient way to eliminate from consideration all the parts of the circuit that are working correctly. Assume that data have been loaded into the four transmit registers of Figure 9-32 and the transmit pulse has occurred, but after the next 16 clock pulses, no new data have appeared in the receive registers shown in Figure 9-33. How can we most efficiently find the problem?

#### Solution

In a synchronous digital system that is simply not functioning, it is reasonable first to check to see if the power supply and clock are working, just as you might check for a pulse if you found a person lying on the ground. However, assuming the clock is oscillating, there is a much more efficient way to isolate the problem than randomly picking points in the circuit and determining if the correct signal is present. We want to perform a test on this circuit such that, if we obtain the desired results, we know that half of the circuit is working correctly and we can eliminate that half from consideration. In this circuit the best place to look is at the *transmit\_data* line. A logic probe should be placed on the *transmit\_data* line and the *transmit* signal should be activated. If a burst of pulses is observed on the logic probe, it means that the transmit section is functioning. We may not know if the data are correct, but remember, the receiver is not getting incorrect data but rather no data at all. However, if no burst of pulses is observed, there is certainly a problem in the transmit section.

A troubleshooting tree diagram as shown in Figure 9-38 is helpful in isolating problems in a system. Let's assume there were no pulses on *transmit\_data*. Now we need to perform a test on the transmitter to prove that half of the transmitter is working properly. In this case the circuit does not divide exactly in half easily. A good choice might be to examine the output of the word counter. A logic probe should be placed on the select inputs of the MUX and the *transmit* signal activated. If brief pulses occur immediately after transmit, then the entire control section (made up of two counters and two flip-flops) is probably functioning properly and we can look elsewhere. The next place to look is at the outputs of the PISO registers (or data inputs of the MUX). If data pulses are present on each line after *transmit* is activated, the problem must be in the MUX. If not, we can further break down the PISO section. Each test that is performed should eliminate the largest possible amount of the remaining circuitry until all that is left is a small block containing the fault.

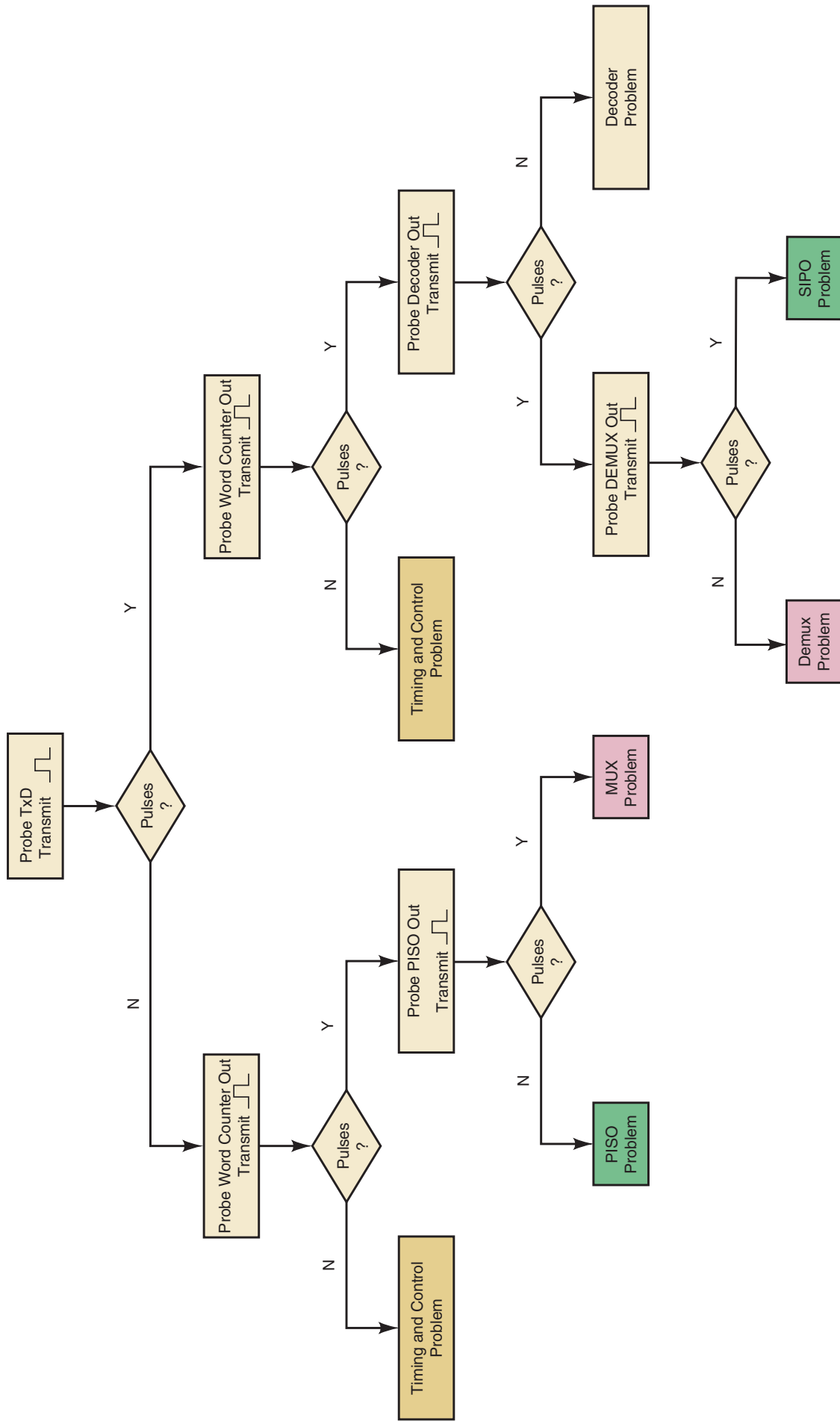


FIGURE 9-38 Example 9-14: A troubleshooting tree diagram.

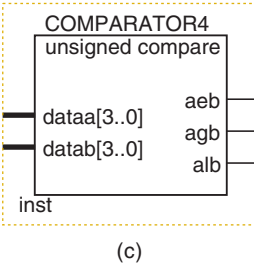
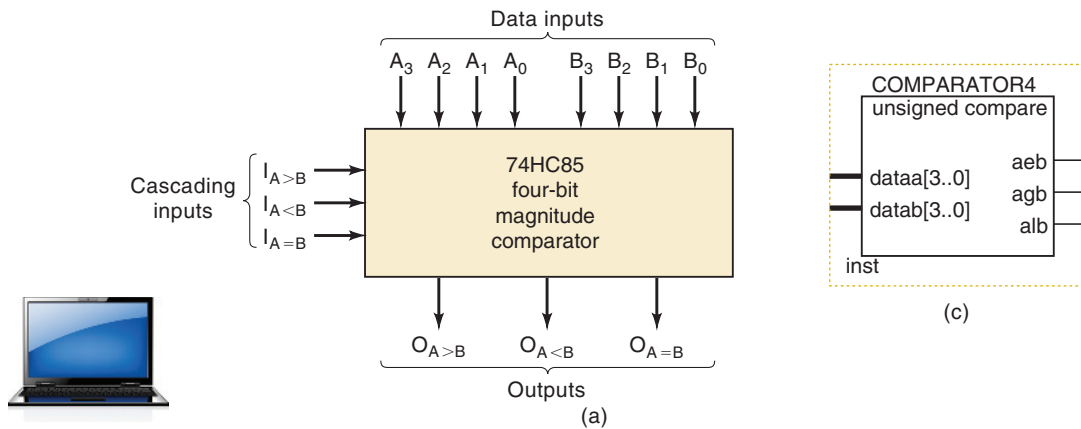
## 9-10 MAGNITUDE COMPARATOR

### OUTCOMES

Upon completion of this section, you will be able to:

- Define the term *magnitude comparator*.
- Identify and define the role of each input and output of a magnitude comparator.
- Cascade modular comparator blocks.

Another useful member of the MSI category of ICs is the **magnitude comparator**. It is a combinational logic circuit that compares two input binary quantities and generates outputs to indicate which one has the greater magnitude. Figure 9-39(a) shows the logic symbol and part (b) shows the truth table for the 74HC85 four-bit magnitude comparator. Figure 9-39(c)



TRUTH TABLE

Comparing Inputs				Cascading Inputs			Outputs		
A <sub>3</sub> , B <sub>3</sub>	A <sub>2</sub> , B <sub>2</sub>	A <sub>1</sub> , B <sub>1</sub>	A <sub>0</sub> , B <sub>0</sub>	I <sub>A&gt;B</sub>	I <sub>A&lt;B</sub>	I <sub>A=B</sub>	O <sub>A&gt;B</sub>	O <sub>A&lt;B</sub>	O <sub>A=B</sub>
A <sub>3</sub> >B <sub>3</sub>	X	X	X	X	X	X	H	L	L
A <sub>3</sub> <B <sub>3</sub>	X	X	X	X	X	X	L	H	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> >B <sub>2</sub>	X	X	X	X	X	H	L	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> <B <sub>2</sub>	X	X	X	X	X	L	H	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> >B <sub>1</sub>	X	X	X	X	H	L	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> <B <sub>1</sub>	X	X	X	X	L	H	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> =B <sub>1</sub>	A <sub>0</sub> >B <sub>0</sub>	X	X	X	H	L	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> =B <sub>1</sub>	A <sub>0</sub> <B <sub>0</sub>	X	X	X	L	H	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> =B <sub>1</sub>	A <sub>0</sub> =B <sub>0</sub>	H	L	L	H	L	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> =B <sub>1</sub>	A <sub>0</sub> =B <sub>0</sub>	L	H	L	L	H	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> =B <sub>1</sub>	A <sub>0</sub> =B <sub>0</sub>	X	X	H	L	L	H
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> =B <sub>1</sub>	A <sub>0</sub> =B <sub>0</sub>	L	L	L	H	H	L
A <sub>3</sub> =B <sub>3</sub>	A <sub>2</sub> =B <sub>2</sub>	A <sub>1</sub> =B <sub>1</sub>	A <sub>0</sub> =B <sub>0</sub>	H	H	L	L	L	L

H = HIGH voltage level  
 L = LOW voltage level  
 X = Immaterial

(b)

**FIGURE 9-39** (a) Logic symbol; (b) truth table for a 74HC85 (7485, 74LS85) four-bit magnitude comparator; (c) a similar megafunction.

shows the megafunction symbol. Cascading inputs are not necessary on a megafunction because there is no need to cascade. Instead, simply specify larger data input ports.

### Data Inputs

The 74HC85 compares two *unsigned* four-bit binary numbers. One of them is  $A_3A_2A_1A_0$ , which is called word *A*; the other is  $B_3B_2B_1B_0$ , which is called word *B*. The term *word* is used in the digital computer field to designate a group of bits that represents some specific type of information. Here, word *A* and word *B* represent numerical quantities.

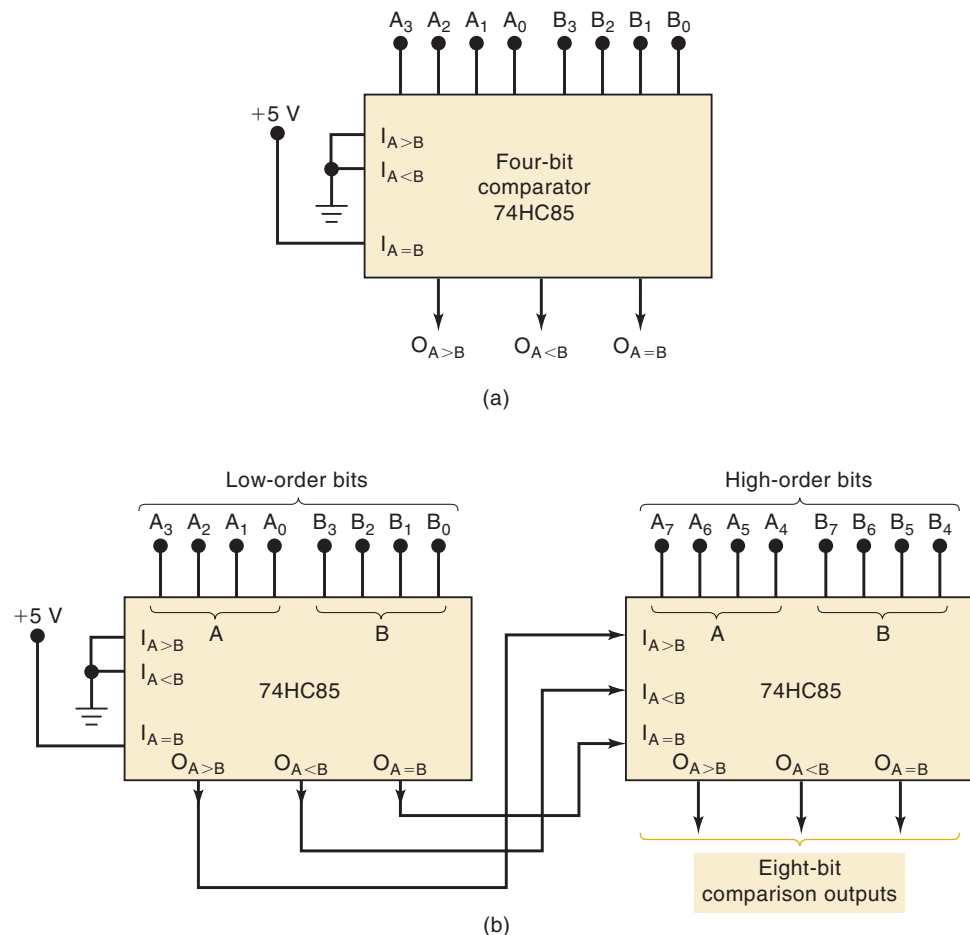
### Outputs

The 74HC85 has three active-HIGH outputs. Output  $O_{A>B}$  will be HIGH when the magnitude of word *A* is greater than the magnitude of word *B*. Output  $O_{A<B}$  will be HIGH when the magnitude of word *A* is less than the magnitude of word *B*. Output  $O_{A=B}$  will be HIGH when word *A* and word *B* are identical.

### Cascading Inputs

Cascading inputs provide a means for expanding the comparison operation to more than four bits by cascading two or more four-bit comparators. Note that the cascading inputs are labeled the same as the outputs. When a four-bit comparison is being made, as in Figure 9-40(a), the cascading inputs

**FIGURE 9-40** (a) 74HC85 wired as a four-bit comparator; (b) two 74HC85s cascaded to perform an eight-bit comparison.



should be connected as shown in order for the comparator to produce the correct outputs.

When two comparators are to be cascaded, the outputs of the lower-order comparator are connected to the corresponding inputs of the higher-order comparator. This is shown in Figure 9-40(b), where the comparator on the left is comparing the lower-order four bits of the two eight-bit words:  $A_7A_6A_5A_4A_3A_2A_1A_0$  and  $B_7B_6B_5B_4B_3B_2B_1B_0$ . Its outputs are fed to the cascade inputs of the comparator on the right, which is comparing the high-order bits. The outputs of the high-order comparator are the final outputs that indicate the result of the eight-bit comparison.

**EXAMPLE 9-15**

Describe the operation of the eight-bit comparison arrangement in Figure 9-40(b) for the following cases:

- (a)  $A_7A_6A_5A_4A_3A_2A_1A_0 = 10101111$ ;  $B_7B_6B_5B_4B_3B_2B_1B_0 = 10110001$   
 (b)  $A_7A_6A_5A_4A_3A_2A_1A_0 = 10101111$ ;  $B_7B_6B_5B_4B_3B_2B_1B_0 = 10101001$

**Solution**

- (a) The high-order comparator compares its inputs  $A_7A_6A_5A_4 = 1010$  and  $B_7B_6B_5B_4 = 1011$  and produces  $O_{A<B} = 1$  regardless of what levels are applied to its cascade inputs from the low-order comparator. In other words, once the high-order comparator senses a difference in the high-order bits of the two eight-bit words, it knows which eight-bit word is greater without having to look at the results of the low-order bit comparison.
- (b) The high-order comparator sees  $A_7A_6A_5A_4 = B_7B_6B_5B_4 = 1010$ , so it must look at its cascade inputs to see the result of the low-order bit comparison. The low-order comparator has  $A_3A_2A_1A_0 = 1111$  and  $B_3B_2B_1B_0 = 1001$ , which produces a 1 at its  $O_{A>B}$  output and the  $I_{A>B}$  input of the high-order comparator. The high-order comparator senses this 1, and because its data inputs are equal, it produces a HIGH at its  $O_{A>B}$  to indicate the result of the eight-bit comparison.

**Applications**

Magnitude comparators are also useful in control applications where a binary number representing the physical variable being controlled (e.g., position, speed, or temperature) is compared with a reference value. The comparator outputs are used to actuate circuitry to drive the physical variable toward the reference value. The following example will illustrate one application. We will examine another comparator application in Problem 9-52.

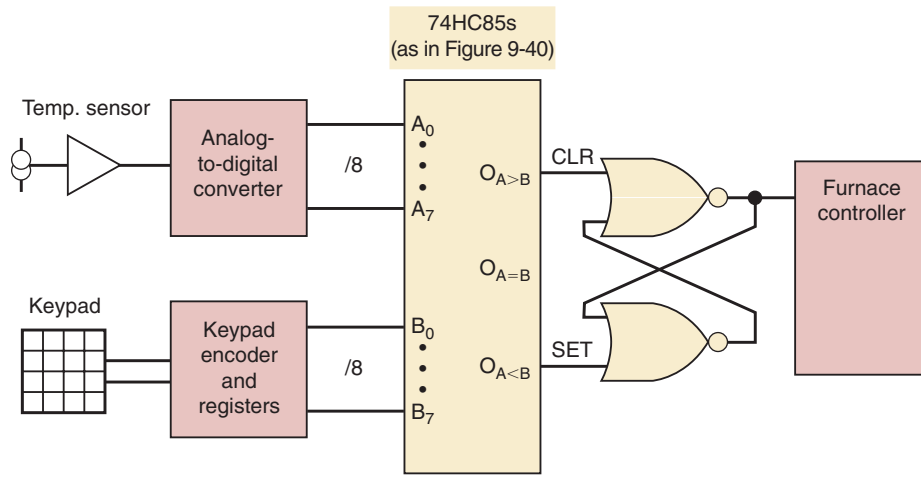
**EXAMPLE 9-16**

Consider a digital thermostat in which the measured room temperature is converted to a digital number and applied to the  $A$  inputs of a comparator. The desired room temperature, entered from a keypad, is stored in a register that is connected to the  $B$  inputs. If  $A < B$ , the furnace should be activated to heat the room. The furnace should continue to heat while  $A = B$  and shut off when  $A > B$ . As the room cools off, the furnace should stay off while  $A = B$  and turn on again when  $A < B$ . What digital circuit can be used to interface a magnitude comparator to a furnace to perform the thermostat control application described here?

**Solution**

Using the  $O_{A<B}$  output to drive the furnace directly would cause it to turn off as soon as the values become equal. This can cause severe on/off cycling of the furnace when the actual temperature is very close to the boundary between  $A < B$  and  $A = B$ . By using a NOR gate SET-CLEAR latch circuit (refer to Chapter 5) as shown in Figure 9-41, the system will operate as described. Notice that  $O_{A<B}$  is connected to the SET input and  $O_{A>B}$  is connected to the CLEAR input of the latch. When the temperature is hotter than desired, it clears the latch, shutting off the furnace. When the temperature is cooler than desired, it sets the latch, turning the furnace on.

**FIGURE 9-41** Magnitude comparator used in a digital thermostat.



### OUTCOME ASSESSMENT QUESTIONS

1. What is the purpose of the cascading inputs of the 74HC85?
2. What are the outputs of a 74HC85 with the following inputs:  $A_3A_2A_1A_0 = B_3B_2B_1B_0 = 1001$ ,  $I_{A>B} = I_{A<B} = 0$ , and  $I_{A=B} = 1$ ?
3. Why are there no cascading inputs on a Quartus comparator megafunction?

## 9-11 CODE CONVERTERS

### OUTCOME

Upon completion of this section, you will be able to:

- Describe circuits that convert from one coding method to another.

**TABLE 9-6** Common conversions.

BCD to 7-segment
BCD to binary
Binary to BCD
Binary to Gray code
Gray code to binary

A code converter is a logic circuit that changes data presented in one type of binary code to another type of binary code. The BCD-to-7-segment decoder-driver that we presented earlier is a code converter because it changes a BCD input code to the 7-segment code needed by the LED display. A partial list of some of the more common code conversions is given in Table 9-6.

As an example of a code converter circuit, let's consider a BCD-to-binary converter. Before we get started on the circuit implementation, we should review the BCD representation.

Two-digit decimal values ranging from 00 to 99 can be represented in BCD by two four-bit code groups. For example,  $57_{10}$  is represented as

$$\begin{array}{cc} 5 & 7 \\ \overline{0101} & \overline{0111} \end{array} \quad (\text{BCD})$$

The straight binary representation for decimal 57 is

$$57_{10} = 111001_2$$

The largest two-digit decimal value of 99 has the following representations:

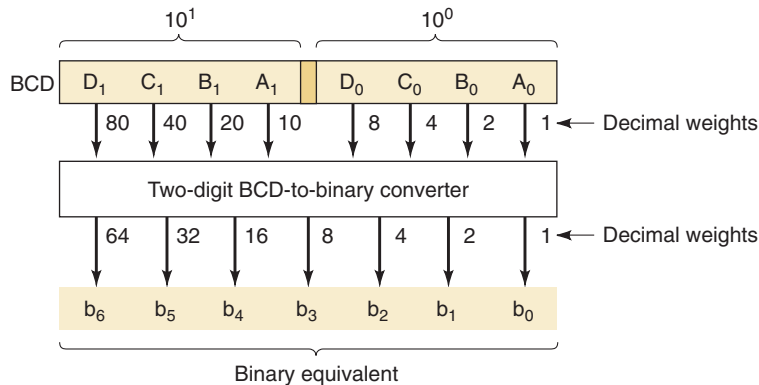
$$99_{10} = 10011001 (\text{BCD}) = 1100011_2$$

Note that the binary representation requires only seven bits.

### Basic Idea

The diagram of Figure 9-42 shows the basic idea for a two-digit BCD-to-binary converter. The inputs to the converter are the two four-bit code groups  $D_0C_0B_0A_0$ , representing the  $10^0$  or units digit, and  $D_1C_1B_1A_1$ , representing the  $10^1$  or tens digit of the decimal value. The outputs from the converter are  $b_6b_5b_4b_3b_2b_1b_0$ , the seven bits of the binary equivalent of the same decimal value. Note the difference in the decimal weights of the BCD bits and those of the binary bits.

**FIGURE 9-42** Basic idea of a two-digit BCD-to-binary converter.



A typical use of a BCD-to-binary converter would be where BCD data from an instrument such as a DMM (digital multimeter) are being transferred to a computer for storage or processing. The data must be converted to binary so that they can be operated on in binary by the computer ALU, which may not have the capability of performing arithmetic operations on BCD data. The BCD-to-binary conversion can be accomplished with either hardware or software. The hardware method (which we will look at momentarily) is generally faster but requires extra circuitry. The software method uses no extra circuitry, but it takes more time because the software does the conversion step by step. The method chosen in a particular application depends on whether or not conversion time is an important consideration.

### Conversion Process

The bits in a BCD representation have decimal weights that are 8, 4, 2, 1 within each code group but that differ by a factor of 10 from one code group (decimal digit) to the next. Figure 9-42 shows the bit weights for the two-digit BCD representation.



**TABLE 9-7** Binary equivalents of decimal weights of each BCD bit.

BCD Bit	Decimal Weight	Binary Equivalent							
		$b_6$	$b_5$	$b_4$	$b_3$	$b_2$	$b_1$	$b_0$	
$A_0$	1	0	0	0	0	0	0	1	
$B_0$	2	0	0	0	0	0	1	0	
$C_0$	4	0	0	0	0	1	0	0	
$D_0$	8	0	0	0	1	0	0	0	
$A_1$	10	0	0	0	1	0	1	0	
$B_1$	20	0	0	1	0	1	0	0	
$C_1$	40	0	1	0	1	0	0	0	
$D_1$	80	1	0	1	0	0	0	0	

The decimal weight of each bit in the BCD representation can be converted to its binary equivalent. The results are given in Table 9-7. Using these weights, we can perform the BCD-to-binary conversion by simply doing the following:

**Compute the binary sum of the binary equivalents of all bits in the BCD representation that are 1s.**

The following example will illustrate.

#### EXAMPLE 9-17

Convert 01010010 (BCD for decimal 52) to binary. Repeat for 10010101 (decimal 95).

#### Solution

Write down the binary equivalents for all the 1s in the BCD representation. Then add them all together in binary.

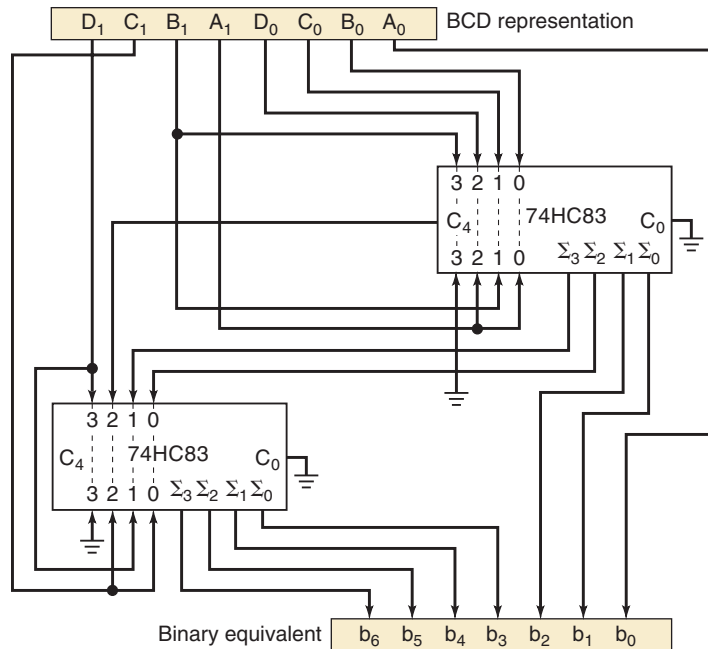
$$\begin{array}{r}
 01010010 \quad (\text{BCD}) \\
 \begin{array}{l}
 \rightarrow 000010 \quad (\text{binary for } 2) \\
 \rightarrow 0001010 \quad (\text{binary for } 10) \\
 \rightarrow + 0101000 \quad (\text{binary for } 40) \\
 \hline
 0110100 \quad (\text{binary for } 52)
 \end{array} \\
 10010101 \quad (\text{BCD}) \\
 \begin{array}{l}
 \rightarrow 0000001 \quad (\text{binary for } 1) \\
 \rightarrow 0000100 \quad (\text{binary for } 4) \\
 \rightarrow 0001010 \quad (\text{binary for } 10) \\
 \rightarrow + 1010000 \quad (\text{binary for } 80) \\
 \hline
 1011111 \quad (\text{binary for } 95)
 \end{array}
 \end{array}$$

### Circuit Implementation

Clearly, one way to implement the logic circuit that performs this conversion process is to use binary adder circuits. Figure 9-43 shows how two 74HC83 four-bit parallel adders can be wired to perform the conversion. This is one of several possible adder arrangements that will work. You may want to review the operation of this IC in Section 6-14.

The two adder ICs perform the addition of the BCD bits in the proper combinations according to Table 9-7. For instance, Table 9-7 shows that  $A_0$  is the only BCD bit that contributes to the LSB,  $b_0$ , of the binary equivalent. Because there is no carry into this bit position,  $A_0$  is connected directly as

**FIGURE 9-43** BCD-to-binary converter implemented with 74HC83 four-bit parallel adders.



output  $b_0$ . The table also shows that only BCD bits  $B_0$  and  $A_1$  contribute to bit  $b_1$  of the binary output. These two bits are combined in the upper-right adder to produce output  $b_1$ . Likewise, only BCD bits  $D_0$ ,  $A_1$ , and  $C_1$  contribute to bit  $b_3$ . The upper-right adder combines  $D_0$  and  $A_1$  to generate  $\Sigma_2$ , which is connected to the lower-left adder, where  $C_1$  is added to it to produce  $b_3$ .

### EXAMPLE 9-18

The BCD representation for decimal 56 is applied to the converter of Figure 9-43. Determine the  $\Sigma$  outputs from each adder and the final binary output.

#### Solution

Write down the bits of the BCD representation 01010110 on the circuit diagram. Because  $A_0 = 0$ , the  $b_0$  bit of the output is 0.

The top inputs to the upper adder are 0011. The bottom inputs are 0101. This adder adds these to produce

$$\begin{array}{r} 0011 \\ +0101 \\ \hline 1000 = \Sigma_3\Sigma_2\Sigma_1\Sigma_0 \text{ outputs of the upper adder} \end{array}$$

The  $\Sigma_1$  and  $\Sigma_0$  bits become binary outputs  $b_2$  and  $b_1$ , respectively. The  $\Sigma_3$  and  $\Sigma_2$  bits are fed to the lower adder. The top inputs to the lower adder are therefore 0010. The bottom inputs are 0101. This adder adds these to produce

$$\begin{array}{r} 0010 \\ +0101 \\ \hline 0111 = \Sigma_3\Sigma_2\Sigma_1\Sigma_0 \text{ outputs of the lower adder} \end{array}$$

These bits become  $b_6b_5b_4b_3$ , respectively.

Thus, we have  $b_6b_5b_4b_3b_2b_1b_0 = 0111000\ 0111000$  as the correct binary equivalent for decimal 56.

## Other Code Converter Implementations

Whereas all types of code converters can be made by combining logic gates, adder circuits, or other combinational logic, the circuitry can become quite complex, requiring many ICs. It is often more efficient to use a read-only memory (ROM) or programmable logic device (PLD) to function as a code converter. As we will see in Chapters 12 and 13, these devices contain the equivalent of hundreds of logic gates, and they can be programmed to provide a wide range of logic functions.

### OUTCOME ASSESSMENT QUESTIONS

1. What is a code converter?
2. How many binary outputs would a three-digit BCD-to-binary converter have?

## 9-12 DATA BUSING

### OUTCOMES

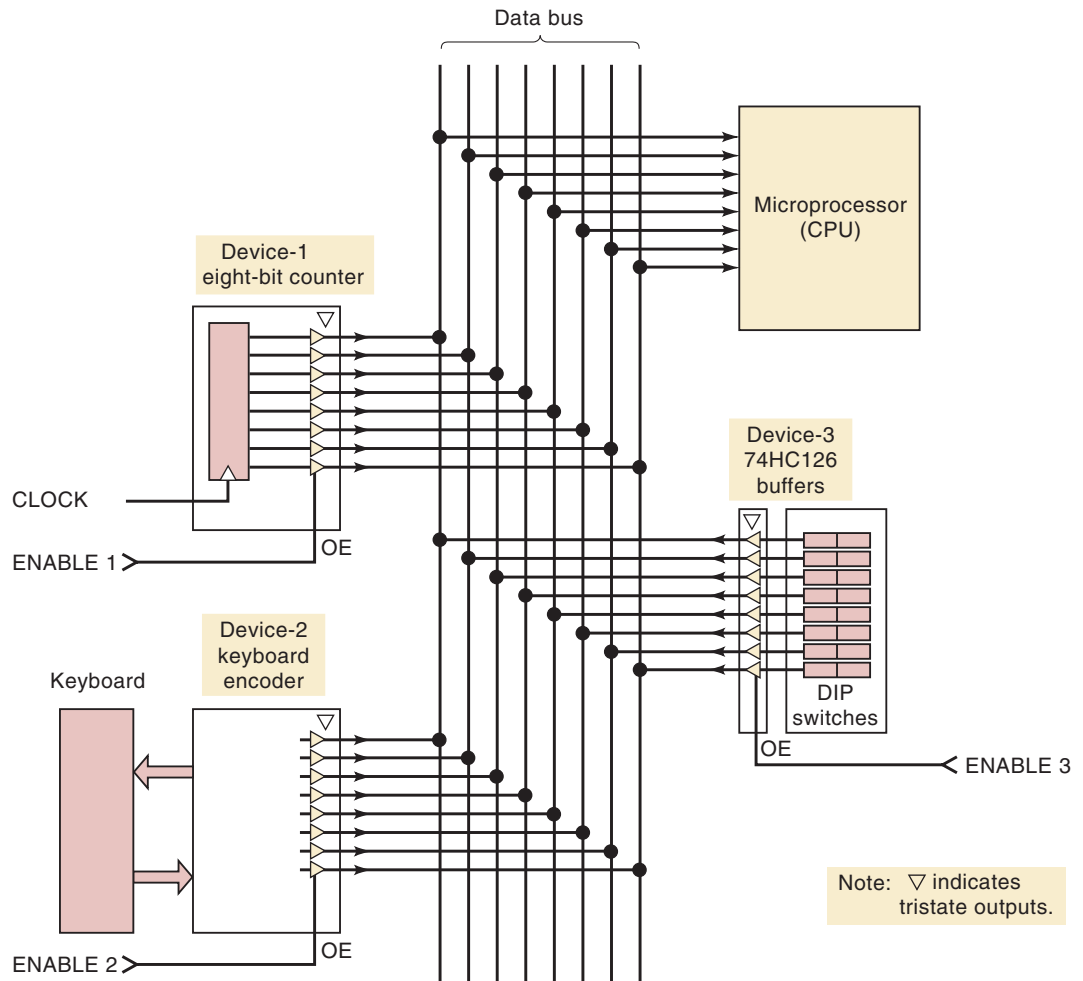
*Upon completion of this section, you will be able to:*

- Define a data bus.
- Use tristate logic to interface devices to a data bus.

In computers, the transfer of data takes place over a common set of connecting lines called a **data bus**. In these bus-organized computers, many different devices can have their outputs and inputs tied to the common data bus lines. Because of this, the devices that are tied to the data bus will often have tristate outputs, or they will be tied to the data bus through tristate buffers.

Some of the devices that are commonly connected to a data bus are (1) microprocessors; (2) semiconductor memory chips, covered in Chapter 12; and (3) digital-to-analog converters (DACs) and analog-to-digital converters (ADCs), described in Chapter 11.

Figure 9-44 illustrates a typical situation in which a microprocessor (the CPU chip in a microcomputer) is connected to several devices over an eight-line data bus. The data bus is simply a collection of conducting paths over which digital data are transmitted from one device to another. Each device provides an eight-bit output that is sent to the inputs of the microprocessor over the eight-line data bus. Clearly, because the outputs of each of the three devices are connected to the same microprocessor inputs over the data bus conducting paths, we must be aware of bus contention problems (Section 8-12), where two or more signals tied to the same bus line are active and are essentially fighting each other. Bus contention is avoided if the devices have tristate outputs or are connected to the bus through tristate buffers (Section 8-12). The output enable inputs (*OE*) to each device (or its buffer) are used to ensure that no more than one device's outputs are active at a given time.



**FIGURE 9-44** Three different devices can transmit eight-bit data over an eight-line data bus to a microprocessor; only one device at a time is enabled so that bus contention is avoided.

### EXAMPLE 9-19

- For Figure 9-44, describe the conditions necessary to transmit data from device 3 to the microprocessor.
- What will the status of the data bus be when none of the devices is enabled?

### Solution

- ENABLE 3 must be activated; ENABLE 1 and ENABLE 2 must be in their inactive state. This will put the outputs of device 1 and device 2 in the Hi-Z state and essentially disconnect them from the bus. The outputs of device 3 will be activated so that their logic levels will appear on the data bus lines and be transmitted to the inputs of the microprocessor. We can visualize this by covering up device 1 and device 2 as if they are not even part of the circuit; then we are left with device 3 alone connected to the microprocessor over the data bus.

- (b) If none of the device enable inputs are activated, all of the device outputs are in the Hi-Z state. This disconnects all device outputs from the bus. Thus, there is no definite logic level on any of the data bus lines; they are in the indeterminate state. This condition is known as a **floating bus**, and each data bus line is said to be in a *floating* (indeterminate) state. An oscilloscope display of a floating bus line would be unpredictable. A logic probe would indicate an indeterminate logic level.

### OUTCOME ASSESSMENT QUESTIONS

1. What is meant by the term *data bus*?
2. What is *bus contention*, and what must be done to prevent it?
3. What is a *floating bus*?

## 9-13 THE 74ALS173/HC173 TRISTATE REGISTER

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the inputs and outputs of a typical tristate register.
- Predict the output of a 74173 for any given set of inputs.

The devices connected to a data bus will contain registers (usually flip-flops) that hold the device data. The outputs of these registers are usually connected to tristate buffers that allow them to be tied to a data bus. We will demonstrate the details of a tristate data bus operation by using an IC register that includes the tristate buffers on the same chip: the TTL 74ALS173 (also available in CMOS 74HC173 versions). Its logic diagram and truth table are shown in Figure 9-45.

The 74ALS173 is a four-bit register with parallel in/parallel out capability. Note that the FF outputs are connected to tristate buffers that provide outputs  $O_0$  through  $O_3$ . Also note that the data inputs  $D_0$  through  $D_3$  are connected to the  $D$  inputs of the register FFs through logic circuitry. This logic allows two modes of operation: (1) *load*, where the data at inputs  $D_0$  to  $D_3$  are transferred into the FFs on the PGT of the clock pulse at  $CP$ ; and (2) *hold*, where the data in the register do not change when the PGT of  $CP$  occurs.

### EXAMPLE 9-20

For a 74ALS173,

- (a) What input conditions will produce the load operation?
- (b) What input conditions will produce the hold operation?
- (c) What input conditions will allow the internal register outputs to appear at  $O_0$  to  $O_3$ ?

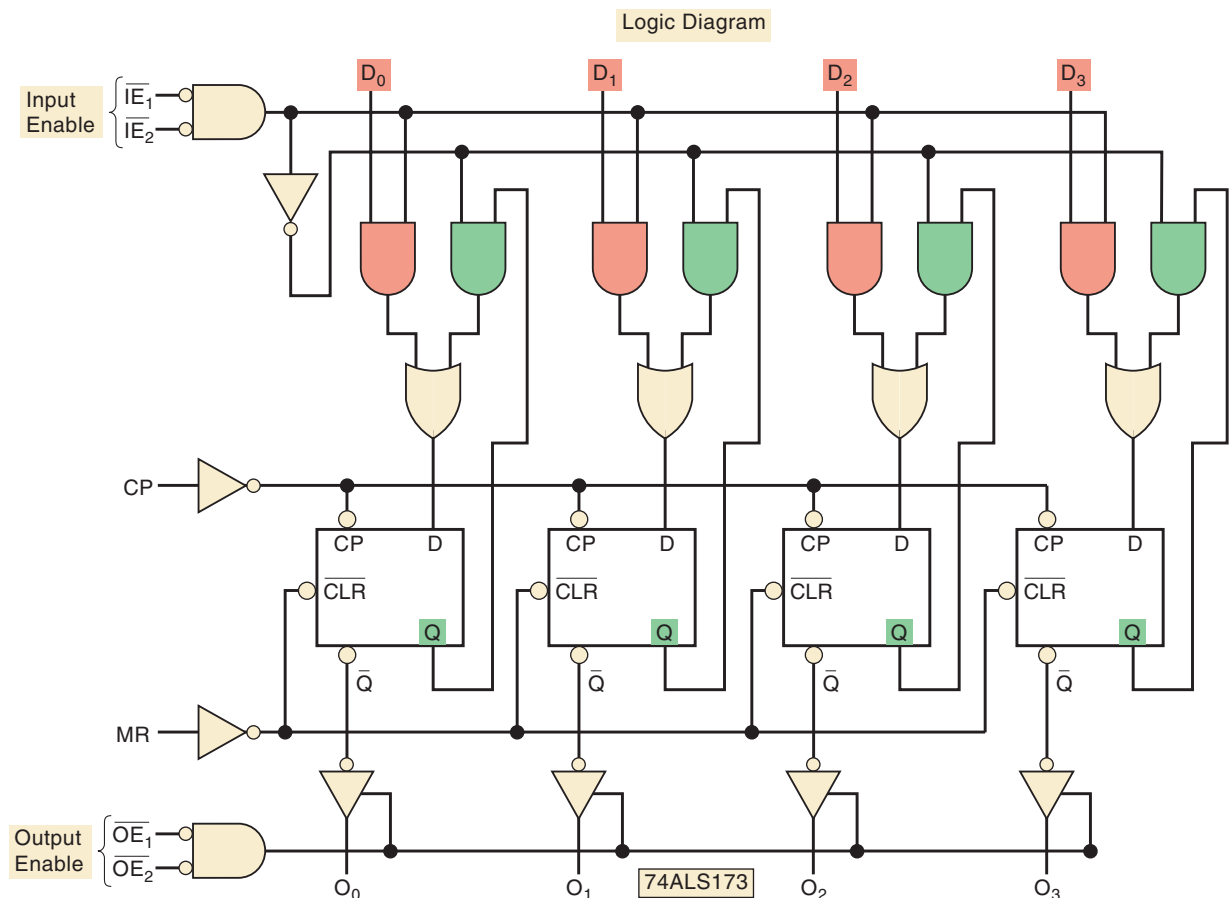
### Solution

- (a) The last two entries in the truth table of Figure 9-45 show that the  $Q$  output of each FF takes on the value present at its  $D$  input when a PGT occurs at  $CP$  provided that  $MR$  is LOW and *both* input-enable inputs,  $\overline{IE}_1$  and  $\overline{IE}_2$ , are LOW.
- (b) The third and fourth lines of the truth table state that when either  $\overline{IE}$  input is HIGH, the  $D$  inputs have no effect, and the  $Q$  outputs will retain their current values when the PGT occurs.

Inputs					FF Outputs
MR	CP	$\overline{IE}_1$	$\overline{IE}_2$	$D_n$	Q
H	X	X	X	X	L
L	L	X	X	X	$Q_0$
L	$\downarrow$	H	X	X	$Q_0$
L	$\downarrow$	X	H	X	$Q_0$
L	$\downarrow$	L	L	L	L
L	$\downarrow$	L	L	H	H

When either  $\overline{OE}_1$  or  $\overline{OE}_2$  is HIGH, the output is in the OFF state (high impedance); however, this does not affect the contents or sequential operating of the register.

H = HIGH voltage level       $Q_0$  = output prior to PGT  
 L = LOW voltage level  
 X = immaterial



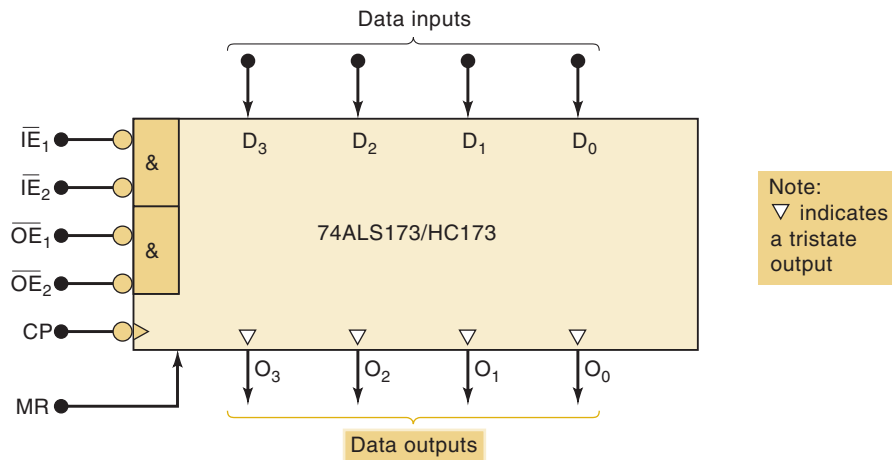
**FIGURE 9-45** Truth table and logic diagram for the 74ALS173 tristate register.

- (c) The output buffers are enabled when *both* output-enable inputs,  $\overline{OE}_1$  and  $\overline{OE}_2$ , are LOW. This will pass the register outputs through to the external outputs  $O_0$  to  $O_3$ . If either output-enable input is HIGH, the buffers will be disabled, and the outputs will be in the Hi-Z state.

Note that the  $\overline{OE}$  inputs have no effect on the data load operation. They are used only to control whether or not the register outputs are passed to the external outputs.

The logic symbol for the 74ALS173/HC173 is given in Figure 9-46. We have included the IEEE/ANSI “&” notation to indicate the AND relationship of the two pairs of enable inputs.

**FIGURE 9-46** Logic symbol for the 74ALS173/HC173 IC.



### OUTCOME ASSESSMENT QUESTIONS

1. Assume that both  $\overline{IE}$  inputs are LOW and that  $D_0D_1D_2D_3 = 1011$ . What logic levels are present at the FF  $D$  inputs?
2. *True or false:* The register cannot be loaded when the master reset input ( $MR$ ) is held HIGH.
3. What will the output levels be when  $MR = \text{HIGH}$  and both  $OE$  inputs are held low?

## 9-14 DATA BUS OPERATION

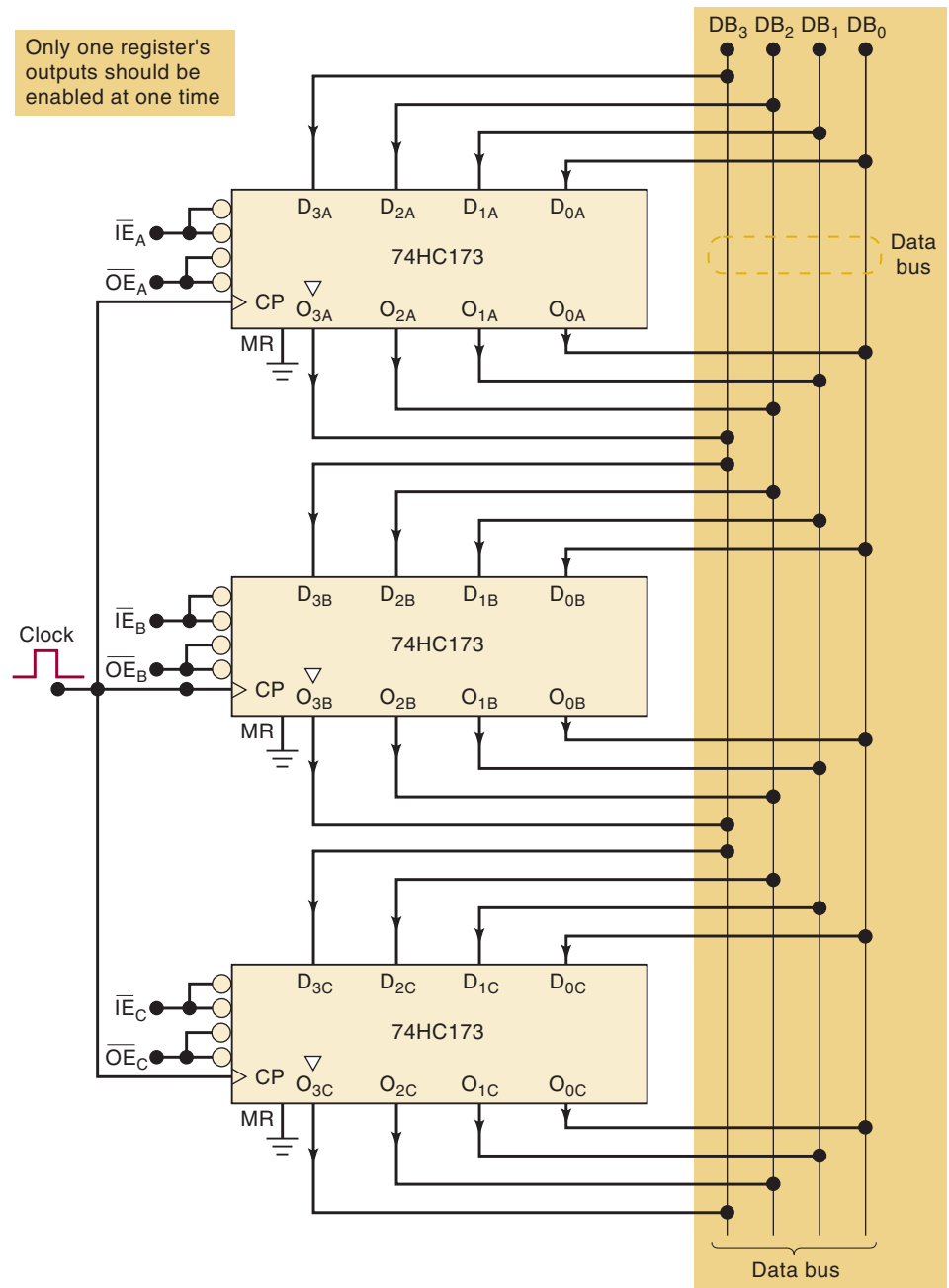
### OUTCOMES

Upon completion of this section, you will be able to:

- Describe data bus operation graphically, through timing diagrams, and verbally.
- Analyze data transfers between devices on a bus.

The data bus is very important in computer systems, and its significance will not be appreciated until our later studies of memories and microprocessors. For now, we will illustrate the data bus operation for register-to-register data transfer. Figure 9-47 shows a bus-organized system for three 74HC173 tristate registers. Note that each register has its pair of  $\overline{OE}$  inputs tied together as one  $\overline{OE}$  input and likewise for the  $\overline{IE}$  inputs. Also note that the registers will be referred to as registers A, B, and C from top to bottom. This is indicated by the subscripts on each input and output.

In this arrangement, the data bus consists of four lines labeled  $DB_0$  to  $DB_3$ . Corresponding outputs of each register are connected to the same data bus line (e.g.,  $O_{3A}$ ,  $O_{3B}$ , and  $O_{3C}$  are connected to  $DB_3$ ). Because the three registers have their outputs connected together, it is imperative that only one register have its outputs enabled and that the other two register outputs remain in the Hi-Z state. Otherwise, there will be bus contention (two or more sets of outputs fighting each other), producing uncertain levels on the bus and possible damage to the register output buffers.



**FIGURE 9-47** Tristate registers connected to a data bus.

Corresponding register inputs are also tied to the same bus line (e.g.,  $D_{3A}$ ,  $D_{3B}$ , and  $D_{3C}$  are tied to  $DB_3$ ). Thus, the levels on the bus will always be ready to be transferred to one or more of the registers depending on the  $\overline{IE}$  inputs.

### Data Transfer Operation

The contents of any one of the three registers can be parallel-transferred over the data bus to one of the other registers through the proper application of logic levels to the register enable inputs. In a typical system,



the control unit of a computer (i.e., the CPU) will generate the signals that select which register will put its data on the data bus and which one will take the data from the data bus. The following example will illustrate this.

### EXAMPLE 9-21

Describe the input signal requirements for transferring  $[A] \rightarrow [C]$ .

#### Solution

First of all, only register A should have its outputs enabled. That is, we need

$$\overline{OE}_A = 0 \quad \overline{OE}_B = \overline{OE}_C = 1$$

This will place the contents of register A onto the data bus lines.

Next, only register C should have its inputs enabled. For this, we want

$$\overline{IE}_C = 0 \quad \overline{IE}_A = \overline{IE}_B = 1$$

This will allow only register C to accept data from the data bus when the PGT of the clock signal occurs.

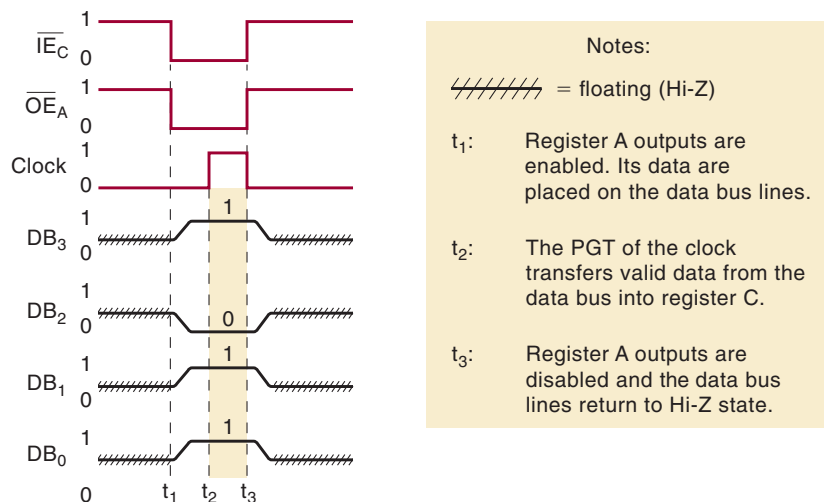
Finally, a clock pulse is required to transfer the data from the bus into the register C flip-flops.

### Bus Signals

The timing diagram in Figure 9-48 shows the various signals involved in the transfer of the data 1011 from register A to register C. The  $\overline{IE}$  and  $\overline{OE}$  lines that are not shown are assumed to be in their inactive HIGH state. Prior to time  $t_1$ , the  $\overline{IE}_C$  and  $\overline{OE}_A$  lines are also HIGH, so that all of the register outputs are disabled, and none of the registers will be placing their data on the bus lines. In other words, the data bus lines are in the Hi-Z or “floating” state as represented by the hatched lines on the timing diagram. The Hi-Z state does not correspond to any particular voltage level.

At  $t_1$  the  $\overline{IE}_C$  and  $\overline{OE}_A$  inputs are activated. The outputs of register A are enabled, and they start changing the data bus lines  $DB_3$  through  $DB_0$  from

**FIGURE 9-48** Signal activity during the transfer of the data 1011 from register A to register C.



the Hi-Z state to the logic levels 1011. After allowing time for the logic levels to stabilize on the bus, the PGT of the clock is applied at  $t_2$ . This PGT will transfer these logic levels into register C because  $\overline{I\overline{E}}_C$  is active. If the PGT occurs before the data bus has valid logic levels, unpredictable data will be transferred into C.

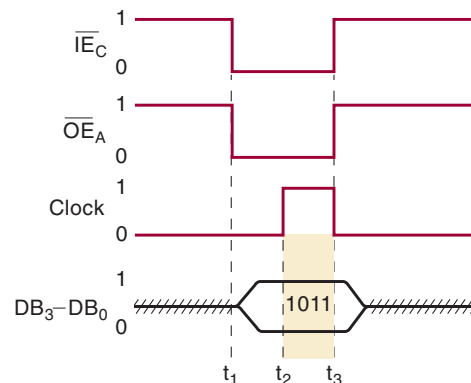
At  $t_3$ , the  $\overline{I\overline{E}}_C$  and  $\overline{O\overline{E}}_A$  lines return to the inactive state. As a result, register A's outputs go to the Hi-Z state. This removes the register A output data from the bus lines, and the bus lines return to the Hi-Z state.

Note that the data bus lines show valid logic levels only during the time interval when register A's outputs are enabled. At all other times, the data bus lines are floating, and there is no way to predict easily what they would look like if displayed on an oscilloscope. A logic probe would give an "indeterminate" indication if it were monitoring a floating bus line. Also note the relatively slow rate at which the signals on the data bus lines are changing. Although this effect has been somewhat exaggerated in the diagram, it is a characteristic common to bus systems and is caused by the capacitive load on each line. This load consists of a combination of parasitic capacitance and the capacitances contributed by each input and output connected to the line.

### Simplified Bus Timing Diagram

The timing diagram in Figure 9-48 shows the signals on each of the four data bus lines. This same kind of signal activity occurs in digital systems that use the more common data buses of 8, 16, or 32 lines. For these larger buses, the timing diagrams like Figure 9-48 would get excessively large and cumbersome. There is a simplified method for showing the signal activity that occurs on a set of bus lines that uses only a single timing waveform to represent the complete set of bus lines. This is illustrated in Figure 9-49 for the same data transfer situation depicted in Figure 9-48. Notice how the data bus activity is represented. Especially note how the valid data 1011 are indicated on the diagram during the  $t_2$ – $t_3$  interval. We will generally use this simplified bus timing diagram from now on.

**FIGURE 9-49** Simplified way to show signal activity on data bus lines.



### Expanding the Bus

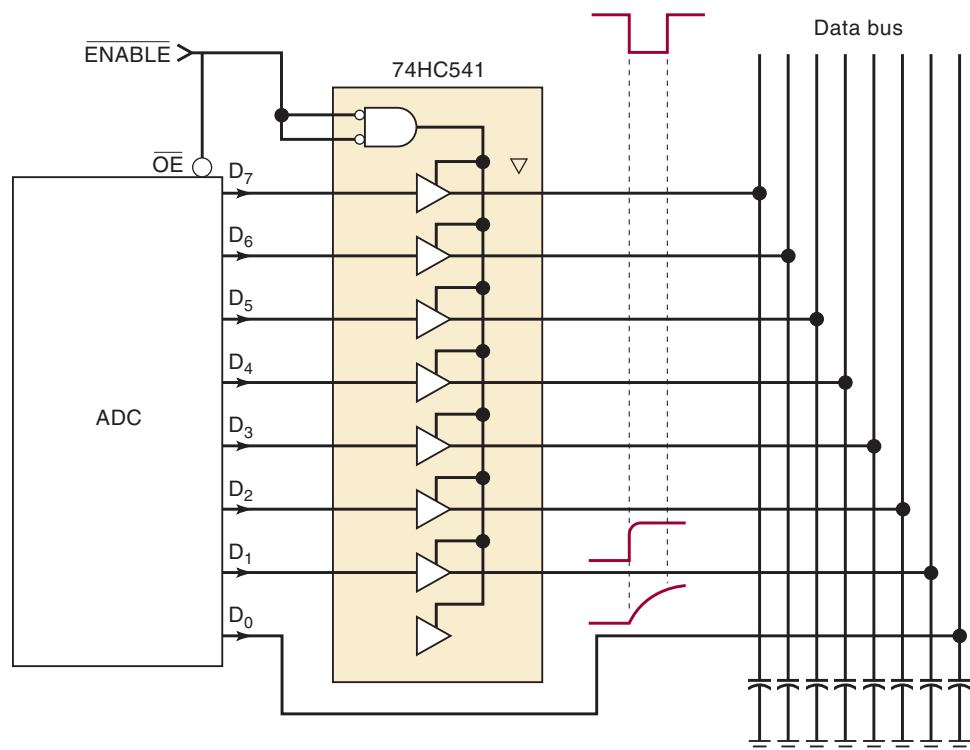
The data transfer operation of the four-line data bus of Figure 9-47 is typical of the operation of larger data buses found in most computers and other digital systems, usually the 8-, 16-, or 32-line data buses. These larger buses generally have many more than three devices tied to the bus, but the basic data transfer operation is the same: *one device has its outputs enabled so that*

its data are placed on the data bus; another device has its inputs enabled so that it can take these data off the bus and latch them into its internal circuitry on the appropriate clock edge.

The number of lines on the data bus will depend on the size of the data **word** (unit of data) that is to be transferred over the bus. A computer that has an eight-bit word size will have an eight-line data bus, a computer that has a 16-bit word size will have a 16-line data bus, and so on. The number of devices connected to a data bus varies from one computer to another and depends on factors such as how much memory the computer has and the number of input and output devices that must communicate with the CPU over the data bus.

All device outputs must be tied to the bus through tristate buffers. Some devices, such as the 74173 register, have these buffers on the same chip. Other devices will need to be connected to the bus through an IC called a **bus driver**. A bus driver IC has tristate outputs with a very low output impedance that can rapidly charge and discharge the bus capacitance. This bus capacitance represents the cumulative effect of all of the parasitic capacitances of the different inputs and outputs tied to the bus, and it can cause deterioration of the bus signal transition times if they are not driven from a low-impedance signal source. Figure 9-50 shows a 74HC541 octal bus driver IC connecting the outputs of an eight-bit analog-to-digital converter to a data bus. The ADC has tristate outputs but lacks the drive capability to charge the bus capacitance (shown as capacitors to ground in the drawing). Notice that data bit 0 is driving the bus directly, without the assistance of the bus driver. If the transition time is slow enough, the voltage may never reach a HIGH logic level in the allotted enable time. The bus driver's two enable inputs are tied together so that a LOW on the common *ENABLE* line will allow the ADC's outputs through the buffers and onto the the data bus, from which they can be transferred to another device.

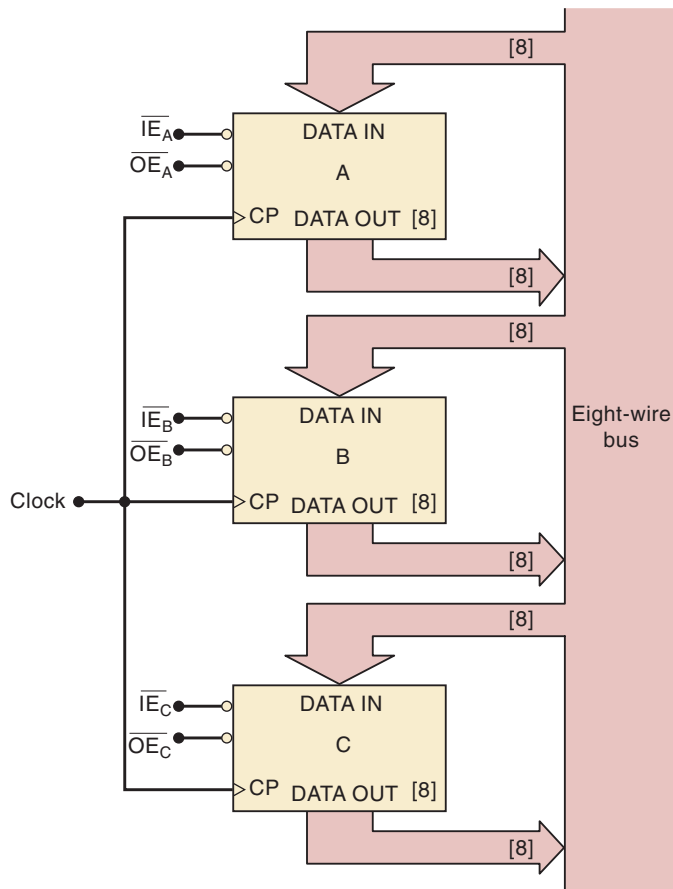
**FIGURE 9-50** A 74HC541 octal bus driver connects the outputs of an analog-to-digital converter (ADC) to an eight-line data bus. The  $D_0$  output connects directly to the bus showing the capacitive effects.



## Simplified Bus Representation

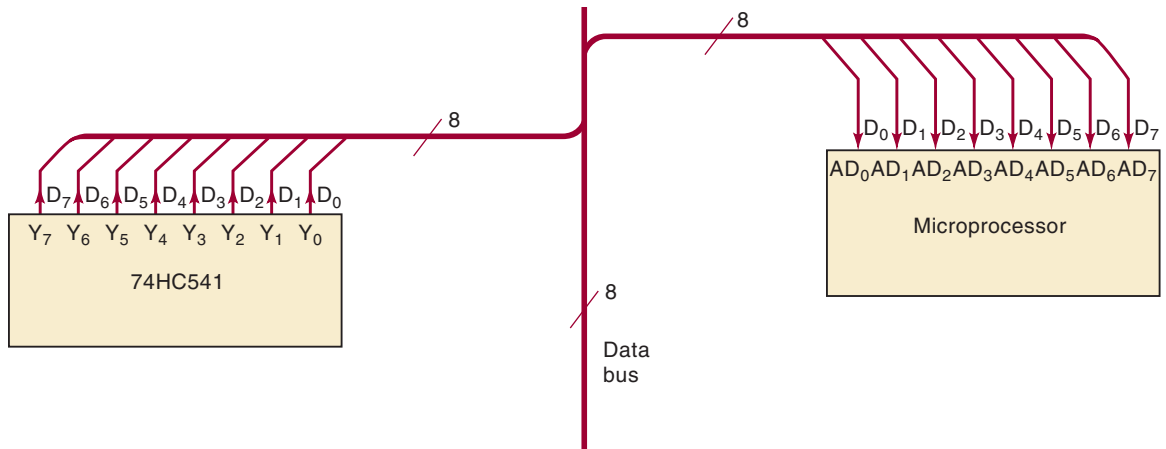
Usually, many devices are connected to the same data bus. On a circuit schematic, this can produce a confusing array of lines and connections. For this reason, a more simplified representation of data bus connections is often used on block diagrams and in some circuit schematics. One type of simplified representation is shown in Figure 9-51 for an eight-line data bus.

**FIGURE 9-51** Simplified representation of bus arrangement.



The connections to and from the data bus are represented by wide arrows. The numbers in brackets indicate the number of bits that each register contains, as well as the number of lines connecting the register inputs and outputs to the bus.

Another common method for representing buses on a schematic is presented in Figure 9-52 for an eight-line data bus. It shows the eight individual output lines from a 74HC541 bus driver labeled  $D_7$ – $D_0$  bundled (not connected) together and shown as a single line. These bundled data output lines are then connected to the data bus, which is also shown as one line (i.e., the eight data bus lines are bundled together). The “/8” notation indicates the number of lines represented by each bundle. This bundle method is used to represent the connections from the data bus to the eight microprocessor data inputs. When the bundle method is used, it is very important to label both ends of every wire that is in the bundle because the connection cannot be traced visually on the diagram.



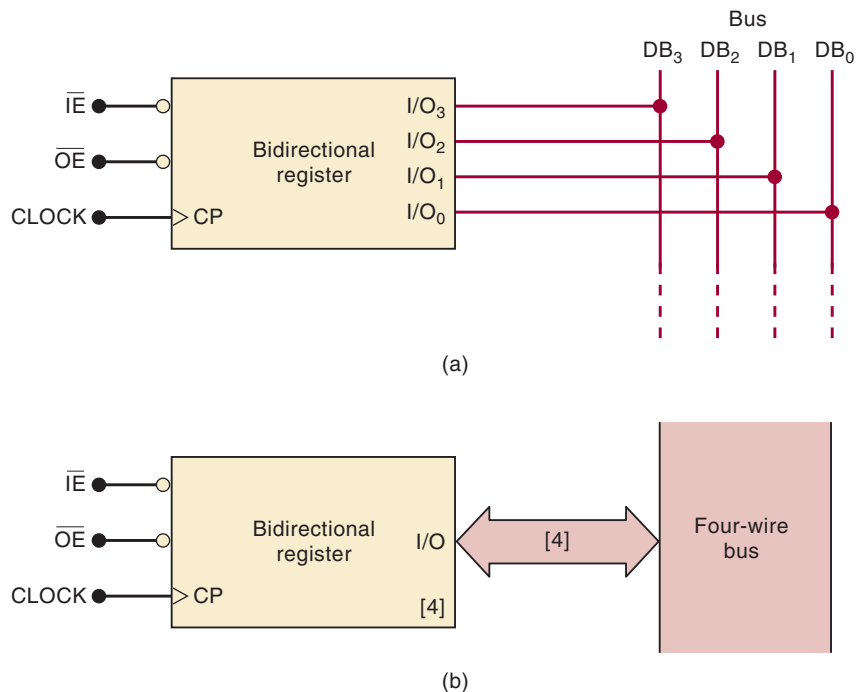
**FIGURE 9-52** Bundle method for simplified representation of data bus connections. The “/8” denotes an eight-line data bus.

### Bidirectional Busing

Each register in Figure 9-47 has both its inputs and its outputs connected to the data bus, so that corresponding inputs and outputs are shorted together. For example, each register has output  $O_2$  connected to input  $D_2$  because of their common connection to  $DB_2$ . When an integrated circuit is used in this way, it is unnecessary to have both input pins and output pins.

Because inputs and outputs are often connected together in bus systems, IC manufacturers have developed ICs that connect inputs and outputs together *internal* to the chip in order to reduce the number of IC pins and the number of connections to the bus. Figure 9-53 illustrates this for a four-bit register. The separate data input lines ( $D_0$  to  $D_3$ ) and output lines ( $O_0$  to  $O_3$ ) have been replaced by input/output lines ( $I/O_0$  to  $I/O_3$ ).

**FIGURE 9-53**  
Bidirectional register  
connected to data bus.



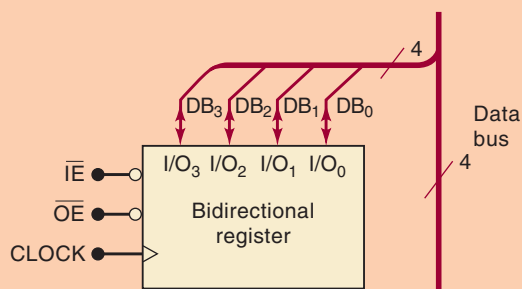
Each I/O line will function as either an input or an output depending on the states of the enable inputs. Thus, they are called **bidirectional data lines**. The 74ALS299 is an eight-bit register with common I/O lines. Many memory ICs and microprocessors have bidirectional transfer of data.

We will return to the important topic of data busing in our comprehensive coverage of memory systems in Chapter 12.

### OUTCOME ASSESSMENT QUESTIONS

1. What will happen if  $\overline{OE}_A = \overline{OE}_B = \text{LOW}$  in Figure 9-47?
2. What logic level is on a data bus line when all devices tied to the bus are disabled?
3. What is the function of a bus driver?
4. What are the reasons for having registers with common I/O lines?
5. Redraw Figure 9-53(a) using the bundled line representation. (The answer is shown in Figure 9-54.)

**FIGURE 9-54** Bus notation showing four data bus lines bundled together.



## 9-15 DECODERS USING HDL

### OUTCOME

Upon completion of this section, you will be able to:

- Use HDL to describe any type of decoder.

Section 9-1 introduced the decoder as a device that can recognize a binary number on its input and activate a corresponding output. Specifically, the 74138 1-of-8 decoder was presented. It uses three binary inputs to activate one of the eight outputs when the chip is enabled. In order to study HDL methods for implementing the types of digital devices that are covered in this chapter, we will focus primarily on conventional MSI parts, which have been discussed earlier. Not only is the operation of these devices already described in this book, but further reference material is readily available in logic data books. In all of these cases, it is vital that you understand what the device is supposed to do before trying to dissect the HDL code that describes it.

In actual practice, we are not recommending, for example, that new code be written to perform the task of a 74138. After all, there is a macrofunction already available that works exactly like this standard part.

Using these devices as examples and showing the HDL techniques used to create them opens the door for embellishment of these devices so that a circuit that will uniquely fit the application at hand can be described. In some instances, we will add our own embellishments to a circuit that has been described; in other instances, we will describe a simpler version of a part in order to focus on the core principle in HDL and avoid other confusing features.

The methods used to define the inputs and outputs should take into consideration the purpose of these signals. In the case of a 1-of-8 decoder such as the 74138 described in Figure 9-3, there are three enable inputs ( $\bar{E}_1$ ,  $\bar{E}_2$ , and  $E_3$ ) that should be described as individual inputs to the device. On the other hand, the binary inputs that are to be decoded ( $A_2$ ,  $A_1$ ,  $A_0$ ) should be described as three-bit numbers. The outputs can be described as eight individual bits. They can also be described as an array of eight bits, with output 0 represented by element 0 in the array, and so on, to output 7 represented by element 7. Depending on the way the code is written, one strategy may be easier to write than the other. Generally, using individual names can make the purpose of each I/O bit clearer, and using bit arrays makes it easier to write the code.

When an application such as a decoder calls for a unique response from the circuit corresponding to each combination of its input variables, the two methods that best serve this purpose are the CASE construct and the truth TABLE. The interesting aspect of this decoder is that the output response should happen only when *all* the enables are activated. If any of the enables are not in their active state, it should cause all the outputs to go HIGH. Each of the examples that follow will demonstrate ways to decode the input number only when *all* of the enables are activated.

## AHDL DECODERS

The first illustration of an AHDL decoder, shown in Figure 9-55, is intended to demonstrate the use of a CASE construct that is evaluated only under the condition that all enables are active. The outputs must all revert back to HIGH as soon as any enable is deactivated. This example also illustrates a way to accomplish this without explicitly assigning a value to each output for each case, and it uses individually named output bits.

Line 3 defines the three-bit binary number that will be decoded. Line 4 defines the three enable inputs, and line 5 specifically names each output. The unique property of this solution is the use of the **DEFAULTS** keyword in AHDL (lines 10 to 13) to establish a value for variables that are not specified elsewhere in the code. This maneuver allows each case to force one bit LOW without specifically stating that the others must go HIGH.

The next illustration, in Figure 9-56, is intended to demonstrate the same decoder using the truth table approach. Notice that the outputs are defined as bit arrays but are still numbered  $y[7]$  down to  $y[0]$ . The unique aspect of this code is the use of the don't-care values in the truth table. Line 11 is used to concatenate the six input bits into a single variable (bit array) named *inputs[]*. Notice that in lines 14, 15, and 16 of the table, only one bit value is specified as 1 or 0. The others are all in the don't-care state (X). Line 14 says, "As long as  $e3$  is *not* enabled, it does not matter what the other inputs are doing; the outputs will be HIGH." Lines 15 and 16 do the same thing, making sure that if  $e2bar$  or  $e1bar$  is HIGH (disabled), the outputs

**FIGURE 9-55** AHDL equivalent to the 74138 decoder.

```

1  SUBDESIGN fig9_55
2  (
3      a[2..0]                :INPUT;  -- binary inputs
4      e3, e2bar, elbar      :INPUT;  -- enable inputs
5      y7,y6,y5,y4,y3,y2,y1,y0 :OUTPUT; -- decoded outputs
6  )
7  VARIABLE
8      enable                :NODE;
9  BEGIN
10     DEFAULTS
11         y7=VCC;y6=VCC;y5=VCC;y4=VCC;
12         y3=VCC;y2=VCC;y1=VCC;y0=VCC; -- defaults all HIGH out
13     END DEFAULTS;
14     enable = e3 & !e2bar & !elbar; -- all enables activated
15     IF enable THEN
16         CASE a[] IS
17             WHEN 0 => y0 = GND;
18             WHEN 1 => y1 = GND;
19             WHEN 2 => y2 = GND;
20             WHEN 3 => y3 = GND;
21             WHEN 4 => y4 = GND;
22             WHEN 5 => y5 = GND;
23             WHEN 6 => y6 = GND;
24             WHEN 7 => y7 = GND;
25         END CASE;
26     END IF;
27 END;
```

**FIGURE 9-56** AHDL decoder using a TABLE.

```

1  SUBDESIGN fig9_56
2  (
3      a[2..0]                :INPUT;  -- decoder inputs
4      e3, e2bar, elbar      :INPUT;  -- enable inputs
5      y[7..0]               :OUTPUT;  -- decoded outputs
6  )
7  VARIABLE
8      inputs[5..0]          :NODE;  -- all 6 inputs combined
9
10 BEGIN
11     inputs[] = (e3, e2bar, elbar, a[]); -- concatenate the inputs
12     TABLE
13         inputs[] => y[];
14         B"0XXXXX" => B"11111111"; -- e3 not enabled
15         B"X1XXXX" => B"11111111"; -- e2bar disabled
16         B"XX1XXX" => B"11111111"; -- elbar disabled
17         B"100000" => B"11111110"; -- Y0 active
18         B"100001" => B"11111101"; -- Y1 active
19         B"100010" => B"11111011"; -- Y2 active
20         B"100011" => B"11110111"; -- Y3 active
21         B"100100" => B"11101111"; -- Y4 active
22         B"100101" => B"11011111"; -- Y5 active
23         B"100110" => B"10111111"; -- Y6 active
24         B"100111" => B"01111111"; -- Y7 active
25     END TABLE;
26 END;
```



will be HIGH. Lines 17 through 24 state that as long as the first three bits (enables) are “100,” the proper decoder output will be activated to correspond with the lower three bits of *inputs*[].

## VHDL DECODERS

The VHDL solution presented in Figure 9-57 essentially uses a truth table approach. The key strategy in this solution involves the concatenation of the three enable bits (*e3*, *e2bar*, *e1bar*) with the binary input *a* on line 11. The VHDL selected signal assignment is used to assign a value to a signal when a specific combination of inputs is present. Line 12 (WITH inputs SELECT) indicates that we are using the value of the intermediate signal *inputs* to determine which value is assigned to *y*. Each of the *y* outputs is listed on lines 13–20. Notice that only combinations that begin with 100 follow the WHEN clause on lines 13–20. This combination of *e3*, *e2bar*, and *e1bar* is necessary to make each of the enables active. Line 21 assigns a disabled state to each output when any combination other than 100 is present on the enable inputs.

```

1  ENTITY fig9_57 IS
2  PORT (
3      a                :IN BIT_VECTOR  (2 DOWNTO 0);
4      e3, e2bar, elbar :IN BIT;
5      y                :OUT BIT_VECTOR (7 DOWNTO 0)
6  );
7  END fig9_57;
8  ARCHITECTURE truth OF fig9_57 IS
9  SIGNAL inputs: BIT_VECTOR (5 DOWNTO 0); --combine enables w/ binary in
10 BEGIN
11     inputs <= e3 & e2bar & elbar & a;
12     WITH inputs SELECT
13         y <= "11111110" WHEN "100000", --Y0 active
14             "11111101" WHEN "100001", --Y1 active
15             "11111101" WHEN "100010", --Y2 active
16             "11110111" WHEN "100011", --Y3 active
17             "11101111" WHEN "100100", --Y4 active
18             "11011111" WHEN "100101", --Y5 active
19             "10111111" WHEN "100110", --Y6 active
20             "01111111" WHEN "100111", --Y7 active
21             "11111111" WHEN OTHERS;   --disabled
22 END truth;

```

FIGURE 9-57 VHDL equivalent to the 74138 decoder.

### OUTCOME ASSESSMENT QUESTIONS

1. What is the purpose of the 74138's three inputs *e3*, *e2bar*, and *e1bar*?
2. Name two AHDL methods to describe a decoder's operation.
3. Name two VHDL methods to describe a decoder's operation.

## 9-16 THE HDL 7-SEGMENT DECODER/DRIVER

---

### OUTCOME

Upon completion of this section, you will be able to:

- Use HDL to describe any type of display decoder/driver.

Section 9-2 described a BCD-to-7-segment decoder/driver. The standard part number for the circuit described is a 7447. In this section, we look into the HDL code necessary to produce a device that meets the same criteria as the 7447. Recall that the  $\overline{BI}$  (blanking input) is the overriding control that turns all segments off regardless of other input levels. The  $\overline{LT}$  (lamp test) input is used to test all the segments on the display by lighting them up. The  $\overline{RBO}$  (ripple blanking output) is designed to go LOW when  $\overline{RBI}$  (ripple blanking input) is LOW and the BCD input value is 0. Typically, in multiple-digit display applications, each  $\overline{RBO}$  pin is connected to the  $\overline{RBI}$  pin of the next digit to the right. This setup creates the feature of blanking all leading zeros in a display value without blanking zeros in the middle of a number. For example, the number 2002 would display as 2002, but the number 0002 would *not* display as 0002, but rather  $\_ \_ \_ 2$ . One feature of the 7447 that would be difficult to replicate in HDL is the combination input/output pin named  $\overline{BI}/\overline{RBO}$ . Rather than complicate the code, we have decided to create a separate input ( $\overline{BI}$ ) and an output ( $\overline{RBO}$ ) on two different pins. This discussion also makes no attempt to replicate the non-BCD display characters of a 7447 but simply blanks all segments for values greater than 9.

Several decisions must be made when designing a circuit such as this one. The first involves the type of display we intend to use. If it is a common cathode, then a logic 1 lights the LED segment. If it is a common anode, then a logic 0 is required to turn on a segment. Next, we must decide on the type of inputs, outputs, and intermediate variables. We have decided that the outputs for each individual segment should be assigned a bit name ( $a$ - $g$ ) rather than using a bit array. This arrangement will make it clearer when connecting the display to the IC. These individual bits can be grouped as a set of bits and assigned binary values, as we have done in AHDL, or an intermediate variable bit array can be used to make it convenient when assigning all seven bit levels in a single statement, as we have done in VHDL. The BCD inputs are treated as a four-bit number, and the blanking controls are individual bits. The other issue that greatly affects the bit patterns in the HDL code is the arbitrary decision of the order of the segment names  $a$ - $g$ . In this discussion, we have assigned segment  $a$  to the leftmost bit in the binary bit pattern, with the bits moving alphabetically left to right.

Some of the controls must have precedence over other controls. For example, the  $\overline{LT}$  (lamp test) should override any regular digit display, and the  $\overline{BI}$  (blanking input) should override even the lamp test input. In these illustrations, the IF/ELSE control structure is used to establish precedence. The first condition that is evaluated as true will determine the resulting output, regardless of the other input levels. Subsequent ELSE statements will have no effect, which is why the code tests first for  $\overline{BI}$ , then  $\overline{LT}$ , then  $\overline{RBI}$ , and finally determines the correct segment pattern.

## AHDL DECODER/DRIVER

The AHDL code for this circuit is shown in Figure 9-58. AHDL allows output bits to be grouped in a set by separating the bits with commas and enclosing them in parentheses. A group of binary states can be assigned directly to these bit sets, as shown on lines 9, 11, 13, and 15. This convention avoids the need for an intermediate variable and is much shorter than eight separate assignment statements. The TABLE feature of AHDL is useful in this application to correlate an input BCD value to a 7-segment bit pattern.

```

1  SUBDESIGN fig9_58
2  (
3    bcd[3..0]      :INPUT;      -- 4-bit number
4    lt, bi, rbi    :INPUT;      -- 3 independent controls
5    a,b,c,d,e,f,g,rbo :OUTPUT;  -- individual outputs
6  )
7  BEGIN
8    IF !bi THEN
9      (a,b,c,d,e,f,g,rbo) = (1,1,1,1,1,1,1,0); % blank all %
10   ELSIF !lt THEN
11     (a,b,c,d,e,f,g,rbo) = (0,0,0,0,0,0,0,1); % test segments %
12   ELSIF !rbi & bcd[] == 0 THEN
13     (a,b,c,d,e,f,g,rbo) = (1,1,1,1,1,1,1,0); % blank leading 0's %
14   ELSIF bcd[] > 9 THEN
15     (a,b,c,d,e,f,g,rbo) = (1,1,1,1,1,1,1,1); % blank non BCD input %
16   ELSE
17     TABLE % display 7 segment Common Anode pattern %
18     bcd[] => a,b,c,d,e,f,g,rbo;
19     0 => 0,0,0,0,0,0,1,1;
20     1 => 1,0,0,1,1,1,1,1;
21     2 => 0,0,1,0,0,1,0,1;
22     3 => 0,0,0,0,1,1,0,1;
23     4 => 1,0,0,1,1,0,0,1;
24     5 => 0,1,0,0,1,0,0,1;
25     6 => 1,1,0,0,0,0,0,1;
26     7 => 0,0,0,1,1,1,1,1;
27     8 => 0,0,0,0,0,0,0,1;
28     9 => 0,0,0,1,1,0,0,1;
29   END TABLE;
30   END IF;
31 END;
```

FIGURE 9-58 AHDL 7-segment BCD display decoder.

## VHDL DECODER/DRIVER

The VHDL code for this circuit is shown in Figure 9-59. This illustration demonstrates the use of a VARIABLE as opposed to a SIGNAL. A VARIABLE can be thought of as a piece of scrap paper used to write down some numbers that will be needed later. A SIGNAL, on the other hand, is usually thought of as a wire connecting two points in the circuit. In line 12, the keyword VARIABLE is used to declare *segments* as a bit vector with

```

1  ENTITY fig9_59 IS
2  PORT (
3      bcd                :IN INTEGER RANGE 0 TO 15;
4      lt, bi, rbi       :IN BIT;
5      a,b,c,d,e,f,g,rbo :OUT BIT
6  );
7  END fig9_59;
8
9  ARCHITECTURE vhd1 OF fig9_59 IS
10 BEGIN
11 PROCESS (bcd, lt, bi, rbi)
12 VARIABLE segments      :BIT_VECTOR (0 TO 6);
13 BEGIN
14     IF bi = '0' THEN
15         segments := "1111111";    rbo <= '0'; -- blank all
16     ELSIF lt = '0' THEN
17         segments := "0000000";    rbo <= '1'; -- test segments
18     ELSIF (rbi = '0' AND bcd = 0) THEN
19         segments := "1111111";    rbo <= '0'; -- blank leading 0's
20     ELSE
21         rbo <= '1';
22         CASE bcd IS          -- display 7 segment Common Anode pattern
23             WHEN 0          => segments := "0000001";
24             WHEN 1          => segments := "1001111";
25             WHEN 2          => segments := "0010010";
26             WHEN 3          => segments := "0000110";
27             WHEN 4          => segments := "1001100";
28             WHEN 5          => segments := "0100100";
29             WHEN 6          => segments := "1100000";
30             WHEN 7          => segments := "0001111";
31             WHEN 8          => segments := "0000000";
32             WHEN 9          => segments := "0001100";
33             WHEN OTHERS => segments := "1111111";
34         END CASE;
35     END IF;
36     a <= segments(0); --assign bits of array to output pins
37     b <= segments(1);
38     c <= segments(2);
39     d <= segments(3);
40     e <= segments(4);
41     f <= segments(5);
42     g <= segments(6);
43     END PROCESS;
44 END vhd1;

```

**FIGURE 9-59** VHDL 7-segment BCD display decoder.

seven bits. Take special note of the order of the indices for this variable. They are declared as 0 TO 6. In VHDL, this means that element 0 appears on the left end of the binary bit pattern and element 6 appears on the right end. This is exactly opposite of the way most examples in this text have presented variables, but it is important to realize the significance

of the declaration statement in VHDL. For this illustration, segment *a* is bit 0 (on the left), segment *b* is bit 1 (moving to the right), and so on.

Notice that on line 3, the BCD input is declared as an INTEGER. This allows us to refer to it by its numeric value in decimal rather than being limited to bit pattern references. A PROCESS is employed here in order to allow us to use the IF/ELSE constructs to establish the precedence of one input over the other. Notice that the sensitivity list contains all the inputs. The code within the PROCESS describes the behavioral operation of the circuit that is necessary whenever any of the inputs in the sensitivity list changes state. Another very important point in this illustration is the assignment operator for variables. Notice in line 15, for example, the statement *segments := "111111"*. The variable assignment operator := is used for variables in place of the <= operator that was used for signal assignments. In lines 36–42, the individual bits that were established in the IF/ELSE decisions are assigned to the proper output bits.

### OUTCOME ASSESSMENT QUESTIONS

1. What feature of a 7447 is very difficult to duplicate in PLD hardware and HDL code?
2. Are the HDL decoder driver examples intended to drive common-anode or common-cathode 7-segment displays?
3. How are certain inputs (e.g., lamp test) given precedence over other inputs (e.g., RBI) in the HDL code in this section?

## 9-17 ENCODERS USING HDL

### OUTCOMES

Upon completion of this section, you will be able to:

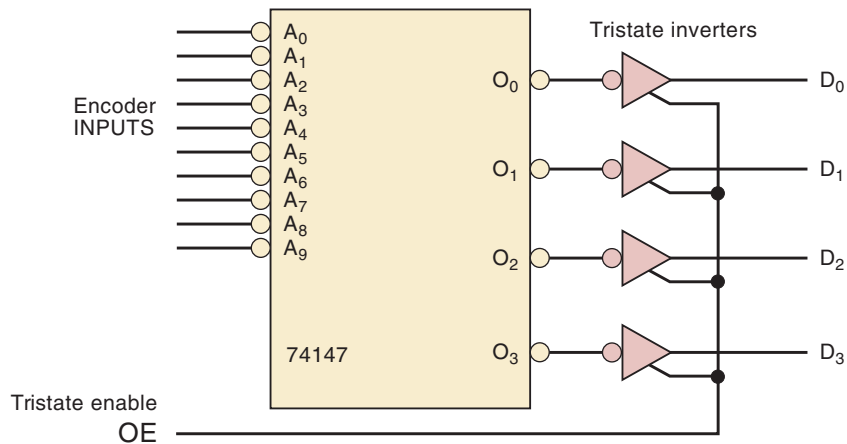
- Use HDL to describe any type of encoder.
- Arbitrate multiple input activations by priority in AHDL and VHDL.
- Describe tristate outputs in AHDL and VHDL.

In Section 9-4, we discussed encoders and priority encoders. Similarities exist, of course, between decoders and encoders. Decoders take a binary number and activate one output that corresponds to that number. An encoder works in the other direction by monitoring one of its several inputs; when one of the inputs is activated, it produces a binary number corresponding to that input. If more than one of its inputs is activated at the same time, a priority encoder ignores the input of lower significance and produces the binary value that corresponds to the most significant input. In other words, it gives more significant inputs priority over less significant inputs. This section focuses on the methods that can be used in HDL to describe circuits that have this characteristic of priority for some inputs over others.

Another very important concept, which was presented in Chapter 8, was the tristate output circuit. Devices with tristate outputs can produce a logic HIGH or a logic LOW, just like a normal circuit, when their output is enabled. However, these devices can have their outputs disabled, which puts them in a “disconnected” or a high-impedance state. This is very important for devices connected to common buses, as described in Section 9-12. The next logical question is, “How do we describe tristate outputs using

HDL?” This section incorporates tristate outputs in the encoder design to address this issue. In order to keep the discussion focused on the essentials, we create a circuit that emulates the 74147 priority encoder, with one added feature of having active-HIGH tristate outputs. Other features like cascading inputs and outputs (such as those found on a 74148) are left for you to try on your own. A symbol for the circuit we are describing is shown in Figure 9-60. Because the inputs are all labeled in a manner very similar to bit array notation, it makes sense to use a bit array to describe the encoder inputs. The tristate enable must be a single bit, and the encoded outputs can be described as an integer numeric value.

**FIGURE 9-60** Graphic description of an encoder with tristate outputs.



## AHDL ENCODER

The most important point to be made from Figure 9-61 is the method of establishing priority, but also note the I/O assignments. The AHDL input/output descriptions do not provide a separate type for integers but allow a bit array to be referred to as an integer. Consequently, line 4 describes the outputs as a bit array. In this illustration, a TABLE is used that is very similar to the tables often found in data books describing this circuit's operation. The key to this table is the use of the don't-care state (X) on inputs. The priority is described by the way we position these don't-care states in the truth table. Reading line 15, for instance, we see that as soon as we encounter an active input (LOW on input  $a[4]$ ), the lower order input bits do not matter. The output has been determined to be 4. The tristate outputs are made possible by using the built-in primitive function :TRI on line 6. This line assigns the attributes of a tristate buffer to the variable that has been named *buffer*. Recall that this is the same way a flip-flop is described in AHDL. The ports of a tristate buffer are quite straightforward. They represent the input (*in*), the output (*out*), and the tristate output enable (*oe*).

The next illustration (Figure 9-62) uses the IF/ELSE construct to establish priority, very much like the method demonstrated in the 7-segment decoder example. The first IF condition that evaluates TRUE will THEN cause the corresponding value to be applied to the tristate buffer inputs. The priority is established by the order in which we list the IF conditions. Notice that they start with input 9, the highest-order input. This illustration adds another feature of putting the outputs into the high-impedance state when no input is being activated. Line 20 shows that the output enables

**FIGURE 9-61** AHDL priority encoder with tristate outputs.

```

1  SUBDESIGN fig9_61
2  (
3      a[9..0], oe      :INPUT;
4      d[3..0]         :OUTPUT;
5  )
6  VARIABLE buffer[3..0] :TRI;
7  BEGIN
8      TABLE
9          a[]           => buffer[].in;
10         B"1111111111" => B"1111";    -- no input active
11         B"1111111110" => B"0000";    -- 0
12         B"111111110X" => B"0001";    -- 1
13         B"11111110XX" => B"0010";    -- 2
14         B"1111110XXX" => B"0011";    -- 3
15         B"111110XXXX" => B"0100";    -- 4
16         B"11110XXXXX" => B"0101";    -- 5
17         B"1110XXXXXX" => B"0110";    -- 6
18         B"110XXXXXXX" => B"0111";    -- 7
19         B"10XXXXXXXX"  => B"1000";    -- 8
20         B"0XXXXXXXXX"  => B"1001";    -- 9
21     END TABLE;
22     buffer[].oe = oe;    -- hook up enable line
23     d[] = buffer[].out; -- hook up outputs
24 END;
```

```

1  SUBDESIGN fig9_62
2  (
3      sw[9..0], oe    :INPUT;
4      d[3..0]         :OUTPUT;
5  )
6  VARIABLE
7      buffers[3..0]  :TRI;
8  BEGIN
9      IF      !sw[9]  THEN buffers[].in = 9;
10     ELSIF !sw[8]  THEN buffers[].in = 8;
11     ELSIF !sw[7]  THEN buffers[].in = 7;
12     ELSIF !sw[6]  THEN buffers[].in = 6;
13     ELSIF !sw[5]  THEN buffers[].in = 5;
14     ELSIF !sw[4]  THEN buffers[].in = 4;
15     ELSIF !sw[3]  THEN buffers[].in = 3;
16     ELSIF !sw[2]  THEN buffers[].in = 2;
17     ELSIF !sw[1]  THEN buffers[].in = 1;
18     ELSE          buffers[].in = 0;
19     END IF;
20     buffers[].oe = oe & sw[!b"1111111111"]; -- enable on any input
21     d[] = buffers[].out;                    -- connect to outputs
22 END;
```

**FIGURE 9-62** AHDL priority encoder using IF/ELSE.

will be activated only when the *oe* pin is activated and one of the inputs is activated. Another item of interest in this illustration is the use of bit array notation to describe individual inputs. For example, line 9 states that IF switch input 9 is activated (LOW), THEN the inputs to the tristate buffer will be assigned the value 9 (in binary, of course).

## VHDL ENCODER

Two very important VHDL techniques are demonstrated in this description of a priority encoder. The first is the use of tristate outputs in VHDL, and the second is a new method of describing priority. Figure 9-63 shows the input/output definitions for this encoder circuit. Notice on line 6 that the input switches are defined as bit vectors with indices from 9 to 0. Also note that the *d* output is defined as an IEEE standard bit array (*std\_logic\_vector* type). This definition is necessary to allow the use of high-impedance states (tristate) on the outputs and also explains the need for the LIBRARY and USE statements on lines 1 and 2. As we mentioned, a very important point of this illustration is the method of describing precedence for the inputs. This code uses the **conditional signal assignment statement** starting on line 14 and continuing through line 24. On line 14, it assigns the value listed to the right of `<=` to the variable *d* on the left, assuming the condition following WHEN is true. If this clause is not true, the clauses following ELSE are evaluated one at a time until one that is true is found. The value preceding WHEN will then be assigned to *d*. A very important attribute of the conditional signal assignment statement is the sequential evaluation. The precedence of these statements is established by the order in which they are

**FIGURE 9-63**  
VHDL priority encoder using conditional signal assignment.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY fig9_63 IS
5  PORT (
6      sw      :IN BIT_VECTOR (9 DOWNT0 0); -- standard logic not needed
7      oe      :IN BIT;                    -- standard logic not needed
8      d       :OUT STD_LOGIC_VECTOR (3 DOWNT0 0) -- std logic for hi-Z
9  );
10 END fig9_63;
11
12 ARCHITECTURE a OF fig9_63 IS
13 BEGIN
14     d <= "ZZZZ" WHEN ((oe = '0') OR (sw = "1111111111")) ELSE
15         "1001" WHEN sw(9) = '0' ELSE
16         "1000" WHEN sw(8) = '0' ELSE
17         "0111" WHEN sw(7) = '0' ELSE
18         "0110" WHEN sw(6) = '0' ELSE
19         "0101" WHEN sw(5) = '0' ELSE
20         "0100" WHEN sw(4) = '0' ELSE
21         "0011" WHEN sw(3) = '0' ELSE
22         "0010" WHEN sw(2) = '0' ELSE
23         "0001" WHEN sw(1) = '0' ELSE
24         "0000" WHEN sw(0) = '0';
25 END a;
```



listed. Notice that in this illustration, the first condition being tested (line 14) is the enabling of the tristate outputs. Recall from Chapter 8 that the three states of a tristate output are HIGH, LOW, and high impedance, which is referred to as Hi-Z. When the value “ZZZZ” is assigned to the output, each output is in the high-impedance state. If the outputs are to be disabled (Hi-Z), then none of the other encoding matters. Line 15 tests the highest priority input, which is bit 9 of the *sw* input array. If it is active (LOW), then a value of 9 is output regardless of whether other inputs are being activated at the same time.

### OUTCOME ASSESSMENT QUESTIONS

1. Name two methods in AHDL for giving priority to some inputs over others.
2. Name two methods in VHDL for giving priority to some inputs over others.
3. In AHDL, how are tristate outputs implemented?
4. In VHDL, how are tristate outputs implemented?

## 9-18 HDL MULTIPLEXERS AND DEMULTIPLEXERS

### OUTCOME

Upon completion of this section, you will be able to:

- Use HDL to describe any type of multiplexer or demultiplexer.

A multiplexer is a device that acts like a selector switch for digital signals. The select inputs are used to specify the input channel that is to be “connected” to the output pins. A demultiplexer works in the opposite direction by taking a digital signal as an input and distributing it to one of its outputs. Figure 9-64 shows a multiplexer/demultiplexer system with four data input channels. Each input is a four-bit number. These devices are not exactly like any of the multiplexers or demultiplexers described earlier in this chapter, but they operate in the same way. The system in this illustration allows the

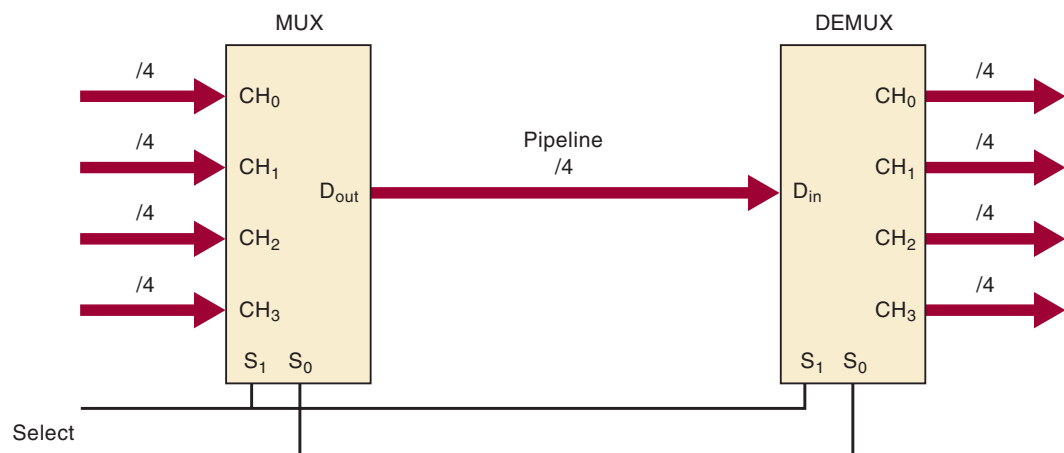


FIGURE 9-64 Four channels of data sharing a common data path.

four digital signals to share a common “pipeline” in order to get data from one point to the other. The select inputs are used to decide which signal is going through the pipeline at any time.

In this section, we examine some code that implements both the multiplexer and the demultiplexer. The key HDL issue in both the MUX and DEMUX is assigning signals under certain conditions. For the DEMUX, another issue is assigning a state to whichever outputs are not currently selected to distribute data. In other words, when an output is not being used for data (not selected), do we want it to have all bits HIGH, all bits LOW, or the tristate disabled? In the following descriptions, we have chosen to make them all HIGH when not selected, but with the structure shown, it would be a simple matter to change to one of the other possibilities.

### AHDL MUX AND DEMUX

We will implement the multiplexer first. Figure 9-65 describes a multiplexer with four inputs of four bits each. Each input channel is named in a way that identifies its channel number. In this figure, each input is described as a four-bit array. The select input ( $s[ ]$ ) requires two bits to specify the four channel numbers (0–3). A CASE construct is used here to assign an input channel conditionally to the output pins. Line 9, for example, states that in the case when the select inputs ( $s[ ]$ ) are set to 0 (i.e., binary 00), the circuit should connect the channel 0 input to the data output. Notice that when assigning connections, the destination (output) of the signal is on the left of the = sign and the source (input) is on the right.

The demultiplexer code works in a similar way but has only one input channel and four output channels. It must also ensure that the outputs are all HIGH when they are not selected. In Figure 9-66, the inputs and outputs are declared as usual on lines 3–5. The default condition for each channel is specified after the keyword DEFAULTS, which tells the compiler to generate a circuit that will have HIGHS on the outputs unless specifically assigned a value elsewhere in the code. If this default section is not specified, the output values would default automatically to all LOW. Notice on lines 16–19 that the input signal is assigned conditionally to one of the output channels. Consequently, the output channel is on the left of the = sign and the input signal is on the right.

**FIGURE 9-65**  
Four-bit × four-channel MUX in AHDL.

```

1  SUBDESIGN fig9_65
2  (
3      ch0[3..0], ch1[3..0], ch2[3..0], ch3[3..0]:INPUT;
4      s[1..0]                                :INPUT; -- select inputs
5      dout[3..0]                             :OUTPUT;
6  )
7  BEGIN
8      CASE s[] IS
9          WHEN 0 =>      dout[] = ch0[];
10         WHEN 1 =>      dout[] = ch1[];
11         WHEN 2 =>      dout[] = ch2[];
12         WHEN 3 =>      dout[] = ch3[];
13     END CASE;
14 END;
```

**FIGURE 9-66**

Four-bit  $\times$  four-channel  
DEMUX in AHDL.

```

1  SUBDESIGN fig9_66
2  (
3    ch0[3..0], ch1[3..0], ch2[3..0], ch3[3..0] :OUTPUT;
4    s[1..0] :INPUT;
5    din[3..0] :INPUT;
6  )
7  BEGIN
8    DEFAULTS
9      ch0[] = B"1111";
10     ch1[] = B"1111";
11     ch2[] = B"1111";
12     ch3[] = B"1111";
13   END DEFAULTS;
14
15   CASE s[] IS
16     WHEN 0 => ch0[] = din[];
17     WHEN 1 => ch1[] = din[];
18     WHEN 2 => ch2[] = din[];
19     WHEN 3 => ch3[] = din[];
20   END CASE;
21   END;
```

## VHDL MUX AND DEMUX

Figure 9-67 shows the code that creates a four-channel MUX with four bits per channel. The inputs are declared as bit arrays on line 3. They could have been declared just as easily as integers ranging from 0 to 15. Whichever way the inputs are declared, the outputs must be of the same type. Notice on line 4 that the select input (*s*) is declared as a decimal integer from 0 to 3 (equivalent to binary 00 to 11). This allows us to refer to it by its decimal channel number in the code, making it easier to understand. Lines 11–15 use the selected signal assignment statement to “connect” the appropriate input to the output, depending on the value on the select inputs. For example, line 15 states that channel 3 should be selected to connect to the data outputs when the select inputs are set to 3.

The demultiplexer code works in a similar way but has only one input channel and four output channels. In Figure 9-68, the inputs and outputs are declared as usual on lines 3–5. Notice that in line 3, the select input(*s*) is typed as an integer, just like the MUX code in Figure 9-67. The operation of a DEMUX is described most easily using several conditional signal assignment statements, as shown in lines 11–14. We decided earlier that the code for this DEMUX must ensure that the outputs are all HIGH when they are not selected. This is accomplished with the ELSE clause of each conditional signal assignment. If the ELSE clause is not used, the output values would default automatically to all LOW. For example, line 13 states that channel 2 will be connected to the data inputs whenever the select inputs are set to 2. If *s* is set to any other value, then channel 2 will be forced to have all bits HIGH.

**FIGURE 9-67**  
Four-bit  $\times$  four-channel  
MUX in VHDL.

```

1  ENTITY fig9_67 IS
2  PORT    (
3          ch0, ch1, ch2, ch3  :IN BIT_VECTOR (3 DOWNT0 0);
4          s                    :IN INTEGER RANGE 0 TO 3;
5          dout                 :OUT BIT_VECTOR (3 DOWNT0 0)
6          );
7  END fig9_67;
8
9  ARCHITECTURE selector OF fig9_67 IS
10 BEGIN
11     WITH s SELECT
12     dout  <= ch0 WHEN 0, -- switch channel 0 to output
13           ch1 WHEN 1, -- switch channel 1 to output
14           ch2 WHEN 2, -- switch channel 2 to output
15           ch3 WHEN 3; -- switch channel 3 to output
16 END selector;
```

**FIGURE 9-68**  
Four-bit  $\times$  four-channel  
DEMUX in VHDL.

```

1  ENTITY fig9_68 IS
2  PORT    (
3          s                    :IN INTEGER RANGE 0 TO 3;
4          din                 :IN BIT_VECTOR (3 DOWNT0 0);
5          ch0, ch1, ch2, ch3  :OUT BIT_VECTOR(3 DOWNT0 0)
6          );
7  END fig9_68;
8
9  ARCHITECTURE selector OF fig9_68 IS
10 BEGIN
11     ch0 <= din WHEN s = 0 ELSE "1111";
12     ch1 <= din WHEN s = 1 ELSE "1111";
13     ch2 <= din WHEN s = 2 ELSE "1111";
14     ch3 <= din WHEN s = 3 ELSE "1111";
15 END selector;
```

#### OUTCOME ASSESSMENT QUESTIONS

1. For the four-bit by four-channel MUX, name the data inputs, the data outputs, and the control inputs that choose one channel of the four.
2. For the four-bit by four-channel DEMUX, name the data inputs, the data outputs, and the control inputs that choose one channel of the four.
3. In the AHDL example, how are the logic states determined for the channels that are not selected?
4. In the VHDL example, how are the logic states determined for the channels that are not selected?

## 9-19 HDL MAGNITUDE COMPARATORS

### OUTCOME

Upon completion of this section, you will be able to:

- Use HDL to describe any magnitude comparator.

In Section 9-10, we studied a 7485 magnitude comparator chip. As the name implies, this device compares the magnitude of two binary numbers and indicates the relationship between the two (greater than, less than, equal to). Control inputs are provided for the purpose of cascading these chips. These chips are interconnected so that the chip comparing the lower-order bits has its outputs connected to the control inputs of the next higher-order chip, as shown in Figure 9-40. When the highest-order stage detects that its data inputs have equal magnitude, it will look to the next lower stage and use these control inputs to make the final decision. This gives us a chance to look at one of the defining differences between using traditional logic ICs and using HDL to design a circuit. If we need to compare bigger values using HDL we can simply adjust the size of the comparator input ports to be whatever we need, rather than trying to cascade several four-bit comparators. Consequently, there is no need for cascading input controls in the HDL version.

There are many possible ways to describe the operation of a comparator. However, it is best to use an IF/ELSE construct because each IF clause evaluates a relationship between two values, as opposed to looking for the single value of a variable, like a CASE. The two inputs being compared should definitely be declared as numerical values. The three comparator outputs should be declared as individual bits in order to label each bit's purpose clearly.

### AHDL COMPARATOR

The AHDL code in Figure 9-69 follows the algorithm we have described using IF/ELSE constructs. Notice in line 3 that the data values are declared as four-bit numbers. Also note in lines 8, 10, and 11 that several statements can be used to specify the circuit's operation when the IF clause is true. Each statement is used to set the level on one of the outputs. These three statements are considered concurrent, and the order in which they are listed makes no difference. For example, in line 8, when *A* is greater than *B*, the *agtb* output will go HIGH at the same time the other two outputs (*altb*, *aeqb*) go LOW.

```

1  SUBDESIGN fig9_69
2  (
3      a[3..0], b[3..0]   :INPUT;
4      agtb, altb, aeqb  :OUTPUT;
5  )
6  BEGIN
7      IF    a[] > b[] THEN
8          agtb = VCC;  altb = GND;  aeqb = GND;
9      ELSIF a[] < b[] THEN
10         agtb = GND;  altb = VCC;  aeqb = GND;
11      ELSE    agtb = GND;  altb = GND;  aeqb = VCC;
12      END IF;
13  END;
```

FIGURE 9-69 Magnitude comparator in AHDL.

## VHDL COMPARATOR

The VHDL code in Figure 9-70 follows the algorithm we have described using IF/ELSE constructs. Notice in line 2 that the data values are declared as four-bit integers. Remember, in VHDL, the IF/ELSE constructs can be used only inside a PROCESS. In this case, we want to evaluate the PROCESS whenever any of the inputs change state. Consequently, each input is listed in the sensitivity list within the parentheses. Also note in lines 10, 11, and 12 that several statements can be used to specify the circuit's operation when the IF clause is true. Each statement is used to set the level on one of the outputs. These three statements are considered concurrent, and the order in which they are listed makes no difference. For example, on line 11, when *A* is greater than *B*, the *agtb* output will go HIGH at the same time the other two outputs (*altb*, *aeqb*) go LOW.

```

1  ENTITY fig9_70 IS
2  PORT ( a, b           : IN INTEGER RANGE 0 TO 15;
3         agtb, altb, aeqb : OUT BIT);
4  END fig9_70
5
6  ARCHITECTURE vhdl OF fig9_70 IS
7  BEGIN
8      PROCESS (a, b)
9      BEGIN
10         IF    a < b THEN      altb <= '1';  agtb <= '0';  aeqb <= '0';
11         ELSIF a > b THEN      altb <= '0';  agtb <= '1';  aeqb <= '0';
12         ELSE                  altb <= '0';  agtb <= '0';  aeqb <= '1';
13         END IF;
14     END PROCESS;
15 END vhdl;

```

**FIGURE 9-70** Magnitude comparator in VHDL.

### OUTCOME ASSESSMENT QUESTIONS

1. What type of data objects must be declared for data inputs to a comparator?
2. What is the key control structure used to describe a comparator?
3. What are the key operators used?

## 9-20 HDL CODE CONVERTERS

### OUTCOME

*Upon completion of this section, you will be able to:*

- Use HDL to describe any code converter.

Section 9-11 demonstrated some methods using adder circuits in an interesting but not at all intuitive way to create a BCD-to-binary conversion. In Chapter 6, we discussed adder circuits, and the circuit of Figure 9-43 can certainly be implemented using HDL and 7483 macrofunctions or adder descriptions that we know how to write. However, this is an excellent opportunity

to point out the tremendous advantage that HDL can offer because it allows a circuit to be described in a way that makes the most sense. In the case of BCD-to-binary conversion, the sensible method of conversion is to use the concepts that we all learned in the second grade about the decimal number system. You were once taught that the number 275 was actually:

$$\begin{array}{r} 2 \times 100 = 200 \\ + 7 \times 10 = 70 \\ + 5 \times 1 = 5 \\ \hline 275 \end{array}$$

Now we have studied the BCD number system and realize that 275 is represented in BCD as 0010 0111 0101. Each digit is simply represented in binary. If we could multiply these four-bit binary-coded decimal digits by the decimal weight (represented in binary) and add them, we would have a binary answer that is equivalent to the BCD quantity. For example, let's try using the BCD representation for 275:

<i>BCD</i>	<i>Decimal Weight (in binary)</i>	<i>Partial Product (in binary)</i>
0010 ×	1100100 =	11001000
+ 0111 ×	1010 =	01000110
+ 0101 ×	1 =	0101
		100010011 = 275 <sub>10</sub>

We will illustrate the conversion of a two-digit decimal number represented by an eight-bit BCD code to its equivalent binary value using HDL. The solution will use the following strategy:

**Take the most significant BCD digit (the tens place) and multiply it by 10. Add this product to the least significant BCD digit (the ones place).**

The answer will be a binary number representing the BCD quantity. It is important to realize that the HDL compiler does not necessarily try to implement an actual multiplier circuit in its solution. It will create the most efficient circuit that will do the job, which allows the designer to describe its behavior in the most sensible way.

### AHDL BCD-TO-BINARY CODE CONVERTER

The key to this strategy is in being able to multiply by 10. AHDL does not offer a multiplication operator, so in order to use this overall strategy, we need some math tricks. We will use the shifting of bits to perform multiplication and then employ the distributive property from algebra to multiply by 10. In the same way that we can shift a decimal number left by one digit, thus multiplying it by 10, we can likewise shift a binary number one place to the left and multiply it by 2. Shifting two places multiplies a binary number by 4, and shifting three places multiplies by 8. The distributive property tells us that:

$$\text{num} \times 10 = \text{num} \times (8 + 2) = (\text{num} \times 8) + (\text{num} \times 2)$$

If we can take the BCD tens digit and shift it left three bit positions (i.e., multiply it by 8), then take the same number and shift it left one place (i.e., multiply it by 2), and then add them together, the result will be the same as multiplying the BCD digit by 10. This value is then added to the BCD ones digit to produce the binary equivalent of the two-digit BCD input.

The next challenge is to shift the BCD digit left using AHDL. Because AHDL allows us to make up sets of variables, we can shift the bits by appending zeros to the right end of the array. For example, if we have the number 5 in BCD (0101) and we want to shift it three places, we can concatenate the number 0101 with the number 000 in a set, as follows:

$$(B"0101", B"000") = B"0101000"$$

The AHDL code in Figure 9-71 begins by declaring inputs for the BCD ones and tens digits. The binary output must be able to represent  $99_{10}$ , which requires seven bits. We also need a variable to hold the product of the BCD digit multiplied by 10. Line 5 declares this variable as a seven-bit number. Line 8 performs the shifting of the *tens[]* array three times and adds it to the *tens[]* array shifted one place to the left. Notice that this latter set must have seven bits in order to be added to the first set, thus the need to concatenate B"00" on the left end. Finally, in line 10, the result from line 8 is added to the BCD ones digit with leading zero extensions (to make seven bits) to form the binary output.

```

1  SUBDESIGN fig9_71
2  ( ones[3..0], tens[3..0]      :INPUT;
3    binary[6..0]                :OUTPUT; )
4
5  VARIABLE times10[6..0]        :NODE;    % variable for tens digit times 10%
6
7  BEGIN
8    times10[] = (tens[],B"000") + (B"00",tens[],B"0");
9    % shift left 3X (times 8) + shift left 1X (times 2) %
10   binary[] = times10[] + (B"000",ones[]);
11   % tens digit times 10 + ones digit %
12  END;
```

**FIGURE 9-71** BCD-to-binary code converter in AHDL.

## VHDL BCD-TO-BINARY CODE CONVERTER

The VHDL solution in Figure 9-72 is very simple due to the powerful math operations available in the language. The inputs and outputs must be declared as integers because we intend to perform arithmetic operations on them. Notice that the range is specified based on the largest valid BCD number using two digits. In line 9, the tens digit is multiplied by ten, and in line 10, the ones digit is added to form the binary equivalent of the BCD input.



**FIGURE 9-72** BCD-to-binary code converter in VHDL.

```

1  ENTITY fig9_72 IS
2  PORT (  ones, tens  :IN INTEGER RANGE 0 TO 9;
3         binary      :OUT INTEGER RANGE 0 TO 99);
4  END fig9_72;
5
6  ARCHITECTURE vhdl OF fig9_72 IS
7  SIGNAL times10      :INTEGER RANGE 0 TO 90;
8  BEGIN
9      times10 <= tens * 10;
10     binary <= times10 + ones;
11 END vhdl;

```

### OUTCOME ASSESSMENT QUESTIONS

1. For a two-digit BCD (eight-bit) number, what is the decimal weight of the most significant digit?
2. In AHDL, how is multiplication by 10 accomplished?
3. In VHDL, how is multiplication by 10 accomplished?

## SUMMARY

1. A decoder is a device whose output is activated only when a unique binary combination (code) is presented on its inputs. Many MSI decoders have several outputs, each one corresponding to only one of the many possible input combinations.
2. Digital systems often need to display decimal numbers. This is done using 7-segment displays that are driven by special chips that decode the binary number and translate it into segment patterns that represent decimal numbers to people. The segment elements can be light-emitting diodes, liquid crystals, or glowing electrodes surrounded by neon gas.
3. Graphical LCDs use a matrix of picture elements called pixels to create an image on a large screen. Each pixel is controlled by activating the row and column that have that pixel in common. The brightness level of each pixel is stored as a binary number in the video memory. A fairly complex digital circuit must scan through the video memory and all the row/column combinations, controlling the amount of light that can pass through each pixel.
4. An encoder is a device that generates a unique binary code in response to the activation of each individual input.
5. Troubleshooting a digital system involves *observation/analysis* to identify the possible causes, and a process of elimination called *divide-and-conquer* to isolate and identify the cause.
6. Multiplexers act like digitally controlled switches that select and connect one logic input at a time to the output pin. By taking turns, many different data signals can share the same data path using multiplexers. Demultiplexers are used at the other end of the data path to separate

the signals that are sharing a data path and distribute them to their respective destinations.

7. Magnitude comparators serve as an indicator of the relationship between two binary numbers, with outputs that show  $>$ ,  $<$ , and  $=$ .
8. It is often necessary to translate between and among various methods of representing quantities with binary numbers. Code converters are devices that take in one form of binary representation and convert it to another form.
9. In digital systems, many devices must often share the same data path. This data path is often called a *data bus*. Even though many devices can be “riding” on the bus, there can be only one bus “driver” at any one time. Thus, devices must take turns applying logic signals to the data bus.
10. In order to take turns, the devices must have *tristate outputs* that can be disabled when another device is driving the bus. In the disabled state, the device’s output is essentially electrically disconnected from the bus by going into a state that offers a high-impedance path to both ground and the positive power supply. Devices designed to interface to a bus have outputs that can be HIGH, LOW, or disabled (high impedance).
11. PLDs offer an alternative to the use of MSI circuits to implement digital systems. Boolean equations can be used to describe the operation of these circuits, but HDLs also offer high-level language constructs.
12. HDL macrofunctions are available for many MSI standard parts described in this chapter.
13. Custom code can be written in HDL to describe each of the common logic functions presented in this chapter.
14. Priority and precedence can be established in AHDL using don’t-care entries in truth tables and using IF/ELSE decisions. Priority and precedence can be established in VHDL using conditional signal assignments or using a PROCESS containing IF/ELSE or CASE decisions.
15. Tristate outputs can be created in HDL. AHDL uses :TRI primitives that drive the outputs. VHDL assigns Z (high impedance) as a valid state for STD\_LOGIC outputs.
16. The DEFAULTS statement in AHDL can be used to define the proper level for outputs that are not explicitly defined in the code.
17. The ELSE clause in the conditional signal assignment statement of VHDL can be used to define the default state of an output.

## IMPORTANT TERMS

---

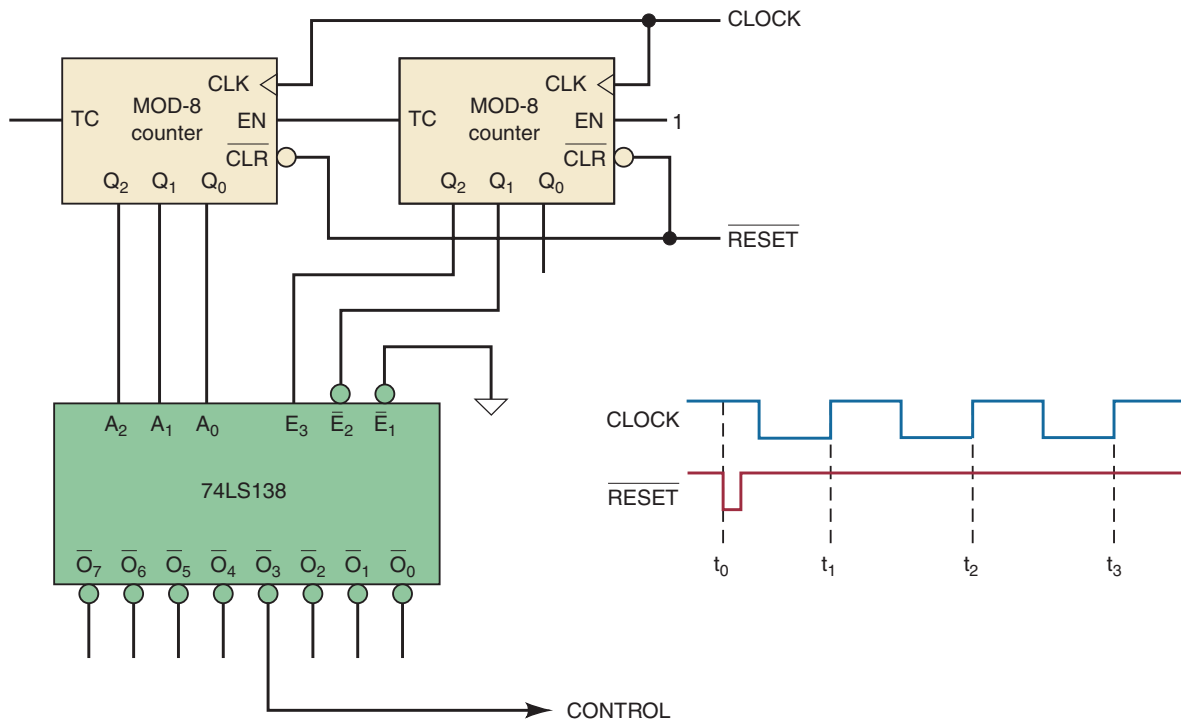
decoder	encoder	magnitude comparator
BCD-to-decimal decoder	priority encoder	data bus
driver	observation/analysis	floating bus
BCD-to-7-segment	divide-and-conquer	word
decoder/driver	multiplexer (MUX)	bus driver
common anode	multiplexing	bidirectional data lines
common cathode	parallel-to-serial	DEFAULTS
LCD	conversion	conditional signal assign-
backplane	demultiplexer (DEMUX)	ment statement
pixel	time division	
encoding	multiplexing (TDM)	

---

## PROBLEMS

### SECTION 9-1

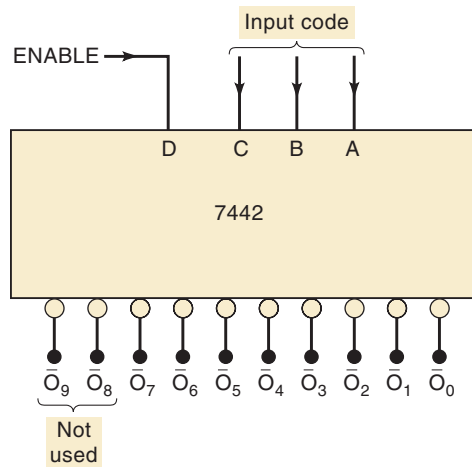
- B** 9-1. Refer to Figure 9-3. Determine the levels at each decoder output for the following sets of input conditions.
- All inputs LOW
  - All inputs LOW except  $E_3 = \text{HIGH}$
  - All inputs HIGH except  $\bar{E}_1 = \bar{E}_2 = \text{LOW}$
  - All inputs HIGH
- B** 9-2.\* What is the number of inputs and outputs of a decoder that accepts 64 different input combinations?
- B** 9-3. For a 74ALS138, what input conditions will produce the following outputs:
- LOW at  $\bar{O}_6$
  - LOW at  $\bar{O}_3$
  - LOW at  $\bar{O}_5$
  - LOW at  $\bar{O}_0$  and  $\bar{O}_7$ , simultaneously
- D** 9-4. Show how to use 74LS138s to form a 1-of-16 decoder.
- 9-5.\* Figure 9-73 shows how a decoder can be used in the generation of control signals. Assume that a RESET pulse has occurred at time  $t_0$ , and determine the CONTROL waveform for 32 clock pulses.



**FIGURE 9-73** Problems 9-5 and 9-6.

\* Answers to problems marked with an asterisk can be found in the back of the text.

- D** 9-6. Modify the circuit of Figure 9-73 to generate a CONTROL waveform that goes LOW from  $t_{20}$  to  $t_{24}$ . (*Hint:* The modification does not require additional logic.)
- 9-7. (a)\* The 7442 decoder of Figure 9-5 does not have an ENABLE input. However, we can operate it as a 3-line-to-8-line decoder by not using outputs  $\bar{O}_8$  and  $\bar{O}_9$  and by using the  $D$  input as an ENABLE. This is illustrated in Figure 9-74. Describe how this arrangement works as an enabled 1-of-8 decoder, and state how the level on  $D$  either enables or disables the outputs.
- (b) Use a megafunction to create a decimal decoder with enable.

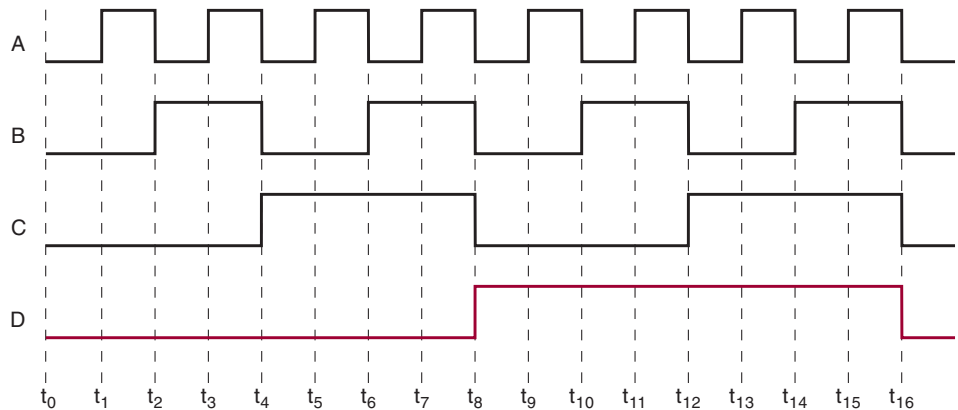


**FIGURE 9-74** Problem 9-7.

- 9-8. Consider the waveforms in Figure 9-75. Apply these signals to the 74LS138 as follows:

$$A \rightarrow A_0 \quad B \rightarrow A_1 \quad C \rightarrow A_2 \quad D \rightarrow E_3$$

Assume that  $\bar{E}_1$  and  $\bar{E}_2$  are tied LOW, and draw the waveforms for outputs  $\bar{O}_0$ ,  $\bar{O}_3$ ,  $\bar{O}_6$ , and  $\bar{O}_7$ .



**FIGURE 9-75** Problems 9-8, 9-15, and 9-41.

- D** 9-9. Modify the circuit of Figure 9-6 so that relay  $K_1$  stays energized from PGT 3 to 5 and  $K_2$  stays energized from PGT 6 to 9. (*Hint:* This modification requires no additional circuitry.)

## SECTIONS 9-2 AND 9-3

- B, D** 9-10\* Show how to connect BCD-to-7-segment decoder/drivers and LED 7-segment displays to the counter circuit of Figure 7-22. Assume that each segment is to operate at approximately 10 mA at 2.5 V.
- B** 9-11. (a) Refer to Figure 9-10 and draw the segment and backplane waveforms relative to ground for CONTROL = 0. Then draw the waveform of segment voltage relative to backplane voltage.  
(b) Repeat part (a) for CONTROL = 1.
- C, D** 9-12\* The BCD-to-7-segment decoder/driver of Figure 9-8 contains the logic for activating each segment for the appropriate BCD inputs. Design the logic for activating the *g* segment.

## SECTION 9-4

**B** 9-13\* DRILL QUESTION

For each item, indicate whether it is referring to a decoder or an en-coder.

- (a) Has more inputs than outputs.  
(b) Is used to convert key actuations to a binary code.  
(c) Only one output can be activated at one time.  
(d) Can be used to interface a BCD input to an LED display.  
(e) Often has driver-type outputs to handle large *I* and *V*.

9-14. Determine the output levels for the 74147 encoder when  $\bar{A}_8 = \bar{A}_4 = 0$  and all other inputs are HIGH.

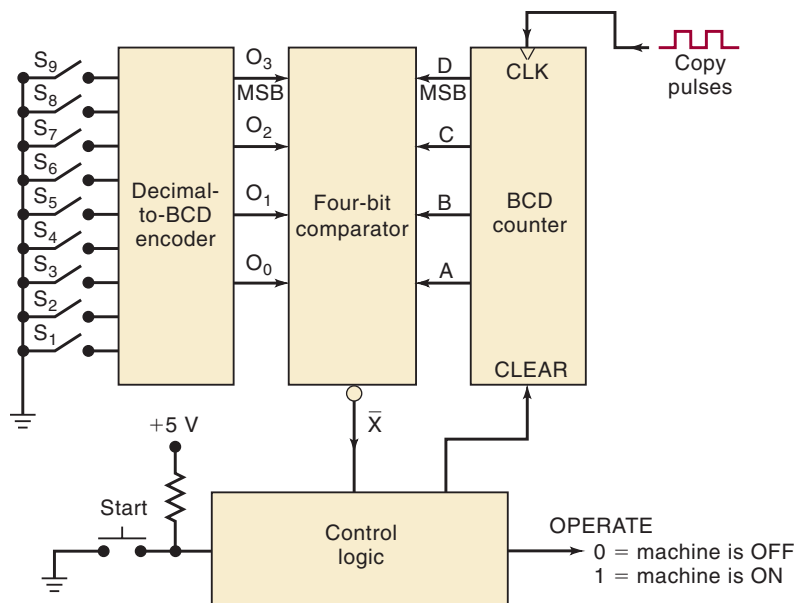
9-15. Apply the signals of Figure 9-75 to the inputs of a 74147 as follows:

$$A \rightarrow \bar{A}_7 \quad B \rightarrow \bar{A}_4 \quad C \rightarrow \bar{A}_2 \quad D \rightarrow \bar{A}_1$$

Draw the waveforms for the encoder's outputs.

- C, D** 9-16. Figure 9-76 shows the block diagram of a logic circuit used to control the number of copies made by a copy machine. The machine operator selects the number of desired copies by closing one of the selector switches  $S_1$  to  $S_9$ . This number is encoded in BCD by the encoder and is sent to a comparator circuit. The operator then hits a momentary-contact START switch, which clears the counter and initiates a HIGH OPERATE output that is sent to the machine to signal it to make copies. As the machine makes each copy, a copy pulse is generated and fed to the BCD counter. The counter outputs are continually compared with the switch encoder outputs by the comparator. When the two BCD numbers match, indicating that the desired number of copies has been made, the comparator output  $\bar{X}$  goes LOW; this causes the OPERATE level to return LOW and stop the machine so that no more copies are made. Activating the START switch will cause this process to be repeated. Design the complete logic circuitry for the comparator and control sections of this system.
- C, D** 9-17\* The keyboard circuit of Figure 9-16 is designed to accept a three-digit decimal number. What would happen if *four* digit keys were activated (e.g., 3095)? Design the necessary logic to be added to this circuit so that after three digits have been entered, any additional digits will be ignored until the CLEAR key is depressed. In other words, if 3095 is entered on the keyboard, the output registers will

**FIGURE 9-76** Problems 9-16 and 9-52.



display 309 and will ignore the 5 and any subsequent digits until the circuit is cleared.

**SECTION 9-5**

- T** 9-18\* A technician breadboards the keyboard entry circuit of Figure 9-16 and tests its operation by trying to enter a series of three-digit numbers. He finds that sometimes the digit 0 is entered instead of the digit he pressed. He also observes that it happens with all of the keys more or less randomly, although it is worse for some keys than others. He replaces all of the ICs, and the malfunction persists. Which of the following circuit faults would explain his observations? Explain each choice.
  - (a) The technician forgot to ground the unused inputs of the OR gate.
  - (b) He has mistakenly used  $\bar{Q}$  instead of  $Q$  from the one-shot.
  - (c) The switch bounce from the digit keys lasts longer than 20 ms.
  - (d) The Y and Z outputs are shorted together.
- T** 9-19. Repeat Problem 9-18 with the following symptom: the registers and displays stay at 0 no matter how many times a key is pressed.
- T** 9-20\* While testing the circuit of Figure 9-16, a technician finds that every odd-numbered key results in the correct digit being entered, but every even-numbered key results in the wrong digit being entered as follows: the 0 key causes a 1 to be entered, the 2 key causes a 3 to be entered, the 4 key causes a 5 to be entered, and so on. Consider each of the following faults as possible causes of the malfunction. For each one, explain why it can or cannot be the actual cause.
  - (a) There is an open connection from the output of the LSB inverter to the D inputs of the FFs.
  - (b) The D input of flip-flop  $Q_8$  is internally shorted to  $V_{CC}$ .
  - (c) A solder bridge is shorting  $\bar{O}_0$  to ground.

- T** 9-21\* A technician tests the circuit of Figure 9-4 as described in Example 9-7, and she obtains the following results: all of the outputs work except  $\bar{O}_{16}$  to  $\bar{O}_{19}$  and  $\bar{O}_{24}$  to  $\bar{O}_{27}$ , which are permanently HIGH. What is the most probable circuit fault?
- T** 9-22. A technician tests the circuit of Figure 9-4 as described in Example 9-7 and finds that the correct output is activated for each possible input code except those listed in Table 9-8. Examine this table and determine the probable cause of the malfunction.

TABLE 9-8

Input Code					Activated Outputs
$A_4$	$A_3$	$A_2$	$A_1$	$A_0$	
1	0	0	0	0	$\bar{O}_{16}$ and $\bar{O}_{24}$
1	0	0	0	1	$\bar{O}_{17}$ and $\bar{O}_{25}$
1	0	0	1	0	$\bar{O}_{18}$ and $\bar{O}_{26}$
1	0	0	1	1	$\bar{O}_{19}$ and $\bar{O}_{27}$
1	0	1	0	0	$\bar{O}_{20}$ and $\bar{O}_{28}$
1	0	1	0	1	$\bar{O}_{21}$ and $\bar{O}_{29}$
1	0	1	1	0	$\bar{O}_{22}$ and $\bar{O}_{30}$
1	0	1	1	1	$\bar{O}_{23}$ and $\bar{O}_{31}$

- T** 9-23\* Suppose that a  $22\text{-}\Omega$  resistor was mistakenly used for the  $g$  segment in Figure 9-8. How would this affect the display? What possible problems could occur?
- T** 9-24. Repeat Example 9-8 with the observed sequence shown below:

COUNT	0	1	2	3	4	5	6	7	8	9
Observed display										

- T** 9-25\* Repeat Example 9-8 with the observed sequence shown below:

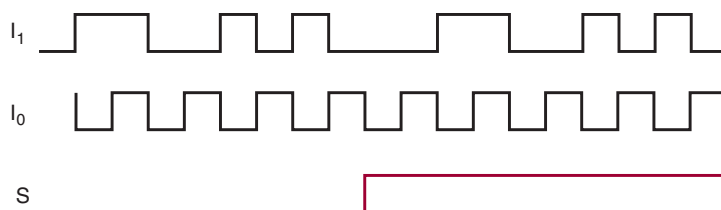
COUNT	0	1	2	3	4	5	6	7	8	9
Observed display										

- T** 9-26\* To test the circuit of Figure 9-10, a technician connects a BCD counter to the 74HC4511 inputs and pulses the counter at a very slow rate. She notices that the  $f$  segment works erratically, and no particular pattern is evident. What are some of the possible causes of the malfunction? (*Hint*: Remember, the ICs are CMOS.)

### SECTIONS 9-6 AND 9-7

- B** 9-27. The timing diagram in Figure 9-77 is applied to Figure 9-19. Draw the output waveform  $Z$ .

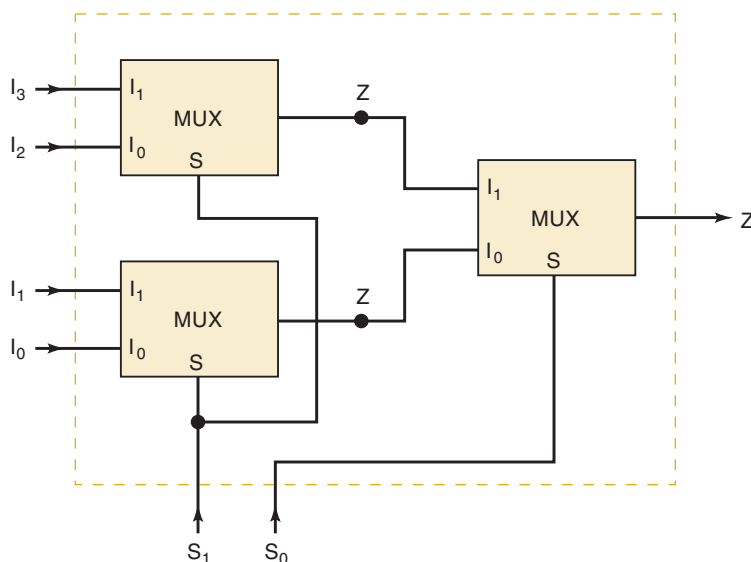
**FIGURE 9-77** Problem 9-27.



9-28. Figure 7-73 shows an eight-bit shift register that could be used to delay a signal by 1 to 8 clock periods. Show how to wire a 74151 to this shift register in order to select the desired  $Q$  output and indicate the logic level necessary on the select inputs to provide a delay of  $6 \times T_{\text{clk}}$ .

9-29\* The circuit in Figure 9-78 uses three two-input multiplexers (Figure 9-19). Determine the function performed by this circuit.

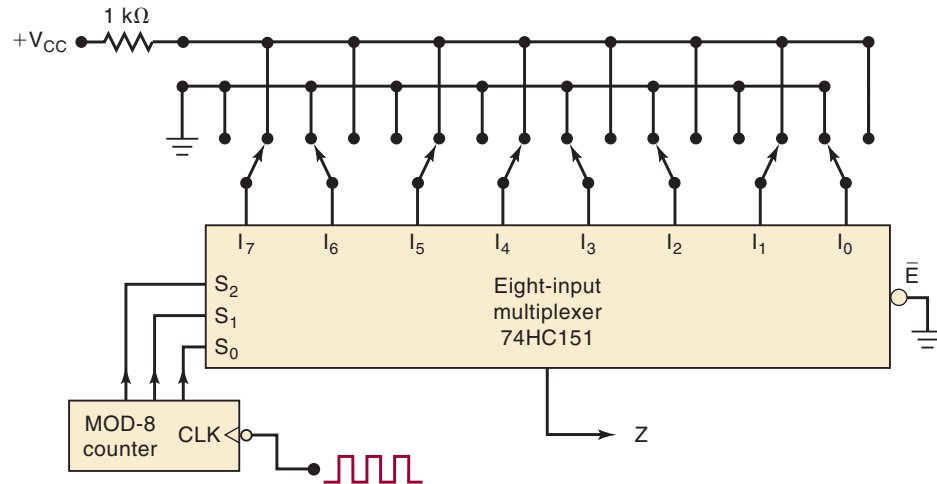
**FIGURE 9-78** Problem 9-29.



- D** 9-30. (a) Use the idea from Problem 9-29 to arrange several 74151 1-of-8 multiplexers to form a 1-of-64 multiplexer.
- (b) Use a Quartus II megafunction to create a 1-of-2 MUX, a 1-of-4 MUX, and a 1-of-8 MUX.
- C, D** 9-31. (a)\* Show how two 74157s and a 74151 can be arranged to form a 1-of-16 multiplexer with no other required logic. Label the inputs  $I_0$  to  $I_{15}$  to show how they correspond to the select code.
- (b) Create a 1-of-16 multiplexer using a megafunction.
- D** 9-32. (a) Expand the circuit of Figure 9-24 to display the contents of two three-stage BCD counters.
- (b)\* Count the number of connections in this circuit, and compare it with the number required if a separate decoder/driver and display were used for each stage of each counter.
- 9-33\* Figure 9-79 shows how a multiplexer can be used to generate logic waveforms with any desirable pattern. The pattern is programmed



**FIGURE 9-79** Problems 9-33 and 9-34.

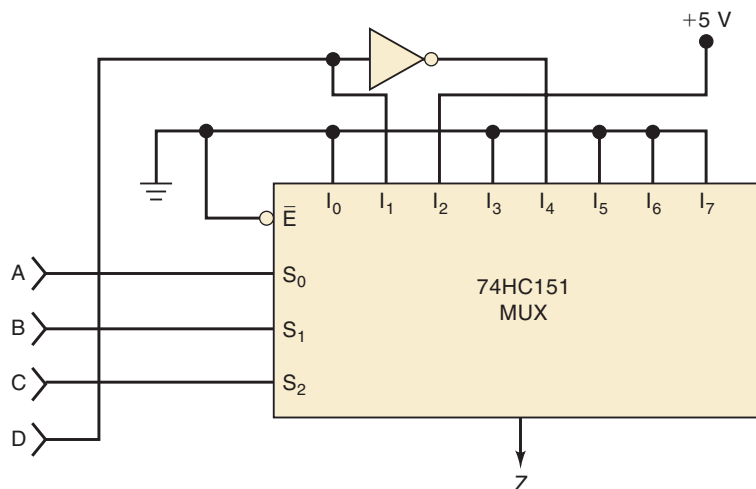


using eight SPDT switches, and the waveform is repetitively produced by pulsing the MOD-8 counter. Draw the waveform at Z for the given switch positions.

- 9-34. Change the MOD-8 counter in Figure 9-79 to a MOD-16 counter, and connect the MSB to the multiplexer  $\bar{E}$  input. Draw the Z waveform.
- D** 9-35\* Show how a 74151 can be used to generate the logic function  $Z = AB + BC + AC$ .
- D** 9-36. Show how a 16-input multiplexer such as the 74150 is used to generate the function  $Z = \bar{A}\bar{B}\bar{C}D + BCD + \bar{A}B\bar{D} + ABC\bar{D}$ .
- N** 9-37\* The circuit of Figure 9-80 shows how an eight-input MUX can be used to generate a four-variable logic function, even though the MUX has only three SELECT inputs. Three of the logic variables A, B, and C are connected to the SELECT inputs. The fourth variable D and its inverse  $\bar{D}$  are connected to selected data inputs of the MUX as required by the desired logic function. The other MUX data inputs are tied to a LOW or a HIGH as required by the function.
- (a) Set up a truth table showing the output Z for the 16 possible combinations of input variables.
- (b) Write the sum-of-products expression for Z and simplify it to verify that

$$Z = \bar{C}\bar{B}\bar{A} + D\bar{C}\bar{B}A + \bar{D}C\bar{B}\bar{A}$$

**FIGURE 9-80** Problems 9-37 and 9-38.



9-38. The hardware method used in Figure 9-80 can be used to generate any four-variable logic function. For example,  $Z = \overline{D} \overline{B} \overline{C} A + \overline{C} B A + D C \overline{B} A + C B \overline{A}$  is implemented by following these steps:

1. Set up a truth table in two halves, side by side as shown in Table 9-9. Notice that the left half shows all combinations of  $CBA$  when  $D = 0$ , and the right half shows all combinations of  $CBA$  when  $D = 1$ .
2. Write the value of  $Z$  for each four-bit combination when  $D = 0$  and also when  $D = 1$ .

TABLE 9-9

DCBA	D = 0		D = 1		$I_n$
	Z	DCBA	Z		
0000	0	1000	0		$I_0 = 0$
0001	1	1001	0		$I_1 = \overline{D}$
0010	0	1010	0		$I_2 = 0$
0011	1	1011	1		$I_3 = 1$
0100	0	1100	0		$I_4 = 0$
0101	0	1101	1		$I_5 = D$
0110	1	1110	1		$I_6 = 1$
0111	0	1111	0		$I_7 = 1$

3. Make a column on the right side as shown, which describes what must be connected to each of the eight MUX inputs  $I_n$ .
  4. For each line of this table, compare the value for  $Z$  when  $D = 0$  with the value for  $Z$  when  $D = 1$ . Enter the appropriate information for  $I_n$  as follows:
    - When  $Z = 0$  regardless of whether  $D = 0$  or  $1$ , THEN  $I_n = 0$  (GND).
    - When  $Z = 1$  regardless of whether  $D = 0$  or  $1$ , THEN  $I_n = 1$  (VCC).
    - When  $Z = 0$  when  $D = 0$  AND  $Z = 1$  when  $D = 1$ , THEN  $I_n = D$ .
    - When  $Z = 1$  when  $D = 0$  AND  $Z = 0$  when  $D = 1$ , THEN  $I_n = \overline{D}$ .
- (a) Verify the design of Figure 9-80 using this method.
- (b) Use this method to implement a function that will produce a HIGH only when the four input variables are at the same level or when the  $B$  and  $C$  variables are at different levels.

**SECTION 9-8**

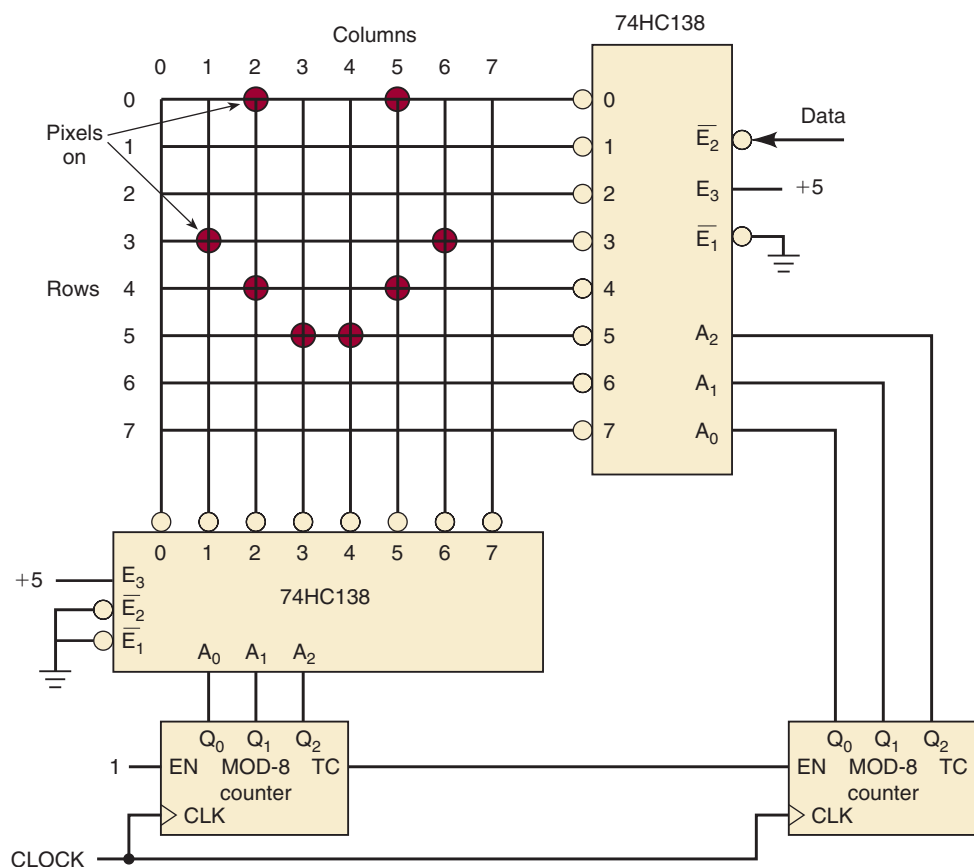
**B 9-39\* DRILL QUESTION**

For each item, indicate whether it is referring to a decoder, an encoder, a MUX, or a DEMUX.

- (a) Has more inputs than outputs.
- (b) Uses SELECT inputs.
- (c) Can be used in parallel-to-serial conversion.
- (d) Produces a binary code at its output.
- (e) Only one of its outputs can be active at one time.
- (f) Can be used to route an input signal to one of several possible outputs.
- (g) Can be used to generate arbitrary logic functions.

- 9-40. Show how the 7442 decoder can be used as 1-to-8 demultiplexer. (*Hint*: See Problem 9-7.)
- 9-41.\* Apply the waveforms of Figure 9-75 to the inputs of the 74ALS138 DEMUX of Figure 9-30(a) as follows:
- $$D \rightarrow A_2 \quad C \rightarrow A_1 \quad B \rightarrow A_0 \quad A \rightarrow \bar{E}_1$$
- Draw the waveforms at the DEMUX outputs.
- 9-42. Consider the system of Figure 9-31. Assume that the clock frequency is 10 pps. Describe what the monitoring panel indications will be for each of the following cases.
- All doors closed
  - All doors open
  - Doors 2 and 6 open
- C, D** 9-43.\* Modify the system of Figure 9-31 to handle 16 doors. Use a 74150 16-input MUX and two 74LS138 DEMUXes. How many lines are going to the remote monitoring panel?
- 9-44. Draw the waveforms at transmit\_data, and DEMUX outputs  $O_0$ ,  $O_1$ ,  $O_2$ , and  $O_3$  in Figure 9-33 for the following register data loaded into the transmit registers in Figure 9-32:  $[A] = 0011$ ,  $[B] = 0110$ ,  $[C] = 1001$ ,  $[D] = 0111$ .
- 9-45. Figure 9-81 shows an  $8 \times 8$  graphic LCD display grid controlled by a 74HC138 configured as a decoder, and a 74HC138 configured as a demultiplexer. Draw 48 cycles of the clock and the data input necessary to activate the pixels shown on the display.

**FIGURE 9-81** Problem 9-45.



**SECTION 9-9**

- T** 9-46. Consider the control sequencer of Figure 9-26. Describe how each of the following faults will affect the operation.
  - (a)\*The  $I_3$  input of the MUX is shorted to ground.
  - (b) The connections from sensors 3 and 4 to the MUX are reversed.
- T** 9-47\* Consider the circuit of Figure 9-24. A test of the circuit yields the results shown in Table 9-10. What are the possible causes of the malfunction?

**TABLE 9-10**

		Actual Count	Displayed Count
Case 1	Counter 1	33	33
	Counter 2	47	47
Case 2	Counter 1	82	02
	Counter 2	64	64
Case 3	Counter 1	63	63
	Counter 2	95	15

- T** 9-48\* A test of the security monitoring system of Figure 9-31 produces the results recorded in Table 9-11. What are the possible faults that could cause this operation?

**TABLE 9-11**

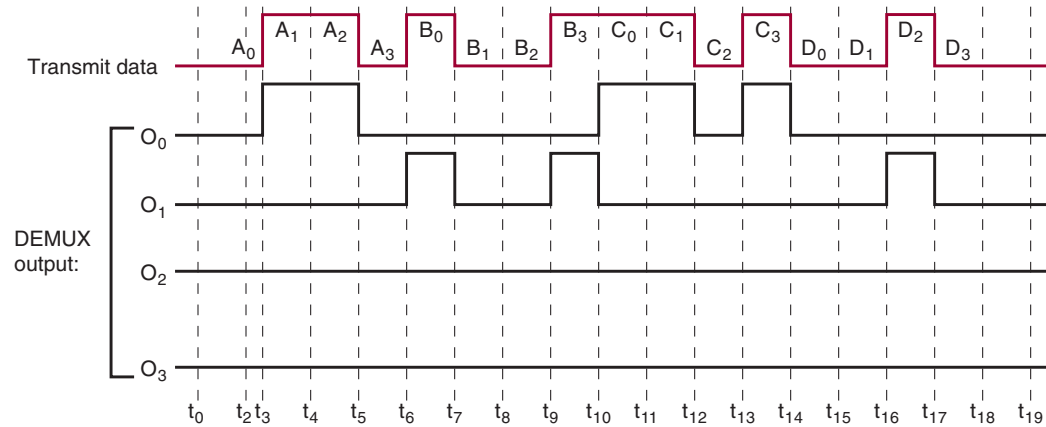
Condition	LEDs
All doors closed	All LEDs off
Door 0 open	LED 0 flashing
Door 1 open	LED 2 flashing
Door 2 open	LED 1 flashing
Door 3 open	LED 3 flashing
Door 4 open	LED 4 flashing
Door 5 open	LED 6 flashing
Door 6 open	LED 5 flashing
Door 7 open	LED 7 flashing

- T** 9-49\* A test of the security monitoring system of Figure 9-31 produces the results recorded in Table 9-12. What are the possible faults that could cause this operation? How can this be verified or eliminated as a fault?

**TABLE 9-12**

Condition	LEDs
All doors closed	All LEDs off
Door 0 open	LED 0 flashing
Door 1 open	LED 1 flashing
Door 2 open	LED 2 flashing
Door 3 open	LED 3 flashing
Door 4 open	LED 4 flashing
Door 5 open	LED 5 flashing
Door 6 open	No LED flashing
Door 7 open	No LED flashing
Doors 6 and 7 open	LEDs 6 and 7 flashing

- T** 9-50\* The synchronous data transmission system of Figure 9-32 and Figure 9-33 is malfunctioning. An oscilloscope is used to monitor the MUX and DEMUX outputs during the transmission cycle, with the results shown in Figure 9-82. What are the possible causes of the malfunction?



**FIGURE 9-82** Problem 9-50.

- T** 9-51. The synchronous data transmission system of Figures 9-32 and 9-33 is not working properly and the troubleshooting tree diagram of Figure 9-38 has been used to isolate the problem to the timing and control section of the receiver. Draw a troubleshooting tree diagram that will isolate the problem further to one of the four blocks in that section (FF1, Bit counter, Word counter, or FF2). Assume that all wires are connected as shown, with no wiring errors.

### SECTION 9-10

- C, D** 9-52. Redesign the circuit of Problem 9-16 using a 74HC85 magnitude comparator. Add a “copy overflow” feature that will activate an ALARM output if the OPERATE output fails to stop the machine when the requested number of copies is done.
- D** 9-53. (a)\* Show how to connect 74HC85s to compare two 10-bit numbers.  
(b) Create a 10-bit comparator using a megafunction.

### SECTION 9-11

- 9-54. Assume a BCD input of 69 to the code converter of Figure 9-43. Determine the levels at each  $\Sigma$  output and at the final binary output.
- T** 9-55\* A technician tests the code converter of Figure 9-43 and observes the following results:

<i>BCD Input</i>	<i>Binary Output</i>
52	0110011
95	1100000
27	0011011

What is the probable circuit fault?

**SECTIONS 9-12 TO 9-14**

**B 9-56. DRILL QUESTION**

*True or false:*

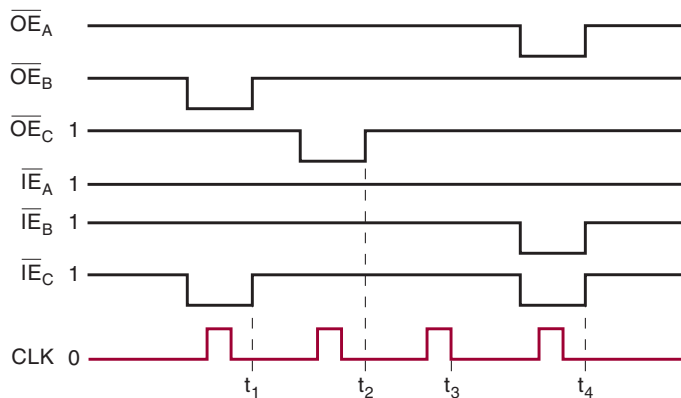
- (a) A device connected to a data bus should have tristate outputs.
- (b) Bus contention occurs when more than one device takes data from the bus.
- (c) Larger units of data can be transferred over an eight-line data bus than over a four-line data bus.
- (d) A bus driver IC generally has a high output impedance.
- (e) Bidirectional registers and buffers have common I/O lines.

9-57\* For the bus arrangement of Figure 9-47, describe the input signal requirements for simultaneously transferring the contents of register C to both of the other registers.

9-58. Assume that the registers in Figure 9-47 are initially [A] = 1011, [B] = 1000, and [C] = 0111. The signals in Figure 9-83 are applied to the register inputs.

- (a) Determine the contents of each register at times  $t_1$ ,  $t_2$ ,  $t_3$ , and  $t_4$ .
- (b) Describe what would happen if  $\overline{IE}_A$  were LOW when the third clock pulse occurred.

**FIGURE 9-83** Problems 9-58 and 9-59.



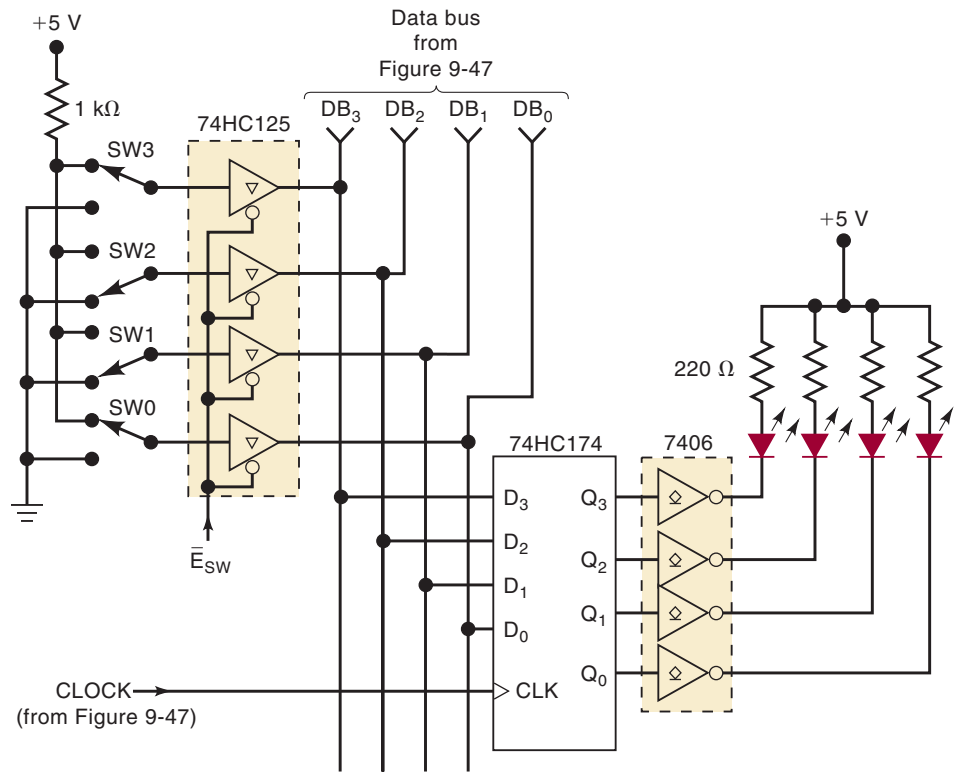
9-59. Assume the same initial conditions of Problem 9-58, and sketch the signal on  $DB_3$  for the waveforms of Figure 9-83.

9-60. Figure 9-84 shows two more devices that are to be added to the data bus of Figure 9-47. One is a set of buffered switches that can be used to enter data manually into any of the bus registers. The other device is an output register that is used to latch any data that are on the bus during a data transfer operation and display them on a set of LEDs.

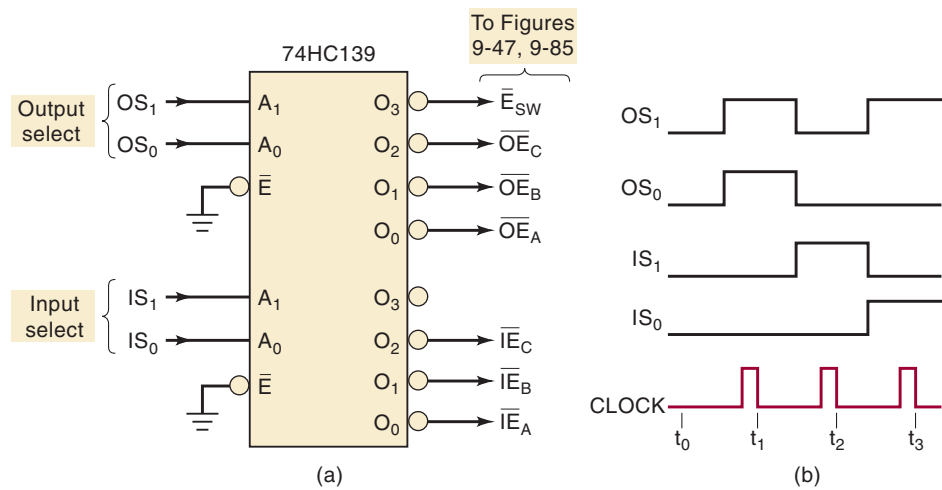
- (a) Assume that all registers contain 0000. Outline the sequence of operations needed to load the registers with the following data from the switches: [A] = 1011, [B] = 0001, [C] = 1110.
- (b) What will the state of the LEDs be at the end of this sequence?

**C 9-61.** Now that the circuitry of Figure 9-85 has been added to Figure 9-47, a total of five devices are connected to the data bus. The circuit in Figure 9-85(a) will now be used to generate the enable signals needed to perform the different data transfers over the data bus. It uses a 74HC139 chip that contains two identical independent 1-of-4 decoders with an active-LOW enable. The top decoder is used to select the device that will put data on the data bus (output select), and the

**FIGURE 9-84** Problems 9-60, 9-61, and 9-62.



**FIGURE 9-85** Problem 9-61.



bottom decoder is used to select the device that is to take the data from the data bus (input select). Assume that the decoder outputs are connected to the corresponding enable inputs of the devices tied to the data bus. Also assume that all registers initially contain 0000 at time  $t_0$ , and the switches are in the positions shown in Figure 9-84.

(a)\* Determine the contents of each register at times  $t_1$ ,  $t_2$ , and  $t_3$  in response to the waveforms in Figure 9-85(b).

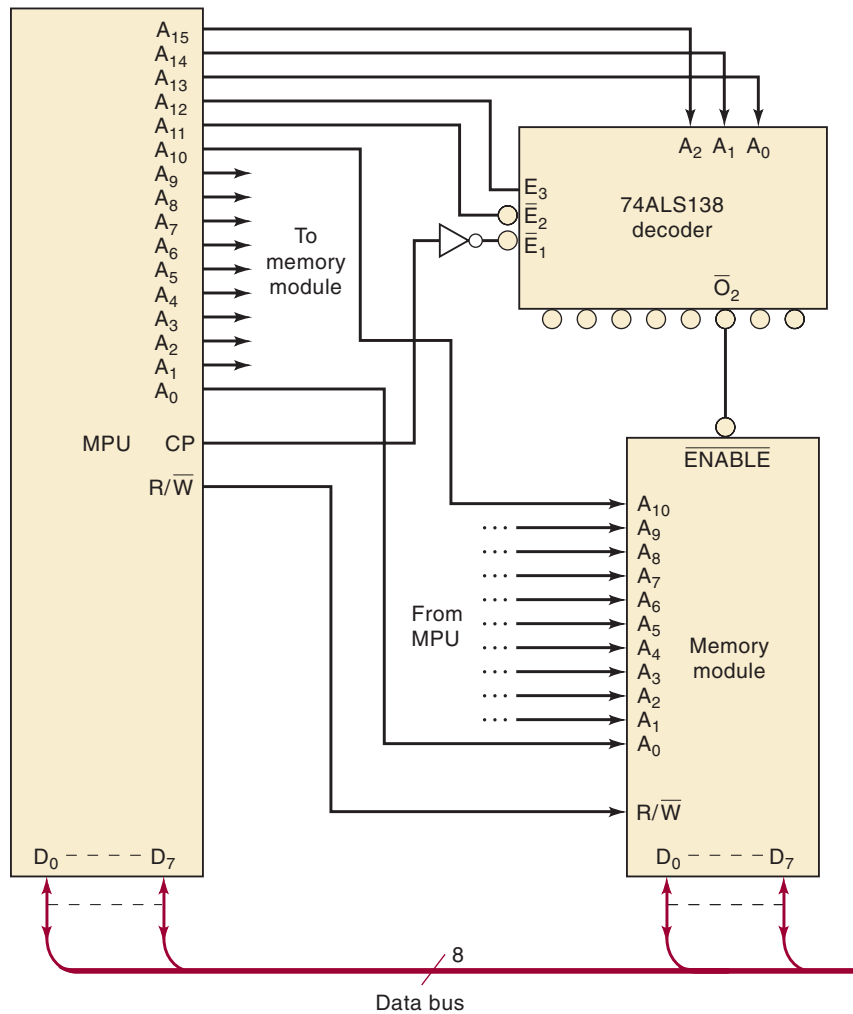
(b) Can bus contention ever occur with this circuit? Explain.

9-62. Show how a 74HC541 (Figure 9-50) can be used in the circuit of Figure 9-84.

**MICROCOMPUTER APPLICATIONS**

- C, N** 9-63.\*Figure 9-86 shows the basic circuitry to interface a microprocessor (MPU) to a memory module. The memory module will contain one or more memory ICs (Chapter 12) that can either receive data from the MPU (a WRITE operation) or send data to the MPU (a READ operation). The data are transferred over the eight-line data bus. The MPU's data lines and the memory's I/O data lines are connected to this common bus. For now we will be concerned with how the MPU controls the selection of the memory module for a READ or WRITE operation. The steps involved are as follows:
1. The MPU places the memory address on its address output lines  $A_{15}$  to  $A_0$ .
  2. The MPU generates the  $R/\overline{W}$  signal to inform the memory module which operation is to be performed:  $R/\overline{W} = 1$  for READ,  $R/\overline{W} = 0$  for WRITE.
  3. The upper five bits of the MPU address lines are decoded by the 74ALS138, which controls the ENABLE input of the memory module. This ENABLE input must be active in order for the memory module to do a READ or WRITE operation.

**FIGURE 9-86** Basic microprocessor-to-memory interface circuit for Problem 9-63.





4. The other 11 address bits are connected to the memory module, which uses them to select the specific *internal* memory location being accessed by the MPU, provided that ENABLE is active.

In order to read from or write into the memory module, the MPU must put the correct address on the address lines to enable the memory, and then pulse *CP* to the HIGH state.

- (a) Determine which, if any, of these hexadecimal addresses will activate the memory module: 607F, 57FA, 5F00.
- (b) Determine what range of hex addresses will activate the memory. (*Hint*: Inputs  $A_0$  to  $A_{10}$  to memory can be any combination.)
- (c) Assume that a second identical memory module is added to the circuit with its address,  $R/\overline{W}$ , and data I/O lines connected exactly the same as the first module *except* that its ENABLE input is tied to decoder output  $\overline{O}_4$ . What range of hex addresses will activate this second module?
- (d) Is it possible for the MPU to read from or write to both modules at the same time? Explain.

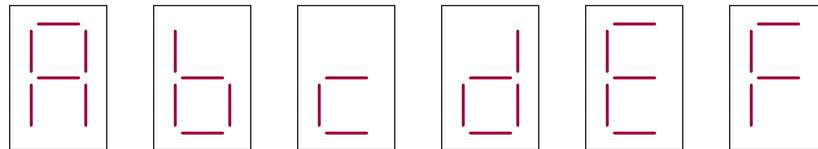
#### DESIGN PROBLEM

- C, D** 9-64. The keyboard entry circuit of Figure 9-16 is to be used as part of an electronic digital lock that operates as follows: when activated, an UNLOCK output goes HIGH. This HIGH is used to energize a solenoid that retracts a bolt and allows a door to be opened. To activate UNLOCK, the operator must press the CLEAR key and then enter the correct three-key sequence.
- (a) Show how 74HC85 comparators and any other needed logic can be added to the keyboard entry circuit to produce the digital lock operation described above for a key sequence of CLEAR-3-5-8.
  - (b) Modify the circuit to activate an ALARM output if the operator enters something other than the correct three-key sequence.

#### SECTIONS 9-15 TO 9-20

- H, D** 9-65\* Write the HDL code for a BCD-to-decimal decoder (the equivalent of a 7442).
- H, D** 9-66. Write the HDL code for a hex decoder/driver for a 7-segment display. The first 10 characters should appear as shown in Figure 9-7. The last six characters should appear as shown in Figure 9-87.

**FIGURE 9-87** Hex characters for Problem 9-66.



- B, H** 9-67. Write a low-priority ENCODER description that will always encode the lowest number if two inputs are activated simultaneously.
- H** 9-68. Rewrite the code of the four-bit comparator of Figures 9-69 or 9-70 to make an eight-bit comparator without using macrofunctions.
- H** 9-69. Use HDL to describe a four-bit binary number to a two-digit BCD code converter.
- H** 9-70. Use HDL to describe a three-digit BCD code to eight-bit binary number converter. (Maximum BCD input is 255.)

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

---

### SECTION 9-1

1. No    2. The enable input controls whether or not the decoder logic responds to the input binary code.    3. The 7445 has open-collector outputs that can handle up to 30 V and 80 mA.    4. 24 pins: 2 enables, 4 inputs, 16 outputs,  $V_{CC}$ , and ground

### SECTION 9-2

1.  $a, b, c, f, g$     2. True

### SECTION 9-3

1. LEDs: (a), (e), (f). LCDs: (b), (c), (d), (e)    2. (a) four-bit BCD, (b) seven- or eight-bit ASCII, (c) binary value for pixel intensity

### SECTION 9-4

1. An encoder produces an output code corresponding to the activated input. A decoder activates one output corresponding to an applied input code.    2. In a priority encoder, the output code corresponds to the *highest*-numbered input that is activated.    3. Normal BCD = 0110    4. (a) produces a PGT when a key is pressed, (b) converts key actuation to its BCD code, (c) generates bounce-free pulse to trigger the ring counter, (d) forms a ring counter that sequentially clocks output registers, (e) stores BCD codes generated by key actuations    5.  $\overline{E}_1$  and  $\overline{E}_0$  are used for cascading and  $\overline{GS}$  indicates an active input.

### SECTION 9-5

1. Observation/analysis and divide and conquer.    2. Observe the symptoms and analyze the system for possible causes.    3. The circuit is divided into two sections: one section is proven to work properly and the other section contains the fault.

### SECTION 9-6

1. The binary number at the select inputs determines which data input will pass through to the output.    2. Thirty-two data inputs and five select inputs

### SECTION 9-7

1. Parallel-to-serial conversion, data routing, logic-function generation, operations sequencing    2. False; they are applied to the select inputs.    3. Counter

### SECTION 9-8

1. A MUX selects one of many input signals to be passed to its output; a DEMUX selects one of many outputs to receive the input signal.    2. True, provided that the decoder has an ENABLE input    3. The LEDs will go on and off in sequence.

### SECTION 9-10

1. To provide a means for expanding the compare operations to numbers with more than four bits.    2.  $O_{A=B} = 1$ ; other outputs are 0.    3. Expanding to compare larger numbers is accomplished by specifying more input bits.

### SECTION 9-11

1. A code converter takes input data represented in one type of binary code and converts it to another type of binary code.    2. Three digits can represent decimal values up to 999. To represent 999 in straight binary requires 10 bits.

---

**SECTION 9-12**

1. A set of connecting lines to which the inputs and outputs of many different devices can be connected    2. Bus contention occurs when the outputs of more than one device connected to a bus are enabled at the same time. It is prevented by controlling the device enable inputs so that this cannot happen.    3. A condition in which all devices connected to a bus are in the Hi-Z state

**SECTION 9-13**

1. 1011    2. True    3. 0000

**SECTION 9-14**

1. Bus contention    2. Floating, Hi-Z    3. Provides tristate low-impedance outputs    4. Reduces the number of IC pins and the number of connections to the data bus    5. See Figure 9-54.

**SECTION 9-15**

1. They are enable inputs. All must be active for the decoder to work.    2. The CASE and the TABLE    3. The selected signal assignment statement and the CASE

**SECTION 9-16**

1. The combination input/output pin BI/RBO    2. Common anode. The outputs connect to the cathodes and go LOW to light the segments.    3. The IF/ELSE control structure is evaluated sequentially and gives precedence in the order in which decisions are listed.

**SECTION 9-17**

1. The don't-care entry in a truth table and the IF/ELSE control structure  
 2. The IF/ELSE control structure and the conditional signal assignment statement  
 3. By use of the :TRI primitive and assigning a value to OE    4. By use of the IEEE STD\_LOGIC or STD\_LOGIC\_VECTOR data types that have a possible value of Z

**SECTION 9-18**

1. Inputs: *ch0*, *ch1*, *ch2*, *ch3*; output: *dout*; control inputs (Select): *s*    2. Input *din*; outputs: *ch0*, *ch1*, *ch2*, *ch3*; control inputs (Select): *s*    3. DEFAULTS    4. ELSE

**SECTION 9-19**

1. Numerical data objects (e.g., INTEGER in VHDL)    2. IF/ELSE  
 3. Relational operators (<, >)

**SECTION 9-20**

1. 10    2. By multiplying by 8 + 2. Shifting the BCD digit three places left multiplies by 8, and shifting the same BCD digit one place left multiplies by 2. Adding these results produced the BCD digit multiplied by 10.    3. VHDL simply uses the \* operator to multiply.

*This page intentionally left blank*



**DIGITAL SYSTEM  
PROJECTS USING HDL**

■ **OUTLINE**

10-1 Small-Project Management

10-2 Stepper Motor Driver  
Project

10-3 Keypad Encoder Project

10-4 Digital Clock Project

10-5 Microwave Oven Project

10-6 Frequency Counter  
Project

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Analyze the operation of systems made of several components that have been covered earlier in this textbook.
- Describe an entire project with one HDL file.
- Describe the process of hierarchical project management.
- Divide a project into manageable pieces.
- Use Quartus II software tools to implement a hierarchical modular project.
- Develop strategies to test the operation of digital circuits.

## ■ INTRODUCTION

Throughout the first nine chapters of this book, we have explained the fundamental building blocks of digital systems. Now that we have taken out each block and looked it over, we do not want to put them all away and forget them; it is time to build something with the blocks. Some of the examples we have used to demonstrate the operation of individual circuits are really digital systems in their own right, and we have studied how they work. In this chapter, we want to focus more on the building process.

Surveys of graduates show us that most of the professionals in the electrical and computer engineering and technology field have the responsibility of project management. Experience with students has also shown us that the most efficient way to manage a project is not intuitively obvious to everyone, which explains why so many of us end up attending the school of hard knocks (learning through trial and error). This chapter is intended to give you a strategic plan for managing projects while learning about digital systems and the modern tools used to develop them. The principles here are not limited to digital or even electronic projects in general. They could apply to building a house or building your own business. They will definitely improve your success rate and reduce the frustration factor.

Hardware description languages were really created for the purpose of managing large digital systems: for documentation, simulation testing, and the synthesis of working circuits. Likewise, the Altera software tools are specifically designed to work with managing projects that go far beyond the scope of this text. We will describe some of the features of the Altera software packages as we go through the steps of developing these small projects. This concept of modular, hierarchical project development, which was introduced in Chapter 4, will be demonstrated here through a series of examples.

## 10-1 SMALL-PROJECT MANAGEMENT

---

### OUTCOMES

*Upon completion of this section, you will be able to:*

- List the steps of project management.
- Identify subtasks, strategies, and purpose within each step.

The first projects described here are relatively small systems that consist of a small number of building blocks. These projects can be developed in separate modules, but this approach would only add to the complexity. They are small enough that it makes sense to implement the entire project in a single HDL design file. This does not mean, however, that a structured process should *not* be followed to complete the project. In fact, most of the same steps that should be employed in a large modular project are also applicable in these examples. The steps that should be followed are (1) overall definition, (2) strategic planning (problem decomposition) to break the project into small pieces, (3) synthesis and testing of each piece, and (4) system integration and testing.

### Definition

The first step in any project is the thorough definition of its scope. In this step, the following issues should be determined:

- How many bits of data are needed?
- How many devices are controlled by the outputs?
- What are the names of each input and output?
- Are the inputs and outputs active-HIGH or active-LOW?
- What are the speed requirements?
- Do I understand fully how this device should operate?
- What will define successful completion of this project?

From this step should come a complete and thorough description of the overall project's operation, a definition of its inputs and outputs, and complete numeric specifications that define its capabilities and limitations.

### Strategic Planning/Problem Decomposition

The second step involves developing a strategy for dividing this overall project into manageable pieces. This process is often referred to as decomposition of the problem because the overall function is defined in terms of several simpler functional blocks. The requirements of the pieces are:

- A way to test each piece must be developed.
- Each piece must fit together to make up the whole system.
- We must know the nature of all the signals that connect the pieces.
- The exact operation of each block must be thoroughly defined and understood.
- We must have a clear vision of how to make each block work.

This last requirement might seem obvious, but it is amazing how many projects are planned around one central block that involves a not-yet-discovered technical miracle or violates silly little laws like conservation of energy.

---

In this stage, each subsystem (section block) becomes somewhat of a project in and of itself, with the possibility of additional subsystems defined within its boundaries. This is the concept of hierarchical design.

### Synthesis and Testing

Each subsystem should be built starting at the simplest level. In the case of a digital system designed using HDL, it means writing pieces of code. It also means developing a plan for testing that code to make sure it meets all the criteria. This is often accomplished through some sort of simulation. When a circuit is simulated on a computer, the designer must create all the different scenarios that will be experienced by the actual circuit and must also know what the proper response to those inputs should be. This testing often takes a great deal of thought and is not an area that should be overlooked. The worst mistake you can make is to conclude that a fundamental block works perfectly, only to find later those few situations where it fails. This predicament often forces you to rethink many of the other blocks, thus nullifying much of your work.

### System Integration and Testing

The last step is to put the blocks together and test them as a unit. Blocks are added and tested at each stage until the entire project is working. This area is often trivialized but rarely goes smoothly. Even if you took care of all the details you thought about, there are always the “gotchas” that nobody thought about.

Some aspects of project planning and management go beyond the scope of this text. One is the selection of a hardware platform that will best fit your application. In Chapter 13, we will explore the broad field of digital systems and look specifically at the capabilities and limitations of PLDs in various categories. Another very critical dimension in project management is time. Your boss will give you only a certain amount of time to complete your project, and you must plan your work (and effort) to meet this deadline. We will not be able to cover time management in this text, but as a general rule you will find that most facets of the project will actually take two to three times longer than you think they will when you begin.

#### OUTCOME ASSESSMENT QUESTIONS

1. Name the steps of project management.
2. At what stage should you decide how to measure success?

## 10-2 STEPPER MOTOR DRIVER PROJECT

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe the requirements of a stepper motor driver system.
- Define each input and output.
- Describe the logical operation of the system using HDL.

The purpose of this section is to demonstrate a typical application of counters combined with decoding circuits. A digital system often contains a counter that cycles through a specified sequence and whose output states are



decoded by a combinational logic circuit, which in turn controls the operation of the system. Many applications also have external inputs that are used to put the system into various modes of operation. This section discusses all these features to control a stepper motor.

In a real project, the first step of definition often involves some research on the part of the project manager. In this section (or project), it is vital that we understand what a stepper motor is and how it works before we try to create a circuit that is supposed to control it. In Section 7-10, we showed you how to design a simple synchronous counter that could be used to drive a stepper motor. The sequence demonstrated in that section is called the *full-step sequence*. As you recall, it involved two flip-flops and their  $Q$  and  $\bar{Q}$  outputs driving the four coils of the motor. The full-step sequence always has two coils of the stepper motor energized in any state of the sequence and typically causes  $15^\circ$  of shaft rotation per step. Other sequences, however, will also cause a stepper to rotate. If you look at the full-step sequence of Table 10-1, you will notice that each state transition involves turning off one coil and simultaneously turning on another coil. For example, look at the first state (1010) in the full-step sequence. When it switches to the second state in the sequence, *coil 1* is turned off and *coil 0* is turned on. The *half-step sequence* is created by inserting a state with only one coil energized between full steps, as shown in the middle column of Table 10-1. In this sequence, one coil is de-energized before the other is energized. The first state is 1010 and the second state is 1000, meaning that *coil 1* is turned off for one state before *coil 0* is turned on. This intermediate state causes the stepper shaft to rotate half as far ( $7.5^\circ$ ) as it would in the full-step sequence ( $15^\circ$ ). The half-step sequence is used when smaller steps are desirable and more steps per revolution are acceptable. As it turns out, the stepper motor will rotate in a manner similar to the full-step sequence ( $15^\circ$  per step) if you apply only the sequence of intermediate states with one coil energized at a time. This sequence, called the *wave-drive sequence*, has less torque but operates more smoothly than the full-step sequence at moderate speeds. The wave-drive sequence is shown in the right-hand column of Table 10-1.

**TABLE 10-1** Stepper motor coil drive sequences.

Full-Step Sequence	Half-Step Sequence	Wave-Drive Sequence
Coil 3210	Coil 3210	Coil 3210
1010	1010	
	1000	1000
1001	1001	
	0001	0001
0101	0101	
	0100	0100
0110	0110	
	0010	0010

### Problem Definition

A microprocessor laboratory needs a universal interface to drive a stepper motor. In order to experiment with microcontrollers driving stepper motors, it would be useful to have a single universal interface IC wired to the stepper motor. This circuit needs to accept any of the typical forms of stepper drive signals from a microcontroller and activate the windings of the motor to make it move in the desired manner. The interface needs to operate in

one of four modes: decoded full-step, decoded half-step, decoded wave-drive, or nondecoded direct-drive. The mode is selected by controlling the logic levels on the *M1*, *M0* input pins. In the first three modes, the interface receives just two control bits—a step pulse and a direction control bit—from the microcontroller. Each time it sees a *rising* edge on the step input, the circuit must cause the motor to move one increment of motion clockwise or counterclockwise, depending on the level present on the direction bit. Depending on the mode that the IC is in, the outputs will respond to each step pulse by changing state according to the motor coil drive sequences shown in Table 10-1. The fourth mode of operation of this circuit must allow the microcontroller to control each winding of the motor directly. In this mode, the circuit accepts four control bits from the microcontroller and passes these logic levels directly to its outputs, which are used to energize the stepper coils. The four modes are summarized in Table 10-2.

**TABLE 10-2** Mode definitions for the stepper driver.

Mode	M1	M0	Input Signals	Output
0	0	0	Step, direction	Full-step count sequence
1	0	1	Step, direction	Wave-drive count sequence
2	1	0	Step, direction	Half-step count sequence
3	1	1	Four control inputs	Direct drive from control inputs

In modes 0, 1, and 2, the outputs count through the corresponding count sequence on each rising edge of the step input. The direction input described in Table 10-2 determines whether the output sequence progresses downward or upward through the states in Table 10-1, thus moving the motor clockwise or counterclockwise. From this description, we can make some decisions about the project.

#### Inputs

*step*: rising edge trigger

*direction*: 0 = upward through table, 1 = downward through table

*cin0*, *cin1*, *cin2*, *cin3*, *m1*, *m0*: active-HIGH control inputs

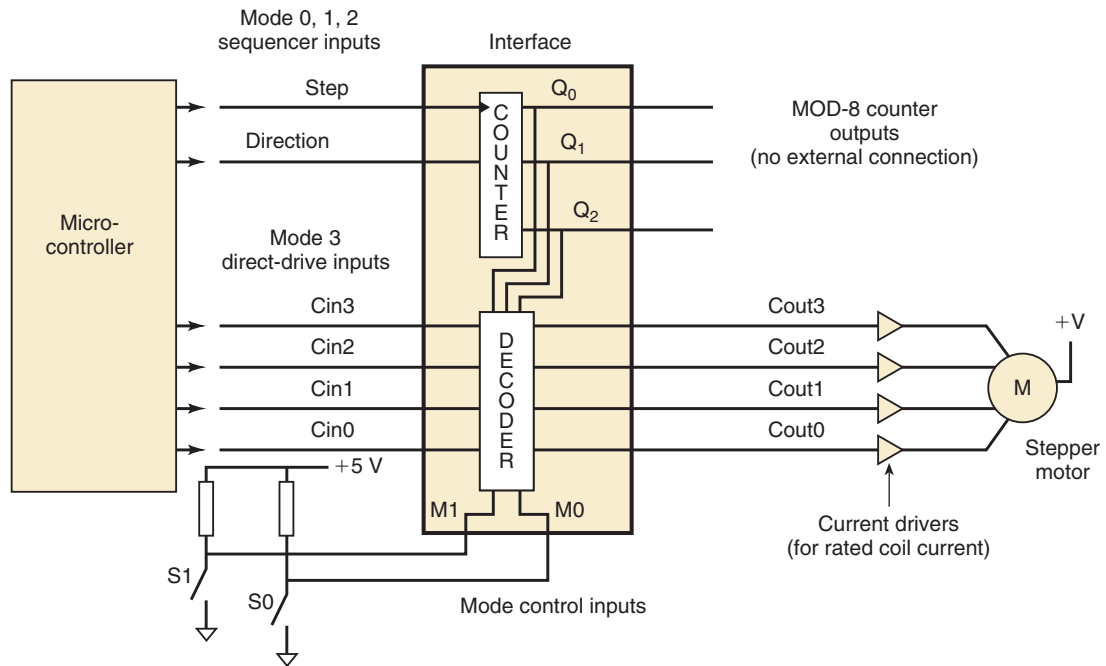
#### Outputs

*cout0*, *cout1*, *cout2*, *cout3*: active-HIGH control outputs

## Strategic Planning/Problem Decomposition

This project has two key requirements. It requires a sequential counter circuit that will control the outputs in three of the modes. In the last mode, the output does not follow a counter but rather follows the control inputs. While there are many ways to divide this project and still fulfill these requirements, we will choose to decompose it into two functional blocks: a counter and a decoder. The first is a simple up/down binary counter that responds to the *step* and *direction* inputs. The second block is a combinational logic circuit that translates (decodes) the binary count into the appropriate output state, depending on the mode input setting. This circuit will also ignore the counter inputs and pass the control inputs directly to the outputs when the mode is set to 3. The circuit diagram is shown in Figure 10-1.

The development and test planning is also fairly straightforward. The first step is to build an up/down counter. This counter should be tested on a simulator using only the direction and step inputs. Next, try to make each decoded sequence work individually with the counter. Then try to get the



**FIGURE 10-1** A universal stepper motor interface circuit.

mode inputs to select one of the decoder sequences and add the direct-drive option (which is fairly trivial). When the circuit can follow the states shown in Table 10-1 in either direction, for each mode sequence, and pass the four *cin* signals directly to *cout* in mode 3, we will be successful.

## Synthesis and Testing

The code in Figures 10-2 and 10-3 shows the first stage of development: designing and testing an up/down counter. We will use an intermediate

```

SUBDESIGN fig10_2
(
  step, dir      :INPUT;
  q[2..0]       :OUTPUT;
)
VARIABLE
count[2..0]     : DFF;

BEGIN
  count[].clk = step;
  IF dir THEN count[].d = count[].q + 1;
  ELSE      count[].d = count[].q - 1;
  END IF;
  q[] = count[].q;
END;

```

**FIGURE 10-2** AHDL MOD-8.

```

ENTITY fig10_3 IS
PORT( step, dir      :IN BIT;
      q              :OUT INTEGER RANGE 0 TO 7);
END fig10_3;

ARCHITECTURE vhdl OF fig10_3 IS
BEGIN
  PROCESS (step)
  VARIABLE count :INTEGER RANGE 0 TO 7;
  BEGIN
    IF (step'EVENT AND step = '1') THEN
      IF dir = '1' THEN count:= count + 1;
      ELSE      count:= count - 1;
      END IF;
    END IF;
    q <= count;
  END PROCESS;
END vhdl;

```

**FIGURE 10-3** VHDL MOD-8.

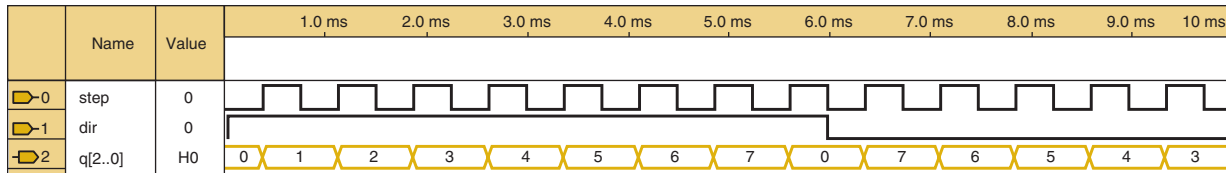


FIGURE 10-4 Simulation testing of a basic MOD-8.

integer variable for the counter value and test it by outputting the count directly to  $q$ . To test this part of the design, we simply need to make sure it can count up and down through the eight states. Figure 10-4 shows the simulation results. We only need to provide the clock pulses and make up a direction control signal, and the simulator demonstrates the counter's response.

The next step is to add one of the decoded outputs and test it, which will require adding the four-bit  $cout$  output specification. The  $q$  output bits of the MOD-8 counter are kept for the sake of continuity. Figure 10-5 shows the AHDL code for this stage of testing, and Figure 10-6 shows the VHDL code for the same stage of testing. Notice that a CASE construct is used to decode the counter and drive the outputs. In the VHDL code, the  $cout$  outputs have been declared as `bit_vector` type because we now want to assign binary bit patterns to them. Figure 10-7 shows the simulated test of its operation with enough clock cycles included to test an entire counter cycle up and down.

The other count sequences are simply variations of the code we just tested. It is probably not necessary to test each one independently, so now is a good time to bring in the mode selector inputs ( $m$ ) and direct-drive coil control inputs ( $cin$ ). Notice that the new inputs have been defined in Figures 10-8 (AHDL) and 10-9 (VHDL). Because the mode control has

```

SUBDESIGN fig10_5
(
  step, dir      :INPUT;
  q[2..0]       :OUTPUT;
  cout[3..0]    :OUTPUT;
)
VARIABLE
  count[2..0]   : DFF;

BEGIN
  count[].clk = step;
  IF dir THEN count[].d = count[].q + 1;
  ELSE count[].d = count[].q - 1;
  END IF;
  q[] = count[].q;
  CASE count[] IS
    WHEN B"000" => cout[] = B"1010";
    WHEN B"001" => cout[] = B"1001";
    WHEN B"010" => cout[] = B"0101";
    WHEN B"011" => cout[] = B"0110";
    WHEN B"100" => cout[] = B"1010";
    WHEN B"101" => cout[] = B"1001";
    WHEN B"110" => cout[] = B"0101";
    WHEN B"111" => cout[] = B"0110";
  END CASE;
END;

```

FIGURE 10-5 AHDL full-step sequence decoder.

```

ENTITY fig10_6 IS
PORT ( step, dir  :IN BIT;
      q           :OUT INTEGER RANGE 0 TO 7;
      cout       :OUT BIT_VECTOR (3 downto 0));
END fig10_6;

ARCHITECTURE vhdl OF fig10_6 IS
BEGIN
  PROCESS (step)
  VARIABLE count :INTEGER RANGE 0 TO 7;
  BEGIN
    IF (step'EVENT AND step = '1') THEN
      IF dir = '1' THEN count := count + 1;
      ELSE count := count - 1;
      END IF;
      q <= count;
    END IF;
    CASE count IS
      WHEN 0 => cout <= B"1010";
      WHEN 1 => cout <= B"1001";
      WHEN 2 => cout <= B"0101";
      WHEN 3 => cout <= B"0110";
      WHEN 4 => cout <= B"1010";
      WHEN 5 => cout <= B"1001";
      WHEN 6 => cout <= B"0101";
      WHEN 7 => cout <= B"0110";
    END CASE;
  END PROCESS;
END vhdl;

```

FIGURE 10-6 VHDL full-step sequence decoder.

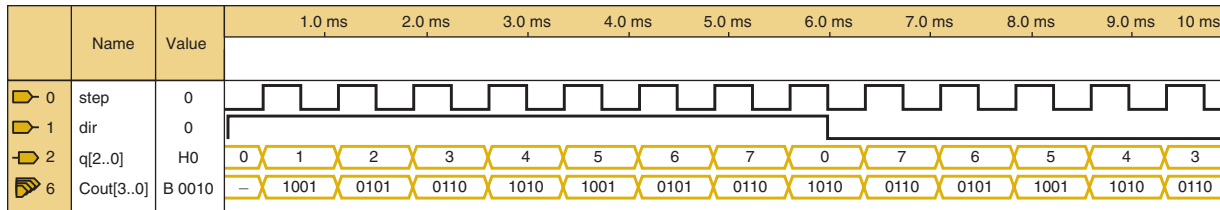


FIGURE 10-7 Simulation testing of decoded sequence.

FIGURE 10-8 AHDL stepper driver.

```

SUBDESIGN fig10_8
(
  step, dir           :INPUT;
  m[1..0], cin[3..0] :INPUT;
  cout[3..0], q[2..0] :OUTPUT;
)
VARIABLE
  count[2..0] : DFF;
BEGIN
  count[].clk = step;
  IF dir THEN count[].d = count[].q + 1;
  ELSE       count[].d = count[].q - 1;
  END IF;
  q[] = count[].q;
  CASE m[] IS
  WHEN 0 =>
    CASE count[] IS          -- FULL STEP
    WHEN B"000" => cout[] = B"1010";
    WHEN B"001" => cout[] = B"1001";
    WHEN B"010" => cout[] = B"0101";
    WHEN B"011" => cout[] = B"0110";
    WHEN B"100" => cout[] = B"1010";
    WHEN B"101" => cout[] = B"1001";
    WHEN B"110" => cout[] = B"0101";
    WHEN B"111" => cout[] = B"0110";
    END CASE;
  WHEN 1 =>
    CASE count[] IS          -- WAVE DRIVE
    WHEN B"000" => cout[] = B"1000";
    WHEN B"001" => cout[] = B"0001";
    WHEN B"010" => cout[] = B"0100";
    WHEN B"011" => cout[] = B"0010";
    WHEN B"100" => cout[] = B"1000";
    WHEN B"101" => cout[] = B"0001";
    WHEN B"110" => cout[] = B"0100";
    WHEN B"111" => cout[] = B"0010";
    END CASE;
  WHEN 2 =>
    CASE count[] IS          -- HALF STEP
    WHEN B"000" => cout[] = B"1010";
    WHEN B"001" => cout[] = B"1000";
    WHEN B"010" => cout[] = B"1001";
    WHEN B"011" => cout[] = B"0001";
    WHEN B"100" => cout[] = B"0101";
    WHEN B"101" => cout[] = B"0100";
    WHEN B"110" => cout[] = B"0110";
    WHEN B"111" => cout[] = B"0010";
    END CASE;
  WHEN 3 => cout[] = cin[]; -- Direct Drive
  END CASE;
END;

```

**FIGURE 10-9** VHDL stepper driver.

```
ENTITY fig10_9 IS
PORT (  step, dir      :IN BIT;
        m              :IN BIT_VECTOR (1 DOWNTO 0);
        cin            :IN BIT_VECTOR (3 DOWNTO 0);
        q              :OUT INTEGER RANGE 0 TO 7;
        cout           :OUT BIT_VECTOR (3 DOWNTO 0));
END fig10_9;

ARCHITECTURE vhdl OF fig10_9 IS
BEGIN
  PROCESS (step)
  VARIABLE count      :INTEGER RANGE 0 TO 7;
  BEGIN
    IF (step'EVENT AND step = '1') THEN
      IF dir = '1' THEN count := count + 1;
      ELSE              count := count - 1;
      END IF;
    END IF;
    q <= count;
  CASE m IS
    WHEN "00" =>                -- FULL STEP
      CASE count IS
        WHEN 0  => cout <= "1010";
        WHEN 1  => cout <= "1001";
        WHEN 2  => cout <= "0101";
        WHEN 3  => cout <= "0110";
        WHEN 4  => cout <= "1010";
        WHEN 5  => cout <= "1001";
        WHEN 6  => cout <= "0101";
        WHEN 7  => cout <= "0110";
      END CASE;
    WHEN "01" =>                -- WAVE DRIVE
      CASE count IS
        WHEN 0  => cout <= "1000";
        WHEN 1  => cout <= "0001";
        WHEN 2  => cout <= "0100";
        WHEN 3  => cout <= "0010";
        WHEN 4  => cout <= "1000";
        WHEN 5  => cout <= "0001";
        WHEN 6  => cout <= "0100";
        WHEN 7  => cout <= "0010";
      END CASE;
    WHEN "10" =>                -- HALF STEP
      CASE count IS
        WHEN 0  => cout <= "1010";
        WHEN 1  => cout <= "1000";
        WHEN 2  => cout <= "1001";
        WHEN 3  => cout <= "0001";
        WHEN 4  => cout <= "0101";
        WHEN 5  => cout <= "0100";
        WHEN 6  => cout <= "0110";
        WHEN 7  => cout <= "0010";
      END CASE;
    WHEN "11" =>                cout <= cin;--Direct Drive
  END CASE;
  END PROCESS;
END vhdl;;
```

four possible states and we want to do something different for each state, another CASE construct works best. In other words, we have chosen to use a CASE structure to select the mode and a CASE structure within each mode to select the proper output. Using one construct inside another is known as **nesting**. The use of indentation is very important to show the structure and logic of the code, especially when nesting is used.

The simulations of Figure 10-10 verify that the circuit seems to be working properly. Figure 10-10(a) shows each state decoding in mode 0 (full-step) and completing the cycle in both directions. Notice that after the mode ( $m$ ) changes to  $01_2$ , the output ( $cout$ ) is decoded as the wave-drive sequence. Figure 10-10(b) shows the wave-drive (mode 1) sequence in both directions and then changes the mode to  $10_2$ , resulting in the half-step sequence being decoded from the MOD-8 counter. Finally, Figure 10-10(c) shows the half-step mode cycling up and starting back down. It then switches to mode 3 (direct-drive) at 7.5 ms, showing that the data on  $cin$  is transferred asynchronously to the outputs. Notice that the values chosen for  $cin$  ensure that each bit can go HIGH and LOW.

Final integration and testing should involve more than just simulation. A real stepper motor and current driver should be connected to the circuit and tested. In this case, the step rate that the simulation used would probably be faster than the actual stepper motor could handle and would need to be slowed down for a real hardware functional test.

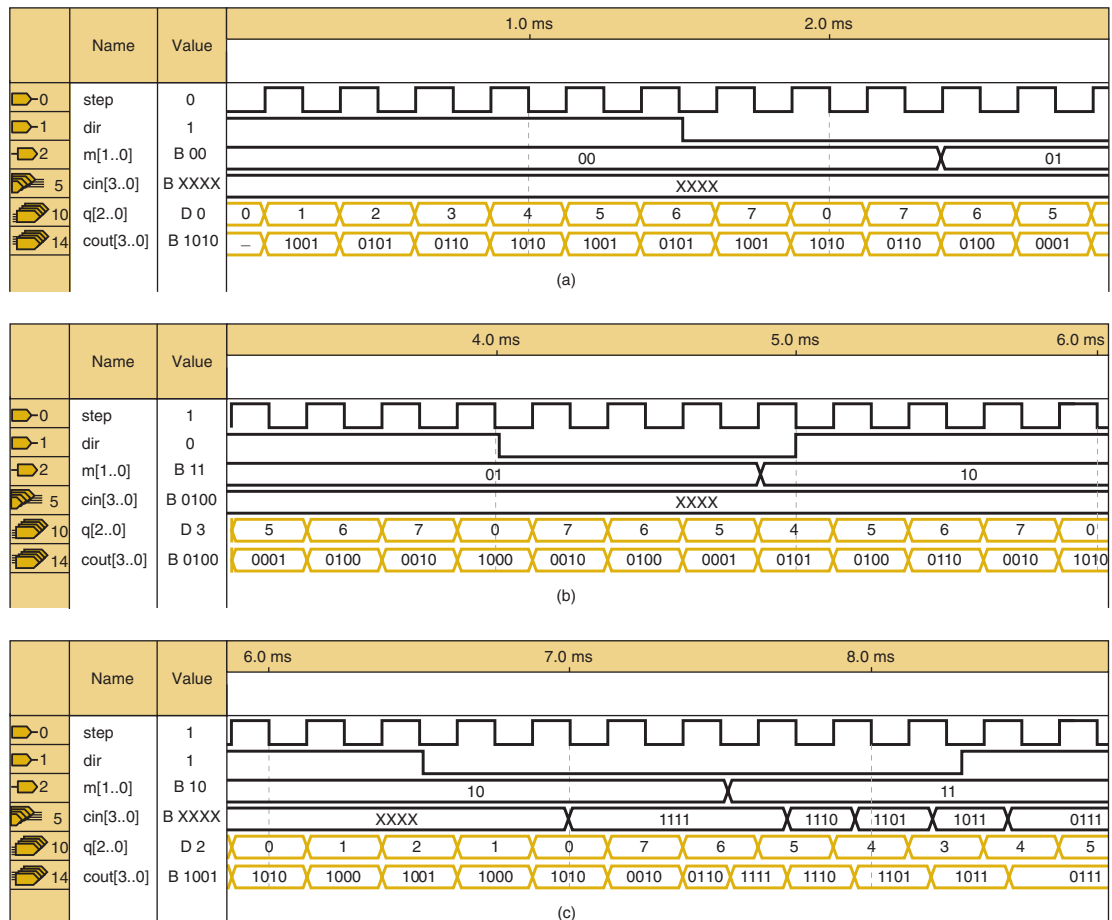


FIGURE 10-10 Simulation testing of the complete stepper driver.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What are the four modes of operation for this stepper motor driver?
2. What are the inputs for the direct-drive mode?
3. What are the inputs for the wave-drive mode?
4. How many states are in the half-step sequence?

## 10-3 KEYPAD ENCODER PROJECT

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the requirements of a keypad encoder project.
- Define each input and output.
- Describe the logical operation of the system using HDL.

Another important skill that we are trying to reinforce is circuit analysis. That may sound like something out of an analog textbook, but we really need to be able to analyze and understand how existing digital circuits operate. In this section, we present a circuit and analyze how it operates. Then we use the skills we have acquired to redesign this circuit and write the code for it in HDL.

### Problem Analysis

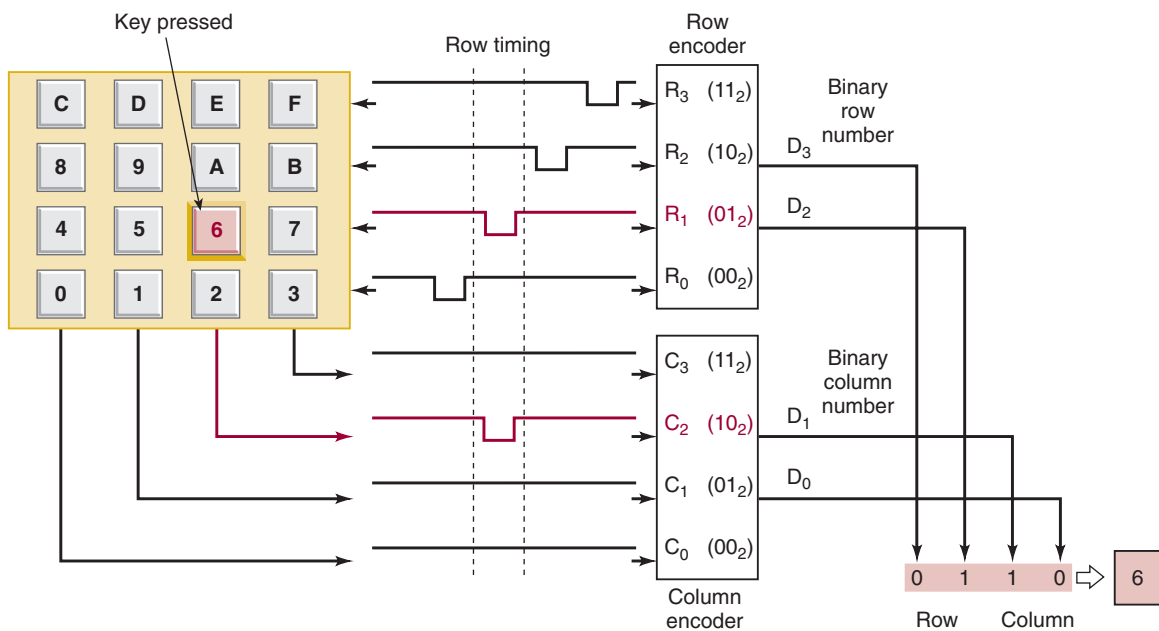
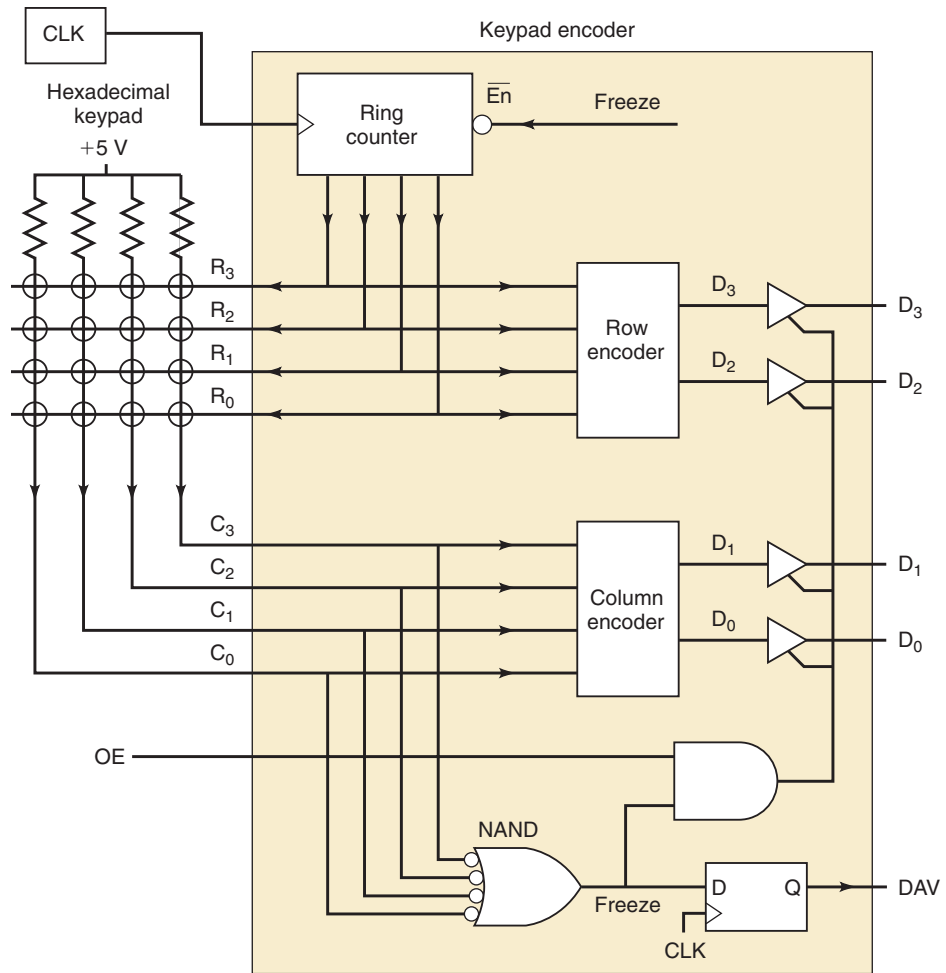
To reinforce the encoding concepts of Chapter 9, we present a very useful digital circuit that encodes a hexadecimal (16-key) keypad into a four-bit binary output. Encoders such as this generally have a strobe output that indicates when someone presses and releases a key. Because keypads are often interfaced to a microcomputer's bus system, the encoded outputs should have tristate enables. Figure 10-11 shows the block diagram of the keypad encoder.

The priority encoder method shown in Chapter 9, Figure 9-15, is effective for small keypads. However, large keyboards such as those found on personal computers must use a different technique. In these keyboards, each key is not an independent switch to  $V_{CC}$  or ground. Instead, each key switch is used to connect a row to a column in the keyboard matrix. When keys are not pressed, there are no connections between the rows and columns. The trick of knowing which key is pressed is accomplished by activating (pulling LOW) one row at a time and then checking to see if any of the columns have gone LOW. If one of the columns has a LOW on it, then the key being pressed is at the intersection of the activated row and the column that is currently LOW. If no columns are LOW, we know that no keys in the activated row are being pressed and we can check the next row by pulling it LOW. Sequentially activating rows is called *scanning* the keyboard. The advantage of this method is the reduction in connections to the keypad. In this case, 16 keys can be encoded using eight inputs/outputs.

Each key represents a unique combination of a row number and a column number. By strategically numbering the rows and columns, we can combine the binary row and column numbers to create the binary value of the hexadecimal keys as shown in Figure 10-12. In this figure, row 1 ( $01_2$ ) is pulled LOW and the data on the column encoder is  $10_2$  so the button at



**FIGURE 10-11** Keypad encoder block diagram.



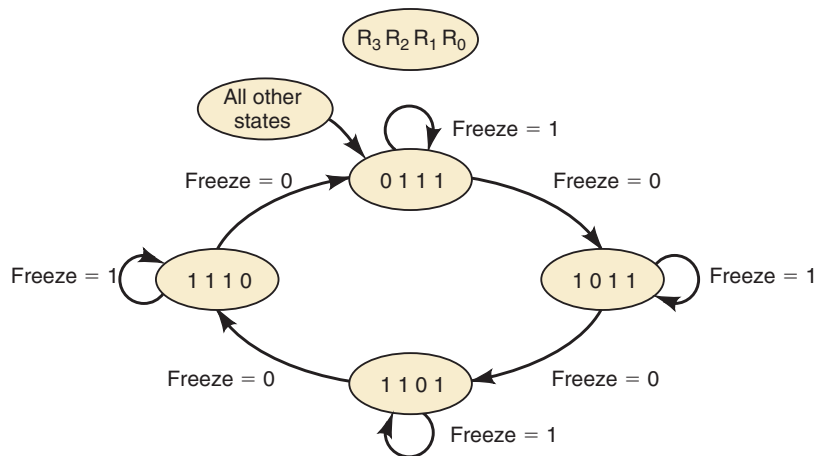
**FIGURE 10-12** Encoder operation when pressing the “6” key.

row 1, column 2 is evidently pressed. The NAND gate in Figure 10-11 is used to determine if any column is LOW, indicating that a key is pressed in the currently active row. The output of this gate is named *Freeze* because when a key is pressed, we want to freeze the ring counter and quit scanning until the key is released. As the encoders go through their propagation delay and the tristate buffers become enabled, the data outputs are in a transient state. On the next rising edge of the clock, the D flip-flop will transfer a HIGH from *Freeze* to the *DAV* output, indicating that a key is being pressed and the valid data is available.

A shift register counter (ring counter), as we studied in Chapter 7, is used to generate the sequential scan of the four rows. The count sequence uses four states, each state having a different bit pulled LOW. When a key press is detected, the ring counter must hold in its current state (freeze) until the key is released. Figure 10-13 shows the state transition diagram. Each state of this counter must be encoded to generate a two-bit binary row number. Each column value must also be encoded to generate a two-bit binary column number. The system will require the following inputs and outputs.

4	Row drive outputs	$R_0-R_3$
4	Column read inputs	$C_0-C_3$
4	Encoded data outputs	$D_0-D_3$
1	Data available strobe output	<i>DAV</i>
1	Tristate enable input	<i>OE</i>
1	Clock input	<i>CLK</i>

**FIGURE 10-13** Row drive ring counter state diagram.



### Strategic Planning/Problem Decomposition

This circuit is already structured so that we can easily write pieces of HDL code to emulate each section of the system. The major blocks are as follows:

- A ring counter with active-LOW outputs.
- Two encoders for the row and column numbers.
- Key-press detection and tristate enable circuits.

Because these circuits have been explored in previous chapters, we will not show the development and testing of each block here. The solutions that follow jump directly to the integration and testing phase of the project.

## AHDL SOLUTION

The inputs and outputs (see Figure 10-14 ) are defined on lines 3–8 and follow the description obtained from analyzing the schematic. The VARIABLE section defines several features of this encoder circuit. The *freeze* bit detects

```

1  SUBDESIGN fig10_14
2  (
3      clk           :INPUT;
4      col[3..0]    :INPUT;
5      oe           :INPUT;    --tristate output enable
6      row[3..0]    :OUTPUT;
7      d[3..0]     :OUTPUT;
8      dav         :OUTPUT;    --data available
9  )
10 VARIABLE
11 freeze         :NODE;
12 data[3..0]    :NODE;
13 ts[3..0]     :TRI;
14 data_avail    :DFF;
15 ring: MACHINE OF BITS (row[3..0])
16 WITH STATES (s1 = B"1110", s2 = B"1101", s3 = B"1011", s4 = B"0111",
17             % s = ring states %
18             f1 = B"0001", f2 = B"0010", f3 = B"0011", f4 = B"0100",
19             f5 = B"0101", f6 = B"0110", f7 = B"1000", f8 = B"1001",
20             f9 = B"1010", fa = B"1100", fb = B"1111", fc = B"0000");
21             % f = unused states --> self-correcting design %
22 BEGIN
23     ring.CLK = clk;
24     ring.ENA = !freeze;
25     data_avail.CLK = clk;
26     data_avail.D = freeze;
27     dav = data_avail.Q;
28     ts[].OE = oe & freeze;
29     ts[].IN = data[];
30     d[] = ts[].OUT;
31
32     CASE ring IS
33         WHEN s1 =>   ring = s2;   data[3..2] = B"00";
34         WHEN s2 =>   ring = s3;   data[3..2] = B"01";
35         WHEN s3 =>   ring = s4;   data[3..2] = B"10";
36         WHEN s4 =>   ring = s1;   data[3..2] = B"11";
37         WHEN OTHERS => ring = s1;
38     END CASE;
39
40     CASE col[] IS
41         WHEN B"1110" =>   data[1..0] = B"00";   freeze = VCC;
42         WHEN B"1101" =>   data[1..0] = B"01";   freeze = VCC;
43         WHEN B"1011" =>   data[1..0] = B"10";   freeze = VCC;
44         WHEN B"0111" =>   data[1..0] = B"11";   freeze = VCC;
45         WHEN OTHERS =>   data[1..0] = B"00";   freeze = GND;
46     END CASE;
47 END;
```

FIGURE 10-14 AHDL scanning keypad encoder.

when a key is pressed. The data node is used to combine the row and column encoder data. The *ts* bit array (line 13) represents a tristate buffer, as we studied in Chapter 9. Recall that each bit of this buffer has an input, (*ts*[*j*].*IN*), an output (*ts*.*OUT*), and an output enable (*ts*[*j*].*OE*). The *data\_avail* bit (line 14) represents a D flip-flop with inputs *data\_avail*.*CLK*, *data\_avail*.*D*, and output *data\_avail*.*Q*.

Lines 15–20 demonstrate a powerful feature of AHDL that allows us to define a state machine, with each state made up of the bit pattern we need. On line 15, the name *ring* was given to this state machine because it acts like a ring counter. The bits that make up this ring counter machine are the four row bits that were defined on line 6. These states are labeled *s1–s4* and have their bit patterns assigned to them so that one bit of the four is LOW for each state, like an active-LOW ring counter. The other twelve states are specified by an arbitrary label that starts with *f* to indicate they are not valid states. Lines 23 through 30 essentially connect all the components as shown in the circuit drawing of Figure 10-11. Both the ring count sequence and the encoding of the row value are described on lines 32–38. For each PRESENT state value of *ring*, the NEXT state is defined as well as the proper output of the row encoder (*data*[3..2]). Line 37 ensures that this counter will self-start by sending it to *s1* from any state other than *s1–s4*. The encoding of the column value is described on lines 40–46. Notice that the generation of the *freeze* signal in this design does not follow the diagram of Figure 10-11 exactly. In this design, rather than NANDing the columns, the CASE structure activates *freeze* only when one (and only one) column is LOW. Thus, if multiple keys in the same row were pressed, the encoder would not recognize any as a valid key press and would not activate *dav*.

## VHDL SOLUTION

Compare the VHDL description in Figure 10-15 with the circuit drawing of Figure 10-11. The inputs and outputs are defined on lines 5–9 and follow the description obtained from analyzing the schematic. Two SIGNALs are defined on lines 13 and 14 for this design. The *freeze* bit detects when a key is pressed. The *data* signal is used to combine the row and column encoder data to make a four-bit value representing the key that was pressed. The ring counter is implemented using a PROCESS that responds to the *clk* input. Line 26 ensures that this counter will self-start by sending it to state “1110” from any state other than those in the *ring* sequence. Notice that on line 20, the status of *freeze* is checked before a CASE is used to assign a NEXT state to *ring*. This is the way the count enable is implemented in this design. On line 29, the data available output (*dav*) is updated synchronously with the value of *freeze*. It is synchronous because it is within the IF structure (lines 19–30) that detects the active clock edge. The remaining statements (lines 31–52) do not depend on the active clock edge but describe what the circuit will do on either edge of the clock.

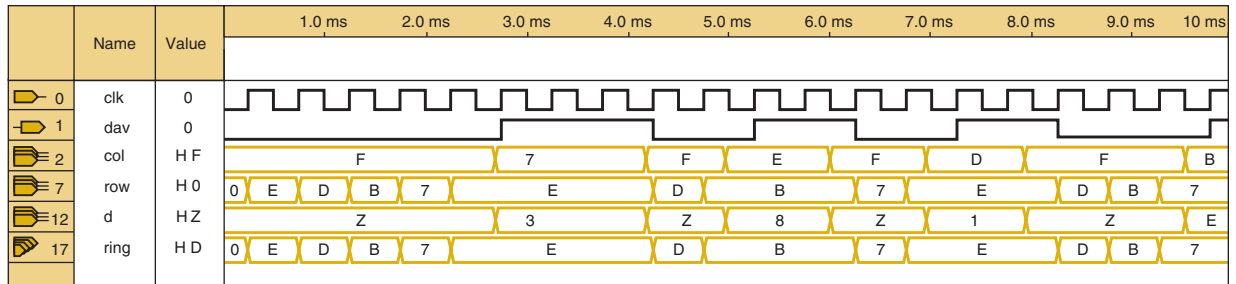
The encoding of the row value is described on lines 33–39. For each PRESENT state value of *ring*, the output of the row encoder *data*(3 DOWNTO 2) is defined. The encoding of the column value is described on lines 41–47. Notice that the generation of the *freeze* signal in this design does not follow the diagram of Figure 10-11 exactly. In this design, rather than NANDing the columns, the CASE structure activates *freeze* only when one (and only one) column is LOW. Thus, if multiple keys in the same row were pressed, the encoder would not recognize any as a valid key press and would not activate *dav*.

**FIGURE 10-15**  
VHDL scanning  
keypad encoder.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY fig10_15 IS
5  PORT (   clk           :IN STD_LOGIC;
6          col           :IN STD_LOGIC_VECTOR (3 DOWNTO 0);
7          row           :OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
8          d             :OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
9          dav           :OUT STD_LOGIC
10         );
11
12  ARCHITECTURE vhdl OF fig10_15 IS
13  SIGNAL freeze        :STD_LOGIC;
14  SIGNAL data          :STD_LOGIC_VECTOR (3 DOWNTO 0);
15  BEGIN
16  PROCESS (clk)
17  VARIABLE ring        :STD_LOGIC_VECTOR (3 DOWNTO 0);
18  BEGIN
19  IF (clk'EVENT AND clk = '1') THEN
20  IF freeze = '0' THEN
21  CASE ring IS
22  WHEN "1110" => ring := "1101";
23  WHEN "1101" => ring := "1011";
24  WHEN "1011" => ring := "0111";
25  WHEN "0111" => ring := "1110";
26  WHEN OTHERS => ring := "1110";
27  END CASE;
28  END IF;
29  dav <= freeze;
30  END IF;
31  row <= ring;
32
33  CASE ring IS
34  WHEN "1110" => data(3 DOWNTO 2) <= "00";
35  WHEN "1101" => data(3 DOWNTO 2) <= "01";
36  WHEN "1011" => data(3 DOWNTO 2) <= "10";
37  WHEN "0111" => data(3 DOWNTO 2) <= "11";
38  WHEN OTHERS => data(3 DOWNTO 2) <= "00";
39  END CASE;
40
41  CASE col IS
42  WHEN "1110" => data(1 DOWNTO 0) <= "00";   freeze <= '1';
43  WHEN "1101" => data(1 DOWNTO 0) <= "01";   freeze <= '1';
44  WHEN "1011" => data(1 DOWNTO 0) <= "10";   freeze <= '1';
45  WHEN "0111" => data(1 DOWNTO 0) <= "11";   freeze <= '1';
46  WHEN OTHERS => data(1 DOWNTO 0) <= "00";   freeze <= '0';
47  END CASE;
48
49  IF freeze = '1' THEN d <= data;
50  ELSE                 d <= "ZZZZ";
51  END IF;
52  END PROCESS;
53  END vhdl;

```



**FIGURE 10-16** Simulation of the scanning keypad encoder.

The simulation of the project is shown in Figure 10-16. The column values (*col*) are entered by the designer as a test input that simulates the value being read from the columns of the keypad as the rows are being scanned. As long as all columns are HIGH (i.e., the hex value F is on *col*), the *ring* counter is enabled and counting, *dav* is LOW, and the *d* outputs are in the Hi-Z state. Just before the 3.0-ms mark, a 7 is simulated as a *col* input, which means that one of the columns went LOW. This simulates a key being detected in the most significant column (C3) of the keypad matrix. Notice that as a result of the column going LOW, on the next active (rising) clock edge, the *dav* line goes HIGH and the ring counter does not change state. It is disabled from going to its NEXT state as long as the key is pressed. At this point, the *row* value is E hex (1110<sub>2</sub>), which means that the least significant row (R0) is being pulled LOW by the ring counter. The row encoder translates this into the binary row number (00). The key located at the intersection of the least significant row (00<sub>2</sub>) and the most significant column (11<sub>2</sub>) is the 3 key (see Figure 10-12). At this point, the *d* outputs hold the encoded key value of 3 (0011<sub>2</sub>). Just after the 4-ms mark, the simulation imitates the release of the key by changing the column value back to F hex, which causes the *d* output to go into its Hi-Z state. On the next rising clock edge, the *dav* line goes LOW and the ring counter resumes its count sequence.

### OUTCOME ASSESSMENT QUESTIONS

1. How many rows on the scanned keyboard are activated at any point in time?
2. If two keys in the same column are pressed simultaneously, which key will be encoded?
3. What is the purpose of the D flip-flop on the DAV pin?
4. Will the time between the key being pressed and DAV going HIGH always be the same?
5. When are the data output pins in the Hi-Z state?

## 10-4 DIGITAL CLOCK PROJECT

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the requirements of a digital clock project.
- Decompose the project to simpler blocks.
- Define the input and output of each block.
- Describe the logical operation of the system using HDL.
- Integrate the blocks using graphic and HDL techniques.

One of the most common applications of counters is the digital clock—a time clock that displays the time of day in hours, minutes, and sometimes seconds.

In order to construct an accurate digital clock, a closely controlled basic clock frequency is required. For battery-operated digital clocks or watches, the basic frequency is normally obtained from a quartz-crystal oscillator. Digital clocks operated from the AC power line can use the 60-Hz power frequency as the basic clock frequency. In either case, the basic frequency must be divided to a frequency of 1 Hz or 1 pulse per second (pps). Figure 10-17 shows the basic block diagram for a digital clock operating from 60 Hz.

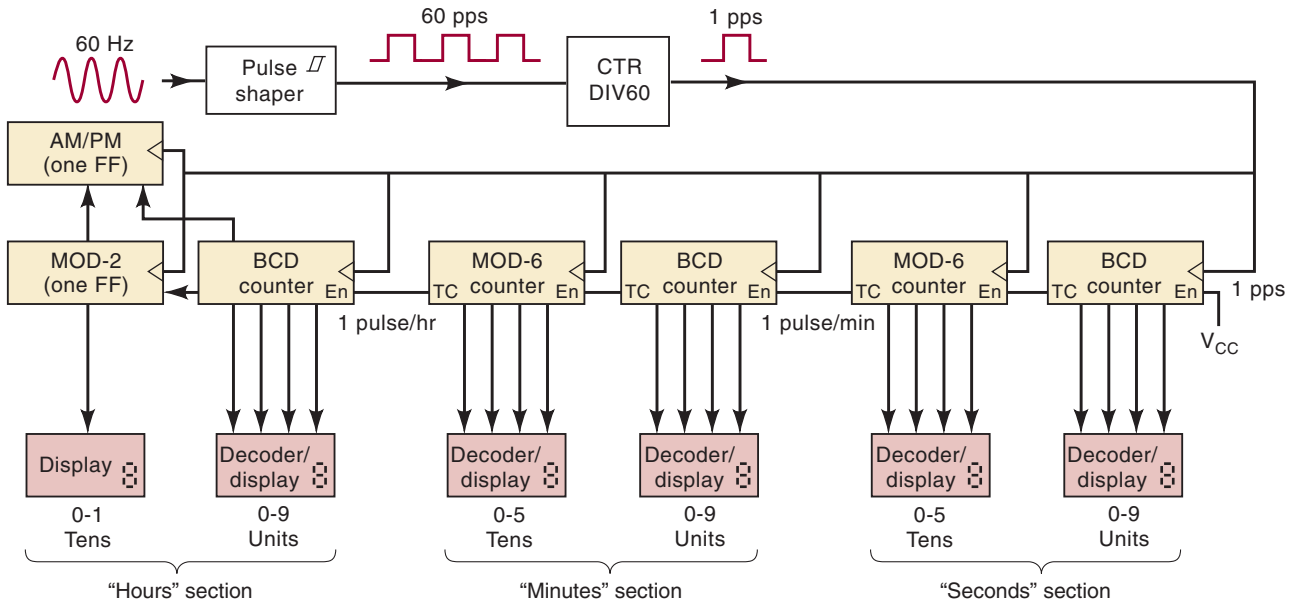


FIGURE 10-17 Block diagram for a digital clock.

The 60-Hz signal is sent through a Schmitt-trigger circuit to produce square pulses at the rate of 60 pps. This 60-pps waveform is fed into a MOD-60 counter that is used to divide the 60 pps down to 1 pps. The 1-pps signal is used as a synchronous clock for all of the counter stages, which are synchronously cascaded. The first stage is the SECONDS section, which is used to count and display seconds from 0 through 9. The BCD counter advances one count per second. When this stage reaches 9 seconds, the BCD counter activates its terminal count output ( $tc$ ), and on the next active clock edge, it recycles to 0. The BCD terminal count enables the MOD-6 counter and causes it to advance by one count at the same time that the BCD counter recycles. This process continues for 59 seconds, at which point the MOD-6 counter is at the 101 (5) count and the BCD counter is at 1001 (9) so that the display reads 59 s and  $tc$  of the MOD-6 is HIGH. The next pulse recycles the BCD counter and the MOD-6 counter to zero (remember, the MOD-6 counts from 0 through 5).

The  $tc$  output of the MOD-6 counter in the SECONDS section has a frequency of 1 pulse per minute (i.e., the MOD-6 recycles every 60 s). This signal is fed to the MINUTES section, which counts and displays minutes from 0 through 59. The MINUTES section is identical to the SECONDS section and operates in exactly the same manner.

The  $tc$  output of the MOD-6 counter in the MINUTES section has a frequency of 1 pulse per hour (i.e., the MOD-6 recycles every 60 min). This signal is fed to the HOURS section, which counts and displays hours from 1 through 12. This HOURS section is different from the SECONDS and MINUTES sections because it never goes to the 0 state. The circuitry in this section is sufficiently unusual to warrant a closer investigation.

Figure 10-18 shows the detailed circuitry contained in the HOURS section. It includes a BCD counter to count units of hours and a single FF (MOD-2)

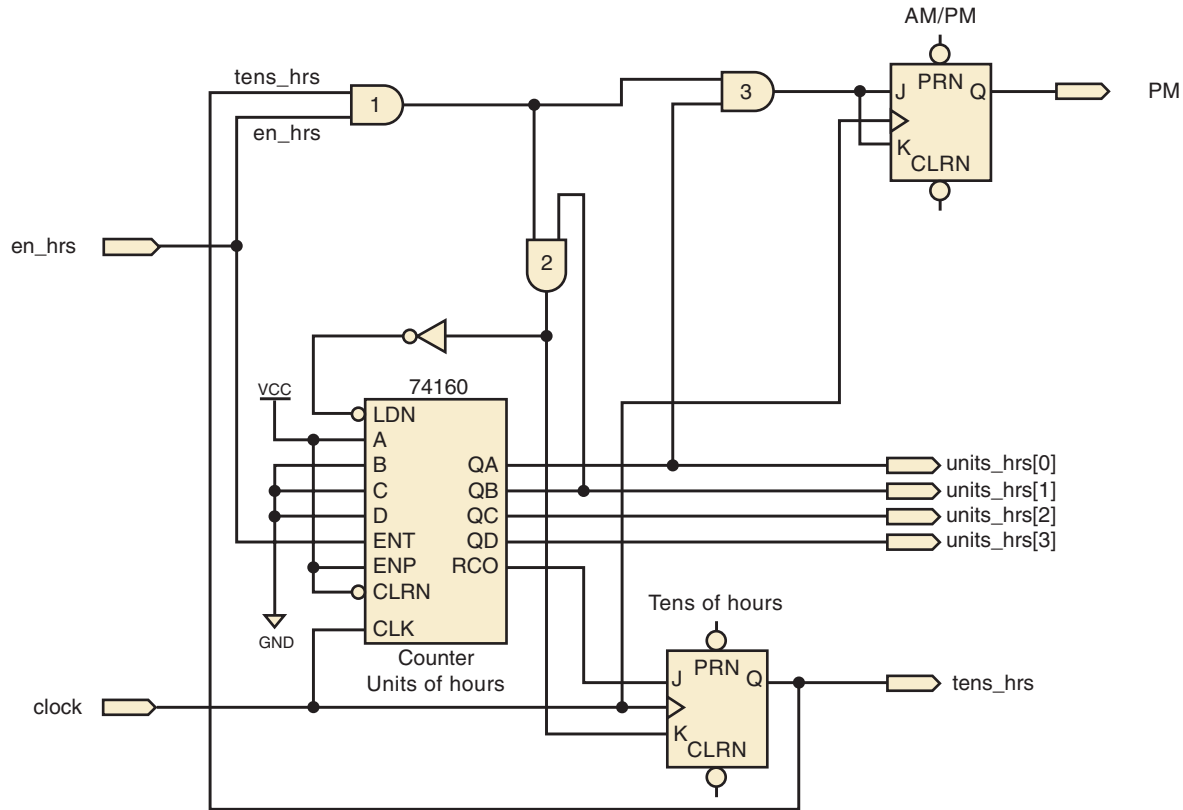


FIGURE 10-18 Detailed circuitry for the HOURS section.

to count tens of hours. The BCD counter is a 74160, which has two active-HIGH inputs, ENT and ENP, that are ANDed together internally to enable the count. The ENT input also enables the active-HIGH ripple carry out (RCO) that detects the BCD terminal count of 9. The ENT input and RCO output can therefore be used for synchronous counter cascading. The ENP input is tied HIGH so that the counter will increment whenever ENT is HIGH.

The hours counter is enabled by the minutes and seconds stages for only one clock pulse every hour. When this condition occurs, ENT is HIGH, which means that the minutes:seconds stages are at 59:59. For example, at 9:59:59, the tens of hours flip-flop holds a 0, the 74160 holds  $1001_2$  (9), and the RCO output is HIGH, putting the tens of hours flip-flop in the SET mode. The two display digits for the hours show 09. On the next rising clock edge, the BCD counter advances to its natural NEXT state of  $0000_2$ , RCO goes LOW, and the tens of hours flip-flop advances to 1 so that the hours display digits now show 10.

When it is 11:59:59, AND gate 1 detects that the tens of hours is 1 and the enable input is active (previous stages are at 59:59). AND gate 3 combines the conditions of AND gate 1 and the condition that the BCD counter is in the state  $0001_2$ . The output of AND gate 3 will be HIGH only at 11:59:59 in the hours count sequence. On the next clock pulse, the AM/PM flip-flop toggles, indicating noon (HIGH) or midnight (LOW). At the same time, the BCD counter advances to 2 and the minutes:seconds stages roll over to 00:00, resulting in a BCD display of 12:00:00. At 12:59:59, AND gate 1 detects that the tens digit is 1 and it is time to advance the hours. AND gate 2 detects that the BCD counter is at 2. The output of AND gate 2 prepares to do two tasks on the next clock edge: reset the tens of hours flip-flop, and load the 74160 counter with the value  $0001_2$ . After the next clock pulse, it is 01:00:00 o'clock.



The operation of counter circuits should make sense now, and you should have a good grasp on how you can connect MSI chips to make this digital clock. Notice that it is really made up of several small and relatively simple circuits that are strategically interconnected to make the clock. Recall that in Chapter 4, we mentioned briefly the concept of modular, hierarchical design and development of digital systems. Now it is time to apply these principles to a project that is within your scope of understanding using the Quartus II development system from Altera. You must understand the operation of the circuits that have just been described before proceeding with the design of this clock using HDL. Take some time to review this material.

### Top-Down Hierarchical Design

When problems are big and complex, it is necessary to go through multiple levels of problem decomposition. These multiple levels are often referred to as a hierarchy. The strategic aspect of how the problem is decomposed becomes even more important as the complexity increases. At each level, the interconnections between blocks should be as simple as possible with a clear vision of each block's function, a plan for testing it, and a watchful eye for common elements that can be perfected once and reused in multiple places. The alarm clock that we just discussed was already decomposed into small blocks that were analyzed from the "bottom up." This section will take us through the design process that starts at the top.

Top-down design means that we want to start at the highest level of complexity in the hierarchy, or that the entire project is considered to exist in a closed, dark box with inputs and outputs. The details regarding what is in the box are not yet known. We can only say at this point how we want it to behave. The digital clock was chosen because everyone is familiar with the end result of the operation of this device. An important aspect of this stage of the design process is establishing the scope of the project. For example, this digital clock is not going to have a way to set the time, set an alarm time, shut off the alarm, snooze, or incorporate other features that you may find on the clock beside your bed. To add all these features now would only clutter the example with unnecessary complexity for our immediate purpose. We are also not going to include the signal conditioning that transforms a 60-Hz sine wave into a 60-pps digital waveform, or the decoder/display circuits. The project we are tackling has the following specifications:

Inputs: 60 pps CMOS compatible waveform (accuracy dependent on line frequency)

Outputs: BCD Hours: 1 bit TENS 4 bits UNITS  
 BCD Minutes: 3 bits TENS 4 bits UNITS  
 BCD Seconds: 3 bits TENS 4 bits UNITS  
 PM indicator

Minutes and Seconds sequence: BCD MOD 60  
 00–59 (decimal representation of BCD)

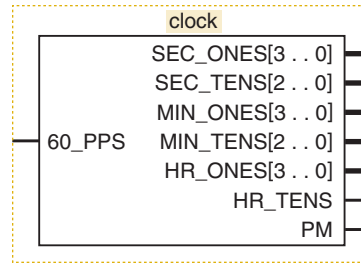
Hours sequence: BCD MOD 12  
 01–12 (decimal representation of BCD)

Overall range of display:  
 01:00:00–12:59:59

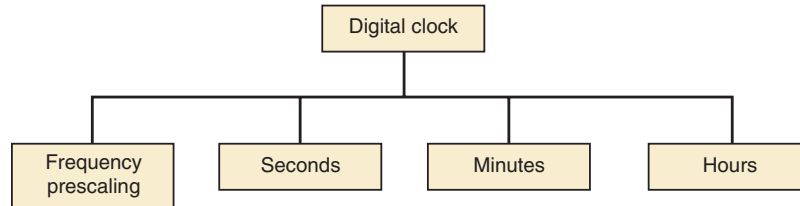
AM/PM indicator toggles at 12:00:00

A **hierarchy** is a group of objects arranged in rank order of magnitude, importance, or complexity. A block diagram of the overall project (highest level of the hierarchy) is shown in Figure 10-19. Notice that there are four bits for

**FIGURE 10-19** The top level block of the hierarchy.



**FIGURE 10-20** The four subsections in the second level of the hierarchy.



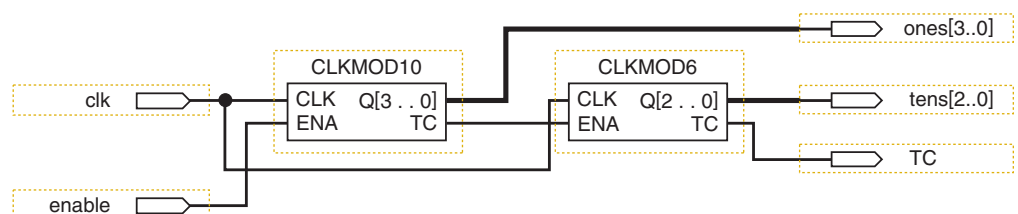
each of the BCD units outputs and only three bits for each of the minute and second BCD tens outputs. Because the most significant BCD digit for the tens place is 5 ( $101_2$ ), only three bits are needed. Notice also that the tens place for the hours (HR\_TENS) is only one bit. It will never have a value other than 0 or 1.

The next phase is to break this problem into more manageable sections. First, we need to take the 60-pps input and transform it into a 1-pps timing signal. A circuit that divides a reference frequency to a rate required by the system is called a **prescaler**. Next, it makes sense to have individual sections for a seconds counter, minutes counter, and hours counter. So far, the hierarchy diagram looks like Figure 10-20, which shows the project broken into four subsections.

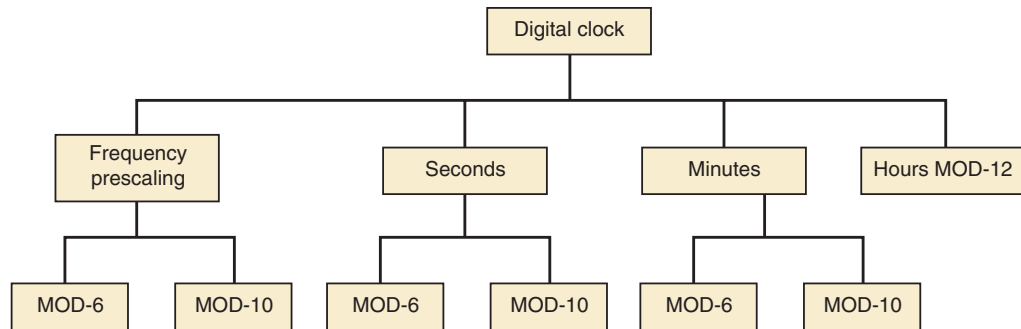
The entire purpose of the frequency prescaler section is to divide the 60-pps input to a frequency of one pulse every second. This requires a MOD-60 counter, and the sequence of the count does not really matter. In this example, the minutes and seconds sections both require MOD-60 counters that count from 00 to 59 in BCD. Looking for similarities like this is very important in the design process. In this case, we can use the exact same circuit design to implement the frequency prescaler, the minutes counter, and the seconds counters.

A MOD-60 BCD counter can be made quite easily from a MOD-10 (decade) counter cascaded to a MOD-6 BCD counter, as we saw in Figure 10-17. This means that inside each of these MOD-60 blocks, we would find a diagram similar to Figure 10-21. The hierarchy of the project now appears as shown in Figure 10-22.

The final design decision is whether or not to break down the MOD-12 section for hours into two stages, as shown in Figure 10-18. One option is to connect the macrofunctions of these standard parts from the HDL library, as we have discussed in previous chapters. Because this circuit is rather unusual, we have decided instead to describe the MOD-12 hours counter using a single HDL module. We will also describe the MOD-6 and MOD-10



**FIGURE 10-21** The blocks inside the MOD-60 section.



**FIGURE 10-22** The complete hierarchy of the clock project.

building blocks using HDL. The entire clock circuit can then be built using these three basic circuit descriptions. Of course, even these blocks can be broken down into smaller flip-flop blocks and designed using the schematic entry, but it will be much easier using HDL at this level.

### Building the Blocks from the Bottom Up

Each of the basic blocks are presented here in both AHDL and VHDL. We present the MOD-6 as a simple modification of the MOD-5 synchronous counter descriptions presented earlier in Chapter 7 (see Figures 7-43 and 7-44). Then we modify this code further to create the MOD-10 counter and finally design the MOD-12 hours counter. We construct the entire clock from these three basic blocks.

### MOD-6 COUNTER AHDL

The only additional features that this design needs that are not covered in Figure 7-43 are the count *enable* input and terminal count (*tc*) output shown in Figure 10-23. Notice that the extra input (*enable*, line 3) and output

```

1  SUBDESIGN fig10_23
2  (
3    clock, enable      :INPUT;      -- synch clock and enable.
4    q[2..0], tc       :OUTPUT;     -- 3-bit counter
5  )
6  VARIABLE
7    count[2..0]      :DFF;        -- declare a register of D flip-flops.
8
9  BEGIN
10   count[].clk = clock;          -- connect all clocks to synchronous source
11   IF enable THEN
12     IF count[].q < 5 THEN
13       count[].d = count[].q + 1; -- increment current value by one
14     ELSE count[].d = 0;         -- recycle, force unused states to 0
15     END IF;
16   ELSE count[].d = count[].q;   -- not enabled: hold at this count
17   END IF;
18   tc = enable & count[].q == 5; -- detect maximum count if enabled
19   q[] = count[].q;             -- connect register to outputs
20  END;
```

**FIGURE 10-23** The MOD-6 design in AHDL.

(*tc*, line 4) are included in the I/O definition. A new line (line 11) in the architecture description tests *enable* before deciding how to update the value of *count* (lines 12–15). If *enable* is LOW, the same value is held on *count* at every clock edge by the ELSE branch (line 16). Remember always to match an IF with an END IF, as we did on lines 15 and 17. Terminal count (*tc*, line 18) will be HIGH when it is *true* that *count* = 5 AND *enable* is active. Notice the use of double equal signs (==) to evaluate equality in AHDL.

## MOD-6 COUNTER VHDL

The only additional features that this design needs that are not covered in Figure 7-44 are the count *enable* input and terminal count (*tc*) output shown in Figure 10-24. Notice that the extra input (*enable*, line 2) and output (*tc*, line 4) are included in the I/O definition. A new line (line 15) in the architecture description tests *enable* before deciding how to update the value of *count* (lines 16–20). In the case that *enable* is LOW, the current value is held in the variable *count* and does not count up. Remember always to match an IF with an END IF, as we did on lines 20–22. The terminal count indicator (*tc*, lines 24 and 25) will be HIGH when it is *true* that *count* = 5 AND *enable* is active.

```

1  ENTITY fig10_24 IS
2  PORT( clock, enable      :IN BIT ;
3        q                 :OUT INTEGER RANGE 0 TO 5;
4        tc                :OUT BIT
5  );
6  END fig10_24;
7
8  ARCHITECTURE a OF fig10_24 IS
9  BEGIN
10     PROCESS (clock)                -- respond to clock
11     VARIABLE count :INTEGER RANGE 0 TO 5;
12
13     BEGIN
14         IF (clock = '1' AND clock'event) THEN
15             IF enable = '1' THEN    -- synchronous cascade input
16                 IF count < 5 THEN  -- < max (terminal) count?
17                     count := count + 1;
18                 ELSE
19                     count := 0;
20                 END IF;
21             END IF;
22         END IF;
23         IF (count = 5) AND (enable = '1') THEN -- synch cascade output
24             tc <= '1';              -- indicate terminal ct
25         ELSE tc <= '0';
26         END IF;
27         q <= count;                 -- update outputs
28     END PROCESS;
29 END a;
```

**FIGURE 10-24** The MOD-6 design in VHDL.

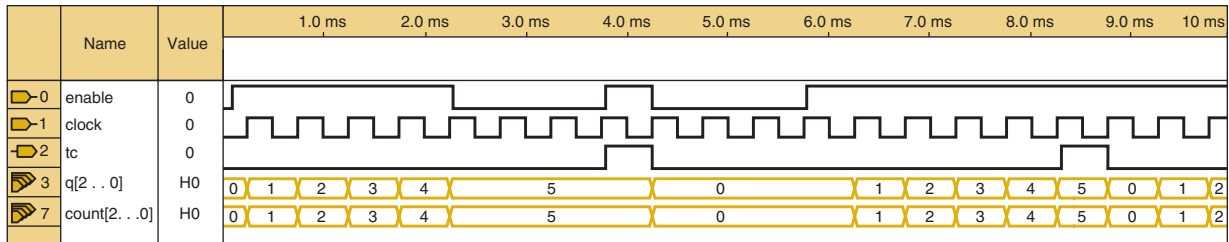


FIGURE 10-25 Simulation of the MOD-6 counter.

The simulation testing of the MOD-6 counter in Figure 10-25 verifies that it counts 0–5 and that it responds to the count enable input by ignoring the clock pulses and freezing the count whenever *enable* is LOW. It also generates the *tc* output when it is enabled at its maximum count of 5.

### MOD-10 COUNTER AHDL

The MOD-10 counter varies only slightly from the MOD-6 counter that was described in Figure 10-23. The only changes that are necessary involve changing the number of bits in the output port and the register (in the VARIABLE section) along with the maximum value that the counter should reach before rolling over. Figure 10-26 presents the MOD-10 design.

```

1  SUBDESIGN fig10_26
2  (
3      clock, enable      :INPUT;      -- synch clock and enable.
4      q[3..0], tc       :OUTPUT;     -- 4-bit Decade counter
5  )
6  VARIABLE
7      count[3..0] :DFF;              -- declare a register of D flip flops.
8
9  BEGIN
10     count[].clk = clock;            -- connect all clocks to synchronous source
11     IF enable THEN
12         IF count[].q < 9 THEN
13             count[].d = count[].q + 1;  -- increment current value by one
14             ELSE count[].d = 0;        -- recycle, force unused states to 0
15         END IF;
16     ELSE count[].d = count[].q;      -- not enabled: hold at this count
17     END IF;
18     tc = enable & count[].q == 9;    -- detect maximum count
19     q[] = count[].q;                -- connect register to outputs
20 END;
```

FIGURE 10-26 The MOD-10 design in AHDL.

### MOD-10 COUNTER VHDL

The MOD-10 counter varies only slightly from the MOD-6 counter that was described in Figure 10-24. The only changes that are necessary involve changing the number of bits in the output port and the variable *count* (using INTEGER RANGE) along with the maximum value that the counter should reach before rolling over. Figure 10-27 presents the MOD-10 design.

```

1  ENTITY fig10_27 IS
2  PORT( clock, enable      :IN BIT;
3         q                 :OUT INTEGER RANGE 0 TO 9;
4         tc                :OUT BIT
5         );
6  END fig10_27;
7
8  ARCHITECTURE a OF fig10_27 IS
9  BEGIN
10     PROCESS (clock)                -- respond to clock
11     VARIABLE count :INTEGER RANGE 0 TO 9;
12
13     BEGIN
14         IF (clock = '1' AND clock'event) THEN
15             IF enable = '1' THEN    -- synchronous cascade input
16                 IF count < 9 THEN  -- decade counter
17                     count := count + 1;
18                 ELSE
19                     count := 0;
20                 END IF;
21             END IF;
22         END IF;
23         IF (count = 9) AND (enable = '1') THEN -- synch cascade output
24             tc <= '1';
25         ELSE tc <= '0';
26         END IF;
27         q <= count;                -- update outputs
28     END PROCESS;
29 END a;

```

**FIGURE 10-27** The MOD-10 design in VHDL.

### MOD-12 Design

We have already decided that the hours counter is to be implemented as a single design file using HDL. It must be a MOD-12 BCD counter that follows the hours sequence of a clock (1–12) and provides the AM/PM indicator. Recall from the initial design step that the BCD outputs need to be a four-bit array for the low-order digit and a single bit for the high-order digit. To design this counter circuit, consider how it needs to operate. Its sequence is:

01 02 03 04 05 06 07 08 09 10 11 12 01 ...

By observing this sequence, we can conclude that there are four critical areas that define the operations needed to produce the proper NEXT state:

1. When the value is 01 through 08, increment the low digit and keep the high digit the same.
2. When the value is 09, reset the low digit to 0 and force the high digit to 1.
3. When the value is 10 or 11, increment the low digit and keep the high digit the same.
4. When the value is 12, reset the low digit to 1 and the high digit to 0.

Because these conditions need to evaluate a range of values, it is most appropriate to use an IF/ELSIF construct rather than a CASE construct. There is

also a need to identify when it is time to toggle the AM/PM indicator. This time occurs when the hour state is 11 and the enable is HIGH, which means that the lower-order counters are at their maximum (59:59).

### MOD-12 COUNTER IN AHDL

The AHDL counter needs a bank of four D flip-flops for the low-order BCD digit and only a single D flip-flop for the high-order BCD digit because its value will always be 0 or 1. A J-K flip-flop is also needed to keep track of A.M. and P.M. These primitives are declared on lines 7–9 of Figure 10-28. Also note that in this design, the same names are used for the output ports. This is a convenient feature of AHDL. When the enable input (*ena*) is active, the circuit evaluates the IF/ELSE statements of lines 16–28 and performs the proper operation on

```

1  SUBDESIGN fig10_28
2  (
3      clk, ena          :INPUT;
4      low[3..0], hi, pm :OUTPUT;
5  )
6  VARIABLE
7      low[3..0]      :DFF;
8      hi             :DFF;
9      am_pm         :JKFF;
10     time           :NODE;
11 BEGIN
12     low[].clk = clk;          -- synchronous clocking
13     hi.clk = clk;
14     am_pm.clk = clk;
15     IF ena THEN             -- use enable to count
16         IF low[].q < 9 & hi.q == 0 THEN
17             low[].d = low[].q + 1; --inc lo digit
18             hi.d = hi.q; -- hold hi digit
19         ELSIF low[].q == 9 THEN
20             low[].d = 0;
21             hi.d = VCC;
22         ELSIF hi.q == 1 & low[].q < 2 THEN
23             low[].d = low[].q + 1;
24             hi.d = hi.q;
25         ELSIF hi.q == 1 & low[].q == 2 THEN
26             low[].d = 1;
27             hi.d = GND;
28         END IF;
29     ELSE
30         low[].d = low[].q;
31         hi.d = hi.q;
32     END IF;
33     time = hi.q == 1 & low[3..0].q == 1 & ena; -- detect 11:59:59
34     am_pm.j = time; -- toggle am/pm at noon and midnight
35     am_pm.k = time;
36     pm = am_pm.q;
37 END;
```

FIGURE 10-28 The MOD-12 hours counter in AHDL.

the high and low nibble of the BCD number. Whenever the enable input is LOW, the value remains the same, as shown on lines 30 and 31. Line 33 detects when the count reaches 11 while the counter is enabled. This signal is applied to the *J* and *K* inputs of the *am\_pm* flip-flop to cause it to toggle at 11:59:59.

## MOD-12 COUNTER IN VHDL

The VHDL counter of Figure 10-29 needs a four-bit output for the low-order BCD digit and a single output bit for the high-order BCD digit because its value will always be 0 or 1. These outputs (lines 3 and 4) and also the

```

1  ENTITY fig10_29 IS
2  PORT( clk, ena      :IN BIT ;
3        low          :OUT INTEGER RANGE 0 TO 9;
4        hi           :OUT INTEGER RANGE 0 TO 1;
5        pm           :OUT BIT
6  );
7  END fig10_29;
8
9  ARCHITECTURE a OF fig10_29 IS
10 BEGIN
11   PROCESS (clk)
12     -- respond to clock
13     VARIABLE am_pm :BIT;
14     VARIABLE ones  :INTEGER RANGE 0 TO 9; -- 4-bit units signal
15     VARIABLE tens  :INTEGER RANGE 0 TO 1; -- 1-bit tens signal
16   BEGIN
17     IF (clk = '1' AND clk'EVENT) THEN
18       IF ena = '1' THEN
19         -- synchronous cascade input
20         IF (ones = 1) AND (tens = 1) THEN
21           -- at 11:59:59
22           am_pm := NOT am_pm;
23           -- toggle am/pm
24         END IF;
25       IF (ones < 9) AND (tens = 0) THEN
26         -- states 00-08
27         ones := ones + 1;
28         -- increment units
29       ELSIF ones = 9 THEN
30         -- state 09...set to 10:00
31         ones := 0;
32         -- units reset to zero
33         tens := 1;
34         -- tens bump up to 1
35       ELSIF (tens = 1) AND (ones < 2) THEN
36         -- states 10, 11
37         ones := ones + 1;
38         -- increment units
39       ELSIF (tens = 1) AND (ones = 2) THEN
40         -- state 12
41         ones := 1;
42         -- set to 01:00
43         tens := 0;
44       END IF;
45     END IF;
46
47     -----
48     -- This space is the alternate location for updating am/pm
49     -----
50   END IF;
51 END IF;
52 pm <= am_pm;
53 low <= ones;
54 hi <= tens;
55 -- update outputs
56 END PROCESS;
57 END a;
```

FIGURE 10-29 The MOD-12 hours counter in VHDL.



variables that will produce the outputs (lines 12 and 13) are declared as integers because this makes “counting” possible by simply adding 1 to the variable value. On each active edge of the clock, when the enable input is active, the circuit needs to decide what to do with the BCD units-of-hours counter, the single bit tens-of-hours flip-flop, and also the AM/PM flip-flop.

This example is an excellent opportunity to point out some of the advanced features of VHDL that allow the designer to describe precisely the operation of the final hardware circuit. In previous chapters, we discussed the issue of statements within a PROCESS being evaluated sequentially. Recall that statements outside the PROCESS are considered concurrent, and the order in which they are written in the design file has no effect on the operation of the final circuit. In this example, we must evaluate the current state to decide whether to toggle the AM/PM indicator and also advance the counter to the NEXT state. The issues involved include the following:

1. How do we “remember” the current count value in VHDL?
2. Do we evaluate the current count to see if it is 11 (to determine if we need to toggle the AM/PM flip-flop) and then increment to 12, or do we increment the counter’s state from 11 to 12 and then evaluate the count to see if it is 12 (to know we need to toggle the AM/PM flip flop)?

Regarding the first issue, there are two ways to remember the current state of a counter in VHDL. Both SIGNALS and VARIABLES hold their value until they are updated. Generally, SIGNALS are used to connect nodes in the circuit like wires, and VARIABLES are used like a register to store data between updates. Consequently, VARIABLES are generally used to implement counters. The major differences are that VARIABLES are local to the PROCESS in which they are declared and SIGNALS are global. Also, VARIABLES are considered to be updated immediately within a sequence of statements in a PROCESS, whereas SIGNALS referred to in a PROCESS are updated when the PROCESS suspends. In this example, we have chosen to use VARIABLES, which are local to the PROCESS that describes what should happen when the active clock edge occurs.

For the second issue, either of these strategies will work, but how do we describe them using VHDL? If we want the circuit to toggle A.M. and P.M. by detecting 11 prior to the counter updating (like a synchronous cascade), then the test must occur in the code before the VARIABLES are updated. This test is demonstrated in the design file of Figure 10-29 on lines 17–19. On the other hand, if we want the circuit to toggle A.M. and P.M. by detecting when the hour 12 has arrived after the clock edge (more like a ripple cascade), then the VARIABLES must be updated prior to testing for the value 12. To modify the design in Figure 10-29 to accomplish this task, the IF construct of lines 17–19 can be moved to the blank area of lines 31–33 and edited as shown in bold below:

```

31   IF (ones = 2) AND (tens = 1) THEN -- at 12:00:00
32       am_pm := NOT am_pm;           -- toggle am/pm
33   END IF;
```

The order of the statements and the value that is decoded make all the difference in how the circuit operates. On lines 36–38, the *am\_pm* VARIABLE is connected to the *pm* port, the units BCD digit is applied to the lower four bits of the output (*low*), and the tens digit (a single-bit variable) is applied to the most significant digit (*hi*) of the output port. Because all these VARIABLES are local, these statements must occur prior to END PROCESS on line 39.

After the design is compiled, it must be simulated to verify its operation, especially at the critical areas. Figure 10-30 shows an example of a simulation to test this counter. On the left side of the timing diagram, the counter is disabled and is holding the hour 11 because the *hi* digit is at 1 and the *low[]* digit is at 1. On the rising edge of the clock, after the enable goes HIGH, the hour goes from 11 to 12 and causes the PM indicator to go HIGH, which means it is noon. The next active edge causes the count to roll over from 12 to 01. On the right half of the timing, the same sequence is simulated, showing that there would actually be many clock pulses between the times the hour increments. On the clock cycle before it must increment, the enable is driven HIGH by the terminal count of the previous stage.

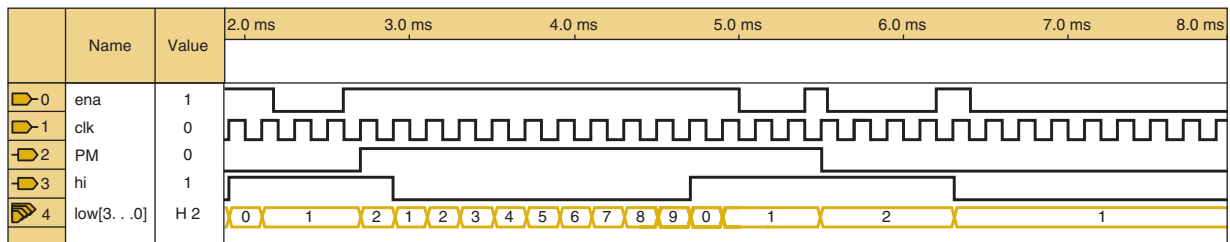


FIGURE 10-30 Simulation of the MOD-12 hours counter.

## Combining Blocks Graphically

The building blocks of the project have been defined, created, and individually simulated to verify that they work correctly. Now it is time to combine the blocks to make sections and to combine the sections to make the final product. Altera's software offers several ways to accomplish the integration of all the pieces of a project. In Chapter 4, we mentioned that all different types of design files (AHDL, VHDL, VERILOG, Schematic) can be combined graphically. This technique is made possible by a feature that allows us to create a "symbol" to represent a particular design file. For example, the MOD-6 counter design file that was written in the VHDL design file fig10\_24 can be represented in the software as the circuit block, as shown in Figure 10-31(a). The Quartus II software creates this symbol at the click of a button. From that point, it will recognize the symbol as operating according to the design specified in the HDL code. The symbol of Figure 10-31(b) was created from the AHDL file for the MOD-10 counter of Figure 10-26, and the symbol of Figure 10-31(c) was created from the VHDL file for the MOD-12 counter of Figure 10-29. (The reason these blocks are named by figure number is simply to make it easier to locate the design files on the web site. In a design environment [rather than in a textbook], they should be named according to their purpose, with names like MOD6, MOD10, and CLOCK\_HOURS.)

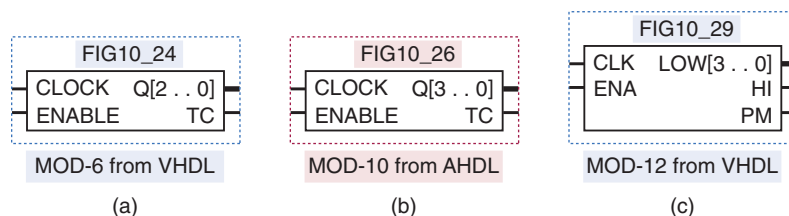
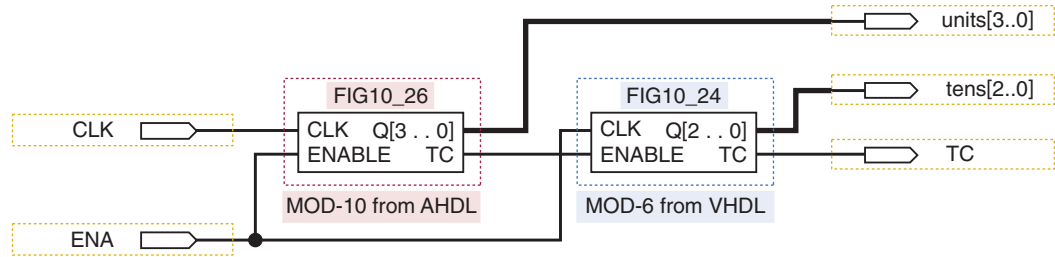
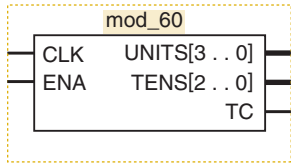


FIGURE 10-31 Graphic block symbols generated from HDL design files: (a) MOD-6 from VHDL; (b) MOD-10 from AHDL; (c) MOD-12 from VHDL.



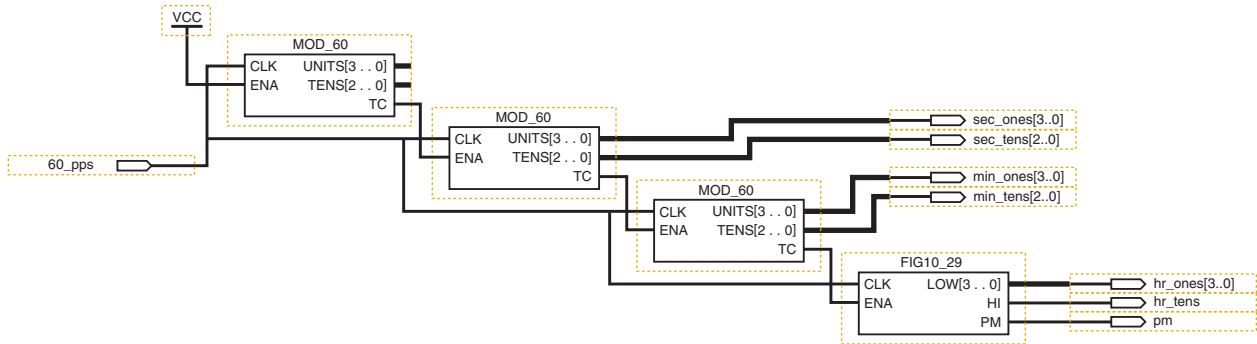
**FIGURE 10-32** Graphically combining HDL blocks to make a MOD-60.

Following the design hierarchy that we established, the next step is to combine the MOD-6 and MOD-10 counters to make a MOD-60 block. Quartus II software uses block design files (.bdf) to integrate the block symbols by drawing lines that connect the input ports, symbols, and output ports. The result is shown in Figure 10-32, which represents a BDF file in Quartus II. This block design file can be compiled and used to simulate the operation of the MOD-60 counter. When the design has been verified as working properly, the Quartus II system allows us to take this circuit and create a block symbol for it, as shown in Figure 10-33.



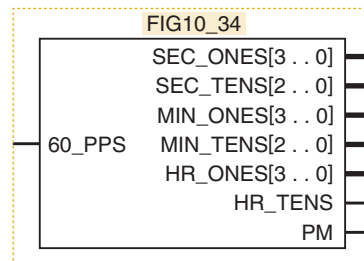
**FIGURE 10-33** The MOD-60 counter.

The MOD-60 symbol can be used repeatedly along with the MOD-12 symbol to create the system-level block symbol diagram shown in Figure 10-34. Even this system-level diagram can be represented by a block symbol for the entire project, as shown in Figure 10-35.



**FIGURE 10-34** The complete clock project connected using block symbols.

**FIGURE 10-35** The entire clock represented by one symbol.



### Combining Blocks Using Only HDL

The graphic approach works well as long as it is available and adequate for the purpose at hand. As we mentioned previously, HDL was developed to provide a convenient way to document complex systems and to store

the information in a more timeless and software-independent manner. It is reasonable to assume that with AHDL, the option of graphic integration of subdesigns will always be available with the tools from Altera; however, this assumption is not reasonable for users of VHDL. Many VHDL development systems do not offer any equivalent to the graphic block integration of Altera, which is why it is important to address the same concept of modular, hierarchical development and project integration using only text-based language tools. Our coverage of AHDL integration will not be as in-depth as our coverage of VHDL because the graphic method is generally preferred.

## AHDL MODULE INTEGRATION

Let's go back to the two AHDL files for the MOD-6 and MOD-10 counters. How do we combine these files into a MOD-60 counter using only text-based AHDL? The method is really very similar to that of graphic integration. Instead of creating a "symbol" representation of the MOD-6 and MOD-10 files, a new kind of file called an "INCLUDE" file is created. It contains all the important information about the AHDL file it represents. To describe a MOD-60 counter, a new TDF file, shown in Figure 10-36, is opened. The building block files are "included" at the top, as shown on lines 1 and 2. Next, the names that were used for the building blocks are used like library components or primitives to define the nature of a variable. On line 10, the variable *mod10* is now used to represent the MOD-10 counter in the other module (fig10\_26). *MOD10* now has all the attributes (inputs, outputs, functional operation) described in fig10\_26.tdf. Likewise, on line 11, the variable *mod6* is given the attributes of the MOD-6 counter of fig10\_23.tdf. Lines 13–19 accomplish the exact same task as drawing lines on the BDF file to connect the components to one another and to the input/output ports.

```

1  INCLUDE "fig10_26.inc";      -- mod-10 counter module
2  INCLUDE "fig10_23.inc";      -- mod-6 counter module
3
4  SUBDESIGN fig10_36
5  (
6      clk, ena                :INPUT;
7      ones[3..0], tens[2..0], tc :OUTPUT;
8  )
9  VARIABLE
10     mod10                    :fig10_26;      -- mod-10 for units
11     mod6                     :fig10_23;      -- mod-6 for tens
12 BEGIN
13     mod10.clock = clk;        -- synchronous clocking
14     mod6.clock = clk;
15     mod10.enable = ena;
16     mod6.enable = mod10.tc;   -- cascade
17     ones[3..0] = mod10.q[3..0]; -- 1s
18     tens[2..0] = mod6.q[2..0]; -- 10s
19     tc = mod6.tc;            -- Make terminal count at 59
20 END;
```

**FIGURE 10-36** The MOD-60 made from MOD-10 and MOD-6 in AHDL.

This file (FIG10\_36.TDF) can be translated into an “include” file (fig10\_36.inc) by the compiler and then used in another tdf file that describes the interconnection of major sections to make up the system. Each level of the hierarchy refers back to the constituent modules of the lower levels.

## VHDL MODULE INTEGRATION

Let’s go back to the two VHDL files for the MOD-6 and MOD-10 counters, which were shown in Figures 10-24 and 10-27, respectively. How do we combine these files into a MOD-60 counter using only text-based VHDL? The method is really very similar to that of graphic integration. Instead of creating a “symbol” representation of the MOD-6 and MOD-10 files, these design files are described as a COMPONENT, like we studied in Chapter 5. It contains all the important information about the VHDL file it represents. To describe a MOD-60 counter, a new VHDL file, shown in Figure 10-37, is opened. The building block files are described as “components,” as shown

```

1  ENTITY fig10_37 IS
2  PORT( clk, ena      :IN BIT ;
3         tens        :OUT INTEGER RANGE 0 TO 5;
4         ones        :OUT INTEGER RANGE 0 TO 9;
5         tc          :OUT BIT
6         );
7  END fig10_37;
8
9  ARCHITECTURE a OF fig10_37 IS
10     SIGNAL cascade_wire :BIT;
11     COMPONENT fig10_24   -- MOD-6 module
12     PORT( clock, enable  :IN BIT ;
13           q              :OUT INTEGER RANGE 0 TO 5;
14           tc             :OUT BIT);
15     END COMPONENT;
16     COMPONENT fig10_27   -- MOD-10 module
17     PORT( clock, enable  :IN BIT ;
18           q              :OUT INTEGER RANGE 0 TO 9;
19           tc             :OUT BIT);
20     END COMPONENT;
21 BEGIN
22     mod10:fig10_27
23     PORT MAP(  clock => clk,
24               enable => ena,
25               q  => ones,
26               tc => cascade_wire);
27
28     mod6:fig10_24
29     PORT MAP(  clock => clk,
30               enable => cascade_wire,
31               q  => tens,
32               tc => tc);
33 END a;
```

**FIGURE 10-37** The MOD-60 made from MOD-10 and MOD-6 in VHDL.

on lines 10–14 and lines 15–19 in the architecture description. Next, the names that were used for the building blocks (components) are used along with the PORT MAP keywords to describe the interconnection of these components. The information in the PORT MAP sections describes the exact same operations as drawing wires on a schematic diagram in a BDF file.

Finally, the VHDL file that represents the block at the top of the hierarchy is created using components from Figure 10-37 (MOD-60) and Figure 10-29 (MOD-12). This file is shown in Figure 10-38. Notice that the general form is as follows:

Define I/O: lines 1–7

Define signals: lines 10–11

Define components: lines 12–23

Instantiate components and connect them together: lines 27–52

```

1  ENTITY fig10_38 IS
2  PORT( pps_60                :IN BIT ;
3        hour_tens             :OUT INTEGER RANGE 0 TO 1;
4        hour_ones, min_ones, sec_ones :OUT INTEGER RANGE 0 TO 9;
5        min_tens, sec_tens      :OUT INTEGER RANGE 0 to 5;
6        pm                    :OUT BIT          );
7  END fig10_38;
8
9  ARCHITECTURE a OF fig10_38 IS
10 SIGNAL cascade_wire1, cascade_wire2, cascade_wire3 :BIT;
11 SIGNAL enabled                                     :BIT;
12 COMPONENT fig10_37 -- MOD-60
13 PORT( clk, ena      :IN BIT ;
14       tens         :OUT INTEGER RANGE 0 TO 5;
15       ones        :OUT INTEGER RANGE 0 TO 9;
16       tc          :OUT BIT          );
17 END COMPONENT;
18 COMPONENT fig10_29 -- MOD-12
19 PORT( clk, ena      :IN BIT ;
20       low          :OUT INTEGER RANGE 0 TO 9;
21       hi          :OUT INTEGER RANGE 0 TO 1;
22       pm          :OUT BIT          );
23 END COMPONENT;
24 BEGIN
25     enabled <= '1';
26
27     prescale:fig10_37 -- MOD-60 prescaler
28     PORT MAP(  clk => pps_60,
29              ena => enabled,
30              tc  => cascade_wire1);
31
32     second:fig10_37 -- MOD-60 seconds counter
33     PORT MAP(  clk => pps_60,
34              ena => cascade_wire1,
35              ones => sec_ones,
36              tens => sec_tens,
37              tc  => cascade_wire2);

```

**FIGURE 10-38** The complete clock in VHDL.

```

38
39     minute:fig10_37      -- MOD-60 minutes counter
40     PORT MAP(  clk  => pps_60,
41               ena  => cascade_wire2,
42               ones => min_ones,
43               tens => min_tens,
44               tc   => cascade_wire3);
45
46     hour:fig10_29      -- MOD-12 hours counter
47     PORT MAP(  clk  => pps_60,
48               ena  => cascade_wire3,
49               low  => hour_ones,
50               hi   => hour_tens,
51               pm   => pm);
52     END a;

```

FIGURE 10-38 Continued

### OUTCOME ASSESSMENT QUESTIONS

1. What is being defined at the top level of a hierarchical design?
2. Where does the design process start?
3. Where does the building process start?
4. At which stage(s) should simulation testing be done?

## 10-5 MICROWAVE OVEN PROJECT

### OUTCOMES

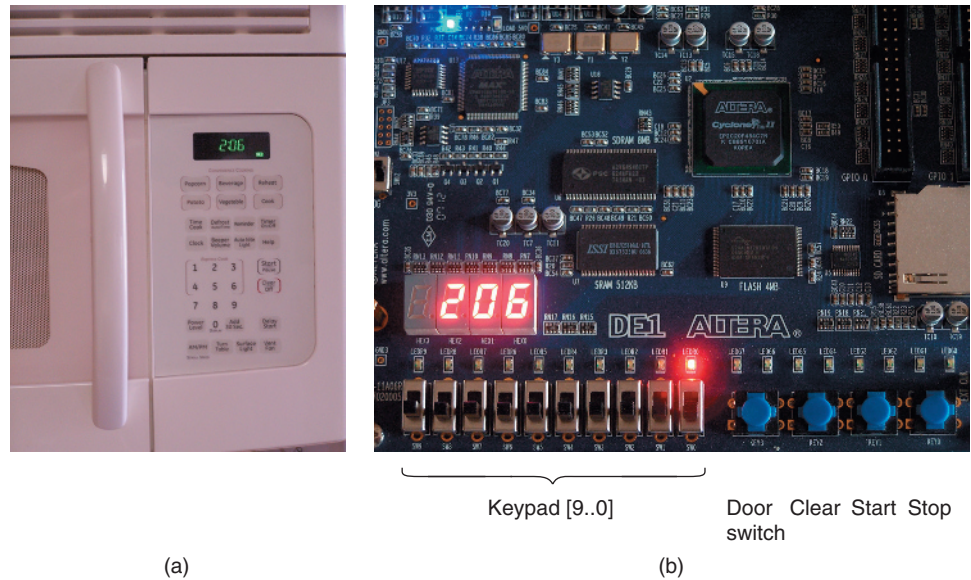
*Upon completion of this section, you will be able to:*

- Describe the requirements of a microwave oven system.
- Decompose the system into simpler blocks.
- Define each input and output for each block.
- Describe the logic of the system using HDL.

At this point we have discussed the primary building blocks of digital systems and looked at simple system examples that use a few of these blocks. In this section, we will cover a complete system that everyone is familiar with from a user's point of view: the microwave oven. This system contains many of the building blocks that make up all digital systems and demonstrates how they can be combined to control one of the most life-changing inventions of all time.

Microwave ovens simply use a radio frequency (rf) generator with enough power to excite the molecules in food and heat it up. The four basic components used in these appliances have not changed much since microwave cooking was invented in the 1960s: a high-voltage transformer, a diode, a capacitor, and a magnetron tube. The only important thing that you need to know about this circuit is that when you apply 120 VAC to the transformer, the oven cooks your food and when you disconnect the 120 VAC, the oven turns off. In other words, it can be controlled by a 1 or a 0. The circuits that control the microwave oven have changed over the years, from simple mechanical timers to sophisticated digital systems. The controller we will design allows the user

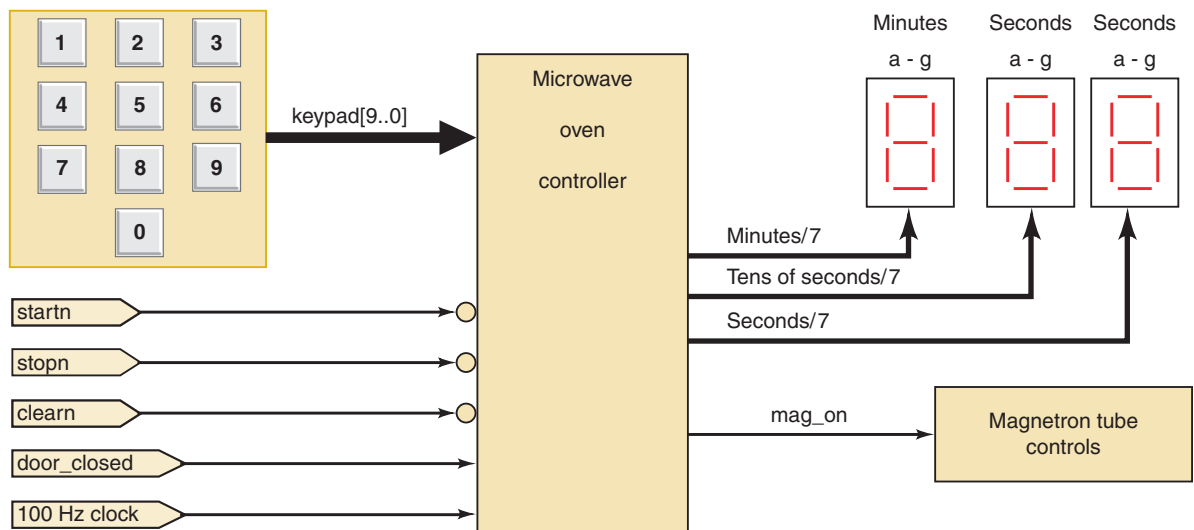
**FIGURE 10-39** (a) A typical microwave oven; (b) the microwave oven project on a DE1 board.



to enter the desired cook time in minutes and seconds and offers the normal basic user controls for a typical oven similar to the picture in Figure 10-39(a).

### Definition of the Project

Let's start at the top and define the system inputs and outputs for our microwave controller. It should be noted that the intent of this project is to implement it on the Altera DE1 (or DE2 or similar) development board as shown in Figure 10-39(b). Because the active logic levels for the system match the hardware resources (switches and displays) of the Altera boards, we have decided to use ten individual switches to represent the keypad inputs, similar to the encoder shown in Figure 9-15. An alternative approach would be to use an external matrix keypad as described in Section 10-3. This example opts to avoid using hardware external to the development board. The block diagram of Figure 10-40 defines the scope of the project along with the detailed specifications listed in Table 10-3.



**FIGURE 10-40** System block diagram for the microwave oven.



**TABLE 10-3** Microwave signal specifications.

Input Signal	Signal Specification
clock	100 Hz, 3.3 V standard logic levels
startn, stopn, clearn	Normally HIGH, active-LOW momentary push buttons, 3.3 V standard logic levels
door_closed	HIGH when door is closed, LOW when door is open
keypad (0–9)	10 individual keypad buttons: active-HIGH (slide switches used on DE1)
Output Signal	Signal Specification
mag_on	Active-HIGH output used to apply 120 VAC to magnetron circuit
min_segs (a–g)	Active-LOW outputs to high (minutes) display digit: segments a–g, respectively
sec_tens_segs (a–g)	Active-LOW outputs to mid (seconds tens) display digit: segments a–g, respectively
sec_ones_segs (a–g)	Active-LOW outputs to low (seconds ones) display digit: segments a–g, respectively

The system should function like a typical microwave oven. When the oven is not cooking food, you should be able to enter the desired cook time by pressing keys on the keypad. Each key that is pressed shows up on the right-most digit of the display, and the other digits shift to the left. When the start button is pressed, assuming the door is closed, the magnetron tube is activated and the digits count down in minutes and seconds. Leading zeros are blanked on the display. If the door is opened or the stop button is pressed, the time holds at its current value and the magnetron is turned off. Pressing clear at any time forces the count to 0. When the count reaches 0, the magnetron is turned off and the time reads 0. If a person enters an initial value for seconds that is greater than 59 (i.e., 60 to 99), the seconds counter should still count down from that value to 00.

### Strategic Planning/Problem Decomposition

The first strategic decision should be whether to use a microcontroller or a custom-designed digital circuit. For this application there is no reason that a microcontroller could not be used to control the oven. In fact, your microwave oven probably uses a microcontroller for this purpose. Recall that a microcontroller is a small computer system on a chip. It performs sequential instructions that the designer stored in its memory. Instructions such as checking each input, performing any calculations, and updating the outputs can be performed faster than a person's finger can press and release a button. Any microcontroller is fast enough to keep up with human motion, so the speed of custom-designed digital circuitry is not necessary for this application. However, this is a digital systems text and we have chosen to include as many of the building blocks of digital systems as possible in this solution. The microwave will be implemented as a digital circuit in a field programmable gate array (FPGA) rather than a microcontroller. This decision affects the way we plan the blocks of the project.

The features of the microwave oven allow us to easily identify major functional blocks of this system. There are many ways that any system can be broken down. For example, the designer must decide how many functional blocks are appropriate and how many levels of hierarchy are necessary, and also make some strategic decisions that will affect the degree of difficulty of solving each functional block. This project is decomposed

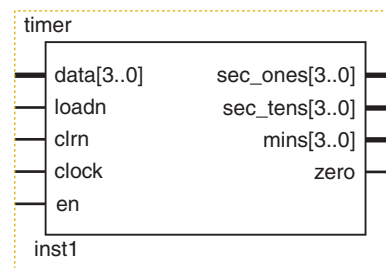
such that there are three levels of hierarchy and four functional blocks on level 2. The complexities of the blocks range from simple to difficult. When tasks are assigned to a project team, a manager is able to align the task to the experience/skill level of the team members.

Perhaps the most obvious functional block in the microwave oven system is the minutes/seconds timer, which is simply a circuit that counts down at one-second intervals. The requirements of this counter block are:

- It must count down and stop counting when it reaches zero.
- It must be loaded (minutes and seconds) one BCD digit at a time. Digits must shift left.
- It must be able to be cleared.
- It must be able to be disabled (hold the current count).

The second strategic decision must be made at this point. Should the design use a straight binary counter or a BCD counter for the oven's timer? A straight binary counter is easy to describe or build, but how can it be loaded with the proper binary number each time a key is pressed? Remember, each key entry simply produces a BCD digit. The circuit that could change BCD entries into a binary number and load this counter would require some sophisticated math operations. On the output side, the circuit block that translates this binary number (from the counter) to 7-segment BCD would also be very complex. Using a binary counter may not be a good idea. On the other hand, if we can make the counter operate as BCD stages that are cascaded together, then the counter block will be slightly more complex than a simple binary counter; but the loading of the data and displaying the data will be much simpler. Remember, every decision has consequences, so it is a very good investment to spend plenty of time thinking through the ramifications of your decisions. We have decided to use a cascaded BCD counter. Consequently, the required inputs are a single BCD digit (i.e., a four-bit data value), a *load* control line, a *clock*, an *enable*, and a *clear* line as shown in Figure 10-41. Notice from the labels that the *loadn* and *clearn* lines are active-LOW while the *enable* is active-HIGH. The outputs are three BCD digits and a signal line (zero) that indicates when the counter has reached 0.

**FIGURE 10-41** The minutes/seconds counter block of the microwave system.



Let's break the minutes/seconds counter block down further into functional modules, creating a third level of the hierarchy. Figure 10-42 shows graphically how three BCD counter stages can be cascaded to create this functionality. Each stage is a single-digit BCD down counter. However, since the seconds counter must count down in seconds from 59 to 00, the tens of seconds, counter (middle digit) must be a MOD 6 BCD counter. The other two stages, the units of seconds and minutes counters, are identical MOD 10 counters. We have now reduced the problem to simply creating a single-digit MOD-*N* BCD down counter with the following features: Each stage must have

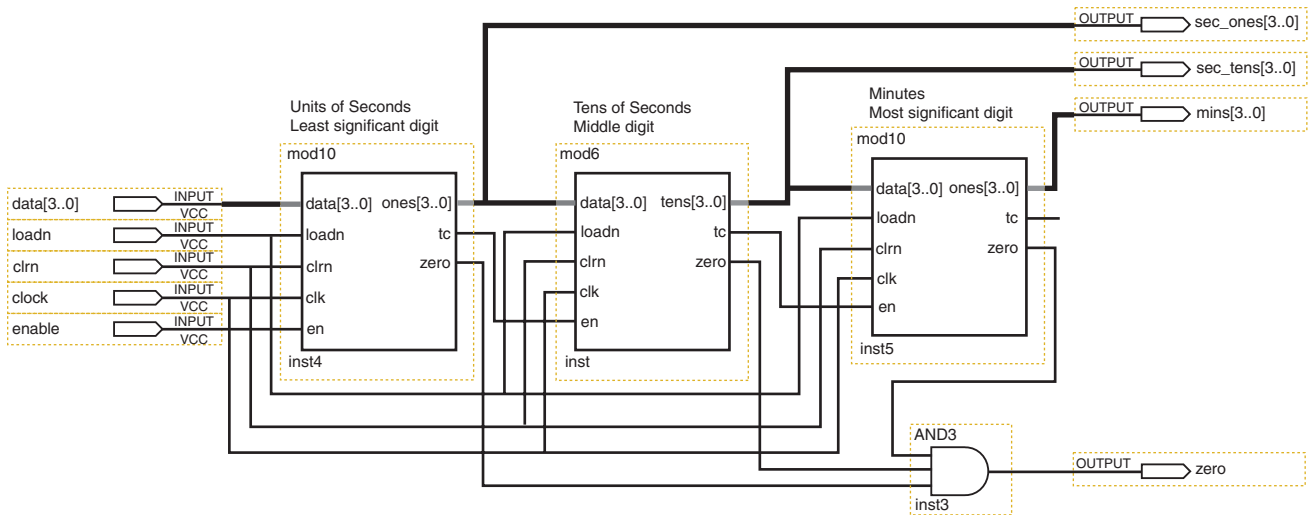


FIGURE 10-42 Level 3: The three-digit BCD down counter block for minutes and seconds.

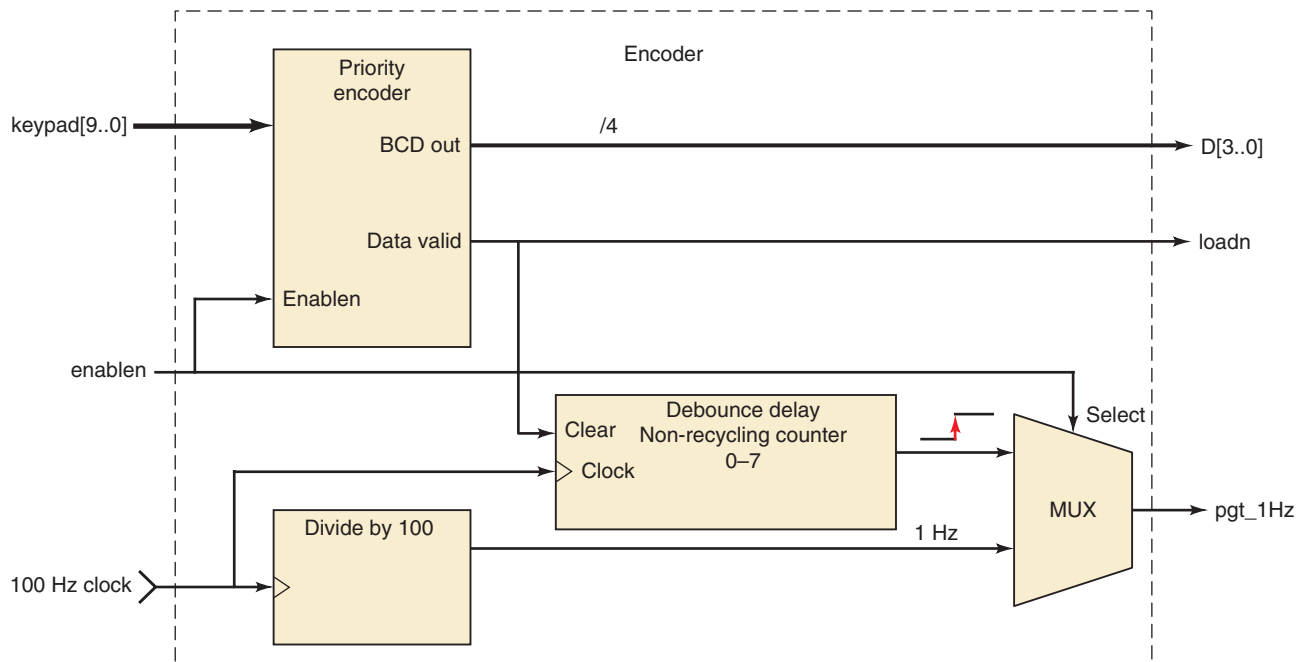
synchronous parallel load capability with an active-LOW *loadn* control. The *clearn* feature is asynchronous and active-LOW while *enable* (*en*) is active-HIGH. All the *load*, *clear*, and *clock* signals for each digit are tied in common and driven by the corresponding input signal. Each block has an output called *tc* (terminal count). The purpose of *tc* is to indicate when that counter digit is at its minimum value (0) and will roll over to its maximum value on the next clock. *tc* goes HIGH when the counter reaches zero, assuming the counter is enabled.

Cascading of these three counters is accomplished by connecting the *tc* of the lower stage to the enable of the next higher stage. This way, when the lowest-order stage is disabled, all the stages hold their current value. Also, notice that the BCD output of the least-significant digit stage is connected to the BCD data input of the middle-digit stage and that the BCD output of the middle-digit stage is connected to the BCD data input of the MSD. This is done to accomplish the data transfer or shifting operation each time a new digit is entered.

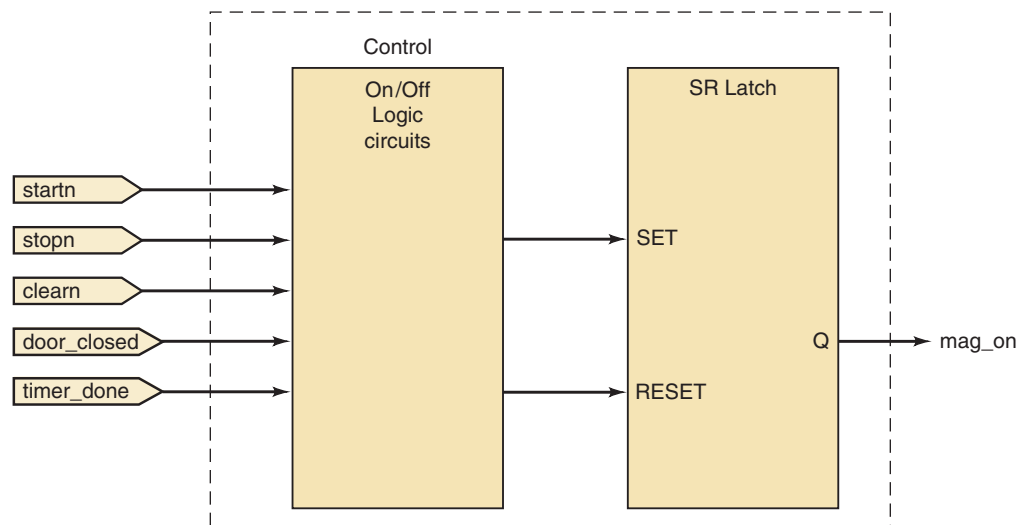
The second functional block (level 2 of hierarchy) in the system is the timer input/control block. It is responsible for recognizing key entries and controlling the counter block. This design block has the 10 keys from the keypad as inputs, a 100-Hz *clock*, and an active-LOW *enablen* that will allow the keypad encoder to function and determine which signal is sent to the clock input of the counter block. The *clock* for the counter must be a 1-Hz clock waveform when the magnetron is energized. When magnetron is off and keys are being pressed to enter the cook time, the counter block's clock input must receive a single positive-going transition (PGT) a few milliseconds after each key is pressed. It must not receive another PGT until that key is released or else it would load multiple digits for a single key entry. Assuring that one clean entry is received from each switch activation is referred to as *debouncing*, which you studied in Chapter 5. In this case, debouncing is accomplished by waiting (delaying) a few milliseconds after the button is pressed before creating the PGT that will clock the BCD digit into the counter. To create this delay, a non-recycling three-bit counter starts counting when a key is pressed. When the counter reaches 4 (40 ms later) the output goes HIGH, creating a PGT. The counter continues to count up but holds at 7 until the clear

control is activated when the key is released. Figure 10-43 shows this control block with its functional elements (level 3). Notice all the different digital building blocks that are used here: encoding, frequency division, multiplexing, and counting.

The third functional block of level 2 is the magnetron control block. It is a logic block used to control the magnetron tube output (*mag\_on*). It must turn on the magnetron when the start button is pressed and remember to stay on after the start button is released. This means there is latching action needed within this block. This block also needs some combinational logic to determine the conditions that can turn the magnetron tube on and off. Figure 10-44 shows the functional elements of this block. Notice the inputs are the four input switches of the system (user controls) as well as a signal that indicates that the timer has expired (*timer\_done*).



**FIGURE 10-43** The encoder/timer input control block decomposed to basic functional blocks.



**FIGURE 10-44** The magnetron control block decomposed to basic functional blocks.

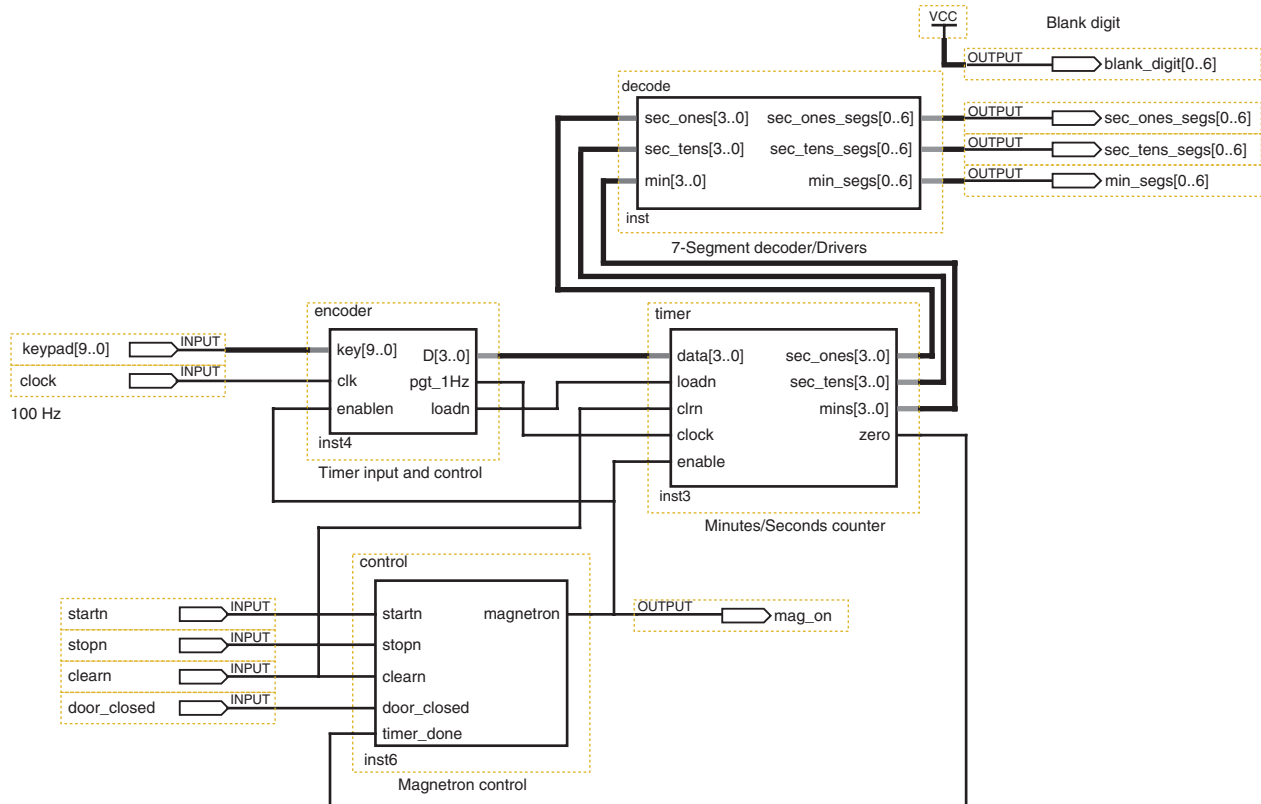


FIGURE 10-45 Level 2 of the hierarchy showing blocks and signals.

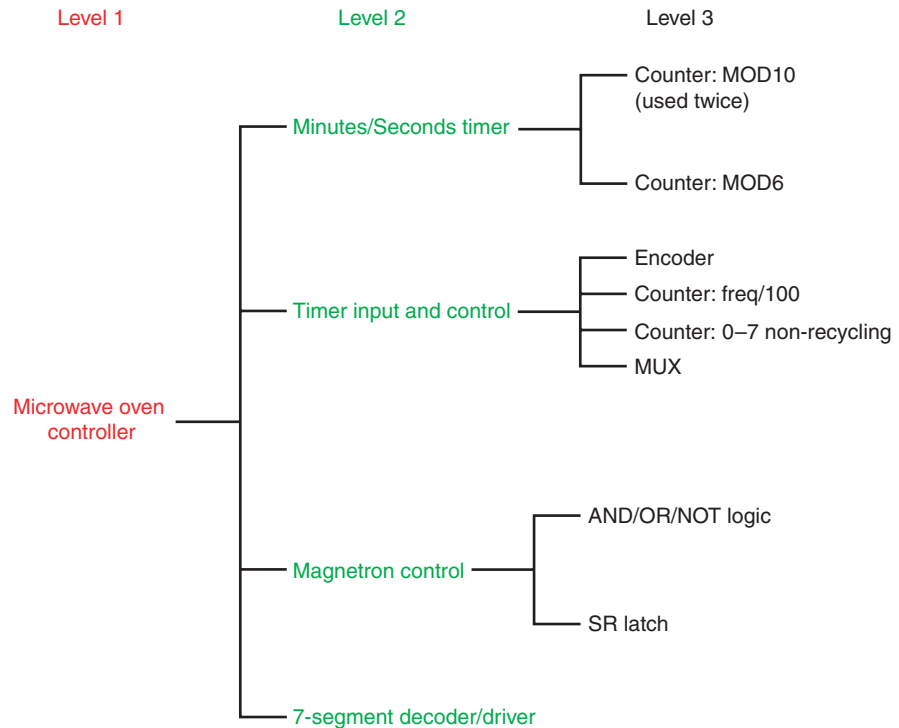
The fourth functional block must decode the three BCD digits, drive the 7-segment LED displays, and provide leading zero blanking features. The decode block could be decomposed into three 7447 decoder/driver circuits (level 3 of the hierarchy) that were described in Chapter 9. The complete functionality of the decode block can also be easily implemented with a single HDL source file, making a third level of hierarchy unnecessary.

Figure 10-45 shows the entire block diagram for level 2 of the hierarchy with all interconnecting signals.

## Synthesis/Integration and Testing

The microwave oven project has now been decomposed as shown in the three levels of hierarchy of Figure 10-46. Notice at the lowest levels (level 3) we have only basic familiar functional blocks. Each of these functional blocks can be implemented using circuits that are similar to other examples in this text using models of TTL ICs (maxplus2 functions), Quartus megafunctions, or by describing their operation using a hardware description language. It is now up to you to solve the problem of synthesizing the small blocks, testing each one, and integrating the system. If this project is implemented on a DE1 (or DE2) board from Altera, there are two more things to notice. The ports connected to VCC in the upper right corner of Figure 10-45 are the segments of the most significant 7-segment display (labeled Blank Digit). This is done in order to turn off the fourth (unused) digit. Also, the 100-Hz clock can be supplied through the external clock input, or by using the on-board 50-MHz crystal and divide it by 500,000 (prescaling) using a megafunction counter as described in Chapter 7.

**FIGURE 10-46**  
Decomposition of the project into three levels of hierarchy.



### OUTCOME ASSESSMENT QUESTIONS

1. What are the names of the functional blocks in level 2 of the microwave oven hierarchy?
2. Describe the signal that drives the clock input to the minutes/seconds timer whenever no buttons on the keypad are being pressed.
3. Describe the signal that drives the clock input to the minutes/seconds timer whenever any buttons on the keypad are being pressed.

## 10-6 FREQUENCY COUNTER PROJECT

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe the requirements of a frequency counter system.
- Decompose the system into simpler blocks.
- Define each input and output for each block.

The project in this section demonstrates the use of counters and other standard logic functions to implement a system called a frequency counter, which is similar to the piece of test equipment that you have probably used in the laboratory. The theory of operation will be described in terms of conventional MSI logic devices and then related to the building blocks that can be developed using HDL. As with most projects, this example consists of several circuits that we have studied in earlier chapters. They are combined here to form a digital system with a unique purpose. First, let us define a frequency counter.

A **frequency counter** is a circuit that can measure and display the frequency of a signal. As you know, the frequency of a periodic waveform is simply the number of cycles per second. Shaping each cycle of the unknown frequency into a digital pulse allows us to use a digital circuit to count the cycles. The general idea behind measuring frequency involves enabling a counter to count the number of cycles (pulses) of the incoming waveform during a precisely specified period of time called the **sampling interval**. The length of the sampling interval determines the range of frequencies that can be measured. A longer interval provides improved precision for low frequencies but will overflow the counter at high frequencies. A shorter sample interval provides a less precise measurement of low frequencies but can measure a much higher maximum frequency without exceeding the upper limit of the counter.

**EXAMPLE 10-1**

Assume that a frequency counter uses a four-digit BCD counter. Determine the maximum frequency that can be measured using each of the following sample intervals:

- (a) 1 second      (b) 0.1 second      (c) 0.01 second

**Solution**

- (a) With a sampling interval of 1 second, the four-digit counter can count up to 9999 pulses. The frequency is 9999 pulses per second or 9.999 kHz.
- (b) The counter can count up to 9999 pulses within the sampling interval of 0.1 second. This translates into a frequency of 99,990 pulses per second or 99.99 kHz.
- (c) The counter can count up to 9999 pulses within the sampling interval of 0.01 second. This translates into a frequency of 999,900 pulses per second or 999.9 kHz.

**EXAMPLE 10-2**

If a frequency of 3792 pps is applied to the input of the frequency counter, what will the counter read under each of the following sample intervals?

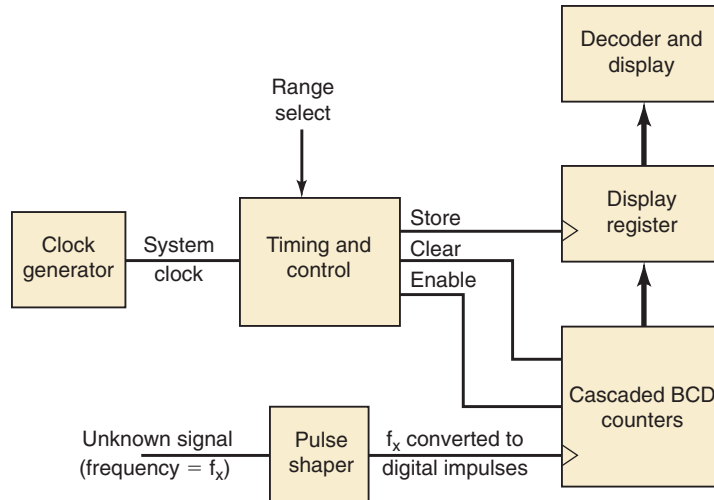
- (a) 1 second      (b) 0.1 second      (c) 10 ms

**Solution**

- (a) During a sampling interval of 1 second, the counter will count 3792 cycles. The frequency will read 3.792 kpps.
- (b) During a sampling interval of 0.1 second, the number of pulses that will be counted is 379 or 380 cycles, depending on where the sample interval begins. The frequency will read 03.79 kpps or 03.80 kpps.
- (c) During a sampling interval of 0.01 second, the number of pulses that will be counted is 37 or 38 cycles, depending on where the sample interval begins. The frequency will read 003.7 kpps or 003.8 kpps.

One of the most straightforward methods for constructing a frequency counter is shown as a block diagram in Figure 10-47. The major blocks are the counter, the display register, the decoder/display, and the timing and control unit. The counter block contains several cascaded BCD counters that are used to count the number of pulses produced by the unknown signal applied to the clock input. The counter block has count enable and clear controls. The

**FIGURE 10-47** Basic frequency counter block diagram.



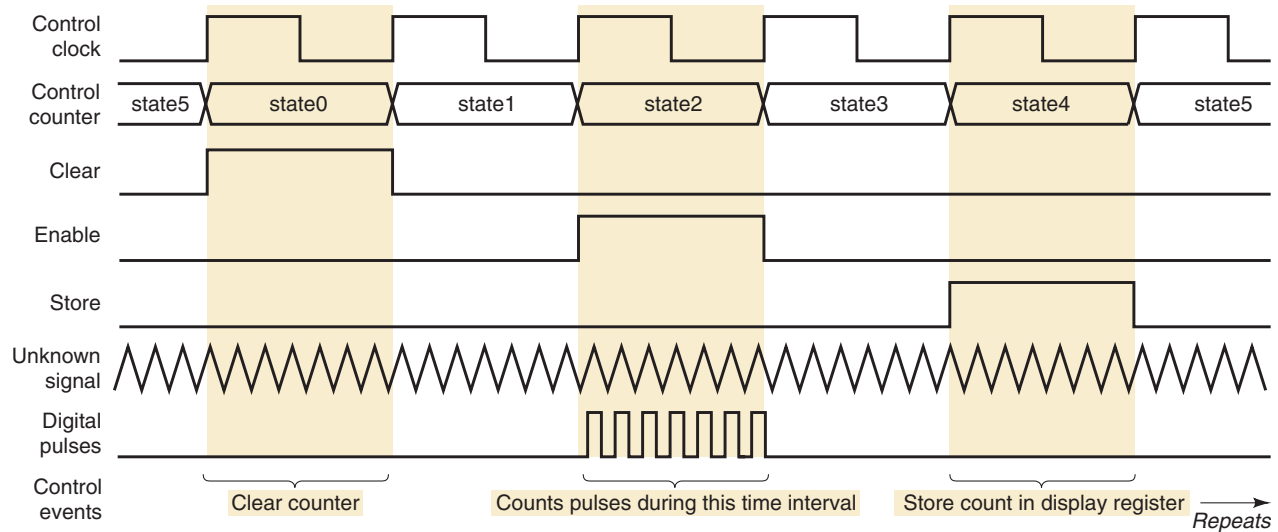
time period for counting (sample interval) is controlled by an enable signal that is produced by the timing and control block. The length of time for the BCD counters to be enabled can be selected with the range select input to the timing and control block. This allows the user to select the desired frequency range to be measured and effectively determines the location of the decimal point in the digital readout. The pulse width of the enable signal (sample interval) is critical for taking an accurate frequency measurement. The counter must be cleared before it is enabled for a new frequency measurement of the unknown signal. After a new count has been taken, the counter is disabled, and the most recent frequency measurement is stored in the display register. The output of the display register is input to the decoder and display block, where the BCD values are converted into decimal for the display readout. Using a separate display register allows the frequency counter to take a new measurement in the background so that the user does not watch the counter while it is totaling the number of pulses for a new reading. The display is instead updated periodically with the last frequency reading.

The accuracy of this frequency counter depends almost entirely on the accuracy of the system clock frequency, which is used to create the proper pulse width for the counter enable signal. A crystal-controlled clock generator is used in Figure 10-47 to produce an accurate system clock for the timing and control block.

A pulse shaper block is needed to ensure that the unknown signal whose frequency is to be measured will be compatible with the clock input for the counter block. A Schmitt-trigger circuit may be used to convert “non-square” waveforms (sine, triangle, etc.) as long as the unknown input signal is of satisfactory amplitude. If the unknown signal might have a larger or smaller amplitude than is compatible with a given Schmitt trigger, then additional analog signal conditioning circuitry, such as an automatic gain control, will be required for the pulse shaper block.

The timing diagram for the control of the frequency counter is shown in Figure 10-48. The control clock is derived from the system clock signal by frequency dividers contained in the control and timing block. The period of the control clock signal is used to create the desired enable pulse width. A recycling control counter inside the control and timing block is clocked by the control clock signal. It has selected states decoded to produce the repeating control signal sequence (clear, enable, and store). The counter (cascaded BCD stages) is first cleared. Then the counter is enabled for the

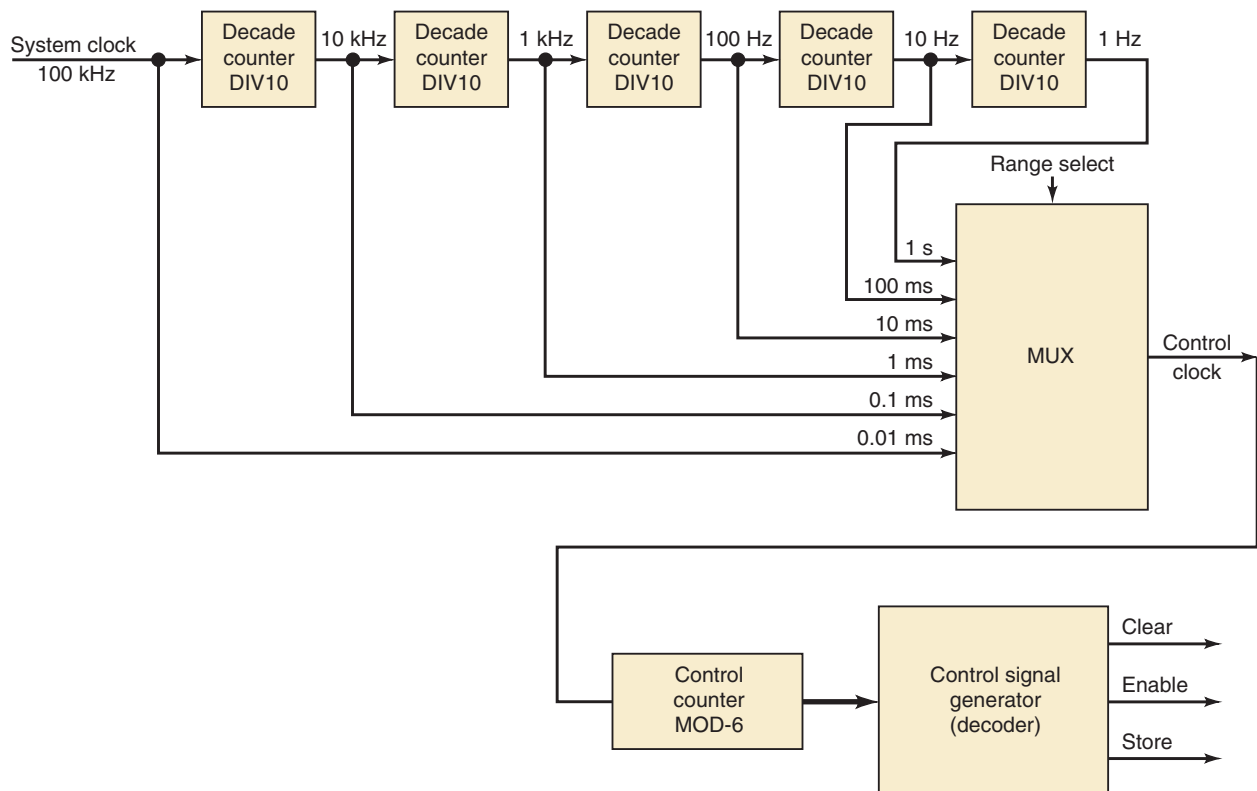




**FIGURE 10-48** Frequency counter timing diagram.

proper sample interval to count the digital pulses, which have the same frequency as the unknown signal. After disabling the counter, the new count is stored in the display register.

The counter, display register, and decoder/display sections are straightforward and are not described any further here. The timing and control block provides the “brains” for our frequency counter and deserves a little more discussion to explain its operation. Figure 10-49 shows the sub-blocks within the timing and control block. For our example design, we will assume



**FIGURE 10-49** Timing and control block for frequency counter.

that the clock generator produces a 100-kHz system clock signal. The system clock frequency is divided by a set of five decade counters (MOD-10). This gives the user six different frequencies that can be selected by the multiplexer for the control clock frequency using the range select control. Because the period of the control clock is the same as the pulse width of the counter enable, this setup allows the frequency counter to have six different frequency measurement ranges. The control counter is a MOD-6 counter that has three selected states decoded by the control signal generator to produce the clear, enable, and store control signals.

### EXAMPLE 10-3

Assume that the BCD counter in Figure 10-47 consists of three cascaded BCD stages and their associated displays. If the unknown frequency is between 1 kpps and 9.99 kpps, which range (sample interval) should be selected using the MUX of Figure 10-49 ?

#### Solution

With three BCD counters, the total capacity of the counter is 999. A 9.99-kpps frequency produces a count of 999 if a 0.1-s sample interval were used. Thus, in order to use the full capacity of the counter, the MUX should select the 0.1-s clock period (10 Hz). If a 1-s sampling interval were used, the counter capacity would always be exceeded for frequencies in the specified range. If a shorter sample interval were used, the counter would count only between 1 and 99, which would give a reading to only two significant figures and would be a waste of the counter's capacity.

### OUTCOME ASSESSMENT QUESTIONS

1. What is the purpose of running the unknown signal through a pulse shaper?
2. What are the units of a frequency measurement?
3. What does the display show during the sample interval?

## SUMMARY

1. Successful project management can be accomplished by the following steps: overall project definition; breaking the project into small, strategic pieces; synthesis and testing of each piece; and system integration.
2. Small projects like the stepper motor driver can be completed in a single design file, even though these projects are developed modularly.
3. Projects that consist of several simple building blocks, like the keypad encoder, can produce very useful systems.
4. Larger projects like the digital clock and the microwave oven can often take advantage of standard common modules that can be used repeatedly in the overall design.
5. Projects should be built and tested in modules starting at the lowest levels of hierarchy.
6. Preexisting modules can easily be combined with new custom modules using both graphical and text-based description methods.
7. Modules can be combined and represented as a single block in the next higher level of the hierarchy using the Altera design tools.

## IMPORTANT TERMS

---

nesting  
hierarchy

prescaler  
frequency counter

sampling interval

## PROBLEMS

---

### SECTION 10-1

- B** 10-1. The security monitoring system of Section 9-8 can be developed as a project.
- Write a project definition with specifications for this system.
  - Define three major blocks of this project.
  - Identify the signals that interconnect the blocks.
  - \*At what frequency must the oscillator run for a 2.5-Hz flash rate?
  - \*Why is it reasonable to use only one current limiting resistor for all eight LEDs?

### SECTION 10-2

Problems 10-2 through 10-8 refer to stepper motors described in Section 10-2.

- B** 10-2\*How many full steps must occur for a complete revolution?
- B** 10-3\*How many degrees of rotation result from one complete cycle through the full-step sequence in Table 10-1?
- B** 10-4. How many degrees of rotation result from one complete cycle through the half-step sequence in Table 10-1?
- B** 10-5. The *cout* lines of Figure 10-1 started at 1010 and have just progressed through the following sequence: 1010, 1001, 0101, 0110.
- \*How many degrees has the shaft rotated?
  - What sequence will reverse the rotation and return the shaft to its original position?
- B** 10-6. Describe a method to test the stepper driver in:
- Full-step mode
  - Half-step mode
  - Wave-drive mode
  - Direct-drive mode
- D, H** 10-7. Rewrite the stepper driver design file of Figure 10-8 or 10-9 without using a CASE statement. Use your favorite HDL.
- D, H** 10-8. Modify the stepper design file of Figure 10-8 or 10-9 to add an enable input that puts the outputs in the Hi-Z state (tristate) when enable = 0.

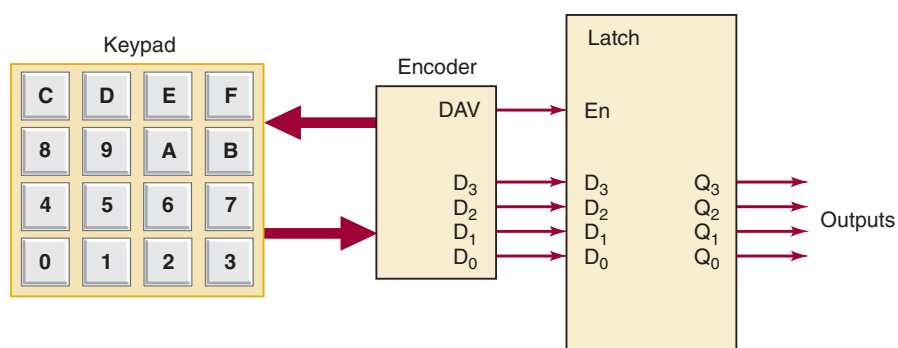
### SECTION 10-3

- B** 10-9. Write the state table for the ring counter shown in Figure 10-11 and described in Figure 10-13.
- B** 10-10\* With no keys pressed, what is the value on c[3..0]?
- B** 10-11. Assume that the ring counter is in state 0111 when someone presses the 7 key. Will the ring counter advance to the NEXT state?

---

\*Answers to problems marked with an asterisk can be found in the back of the text.

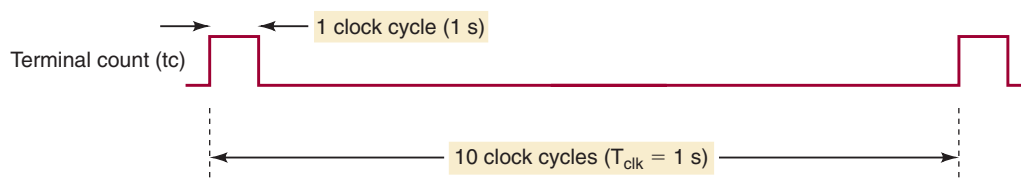
- B** 10-12. Assume the 9 key is pressed and held until  $DAV = 1$ .
  - (a)\*What is the value on the ring counter?
  - (b) What is the value encoded by the row encoder?
  - (c) What is the value encoded by the column encoder?
  - (d) What binary number is on the  $D[3..0]$  lines?
- B** 10-13\* In Problem 10-12, will the data be valid on the falling edge of  $DAV$ ?
- B, D** 10-14. If you wanted to latch data from the keypad into a 74174 register, which signal from the keypad would you connect to the clock of the register? Draw the circuit.
- T** 10-15\* The keypad is connected to a 74373 octal transparent latch as shown in Figure 10-50. The output is correct as long as a key is held. However, it is unable to latch data between key presses. Why will this circuit *not* work correctly?



**FIGURE 10-50** Problem 10-15.

**SECTION 10-4**

- B** 10-16. Assume a 1-Hz clock is applied to the seconds stage of the clock in Figure 10-17. The MOD-10 *units of seconds* counter's terminal count ( $tc$ ) output is shown in Figure 10-51. Draw a similar diagram showing the number of clock cycles between the  $tc$  output pulses of each of the following:
  - (a)\*Tens of seconds counter
  - (b) Units of minutes counter
  - (c) Tens of minutes counter



**FIGURE 10-51** Problem 10-16.

- B** 10-17\* How many cycles of the 60-Hz power line will occur in a 24-hour period? What problem do you think will result if we attempt to simulate the operation of the entire clock circuit?
- D** 10-18\* Many digital clocks are set by simply making them count faster while a push button is held down. Modify the design to add this feature.
- D, H** 10-19. Modify the hours stage of Figure 10-18 to keep military time (00–23 hours).

**SECTION 10-5**

- D, N** 10-20. Refer to Figure 10-42. Each counter block in this figure represents the lowest level of the hierarchy established for this project: a primary functional block. Its specifications are MOD 10 (or 6), BCD down counter, active-LOW synchronous load, active-LOW asynchronous clear, active-HIGH enable, positive-edge triggered, active-HIGH terminal count output (gated by enable), active-HIGH decode zero output.
- Create an Altera megafunction to implement this block.
  - Write the AHDL code to implement this block.
  - Write the VHDL code to implement this block.
- D, N** 10-21. Refer to Figure 10-43. The sub-blocks (encoder, divide-by counter, non-recycling counter, MUX) could be implemented as separate blocks in the third level of hierarchy of this project. Code can be found in previous examples that will work for each of these blocks with only slight modification. These functional elements can also be combined in a single HDL source file. Write the code for the entire encoder/timer control block in:
- AHDL
  - VHDL
- D, N** 10-22. Refer to Figure 10-44. The block on the left is simply combinational logic that must control the S-R latch that turns on and off the magnetron tube.
- Draw a logic diagram using only gates to implement this circuit.
  - Describe this block using AHDL.
  - Describe this block using VHDL.
- D, N** 10-23. Refer to Figure 10-45. This block decodes the three BCD digits from the timer block and drives the active-LOW 7-segment LED displays. It must also accomplish leading zero blanking.
- Use 7447 standard logic blocks to implement this block.
  - Use AHDL to implement this block.
  - Use VHDL to implement this block.

**SECTION 10-6**

- B** 10-24. Draw the hierarchy diagram for the frequency counter project.
- D, H** 10-25. Write the HDL code for the MOD-6 control counter and control signal generator in Figure 10-49.
- D, H** 10-26\* Write the HDL code for the MUX of Figure 10-49.
- D** 10-27. Use graphic design techniques and the BCD counter described in Figure 10-31, the MUX, and the control signal generator design to create the entire timing and control block for the frequency counter project.
- D, H** 10-28. Write the HDL code for the timing and control section of the frequency counter.

**ANSWERS TO OUTCOME ASSESSMENT QUESTIONS****SECTION 10-1**

- Definition, strategic planning/problem decomposition, synthesis and testing, system integration and testing
- The definition stage

**SECTION 10-2**

1. Full-step, half-step, wave-drive, and direct-drive
2.  $\text{cin}_0\text{-cin}_3$  [mode selector switches set to (1,1)]
3. Step, direction [mode selector switches set to (0,1)]
4. Eight states

**SECTION 10-3**

1. Only one
2. The first one scanned after being pressed (usually the first one pressed)
3. To make DAV go HIGH after the data stabilizes
4. No, it goes HIGH on the next clock after the key is pressed.
5. Whenever OE is LOW or when no keys are pressed

**SECTION 10-4**

1. The overall operating specifications and the system inputs and outputs.
2. At the top of the hierarchy
3. At the bottom, building the simplest blocks first
4. At each stage of modular implementation

**SECTION 10-5**

1. Minutes/Seconds counter; Timer input/control; Magnetron Control; 7-segment decoder/driver
2. It is a clock waveform at a frequency of 1 Hz.
3. It is a single PGT that occurs about 40 ms after the key is pressed. It stays HIGH until the key is released.

**SECTION 10-6**

1. To change the shape of the analog signal into a digital signal of the same frequency
  2. Cycles per second (Hz) or pulses per second (pps)
  3. The display shows the frequency measured during the previous sample interval.
-



# INTERFACING WITH THE ANALOG WORLD

## ■ OUTLINE

- 11-1 Review of Digital Versus Analog
- 11-2 Digital-to-Analog Conversion
- 11-3 DAC Circuitry
- 11-4 DAC Specifications
- 11-5 An Integrated-Circuit DAC
- 11-6 DAC Applications
- 11-7 Troubleshooting DACs
- 11-8 Analog-to-Digital Conversion
- 11-9 Digital-Ramp ADC
- 11-10 Data Acquisition
- 11-11 Successive-Approximation ADC
- 11-12 Flash ADCs
- 11-13 Other A/D Conversion Methods
- 11-14 Typical ADC Architectures for Applications
- 11-15 Sample-and-Hold Circuits
- 11-16 Multiplexing
- 11-17 Digital Signal Processing (DSP)
- 11-18 Applications of Analog Interfacing

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Describe the theory of operation and the circuit limitations of several types of digital-to-analog converters (DACs).
- Explain the various DAC manufacturer specifications.
- Use different test procedures to troubleshoot DAC circuits.
- Compare the advantages and disadvantages of the major analog-to-digital converter (ADC) architectures.
- Analyze the process by which a computer, in conjunction with an ADC, digitizes an analog signal and then reconstructs that analog signal from the digital data.
- Explain the need for using sample-and-hold circuits in conjunction with ADCs.
- Describe the operation of an analog multiplexing system.
- Describe the basic concepts of digital signal processing.

## 11-1 REVIEW OF DIGITAL VERSUS ANALOG

### OUTCOME

*Upon completion of this section, you will be able to:*

- Describe the role of each element of a digital control system.

A **digital quantity** has a value that is specified as one of two possibilities, such as 0 or 1, LOW or HIGH or true or false. In practice, a digital quantity such as a voltage may actually have a value that is anywhere within specified ranges, and we define values within a given range to have the same digital value. For example, for TTL logic, we know that

$$\begin{aligned}0 \text{ V to } 0.8 \text{ V} &= \text{logic } 0 \\2 \text{ V to } 5 \text{ V} &= \text{logic } 1\end{aligned}$$

Any voltage falling in the range from 0 to 0.8 V is given the digital value 0, and any voltage in the range 2 to 5 V is assigned the digital value 1. The exact voltage values are not significant because the digital circuits respond in the same way to all voltage values within a given range.

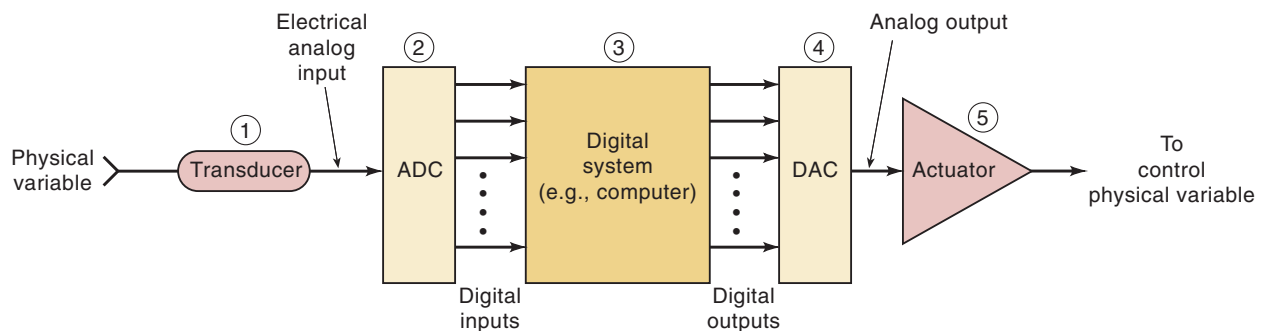
By contrast, an **analog quantity** can take on any value over a continuous range of values and, most important, its exact value is significant. For example, the output of an analog temperature-to-voltage converter might be measured as 2.76 V, which may represent a specific temperature of 27.6°C. If the voltage were measured as something different, such as 2.34 V or 3.78 V, this would represent a completely different temperature. In other words, each possible value of an analog quantity has a different meaning. Another



example of this is the output voltage from an audio amplifier into a speaker. This voltage is an analog quantity because each of its possible values produces a different response in the speaker.

Most physical variables are analog in nature and can take on any value within a continuous range of values. Examples include temperature, pressure, light intensity, audio signals, position, rotational speed, and flow rate. Digital systems perform all of their internal operations using digital circuitry and digital operations. Any information that must be input to a digital system must first be put into digital form. Similarly, the outputs from a digital system are always in digital form. When a digital system such as a computer is to be used to monitor and/or control a physical process, we must deal with the difference between the digital nature of the computer and the analog nature of the process variables. Figure 11-1 illustrates the situation. This diagram shows the five elements that are involved when a computer is monitoring and controlling a physical variable that is assumed to be analog:

1. **Transducer.** The physical variable is normally a nonelectrical quantity. A **transducer** is a device that converts the physical variable to an electrical variable. Some common transducers include thermistors, photocells, photodiodes, flow meters, pressure transducers, and tachometers. The electrical output of the transducer is an analog current or voltage that is proportional to the physical variable that it is monitoring. For example, the physical variable could be the temperature of water in a large tank that is being filled from cold and hot water pipes. Let's say that the water temperature varies from 80 to 150°F and that a thermistor and its associated circuitry convert this water temperature to a voltage ranging from 800 to 1500 mV. Note that the transducer's output is directly proportional to temperature such that each 1°F produces a 10-mV output. This proportionality factor was chosen for convenience.
2. **Analog-to-digital converter (ADC).** The transducer's electrical analog output serves as the analog input to the **analog-to-digital converter (ADC)**. The ADC converts this analog input to a digital output. This digital output consists of a number of bits that represent the value of the analog input. For example, the ADC might convert the transducer's 800 to 1500-mV analog values to binary values ranging from 01010000 (80) to 10010110 (150). Note that the binary output from the ADC is proportional to the analog input voltage so that each unit of the digital output represents 10 mV.



**FIGURE 11-1** Analog-to-digital converter (ADC) and digital-to-analog converter (DAC) are used to interface a computer to the analog world so that the computer can monitor and control a physical variable.

3. **Computer.** The digital representation of the process variable is transmitted from the ADC to the digital computer, which stores the digital value and processes it according to a program of instructions that it is executing. The program might perform calculations or other operations on this digital representation of temperature to come up with a digital output that will eventually be used to control the temperature.
4. **Digital-to-analog converter (DAC).** This digital output from the computer is connected to a **digital-to-analog converter (DAC)**, which converts it to a proportional analog voltage or current. For example, the computer might produce a digital output ranging from 00000000 to 11111111, which the DAC converts to a voltage ranging from 0 to 10 V.
5. **Actuator.** The analog signal from the DAC is often connected to some device or circuit that serves as an actuator to control the physical variable. For our water temperature example, the actuator might be an electrically controlled valve that regulates the flow of hot water into the tank in accordance with the analog voltage from the DAC. The flow rate would vary in proportion to this analog voltage, with 0 V producing no flow and 10 V producing the maximum flow.

Thus, we see that ADCs and DACs function as *interfaces* between a completely digital system, such as a computer, and the analog world. This function has become increasingly more important as inexpensive microcomputers have moved into areas of process control where computer control was previously not feasible.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the function of a transducer?
2. What is the function of an ADC?
3. What does a computer often do with the data that it receives from an ADC?
4. What function does a DAC perform?
5. What is the function of an actuator?

## 11-2 DIGITAL-TO-ANALOG CONVERSION

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Determine the full-scale output of a DAC.
- Given the digital input, calculate the analog output.
- Given the analog output, calculate the digital input.
- Determine the step size or resolution of a DAC.
- Calculate the percentage resolution.

We will now begin our study of digital-to-analog (D/A) and analog-to-digital (A/D) conversion. Many A/D conversion methods utilize the D/A conversion process, so we will examine D/A conversion first.

Basically, *D/A conversion* is the process of taking a value represented in *digital code* (such as straight binary or BCD) and converting it to a voltage or current that is proportional to the digital value. Figure 11-2(a) shows the

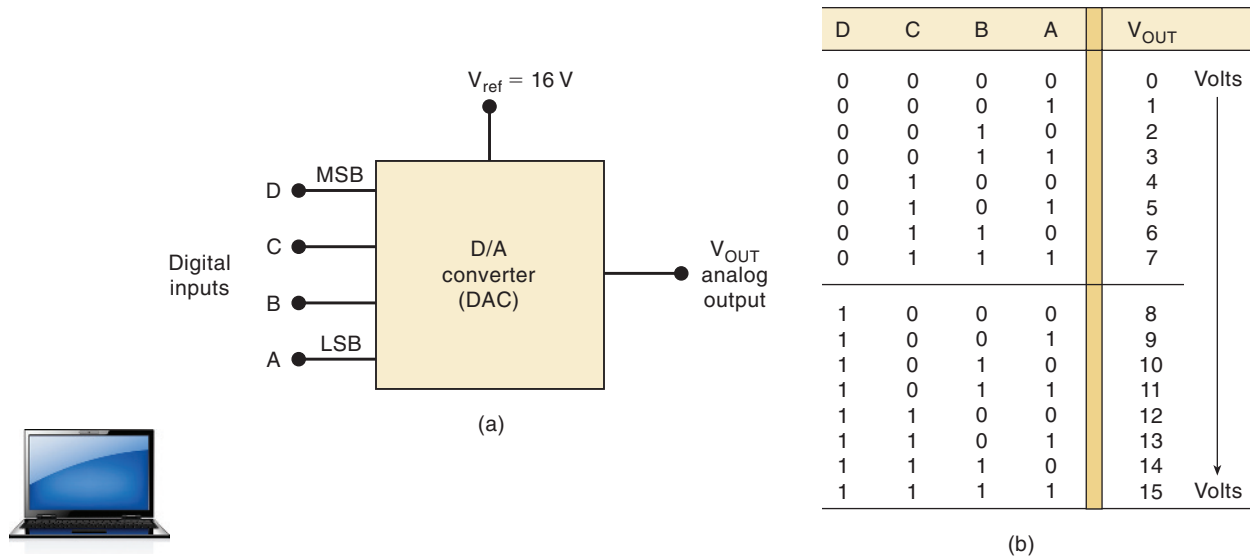


FIGURE 11-2 Four-bit DAC with voltage output.

symbol for a typical four-bit D/A converter. We will not concern ourselves with the internal circuitry until later. For now, we will examine the various input/output relationships.

Notice that there is an input for a voltage reference,  $V_{ref}$ . This input is used to determine the **full-scale output** or maximum value that the D/A converter can produce. The digital inputs  $D$ ,  $C$ ,  $B$ , and  $A$  are usually derived from the output register of a digital system. The  $2^4 = 16$  different binary numbers represented by these four bits are listed in Figure 11-2(b). For each input number, the D/A converter output voltage is a unique value. In fact, for this case, the analog output voltage  $V_{OUT}$  is equal in volts to the binary number. It could also have been twice the binary number or some other proportionality factor. The same idea would hold true if the D/A output were a current  $I_{OUT}$ .

In general,

$$\text{analog output} = K \times \text{digital input} \quad (11-1)$$

where  $K$  is the proportionality factor and is a constant value for a given DAC connected to a fixed reference voltage. The analog output can, of course, be a voltage or a current. When it is a voltage,  $K$  will be in voltage units, and when the output is a current,  $K$  will be in current units. For the DAC of Figure 11-2,  $K = 1 \text{ V}$ , so that

$$V_{OUT} = (1 \text{ V}) \times \text{digital input}$$

We can use this to calculate  $V_{OUT}$  for any value of digital input. For example, with a digital input of  $1100_2 = 12_{10}$ , we obtain

$$V_{OUT} = 1 \text{ V} \times 12 = 12 \text{ V}$$

#### EXAMPLE 11-1A

A five-bit DAC has a current output. For a digital input of 10100, an output current of 10 mA is produced. What will  $I_{OUT}$  be for a digital input of 11101?

**Solution**

The digital input  $10100_2$  is equal to decimal 20. Because  $I_{OUT} = 10\text{ mA}$  for this case, the proportionality factor must be  $0.5\text{ mA}$ . Thus, we can find  $I_{OUT}$  for any digital input such as  $11101_2 = 29_{10}$  as follows:

$$\begin{aligned} I_{OUT} &= (0.5\text{ mA}) \times 29 \\ &= 14.5\text{ mA} \end{aligned}$$

Remember, the proportionality factor,  $K$ , varies from one DAC to another and depends on the reference voltage.

**EXAMPLE 11-1B**

What is the largest value of output voltage from an eight-bit DAC that produces  $1.0\text{ V}$  for a digital input of  $00110010$ ?

**Solution**

$$\begin{aligned} 00110010_2 &= 50_{10} \\ 1.0\text{ V} &= K \times 50 \end{aligned}$$

Therefore,

$$K = 20\text{ mV}$$

The largest output will occur for an input of  $11111111_2 = 255_{10}$ .

$$\begin{aligned} V_{OUT(\text{max})} &= 20\text{ mV} \times 255 \\ &= 5.10\text{ V} \end{aligned}$$

**Analog Output**

The output of a DAC is technically not an analog quantity because it can take on only specific values, such as the 16 possible voltage levels for  $V_{OUT}$  in Figure 11-2, as long as  $V_{ref}$  is constant. Thus, in that sense, it is actually digital. As we will see, however, the number of different possible output values can be increased and the difference between successive values decreased by increasing the number of input bits. This will allow us to produce an output that is more and more like an analog quantity that varies continuously over a range of values. In other words, the DAC output is a “pseudo-analog” quantity. We will continue to refer to it as analog, keeping in mind that it is an approximation to a pure analog quantity.

**Input Weights**

For the DAC of Figure 11-2, note that each digital input contributes a different amount to the analog output. This is easily seen if we examine the cases where only one input is HIGH (Table 11-1). The contributions of each digital input are *weighted* according to their position in the binary number. Thus,  $A$ , which is the LSB, has a *weight* of  $1\text{ V}$ ;  $B$  has a weight of  $2\text{ V}$ ;  $C$  has a weight of  $4\text{ V}$ ; and  $D$ , the MSB, has the largest weight,  $8\text{ V}$ . The weights are successively doubled for each bit, beginning with the LSB. Thus, we can consider  $V_{OUT}$  to be the weighted sum of the digital inputs. For instance, to find  $V_{OUT}$  for the digital input  $0111$ , we can add the weights of the  $C$ ,  $B$ , and  $A$  bits to obtain  $4\text{ V} + 2\text{ V} + 1\text{ V} = 7\text{ V}$ .

**TABLE 11-1** Bit contributions to  $V_{OUT}$ .

D	C	B	A		$V_{OUT}$ (V)
0	0	0	1	→	1
0	0	1	0	→	2
0	1	0	0	→	4
1	0	0	0	→	8

### EXAMPLE 11-2

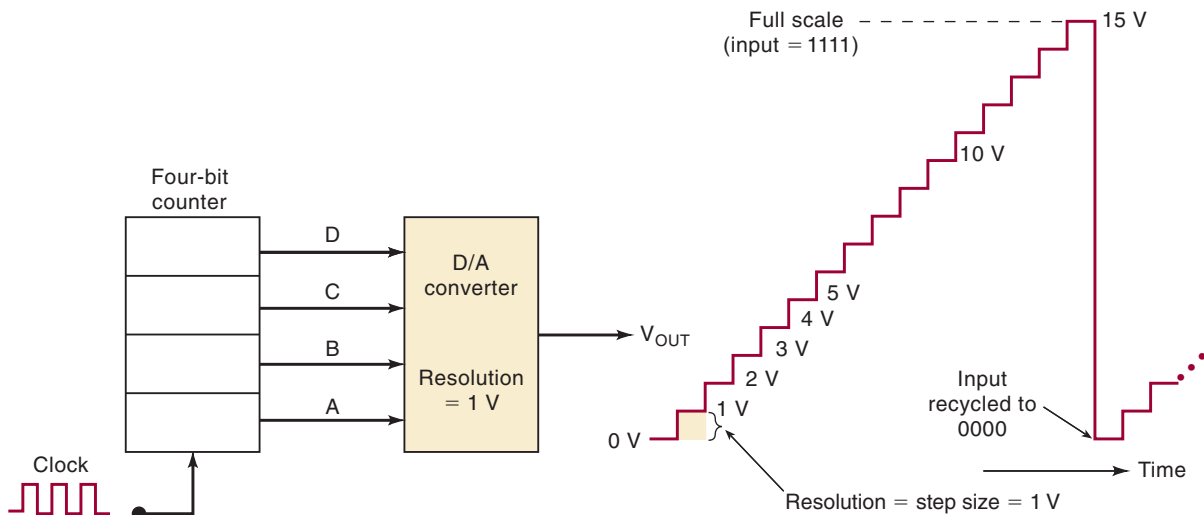
A five-bit D/A converter produces  $V_{OUT} = 0.2$  V for a digital input of 00001. Find the value of  $V_{OUT}$  for an input of 11111.

#### Solution

Obviously, 0.2 V is the weight of the LSB. Thus, the weights of the other bits must be 0.4 V, 0.8 V, 1.6 V, and 3.2 V, respectively. For a digital input of 11111, then, the value of  $V_{OUT}$  will be  $3.2$  V +  $1.6$  V +  $0.8$  V +  $0.4$  V +  $0.2$  V =  $6.2$  V.

### Resolution (Step Size)

**Resolution** of a D/A converter is defined as the smallest change that can occur in the analog output as a result of a change in the digital input. Referring to the table in Figure 11-2, we can see that the resolution is 1 V because  $V_{OUT}$  can change by no less than 1 V when the digital input value is changed. The resolution is always equal to the weight of the LSB and is also referred to as the **step size** because it is the amount that  $V_{OUT}$  will change as the digital input value is changed from one step to the next. This is illustrated better in Figure 11-3, where the outputs from a four-bit binary counter provide the inputs to our DAC. As the counter is being continually cycled through its 16 states by the clock signal, the DAC output is a **staircase** waveform that goes up 1 V per step. When the counter is at 1111, the DAC output is at its maximum value of 15 V; this is its full-scale output. When the counter recycles to 0000, the DAC output returns to 0 V. The resolution (or step size) is the size of the jumps in the staircase waveform; in this case, each step is 1 V.



**FIGURE 11-3** Output waveforms of a DAC as inputs are provided by a binary counter.

Note that the staircase has 16 levels corresponding to the 16 input states, but there are only 15 steps or jumps between the 0-V level and full scale. In general, for an  $N$ -bit DAC the number of different levels will be  $2^N$ , and the number of steps will be  $2^N - 1$ .

You may have already figured out that resolution (step size) is the same as the proportionality factor in the DAC input/output relationship:

$$\text{analog output} = K \times \text{digital input}$$

A new interpretation of this expression would be that the digital input is equal to the number of steps,  $K$  is the amount of voltage (or current) per step, and the analog output is the product of the two. We now have a convenient way of calculating the value of  $K$  for the D/A:

$$\text{resolution} = K = \frac{A_{fs}}{(2^N - 1)} \quad (11-2)$$

where  $A_{fs}$  is the analog full-scale output and  $N$  is the number of bits.

#### EXAMPLE 11-3A

What is the resolution (step size) of the DAC of Example 11-2? Describe the staircase signal out of this DAC.

#### Solution

The LSB for this converter has a weight of 0.2 V. This is the resolution or step size. A staircase waveform can be generated by connecting a five-bit counter to the DAC inputs. The staircase will have 32 levels, from 0 V up to a full-scale output of 6.2 V, and 31 steps of 0.2 V each.

#### EXAMPLE 11-3B

For the DAC of Example 11-2, determine  $V_{OUT}$  for a digital input of 10001.

#### Solution

The step size is 0.2 V, which is the proportionality factor  $K$ . The digital input is  $10001 = 17_{10}$ . Thus, we have

$$\begin{aligned} V_{OUT} &= (0.2 \text{ V}) \times 17 \\ &= 3.4 \text{ V} \end{aligned}$$

### Percentage Resolution

Although resolution can be expressed as the amount of voltage or current per step, it is also useful to express it as a percentage of the *full-scale output*. To illustrate, the DAC of Figure 11-3 has a maximum full-scale output of 15 V (when the digital input is 1111). The step size is 1 V, which gives a percentage resolution of

$$\begin{aligned} \% \text{ resolution} &= \frac{\text{step size}}{\text{full scale (F.S.)}} \times 100\% \\ &= \frac{1 \text{ V}}{15 \text{ V}} \times 100\% = 6.67\% \end{aligned} \quad (11-3)$$

**EXAMPLE 11-4**

A 10-bit DAC has a step size of 10 mV. Determine the full-scale output voltage and the percentage resolution.

**Solution**

With 10 bits, there will be  $2^{10} - 1 = 1023$  steps of 10 mV each. The full-scale output will therefore be  $10\text{ mV} \times 1023 = 10.23\text{ V}$ , and

$$\% \text{ resolution} = \frac{10\text{ mV}}{10.23\text{ V}} \times 100\% \approx 0.1\%$$

Example 11-4 helps to illustrate the fact that the percentage resolution becomes smaller as the number of input bits is increased. In fact, the percentage resolution can also be calculated from

$$\% \text{ resolution} = \frac{1}{\text{total number of steps}} \times 100\% \quad (11-4)$$

For an  $N$ -bit binary input code, the total number of steps is  $2^N - 1$ . Thus, for the previous example,

$$\begin{aligned} \% \text{ resolution} &= \frac{1}{2^{10} - 1} \times 100\% \\ &= \frac{1}{1023} \times 100\% \\ &\approx 0.1\% \end{aligned}$$

This means that it is *only the number of bits* that determines the *percentage resolution*. Increasing the number of bits increases the number of steps to reach full scale, so that each step is a smaller part of the full-scale voltage. Most DAC manufacturers specify resolution as the number of bits.

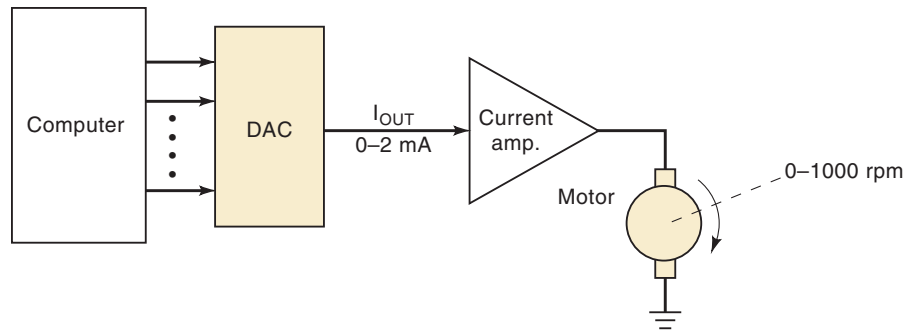
**What Does Resolution Mean?**

A DAC cannot produce a continuous range of output values and so, strictly speaking, its output is not truly analog. A DAC produces a finite set of output values. In our water temperature example of Section 11-1, the computer generates a digital output to provide an analog voltage between 0 and 10 V to an electrically controlled valve. The DAC's resolution (number of bits) determines how many possible voltage values the computer can send to the valve. If a six-bit DAC is used, there will be 63 possible steps of 0.159 V between 0 and 10 V. When an eight-bit DAC is used, there will be 255 possible steps of 0.039 V between 0 and 10 V. The greater the number of bits, the finer the resolution (the smaller the step size).

The system designer must decide what resolution is needed on the basis of the required system performance. The resolution limits how close the DAC output can come to a given analog value. Generally, the cost of DACs increases with the number of bits, and so the designer will use only as many bits as necessary.

**EXAMPLE 11-5**

Figure 11-4 shows a computer controlling the speed of a motor. The 0- to 2-mA analog current from the DAC is amplified to produce motor speeds from 0 to 1000 rpm (revolutions per minute). How many bits should be used



**FIGURE 11-4** Example 11-5.

if the computer is to be able to produce a motor speed that is within 2 rpm of the desired speed?

### Solution

The motor speed will range from 0 to 1000 rpm as the DAC goes from zero to full scale. Each step in the DAC output will produce a step in the motor speed. We want the step size to be no greater than 2 rpm. Thus, we need at least 500 steps ( $1000/2$ ). Now we must determine how many bits are required so that there are at least 500 steps from zero to full scale. We know that the number of steps is  $2^N - 1$ , and so we can say

$$2^N - 1 \geq 500$$

or

$$2^N \geq 501$$

Since  $2^8 = 256$  and  $2^9 = 512$ , the smallest number of bits that will produce at least 500 steps is *nine*. We could use more than nine bits, but this might add to the cost of the DAC.

### EXAMPLE 11-6

Using nine bits, how close to 326 rpm can the motor speed be adjusted?

### Solution

With nine bits, there will be 511 steps ( $2^9 - 1$ ). Thus, the motor speed will go up in steps of  $1000 \text{ rpm} / 511 = 1.957 \text{ rpm}$ . The number of steps needed to reach 326 rpm is  $326 / 1.957 = 166.58$ . This is not a whole number of steps, and so we will round it to 167. The actual motor speed on the 167th step will be  $167 \times 1.957 = 326.8 \text{ rpm}$ . Thus, the computer must output the nine-bit binary equivalent of  $167_{10}$  to produce the desired motor speed within the resolution of the system.

In all of our examples, we have assumed that the DACs have been perfectly accurate in producing an analog output that is directly proportional to the binary input, and that the resolution is the only thing that limits how close we can come to a desired analog value. This, of course, is unrealistic because all devices contain inaccuracies. We will examine the causes and effects of DAC inaccuracy in Sections 11-3 and 11-4.



## Bipolar DACs

Up to this point we have assumed that the binary input to a DAC has been an unsigned number and the DAC output has been a positive voltage or current. Many DACs can also produce negative voltages by making slight changes to the analog circuitry on the output of the DAC. In this case the range of binary inputs (e.g., 00000000 to 11111111) spans a range of  $-V_{\text{ref}}$  to approximately  $+V_{\text{ref}}$ . The value of 10000000 converts to 0 V out. The output of a signed 2's complement digital system can drive this type of DAC by inverting the MSB, which converts the signed binary numbers to the proper values for the DAC as shown in Table 11-2.

**TABLE 11-2** Converting signed integers to DAC input requirements.

	Signed 2's Complement	DAC Inputs	DAC $V_{\text{out}}$
Most positive	01111111	11111111	$\sim +V_{\text{ref}}$
Zero	00000000	10000000	0 V
Most negative	10000000	00000000	$-V_{\text{ref}}$

Other DACs may have the extra circuitry built in and accept 2's complement signed numbers as inputs. For example, suppose that we have a six-bit bipolar DAC that uses the 2's-complement system and has a resolution of 0.2 V. The binary input values range from 100000 ( $-32$ ) to 011111 ( $+31$ ) to produce analog outputs in the range from  $-6.4$  V to  $+6.2$  V. There are 63 steps ( $2^6 - 1$ ) of 0.2 V between these negative and positive limits.

### OUTCOME ASSESSMENT QUESTIONS

1. An eight-bit DAC has an output of 3.92 mA for an input of 01100010. What are the DAC's resolution and full-scale output?
2. What is the weight of the MSB of the DAC of question 1?
3. What is the percentage resolution of an eight-bit DAC?
4. How many different output voltages can a 12-bit DAC produce?
5. For the system of Figure 11-4, how many bits should be used if the computer is to control the motor speed within 0.4 rpm?
6. *True or false:* The percentage resolution of a DAC depends *only* on the number of bits.
7. What is the advantage of a smaller (finer) resolution?

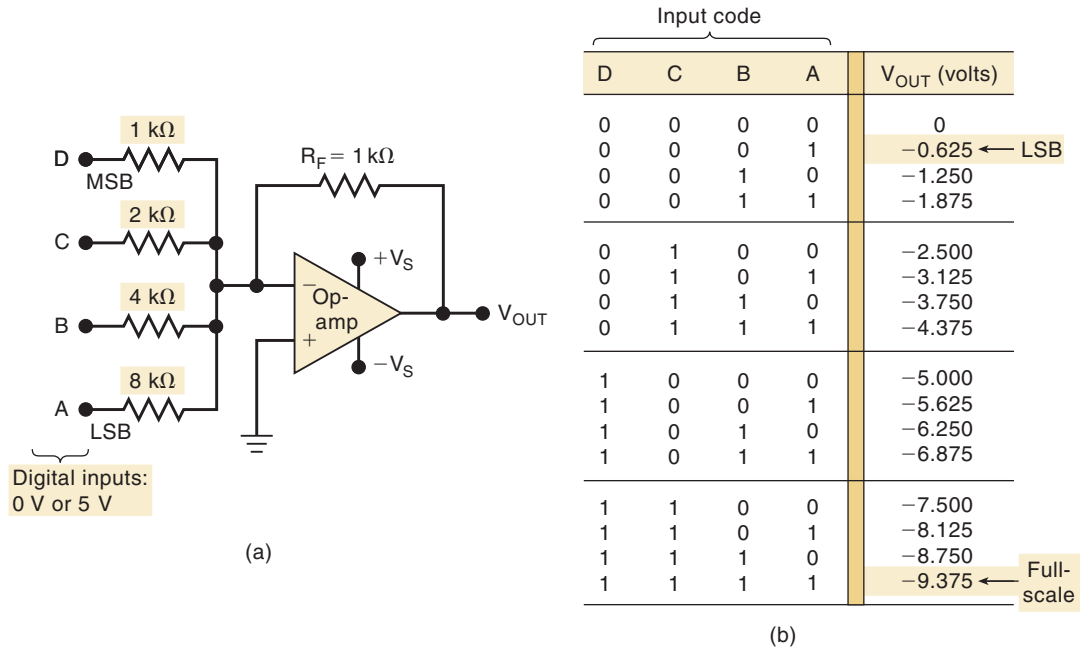
## 11-3 DAC CIRCUITRY

### OUTCOMES

Upon completion of this section, you will be able to:

- Analyze analog circuits that are used to perform digital-to-analog conversion.
- State the factors that affect conversion accuracy in a DAC.

There are several methods and circuits for producing the D/A operation that has been described. We shall examine several of the basic schemes to gain an insight into the ideas used. It is not important to be familiar with all of the various circuit schemes because D/A converters are available as



**FIGURE 11-5** Simple DAC using an op-amp summing amplifier with binary-weighted resistors.



ICs or as encapsulated packages that do not require any circuit knowledge. Instead, it is important to know the significant performance characteristics of DACs, in general, so that they can be used intelligently. These will be covered in Section 11-4.

Figure 11-5(a) shows the basic circuit for one type of four-bit DAC. The inputs  $A$ ,  $B$ ,  $C$ , and  $D$  are binary inputs that are assumed to have values of either 0 or 5 V. The *operational amplifier* is employed as a summing amplifier, which produces the weighted sum of these input voltages. Recall that the summing amplifier multiplies each input voltage by the ratio of the feedback resistor  $R_F$  to the corresponding input resistor  $R_{IN}$ . In this circuit  $R_F = 1\text{ k}\Omega$ , and the input resistors range from 1 to 8 k $\Omega$ . The  $D$  input has  $R_{IN} = 1\text{ k}\Omega$ , so the summing amplifier passes the voltage at  $D$  with no attenuation. The  $C$  input has  $R_{IN} = 2\text{ k}\Omega$ , so that it will be attenuated by  $\frac{1}{2}$ . Similarly, the  $B$  input will be attenuated by  $\frac{1}{4}$  and the  $A$  input by  $\frac{1}{8}$ . The amplifier output can thus be expressed as

$$V_{OUT} = -(V_D + \frac{1}{2} V_C + \frac{1}{4} V_B + \frac{1}{8} V_A) \quad (11-5)$$

The negative sign is present because the summing amplifier is a polarity-inverting amplifier, but it will not concern us here.

Clearly, the summing amplifier output is an analog voltage that represents a weighted sum of the digital inputs, as shown by the table in Figure 11-5(b). This table lists all of the possible input conditions and the resultant amplifier output voltage. The output is evaluated for any input condition by setting the appropriate inputs to either 0 or 5 V. For example, if the digital input is 1010, then  $V_D = V_B = 5\text{ V}$  and  $V_C = V_A = 0\text{ V}$ . Thus, using equation (11-5),

$$\begin{aligned} V_{OUT} &= -(5\text{ V} + 0\text{ V} + \frac{1}{4} \times 5\text{ V} + 0\text{ V}) \\ &= -6.25\text{ V} \end{aligned}$$

The resolution of this D/A converter is equal to the weighting of the LSB, which is  $\frac{1}{8} \times 5\text{ V} = 0.625\text{ V}$ . As shown in the table, the analog output increases by 0.625 V as the binary input number advances one step.

**EXAMPLE 11-7**

- (a) Determine the weight of each input bit of Figure 11-5(a).  
 (b) Change  $R_F$  to  $250\ \Omega$  and determine the full-scale output.

**Solution**

- (a) The MSB passes with gain = 1, so its weight in the output is 5 V. Thus,

$$\begin{aligned} \text{MSB} &\rightarrow 5\ \text{V} \\ \text{2nd MSB} &\rightarrow 2.5\ \text{V} \\ \text{3rd MSB} &\rightarrow 1.25\ \text{V} \\ \text{4th MSB} = \text{LSB} &\rightarrow 0.625\ \text{V} \end{aligned}$$

- (b) If  $R_F$  is reduced by a factor of 4, to  $250\ \Omega$ , each input weight will be four times *smaller* than the values above. Thus, the full-scale output will be reduced by this same factor and becomes  $-9.375/4 = -2.344\ \text{V}$ .

If we look at the input resistor values in Figure 11-5, it should come as no surprise that they are *binarily weighted*. In other words, starting with the MSB resistor, the resistor values increase by a factor of 2. This, of course, produces the desired weighting in the voltage output.

**Conversion Accuracy**

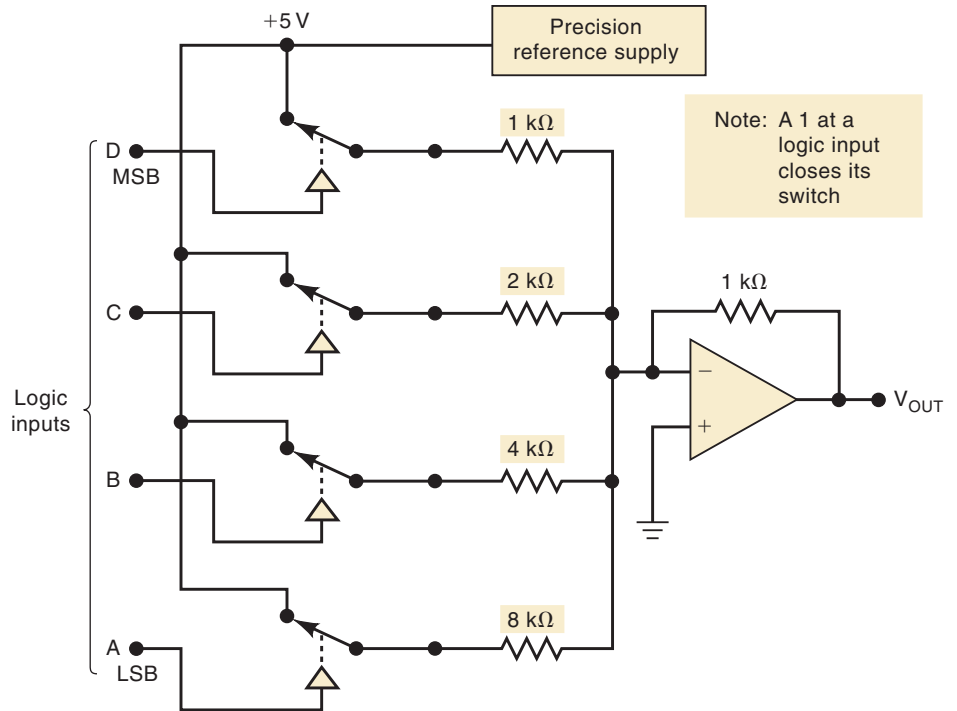
The table in Figure 11-5(b) gives the *ideal* values of  $V_{\text{OUT}}$  for the various input cases. How close the circuit comes to producing these values depends primarily on two factors: (1) the precision of the input and feedback resistors and (2) the precision of the input voltage levels. The resistors can be made very accurate (within 0.01 percent of the desired values) by trimming, but the input voltage levels must be handled differently. It should be clear that the digital inputs cannot be taken directly from the outputs of FFs or logic gates because the output logic levels of these devices are not precise values like 0 V and 5 V but vary within given ranges. For this reason, it is necessary to add some more circuitry between each digital input and its input resistor to the summing amplifier, as shown in Figure 11-6.

Each digital input controls a semiconductor switch such as the CMOS transmission gate we studied in Chapter 8. When the input is HIGH, the switch closes and connects a *precision reference supply* to the input resistor; when the input is LOW, the switch is open. The reference supply produces a very stable, precise voltage needed to generate an accurate analog output.

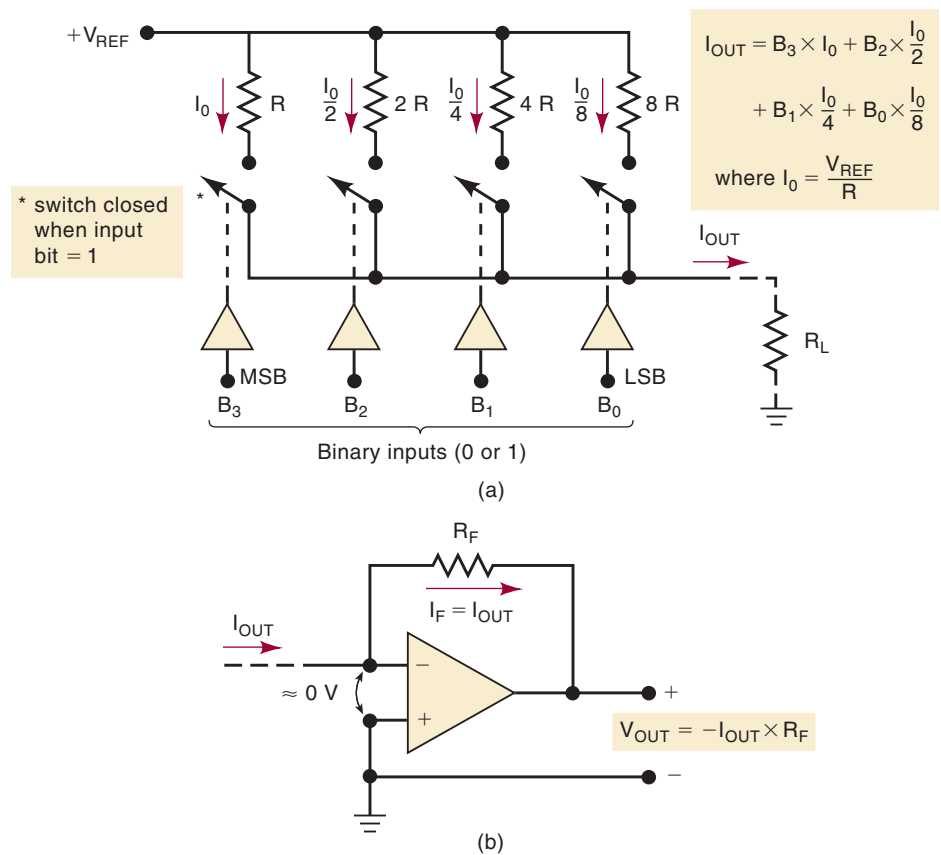
**DAC with Current Output**

Figure 11-7(a) shows one basic scheme for generating an analog output current proportional to a binary input. The circuit shown is a four-bit DAC using binarily weighted resistors. The circuit uses four parallel current paths, each controlled by a semiconductor switch such as the CMOS transmission gate. The state of each switch is controlled by logic levels at the binary inputs. The current through each path is determined by an accurate reference voltage,  $V_{\text{REF}}$ , and a precision resistor in the path. The resistors are binarily weighted so that the various currents will be binarily weighted, and the total current,  $I_{\text{OUT}}$ , will be the sum of the individual currents. The MSB path has the smallest resistor,  $R$ ; the next path has a resistor of twice the value; and so on. The output current can be made to flow through a load  $R_L$  that is much smaller than  $R$ , so that it has no effect on the value of current. Ideally,  $R_L$  should be a short to ground.

**FIGURE 11-6** Complete four-bit DAC including a precision reference supply.



**FIGURE 11-7** (a) Basic current-output DAC; (b) connected to an op-amp current-to-voltage converter.



**EXAMPLE 11-8**

Assume that  $V_{\text{REF}} = 10 \text{ V}$  and  $R = 10 \text{ k}\Omega$ . Determine the resolution and the full-scale output for this DAC. Assume that  $R_L$  is much smaller than  $R$ .

**Solution**

$I_{\text{OUT}} = V_{\text{REF}}/R = 1 \text{ mA}$ . This is the weight of the MSB. The other three currents will be 0.5, 0.25, and 0.125 mA. The LSB is 0.125 mA, which is also the resolution.

The full-scale output will occur when the binary inputs are all HIGH so that each current switch is closed and

$$I_{\text{OUT}} = 1 + 0.5 + 0.25 + 0.125 = 1.875 \text{ mA}$$

Note that the output current is proportional to  $V_{\text{REF}}$ . If  $V_{\text{REF}}$  is increased or decreased, the resolution and the full-scale output will change proportionally.

For  $I_{\text{OUT}}$  to be accurate,  $R_L$  should be a short to ground. One common way to accomplish this is to use an op-amp as a current-to-voltage converter, as shown in Figure 11-7(b). Here, the  $I_{\text{OUT}}$  from the DAC is connected to the op-amp's “-” input, which is virtually at ground. The op-amp negative feedback forces a current equal to  $I_{\text{OUT}}$  to flow through  $R_F$  to produce  $V_{\text{OUT}} = -I_{\text{OUT}} \times R_F$ . Thus,  $V_{\text{OUT}}$  will be an analog voltage that is proportional to the binary input to the DAC. This analog output can drive a wide range of loads without being loaded down.

**R/2R Ladder**

The DAC circuits we have looked at thus far use binary-weighted resistors to produce the proper weighting of each bit. Whereas this method works in theory, it has some practical limitations. The biggest problem is the large difference in resistor values between the LSB and the MSB, especially in high-resolution DACs (i.e., many bits). For example, if the MSB resistor is  $1 \text{ k}\Omega$  in a 12-bit DAC, the LSB resistor will be over  $2 \text{ M}\Omega$ . With the current IC fabrication technology, it is very difficult to produce resistance values over a wide resistance range that maintain an accurate ratio, especially with variations in temperature.

For this reason, it is preferable to have a circuit that uses resistances that are fairly close in value. One of the most widely used DAC circuits that satisfies this requirement is the *R/2R ladder* network, where the resistance values span a range of only 2 to 1. One such DAC is shown in Figure 11-8.

Note how the resistors are arranged, and especially note that only two different values are used,  $R$  and  $2R$ . The current  $I_{\text{OUT}}$  depends on the positions of the four switches, and the binary inputs  $B_3B_2B_1B_0$  control the states of the switches. This current is allowed to flow through an op-amp current-to-voltage converter to develop  $V_{\text{OUT}}$ . We will not perform a detailed analysis of this circuit here, but it can be shown that the value of  $V_{\text{OUT}}$  is given by the expression

$$V_{\text{OUT}} = \frac{-V_{\text{REF}}}{16} \times B \quad (11-6)$$

where  $B$  is the value of the binary input, which can range from 0000 (0) to 1111 (15).

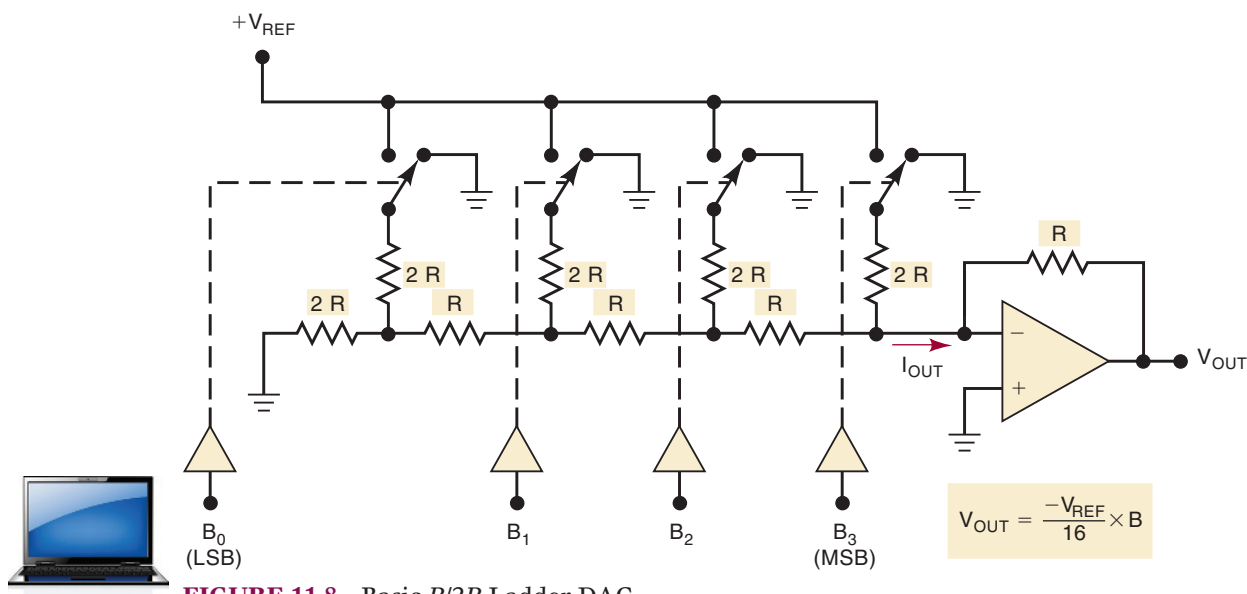


FIGURE 11-8 Basic  $R/2R$  Ladder DAC.

### EXAMPLE 11-9

Assume that  $V_{REF} = 10\text{ V}$  for the DAC in Figure 11-8. What are the resolution and full-scale output of this converter?

#### Solution

The resolution is equal to the weight of the LSB, which we can determine by setting  $B = 0001 = 1$  in equation (11-6):

$$\begin{aligned} \text{resolution} &= \frac{-10\text{ V} \times 1}{16} \\ &= -0.625\text{ V} \end{aligned}$$

The full-scale output occurs for  $B = 1111 = 15_{10}$ . Again using equation (11-6),

$$\begin{aligned} \text{full scale} &= \frac{-10\text{ V} \times 15}{16} \\ &= -9.375\text{ V} \end{aligned}$$

### OUTCOME ASSESSMENT QUESTIONS

1. What is the advantage of  $R/2R$  ladder DACs over those that use binary-weighted resistors?
2. A certain six-bit DAC uses binary-weighted resistors. If the MSB resistor is  $20\text{ k}\Omega$ , what is the LSB resistor?
3. What will the resolution be if the value of  $R_F$  in Figure 11-5 is changed to  $800\ \Omega$ ?
4. What will happen to both resolution and full-scale output when  $V_{REF}$  is increased by 20 percent?

## 11-4 DAC SPECIFICATIONS

### OUTCOMES

Upon completion of this section, you will be able to:

- Define terms associated with DACs.
- Interpret specifications given by manufacturers of DACs.

A wide variety of DACs are available as ICs or as self-contained, encapsulated packages. One should be familiar with the more important manufacturers' specifications in order to evaluate a DAC for a particular application.

### Resolution

As mentioned earlier, the percentage resolution of a DAC depends solely on the number of bits. For this reason, manufacturers usually specify a DAC resolution as the number of bits. A 10-bit DAC has a finer (smaller) resolution than an eight-bit DAC.

### Accuracy

DAC manufacturers have several ways of specifying accuracy. The two most common are called **full-scale error** and **linearity error**, which are normally expressed as a percentage of the converter's full-scale output (% F.S.).

Full-scale error is the maximum deviation of the DAC's output from its expected (ideal) value, expressed as a percentage of full scale. For example, assume that the DAC of Figure 11-5 has an accuracy of  $\pm 0.01\%$  F.S. Because this converter has a full-scale output of 9.375 V, this percentage converts to

$$\pm 0.01\% \times 9.375 \text{ V} = \pm 0.9375 \text{ mV}$$

This means that the output of this DAC can, at any time, be off by as much as 0.9375 mV from its expected value.

Linearity error is the maximum deviation in step size from the ideal step size. For example, the DAC of Figure 11-5 has an expected step size of 0.625 V. If this converter has a linearity error of  $\pm 0.01\%$  F.S., this would mean that the actual *step size* could be off by as much as 0.9375 mV.

It is important to understand that accuracy and resolution of a DAC must be compatible. It is illogical to have a resolution of, say, 1 percent and an accuracy of 0.1 percent, or vice versa. To illustrate, a DAC with a resolution of 1 percent and an F.S. output of 10 V can produce an output analog voltage within 0.1 V of any desired value, assuming perfect accuracy. It makes no sense to have a costly accuracy of 0.01% F.S. (or 1 mV) if the resolution already limits the closeness of the desired value to 0.1 V. The same can be said for having a resolution that is very small (many bits) while the accuracy is poor; it is a waste of input bits.

#### EXAMPLE 11-10

A certain eight-bit DAC has a full-scale output of 2 mA and a full-scale error of  $\pm 0.5\%$  F.S. What is the range of possible outputs for an input of 10000000?

**Solution**

The step size is  $2 \text{ mA}/255 = 7.84 \mu\text{A}$ . Since  $10000000 = 128_{10}$ , the ideal output should be  $128 \times 7.84 \mu\text{A} = 1004 \mu\text{A}$ . The error can be as much as

$$\pm 0.5\% \times 2 \text{ mA} = \pm 10 \mu\text{A}$$

Thus, the actual output can deviate by this amount from the ideal  $1004 \mu\text{A}$ , so the actual output can be anywhere from  $994$  to  $1004 \mu\text{A}$ .

**Offset Error**

Ideally, the output of a DAC will be zero volts when the binary input is all 0s. In practice, however, there will be a very small output voltage for this situation; this is called **offset error**. This offset error, if not corrected, will be added to the expected DAC output for *all* input cases. For example, let's say that a four-bit DAC has an offset error of  $+2 \text{ mV}$  and a *perfect* step size of  $100 \text{ mV}$ . Table 11-3 shows the ideal and the actual DAC output for several input cases. Note that the actual output is  $2 \text{ mV}$  greater than expected; this is due to the offset error. Offset error can be negative as well as positive.

**TABLE 11-3** Output examples demonstrating an offset of  $2 \text{ mV}$ .

Input Code	Ideal Output (mV)	Actual Output (mV)
0000	0	2
0001	100	102
1000	800	802
1111	1500	1502

Many DACs have an external offset adjustment that allows you to zero the offset. This is usually accomplished by applying all 0s to the DAC input and monitoring the output while an *offset adjustment potentiometer* is adjusted until the output is as close to  $0 \text{ V}$  as required.

**Settling Time**

The operating speed of a DAC is usually specified by giving its **settling time**, which is the time required for the DAC output to go from zero to full scale as the binary input is changed from all 0s to all 1s. Actually, the settling time is measured as the time for the DAC output to settle within  $\pm \frac{1}{2}$  step size (resolution) of its final value. For example, if a DAC has a resolution of  $10 \text{ mV}$ , settling time is measured as the time it takes the output to settle within  $5 \text{ mV}$  of its full-scale value.

Typical values for settling time range from  $50 \text{ ns}$  to  $10 \mu\text{s}$ . Generally speaking, DACs with a current output will have shorter settling times than those with voltage outputs. The main reason for this difference is the response time of the op-amp that is used as the current-to-voltage converter.

**Monotonicity**

A DAC is **monotonic** if its output increases as the binary input is incremented from one value to the next. Another way to describe this is that the staircase output will have no downward steps as the binary input is incremented from zero to full scale.



**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Define *full-scale error*.
2. What is *settling time*?
3. Describe offset error and its effect on a DAC output.
4. Why are voltage DACs generally slower than current DACs?

## 11-5 AN INTEGRATED-CIRCUIT DAC

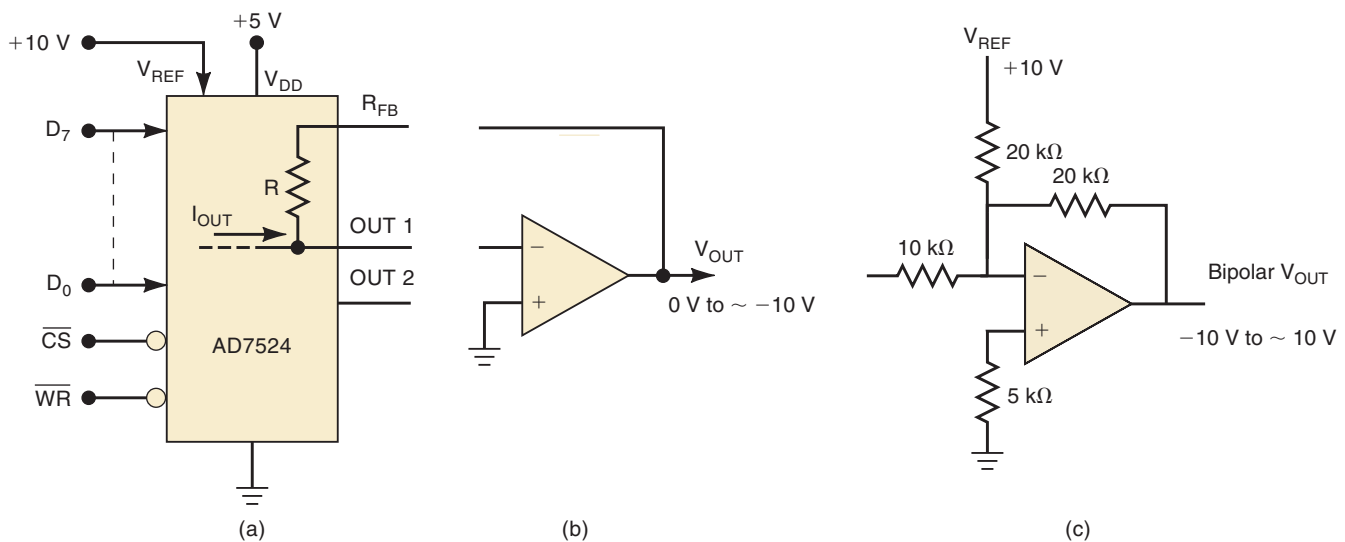
### OUTCOMES

Upon completion of this section, you will be able to:

- Use an AD7524 eight-bit DAC in a digital system.
- Describe the role of each input.
- State the limitations of the AD7524 DAC.

The AD7524, a CMOS IC available from several IC manufacturers, is an eight-bit D/A converter that uses an  $R/2R$  ladder network. Its block symbol is given in Figure 11-9(a). This DAC has an eight-bit input that can be latched internally under the control of the Chip Select ( $\overline{CS}$ ) and WRITE ( $\overline{WR}$ ) inputs. When both of these control inputs are LOW, the digital data inputs  $D_7$ – $D_0$  produce the analog output current  $I_{OUT}$  (the  $OUT\ 2$  terminal is normally grounded). When either control input goes HIGH, the digital input data are latched, and the analog output remains at the level corresponding to that latched digital data. Subsequent changes in the digital inputs will have no effect on  $OUT\ 1$  in this latched state.

The maximum settling time for the AD7524 is typically 100 ns, and its full-range accuracy is rated at  $\pm 0.2\%$  F.S. The  $V_{REF}$  can range over both negative and positive voltages from 0 to 25 V, so that analog output currents of both polarities can be produced. The output current can be converted to a voltage using an op-amp connected as in Figure 11-9(b). Note that the op-amp's



**FIGURE 11-9** (a) AD7524 eight-bit DAC with latched inputs; (b) op-amp current-to-voltage converter provides 0 to approximately -10 V out; (c) op-amp circuit to produce bipolar output from -10 V to approximately +10 V.

feedback resistor is already on the DAC chip. The op-amp circuit shown in Figure 11-9(c) can be added to produce a bipolar output that ranges from  $-V_{REF}$  (when input = 00000000) to almost  $+V_{REF}$  (when input = 11111111).

### OUTCOME ASSESSMENT QUESTIONS

1. Can the AD7524 be connected to a tristate bus?
2. What is the role of the WR line on the AD7524?
3. List three limitations of the AD7524.

## 11-6 DAC APPLICATIONS

### OUTCOME

Upon completion of this section, you will be able to:

- Identify typical applications of DACs.

DACs are used whenever the output of a digital circuit must provide an analog voltage or current to drive an analog device. Some of the most common applications are described in the following paragraphs.

### Control

The digital output from a computer can be converted to an analog control signal to adjust the speed of a motor or the temperature of a furnace, or to control almost any physical variable.

### Automatic Testing

Computers can be programmed to generate the analog signals (through a DAC) needed to test analog circuitry. The test circuit's analog output response will normally be converted to a digital value by an ADC and fed into the computer to be stored, displayed, and sometimes analyzed.

### Signal Reconstruction

In many applications, an analog signal is **digitized**; that is, successive points on the signal are converted to their digital equivalents and stored in memory. This conversion is performed by an analog-to-digital converter (ADC). A DAC can then be used to convert the stored digitized data back to analog—one point at a time—thereby reconstructing the original signal. This combination of digitizing and reconstructing is used in audio compact disk systems and digital audio and video recording. We will look at this further after we learn about ADCs.

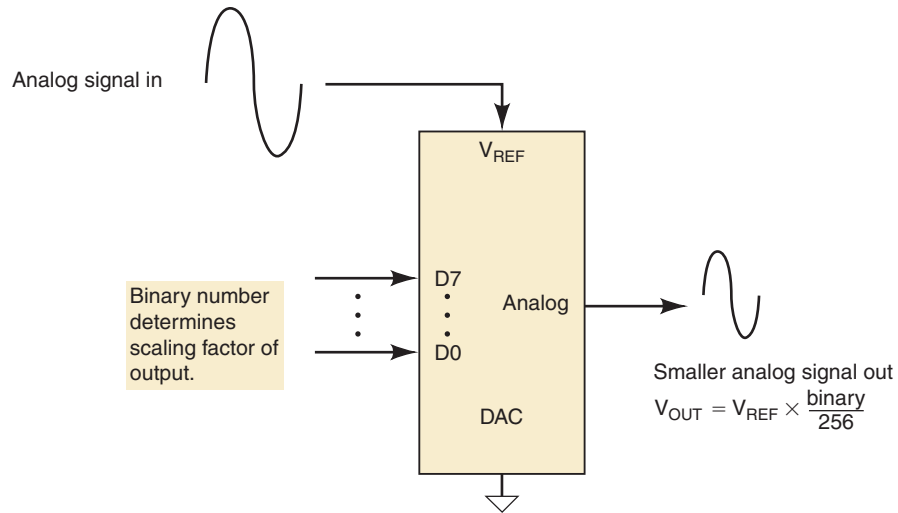
### A/D Conversion

Several types of ADCs use DACs as part of their circuitry, as we shall see in Section 11-8.

### Digital Amplitude Control

DACs can also be used to reduce the amplitude of an analog signal by connecting the analog signal to the  $V_{REF}$  input as shown in Figure 11-10. The binary input scales the signal on  $V_{REF}$ :  $V_{OUT} = V_{REF} \times \text{binary}/2^N$ . When the maximum binary input value is applied, the output is nearly the same as the  $V_{REF}$

**FIGURE 11-10** A DAC used to control the amplitude of an analog signal.



input. However, when a value that represents about half of the maximum (e.g.,  $10000000_2$  for a unipolar eight-bit converter) is applied to the inputs, the output is about half of  $V_{REF}$ . If  $V_{REF}$  is a signal (e.g., a sine wave) that varies within the range of the reference voltage, the output will be the same fully analog wave shape whose amplitude depends on the digital number applied to the DAC. In this way a digital system can control things such as the volume of an audio system or the amplitude of a function generator.

### Serial DACs

Many of these DAC applications involve a microprocessor. The main problem with using the parallel-data DACs that have been described thus far is that they occupy so many port bits of the microcomputer. Even though the data transfer speed is somewhat slower, a microprocessor can output the digital value to a DAC over a serial interface for applications where speed is not a major requirement. Serial DACs are now readily available with a built-in serial in/parallel out shift register. Many of these devices have more than one DAC on the same chip. The digital data, along with a code that specifies which DAC you want, are sent to the chip, one bit at a time. As each bit is presented on the DAC input, a pulse is applied to the serial clock input to shift the bit in. After the proper number of clock pulses, the data value is latched and converted to its analog value.

#### OUTCOME ASSESSMENT QUESTION

1. List three applications of DACs.

## 11-7 TROUBLESHOOTING DACs

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Identify common failure modes and inaccuracies associated with DACs.
- Test a DAC.

DACs are both digital and analog. Logic probes and pulsers can be used on the digital inputs, but a meter or an oscilloscope must be used on the analog output. There are basically two ways to test a DAC's operation: a *static accuracy test* and a *staircase test*.

The static test involves setting the binary input to a fixed value and measuring the analog output with a high-accuracy meter. This test is used to check that the output value falls within the expected range consistent with the DAC's specified accuracy. If it does not, there can be several possible causes. Here are some of them:

- Drift in the DAC's internal component values (e.g., resistor values) caused by temperature, aging, or some other factors. This condition can easily produce output values outside the expected accuracy range.
- Open connections or shorts in any of the binary inputs. This could either prevent an input from adding its weight to the analog output or cause its weight to be permanently present in the output. This situation is especially hard to detect when the fault is in the less significant inputs.
- A faulty voltage reference. Because the analog output depends directly on  $V_{REF}$ , this could produce results that are way off. For DACs that use external reference sources, the reference voltage can be checked easily with a digital voltmeter. Many DACs have internal reference voltages that cannot be checked, except on some DACs that bring the reference voltage out to a pin of the IC.
- Excessive offset error caused by component aging or temperature. This would produce outputs that are off by a fixed amount. If the DAC has an external offset adjustment capability, this type of error can initially be zeroed out, but changes in operating temperature can cause the offset error to reappear.

The staircase test is used to check the monotonicity of the DAC; that is, it checks to see that the output increases step by step as the binary input is incremented as in Figure 11-3. The steps on the staircase must be of the same size, and there should be no missing steps or downward steps until full scale is reached. This test can help detect internal or external faults that cause an input to have either no contribution or a permanent contribution to the analog output. The following example will illustrate.

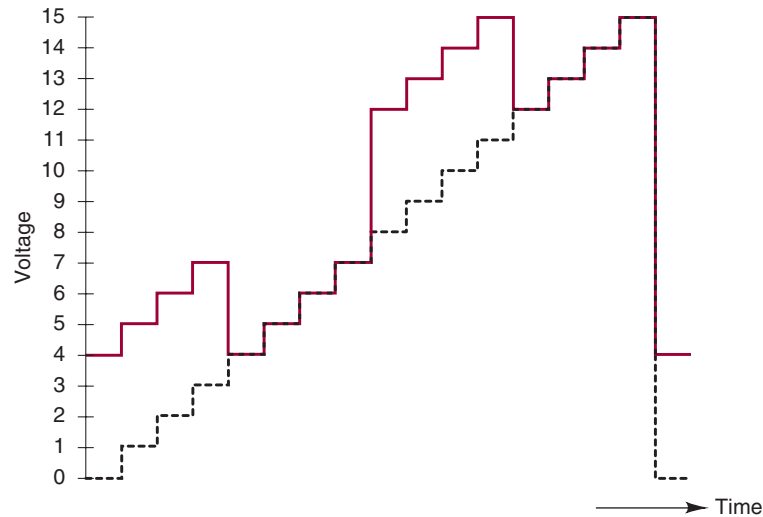
#### EXAMPLE 11-11

How would the staircase waveform appear if the  $C$  input to the DAC of Figure 11-3 is open? Assume that the DAC inputs are TTL-compatible.

#### Solution

An open connection at  $C$  will be interpreted as a constant logic 1 by the DAC. Thus, this will contribute a constant 4 V to the DAC output so that the DAC output waveform will appear as shown in Figure 11-11. The dotted lines are the staircase as it would appear if the DAC were working correctly. Note that the faulty output waveform matches the correct one during those times when the bit  $C$  input would normally be HIGH.

**FIGURE 11-11** Example 11-11.



**OUTCOME  
ASSESSMENT  
QUESTION**

1. List three common failure modes associated with DACs.

## 11-8 ANALOG-TO-DIGITAL CONVERSION

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe the role of a DAC in an analog-to-digital converter.
- Describe a general model of how an ADC works.

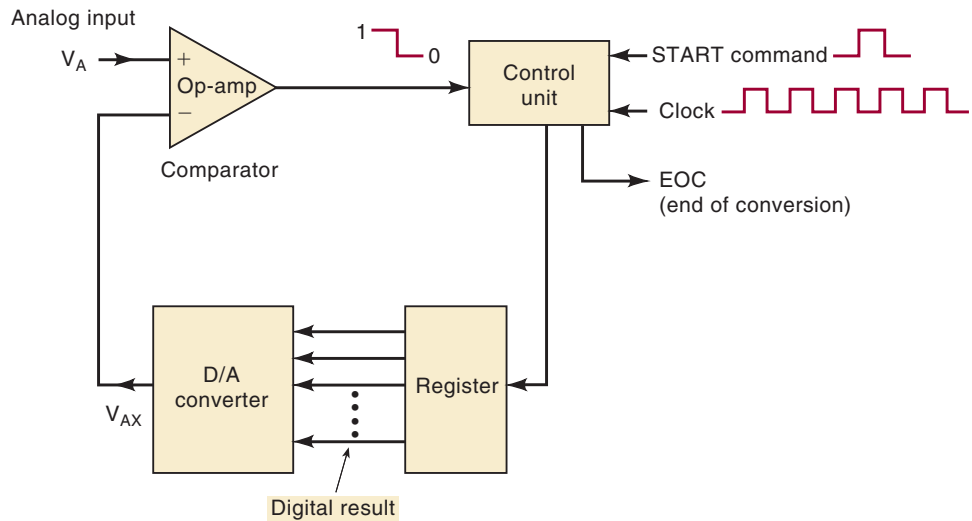
An analog-to-digital converter takes an analog input voltage and, after a certain amount of time, produces a digital output code that represents the analog input. The A/D conversion process is generally more complex and time-consuming than the D/A process, and many different methods have been developed and used. We shall examine several of these methods in detail, even though it may never be necessary to design or construct ADCs (they are available as completely packaged units). However, the techniques that are used provide an insight into what factors determine an ADC's performance.

Several important types of ADCs utilize a DAC as part of their circuitry. Figure 11-12 is a general block diagram for this class of ADC. The timing for the operation is provided by the input clock signal. The control unit contains the logic circuitry for generating the proper sequence of operations in response to the START command, which initiates the conversion process. The op-amp comparator has two *analog* inputs and a *digital* output that switches states, depending on which analog input is greater.

The basic operation of ADCs of this type consists of the following steps:

1. The START command pulse initiates the operation.
2. At a rate determined by the clock, the control unit continually modifies the binary number that is stored in the register.

**FIGURE 11-12** General diagram of one class of ADCs.



3. The binary number in the register is converted to an analog voltage,  $V_{AX}$ , by the DAC.
4. The comparator compares  $V_{AX}$  with the analog input  $V_A$ . As long as  $V_{AX} < V_A$ , the comparator output stays HIGH. When  $V_{AX}$  exceeds  $V_A$  by at least an amount equal to  $V_T$  (threshold voltage), the comparator output goes LOW and stops the process of modifying the register number. At this point,  $V_{AX}$  is a close approximation to  $V_A$ . The digital number in the register, which is the digital equivalent of  $V_{AX}$ , is also the approximate digital equivalent of  $V_A$ , within the resolution and accuracy of the system.
5. The control logic activates the end-of-conversion signal,  $EOC$ , when the conversion is complete.

Several variations of this A/D conversion scheme differ mainly in the manner in which the control section continually modifies the numbers in the register. Otherwise, the basic idea is the same, with the register holding the required digital output when the conversion process is complete.

### OUTCOME ASSESSMENT QUESTIONS

1. What is the function of the comparator in the ADC?
2. Where is the approximate digital equivalent of  $V_A$  when the conversion is complete?
3. What is the function of the  $EOC$  signal?

## 11-9 DIGITAL-RAMP ADC

### OUTCOMES

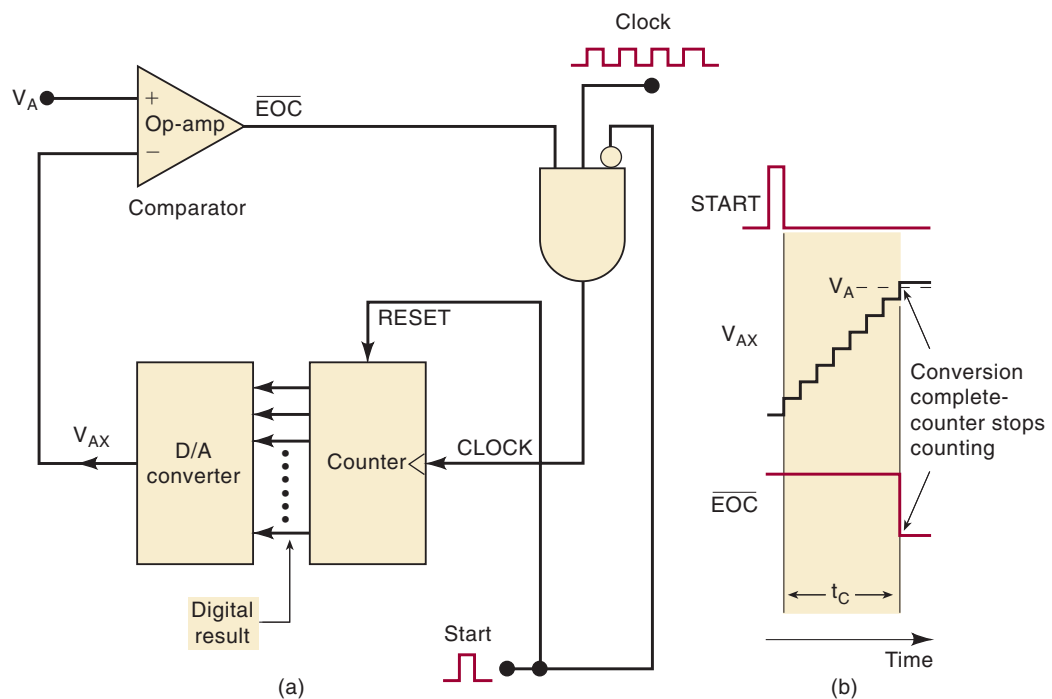
Upon completion of this section, you will be able to:

- Describe the operation of a digital-ramp ADC.
- Define quantization error.
- Determine and compare resolution and accuracy of ADCs.

One of the simplest versions of the general ADC of Figure 11-12 uses a binary counter as the register and allows the clock to increment the counter one step at a time until  $V_{AX} \geq V_A$ . It is called a **digital-ramp ADC** because the waveform at  $V_{AX}$  is a step-by-step ramp (actually a staircase) like the one shown in Figure 11-3. It is also referred to as a *counter-type ADC*.

Figure 11-13 is the diagram for a digital-ramp ADC. It contains a counter, a DAC, an analog comparator, and a control AND gate. The comparator output serves as the active-LOW end-of-conversion signal  $\overline{EOC}$ . If we assume that  $V_A$ , the analog voltage to be converted, is positive, the operation proceeds as follows:

1. A START pulse is applied to reset the counter to 0. The HIGH at START also inhibits clock pulses from passing through the AND gate into the counter.
2. With all 0s at its input, the DAC's output will be  $V_{AX} = 0V$ .
3. Because  $V_A > V_{AX}$ , the comparator output,  $\overline{EOC}$ , will be HIGH.
4. When START returns LOW, the AND gate is enabled and clock pulses get through to the counter.
5. As the counter advances, the DAC output,  $V_{AX}$ , increases one step at a time, as shown in Figure 11-13(b).
6. This process continues until  $V_{AX}$  reaches a step that exceeds  $V_A$  by an amount equal to or greater than  $V_T$  (typically 10 to 100  $\mu V$ ). At this point,  $\overline{EOC}$  will go LOW and inhibit the flow of pulses into the counter, and the counter will stop counting.
7. The conversion process is now complete, as signaled by the HIGH-to-LOW transition at  $\overline{EOC}$ , and the contents of the counter are the digital representation of  $V_A$ .
8. The counter will hold the digital value until the next START pulse initiates a new conversion.



**FIGURE 11-13** Digital-ramp ADC.

**EXAMPLE 11-12**

Assume the following values for the ADC of Figure 11-13: clock frequency = 1 MHz;  $V_T = 0.1$  mV; DAC has F.S. output = 10.23 V and a 10-bit input. Determine the following values.

- The digital equivalent obtained for  $V_A = 3.728$  V
- The conversion time
- The resolution of this converter

**Solution**

- (a) The DAC has a 10-bit input and a 10.23-V F.S. output. Thus, the number of total possible steps is  $2^{10} - 1 = 1023$ , and so the step size is

$$\frac{10.23 \text{ V}}{1023} = 10 \text{ mV}$$

This means that  $V_{AX}$  increases in steps of 10 mV as the counter counts up from 0. Because  $V_A = 3.728$  V and  $V_T = 0.1$  mV,  $V_{AX}$  must reach 3.7281 V or more before the comparator switches LOW. This will require

$$\frac{3.7281 \text{ V}}{10 \text{ mV}} = 372.81 = 373 \text{ steps}$$

At the end of the conversion, then, the counter will hold the binary equivalent of 373, which is 0101110101. This is the desired digital equivalent of  $V_A = 3.728$  V, as produced by this ADC.

- Three hundred seventy-three steps were required to complete the conversion. Thus, 373 clock pulses occurred at the rate of one per microsecond. This gives a total conversion time of 373  $\mu$ s.
- The resolution of this converter is equal to the step size of the DAC, which is 10 mV. Expressed as a percentage, it is  $1/1023 \times 100\% \approx 0.1\%$ .

**EXAMPLE 11-13**

For the same ADC of Example 11-12, determine the approximate range of analog input voltages that will produce the same digital result of  $0101110101_2 = 373_{10}$ .

**TABLE 11-4** Ideal output voltages at several steps.

Step	$V_{AX}$ (V)
371	3.71
372	3.72
373	3.73
374	3.74
375	3.75

**Solution**

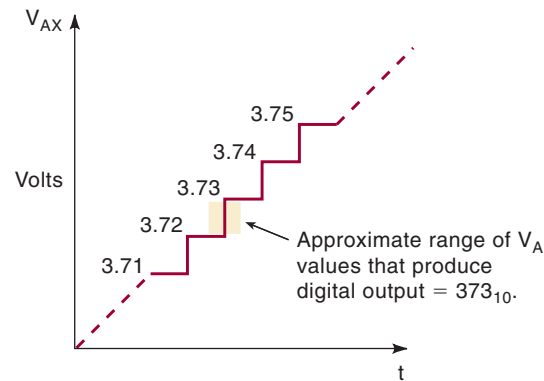
Table 11-4 shows the ideal DAC output voltage,  $V_{AX}$ , for several of the steps on and around the 373rd. If  $V_A$  is slightly smaller than 3.72 V (by an amount  $< V_T$ ), then  $\overline{EOC}$  won't go LOW when  $V_{AX}$  reaches the 3.72-V step, but it will go LOW on the 3.73-V step. If  $V_A$  is slightly smaller than 3.73 V (by an amount  $< V_T$ ), then  $\overline{EOC}$  won't go LOW until  $V_{AX}$  reaches the 3.74-V step. Thus, as long as  $V_A$  is between approximately 3.72 and 3.73 V,  $\overline{EOC}$  will go LOW when  $V_{AX}$  reaches the 3.73-V step. The exact range of  $V_A$  values is

$$3.72 \text{ V} - V_T \quad \text{to} \quad 3.73 \text{ V} - V_T$$

but because  $V_T$  is so small, we can simply say that the range is approximately 3.72 to 3.73 V—a range equal to 10 mV, the DAC's resolution. This is illustrated in Figure 11-14.



**FIGURE 11-14**  
Example 11-13.



### A/D Resolution and Accuracy

It is very important to understand the errors associated with making any kind of measurements. An unavoidable source of error in the digital-ramp method is that the step size or resolution of the internal DAC is the smallest unit of measure. Imagine trying to measure basketball players' heights by standing them next to a staircase with 12-in steps and assigning them the height of the first step higher than their head. Anyone over 6 ft would be measured as 7 ft tall! Likewise, the output voltage  $V_{AX}$  is a staircase waveform that goes up in discrete steps until it exceeds the input voltage,  $V_A$ . By making the step size smaller, we can reduce the potential error, but there will always be a difference between the actual (analog) quantity and the digital value assigned to it. This is called **quantization error**. Thus,  $V_{AX}$  is an approximation to the value of  $V_A$ , and the best we can expect is that  $V_{AX}$  is within 10 mV of  $V_A$  if the resolution (step size) is 10 mV. This quantization error, which can be reduced by increasing the number of bits in the counter and the DAC, is sometimes specified as an error of  $\pm 1$  LSB, indicating that the result could be off by as much as the weight of the LSB.

A more common practice is to make the quantization error symmetrical around an integer multiple of the resolution to make the quantization error  $\pm \frac{1}{2}$  LSB. This is done by making sure the output changes at  $\frac{1}{2}$  resolution unit below and above the nominal input voltage. For example, if the resolution is 10 mV, then the A/D output will ideally switch from 0 to 1 at 5 mV and from 1 to 2 at 15 mV. The nominal value (10 mV), which is represented by the digital value of 1, is ideally always within 5 mV ( $\frac{1}{2}$  LSB) of the actual input voltage. Problem 11-28 explores a method to accomplish this. In any case, there is a small range of input voltages that will produce the same digital output.

The accuracy specification reflects the fact that the output of every ADC does not switch from one binary value to the next at the exact prescribed input voltage. Some change at slightly higher voltage than expected, and some at slightly lower. The inaccuracy and inconsistency is due to imperfect components such as precision resistors, comparators, or current switches. Accuracy can be expressed as % full scale, just as for the DAC, but it is more commonly specified as  $\pm n$  LSB, where  $n$  is a fractional value or 1. For example, if the accuracy is specified as  $\pm \frac{1}{4}$  LSB with a resolution of 10 mV, and assuming the output should ideally switch from 0 to 1 at 5 mV, then we know that the output could change from 0 to 1 at any input voltage between 2.5 and 7.5 mV. In this case, we would be assured that any voltage between 7.5 and 12.5 mV would definitely produce the value 1. However, in the worst case, the output of binary 1 could be representing a nominal value of 10 mV

with an actual applied voltage of 2.5 mV, an error of  $\frac{3}{4}$  bit which is the sum of the quantization error and the accuracy.

**EXAMPLE 11-14**

A certain eight-bit ADC, similar to Figure 11-13, has a full-scale input of 2.55 V (i.e.,  $V_A = 2.55$  V produces a digital output of 11111111). It has a specified error of  $\pm\frac{1}{4}$  LSB. Determine the maximum amount of error in the measurement.

**Solution**

The step size is  $2.55 \text{ V} / (2^8 - 1)$ , which is exactly 10 mV. This means that even if the DAC has no inaccuracies, the  $V_{AX}$  output could be off by as much as 10 mV because  $V_{AX}$  can change only in 10-mV steps; this is the quantization error. The specified error of  $\pm\frac{1}{4}$  LSB is  $\pm\frac{1}{4} \times 10 \text{ mV} = 2.5 \text{ mV}$ . This means that the  $V_{AX}$  value can be off by as much as 2.5 mV because of component inaccuracies. Thus, the total possible error could be as much as  $10 \text{ mV} + 2.5 \text{ mV} = 12.5 \text{ mV}$ .

For example, suppose that the analog input was 1.268 V. If the DAC output were perfectly accurate, the staircase would stop at the 127th step (1.27 V). But let's say that  $V_{AX}$  was off by  $-2 \text{ mV}$ , so it was 1.268 V at the 127th step. This would not be large enough to stop the conversion; it would stop at the 128th step. Thus, the digital output would be  $10000000_2 = 128_{10}$  (representing 1.28 V) for an analog input of 1.268 V, an error of 12 mV.

**Conversion Time,  $t_C$** 

The conversion time is shown in Figure 11-13(b) as the time interval between the end of the START pulse and the activation of the  $\overline{EOC}$  output. The counter starts counting from 0 and counts up until  $V_{AX}$  exceeds  $V_A$ , at which point  $\overline{EOC}$  goes LOW to end the conversion process. It should be clear that the value of the conversion time,  $t_C$ , depends on  $V_A$ . A larger value will require more steps before the staircase voltage exceeds  $V_A$ .

The maximum conversion time will occur when  $V_A$  is just below full scale so that  $V_{AX}$  must go to the last step to activate  $\overline{EOC}$ . For an  $N$ -bit converter, this will be

$$t_C(\text{max}) = (2^N - 1) \text{ clock cycles}$$

For example, the ADC in Example 11-12 would have a maximum conversion time of

$$t_C(\text{max}) = (2^{10} - 1) \times 1 \mu\text{s} = 1023 \mu\text{s}$$

Sometimes, average conversion time is specified; it is half of the maximum conversion time. For the digital-ramp converter, this would be

$$t_C(\text{avg}) = \frac{t_C(\text{max})}{2} \approx 2^{N-1} \text{ clock cycles}$$

The major disadvantage of the digital-ramp method is that conversion time essentially doubles for each bit that is added to the counter, so that resolution can be improved only at the cost of a longer  $t_C$ . This makes this type of ADC unsuitable for applications where repetitive A/D conversions

of a fast-changing analog signal must be made. For low-speed applications, however, the relative simplicity of the digital-ramp converter is an advantage over the more complex, higher-speed ADCs.

### EXAMPLE 11-15

What will happen to the operation of a digital-ramp ADC if the analog input  $V_A$  is greater than the full-scale value?

#### Solution

From Figure 11-13, it should be clear that the comparator output will never go LOW because the staircase voltage can never exceed  $V_A$ . Thus, pulses will be continually applied to the counter, so that the counter will repetitively count up from 0 to maximum, recycle back to 0, count up, and so on. This will produce repetitive staircase waveforms at  $V_{AX}$  going from 0 to full scale, and this will continue until  $V_A$  is decreased below full scale.

### OUTCOME ASSESSMENT QUESTIONS

1. Describe the basic operation of the digital-ramp ADC.
2. Explain *quantization error*.
3. Why does conversion time increase with the value of the analog input voltage?
4. *True or false:* Everything else being equal, a 10-bit digital-ramp ADC will have a better resolution, but a longer conversion time, than an eight-bit ADC.
5. Give one advantage and one disadvantage of a digital-ramp ADC.
6. For the converter of Example 11-12, determine the digital output for  $V_A = 1.345\text{ V}$ . Repeat for  $V_A = 1.342\text{ V}$ .

## 11-10 DATA ACQUISITION

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe typical data acquisition techniques.
- State the limitations of signal reconstruction using A/D and D/A.
- Describe aliasing.
- Determine the alias frequency given the sampling frequency and analog signal frequency.
- Determine minimum sampling frequency to avoid aliasing.

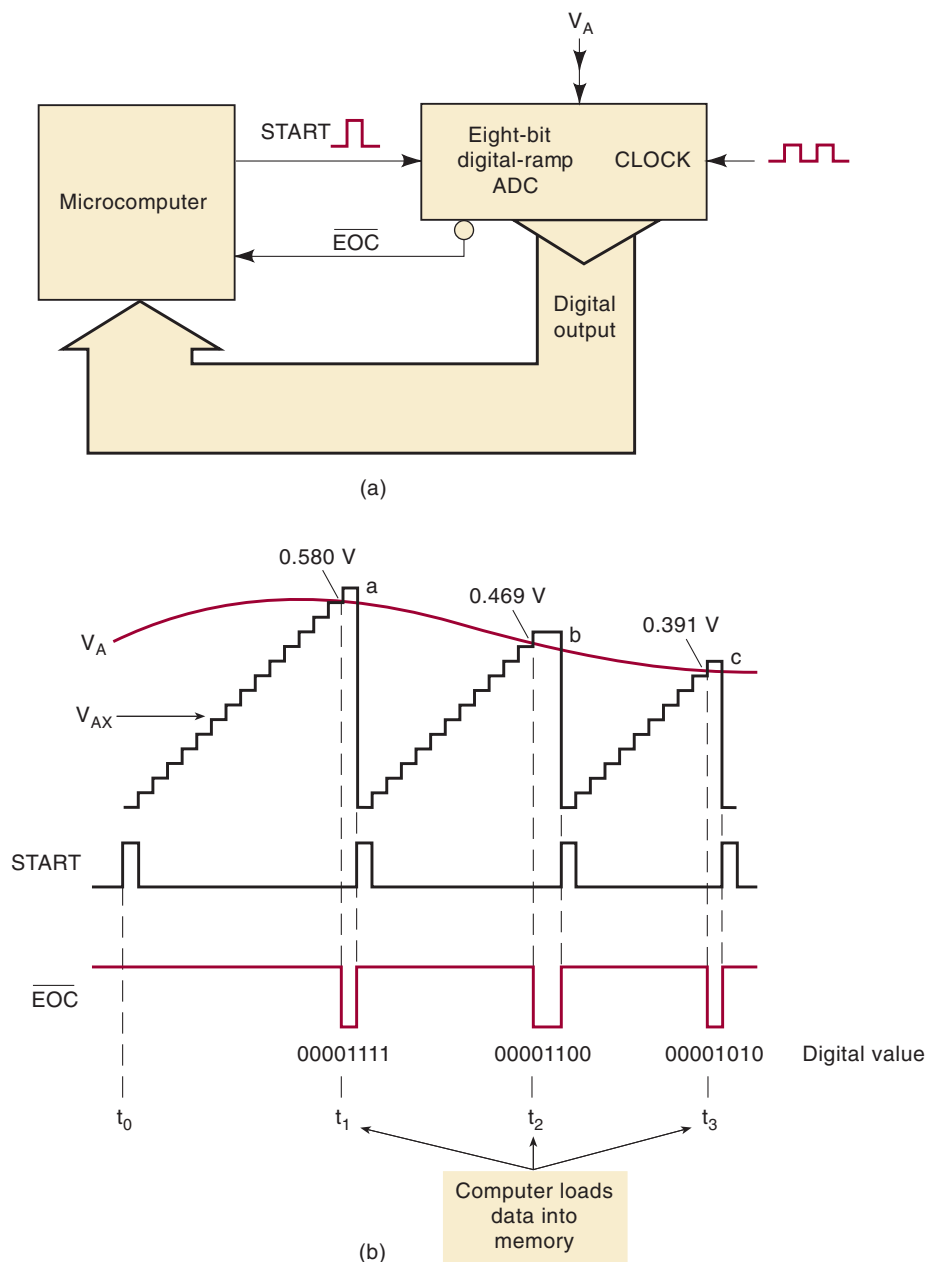
There are many applications in which analog data must be *digitized* (converted to digital) and transferred into a computer's memory. The process by which the computer acquires these digitized analog data is referred to as *data acquisition*. Acquiring a single data point's value is referred to as **sampling** the analog signal, and that data point is often called a *sample*. The computer can do several different things with the data, depending on the application. In a storage application, such as digital audio recording or video

recording, the internal microcomputer will store the data and then transfer them to a DAC at a later time to reproduce the original analog signal. In a process control application, the computer can examine the data or perform computations on them to determine what control outputs to generate.

Figure 11-15(a) shows how a microcomputer is connected to a digital-ramp ADC for the purpose of data acquisition. The computer generates the START pulses that initiate each new A/D conversion. The  $\overline{EOC}$  (end-of-conversion) signal from the ADC is fed to the computer. The computer monitors  $\overline{EOC}$  to find out when the current A/D conversion is complete; then it transfers the digital data from the ADC output into its memory.

The waveforms in Figure 11-15(b) illustrate how the computer acquires a digital version of the analog signal,  $V_A$ . The  $V_{AX}$  staircase waveform that is generated internal to the ADC is shown superimposed on the  $V_A$  waveform for purposes of illustration. The process begins at  $t_0$ , when the computer

**FIGURE 11-15** (a) Typical computer data acquisition system; (b) waveforms showing how the computer initiates each new conversion cycle and then loads the digital data into memory at the end of conversion.



generates a START pulse to start an A/D conversion cycle. The conversion is completed at  $t_1$ , when the staircase first exceeds  $V_A$ , and  $\overline{EOC}$  goes LOW. This NGT at  $\overline{EOC}$  signals the computer that the ADC has a digital output that now represents the value of  $V_A$  at point  $a$ , and the computer will load these data into its memory.

The computer generates a new START pulse shortly after  $t_1$  to initiate a second conversion cycle. Note that this resets the staircase to 0 and  $\overline{EOC}$  back HIGH because the START pulse resets the counter in the ADC. The second conversion ends at  $t_2$  when the staircase again exceeds  $V_A$ . The computer then loads the digital data corresponding to point  $b$  into its memory. These steps are repeated at  $t_3$ ,  $t_4$ , and so on.

The process whereby the computer generates a START pulse, monitors  $\overline{EOC}$ , and loads ADC data into memory is done under the control of a program that the computer is executing. This data acquisition program will determine how many data points from the analog signal will be stored in the computer memory.

### Reconstructing a Digitized Signal

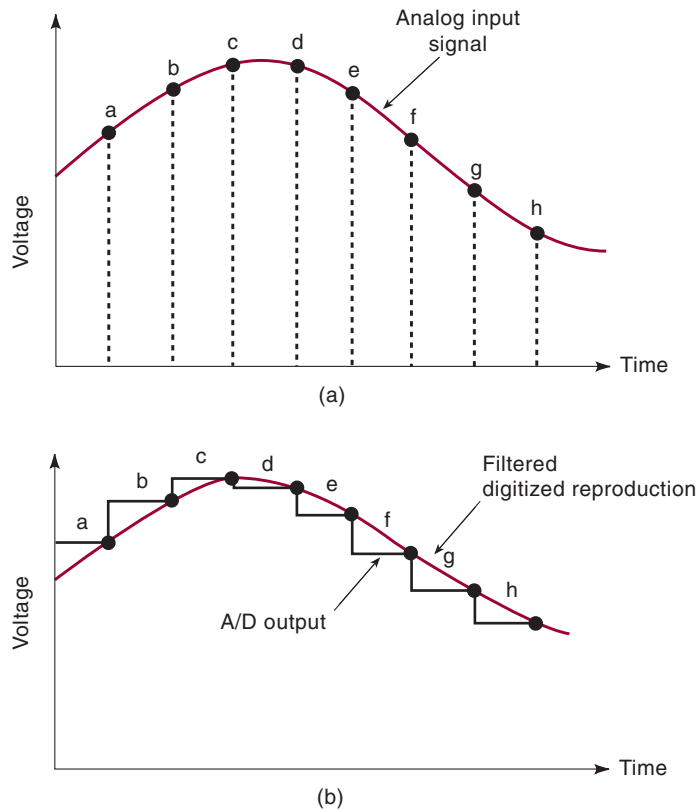
In Figure 11-15(b), the ADC is operating at its maximum speed because a new START pulse is generated immediately after the computer acquires the ADC output data from the previous conversion. Note that the conversion times are not constant because the analog input value is changing. The problem with this method of storing a waveform is that in order to reconstruct the waveform, we would need to know the point in time that each data value is to be plotted. Normally, when storing a digitized waveform, the samples are taken at fixed intervals at a rate that is at least two times greater than the highest frequency in the analog signal. The digital system will store the waveform as a list of sample data values. Table 11-5 shows the list of data that would be stored if the signal in Figure 11-16(a) were digitized.

In Figure 11-16(a), we see how the ADC continually performs conversions to digitize the analog signal at points  $a$ ,  $b$ ,  $c$ ,  $d$ , and so on. If these digital data are used to reconstruct the signal, the result will look like that in Figure 11-16(b). The black line represents the voltage waveform that would actually come out of the D/A converter. The red line would be the result of passing the signal through a simple low-pass RC filter. We can see that it is a fairly good reproduction of the original analog signal because the analog signal does not make any rapid changes between digitized points. If the analog signal contained higher-frequency variations, the ADC would not be able to follow the variations, and the reproduced version would be much less accurate.

**TABLE 11-5** Digitized data samples.

Point	Actual Voltage (V)	Digital Equivalent
$a$	1.22	01111010
$b$	1.47	10010011
$c$	1.74	10101110
$d$	1.70	10101010
$e$	1.35	10000111
$f$	1.12	01110000
$g$	0.91	01011011
$h$	0.82	01010010

**FIGURE 11-16**  
 (a) Digitizing an analog signal; (b) reconstructing the signal from the digital data.

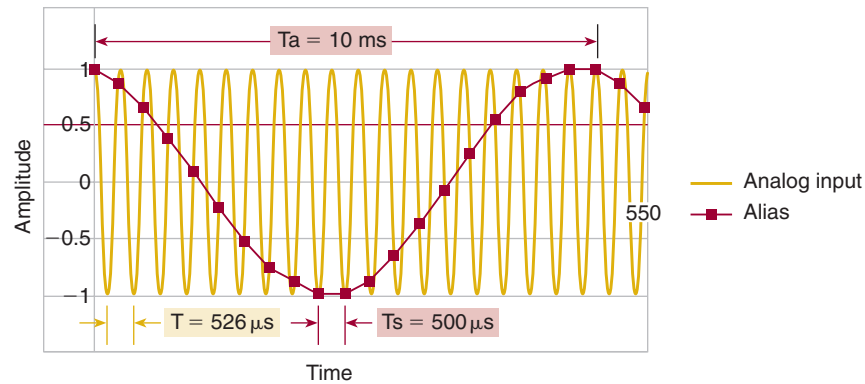


## Aliasing

The obvious goal in signal reconstruction is to make the reconstruction nearly identical to the original analog signal. In order to avoid loss of information, as has been proven by a man named Harry Nyquist, the incoming signal must be sampled at a rate greater than two times the highest-frequency component in the incoming signal. For example, if you are pretty sure that the highest frequency in an audio system will be less than 10 kHz, you must sample the audio signal at 20,000 samples per second in order to be able to reconstruct the signal. The frequency at which samples are taken is referred to as the **sampling frequency**,  $F_S$ . What do you think would happen if for some reason a 12-kHz tone is present in the input signal? Unfortunately, the system would *not* simply ignore it because it is too high! Rather, a phenomenon called *aliasing* would occur. A signal **alias** is produced by sampling the signal at a rate less than the minimum rate identified by Nyquist (twice the highest incoming frequency). In this case, any frequency over 10 kHz will produce an alias frequency. The alias frequency is always the difference between any integer multiple of the sampling frequency  $F_S$  (20 kHz) and the incoming frequency that is being digitized (12 kHz). Instead of hearing a 12-kHz tone in the reconstructed signal, you would hear an 8-kHz tone that was not in the original signal.

To see how aliasing can happen, consider the sine wave in Figure 11-17. Its frequency is 1.9 kHz. The dots show where the waveform is sampled every  $500 \mu\text{s}$  ( $F_S = 2 \text{ kHz}$ ). If we connect the dots that make up the sampled waveform, we discover that they form a cosine wave that has a period of 10 ms and a frequency of 100 Hz. This demonstrates that the alias frequency is equal to the difference between the sample frequency and the incoming frequency. If we could hear the output that results from this data acquisition, it would not sound like 1.9 kHz; it would sound like 100 Hz.

**FIGURE 11-17** An alias signal due to undersampling.



The problem with **undersampling** ( $F_S < 2F_{in \text{ max}}$ ) is that the digital system has no idea that there was actually a higher frequency at the input. It simply samples the input and stores the data. When it reconstructs the signal, the alias frequency (100 Hz) is present, the original 1.9 kHz is missing, and the reconstructed signal does not sound the same. This is why a data acquisition system must not allow frequencies greater than half of  $F_S$  to be placed on the input.

### Serial ADCs

As we have seen in this section, many ADC data acquisition applications will use a microcomputer to control the system and collect the data. As previously mentioned with DAC applications, a parallel-data interface to a microcomputer will require many port bits to input the data. Today, many ADC chips are designed instead to output the data serially, thereby providing a more cost-effective interface with the rest of the data acquisition system. These ADCs have a built-in parallel in/serial out (PISO) shift register to convert the data to a bit stream that can be clocked serially into the microcomputer chip.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is *digitizing a signal*?
2. Describe the steps in a computer data acquisition process.
3. What is the minimum sample frequency needed to reconstruct an analog signal?
4. What occurs if the signal is sampled at less than the minimum frequency determined in question 3?

## 11-11 SUCCESSIVE-APPROXIMATION ADC

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the operation of a successive-approximation ADC.
- Determine the conversion time of an SAADC.
- Compare characteristics of ADC strategies.
- Use an ADC0804 to meet the specifications in data acquisition system.

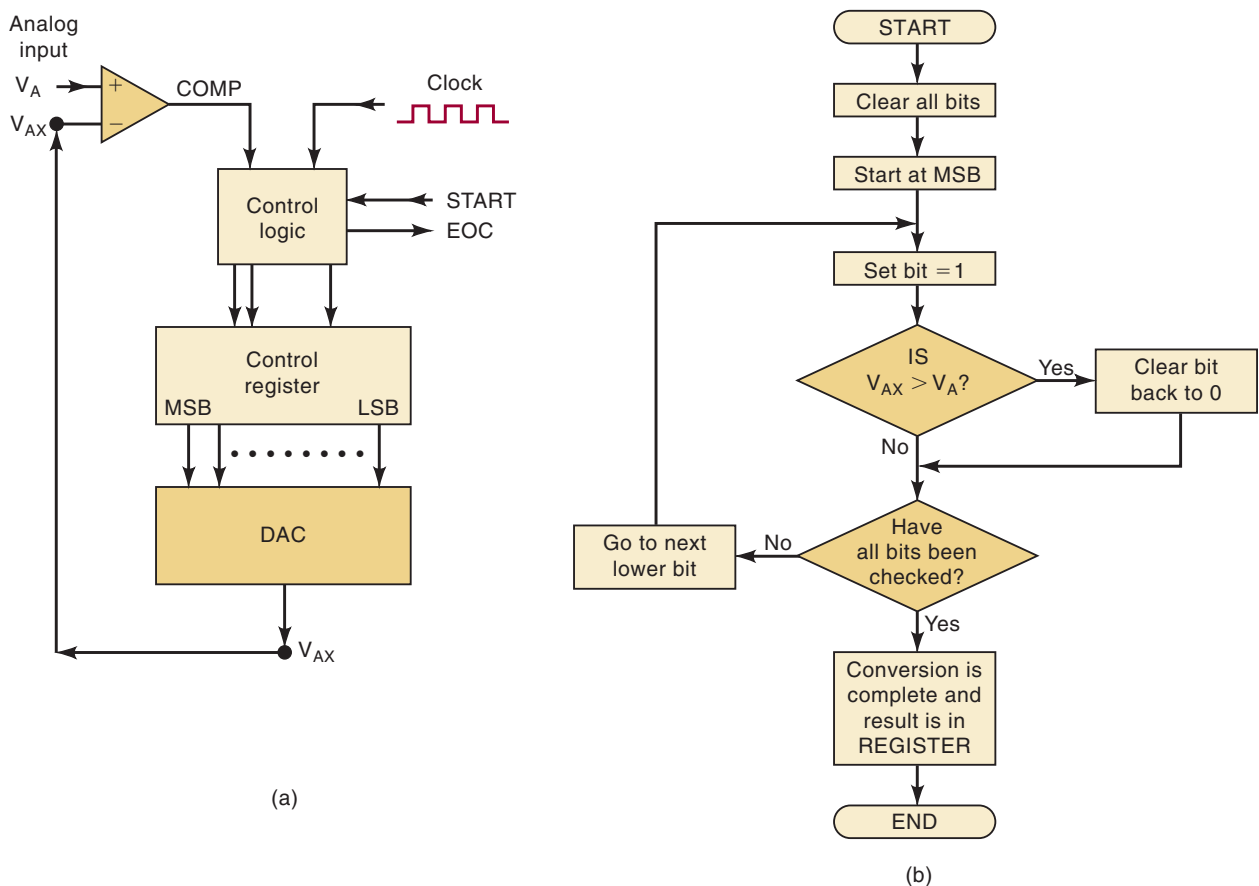
- State the role of each input and output of an ADC0804.
- Determine the resolution of an ADC.

The **successive-approximation converter** is one of the most widely used types of ADC. It has more complex circuitry than the digital-ramp ADC but a much shorter conversion time. In addition, successive-approximation converters (SACs) have a fixed value of conversion time that is not dependent on the value of the analog input.

The basic arrangement, shown in Figure 11-18(a), is similar to that of the digital-ramp ADC. The SAC, however, does not use a counter to provide the input to the DAC block but uses a register instead. The control logic modifies the contents of the register bit by bit until the register data are the digital equivalent of the analog input  $V_A$  within the resolution of the converter. The basic sequence of operation is given by the flowchart in Figure 11-18(b). We will follow this flowchart as we go through the example illustrated in Figure 11-19.

For this example, we have chosen a simple four-bit converter with a step size of 1 V. Even though most practical SACs would have more bits and smaller resolution than our example, the operation will be exactly the same. At this point, you should be able to determine that the four register bits feeding the DAC have weights of 8, 4, 2, and 1 V, respectively.

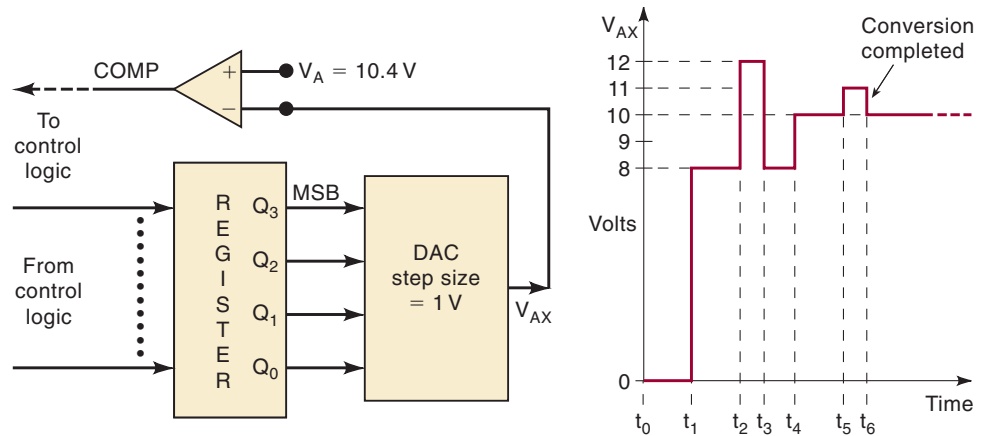
Let's assume that the analog input is  $V_A = 10.4\text{ V}$ . The operation begins with the control logic clearing all of the register bits to 0 so that  $Q_3 = Q_2 = Q_1 = Q_0 = 0$ . We will express this as  $[Q] = 0000$ . This makes



**FIGURE 11-18** Successive-approximation ADC: (a) simplified block diagram; (b) flowchart of operation.



**FIGURE 11-19** Illustration of four-bit SAC operation using a DAC step size of 1 V and  $V_A = 10.4$  V.



the DAC output  $V_{AX} = 0$  V, as indicated at time  $t_0$  on the timing diagram in Figure 11-19. With  $V_{AX} < V_A$ , the comparator output is HIGH.

At the next step (time  $t_1$ ), the control logic sets the MSB of the register to 1 so that  $[Q] = 1000$ . This produces  $V_{AX} = 8$  V. Because  $V_{AX} < V_A$ , the COMP output is still HIGH. This HIGH tells the control logic that the setting of the MSB did not make  $V_{AX}$  exceed  $V_A$ , so that the MSB is kept at 1.

The control logic now proceeds to the next lower bit,  $Q_2$ . It sets  $Q_2$  to 1 to produce  $[Q] = 1100$  and  $V_{AX} = 12$  V at time  $t_2$ . Because  $V_{AX} > V_A$ , the COMP output goes LOW. This LOW signals the control logic that the value of  $V_{AX}$  is too large, and the control logic then clears  $Q_2$  back to 0 at  $t_3$ . Thus, at  $t_3$ , the register contents are back to 1000 and  $V_{AX}$  is back to 8 V.

The next step occurs at  $t_4$ , where the control logic sets the next lower bit  $Q_1$  so that  $[Q] = 1010$  and  $V_{AX} = 10$  V. With  $V_{AX} < V_A$ , COMP is HIGH and tells the control logic to keep  $Q_1$  set at 1.

The final step occurs at  $t_5$ , where the control logic sets the next lower bit  $Q_0$  so that  $[Q] = 1011$  and  $V_{AX} = 11$  V. Because  $V_{AX} > V_A$ , COMP goes LOW to signal that  $V_{AX}$  is too large, and the control logic clears  $Q_0$  back to 0 at  $t_6$ .

At this point, all of the register bits have been processed, the conversion is complete, and the control logic activates its  $\overline{EOC}$  output to signal that the digital equivalent of  $V_A$  is now in the register. For this example, digital output for  $V_A = 10.4$  V is  $[Q] = 1010$ . Notice that 1010 is actually equivalent to 10 V, which is *less than* the analog input; this is a characteristic of the successive-approximation method. Recall that in the digital-ramp method, the digital output was always equivalent to a voltage that was on the step above  $V_A$ .

#### EXAMPLE 11-16

An eight-bit SAC has a resolution of 20 mV. What will its digital output be for an analog input of 2.17 V?

#### Solution

$$2.17 \text{ V} / 20 \text{ mV} = 108.5$$

so that step 108 would produce  $V_{AX} = 2.16$  V and step 109 would produce 2.18 V. The SAC always produces a final  $V_{AX}$  that is at the step *below*  $V_A$ . Therefore, for the case of  $V_A = 2.17$  V, the digital result would be  $108_{10} = 01101100_2$ .

## Conversion Time

In the operation just described, the control logic goes to each register bit, sets it to 1, decides whether or not to keep it at 1, and goes on to the next bit. The processing of each bit takes one clock cycle, so that the total conversion time for an  $N$ -bit SAC will be  $N$  clock cycles. That is,

$$t_c \text{ for SAC} = N \times 1 \text{ clock cycle}$$

This conversion time will be the same *regardless of the value of  $V_A$*  because the control logic must process each bit to see whether or not a 1 is needed.

### EXAMPLE 11-17

Compare the maximum conversion times of a 10-bit digital-ramp ADC and a 10-bit successive-approximation ADC if both utilize a 500-kHz clock frequency.

#### Solution

For the digital-ramp converter, the maximum conversion time is

$$(2^N - 1) \times (1 \text{ clock cycle}) = 1023 \times 2 \mu\text{s} = 2046 \mu\text{s}$$

For a 10-bit successive-approximation converter, the conversion time is always 10 clock periods or

$$10 \times 2 \mu\text{s} = 20 \mu\text{s}$$

Thus, it is about 100 times faster than the digital-ramp converter.

Because SACs have relatively fast conversion times, their use in data acquisition applications will permit more data values to be acquired in a given time interval. This feature can be very important when the analog data are changing at a relatively fast rate.

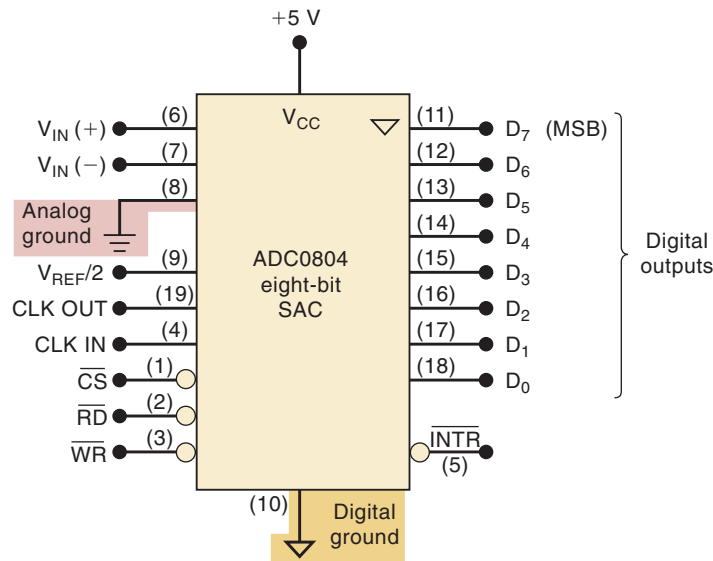
Because many SACs are available as ICs, it is rarely necessary to design the control logic circuitry, and so we will not cover it here. For those who are interested in the details of the control logic, many manufacturers' data books or web sites should provide sufficient detail.

## An Actual IC: The ADC0804 Successive-Approximation ADC

ADCs are available from several IC manufacturers with a wide range of operating characteristics and features. We will take a look at one of the more popular devices to get an idea of what is actually used in system applications. Figure 11-20 is the pin layout for the ADC0804, which is a 20-pin CMOS IC that performs A/D conversion using the successive-approximation method. Some of its important characteristics are as follows:

- It has two analog inputs,  $V_{IN}(+)$  and  $V_{IN}(-)$ , to allow **differential inputs**. In other words, the actual analog input,  $V_{IN}$ , is the difference in the voltages applied to these pins [analog  $V_{IN} = V_{IN}(+) - V_{IN}(-)$ ]. In single-ended measurements, the analog input is applied to  $V_{IN}(+)$ , while  $V_{IN}(-)$  is connected to analog ground. During normal operation, the

**FIGURE 11-20** ADC0804 eight-bit successive-approximation ADC with tristate outputs. The numbers in parentheses are the IC's pin numbers.



converter uses  $V_{CC} = +5\text{ V}$  as its reference voltage, and the analog input can range from 0 to 5 V.

- It converts the differential analog input voltage to an eight-bit tristate buffered digital output. The internal circuitry is slightly more complex than that described in Figure 11-19 in order to make transitions between output values occur at the nominal value  $\pm\frac{1}{2}$  LSB. For example, with 10-mV resolution, the A/D output would switch from 0 to 1 at 5 mV, from 1 to 2 at 15 mV, and so on. For this converter the resolution is calculated as  $V_{REF}/256$ ; with  $V_{REF} = 5.00\text{ V}$ , the resolution is 19.53 mV. The nominal full-scale input is  $255 \times 19.53 = 4.98\text{ V}$ , which should produce an output of 11111111. This converter will output 11111111 for any analog input between approximately 4.971 and 4.990 V.
- It has an internal clock generator circuit that produces a frequency of  $f = 1/(1.1RC)$ , where  $R$  and  $C$  are values of externally connected components. A typical clock frequency is 606 kHz using  $R = 10\text{ k}\Omega$  and  $C = 150\text{ pF}$ . An external clock signal can be used, if desired, by connecting it to the CLK IN pin.
- Using a 606-kHz clock frequency, the conversion time is approximately  $100\ \mu\text{s}$ .
- It has separate ground connections for digital and analog voltages. Pin 8 is the analog ground that is connected to the common reference point of the analog circuit that is generating the analog voltage. Pin 10 is the digital ground that is the one used by all of the digital devices in the system. (Note the different symbols used for the different grounds.) The digital ground is inherently noisy because of the rapid current changes that occur as digital devices change states. Although it is not necessary to use a separate analog ground, doing so ensures that the noise from digital ground is prevented from causing premature switching of the analog comparator inside the ADC.

This IC is designed to be easily interfaced to a microprocessor data bus. For this reason, the names of some of the ADC0804 inputs and outputs are

based on functions that are common to microprocessor-based systems. The functions of these inputs and outputs are defined as follows:

- $\overline{CS}$  (Chip Select). This input must be in its active-LOW state for  $\overline{RD}$  or  $\overline{WR}$  inputs to have any effect. With  $\overline{CS}$  HIGH, the digital outputs are in the Hi-Z state, and no conversions can take place.
- $\overline{RD}$  (READ). This input is used to enable the digital output buffers. With  $\overline{CS} = \overline{RD} = \text{LOW}$ , the digital output pins will have logic levels representing the results of the *last* A/D conversion. The microcomputer can then *read* (fetch) this digital data value over the system data bus.
- $\overline{WR}$  (WRITE). A LOW pulse is applied to this input to signal the start of a new conversion. This is actually a start conversion input. It is called a **WRITE** input because in a typical application, the microcomputer generates a WRITE pulse (similar to one used for writing to memory) that drives this input.
- $\overline{INTR}$  (INTERRUPT). This output signal will go HIGH at the start of a conversion and will return LOW to signal the end of conversion. This is actually an end-of-conversion output signal, but it is called INTERRUPT because in a typical situation, it is sent to a microprocessor's interrupt input to get the microprocessor's attention and let it know that the ADC's data are ready to be read.
- $V_{\text{REF}}/2$ . This is an optional input that can be used to reduce the internal reference voltage and thereby change the analog input range that the converter can handle. When this input is unconnected, its voltage will be  $\frac{1}{2}$  of  $V_{\text{CC}}$  because  $V_{\text{CC}}$  is then being used as the reference. With a nominal  $V_{\text{CC}}$  supply voltage of 5 V,  $V_{\text{REF}}/2$  will be 2.5 V. Note that any deviation of  $V_{\text{CC}}$  from the nominal value, which is likely, will produce a different value for  $V_{\text{REF}}$  and, therefore, a different value for the resolution. By connecting an external voltage (limited to  $V_{\text{REF}}/2 \leq \frac{1}{2} \times V_{\text{CC}}$ ) to this pin, the internal reference is changed to twice that voltage, and the analog input range and resolution is changed accordingly (see Table 11-6).
- CLK OUT. A resistor is connected to this pin to use the internal clock. The clock signal appears on this pin.
- CLK IN. Used for external clock input, or for a capacitor connection when the internal clock is used.

**TABLE 11-6** Examples relating  $V_{\text{REF}}$ ,  $V_{\text{IN}}$  range, and resolution.

$V_{\text{REF}}/2$ (V)	Analog Input Range (V)	Resolution (mV)
Open	0–5	19.5
2.25	0–4.5	17.6
2.0	0–4	15.6
1.5	0–3	11.7

### EXAMPLE 11-18

An ADC0804 is to be used in an application that requires a resolution of 10 mV.

- (a) What voltage should we apply to the  $V_{\text{REF}}/2$  pin?
- (b) What analog input range can this circuit digitize?
- (c) What is the nominal full-scale input voltage?
- (d) What is the minimum input voltage that will produce a full-scale output?
- (e) What is the binary output for an analog input of 2.00 V?

**Solution**

(a) The specified resolution is

$$10 \text{ mV} = \frac{V_{\text{REF}}}{256}$$

Therefore,

$$\begin{aligned} V_{\text{REF}} &= 256 \times 10 \text{ mV} = 2.56 \text{ V} \\ \frac{V_{\text{REF}}}{2} &= \frac{2.56 \text{ V}}{2} = 1.28 \text{ V} \end{aligned}$$

(b) The analog input range is 0–2.56 V.

(c) A full-scale output will be produced for

$$255 \times 10 \text{ mV} = 2.55 \text{ V}$$

(d) The minimum input voltage for full scale is

$$2.55 \text{ V} - \frac{1}{2} \text{ LSB} = 2.55 \text{ V} - \frac{1}{2} \times 10 \text{ mV} = 2.545 \text{ V}$$

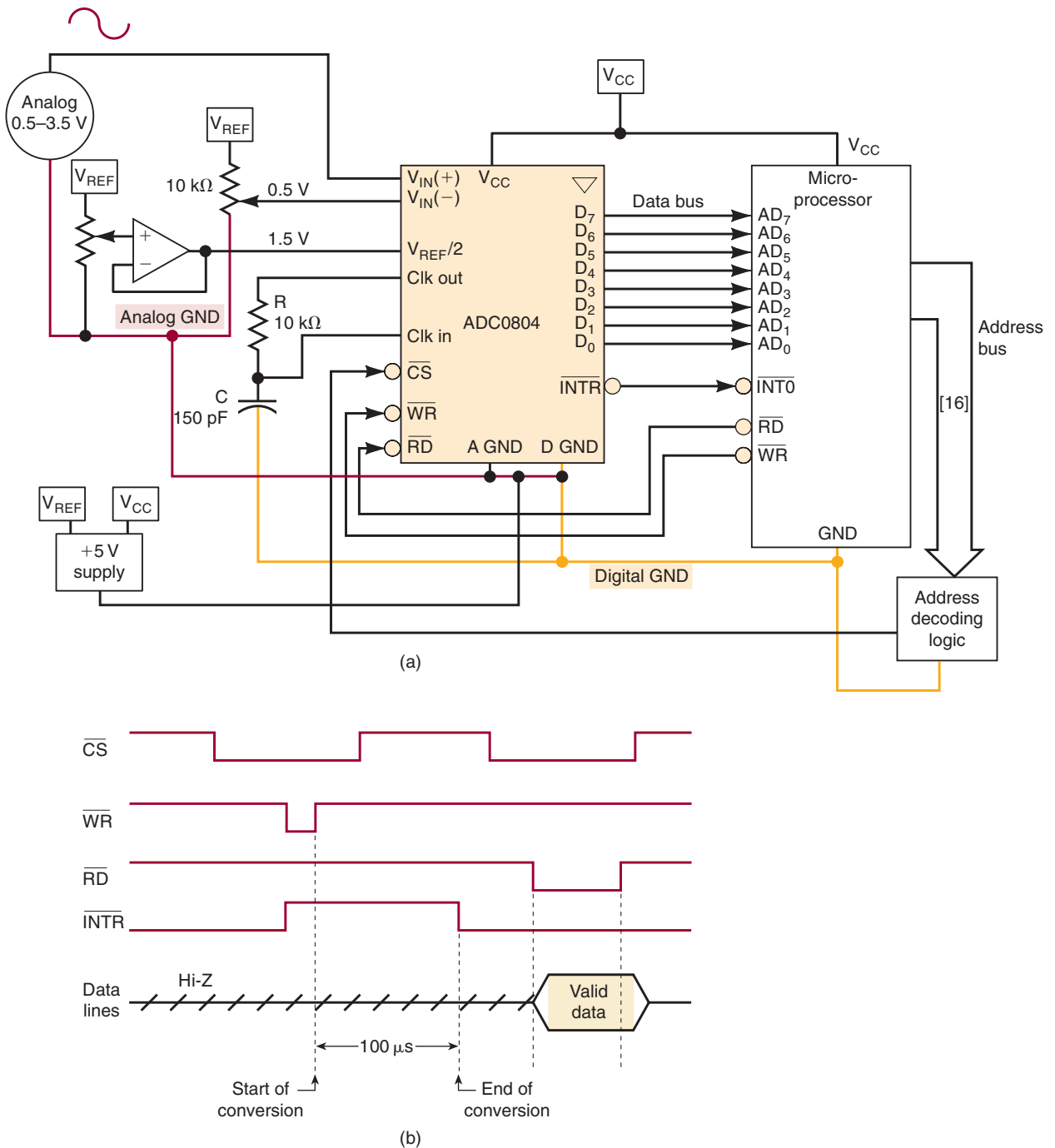
(e) The output is

$$\frac{2.00 \text{ V}}{10 \text{ mV}} = 200_{10} = 11001000_2$$

Figure 11-21(a) shows a typical connection of the ADC0804 to a microcomputer in a data acquisition application. The microcomputer controls when a conversion is to take place by generating the  $\overline{\text{CS}}$  and  $\overline{\text{WR}}$  signals. It then acquires the ADC output data by generating the  $\overline{\text{CS}}$  and  $\overline{\text{RD}}$  signals after detecting an NGT at  $\overline{\text{INTR}}$ , indicating the end of conversion. The waveforms in Figure 11-21(b) show the signal activity during the data acquisition process. Note that  $\overline{\text{INTR}}$  goes HIGH when  $\overline{\text{CS}}$  and  $\overline{\text{WR}}$  are LOW, but the conversion process does not begin until  $\overline{\text{WR}}$  returns HIGH. Also note that the ADC output data lines are in their Hi-Z state until the microcomputer activates  $\overline{\text{CS}}$  and  $\overline{\text{RD}}$ ; at that point the ADC's data buffers are enabled so that the ADC data are sent to the microcomputer over the data bus. The data lines return to the Hi-Z state when either  $\overline{\text{CS}}$  or  $\overline{\text{RD}}$  is returned HIGH.

In this application of the ADC0804, the input signal is varying over a range of 0.5 to 3.5 V. In order to make full use of the eight-bit resolution, the A/D must be matched to the analog signal specifications. In this case, the full-scale range is 3.0 V. However, it is offset from ground by 0.5 V. The offset of 0.5 V is applied to the negative input  $V_{\text{IN}}(-)$ , establishing this as the 0 value reference. The range of 3.0 V is set by applying 1.5 V to  $V_{\text{REF}}/2$ , which establishes  $V_{\text{REF}}$  as 3.0 V. An input of 0.5 V will produce a digital value of 00000000, and an input of 3.5 V (or any value over 3.482) will produce 11111111.

Another major concern when interfacing digital and analog signals is *noise*. Notice that the digital and analog ground paths are separated. The two grounds are tied together at a point that is very close to the A/D converter. A very low-resistance path ties this point directly to the negative terminal of the power supply. It is also wise to route the positive supply lines separately to digital and analog devices and make extensive use of decoupling capacitors (0.01  $\mu\text{F}$ ) from very near each chip's supply connection to ground.



**FIGURE 11-21** (a) An application of an ADC0804; (b) typical timing signals during data acquisition.

### EXAMPLE 11-19

For the ADC0804 in Figure 11-21, determine:

- The binary output produced with an analog input of 1.168 V.
- The nominal analog input voltage that produces an output of 01100111.
- The range of analog input values that will produce an output of 01100111.

**Solution**

- (a) The ADC in this circuit will convert the voltage difference between the two analog inputs.

$$V_{\text{IN}(+)} - V_{\text{IN}(-)} = 1.168 \text{ V} - 0.5 \text{ V} = 0.668 \text{ V}$$

The resolution is

$$\frac{V_{\text{REF}}}{256} = \frac{2 \times 1.5 \text{ V}}{256} = 11.7 \text{ mV}$$

The binary output is

$$\frac{0.668 \text{ V}}{11.7 \text{ mV}} = 57_{10} = 00111001_2$$

- (b) The output is

$$01100111_2 = 103_{10}$$

The nominal analog input is

$$V_{\text{IN}(+)} = (103 \times 11.7 \text{ mV}) + 0.5 \text{ V} = 1.705 \text{ V}$$

- (c) The analog input range is the nominal input voltage  $\pm \frac{1}{2}$  LSB.

$$1.705 \text{ V} \pm \left(\frac{1}{2} \times 11.7 \text{ mV}\right) = 1.699 \text{ to } 1.711 \text{ V}$$

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is the main advantage of a SAC over a digital-ramp ADC?
2. What is its principal disadvantage compared with the digital-ramp converter?
3. *True or false:* The conversion time for a SAC increases as the analog voltage increases.
4. Answer the following concerning the ADC0804.
  - (a) What is its resolution in bits?
  - (b) What is the normal analog input voltage range?
  - (c) Describe the functions of the  $\overline{\text{CS}}$ ,  $\overline{\text{WR}}$ , and  $\overline{\text{RD}}$  inputs.
  - (d) What is the function of the  $\overline{\text{INTR}}$  output?
  - (e) Why does it have two separate grounds?
  - (f) What is the purpose of  $V_{\text{IN}(-)}$ ?

## 11-12 FLASH ADCs

### OUTCOMES

Upon completion of this section, you will be able to:

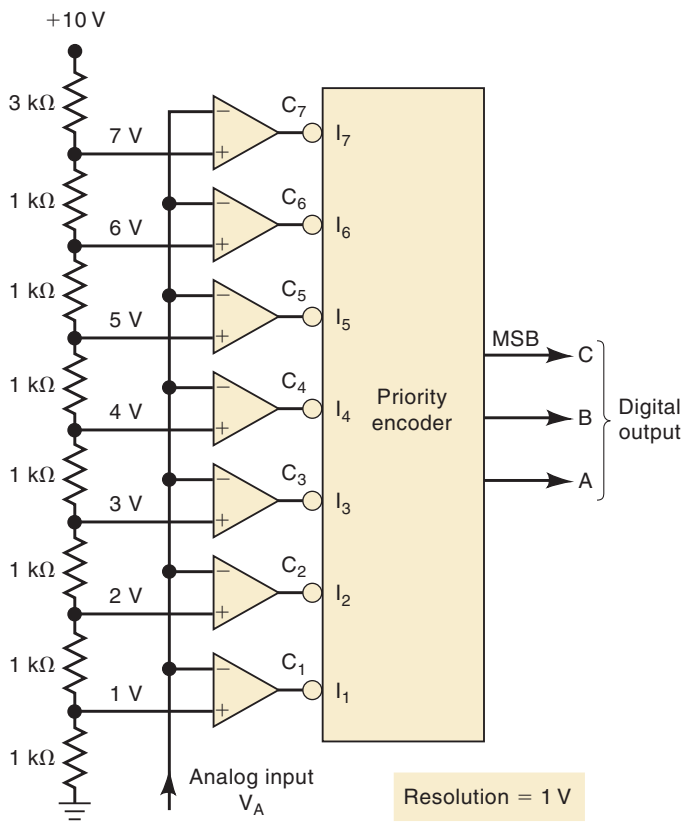
- Describe the operation of a flash ADC.
- Compare ADC strategies.

The flash converter is the highest-speed ADC available, but it requires much more circuitry than the other types. For example, a six-bit flash ADC requires 63 analog comparators, while an eight-bit unit requires 255 comparators, and a ten-bit converter requires 1023 comparators. The large number of comparators has limited the size of flash converters. IC flash converters are commonly available in two- to eight-bit units, and most manufacturers offer nine- and ten-bit units as well.

The principle of operation will be described for a three-bit flash converter in order to keep the circuitry at a workable level. Once the three-bit converter is understood, it should be easy to extend the basic idea to higher-bit flash converters.

The flash converter in Figure 11-22(a) has a three-bit resolution and a step size of 1 V. The voltage divider sets up reference levels for each comparator so that there are seven levels corresponding to 1 V (weight of LSB),

**FIGURE 11-22** (a) Three-bit flash ADC; (b) truth table.



(a)

Analog in $V_A$	Comparator outputs							Digital outputs		
	$C_1$	$C_2$	$C_3$	$C_4$	$C_5$	$C_6$	$C_7$	C	B	A
0–1 V	1	1	1	1	1	1	1	0	0	0
1–2 V	0	1	1	1	1	1	1	0	0	1
2–3 V	0	0	1	1	1	1	1	0	1	0
3–4 V	0	0	0	1	1	1	1	0	1	1
4–5 V	0	0	0	0	1	1	1	1	0	0
5–6 V	0	0	0	0	0	1	1	1	0	1
6–7 V	0	0	0	0	0	0	1	1	1	0
> 7 V	0	0	0	0	0	0	0	1	1	1

(b)





2 V, 3 V, . . . , and 7 V (full scale). The analog input,  $V_A$ , is connected to the other input of each comparator.

With  $V_A < 1$  V, all of the comparator outputs  $C_1$  through  $C_7$  will be HIGH. With  $V_A > 1$  V, one or more of the comparator outputs will be LOW. The comparator outputs are fed into an active-LOW priority encoder that generates a binary output corresponding to the highest-numbered comparator output that is LOW. For example, when  $V_A$  is between 3 and 4 V, outputs  $C_1$ ,  $C_2$ , and  $C_3$  will be LOW and all others will be HIGH. The priority encoder will respond only to the LOW at  $C_3$  and will produce a binary output  $CBA = 011$ , which represents the digital equivalent of  $V_A$ , within the resolution of 1 V. When  $V_A$  is greater than 7 V,  $C_1$  to  $C_7$  will all be LOW, and the encoder will produce  $CBA = 111$  as the digital equivalent of  $V_A$ . The table in Figure 11-22(b) shows the responses for all possible values of analog input.

The flash ADC of Figure 11-22 has a resolution of 1 V because the analog input must change by 1 V in order to bring the digital output to its next step. To achieve finer resolutions, we would have to increase the number of input voltage levels (i.e., use more voltage-divider resistors) and the number of comparators. For example, an eight-bit flash converter would require  $2^8 = 256$  voltage levels, including 0 V. This would require 256 resistors and 255 comparators (there is no comparator for the 0-V level). The 255 comparator outputs would feed a priority encoder circuit that would produce an eight-bit code corresponding to the highest-order comparator output that is LOW. In general, an  $N$ -bit flash converter would require  $2^N - 1$  comparators,  $2^N$  resistors, and the necessary encoder logic.

### Conversion Time

The flash converter uses no clock signal because no timing or sequencing is required. The conversion takes place continuously. When the value of analog input changes, the comparator outputs will change, thereby causing the encoder outputs to change. The conversion time is the time it takes for a new digital output to appear in response to a change in  $V_A$ , and it depends only on the propagation delays of the comparators and encoder logic. For this reason, flash converters have extremely short conversion times and therefore, are suitable for applications that need to digitize very high bandwidth (i.e., high frequency) analog signals such as in data acquisition, communications, radar processing, and sampling oscilloscope applications. Flash converters, however, can be very expensive and tend to have relatively low resolutions and high power consumption.

#### OUTCOME ASSESSMENT QUESTIONS

1. *True or false:* A flash ADC does not contain a DAC.
2. How many comparators would a 12-bit flash converter require? How many resistors?
3. State the major advantage and disadvantage of a flash converter.

## 11-13 OTHER A/D CONVERSION METHODS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe dual-slope integrating ADC.
- Describe voltage-to-frequency ADC.

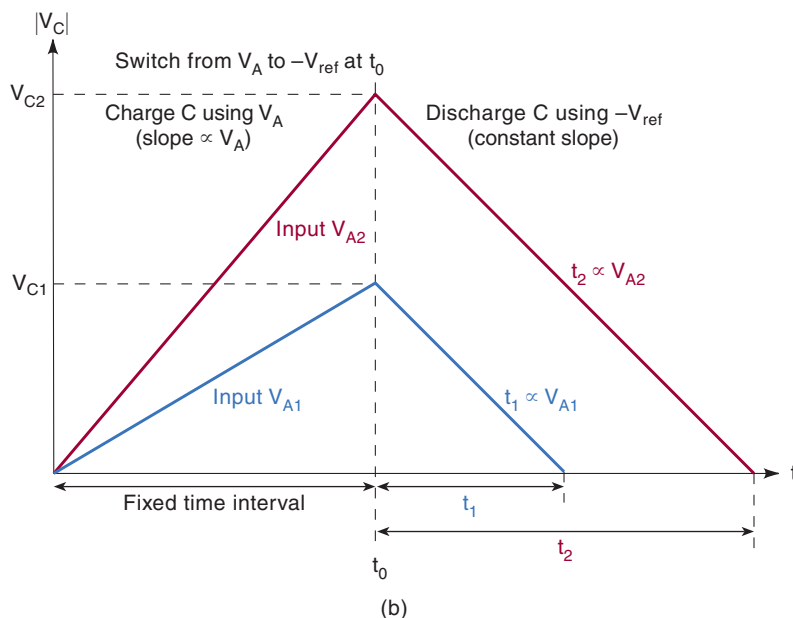
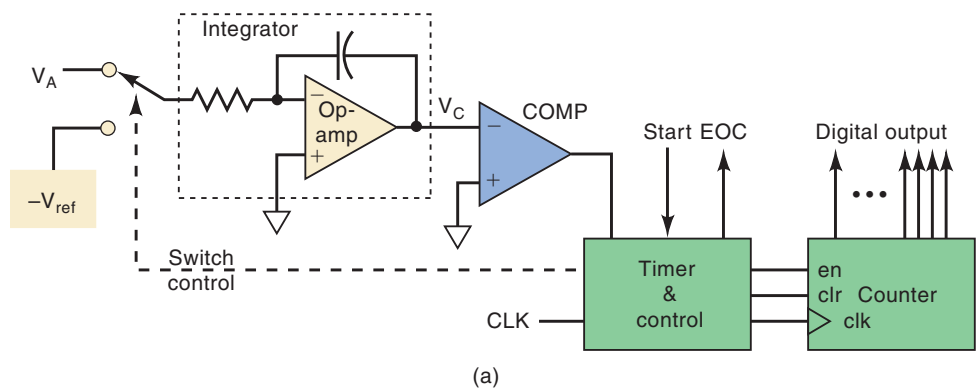
- Describe sigma/delta modulation.
- Describe pipelined ADC.

Several other methods of A/D conversion have been in use for some time, each with its relative advantages and disadvantages. We will briefly describe some of them now.

### Dual-Slope Integrating ADC

The **dual-slope converter** has one of the slowest conversion times (typically 10 to 100 ms) but has the advantage of relatively low cost because it does not require precision components such as a DAC or a VCO (voltage-controlled oscillator). The basic operation of this converter involves the *linear* charging and discharging of a capacitor using an integrator circuit [see Figure 11-23(a)]. First, the capacitor is charged up for a fixed time interval using a constant current that is proportional to the analog input voltage,  $V_A$ , as shown in Figure 11-23(b). Thus, at the end of this fixed charging interval, the capacitor voltage will also be proportional to  $V_A$ . At that point, the capacitor is linearly discharged from a constant current derived from a precise reference voltage,  $-V_{ref}$ . When the voltage comparator detects that the capacitor has discharged to 0 V, the linear discharging is terminated.

**FIGURE 11-23** Dual-slope integrating ADC: (a) block diagram; (b) charging/discharging capacitor.



During the discharge interval, a digital reference frequency is fed to a counter and counted. The duration of the discharge interval will be proportional to the initial capacitor voltage. Thus, at the end of the discharge interval, the counter will hold a count proportional to the initial capacitor voltage, which, as we said, is proportional to  $V_A$ .

In addition to its low cost, dual-slope integrating ADCs provide a high-resolution approach to converting low bandwidth (i.e., low frequency) analog signals. Another advantage of the dual-slope ADC is its low sensitivity to noise and to variations in its component values caused by temperature changes. Because of its slow conversion times, the dual-slope ADC is not used in any data acquisition applications. The slow conversion times, however, are not a problem in applications such as digital voltmeters or multimeters, and this is where they find their major application.

### Voltage-to-Frequency ADC

The **voltage-to-frequency ADC** is simpler than other ADCs because it does not use a DAC. Instead it uses a *linear voltage-controlled oscillator (VCO)* that produces an output frequency proportional to its input voltage. The analog voltage that is to be converted is applied to the VCO to generate an output frequency. This frequency is fed to a counter to be counted for a fixed time interval. The final count is proportional to the value of the analog voltage.

To illustrate, suppose that the VCO generates a frequency of 10 kHz for each volt of input (i.e., 1 V produces 10 kHz, 1.5 V produces 15 kHz, 2.73 V produces 27.3 kHz). If the analog input voltage is 4.54 V, then the VCO output will be a 45.4-kHz signal that clocks a counter for, say, 10 ms. After the 10-ms counting interval, the counter will hold the count of 454, which is the digital representation of 4.54 V.

Although this is a simple method of conversion, it is difficult to achieve a high degree of accuracy because of the difficulty in designing VCOs with accuracies of better than 0.1 percent.

One of the main applications of this type of converter is in noisy industrial environments where small analog signals must be transmitted from transducer circuits to a control computer. The small analog signals can be drastically affected by noise if they are directly transmitted to the control computer. A better approach is to feed the analog signal to a VCO, which generates a digital signal whose output frequency changes according to the analog input. This digital signal is transmitted to the computer and will be much less affected by noise. Circuitry in the control computer will count the digital pulses (i.e., perform a frequency-counting function) to produce a digital value equivalent to the original analog input.

### Sigma/Delta Modulation

Another approach to representing analog information in digital form is called **sigma/delta modulation**. A sigma/delta A/D converter is an oversampling device, which means that it effectively samples the analog information more often than the minimum sample rate. The minimum sample rate is two times higher than the highest frequency in the incoming analog wave. The sigma/delta approach, like the voltage-to-frequency approach, does not directly produce a multibit number for each sample. Instead, it represents the analog voltage by varying the density of logic 1s in a single-bit stream of serial data. To represent the positive portions of a waveform, a stream of bits with a high density of 1s is generated by the modulator (e.g., 01111101111110111110111). To represent the negative

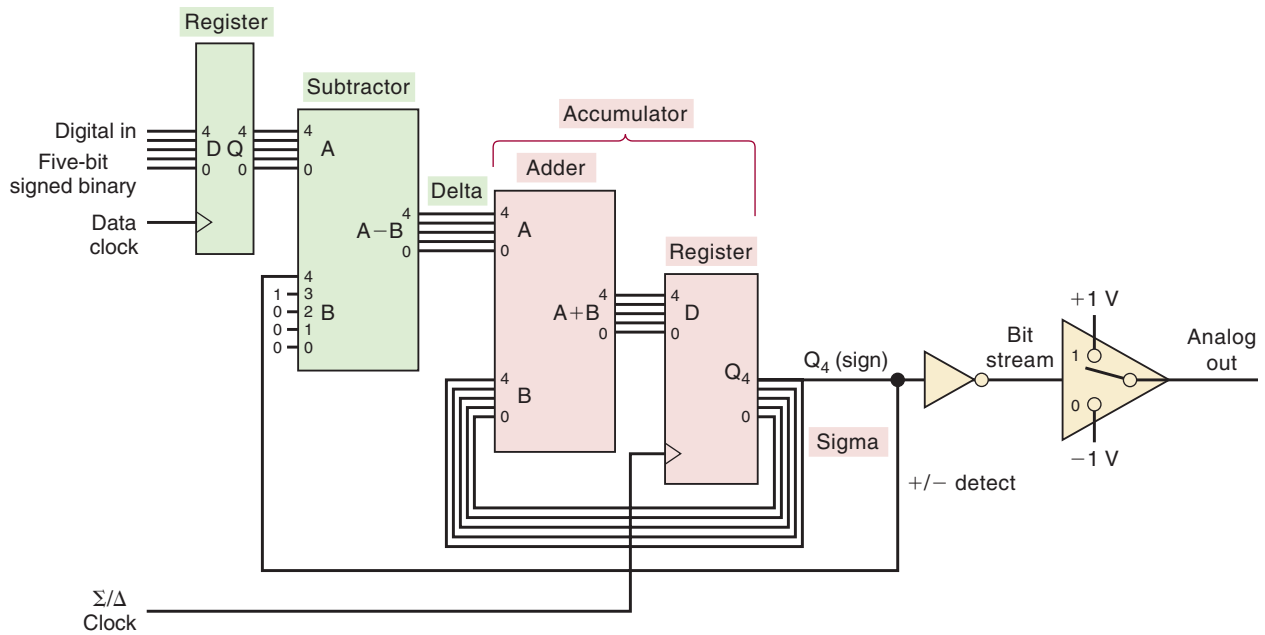


FIGURE 11-24 Sigma/delta modulator in a D/A converter.

portions, a lower density of 1s (i.e., a higher density of 0s) is generated (e.g., 00010001000010001000).

Sigma/delta modulation is used in A/D as well as D/A conversion. One form of a sigma/delta modulator circuit is designed to convert a continuous analog signal into a modulated bit stream (A/D). The other form converts a sequence of digital samples into the modulated bit stream (D/A). We are coming from the perspective of digital systems, so it is easiest to understand the latter of these two circuits because it consists of all digital components that we have studied. Figure 11-24 shows a circuit that takes a five-bit signed digital value as its input and converts it into a sigma/delta bit stream. We will assume that the numbers that can be placed on this circuit's input range from  $-8$  to  $+8$ . The first component is simply a subtractor (the delta section) similar to the one studied in Figure 6-14. The subtractor determines how far the input number is from its maximum or minimum value. This difference is often called the error signal. The second two components (the adder and the D register) form an accumulator very similar to the circuit in Figure 6-10 (the sigma section). For each sample that comes in, the accumulator adds the difference (error signal) to the running total. When the error is small, this running total (sigma) changes by small increments. When the error is large, the sigma changes by large increments. The last component compares the running total from the accumulator with a fixed threshold, which in this case is zero. In other words, it is simply determining if the total is positive or negative. This is accomplished by using the MSB (sign bit) of sigma. As soon as the total goes positive, the MSB goes LOW and feeds back to the delta section the maximum positive value ( $+8$ ). When the MSB of sigma goes negative, it feeds back the maximum negative value ( $-8$ ).

Let's use some examples to investigate the operations of a sigma/delta DAC. Table 11-7 shows the operation of the converter when a value of zero is the input. Notice that the bit stream output alternates between 1 and 0, and the average value of the analog output is 0 volts. Table 11-8 shows what happens when the digital input is 4. If we assume that 8 is full scale, this represents  $\frac{4}{8} = 0.5$ . The output is HIGH for three samples and LOW for one

**TABLE 11-7** Sigma/delta modulator with an input of 0.

Sample ( <i>n</i> )	Digital IN	Delta	Sigma	Bit Stream Out	Analog OUT	Feedback
1	0	-8	0	1	1	8
2	0	8	-8	0	-1	-8
3	0	-8	0	1	1	8
4	0	8	-8	0	-1	-8
5	0	-8	0	1	1	8
6	0	8	-8	0	-1	-8
7	0	-8	0	1	1	8
8	0	8	-8	0	-1	-8

**TABLE 11-8** Sigma/delta modulator with an input of 4.

Sample ( <i>n</i> )	Digital IN	Delta	Sigma	Bit Stream Out	Analog OUT	Feedback
1	4	-4	4	1	1	8
2	4	-4	0	1	1	8
3	4	12	-4	0	-1	-8
4	4	-4	8	1	1	8
5	4	-4	4	1	1	8
6	4	-4	0	1	1	8
7	4	12	-4	0	-1	-8
8	4	-4	8	1	1	8

sample, a pattern that repeats every four samples. The average value of the analog output is  $(1 + 1 + 1 - 1)/4 = 0.5\text{ V}$ .

As a final example, let's use an input of  $-5$ , which represents  $-\frac{5}{8} = -0.625$ . Table 11-9 shows the resulting output. The pattern in the bit stream is not periodic. From the sigma column, we can see that it takes 16 samples for the pattern to repeat. If we take the overall bit density, however, and calculate the average value of the analog output over 16 samples, we will find that it is equal to  $-0.625$ . CD players and MP3 players typically use a sigma/delta D/A converter that operates in this fashion. The 16-bit digital numbers come off the CD serially; then they are formatted into parallel data patterns and clocked into a converter. As the changing numbers come into the converter, the average value of the analog out changes accordingly. Next, the analog output goes through a circuit called a low-pass filter that smoothes out the sudden changes and produces a smoothly changing voltage that is the average value of the bit stream. In your headphones, this changing analog signal sounds just like the original recording. A sigma/delta A/D converter works in a very similar way but converts the analog voltage into the modulated bit stream. To store the digitized data as a list of  $N$ -bit binary numbers, the average bit density of  $2^N$  bit-stream samples is calculated and stored.

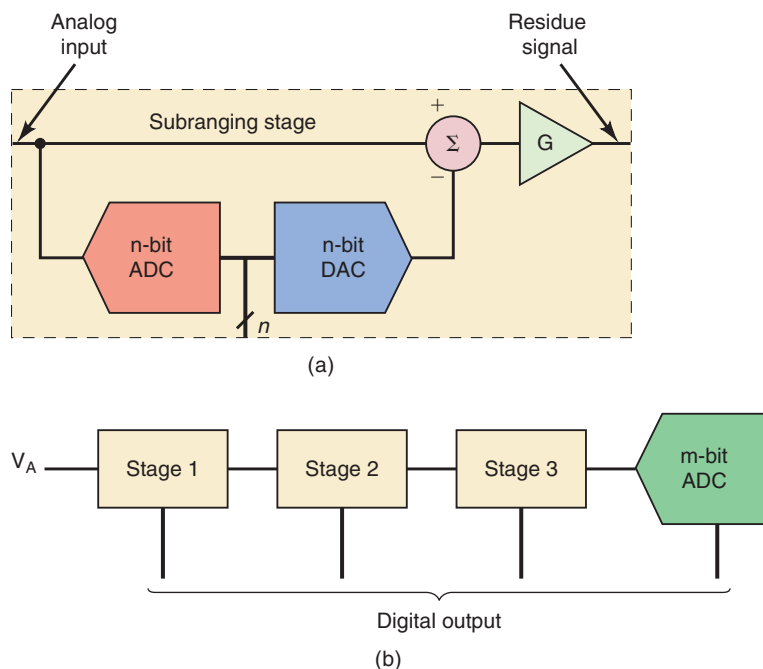
## Pipelined ADC

A **pipelined ADC** uses two or more subranging stages. Each subranging stage contains an  $n$ -bit ADC along with an  $n$ -bit DAC as shown in Figure 11-25(a). The first stage will perform a coarse conversion of the analog input and produce the most significant bits to be used for the digital output. This digital result is reconverted into an internal analog voltage by the DAC. The DAC's output will be subtracted from the original analog input. The difference

**TABLE 11-9** Sigma/delta modulator with an input of  $-5$ .

Sample ( $n$ )	Digital IN	Delta	Sigma	Bit Stream Out	Analog OUT	Feedback
1	-5	3	-5	0	-1	-8
2	-5	3	-2	0	-1	-8
3	-5	-13	1	1	1	8
4	-5	3	-12	0	-1	-8
5	-5	3	-9	0	-1	-8
6	-5	3	-6	0	-1	-8
7	-5	3	-3	0	-1	-8
8	-5	-13	0	1	1	8
9	-5	3	-13	0	-1	-8
10	-5	3	-10	0	-1	-8
11	-5	3	-7	0	-1	-8
12	-5	3	-4	0	-1	-8
13	-5	3	-1	0	-1	-8
14	-5	-13	2	1	1	8
15	-5	3	-11	0	-1	-8
16	-5	3	-8	0	-1	-8
17	-5	3	-5	0	-1	-8
18	-5	3	-2	0	-1	-8

between the input signal and the DAC output will be amplified by a set gain,  $G$ . This amplified difference is referred to as the residue signal, which is then converted to a finer resolution by the next pipeline stage [see Figure 11-25(b)]. Each subranging stage will produce a finer resolution of the analog input. The residue signal produced by the last subranging stage will be digitized by a final ADC block giving the finest resolution bits for the pipelined ADC. The pipelined ADC is essentially a refinement of the successive-approximation

**FIGURE 11-25** Pipelined ADC: (a) block diagram of a single subranging stage; (b) multiple subranging stages pipelined together.

ADC in which the feedback reference signal consists of the interim conversion of a set of bits rather than just the next-most-significant bit. By using three- or four-bit flash ADCs in the subranging stages, the resulting pipelined ADC is fast, has a high resolution, and is relatively inexpensive. This technique has become very popular.

### OUTCOME ASSESSMENT QUESTIONS

1. What do multiple subranging stages produce in a pipelined ADC?
2. What is the main element of a voltage-to-frequency ADC?
3. Cite two advantages and one disadvantage of the dual-slope ADC.
4. Name three types of ADCs that do not use a DAC.
5. How many output data bits does a sigma/delta modulator use?

## 11-14 TYPICAL ADC ARCHITECTURES FOR APPLICATIONS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Categorize the various strategies of ADC.
- Relate the categories to typical applications.

Most ADC applications tend to fall into one of four areas (listed in the order of required conversion speeds from lowest to highest): precision industrial measurement, voice/audio, data acquisition, and high speed. The low sampling rate, high resolution, and good noise rejection of dual-slope integrating ADCs are ideal characteristics for monitoring DC signals with instrumentation, such as digital multimeters. A wide variety of industrial measurement applications that require moderate bandwidths and high resolution, including sensor monitoring and motor control, use sigma/delta ADCs. With their high resolution and inherent oversampling, sigma/delta ADCs currently dominate voice and audio applications. Successive approximation is the principal architecture for the majority of complex data acquisition systems that need to digitize multiple analog data channels. The pipelined architecture is a popular choice for many high-speed applications including digital oscilloscopes, spectrum analyzers, medical imaging, digital video (DVDs and HDTV), radar, communications, and digital cameras. The highest-speed applications may require the flash architecture for ADC but at a relatively high cost and low resolution.

### OUTCOME ASSESSMENT QUESTIONS

1. List applications of sigma/delta A/D converters.
2. List an application of dual-slop integrating A/D converters.
3. List some applications of pipelined ADCs.
4. Which ADC is best included in microcontroller-based data acquisition systems?

## 11-15 SAMPLE-AND-HOLD CIRCUITS

### OUTCOME

Upon completion of this section, you will be able to:

- Describe the role of a sample-and-hold circuit.

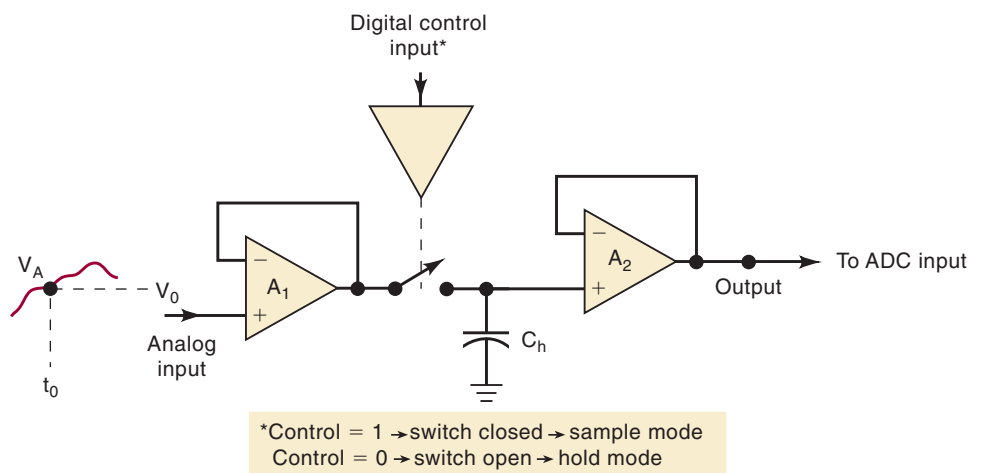
When an analog voltage is connected directly to the input of an ADC, the conversion process can be adversely affected if the analog voltage is changing during the conversion time. The stability of the conversion process can be improved by using a **sample-and-hold (S/H) circuit** to hold the analog voltage constant while the A/D conversion is taking place. A simplified diagram of a sample-and-hold (S/H) circuit is shown in Figure 11-26.

The S/H circuit contains a unity-gain buffer amplifier  $A_1$  that presents a high impedance to the analog signal and has a low output impedance that can rapidly charge the hold capacitor,  $C_h$ . The capacitor will be connected to the output of  $A_1$  when the digitally controlled switch is closed. This is called the *sample* operation. The switch will be closed long enough for  $C_h$  to charge to the present value of the analog input. For example, if the switch is closed at time  $t_0$ , the  $A_1$  output will quickly charge  $C_h$  up to a voltage  $V_0$ . When the switch opens,  $C_h$  will *hold* this voltage so that the output of  $A_2$  will apply this voltage to the ADC. The unity-gain buffer amplifier  $A_2$  presents a high input impedance that will not discharge the capacitor voltage appreciably during the conversion time of the ADC, and so the ADC will essentially receive a DC input voltage  $V_0$ .

In a computer-controlled data acquisition system such as the one discussed earlier, the S/H switch would be controlled by a digital signal from the computer. The computer signal would close the switch in order to charge  $C_h$  to a new sample of the analog voltage; the amount of time the switch would have to remain closed is called the **acquisition time**, and it depends on the value of  $C_h$  and the characteristics of the S/H circuit. The computer signal would then open the switch to allow  $C_h$  to hold its value and provide a relatively constant analog voltage at the  $A_2$  output.

The AD781 is a S/H integrated circuit that has a maximum acquisition time of 700 ns. During the hold time, the capacitor voltage will droop (discharge) at a rate of only  $0.01 \mu\text{V}/\mu\text{s}$ . The voltage droop within the sampling interval should be less than the weight of the LSB. For example, a 10-bit converter with a full-scale range of 10 V would have an LSB weight

**FIGURE 11-26** Simplified diagram of a sample-and-hold circuit.





of approximately 10 mV. It would take 1 s before the capacitor droop would equal the weight of the ADC's LSB. It is not likely, however, that it would ever be necessary to hold the sample for such a long time in the conversion process.

### OUTCOME ASSESSMENT QUESTIONS

1. Describe the function of a S/H circuit.
2. *True or false:* The amplifiers in an S/H circuit are used to provide voltage amplification.

## 11-16 MULTIPLEXING

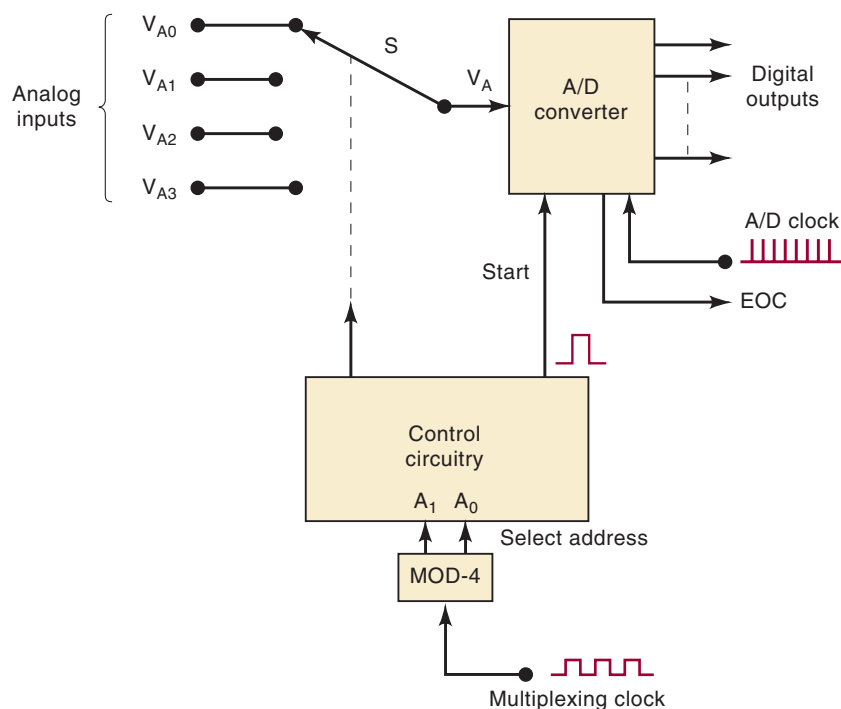
### OUTCOMES

Upon completion of this section, you will be able to:

- Use a multichannel ADC.
- Describe the role of the analog MUX in a multichannel ADC.

When analog inputs from several sources are to be converted, a multiplexing technique can be used so that one ADC may be time-shared. The basic scheme is illustrated in Figure 11-27 for a four-channel acquisition system. Rotary switch  $S$  is used to switch each analog signal to the input of the ADC, one at a time in sequence. The control circuitry controls the switch position according to the *select address* bits,  $A_1, A_0$ , from the MOD-4 counter. For example, with  $A_1A_0 = 00$ , the switch connects  $V_{A0}$  to the ADC input;  $A_1A_0 = 01$  connects  $V_{A1}$  to the ADC input; and so on. Each input channel has a specific address code that, when present, connects that channel to the ADC.

**FIGURE 11-27** Conversion of four analog inputs by multiplexing through one ADC.



The operation proceeds as follows:

1. With select address = 00,  $V_{A0}$  is connected to the ADC input.
2. The control circuit generates a START pulse to initiate the conversion of  $V_{A0}$  to its digital equivalent.
3. When the conversion is complete, *EOC* signals that the ADC output data are ready. Typically, these data will be transferred to a computer over a data bus.
4. The multiplexing clock increments the select address to 01, which connects  $V_{A1}$  to the ADC.
5. Steps 2 and 3 are repeated with the digital equivalent of  $V_{A1}$  now present at the ADC outputs.
6. The multiplexing clock increments the select address to 10, and  $V_{A2}$  is connected to the ADC.
7. Steps 2 and 3 are repeated with the digital equivalent of  $V_{A2}$  now present at the ADC outputs.
8. The multiplexing clock increments the select address to 11, and  $V_{A3}$  is connected to the ADC.
9. Steps 2 and 3 are repeated with the digital equivalent of  $V_{A3}$  now present at the ADC outputs.

The multiplexing clock controls the rate at which the analog signals are sequentially switched into the ADC. The maximum rate is determined by the delay time of the switches and the conversion time of the ADC. The switch delay time can be minimized by using semiconductor switches such as the CMOS bilateral switch described in Chapter 8. It may be necessary to connect a S/H circuit at the input of the ADC if the analog inputs will change significantly during the ADC conversion time.

Many integrated ADCs contain the multiplexing circuitry on the same chip as the ADC. The ADC0808, for example, can multiplex eight different analog inputs into one ADC. It uses a three-bit select input code to determine which analog input is connected to the ADC.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the advantage of this multiplexing scheme?
2. How would the address counter be changed if there were eight analog inputs?

## 11-17 DIGITAL SIGNAL PROCESSING (DSP)

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe the process of performing DSP.
- Describe digital filtering.
- Describe the functional blocks required in a DSP.
- Define terms common to DSP applications.

One of the most dynamic areas of digital systems today is in the field of **digital signal processing (DSP)**. A DSP is a very specialized form of microprocessor

that has been optimized to perform repetitive calculations on streams of digitized data. The digitized data are usually being fed to the DSP from an A/D converter. It is beyond the scope of this text to explain the mathematics that allow a DSP to process these data values, but suffice it to say that for each new data point that comes in, a calculation is performed (very quickly). This calculation involves the most recent data point as well as several of the preceding data samples. The result of the calculation produces a new output data point, which is usually sent to a D/A converter. A DSP system is similar to the block diagram shown in Figure 11-1. The main difference is in the specialized hardware contained in the computer section.

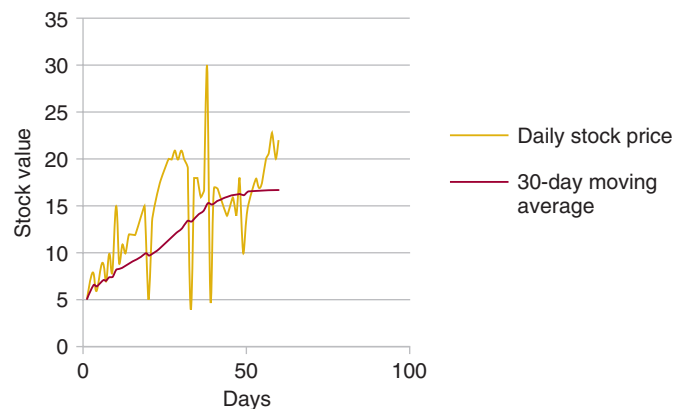
A major application for DSP is in filtering and conditioning of analog signals. As a very simple example, a DSP can be programmed to take in an analog waveform, such as the output from an audio preamplifier, and pass to the output only those frequency components that are below a certain frequency. All higher frequencies are attenuated by the filter. Perhaps you recall from your study of analog circuits that the same thing can be accomplished by a simple low-pass filter made from a resistor and capacitor. The advantage of DSP over resistors and capacitors is the flexibility of being able to change the critical frequency without switching any components. Instead, numbers are simply changed in the calculations to adapt the dynamic response of the filter. Have you ever been in an auditorium when the PA system started to squeal? This can be prevented if the degenerative feedback frequency can be filtered out. Unfortunately, the frequency that causes the squeal changes with the number of people in the room, the clothes they wear, and many other factors. With a DSP-based audio equalizer, the oscillation frequency can be detected and the filters dynamically adjusted to tune it out.

## Digital Filtering

To help you understand digital filtering, imagine you are buying and selling stock. To decide when to buy and sell, you need to know what the market is doing. You want to ignore sudden, short-term (high-frequency) changes but respond to the overall trends (30-day averages). Every day you read the newspaper, take a sample of the closing price for your stock, and write it down. Then you use a formula to calculate the average of the last 30 days' prices. This average value is plotted as shown in Figure 11-28, and the resulting graph is used to make decisions. This is a way of filtering the digital signal (sequence of data samples) that represents the stock market activity.

Now imagine that instead of sampling stock prices, a digital system is sampling an audio (analog) signal from a microphone using an A/D converter.

**FIGURE 11-28** Digital filtering of stock market activity.



Instead of taking a sample once a day, it takes a sample 20,000 times each second (every  $50 \mu\text{s}$ ). For each sample, a weighted averaging calculation is performed using the last 256 data samples and produces a single output data point. A **weighted average** means that some of the data points are considered more important than others. Each of the samples is multiplied by a fractional number (between 0 and 1) before adding them together. This averaging calculation is processing (filtering) the audio signal. The most difficult part of this form of DSP is determining the correct weighting constants for the averaging calculation in order to achieve the desired filter characteristics. Fortunately, there is readily available software for PCs that makes this very easy. The special DSP hardware must perform the following operations:

Read the newest sample (one new number) from A/D.

Replace the oldest sample (of 256) with the new one from A/D.

Multiply each of the 256 samples by their corresponding weight constant.

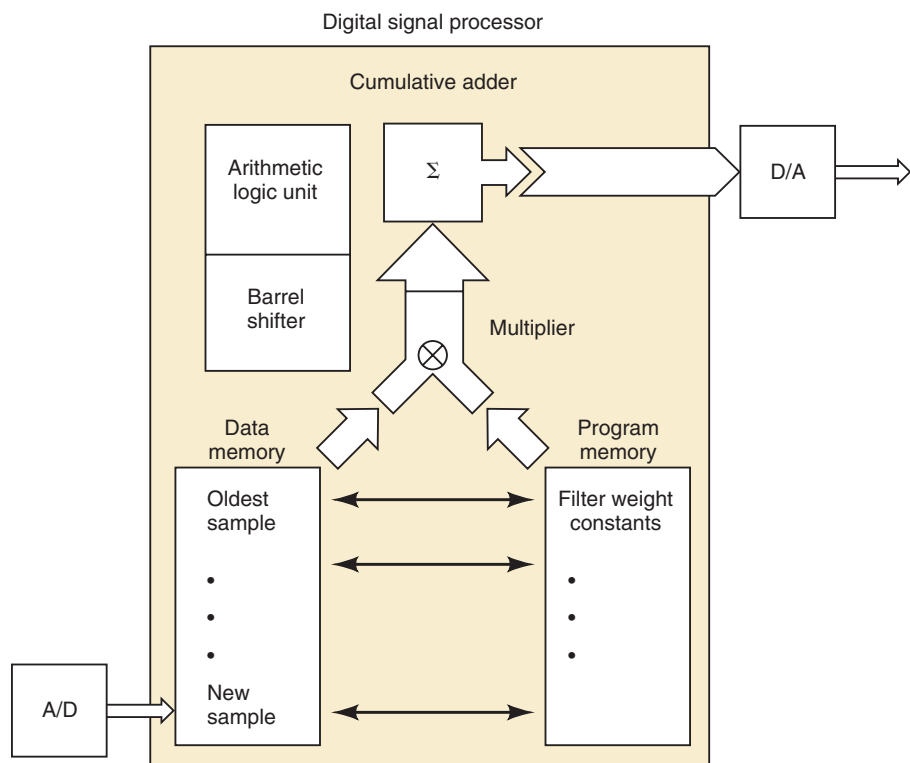
Add all of these products.

Output the resulting sum of products (1 number) to the D/A.

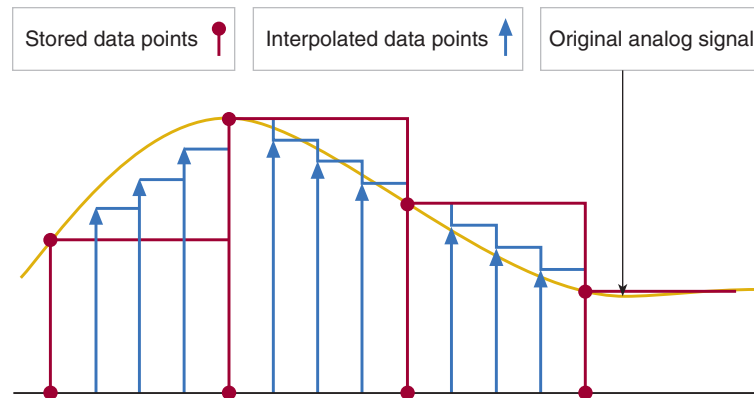
Figure 11-29 shows the basic architecture of a DSP. The multiply and accumulate (**MAC**) section is central to all DSPs and is used in most applications. Special hardware, like you will study in Chapter 12, is used to implement the memory system that stores the data samples and weight values. The **arithmetic logic unit** and **barrel shifter** provide the necessary support to deal with the binary number system while processing signals.

Another useful application of DSP is called **oversampling** or **interpolation filtering**. As you recall, the reconstructed waveform is always an approximation of the original due to quantization error. The sudden step changes from one data point to the next also introduce high-frequency noise into the reconstructed signal. A DSP can insert interpolated data points into the

**FIGURE 11-29** Digital signal processor architecture.



**FIGURE 11-30** Inserting an interpolated data point into a digital signal to reduce noise.



digital signal. Figure 11-30 shows how 4X oversampling interpolation filtering smooths out the waveform and makes final filtering possible with simpler analog circuitry. DSP performs this role in your CD player to provide an excellent audio reproduction. The round dots represent the digitally recorded data on your CD. The triangles represent the interpolated data points that the digital filter in your CD player inserts before the final analog output filter.

Many of the important concepts that you need to understand in order to move on to DSP have been presented in this and previous chapters. A/D and D/A conversion methods and hardware along with data acquisition and sampling concepts are vital. Topics such as signed binary number representations (including fractions), signed binary addition and multiplication (covered in Chapter 6), and shift registers (Chapter 7) are necessary to understand the hardware and programming of a DSP. Memory system concepts, which will be presented in the next chapter, will also be important.

DSP is being integrated into many common systems that you are familiar with. MP3 players use DSP to filter the digital data being read from the file to decompress the audio data and minimize the quantization noise that is unavoidably caused by digitizing the music. Telephone systems use DSP to cancel echoes on the phone lines. Special-effects boxes for guitars and other instruments perform echo, reverb, phasing, and other effects using DSP. Many digital control systems use DSP as the core of the controller for filtering and implementing the control algorithm. Applications of DSP are growing right now at the same rate that microprocessor applications grew in the early 1980s. They provide a digital solution to many traditionally analog problems. Some other examples of applications include speech recognition, telecommunications data encryption, fast Fourier transforms, image processing in digital television, and ultrasonic beam forming in medical electronics. As this trend continues, you can expect to see nearly all electronic systems containing digital signal processing circuitry.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is a major application of DSP?
2. What is the typical source of digital data for a DSP to process?
3. What advantage does a DSP filter have over an analog filter circuit?
4. What is the central hardware feature of a DSP?
5. How many interpolated data points are inserted between samples when performing 4X oversampled digital filtering? How many for 8X oversampling?

## 11-18 APPLICATIONS OF ANALOG INTERFACING

### OUTCOME

Upon completion of this section, you will be able to:

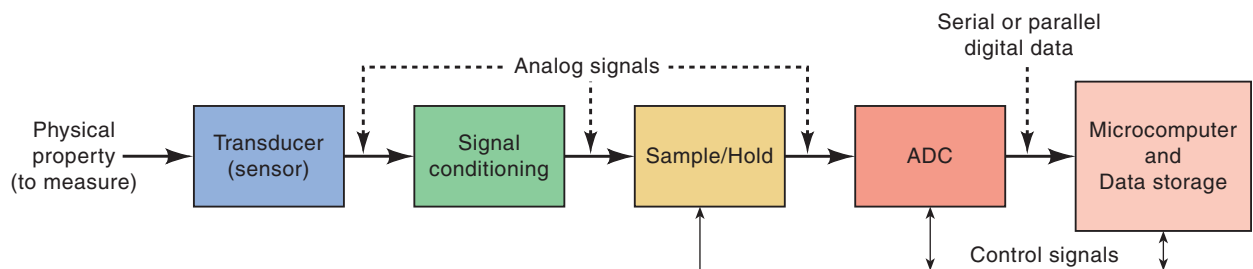
- Identify common applications of A/D and D/A techniques.

The world around us is basically analog, but our electronic systems that deal with the analog information are primarily digital. That, of course, means that it will be necessary to provide interfacing circuitry between the analog portions and the digital portions of the entire system. Including this interfacing circuitry in the system will result in additional system complexity, expense, and processing delays. Our motivation for employing digital is that digital systems have inherent advantages over analog in some very important system characteristics such as speed of operation, system size, accuracy and precision of the processed data, ease of system design, and low overall system cost. Additionally, digital systems are much more flexible than the equivalent analog circuitry would be. Let's look at some common applications that utilize analog interfacing in digital systems.

### Data Acquisition Systems

The process of acquiring and storing digitized information is called data acquisition. Analog data is converted with an ADC, and the resulting binary values are then stored in memory. The data acquisition system may be monitoring many diverse things, such as the environment, industrial processes, or audio and video information. The collected data is usually processed by a computer and is often output to some type of display. The results may also be used to control the operation of a system or process. Computer-based measurement systems are used in a variety of applications.

A typical block diagram of a data acquisition system is shown in Figure 11-31. Transducers (or sensors) are devices that convert one type of physical phenomenon or property (e.g., temperature, strain, pressure, or light) into electrical quantities, such as voltage or resistance. Transducer and ADC characteristics determine the accuracy and resolution of the data acquisition system. These characteristics also define many of the signal-conditioning requirements of the measurement system. The transducer's signals usually have to be changed in some fashion by amplification, attenuation, isolation, or perhaps used in an electrical bridge arrangement. These are all common functions performed by the signal-conditioning block. The sample-and-hold block is used to capture a changing analog signal during the analog-to-digital conversion process. The digital output from the ADC is transferred either serially or in parallel to the computer for storage and processing.

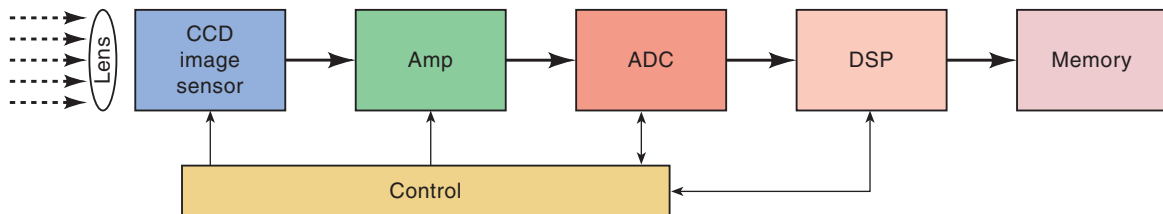


**FIGURE 11-31** Block diagram of a data acquisition system.

For data acquisition of multiple analog input signals, an ADC with multiplexed input channels may be used, along with separate transducers, signal conditioning, and S/H blocks for each analog signal. Heart monitors found in a hospital and the digital storage oscilloscopes that may be in your laboratory are data acquisition systems. Oscilloscopes are designed to measure voltage inputs directly, so they do not have the transducer block in our diagram included.

## Digital Camera

Another familiar application that interfaces analog devices to a digital system is a digital camera. A simplified block diagram of a digital camera is shown in Figure 11-32. This electronic system is very similar to the data acquisition system previously discussed. The transducer used in a digital camera is typically a charge-coupled device (CCD). A CCD consists of a two-dimensional array of capacitors that are connected together to form an analog shift register. An image is projected through the camera's lens onto the surface of the CCD. The light energy will cause each capacitor to accumulate an electric charge that is proportional to the light intensity at that location. The analog signals are then read out of the CCD by shifting the electric charges through the successive capacitors under the control of drivers and timing circuits. The series of analog voltages produced by the capacitor charges are amplified (signal conditioning) and then digitized by the ADC. The DSP block applies an image signal-processing algorithm to the resulting digital data before storing the information in a memory device. The digital data are usually compressed so that it requires less storage space. **Data compression** is the process of encoding information with fewer bits representing the original data. Data compression only works when both the sender and receiver of the information understand the specific encoding scheme.

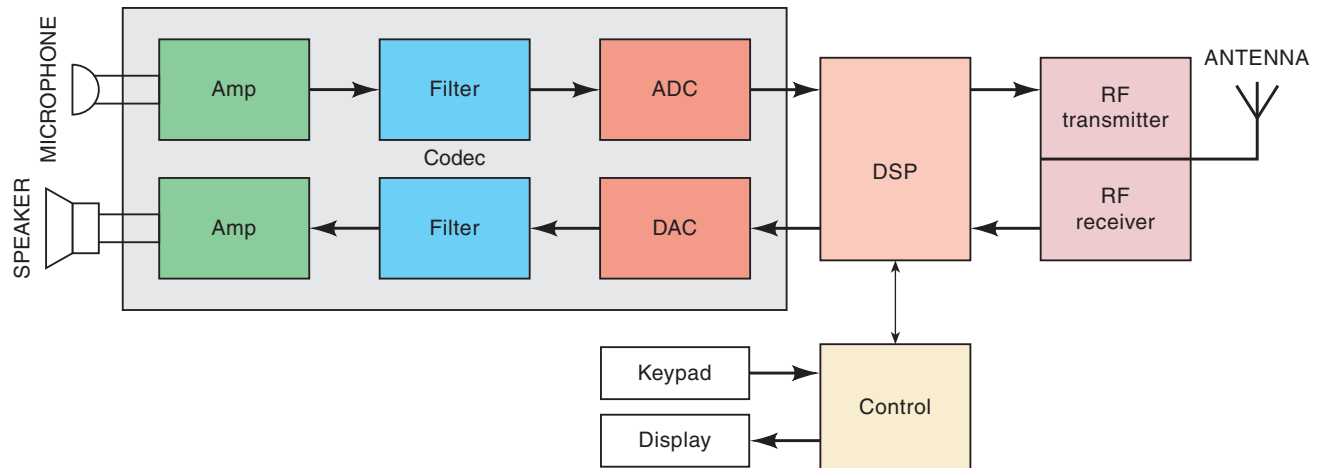


**FIGURE 11-32** Block diagram of a digital camera.

There are many different encoding schemes, so you need to know which one has been applied to be able to decode the compressed information. A common data compression method for photographic images, JPEG, is named after the committee (Joint Photographic Experts Group) that created the standard. This compression method is usually lossy, meaning that some original image information is lost and cannot be restored, thereby possibly affecting the image quality. There are also lossless techniques that can be used for data compression. Factors to consider when choosing to use a data compression scheme include the degree of data compression, the amount of distortion that will be introduced by using a lossy compression technique, and the computational resources required to compress and uncompress the data.

## Digital Cellular Telephone

As our final example of devices that require the interfacing of analog signals to a digital system, let's take another look at the digital cellular telephone. A simplified block diagram of a digital cell phone is shown in Figure 11-33.



**FIGURE 11-33** Simplified block diagram of a digital cellular telephone.

The analog voice signal is picked up with a microphone and needs to be amplified and filtered to reduce its bandwidth (signal conditioning, again) before it is digitized by the ADC. Likewise, the received digital information must be converted back into an analog signal by the DAC. The DAC's analog output then goes through a low-pass filter to recover the lower-frequency voice information that will be amplified so that it can be heard from the phone's speaker. The encoding of the analog voice information into a digital format and decoding back again to an analog output is handled by the voice **codec**. A hardware codec combines the *coder* (ADC) and the *decoder* (DAC) functions as well as the signal conditioning in one complex IC chip. The DSP block performs several processing functions, including data compression and encryption of our digitized speech information from the ADC (or our text and picture data). Data compression reduces the bandwidth requirements for transmission, and encryption provides secure transmission of our data. The RF transmitter block performs modulation and amplification of the radio-frequency signal that is sent to the cell tower. The RF receiver block amplifies the received radio frequency signal from the cell tower and demodulates it to obtain the digital information that is then processed by the DSP block. The DSP block provides decryption and data decompression of the received voice information or other data that has been sent to our phone.

There are a few other important tasks that your cell phone has to be able to do in the background without you even worrying about it. It has to communicate with the nearest cell tower to just let the system know that your phone is there; it has to automatically select communication channels for your call; and it has to be able to hand off your call to another tower as you and your phone move to another cell. It's really a pretty amazing piece of electronics!

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. List 3 applications of analog interfacing.



## SUMMARY

---

1. Physical variables that we want to measure, such as temperature, pressure, humidity, distance, or velocity, are continuously variable quantities. A transducer can be used to translate these quantities into an electrical signal of voltage or current that fluctuates in proportion to the physical variable. These continuously variable voltage or current signals are called *analog* signals.
  2. To measure a physical variable, a digital system must assign a binary number to the analog value that is present at that instant. This is accomplished by an A/D converter. To generate variable voltages or current values that can control physical processes, a digital system must translate binary numbers into a voltage or current magnitude. This is accomplished by a D/A converter.
  3. A D/A converter with  $n$  bits divides a range of analog values (voltage or current) into  $2^n - 1$  pieces. The size or magnitude of each piece is the analog equivalent weight of the least significant bit. This is called the *resolution* or *step size*.
  4. Most D/A converters use resistor networks that can cause weighted amounts of current to flow when any of its binary inputs are activated. The amount of current that flows is proportional to the binary weight of each input bit. These weighted currents are summed to create the analog signal out.
  5. An A/D converter must assign a binary number to an analog (continuously variable) quantity. The precision with which an A/D converter can perform this conversion depends on how many different numbers it can assign and how wide the analog range is. The smallest change in analog value that an A/D can measure is called its *resolution*, the weight of its least significant bit.
  6. By repeatedly sampling the incoming analog signal, converting it to digital, and storing the digital values in a memory device, an analog waveform can be captured. To reconstruct the signal, the digital values are read from the memory device at the same rate at which they were stored, and then they are fed into a D/A converter. The output of the D/A is then filtered to smooth the stair steps and re-create the original waveform. The bandwidth of sampled signals is limited to  $\frac{1}{2}F_S$ . Incoming frequencies greater than  $\frac{1}{2}F_S$  create an *alias* that has a frequency equal to the difference between the nearest integer multiple of  $F_S$  and the incoming frequency. This difference will always be less than  $\frac{1}{2}F_S$ .
  7. A digital-ramp A/D is the simplest to understand but it is not often used due to its variable conversion time. A successive-approximation converter has a constant conversion time and is probably the most common general-purpose converter.
  8. Flash converters use analog comparators and a priority encoder to assign a digital value to the analog input. These are the fastest converters because the only delays involved are propagation delays.
  9. Other popular methods of A/D include pipelined, integrating, voltage-to-frequency conversion, and sigma/delta conversion. Each type of converter has its own niche of applications.
  10. Any D/A converter can be used with other circuitry such as analog multiplexers that select one of several analog signals to be converted, one at a time. S/H circuits can be used to “freeze” a rapidly changing analog signal while the conversion is taking place.
-

11. Digital signal processing is an exciting new growth field in electronics. These devices allow calculations to be performed quickly in order to emulate the operation of many analog filter circuits digitally. The primary architectural feature of a DSP is a hardware multiplier and adder circuit that can multiply pairs of numbers together and accumulate the running total (sum) of these products. This circuitry is used to perform efficiently the weighted moving average calculations that are used to implement digital filters and other DSP functions. DSP is responsible for many of the recent advances in high-fidelity audio, high-definition TV, and telecommunications.

## IMPORTANT TERMS

digital quantity	digital-ramp ADC	sigma/delta
analog quantity	quantization error	modulation
transducer	sampling	pipelined ADC
analog-to-digital	sampling	sample-and-hold
converter (ADC)	frequency, $F_S$	(S/H) circuit
digital-to-analog	alias	acquisition time
converter (DAC)	undersampling	digital signal
full-scale output	successive-	processing (DSP)
resolution	approximation	weighted average
step size	ADC	MAC
staircase	differential inputs	arithmetic logic unit
full-scale error	WRITE	barrel shifter
linearity error	flash ADC	oversampling
offset error	dual-slope ADC	interpolation filtering
settling time	voltage-to-frequency	data compression
monotonicity	ADC	codec
digitization		

## PROBLEMS

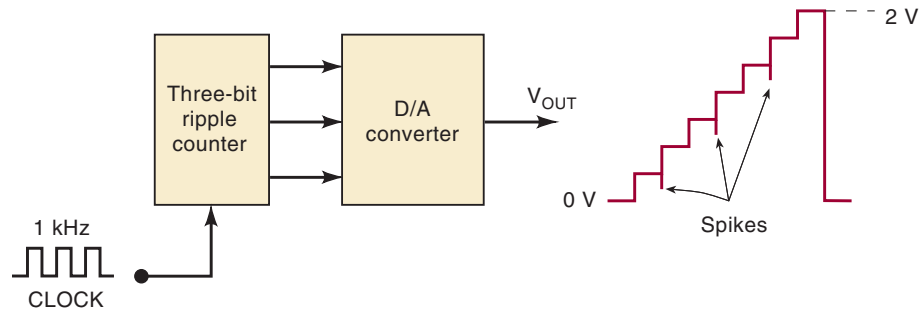
### SECTIONS 11-1 AND 11-2

- B** 11-1. **DRILL QUESTIONS**
- What is the expression relating the output and inputs of a DAC?
  - Define *step size* of a DAC.
  - Define *resolution* of a DAC.
  - Define *full scale*.
  - Define *percentage resolution*.
  - \* *True or false*: A 10-bit DAC will have a smaller resolution than a 12-bit DAC for the same full-scale output.
  - \* *True or false*: A 10-bit DAC with full-scale output of 10 V has a smaller percentage resolution than a 10-bit DAC with 12 V full scale.
- B** 11-2. An eight-bit DAC produces an output voltage of 2.0 V for an input code of 01100100. What will the value of  $V_{OUT}$  be for an input code of 10110011?
- B** 11-3.\* Determine the weight of each input bit for the DAC of Problem 11-2.

\*Answers to problems marked with an asterisk can be found in the back of the text.

- B** 11-4. What is the resolution of the DAC of Problem 11-2? Express it in volts and as a percentage.
- B** 11-5.\* What is the resolution in volts of a 10-bit DAC whose F.S. output is 5 V?
- B** 11-6. How many bits are required for a DAC so that its F.S. output is 10 mA and its resolution is less than  $40\ \mu\text{A}$ ?
- B** 11-7.\* What is the percentage resolution of the DAC of Figure 11-34? What is the step size if the top step is 2 V?

**FIGURE 11-34** Problems 11-7 and 11-8.



- C** 11-8. What is the cause of the negative-going spikes on the  $V_{\text{OUT}}$  waveform of Figure 11-34? (*Hint*: Note that the counter is a ripple counter and that the spikes occur on every other step.)
- B** 11-9.\* Assuming a 12-bit DAC with perfect accuracy, how close to 250 rpm can the motor speed be adjusted in Figure 11-4?
- 11-10. A 12-bit DAC has a full-scale output of 15.0 V. Determine the step size, the percentage resolution, and the value of  $V_{\text{OUT}}$  for an input code of 011010010101.
- 11-11.\* A microcontroller has an eight-bit output port that is to be used to drive a DAC. The DAC that is available has 10 input bits and has a full-scale output of 10 V. The application requires a voltage that ranges between 0 and 10 V in steps of 50 mV or smaller. Which eight bits of the 10-bit DAC will be connected to the output port?
- 11-12. You need a DAC that can span 12 V with a resolution of 20 mV or less. How many bits are needed?

### SECTION 11-3

- D** 11-13.\* The step size of the DAC of Figure 11-5 can be changed by changing the value of  $R_F$ . Determine the required value of  $R_F$  for a step size of 0.5 V. Will the new value of  $R_F$  change the percentage resolution?
- D** 11-14. Assume that the output of the DAC in Figure 11-7(a) is connected to the op-amp of Figure 11-7(b).
- (a) With  $V_{\text{REF}} = 5\ \text{V}$ ,  $R = 20\ \text{k}\Omega$ , and  $R_F = 10\ \text{k}\Omega$ , determine the step size and the full-scale voltage at  $V_{\text{OUT}}$ .
- (b) Change the value of  $R_F$  so that the full-scale voltage at  $V_{\text{OUT}}$  is  $-2\ \text{V}$ .
- (c) Use this new value of  $R_F$ , and determine the proportionality factor,  $K$ , in the relationship  $V_{\text{OUT}} = K(V_{\text{REF}} \times B)$ . ( $B$  is the binary input value.)
- 11-15.\* What is the advantage of the DAC of Figure 11-8 over that of Figure 11-7, especially for a larger number of input bits?

## SECTIONS 11-4 TO 11-6

- 11-16. An eight-bit DAC has a full-scale error of 0.2% F.S. If the DAC has a full-scale output of 10 mA, what is the most that it can be in error for any digital input? If the D/A output reads  $50\ \mu\text{A}$  for a digital input of 0000001, is this within the specified range of accuracy? (Assume no offset error.)
- C 11-17. The control of a positioning device may be achieved using a *servomotor*, which is a motor designed to drive a mechanical device as long as an error signal exists. Figure 11-35 shows a simple servo-controlled system that is controlled by a digital input that could be coming directly from a computer or from an output medium such as magnetic tape. The lever arm is moved vertically by the servomotor. The motor rotates clockwise or counterclockwise, depending on whether the voltage from the power amplifier (P.A.) is positive or negative. The motor stops when the P.A. output is 0.

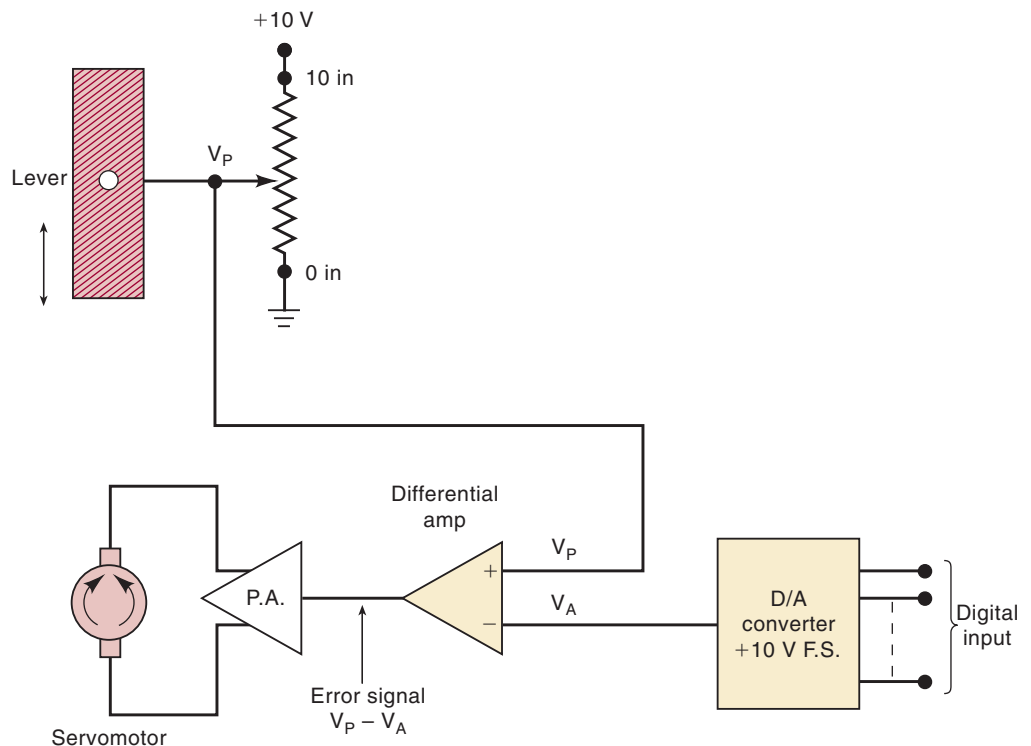


FIGURE 11-35 Problem 11-17.

The mechanical position of the lever is converted to a DC voltage by the potentiometer arrangement shown. When the lever is at its 0 reference point,  $V_P = 0\text{ V}$ . The value of  $V_P$  increases at the rate of 1 V/inch until the lever is at its highest point (10 inches) and  $V_P = 10\text{ V}$ . The desired position of the lever is provided as a digital code from the computer and is then fed to a DAC, producing  $V_A$ . The *difference* between  $V_P$  and  $V_A$  (called *error*) is produced by the *differential* amplifier and is amplified by the P.A. to drive the motor in the direction that causes the error signal to decrease to 0—that is, moves the lever until  $V_P = V_A$ .

- (a)\*If the lever must be positioned within a resolution of 0.1 in, what is the number of bits needed in the digital input code?

- (b) In actual operation, the lever arm might oscillate slightly around the desired position, especially if a *wire-wound* potentiometer is used. Can you explain why?

**B 11-18. DRILL QUESTIONS**

- (a) Define *binary-weighted resistor network*.  
 (b) Define *R/2R ladder network*.  
 (c) Define *DAC settling time*.  
 (d) Define *full-scale error*.  
 (e) Define *offset error*.

11-19.\*A particular six-bit DAC has a full-scale output rated at 1.260 V. Its accuracy is specified as  $\pm 0.1\%$  F.S., and it has an offset error of  $\pm 1$  mV. Assume that the offset error has not been zeroed out. Consider the measurements made on this DAC (Table 11-10), and determine which of them are not within the device's specifications. (*Hint: The offset error is added to the error caused by component inaccuracies.*)

**TABLE 11-10** Data for Problem 11-19.

Input Code	Output
000010	41.5 mV
000111	140.2 mV
001100	242.5 mV
111111	1.258 V

**SECTION 11-7**

- T 11-20.** A certain DAC has the following specifications: eight-bit resolution, full scale = 2.55 V, offset  $\leq 2$  mV; accuracy =  $\pm 0.1\%$  F.S. A static test on this DAC produces the results shown in Table 11-11. What is the probable cause of the malfunction?

**TABLE 11-11** Data for Problem 11-20

Input Code	Output
00000000	8 mV
00000001	18.2 mV
00000010	28.5 mV
00000100	48.3 mV
00001111	158.3 mV
10000000	1.289 V

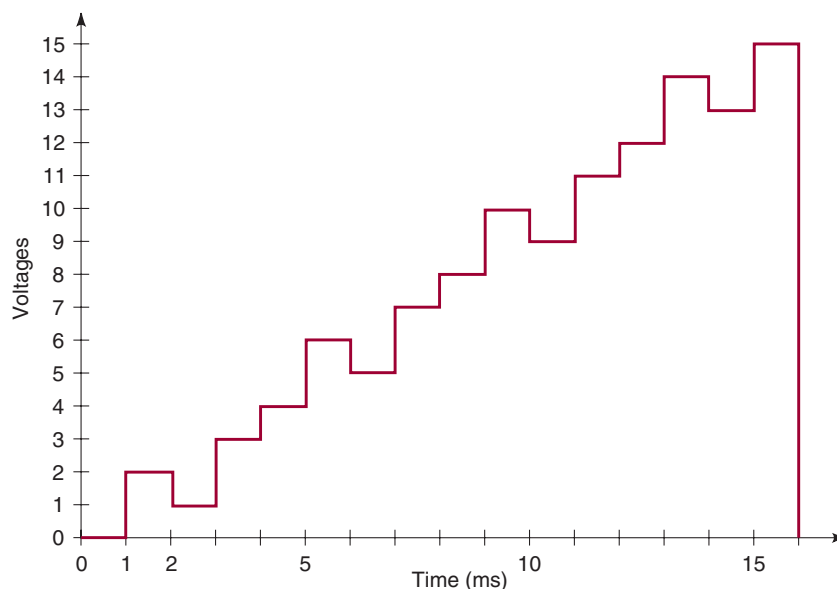
11-21.\*Repeat Problem 11-20 using the measured data given in Table 11-12.

**TABLE 11-12** Data for Problem 11-21.

Input Code	Output
00000000	20.5 mV
00000001	30.5 mV
00000010	20.5 mV
00000100	60.6 mV
00001111	150.6 mV
10000000	1.300 V

- T** 11-22.\* A technician connects a counter to the DAC of Figure 11-3 to perform a staircase test using a 1-kHz clock. The result is shown in Figure 11-36. What is the probable cause of the incorrect staircase signal?

**FIGURE 11-36** Problem 11-22.



### SECTIONS 11-8 AND 11-9

#### 11-23. DRILL QUESTIONS

Fill in the blanks in the following description of the ADC of Figure 11-13. Each blank may be one or more words.

A START pulse is applied to \_\_\_\_ the counter and to keep \_\_\_\_ from passing through the AND gate into the \_\_\_\_\_. At this point, the DAC output,  $V_{AX}$ , is \_\_\_\_ and  $\overline{EOC}$  is \_\_\_\_.

When START returns \_\_\_\_, the AND gate is \_\_\_\_, and the counter is allowed to \_\_\_\_\_. The  $V_{AX}$  signal is increased one \_\_\_\_ at a time until it \_\_\_\_  $V_A$ . At that point, \_\_\_\_ goes LOW to \_\_\_\_ further pulses from \_\_\_\_\_. This signals the end of conversion, and the digital equivalent of  $V_A$  is present at the \_\_\_\_\_.

- B** 11-24. An eight-bit digital-ramp ADC with a 40-mV resolution uses a clock frequency of 2.5 MHz and a comparator with  $V_T = 1$  mV. Determine the following values.
- \*The digital output for  $V_A = 6.000$  V
  - The digital output for 6.035 V
  - The maximum and average conversion times for this ADC
- B** 11-25. Why were the digital outputs the same for parts (a) and (b) of Problem 11-24?
- D** 11-26. What would happen in the ADC of Problem 11-24 if an analog voltage of  $V_A = 10.853$  V were applied to the input? What waveform would appear at the D/A output? Incorporate the necessary logic in this ADC so that an “overscale” indication will be generated whenever  $V_A$  is too large.
- B** 11-27.\* An ADC has the following characteristics: resolution, 12 bits; full-scale error, 0.03% F.S.; full-scale output, +5 V.
- What is the quantization error in volts?
  - What is the total possible error in volts?

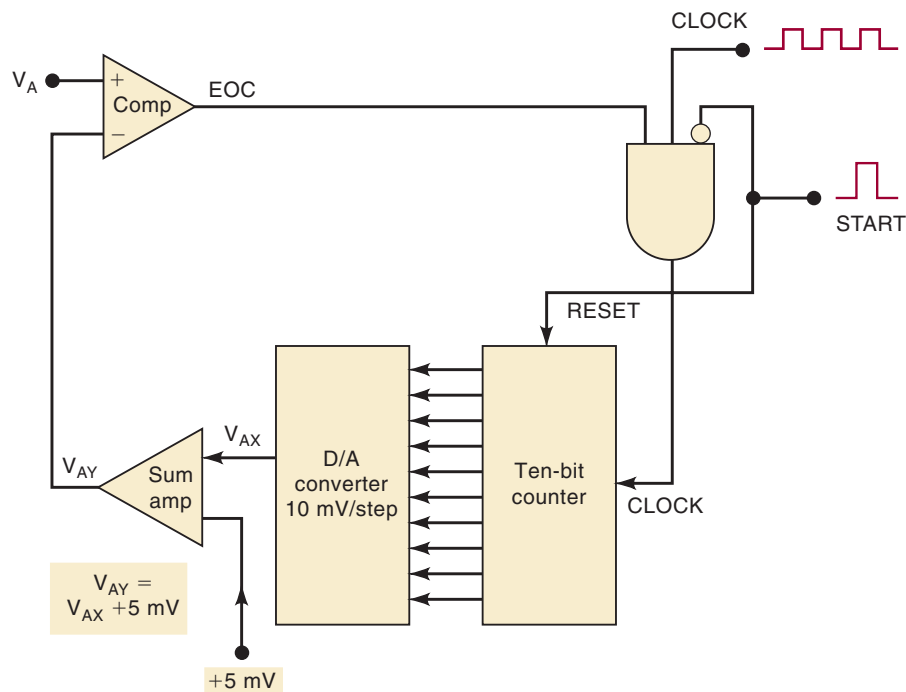
- C 11-28. The quantization error of an ADC such as the one in Figure 11-13 is always positive because the  $V_{AX}$  value must exceed  $V_A$  in order for the comparator output to switch states. This means that the value of  $V_{AX}$  could be as much as 1 LSB greater than  $V_A$ . This quantization error can be modified so that  $V_{AX}$  would be within  $\pm\frac{1}{2}$  LSB of  $V_A$ . This can be done by adding a fixed voltage equal to  $\pm\frac{1}{2}$  LSB ( $\pm\frac{1}{2}$  step) to the value of  $V_A$ . Figure 11-37 shows this symbolically for a converter that has a resolution of 10 mV/step. A fixed voltage of +5 mV is added to the D/A output in the summing amplifier, and the result,  $V_{AY}$ , is fed to the comparator, which has  $V_T = 1$  mV.

For this modified converter, determine the digital output for the following  $V_A$  values.

- (a)  $V_A = 5.022$  V  
 (b)  $V_A = 50.28$  V

Determine the quantization error in each case by comparing  $V_{AX}$  and  $V_A$ . Note that the error is positive in one case and negative in the other.

**FIGURE 11-37** Problems 11-28 and 11-29.



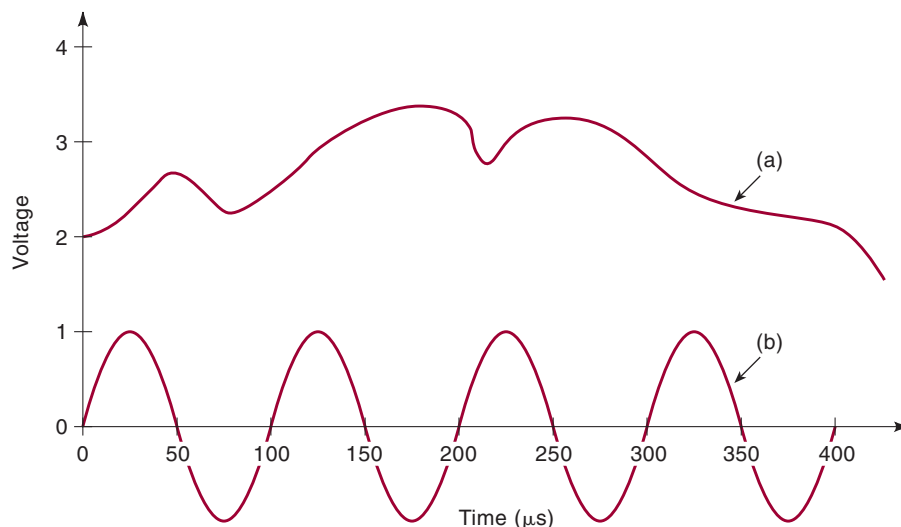
- C 11-29. For the ADC of Figure 11-37, determine the range of analog input values that will produce a digital output of 0100011100.

### SECTION 11-10

- 11-30. Assume that the analog signal in Figure 11-38(a) is to be digitized by performing continuous A/D conversions using an eight-bit digital-ramp converter whose staircase rises at the rate of 1 V every  $25 \mu\text{s}$ . Sketch the reconstructed signal using the data obtained during the digitizing process. Compare it with the original signal, and discuss what could be done to make it a more accurate representation.
- C 11-31.\* On the sine wave of Figure 11-38(b), mark the points where samples are taken by a flash A/D converter at intervals of  $75 \mu\text{s}$  (starting at the origin). Then draw the reconstructed output from the D/A converter

(connect the sample points with straight lines to show filtering). Calculate the sample frequency, the sine input frequency, and the difference between them. Then compare to the resulting reconstructed waveform frequency.

**FIGURE 11-38** Problems 11-30, 11-31, and 11-41.



11-32. A sampled data acquisition system is being used to digitize an audio signal. Assume the sample frequency  $F_S$  is 20 kHz. Determine the output frequency that will be heard for each of the following input frequencies.

- (a)\*Input signal = 5 kHz
- (b)\*Input signal = 10.1 kHz
- (c) Input signal = 10.2 kHz
- (d) Input signal = 15 kHz
- (e) Input signal = 19.1 kHz
- (f) Input signal = 19.2 kHz

### SECTION 11-11

#### B 11-33.\* DRILL QUESTIONS

Indicate whether each of the following statements refers to the digital-ramp ADC, the successive-approximation ADC, or both.

- (a) Produces a staircase signal at its DAC output
- (b) Has a constant conversion time independent of  $V_A$
- (c) Has a shorter average conversion time
- (d) Uses an analog comparator
- (e) Uses a DAC
- (f) Uses a counter
- (g) Has complex control logic
- (h) Has an  $\overline{EOC}$  output

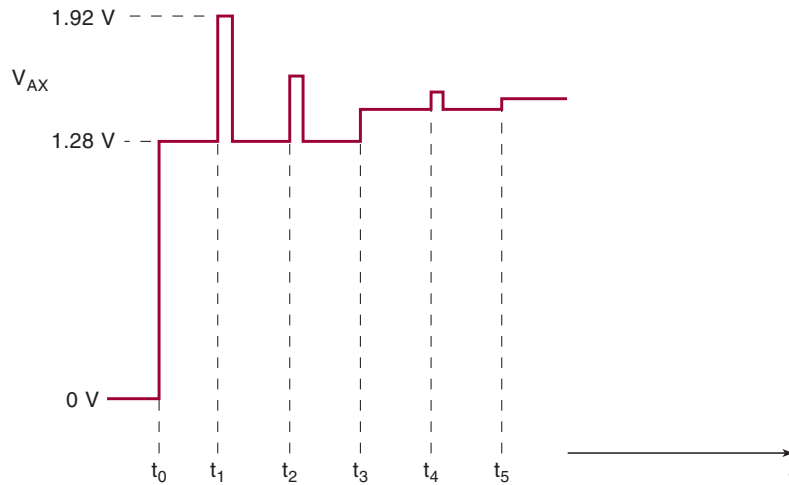
11-34. Draw the waveform for  $V_{AX}$  as the SAC of Figure 11-19 converts  $V_A = 6.7$  V.

11-35. Repeat Problem 11-34 for  $V_A = 16$  V.



- B** 11-36.\*A certain eight-bit successive-approximation converter has 2.55 V full scale. The conversion time for  $V_A = 1$  V is  $80 \mu\text{s}$ . What will be the conversion time for  $V_A = 1.5$  V?
- 11-37. Figure 11-39 shows the waveform at  $V_{AX}$  for a six-bit SAC with a step size of 40 mV during a complete conversion cycle. Examine this waveform and describe what is occurring at times  $t_0$  to  $t_5$ . Then determine the resultant digital output.

**FIGURE 11-39** Problem 11-37.



- B** 11-38.\*Refer to Figure 11-21. What is the approximate value of the analog input if the microcomputer's data bus is at 10010111 when  $\overline{RD}$  is pulsed LOW?
- D** 11-39. Connect a 2.0-V reference source to  $V_{REF}/2$ , and repeat Problem 11-38.
- C, D** 11-40.\*Design the ADC interface to a digital thermostat using an LM34 temperature sensor and the ADC0804. Your system must measure accurately ( $\pm 0.2^\circ\text{F}$ ) from  $50$  to  $101^\circ\text{F}$ . The LM34 puts out  $0.01$  V per degree F ( $0^\circ\text{F} = 0$  V).
- What should the digital value for  $50^\circ\text{F}$  be for the best resolution?
  - What voltage must be applied to  $V_{IN}(-)$ ?
  - What is the full-scale range of voltage that will come in?
  - What voltage must be applied to  $V_{REF}/2$ ?
  - What binary value will represent  $72^\circ\text{F}$ ?
  - What is the resolution in  $^\circ\text{F}$ ? In volts?

### SECTION 11-12

- B** 11-41. Discuss how a flash ADC with a conversion time of  $1 \mu\text{s}$  would work for the situation of Problem 11-30.
- D** 11-42. Draw the circuit diagram for a four-bit flash converter with BCD output and a resolution of  $0.1$  V. Assume that a  $+5$  V precision supply voltage is available.

### DRILL QUESTION

- B** 11-43. For each of the following statements, indicate which type of ADC—digital-ramp, SAC, or flash—is being described.
- Fastest method of conversion
  - Needs a START pulse

- (c) Requires the most circuitry
- (d) Does not use a DAC
- (e) Generates a staircase signal
- (f) Uses an analog comparator
- (g) Has a relatively fixed conversion time independent of  $V_A$

### SECTION 11-13

#### B 11-44. DRILL QUESTIONS

For each statement, indicate what type(s) of ADC is (are) being described.

- (a) Uses subranging stages
- (b) Uses a large number of comparators
- (c) Uses a VCO
- (d) Is used in noisy industrial environments
- (e) Uses a capacitor
- (f) Is relatively insensitive to temperature

### SECTIONS 11-15 AND 11-16

- T 11-45.\* Refer to the S/H circuit of Figure 11-26. What circuit fault would result in  $V_{OUT}$  looking exactly like  $V_A$ ? What fault would cause  $V_{OUT}$  to be stuck at 0?
- C, D 11-46. Use the CMOS 4016 IC (Section 8-15) to implement the switching in Figure 11-27, and design the necessary control logic so that each analog input is converted to its digital equivalent in sequence. The ADC is a 10-bit, successive-approximation type using a 50-kHz clock signal, and it requires a 10- $\mu$ s-duration start pulse to begin each conversion. The digital outputs are to remain stable for 100  $\mu$ s after the conversion is complete before switching to the next analog input. Choose an appropriate multiplexing clock frequency.

### MICROCOMPUTER APPLICATION

- C, D 11-47.\* Figure 11-21 shows how the ADC0804 is interfaced to a microcomputer. It shows three control signals,  $\overline{CS}$ ,  $\overline{RD}$ , and  $\overline{WR}$ , that come from the microcomputer to the ADC. These signals are used to start each new A/D conversion and to read (transfer) the ADC data output into the microcomputer over the data bus.

Figure 11-40 shows one way the address decoding logic could be implemented. The  $\overline{CS}$  signal that activates the ADC0804 is developed from the eight high-order address lines of the MPU address bus. Whenever the MPU wants to communicate with the ADC0804, it places the address of the ADC0804 onto the address bus, and the decoding logic drives the  $\overline{CS}$  signal LOW. Notice that in addition to the address lines, a timing and control signal ( $ALE$ ) is connected to the  $\overline{E}_2$  enable input. Whenever  $ALE$  is HIGH, it means that the address is potentially in transition, so the decoder should be disabled until  $ALE$  goes LOW (at which time the address will be valid and stable). This serves a purpose for timing but has no effect on the address of the ADC.

- (a) Determine the address of the ADC0804.
- (b) Modify the diagram of Figure 11-40 to place the ADC0804 at address E8XX hex.
- (c) Modify the diagram of Figure 11-40 to place the ADC0804 at address FFXX hex.

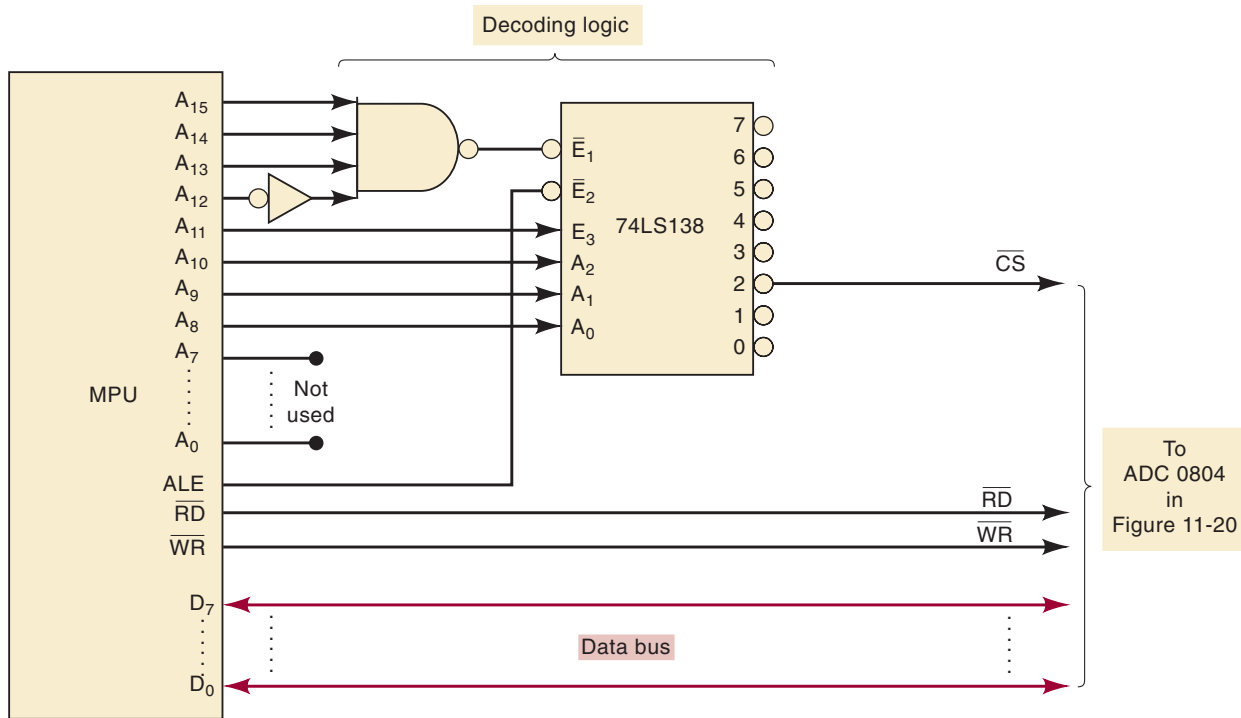


FIGURE 11-40 Problem 11-47: MPU interfaced to the ADC0804 of Figure 11-20.

- D 11-48. You have available a 10-bit SAC A/D converter (AD 573), but your system requires only eight bits of resolution and you have only eight port bits available on your microprocessor. Can you use this A/D converter, and if so, which of the 10 data lines will you attach to the port?

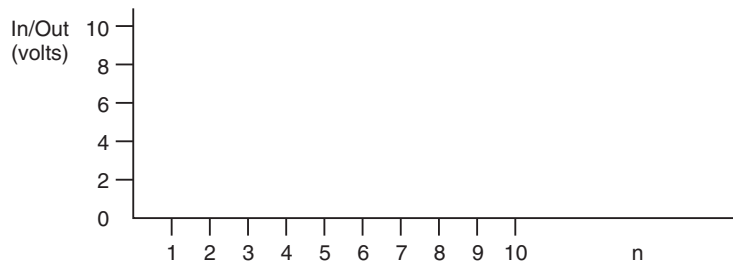
**SECTION 11-17**

11-49. The data in Table 11-13 are input samples taken by an A/D converter. Notice that if the input data were plotted, it would represent a simple step function like the rising edge of a digital signal. Calculate the simple average of the four most recent data points, starting with OUT[4] and proceeding through OUT[10]. Plot the values for IN and OUT against the sample number *n* as shown in Figure 11-41.

TABLE 11-13

Sample <i>n</i>	1	2	3	4	5	6	7	8	9	10
IN[ <i>n</i> ] (V)	0	0	0	0	10	10	10	10	10	10
OUT[ <i>n</i> ] (V)	0	0	0							

FIGURE 11-41 Graph format for Problems 11-49 and 11-50.



Sample calculations:

$$\text{OUT}[n] = (\text{IN}[n - 3] + \text{IN}[n - 2] + \text{IN}[n - 1] + \text{IN}[n])/4 = 0$$

$$\text{OUT}[4] = (\text{IN}[1] + \text{IN}[2] + \text{IN}[3] + \text{IN}[4])/4 = 0$$

$$\text{OUT}[5] = (\text{IN}[2] + \text{IN}[3] + \text{IN}[4] + \text{IN}[5])/4 = 2.5$$

(Notice that this calculation is equivalent to multiplying each sample by  $\frac{1}{4}$  and summing.)

- 11-50. Repeat the previous problem using a weighted average of the last four samples. The weights in this case are placing greater emphasis on recent samples and less emphasis on older samples. Use the weights 0.1, 0.2, 0.3, and 0.4.

$$\text{OUT}[n] = 0.1(\text{IN}[n - 3]) + 0.2(\text{IN}[n - 2]) + 0.3(\text{IN}[n - 1]) + 0.4(\text{IN}[n])$$

$$\text{OUT}[5] = 0.1(\text{IN}[2]) + 0.2(\text{IN}[3]) + 0.3(\text{IN}[4]) + 0.4(\text{IN}[5]) = 4$$

- 11-51. What does the term MAC stand for?

#### 11-52.\* DRILL QUESTIONS

*True or false:*

- (a) A digital signal is a continuously changing voltage.
- (b) A digital signal is a sequence of numbers that represent an analog signal.

When processing an analog signal, the output may be distorted due to:

- (a) Quantization error when converting analog to digital
- (b) Not sampling the original signal frequently enough
- (c) Temperature variation in the processor components
- (d) The high-frequency components associated with sudden changes in voltage out of the DAC
- (e) Electrical noise on the power supply
- (f) Alias signals introduced by the digital system

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

---

### SECTION 11-1

1. Converts a nonelectrical physical quantity to an electrical quantity
2. Converts an analog voltage or current to a digital representation
3. Stores it; performs calculation or some other operation on it
4. Converts digital data to their analog representation
5. Controls a physical variable according to an electrical input signal

### SECTION 11-2

1.  $40 \mu\text{A}$ ; 10.2 mA
2. 5.12 mA
3. 0.39 percent
4. 4096
5. 12
6. True
7. It produces a greater number of possible analog outputs between 0 and full scale.

### SECTION 11-3

1. It uses only two different sizes of resistors.
2. 640 k $\Omega$
3. 0.5 V
4. Increases by 20 percent

### SECTION 11-4

1. Maximum deviation of DAC output from its ideal value, expressed as percentage of full scale
  2. Time it takes DAC output to settle to within  $\frac{1}{2}$  step size of its full-scale value when the digital input changes from 0 to full scale
  3. Offset error adds a small constant positive or negative value to the expected analog
-

output for any digital input. 4. Because of the response time of the op-amp current-to-voltage converter

### SECTION 11-5

1. Yes. 2. It provides the enable signal for the input data latch. The latch is enabled when CS and WR are asserted. 3. 1.  $V_{REF}$  range  $\pm 25$  V, eight-bit resolution, accuracy  $\pm 0.2\%$  F.S., settling time 100 ns

### SECTION 11-6

1. Control, automatic testing, signal reconstruction, A/D conversion

### SECTION 11-7

1. Drift, opens/shorts on inputs, faulty  $V_{REF}$ , offset error

### SECTION 11-8

1. Tells control logic when the DAC output exceeds the analog input 2. At outputs of register 3. Tells us when conversion is complete and digital equivalent of  $V_A$  is at register outputs

### SECTION 11-9

1. The digital input to a DAC is incremented until the DAC staircase output exceeds the analog input. 2. The built-in error caused by the fact that  $V_{AX}$  does not continuously increase but goes up in steps equal to the DAC's resolution. The final  $V_{AX}$  can be different from  $V_A$  by as much as one step size. 3. If  $V_A$  increases, it will take more steps before  $V_{AX}$  can reach the step that first exceeds  $V_A$ . 4. True 5. Simple circuit; relatively long conversion time that changes with  $V_A$  6.  $0010000111_2 = 135_{10}$  for both cases

### SECTION 11-10

1. Process of converting different points on an analog signal to digital and storing the digital data for later use 2. Computer generates START signal to begin an A/D conversion of the analog signal. When  $\overline{EOC}$  goes LOW, it signals the computer that the conversion is complete. The computer then loads the ADC output into memory. The process is repeated for the next point on the analog signal. 3. Twice the highest frequency in the input signal 4. An alias frequency will be present in the output.

### SECTION 11-11

1. The SAC has a shorter conversion time that doesn't change with  $V_A$ . 2. It has more complex control logic. 3. False 4. (a) 8 (b) 0–5 V (c)  $\overline{CS}$  controls the effect of the  $\overline{RD}$  and  $\overline{WR}$  signals;  $\overline{WR}$  is used to start a new conversion;  $\overline{RD}$  enables the output buffers. (d) When LOW, it signals the end of a conversion. (e) It separates the usually noisy digital ground from the analog ground so as not to contaminate the analog input signal. (f) All analog voltages on  $V_{in}(+)$  are measured with reference to this pin. This allows the input range to be offset from ground.

### SECTION 11-12

1. True 2. 4095 comparators and 4096 resistors 3. Major advantage is its conversion speed; disadvantage is the number of required circuit components for a practical resolution.

### SECTION 11-13

1. Finer resolutions of the analog input 2. A VCO 3. Advantages: low cost, temperature immunity; disadvantage: slow conversion time 4. Flash ADC, voltage-to-frequency ADC, and dual-slope ADC 5. One

**SECTION 11-14**

1. Audio, motor control, sensors.
2. DC voltages and DVMs.
3. Video, oscilloscopes, medical imaging
4. Successive-approximation ADC.

**SECTION 11-15**

1. It takes a sample of an analog voltage signal and stores it on a capacitor.
2. False; they are unity-gain buffers with high input impedance and low output impedance.

**SECTION 11-16**

1. Uses a single ADC
2. It would become a MOD-8 counter.

**SECTION 11-17**

1. Filtering analog signals
2. An A/D converter
3. To change their dynamic response, you simply change the numbers in the software program, not the hardware components.
4. The Multiply and Accumulate (MAC) unit
5. 3; 7

**SECTION 11-18**

1. Data acquisition, digital cameras, cell phones.
-



# MEMORY DEVICES

## ■ OUTLINE

- 12-1 Memory Terminology
- 12-2 General Memory Operation
- 12-3 CPU–Memory Connections
- 12-4 Read-Only Memories
- 12-5 ROM Architecture
- 12-6 ROM Timing
- 12-7 Types of ROMs
- 12-8 Flash Memory
- 12-9 ROM Applications
- 12-10 Semiconductor RAM
- 12-11 RAM Architecture
- 12-12 Static RAM (SRAM)
- 12-13 Dynamic RAM (DRAM)
- 12-14 Dynamic RAM Structure and Operation
- 12-15 DRAM Read/Write Cycles
- 12-16 DRAM Refreshing
- 12-17 DRAM Technology
- 12-18 Other Memory Technologies
- 12-19 Expanding Word Size and Capacity
- 12-20 Special Memory Functions

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Define terminology associated with memory systems.
- Describe the difference between read/write memory and read-only memory.
- Discuss the difference between volatile and nonvolatile memory.
- Determine the capacity of a memory device from its inputs and outputs.
- Outline the steps that occur when the CPU reads from or writes to memory.
- Distinguish among the various types of ROMs and cite some common applications.
- Describe the organization and operation of static and dynamic RAMs.
- Compare the relative advantages and disadvantages of EPROM, EEPROM, and flash memory.
- Combine memory ICs to form memory modules with larger word size and/or capacity.

## ■ INTRODUCTION

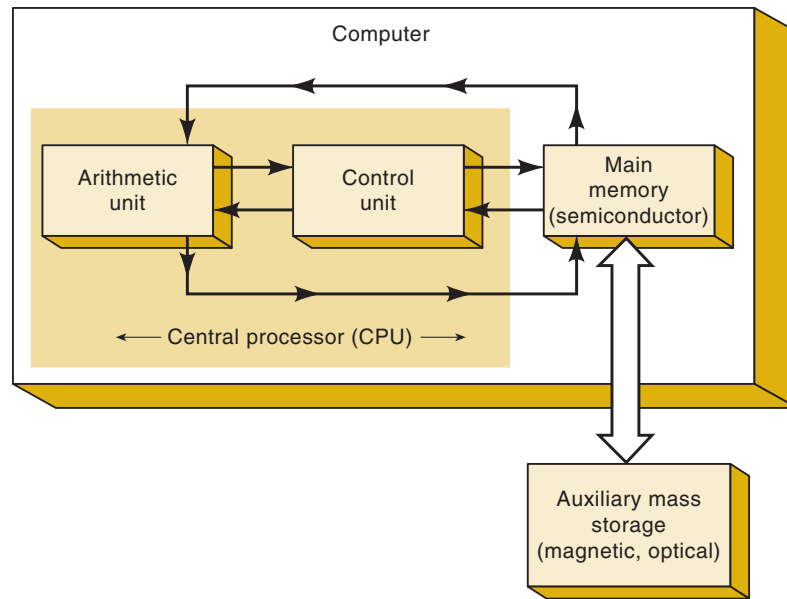
A major advantage of digital over analog systems is the ability to easily store large quantities of digital information and data for short or long periods. This memory capability is what makes digital systems so versatile and adaptable to many situations. For example, in a digital computer, the internal main memory stores instructions that tell the computer what to do under *all* possible circumstances so that the computer will do its job with a minimum amount of human intervention.

This chapter is devoted to a study of the most commonly used types of memory devices and systems. We have already become very familiar with the flip-flop, which is an electronic memory device. We have also seen how groups of FFs called *registers* can be used to store information and how this information can be transferred to other locations. FF registers are high-speed memory elements that are used extensively in the internal operations of a digital computer, where digital information is continually being moved from one location to another. Advances in LSI and VLSI technology have made it possible to obtain large numbers of FFs on a single chip arranged in various memory-array formats. These bipolar and MOS semiconductor memories are the fastest memory devices available, and their cost has been continuously decreasing as VLSI technology improves.

Digital data can also be stored as charges on capacitors, and a very important type of semiconductor memory uses this principle to obtain high-density storage at low power-requirement levels.



**FIGURE 12-1** A computer system normally uses high-speed main memory and slower external auxiliary memory.



Semiconductor memories are used as the **main memory** of a computer (Figure 12-1), where fast operation is important. A computer's main memory—also called its *working memory*—is in constant communication with the central processing unit (CPU) as a program of instructions is being executed. A program and any data used by the program reside in the main memory while the computer is working on that program. RAM and ROM (to be defined shortly) make up main memory.

Another form of storage in a computer is performed by **auxiliary memory** (Figure 12-1), which is separate from the main working memory. Auxiliary memory—also called *mass storage*—has the capacity to store massive amounts of data without the need for electrical power. Auxiliary memory operates at a much slower speed than main memory, and it stores programs and data that are not currently being used by the CPU. This information is transferred to the main memory when the computer needs it. Common auxiliary memory devices are magnetic disk and compact disk (CD), which is accessed optically.

We will take a detailed look at the characteristics of the most common memory devices used as the internal memory of a computer. First, we define some of the common terms used in memory systems.

## 12-1 MEMORY TERMINOLOGY

### OUTCOME

Upon completion of this section, you will be able to:

- Define terms common to memory systems.

The study of memory devices and systems is filled with terminology that can sometimes be overwhelming to a student. Before we get into any comprehensive discussion of memories, it would be helpful if you had the meaning of some of the more basic terms under your belt. Other new terms will be defined as they appear in the chapter.

- **Memory Cell.** A device or an electrical circuit used to store a single bit (0 or 1). Examples of memory cells include a flip-flop, a charged capacitor, and a single spot on magnetic tape or disk.

- **Memory Word.** A group of bits (cells) in a memory that represents instructions or data of some type. For example, a register consisting of eight FFs can be considered to be a memory that is storing an eight-bit word. Word sizes in computers typically range from 8 to 64 bits, depending on the size of the computer.
- **Byte.** A special term used for a group of eight bits. A byte always consists of eight bits. Word sizes can be expressed in bytes as well as in bits. For example, a word size of eight bits is also a word size of one byte, a word size of 16 bits is two bytes, and so on.
- **Capacity.** A way of specifying how many bits can be stored in a particular memory device or complete memory system. To illustrate, suppose that we have a memory that can store 4096 20-bit words. This represents a total capacity of 81,920 bits. We could also express this memory's capacity as  $4096 \times 20$ . When expressed this way, the first number (4096) is the number of words, and the second number (20) is the number of bits per word (word size). The number of words in a memory is often a multiple of 1024. It is common to use the designation "1K" to represent  $1024 = 2^{10}$  when referring to memory capacity. Thus, a memory that has a storage capacity of  $4K \times 20$  is actually a  $4096 \times 20$  memory. The development of larger memories has brought about the designation "1M" or "1 meg" to represent  $2^{20} = 1,048,576$ . Thus, a memory that has a capacity of  $2M \times 8$  is actually one with a capacity of  $2,097,152 \times 8$ . The designation "giga" refers to  $2^{30} = 1,073,741,824$ .

**EXAMPLE 12-1A**

A certain semiconductor memory chip is specified as  $2K \times 8$ . How many words can be stored on this chip? What is the word size? How many total bits can this chip store?

**Solution**

$$2K = 2 \times 1024 = 2048 \text{ words}$$

Each word is eight bits (one byte). The total number of bits is therefore

$$2048 \times 8 = 16,384 \text{ bits}$$

**EXAMPLE 12-1B**

Which memory stores the most bits: a  $5M \times 8$  memory or a memory that stores 1M words at a word size of 16 bits?

**Solution**

$$5M \times 8 = 5 \times 1,048,576 \times 8 = 41,943,040 \text{ bits}$$

$$1M \times 16 = 1,048,576 \times 16 = 16,777,216 \text{ bits}$$

The  $5M \times 8$  memory stores more bits.

- **Density.** Another term for *capacity*. When we say that one memory device has a greater density than another, we mean that it can store more bits in the same amount of space. It is more dense.
- **Address.** A number that identifies the location of a word in memory. Each word stored in a memory device or system has a unique address. Addresses always exist in a digital system as a binary number, although octal, hexadecimal, and decimal numbers are often used to represent the address

**FIGURE 12-2** Each word location has a specific binary address.

Addresses	
000	Word 0
001	Word 1
010	Word 2
011	Word 3
100	Word 4
101	Word 5
110	Word 6
111	Word 7

for convenience. Figure 12-2 illustrates a small memory consisting of eight words. Each of these eight words has a specific address represented as a three-bit number ranging from 000 to 111. Whenever we refer to a specific word location in memory, we use its address code to identify it.

- **Read Operation.** The operation whereby the binary word stored in a specific memory location (address) is sensed and then transferred to another device. For example, if we want to use word 4 of the memory of Figure 12-2 for some purpose, we must perform a read operation on address 100. The read operation is often called a *fetch* operation because a word is being fetched from memory. We will use both terms interchangeably.
- **Write Operation.** The operation whereby a new word is placed into a particular memory location. It is also referred to as a *store* operation. Whenever a new word is written into a memory location, it replaces the word that was previously stored there.
- **Access Time.** A measure of a memory device's operating speed. It is the amount of time required to perform a read operation. More specifically, it is the time between the memory receiving a new address input and the data becoming available at the memory output. The symbol  $t_{ACC}$  is used for access time.
- **Volatile Memory.** Any type of memory that requires the application of electrical power in order to store information. If the electrical power is removed, all information stored in the memory will be lost. Many semiconductor memories are volatile, while all magnetic memories are *nonvolatile*, which means that they can store information without electrical power.
- **Random-Access Memory (RAM).** Memory in which the actual physical location of a memory word has no effect on how long it takes to read from or write into that location. In other words, the access time is the same for any address in memory. Most semiconductor memories are RAMs.
- **Sequential-Access Memory (SAM).** A type of memory in which the access time is not constant but varies depending on the address location. A particular stored word is found by sequencing through all address locations until the desired address is reached. This produces access times that are much longer than those of random-access memories. Sequential-access memories are used where the data to be accessed will always come in a long sequence of successive words. Video memory, for example, must output its contents in the same order over and over again to keep the image refreshed on the computer screen. Two more examples of sequential access memories are magnetic tape backup systems and DVDs. To illustrate the difference between SAM and RAM, consider the information stored on a DVD (Digital Video Disc). A DVD movie is broken up into chapters that can be selected from a menu. The viewer has random access

to the beginning of each chapter. However, to view a particular scene in a chapter requires the use of fast forward or reverse to scan through all the preceding scenes sequentially, like sequential access memory.

- **Read/Write Memory (RWM).** Any memory that can be read from or written into with equal ease.
- **Read-Only Memory (ROM).** A broad class of semiconductor memories designed for applications where the ratio of read operations to write operations is very high. Technically, a ROM can be written into (programmed) only once, and this operation is normally performed at the factory. Thereafter, information can only be read from the memory. Other types of ROM are actually read-mostly memories (RMM), which can be written into more than once; but the write operation is more complicated than the read operation, and it is not performed very often. The various types of ROM will be discussed later. *All ROM is nonvolatile* and will store data when electrical power is removed.
- **Static Memory Devices.** Semiconductor memory devices in which the stored data will remain permanently stored as long as power is applied, without the need for periodically rewriting the data into memory.
- **Dynamic Memory Devices.** Semiconductor memory devices in which the stored data will *not* remain permanently stored, even with power applied, unless the data are periodically rewritten into memory. The latter operation is called a *refresh* operation.
- **Main Memory.** Also referred to as the computer's *working memory*. It stores instructions and data the CPU is currently working on.
- **Cache Memory.** A high-speed block of memory that operates between the slower main memory and the CPU in order to optimize the speed of the computer. The cache memory may be physically located in the CPU or on the mother board or both.
- **Auxiliary Memory.** Also referred to as *mass storage* because it stores massive amounts of information external to the main memory. It is slower in speed than main memory and is always nonvolatile.

### OUTCOME ASSESSMENT QUESTIONS

1. Define the following terms.
  - (a) *Memory cell*
  - (b) *Memory word*
  - (c) *Address*
  - (d) *Byte*
  - (e) *Access time*
2. A certain memory has a capacity of  $8K \times 16$ . How many bits are in each word? How many words are being stored? How many memory cells does this memory contain?
3. Explain the difference between the read (fetch) and write (store) operations.
4. *True or false:* A volatile memory will lose its stored data when electrical power is interrupted.
5. Explain the difference between SAM and RAM.
6. Explain the difference between RWM and ROM.
7. *True or false:* A dynamic memory will hold its data as long as electrical power is applied.

## 12-2 GENERAL MEMORY OPERATION

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the process used to read and write solid-state memory devices.
- Define the role of the control inputs for solid-state memory devices.
- Relate the memory configuration to the number of pins on the IC.

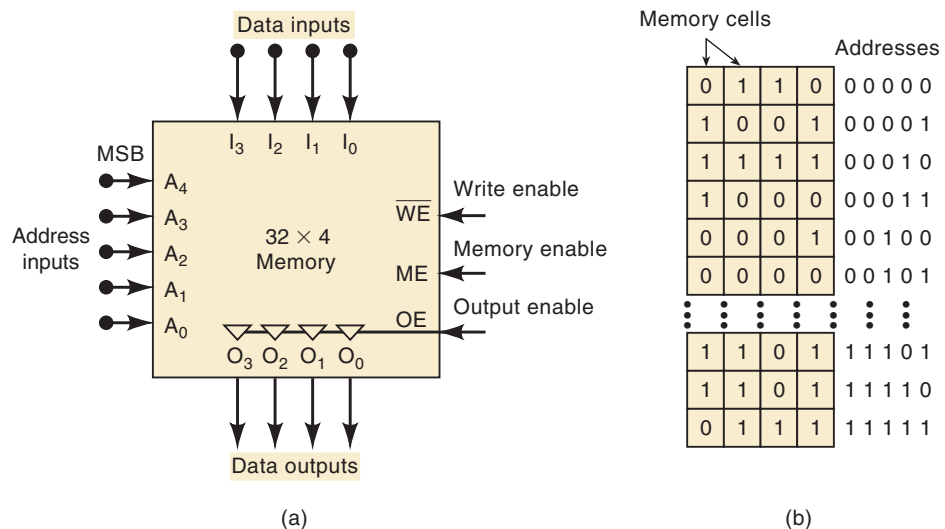
Although each type of memory is different in its internal operation, certain basic operating principles are the same for all memory systems. An understanding of these basic ideas will help in our study of individual memory devices.

Every memory system requires several different types of input and output lines to perform the following functions:

1. Apply the binary address of the memory location that is to be accessed.
2. Enable the memory device to respond to its control inputs.
3. Place the data stored in the specified address on the internal data lines.
4. In the case of a read operation, enable the tristate outputs, which applies the data to the output pins.
5. In the case of a write operation, apply the data to be stored to the data input pins.
6. Enable the write operation, which causes the data to be stored at the specified location.
7. Deactivate the read or write controls when done reading or writing and disable the memory IC.

Figure 12-3(a) illustrates these basic functions in a simplified diagram of a  $32 \times 4$  memory that stores 32 four-bit words. Because the word size is four bits, there are four data input lines  $I_0$  to  $I_3$  and four data output lines  $O_0$  to  $O_3$ . During a write operation, the data to be stored into memory must be applied to the data input lines. During a read operation, the word being read from memory appears at the data output lines.

**FIGURE 12-3** (a) Diagram of a  $32 \times 4$  memory; (b) virtual arrangement of memory cells into 32 four-bit words.



## Address Inputs

Because this memory (Figure 12-3) stores 32 words, it has 32 different storage locations and therefore 32 different binary addresses ranging from 00000 to 11111 (0 to 31 in decimal). Thus, there are five address inputs,  $A_0$  to  $A_4$ . To access one of the memory locations for a read or a write operation, the five-bit address code for that particular location is applied to the address inputs. In general,  $N$  address inputs are required for a memory that has a capacity of  $2^N$  words.

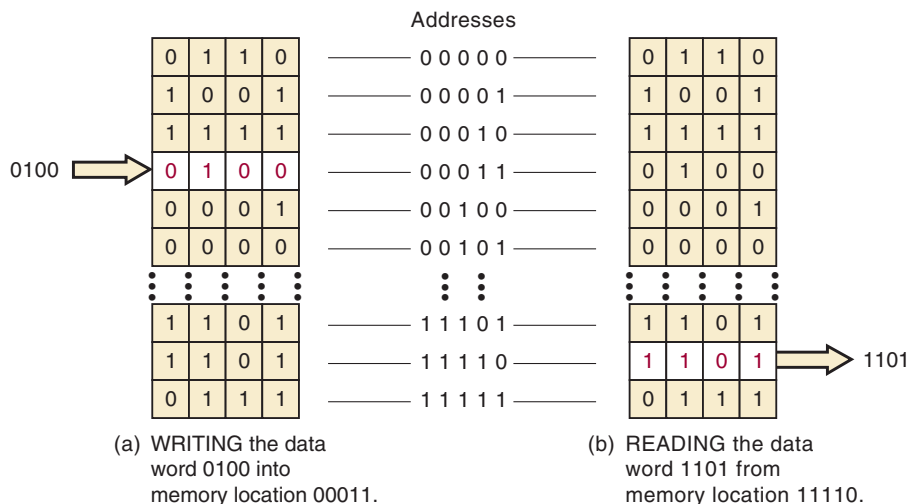
We can visualize the memory of Figure 12-3(a) as an arrangement of 32 registers, with each register holding a four-bit word, as illustrated in Figure 12-3(b). Each address location is shown containing four memory cells that hold 1s and 0s that make up the data word stored at that location. For example, the data word 0110 is stored at address 00000, the data word 1001 is stored at address 00001, and so on.

## The $\overline{WE}$ Input

The  $\overline{WE}$  (write enable) input is activated to allow the memory to store data. The bar over the  $WE$  indicates that the write operation takes place when  $\overline{WE} = 0$ . Other labels are sometimes used for this input. Two of the more common ones are  $\overline{W}$  (write) and  $R/\overline{W}$ . Again, the bar indicates that the write operation occurs when the input is LOW. Regardless of how it is labeled, this input must be HIGH when a read operation occurs. A control signal is normally connected to  $\overline{WE}$  that is active only when the system has placed stable data, which is intended to be stored in the memory, onto the data bus.

A simplified illustration of the read and write operations is shown in Figure 12-4. Figure 12-4(a) shows the data word 0100 being written into the memory register at address location 00011. This data word would have been applied to the memory's data input lines, and it replaces the data previously stored at address 00011. Figure 12-4(b) shows the data word 1101 being read from address 11110. This data word would appear at the memory's data output lines. After the read operation, the data word 1101 is still stored in address 11110. In other words, the read operation does not change the stored data.

**FIGURE 12-4** Simplified illustration of the read and write operations on the  $32 \times 4$  memory: (a) writing the data word 0100 into memory location 00011; (b) reading the data word 1101 from memory location 11110.



## Output Enable (*OE*)

Since most memory devices are designed to operate on a tristate bus, it is necessary to disable the output drivers at all times when data is not being read from the memory. The *OE* pin is activated to enable the tristate buffers and deactivated to place the buffers in the high impedance (Hi-Z) state. A control signal is connected to *OE* that is active only when the bus is ready to receive data from the memory.

## Memory Enable

Many memory systems have some means for completely disabling all or part of the memory so that it will not respond to the other inputs. This is represented in Figure 12-3 as the *memory enable* input, although it can have different names in the various memory systems, such as chip enable (*CE*) or chip select (*CS*). Here, it is shown as an active-HIGH input that enables the memory to operate normally when it is kept HIGH. A LOW on this input disables the memory so that it will not respond to the address, data,  $\overline{WE}$ , and *OE* inputs. This type of input is useful when several memory modules are combined to form a larger memory. We will examine this idea later.

### EXAMPLE 12-2

Describe the conditions at each input and output when the contents of address location 00100 are to be read. Refer to Figure 12-4.

#### Solution

Address inputs: 00100  
 Data inputs: xxxx (not used)  
 $\overline{WE}$ : HIGH  
*memory enable*: HIGH  
 Data outputs: 0001

### EXAMPLE 12-3

Describe the conditions at each input and output when the data word 1110 is to be written into address location 01101. (See Figure 12-4.)

#### Solution

Address inputs: 01101  
 Data inputs: 1110  
 $\overline{WE}$ : LOW  
*memory enable*: HIGH  
 Data outputs: xxxx (not used; usually Hi-Z)

### EXAMPLE 12-4

A certain memory has a capacity of  $4K \times 8$ .

- How many data input and data output lines does it have?
- How many address lines does it have?
- What is its capacity in bytes?

**Solution**

- (a) Eight of each because the word size is eight.
- (b) The memory stores  $4K = 4 \times 1024 = 4096$  words. Thus, there are 4096 memory addresses. Because  $4096 = 2^{12}$ , it requires a 12-bit address code to specify one of 4096 addresses.
- (c) A byte is eight bits. This memory has a capacity of 4096 bytes.

The example memory in Figure 12-3 illustrates the important input and output functions common to most memory systems. Of course, each type of memory may have other input and output lines that are peculiar to that memory. These will be described as we discuss the individual memory types.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. How many address inputs, data inputs, and data outputs are required for a  $16K \times 12$  memory?
2. What is the function of the  $\overline{WE}$  input?
3. What is the function of the *memory enable* input?

## 12-3 CPU–MEMORY CONNECTIONS

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Distinguish between the source and destination of data in a computer system during read and write cycles.
- List the sequence of events necessary for a read and a write operation.
- Identify the major buses of a computer system and state their purpose.

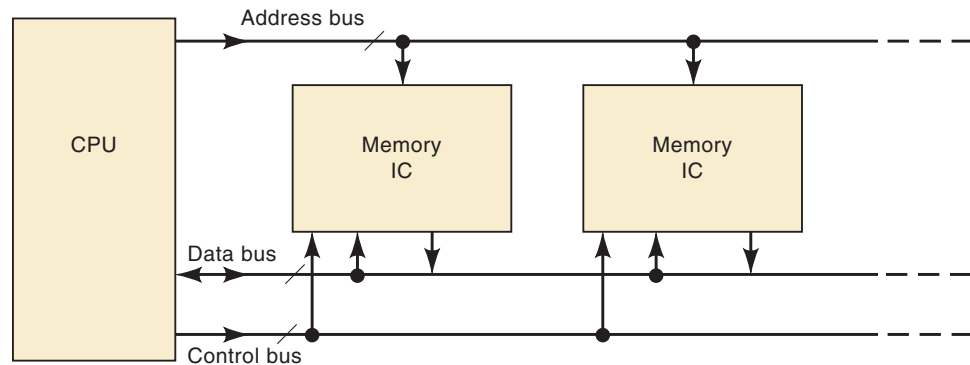
A major part of this chapter is devoted to semiconductor memory, which, as pointed out earlier, makes up the main memory of most modern computers. Remember, this main memory is in constant communication with the central processing unit (CPU). It is not necessary to be familiar with the detailed operation of a CPU at this point, and so the following simplified treatment of the CPU–memory interface will provide the perspective needed to make our study of memory devices more meaningful.

A computer's main memory is made up of RAM and ROM ICs that are interfaced to the CPU over three groups of signal lines or buses. These are shown in Figure 12-5 as the address lines or address bus, the data lines or data bus, and the control lines or control bus. Each of these buses consists of several lines (note that they are represented by a single line with a slash), and the number of lines in each bus will vary from one computer to the next. The three buses play a necessary part in allowing the CPU to write data into memory and to read data from memory.

When a computer is executing a program of instructions, the CPU continually fetches (reads) information from those locations in memory that contain (1) the program codes representing the operations to be performed and (2) the data to be operated upon. The CPU will also store (write) data into memory locations as dictated by the program instructions. Whenever



**FIGURE 12-5** Three groups of lines (buses) connect the main memory ICs to the CPU.



the CPU wants to write data to a particular memory location, the following steps must occur:

### Write Operation

1. The CPU supplies the binary address of the memory location where the data are to be stored. It places this address on the address bus lines.
2. An address decoder activates the memory device's enable input ( $CE$  or  $CS$ ).
3. The CPU places the data to be stored on the data bus lines.
4. The CPU activates the appropriate control signal lines for the memory write operation (e.g.,  $\overline{WR}$  or  $R/\overline{W}$ ) that is normally connected to  $\overline{WE}$  on the memory IC.
5. The memory ICs internally decode the binary address to determine which location is being selected for the store operation.
6. The data on the data bus are transferred to the selected memory location.

Whenever the CPU wants to read data from a specific memory location, the following steps must occur:

### Read Operation

1. The CPU supplies the binary address of the memory location from which data are to be retrieved. It places this address on the address bus lines.
2. An address decoder activates the memory device's enable input ( $CE$  or  $CS$ ).
3. The CPU activates the appropriate control signal lines for the memory read operation (e.g.,  $\overline{RD}$ ) that is normally connected to  $\overline{OE}$  on the memory IC.
4. The memory ICs internally decode the binary address to determine which location is being selected for the read operation.
5. The memory ICs place data from the selected memory location onto the data bus, from which they are transferred to the CPU.

The steps above should make clear the function of each of the system buses:

- **Address Bus.** This *unidirectional* bus carries the binary address outputs from the CPU to the memory ICs to select one memory location.
- **Data Bus.** This *bidirectional* bus carries data between the CPU and the memory ICs.
- **Control Bus.** This bus carries control signals (e.g.,  $\overline{RD}$ ,  $\overline{WR}$ ) from the CPU to the memory ICs.

As we get into discussions of actual memory ICs, we will examine the signal activity that appears on these buses for the read and write operations.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Name the three groups of lines that connect the CPU and the internal memory.
2. Outline the steps that take place when the CPU reads from memory.
3. Outline the steps that occur when the CPU writes to memory.

## 12-4 READ-ONLY MEMORIES

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the inputs and outputs of a ROM.
- Distinguish between volatile and nonvolatile memory.
- Define the programming process for a ROM.

The read-only memory is a type of semiconductor memory designed to hold data that either are permanent or will not change frequently. During normal operation, no new data can be written into a ROM, but data can be read from ROM. For some ROMs, the data that are stored must be built-in during the manufacturing process; for other ROMs, the data can be entered electrically. The process of entering data is called **programming** or *burning* the ROM. Some ROMs cannot have their data changed once they have been programmed; others can be *erased* and reprogrammed as often as desired. We will take a detailed look later at these various types of ROMs. For now, we will assume that the ROMs have been programmed and are holding data.

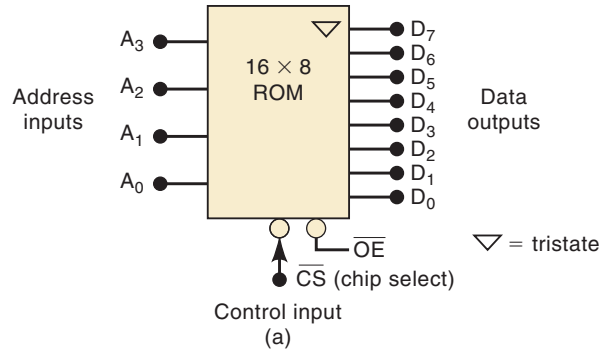
ROMs are used to store data and information that are not to change during the normal operation of a system. A major use for ROMs is in the storage of programs in microcomputers. Because all ROMs are *nonvolatile*, these programs are not lost when electrical power is turned off. When the microcomputer is turned on, it can immediately begin executing the program stored in ROM. ROMs are also used for program and data storage in microprocessor-controlled equipment such as electronic cash registers, appliances, and security systems.

### ROM Block Diagram

A typical block diagram for a ROM is shown in Figure 12-6(a). It has three sets of signals: address inputs, control input(s), and data outputs. From our previous discussions, we can determine that this ROM is storing 16 words because it has  $2^4 = 16$  possible addresses, and each word contains eight bits because there are eight data outputs. Thus, this is a  $16 \times 8$  ROM. Another way to describe this ROM's capacity is to say that it stores 16 bytes of data.

The data outputs of most ROM ICs are tristate outputs, to permit the connection of many ROM chips to the same data bus for memory expansion. The most common numbers of data outputs for ROMs are four, eight, and 16 bits, with eight-bit words being the most common.

The control input  $\overline{CS}$  stands for **chip select**. This is essentially an enable input that enables or disables the ROM outputs. Some manufacturers use different labels for the control input, such as *CE* (chip enable) or *OE* (output enable). Many ROMs have two or more control inputs that must be active in order to enable the data outputs so that data can be read from the selected address. In some ROM ICs, one of the control inputs (usually the *CE*) is used to place the ROM in a low-power standby mode when it is not being used. This reduces the current drain from the system power supply.



Word	Address				Data							
	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
0	0	0	0	0	1	1	0	1	1	1	1	0
1	0	0	0	1	0	0	1	1	1	0	1	0
2	0	0	1	0	1	0	0	0	0	1	0	1
3	0	0	1	1	1	0	1	0	1	1	1	1
4	0	1	0	0	0	0	0	1	1	0	0	1
5	0	1	0	1	0	1	1	1	1	0	1	1
6	0	1	1	0	0	0	0	0	0	0	0	0
7	0	1	1	1	1	1	1	0	1	1	0	1
8	1	0	0	0	0	0	1	1	1	1	0	0
9	1	0	0	1	1	1	1	1	1	1	1	1
10	1	0	1	0	1	0	1	1	1	0	0	0
11	1	0	1	1	1	1	0	0	0	1	1	1
12	1	1	0	0	0	0	1	0	0	1	1	1
13	1	1	0	1	0	1	1	0	1	0	1	0
14	1	1	1	0	1	1	0	1	0	0	1	0
15	1	1	1	1	0	1	0	1	1	0	1	1

(b)

Word	Address				Data
	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	D <sub>7</sub> -D <sub>0</sub>
0					DE
1					3A
2					85
3					AF
4					19
5					7B
6					00
7					ED
8					3C
9					FF
10					B8
11					C7
12					27
13					6A
14					D2
15					5B

(c)

**FIGURE 12-6** (a) Typical ROM block symbol; (b) table showing binary data at each address location; (c) the same table in hex.

The  $\overline{CS}$  input shown in Figure 12-6(a) is active-LOW; therefore, it must be in the LOW state to enable the ROM data to appear at the data outputs. Notice that there is no  $\overline{WE}$  (write enable) input because the ROM cannot be written into during normal operation.

### The Read Operation

Let's assume that the ROM has been programmed with the data shown in the table of Figure 12-6(b). Sixteen different data words are stored at the 16 different address locations. For example, the data word stored at location 0011 is 10101111. Of course, the data are stored in binary inside the ROM, but very often we use hexadecimal notation to show the programmed data efficiently. This is done in Figure 12-6(c).

In order to read a data word from ROM, we need to do two things: (1) apply the appropriate address inputs and then (2) activate the control inputs. For example, if we want to read the data stored at location 0111 of the ROM in Figure 12-6, we must apply  $A_3A_2A_1A_0 = 0111$  to the address inputs and then apply a LOW to  $\overline{CS}$ . The address inputs will be decoded inside the ROM to select the correct data word, 11101101, that will appear at outputs  $D_7$  to  $D_0$  after  $\overline{OE}$  is asserted (LOW). If  $\overline{CS}$  is kept HIGH, the ROM outputs will be disabled and will be in the Hi-Z state.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. True or false: All ROMs are nonvolatile.
2. Describe the procedure for reading from ROM.
3. What is *programming* or *burning-in* a ROM?

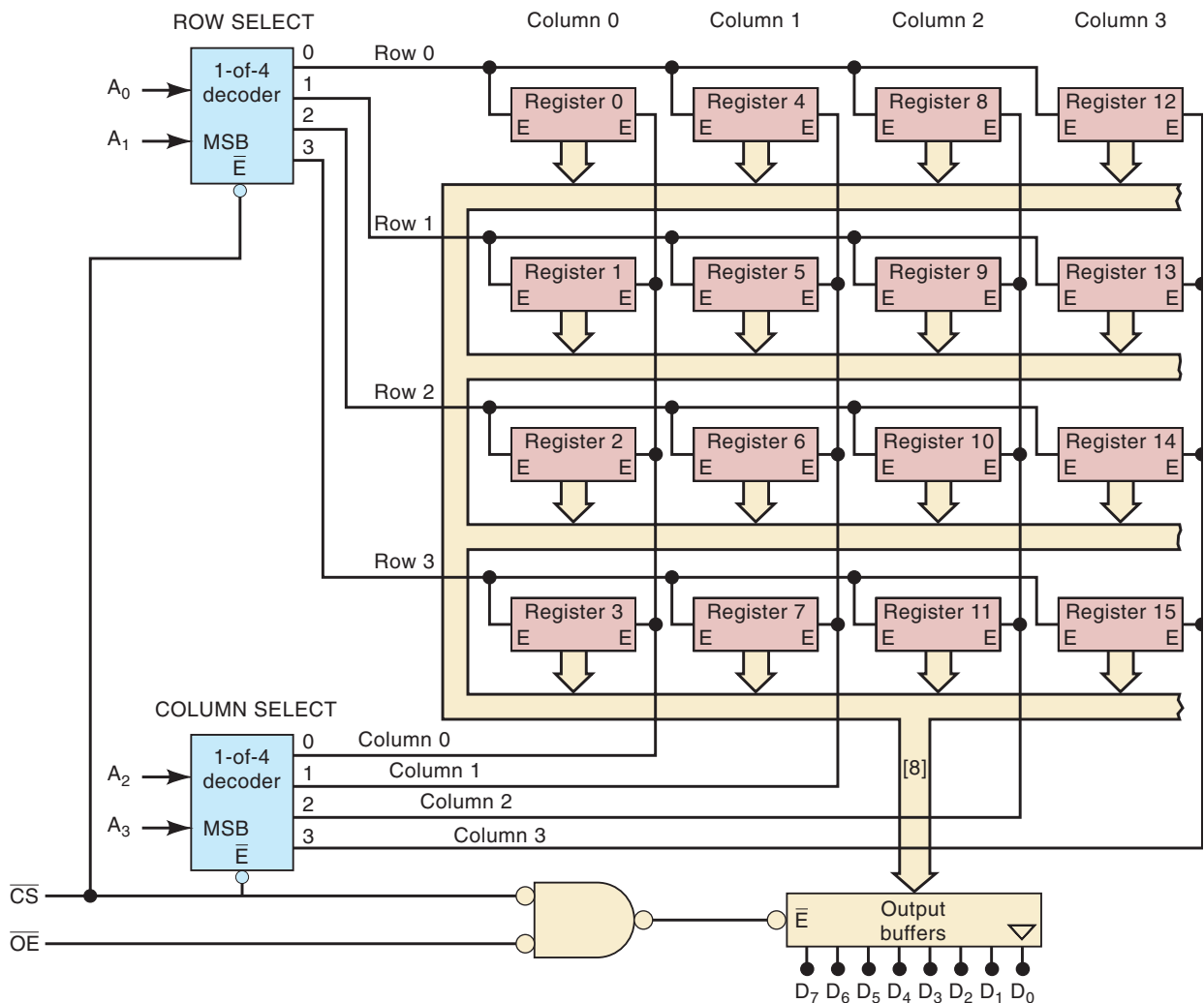
## 12-5 ROM ARCHITECTURE

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the sub-blocks that make up a ROM and the role of each.
- Relate addresses to the physical location of data in the memory.

The internal architecture (structure) of a ROM IC is very complex, and we need not be familiar with all of its detail. It is instructive, however, to look at a simplified diagram of the internal architecture, such as that shown in Figure 12-7, for the  $16 \times 8$  ROM. There are four basic parts: *register array*, *row decoder*, *column decoder*, and *output buffers*.



**FIGURE 12-7** Architecture of a  $16 \times 8$  ROM. Each register stores one eight-bit word.

## Register Array

The register array stores the data that have been programmed into the ROM. Each register contains several memory cells equal to the word size. In this case, each register stores an eight-bit word. The registers are arranged in a square matrix array that is common to many semiconductor memory chips. We can specify the position of each register as being in a specific row and a specific column. For example, register 0 is in row 0, column 0, and register 9 is in row 1, column 2.

The eight data outputs of each register are connected to an internal data bus that runs through the entire circuit. Each register has two enable inputs ( $E$ ); both must be HIGH in order for the register's data to be placed on the bus.

## Address Decoders

The applied address code  $A_3A_2A_1A_0$  determines which register in the array will be enabled to place its eight-bit data word onto the bus. Address bits  $A_1A_0$  are fed to a 1-of-4 decoder that activates one row-select line, and address bits  $A_3A_2$  are fed to a second 1-of-4 decoder that activates one column-select line. Only one register will be in both the row and the column selected by the address inputs, and this one will be enabled.

### EXAMPLE 12-5

Which register will be enabled by input address 1101?

#### Solution

$A_3A_2 = 11$  will cause the column decoder to activate the column 3 select line, and  $A_1A_0 = 01$  will cause the row decoder to activate the row 1 select line. This will place HIGHS at both enable inputs of register 13, thereby causing its data outputs to be placed on the bus. Note that the other registers in column 3 will have only one enable input activated; the same is true for the other row 1 registers.

### EXAMPLE 12-6

What input address will enable register 7?

#### Solution

The enable inputs of this register are connected to the row 3 and column 1 select lines, respectively. To select row 3, the  $A_1A_0$  inputs must be at 11, and to select column 1, the  $A_3A_2$  inputs must be at 01. Thus, the required address will be  $A_3A_2A_1A_0 = 0111$ .

## Output Buffers

The register that is enabled by the address inputs will place its data on the data bus. These data feed into the output buffers, which will pass the data to the external data outputs, provided that  $\overline{CS}$  and  $\overline{OE}$  are LOW. If  $\overline{CS}$  or  $\overline{OE}$  is HIGH, the output buffers are in the Hi-Z state, and  $D_7$  through  $D_0$  will be floating.

The architecture shown in Figure 12-7 is similar to that of many IC ROMs. Depending on the number of stored data words, the registers in some ROMs will not be arranged in a square array. For example, the Intel 27C64 is a CMOS ROM that stores 8192 eight-bit words. Its 8192 registers are arranged in an array of 256 rows  $\times$  32 columns.

**EXAMPLE 12-7**

Describe the internal architecture of a ROM that stores 4K bytes and uses a square register array.

**Solution**

4K is actually  $4 \times 1024 = 4096$ , and so this ROM holds 4096 eight-bit words. Each word can be thought of as being stored in an eight-bit register, and there are 4096 registers connected to a common data bus internal to the chip. Because  $4096 = 64^2$ , the registers are arranged in a  $64 \times 64$  array; that is, there are 64 rows and 64 columns. This requires a 1-of-64 decoder to decode six address inputs for the row select, and a second 1-of-64 decoder to decode six other address inputs for the column select. Thus, a total of 12 address inputs is required. This makes sense because  $2^{12} = 4096$ , and there are 4096 different addresses.

**OUTCOME ASSESSMENT QUESTIONS**

1. What input address code is required if we want to read the data from register 9 in Figure 12-7?
2. Describe the function of the row-select decoder, the column-select decoder, and the output buffers in the ROM architecture.

**12-6 ROM TIMING****OUTCOME**

*Upon completion of this section, you will be able to:*

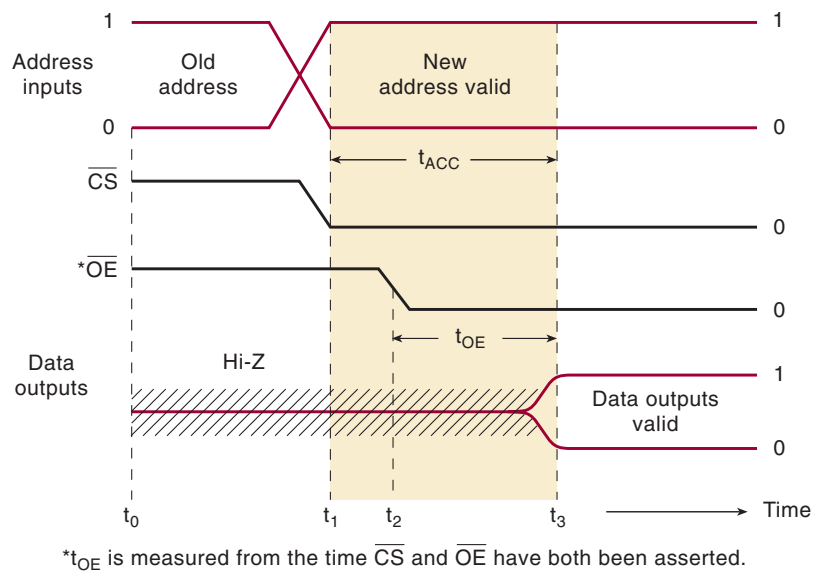
- Interpret timing diagrams that define the operating limitations of ROMs.

There will be a propagation delay between the application of a ROM's inputs and the appearance of the data outputs during a read operation. This time delay, called access time ( $t_{ACC}$ ) is a measure of the ROM's operating speed. Access time is described graphically by the waveforms in Figure 12-8.

The top waveform represents the address inputs; the middle waveforms represent the activation of both an active-LOW output enable,  $\overline{OE}$ , and an active-LOW chip select,  $\overline{CS}$ ; and the bottom waveform represents the data outputs. At time  $t_0$  the address inputs are all at some specific level, some HIGH and some LOW. Either  $\overline{OE}$  or  $\overline{CS}$  is HIGH, so that the ROM data outputs are in their Hi-Z state (represented by the hatched line).

Just prior to  $t_1$ , the address inputs are changing to a new address for a new read operation. At  $t_1$ , the new address is valid; that is, each address input is at a valid logic level, and  $\overline{CS}$  will be activated. At this point, the internal ROM circuitry begins to decode the new address inputs to select the register that is to send its data to the output buffers. At  $t_2$ , the  $\overline{CS}$  and

**FIGURE 12-8** Typical timing for a ROM read operation.



$\overline{OE}$  inputs have both been activated to enable the output buffers. Finally, at  $t_3$ , the outputs change from the Hi-Z state to the valid data, which is the binary number stored at the specified address.

The time delay between  $t_1$ , when the new address becomes valid, and  $t_3$ , when the data outputs become valid, is the access time  $t_{ACC}$ . Typical bipolar ROMs will have access times in the range from 30 to 90 ns; access times of NMOS devices will range from 35 to 500 ns. Improvements to CMOS technology have brought access times into the 20-to-60-ns range. Consequently, bipolar and NMOS devices are rarely produced in newer (larger) ROMs.

Another important timing parameter is the *output enable time* ( $t_{OE}$ ), which is the delay between the  $\overline{OE}$  input and the valid data output. Values for  $t_{OE}$  are always shorter than the access time. This timing parameter ( $t_{OE}$ ) is important in situations where the address inputs are already set to their new values, but the ROM outputs have not yet been enabled. When  $\overline{OE}$  goes LOW to enable the outputs, the delay will be  $t_{OE}$ .

### OUTCOME ASSESSMENT QUESTIONS

1. What signal from a microcontroller would normally be connected to the OE input of a ROM?
2. Which timing specification defines the minimum time from when the CPU puts out a new address until the end of the RD pulse?

## 12-7 TYPES OF ROMs

### OUTCOMES

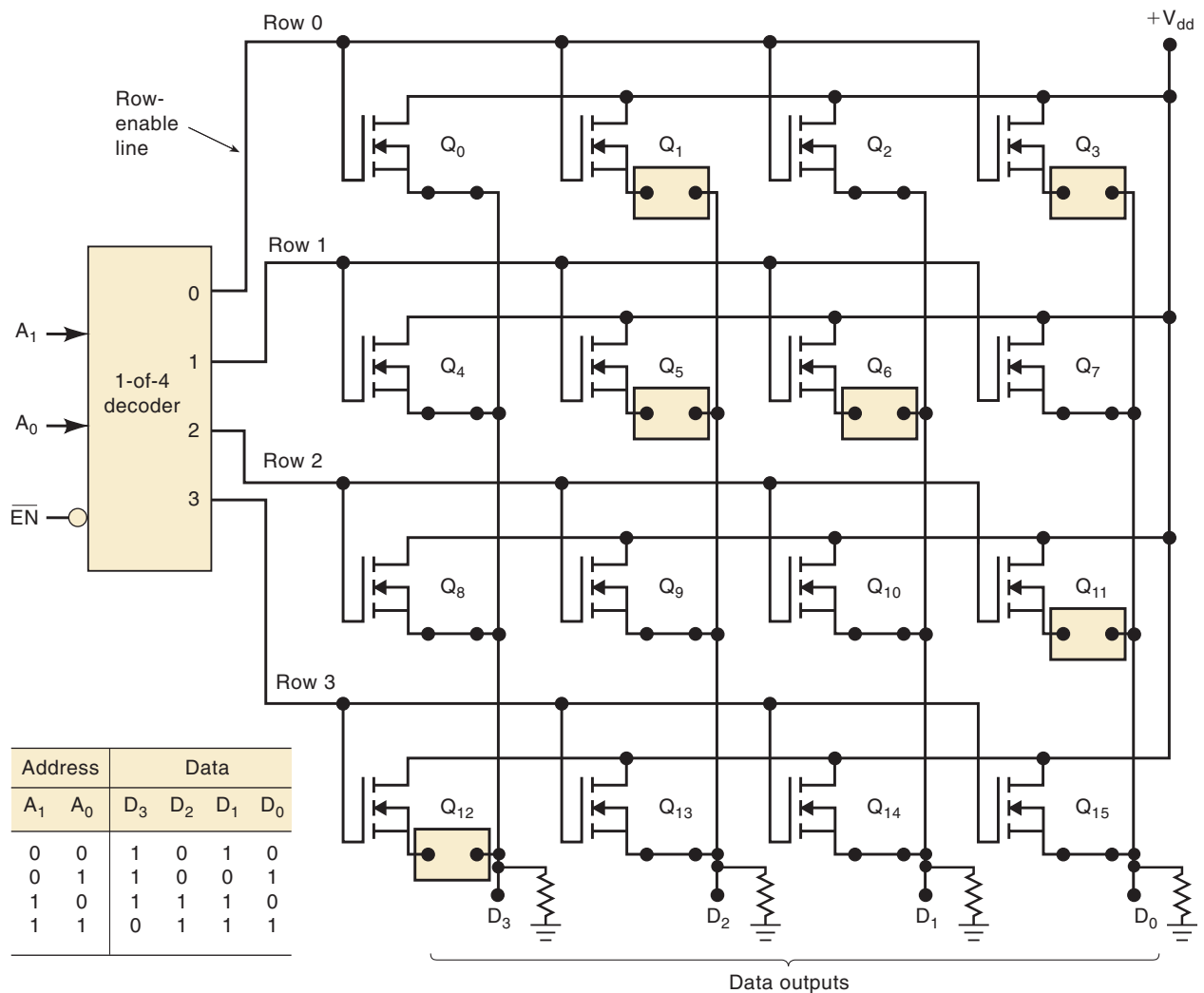
Upon completion of this section, you will be able to:

- Describe the technologies that have contributed to the evolution of ROMs.
- State the means by which a 1 or 0 is stored in the cell of a given type of ROM.
- State the functional features that characterize each type of ROM.
- Identify strengths and weaknesses of each technology.

Now that we have a general understanding of the internal architecture and external operation of ROM devices, we will look at the various types of ROMs to see how they differ in the way they are programmed, erased, and reprogrammed.

### Mask-Programmed ROM

The mask-programmed ROM (MROM) has its information stored at the time the integrated circuit is manufactured. As you can see from Figure 12-9, ROMs are made up of a rectangular array of transistors. Information is stored by either connecting or disconnecting the source of a transistor to the output column. The last step in the manufacturing process is to form all these conducting paths or connections. The process uses a “mask” to deposit metals on the silicon that determine where the connections form in a way similar to using stencils and spray paint but on a much smaller scale. The mask is very precise and expensive and must be made specifically for the customer, with the correct binary information. Consequently, this type of ROM is economical only when many ROMs are being made with exactly the same information.



**FIGURE 12-9** Structure of a MOS MROM shows one MOSFET used for each memory cell. An open source connection stores a “0”; a closed source connection stores a “1.”



Mask-programmed ROMs are commonly referred to as just ROMs, but this can be confusing because the term ROM actually represents the broad category of devices that, during normal operation, are only read from. We will use the abbreviation MROM whenever we refer to mask-programmed ROMs.

Figure 12-9 shows the structure of a small MOS MROM. It consists of 16 memory cells arranged in four rows of four cells. Each cell is an N-channel MOSFET transistor connected in the common-drain configuration (input at gate, output at source). The top row of cells (ROW 0) constitutes a four-bit register. Note that some of the transistors in this row ( $Q_0$  and  $Q_2$ ) have their source connected to the output column line, while others ( $Q_1$  and  $Q_3$ ) do not. The same is true of the cells in each of the other rows. The presence or absence of these source connections determines whether a cell is storing a 1 or a 0, respectively. The condition of each source connection is controlled during production by the photographic mask based on the customer-supplied data.

Notice that the data outputs are connected to column lines. Referring to output  $D_3$ , for instance, any transistor that has a connection from the source (such as  $Q_0$ ,  $Q_4$ , and  $Q_8$ ) to the output column can switch  $V_{dd}$  onto the column, making it a HIGH logic level. If  $V_{dd}$  is not connected to the column line, the output will be held at a LOW logic level by the pull-down resistor. At any given time, a maximum of one transistor in a column will ever be turned on due to the row decoder.

The 1-of-4 decoder is used to decode the address inputs  $A_1A_0$  to select which row (register) is to have its data read. The decoder's active-HIGH outputs provide the ROW enable lines that are the gate inputs for the various rows of cells. If the decoder's enable input,  $\overline{EN}$ , is held HIGH, all of the decoder outputs will be in their inactive LOW state, and all of the transistors in the array will be off because of the absence of any gate voltage. For this situation, the data outputs will all be in the LOW state.

When  $\overline{EN}$  is in its active-LOW state, the conditions at the address inputs determine which row (register) will be enabled so that its data can be read at the data outputs. For example, to read ROW 0, the  $A_1A_0$  inputs are set to 00. This places HIGH at the ROW 0 line; all other row lines are at 0 V. This HIGH at ROW 0 turns on transistors  $Q_0$ ,  $Q_1$ ,  $Q_2$ , and  $Q_3$ . With all of the transistors in the row conducting,  $V_{dd}$  will be switched on to each transistor's source lead. Outputs  $D_3$  and  $D_1$  will go HIGH because  $Q_0$  and  $Q_2$  are connected to their respective columns.  $D_2$  and  $D_0$  will remain LOW because there is no path from the  $Q_1$  and  $Q_3$  source leads to their columns. In a similar manner, application of the other address codes will produce data outputs from the corresponding register. The table in Figure 12-9 shows the data for each address. You should verify how this correlates with the source connections to the various cells.

#### EXAMPLE 12-8

MROMs can be used to store tables of mathematical functions. Show how the MROM in Figure 12-9 can be used to store the function  $y = x^2 + 3$ , where the input address supplies the value for  $x$ , and the value of the output data is  $y$ .

#### Solution

The first step is to set up a table showing the desired output for each set of inputs. The input binary number,  $x$ , is represented by the address  $A_1A_0$ . The output binary number is the desired value of  $y$ . For example, when

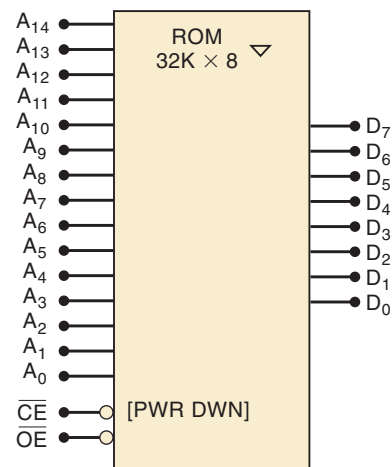
$x = A_1A_0 = 10_2 = 2_{10}$ , the output should be  $2^2 + 3 = 7_{10} = 0111_2$ . The complete table is shown in Table 12-1. This table is supplied to the MROM manufacturer for developing the mask that will make the appropriate connections within the memory cells during the fabrication process. For instance, the first row in the table indicates that the connections to the source of  $Q_0$  and  $Q_1$  will be left unconnected, while the connections to  $Q_2$  and  $Q_3$  will be made.

**TABLE 12-1** ROM data for Example 12-8.

$x$		$y = x^2 + 3$			
$A_1$	$A_0$	$D_3$	$D_2$	$D_1$	$D_0$
0	0	0	0	1	1
0	1	0	1	0	0
1	0	0	1	1	1
1	1	1	1	0	0

MROMs typically have tristate outputs that allow them to be used in a bus system, as we discussed in Chapter 9. Consequently, there must be a control input to enable and disable the tristate outputs. This control input is usually labeled *OE* (for output enable). In order to distinguish this tristate enable input from the address decoder enable input, the latter is usually referred to as a chip enable (*CE*). The chip enable performs more than just enabling the address decoder. When *CE* is disabled, all functions of the chip are disabled, including the tristate outputs, and the entire circuit is placed in a **power-down** mode that draws much less current from the power supply. Figure 12-10 shows a  $32\text{K} \times 8$  MROM. The 15 address lines ( $A_0$ – $A_{14}$ ) can identify  $2^{15}$  memory locations (32, 767, or 32K). Each memory location holds an eight-bit data value that can be placed on the data lines  $D_7$ – $D_0$  when the chip is enabled and the outputs are enabled.

**FIGURE 12-10** Logic symbol for a  $32\text{K} \times 8$  MROM.



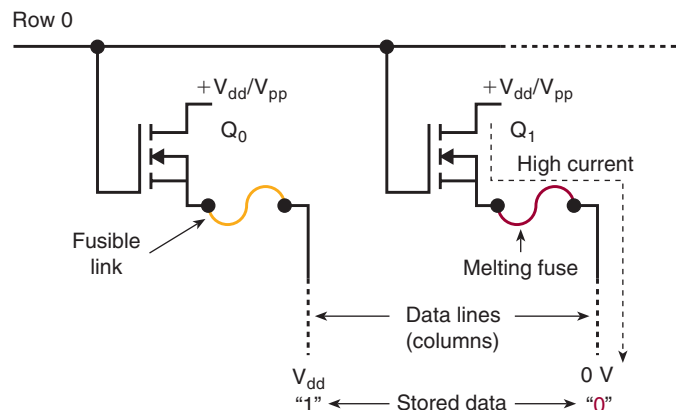
### Programmable ROMs (PROMs)

A mask-programmable ROM would not be used except in high-volume applications, where the cost would be spread out over many units. For lower-volume applications, manufacturers have developed **fusible-link** PROMs that are user-programmable; that is, they are not programmed during the

manufacturing process but are custom-programmed by the user. Once programmed, however, a PROM is like an MROM because it cannot be erased and reprogrammed. Thus, if the information in the PROM is faulty or must be changed, the PROM must be thrown away. For this reason, these devices are often referred to as “one-time programmable” (OTP) ROMs.

The fusible-link PROM structure is similar to the MROM structure because certain connections either are left intact or are opened in order to program a memory cell as a 1 or a 0, respectively. A PROM comes from the manufacturer with a thin, fuse link connection in the source leg of every transistor. In this condition, every transistor stores a 1. The user can then “blow” the fuse for any transistor that needs to store a 0. Typically, data can be programmed or “burned into” a PROM by selecting a row by applying the desired address to the address inputs, placing the desired data on the data pins, and then applying a pulse to a special programming pin on the IC. Figure 12-11 shows the inner workings of how this is done.

**FIGURE 12-11** PROMs use fusible links that can be selectively blown open by the user to program a logic 0 into a cell.



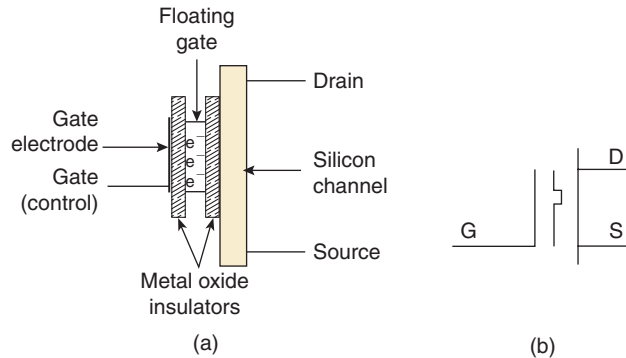
All of the transistors in the selected row (row 0) are turned on, and  $V_{pp}$  is applied to their drain leads. Those columns (data lines) that have a logic 0 on them (e.g.,  $Q_1$ ) will provide a high-current path through the fusible link, burning it open and permanently storing a logic 0. Those columns that have a logic 1 (e.g.,  $Q_0$ ) have  $V_{pp}$  on one side of the fuse and  $V_{dd}$  on the other side, drawing much less current and leaving the fuse intact. Once all address locations have been programmed in this manner, the data are permanently stored in the PROM and can be read over and over again by accessing the appropriate address. The data will not change when power is removed from the PROM chip because nothing will cause an open fuse link to become closed again.

### Erasable Programmable ROM (EPROM)

An EPROM can be programmed by the user, and it can also be *erased* and reprogrammed as often as desired. Once programmed, the EPROM is a *non-volatile* memory that will hold its stored data indefinitely. The process for programming an EPROM is the same as that for a PROM.

The storage element of an EPROM is a MOS transistor with a silicon gate that has no electrical connection (i.e., a floating gate) but is very close to an electrode as shown in Figure 12-12. In its normal state there is no charge stored on the floating gate and the transistor will produce a logic 1 whenever it is selected by the address decoder. To program a 0, a high-voltage pulse is used to leave a net charge on the floating gate. This charge causes the transistor to output a logic 0 when it is selected. Since the charge is trapped on the floating

**FIGURE 12-12** (a) A floating gate MOSFET; (b) its schematic symbol.



gate and has no discharge path, the 0 will be stored until it is erased. The data are erased by restoring all cells to a logic 1. To do this, the charge on the floating electrode is neutralized by exposing the silicon to high-intensity ultraviolet (UV) light for several minutes. UVEPROMS have a glass window that allows the UV light to shine on the silicon. The IC must be removed from the circuit and placed under a high-intensity UV lamp for approximately 20 minutes.

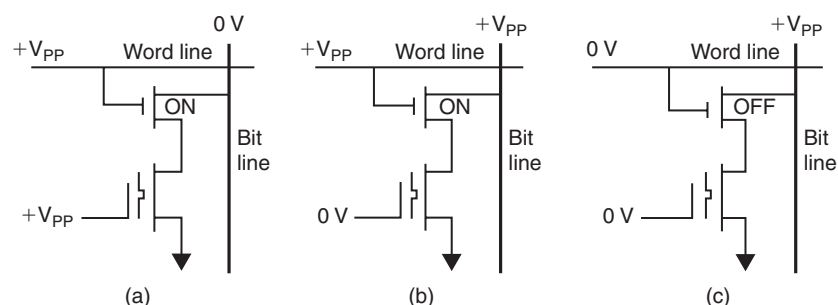
UVEPROMS have become obsolete, but the floating gate transistor technology was a tremendous innovation that has paved the way to the non-volatile memory technology found throughout digital systems today.

### Electrically Erasable PROM (EEPROM)

The disadvantages of the EPROM were overcome by the development of the **electrically erasable PROM (EEPROM)** as an improvement over the EPROM. The EEPROM retained the same floating-gate structure as the EPROM, but with the addition of a very thin oxide region above the drain of the MOSFET memory cell essentially implementing another MOS transistor switch. The circuit diagram is shown in Figure 12-13. This modification produced the EEPROM's major characteristic—its electrical erasability. By applying a high voltage between the MOSFET's gate and drain, a charge could be transferred onto the floating gate, where it remains even when power is removed; reversal of the same voltage causes a removal of the trapped charges from the floating gate. Because this charge-transport mechanism required very low currents, the erasing and programming of an EEPROM could be done *in circuit* (i.e., without a UV light source and a special PROM programmer unit).

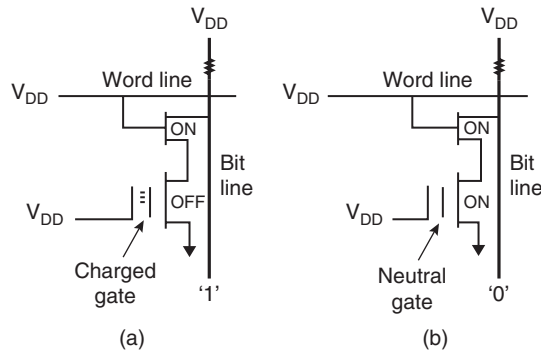
To erase the cell, charges are deposited on the floating gate. This is done by applying the higher than normal programming voltage ( $V_{pp}$ ) to the word line and to the control gate while the bit line is at ground as shown in Figure 12-13(a). The upper transistor is on and the high voltage from gate to channel deposits the charge on the floating gate, storing a logic “1.”

**FIGURE 12-13** (a) Erasing/precharging a cell; (b) writing a ‘0’; (c) writing a ‘1.’



**FIGURE 12-14**

(a) Reading a logic '1';  
 (b) reading a logic '0.'



To store a “0” on the EEPROM cell, the programming voltage ( $V_{pp}$ ) is applied to the bit line and the word line while the control gate is grounded [Figure 12-13(b)]. With the upper transistor on, this essentially applies the reverse voltage across the floating gate region removing the charge from the floating gate and neutralizing it.

Figure 12-13(c) shows how to store a “1” on the EEPROM cell during a write cycle. Note that writing to a cell can only be done after it has been erased. The programming voltage is applied only to the bit line. Zero volt is applied to the word line and also to the control voltage. The upper transistor is off and the reverse voltage is not applied, thereby leaving the charge (which was deposited during the erase operation) on the floating gate.

When reading the cell, a normal logic level ( $V_{DD}$ ) is applied to the word line and to the control gate. If the floating gate is charged, the control gate voltage ( $V_{DD}$ ) cannot turn on the floating gate MOSFET and so it outputs a logic “1” as shown in Figure 12-14(a). If the floating gate is neutralized as in Figure 12-14(b), the logic voltage on the control gate turns on the floating gate MOSFET, grounding the bit line and outputting a logic “0.”

Another advantage of the EEPROM over the EPROM is the ability to erase and rewrite *individual* bytes (eight-bit words) in the memory array electrically. During a write operation, internal circuitry automatically erases all of the cells at an address location prior to writing in the new data. This byte erasability makes it much easier to make changes in the data stored in an EEPROM.

### OUTCOME ASSESSMENT QUESTIONS

1. *True or false:* An MROM can be programmed by the user.
2. How does a PROM differ from an MROM? Can it be erased and reprogrammed?
3. *True or false:* A PROM stores a logic 1 when its fusible link is intact.
4. How is an EPROM erased?
5. *True or false:* There is no way to erase only a portion of an EPROM’s memory.
6. What function is performed by PROM and EPROM programmers?
7. What EPROM shortcomings are overcome by EEPROMs?
8. What are the major drawbacks of EEPROM?
9. What type of ROM can erase one byte at a time?
10. How is a logic 1 represented in a UVEPROM cell?
11. How is logic 1 represented in a EEPROM cell?

## 12-8 FLASH MEMORY

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the technology used to store 1s and 0s in flash memory.
- Identify the role of each input and output of a flash memory IC.
- Distinguish between NAND and NOR flash devices and their characteristics.

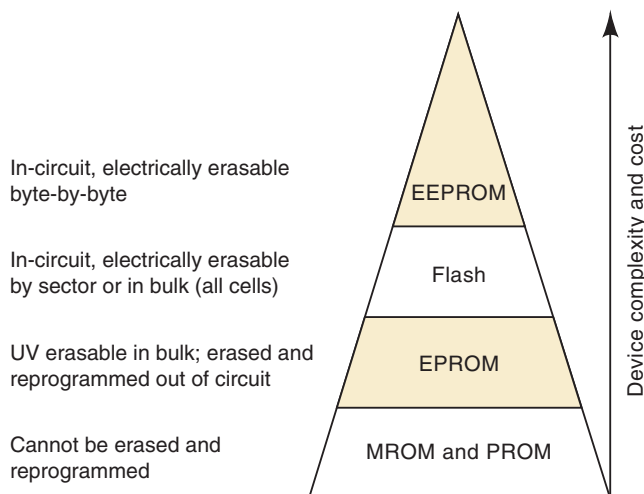
EPROMs were nonvolatile, offered fast read access times and had high density and low cost per bit. They did, however, require removal from their circuit/system to be erased and reprogrammed. EEPROMs were nonvolatile, offered fast read access, and allowed in-circuit erasure and reprogramming of individual bytes. They suffered from lower density and much higher cost than EPROMs.

The challenge for semiconductor engineers was to fabricate a nonvolatile memory with the EEPROM's in-circuit electrical erasability, but with densities and costs much closer to those of EPROMs, while retaining the high-speed read access of both. The response to this challenge was the **flash memory**.

Structurally, a flash memory cell is like the simple single-transistor EPROM cell (and unlike the more complex two-transistor EEPROM cell), being only slightly larger. It has a thinner gate-oxide layer that allows electrical erasability but can be built with much higher densities than EEPROMs. The cost of flash memory is considerably less than for EEPROM. Figure 12-15 illustrates the trade-offs for the various semiconductor nonvolatile memories. As erase/programming flexibility increases (from base to apex of the triangle), so do device complexity and cost. MROM and PROM are the simplest devices, but they cannot be erased and reprogrammed. EEPROM is the most complex and expensive because it can be erased and reprogrammed in circuit on a byte-by-byte basis.

Flash memories are so called because of their rapid erase and write times. Most flash chips can perform a *bulk erase* operation in which all cells on the chip are erased simultaneously or a *sector erase* mode, where specific sectors of the memory array (e.g., 512 bytes) can be erased at one time. This

**FIGURE 12-15** Trade-offs for semiconductor nonvolatile memories show that complexity and cost increase as erase and programming flexibility increases.

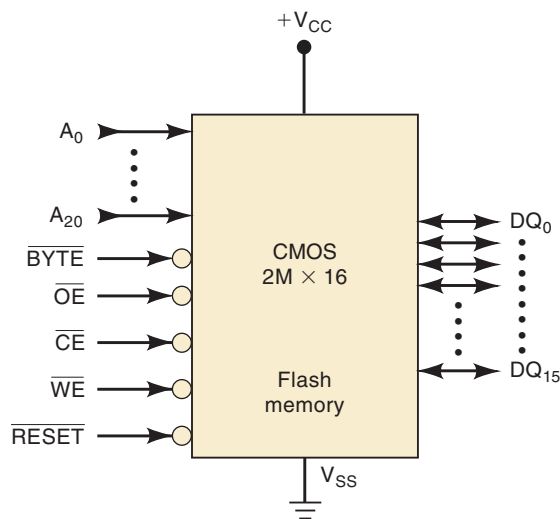


prevents having to erase and reprogram all cells when only a portion of the memory needs to be updated. Flash memory offers a faster write time than EEPROM. Modern IC fabrication technology has brought down the cost per bit of flash memory tremendously. Flash and EEPROM advantages have rendered EPROM, MROM, and PROM technologies obsolete. EEPROM is still used where its advantages over flash justify the extra cost and lower density.

### A Typical CMOS Flash Memory IC

Figure 12-16 shows the logic symbol for a CMOS flash memory chip similar to the IC included on the Altera/Terasic DE1 board, which has a capacity of  $4\text{M} \times 8$  or  $2\text{M} \times 16$ . The diagram shows 21 address inputs ( $A_0$ – $A_{20}$ ) needed to select the different memory addresses; that is,  $2^{21} = 2\text{M} = 2,097,152$ . The 16 data input/output pins ( $DQ_0$ – $DQ_{15}$ ) are used as inputs during memory write operations and as outputs during memory read operations. These data pins float to the Hi-Z state when the chip is deselected ( $\overline{CE} = \text{HIGH}$ ) or when the outputs are disabled ( $\overline{OE} = \text{HIGH}$ ). The write enable input ( $\overline{WE}$ ) is used to control memory write operations.

**FIGURE 12-16** Logic symbol for a typical flash memory chip.



The control inputs ( $\overline{CE}$ ,  $\overline{OE}$ , and  $\overline{WE}$ ) control what happens at the data pins in much the same way as for the 2864 EEPROM. The data pins are normally connected to a data bus. During a write operation, data are transferred over the bus—usually from the microprocessor—and into the chip. During a read operation, data from inside the chip are transferred over the data bus—usually to the microprocessor.

The operation of this flash memory chip can be better understood by looking at its internal structure. Figure 12-17 is a diagram of a typical flash memory chip showing its major functional blocks. The unique feature of this structure is the *command register*, which is used to manage all of the chip functions. Command codes are written into this register to control which operations take place inside the chip (e.g., erase, erase-verify, program, program-verify). These command codes usually come over the data bus from the microprocessor. State control logic examines the contents of the command register and generates logic and control signals to the rest of the chip's circuits to carry out the steps in the operation.

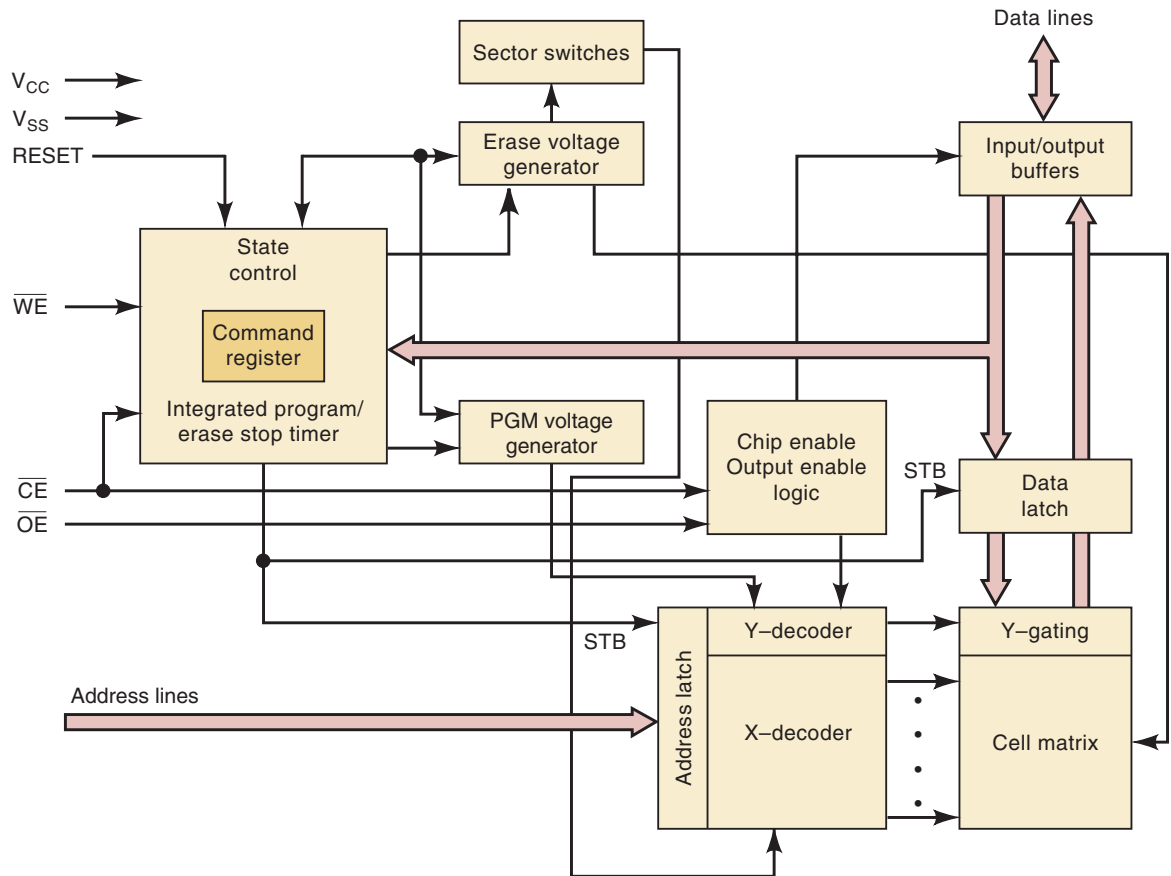


FIGURE 12-17 Functional diagram of a flash memory chip.

### Flash Technology: NOR and NAND

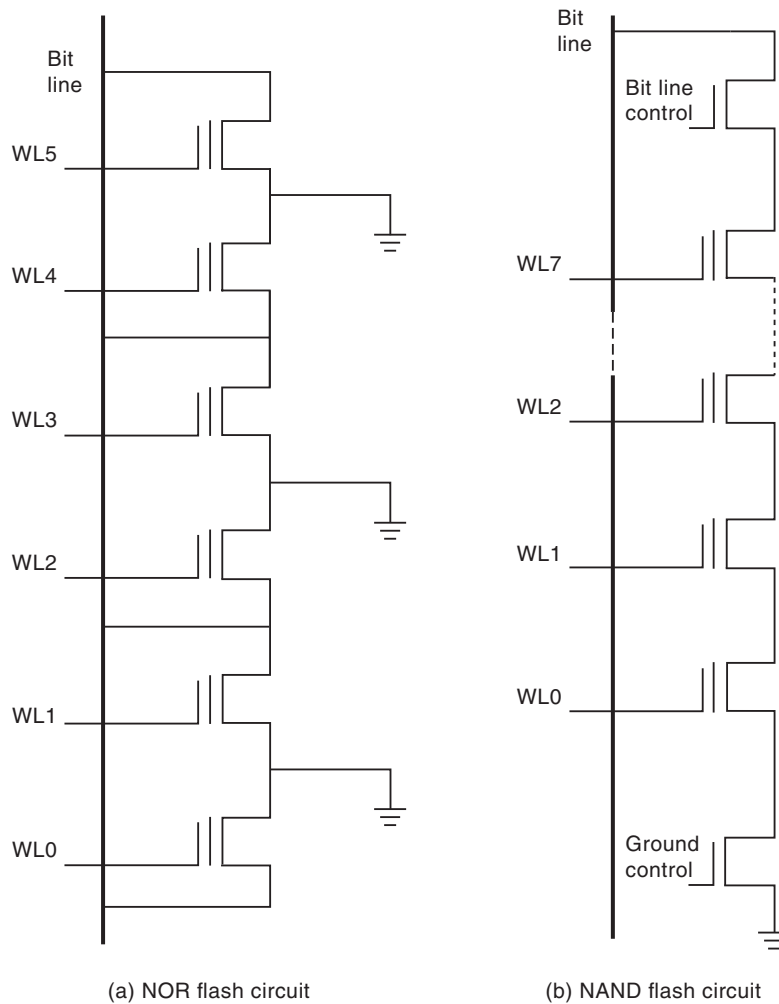
The driving force for technological progress is our demand for devices that have higher capacity, faster operation, lower power, and cost less than what we have today. The first flash devices were created as an attempt to improve on the EEPROM in each of these aspects with the compromise coming in the form of block rather than byte erasure. These flash devices are referred to as *NOR flash* technology. A recent example of a NOR flash IC is the Spansion S29AL032D, which is used on the popular Altera/Terasic DE1 and DE2 boards.

The NOR flash technology utilizes floating-gate MOSFETS (FGMOSFET) that are arranged in parallel with each other between the *bit* line (columns in the matrix) and ground as shown in Figure 12-18(a). Notice that each *word* line (rows in the matrix) controls a transistor switch that can connect the *bit* line (column) to ground. If WL0 OR WL1 OR WL2... OR WL5 is HIGH, then the *bit* line will be pulled LOW. This circuit configuration functions logically like a NOR gate, which is why it is called *NOR flash*. Each transistor can be read or written independent of the status of the other transistors in the group.

The desire to use flash memory as a means to store very large amounts of data resulted in some new design criteria for another category of flash memory products. For mass storage (like a hard disk drive) it is not necessary to have random access to each byte of data. All the data for a document file, a digital picture, or a digital recording are stored sequentially in groups of bytes or sectors. Researchers started to look for ways to improve



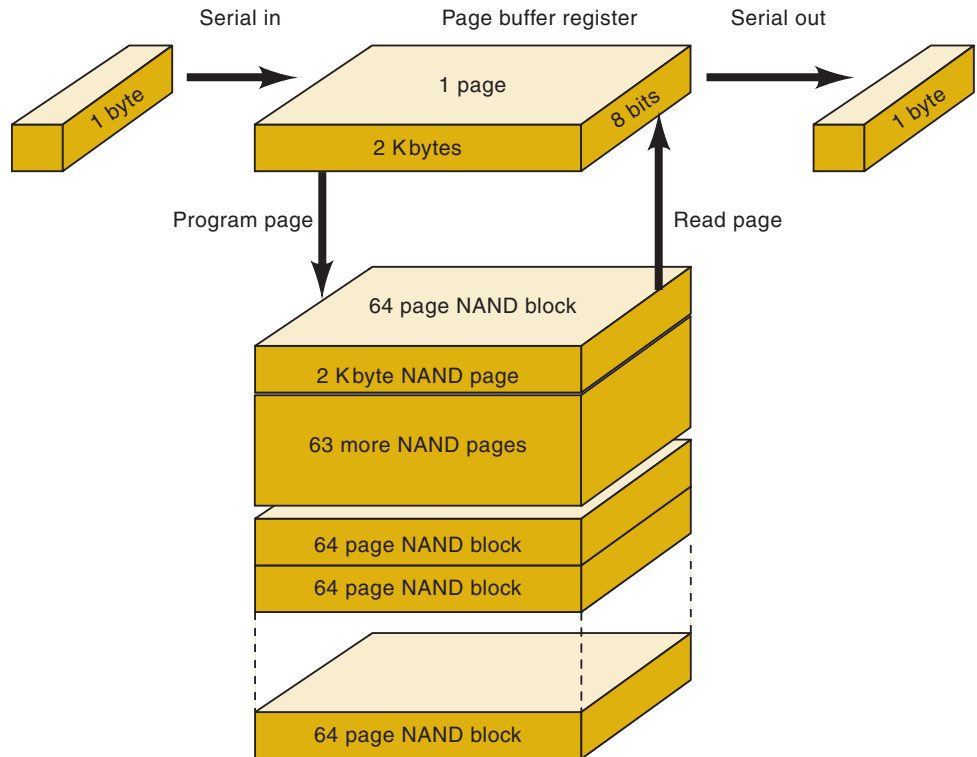
**FIGURE 12-18** (a) Any “ON” transistor can pull the *bit* line LOW; (b) all transistors must be “ON” to pull the *bit* line LOW.



the density of mass-storage flash devices at the sacrifice of random-access capability. The result was **NAND flash** technology, which uses a group of FG MOSFETs in series with each other, connecting the *bit* line with ground as shown in Figure 12-18(b). Notice that to get the *bit* line pulled LOW requires the activation (HIGH) of WL0 AND WL1 AND WL2...AND WL7, which is like the logical function of a NAND gate, hence the name **NAND flash**. For the NAND flash circuit, the data stored on each transistor must be accessed in conjunction with the other *word* lines in its group being activated by a control gate voltage large enough to turn on the other transistors regardless of the amount of charge on the floating gate. For example, to read the data on the transistor connected to WL1, a normal control voltage is applied to WL1, which causes its MOSFET to turn on if a logic 0 is stored on it or remain off if a logic 1 is stored. At the same time, the other transistors are forced on (resulting in a very low resistance between source and drain) by a higher voltage on their *word* lines (WL0, WL2–WL7), which assures that they form a low resistance path and effectively allows the data stored on the MOSFET of WL1 to control the voltage on the *bit* line.

In order to erase/program/read the NAND cells, a page buffer register is associated with each block of NAND cells, as shown in Figure 12-19. Data are shifted into and out of the page buffer register one word at a

**FIGURE 12-19** NAND flash architecture.



time. Some dedicated digital circuitry in the memory IC transfers a page of data from the FG MOSFETs to the page buffer register (for reading) or transfers data from the buffer register to the FG MOSFETs (for writing). It can also erase the data by storing 1s in each transistor. The justification for the increased complexity lies in the space savings of this technique. NAND flash memory can be implemented in a much smaller footprint on the silicon wafer.

Both NAND and NOR flash technologies have advantages and disadvantages. The NAND flash circuit offers fast erase and fast program time but the data must be dealt with in blocks. NOR flash offers quicker read access time and random access. Consequently, NOR flash is usually used for things like BIOS chips for your computer and NAND flash is used for mass storage of pictures, music, and other files in devices like digital camera, memory cards, and USB flash drives. As with most evolving technology, ways are being found to take advantage of the higher density and lower cost per bit of NAND flash, while matching the performance of NOR flash for more and more applications.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the main advantage of flash memory over EPROMs?
2. What is the main advantage of flash memory over EEPROMs?
3. Where does the word *flash* come from?
4. What is the function of a flash memory's command register?
5. Why are logic functions NAND/NOR used to describe flash memory?
6. Which flash configuration is byte erasable?
7. Which flash configuration is used in USB thumb drives?

## 12-9 ROM APPLICATIONS

---

### OUTCOME

Upon completion of this section, you will be able to:

- List and describe some common applications of ROMs in digital systems.

With the exception of MROM and PROM, most ROM devices can be reprogrammed, so technically they are not *read-only* memories. However, the term *ROM* can still be used to include EPROMs, EEPROMs, and flash memory because, during normal operation, the stored contents of these devices is not changed nearly as often as it is read. So ROMs are taken to include all semiconductor, nonvolatile memory devices, and they are used in applications where nonvolatile storage of information, data, or program codes is needed and where the stored data rarely or never change. Here are some of the most common application areas.

### Embedded Microcontroller Program Memory

Microcontrollers are prevalent in most consumer electronic products on the market today. Your car's automatic braking system and engine controller, your cell phone, your digital camcorder, your microwave oven, and many other products have a microcontroller for a brain. These little computers have their program instructions stored in nonvolatile memory—in other words, in a ROM. Most embedded microcontrollers today have flash ROM integrated into the same IC as the CPU. Many also have an area of EEPROM that offers the features of byte erasure and nonvolatile storage.

### Data Transfer and Portability

The need to store and transfer large sets of binary information is a requirement of many low-power battery-operated systems today. Cell phones store photos and video clips. Digital cameras store many pictures on removable memory media. Flash drives connect to a computer's USB port and store gigabytes of information. Your MP3 player is loaded up with music and runs all day on batteries. Your cell phone stores appointment information, email, addresses, and even entire books. All of these common personal electronic gadgets require the low-power, low-cost, high-density, nonvolatile storage with in-circuit write capability that is available in flash memory.

### Bootstrap Memory

Many microcomputers and most larger computers do not have their operating system programs stored in ROM. Instead, these programs are stored in external mass memory, usually magnetic disk. How, then, do these computers know what to do when they are powered on? A relatively small program, called a **bootstrap program**, is stored in ROM. (The term *bootstrap* comes from the idea of pulling oneself up by one's own bootstraps.) When the computer is powered on, it will execute the instructions that are in this bootstrap program. These instructions typically cause the CPU to initialize the system hardware. The bootstrap program then loads the operating system programs from mass storage (disk) into its main internal memory. At that point, the computer begins executing the operating system program and is ready to respond to the user commands. This start-up process is often called "booting up the system."

---

Many of the digital signal processing chips load their internal program memory from an external bootstrap ROM when they are powered on. Some of the more advanced PLDs also load the programming information that configures their logic circuits from an external ROM into a RAM area inside the PLD. This is also done when power is applied. In this way, the PLD is reprogrammed by changing the bootstrap ROM, rather than changing the PLD chip itself.

## Data Tables

ROMs are often used to store tables of data that do not change. Some examples are the trigonometric tables (i.e., sine, cosine, etc.) and code-conversion tables. The digital system can use these data tables to “look up” the correct value. For example, a ROM can be used to store the sine function for angles from  $0^\circ$  to  $90^\circ$ . It could be organized as a  $128 \times 8$  with seven address inputs and eight data outputs. The address inputs represent the angle in increments of approximately  $0.7^\circ$ . For example, address 0000000 is  $0^\circ$ , address 0000001 is  $0.7^\circ$ , address 0000010 is  $1.41^\circ$ , and so on, up to address 1111111, which is  $89.3^\circ$ . When an address is applied to the ROM, the data outputs will represent the approximate sine of the angle. For example, with input address 1000000 (representing approximately  $45^\circ$ ) the data outputs will be 10110101. Because the sine is less than or equal to 1, these data are interpreted as a fraction, that is, 0.10110101, which when converted to decimal equals 0.707 (the sine of  $45^\circ$ ). It is vital that the user of this ROM understands the format in which the data are stored.

Standard look-up-table ROMs for functions such as these were at one time readily available TTL chips. Today, most systems that need to look up equivalent values involve a microprocessor, and the “look-up” table data are stored in the same ROM that holds the program instructions.

## Data Converter

The data-converter circuit takes data expressed in one type of code and produces an output expressed in another type. Code conversion is needed, for example, when a computer is outputting data in straight binary code and we want to convert it to BCD in order to display it on 7-segment LED readouts.

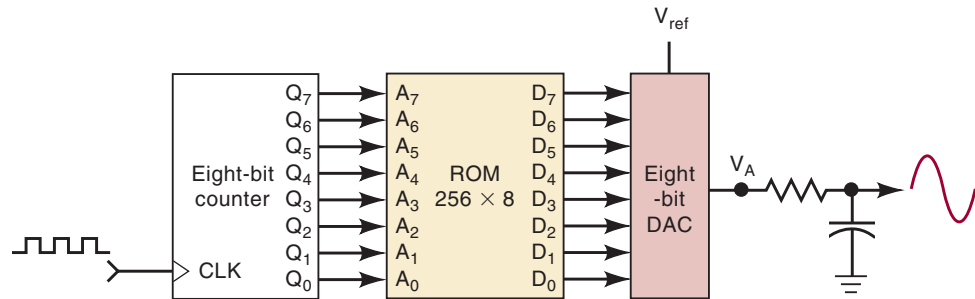
One of the easiest methods of code conversion uses a ROM programmed so that the application of a particular address (the old code) produces a data output that represents the equivalent in the new code. The 74185 is a TTL ROM that stores the binary-to-BCD code conversion for a six-bit binary input. To illustrate, a binary address input of 100110 (decimal 38) will produce a data output of 00111000, which is the BCD code for decimal 38.

## Function Generator

The function generator is a circuit that produces waveforms such as sine waves, sawtooth waves, triangle waves, and square waves. Figure 12-20 shows how a ROM look-up table and a DAC are used to generate a sine-wave output signal.

The ROM stores 256 different eight-bit values, each one corresponding to a different waveform value (i.e., a different voltage point on the sine wave). The eight-bit counter is continuously pulsed by a clock signal to provide sequential address inputs to the ROM. As the counter cycles through the 256 different addresses, the ROM outputs the 256 data points to the DAC. The DAC output will be a waveform that steps through the 256 different analog voltage values corresponding to the data points. The low-pass filter smooths out the steps in the DAC output to produce a smooth waveform.

**FIGURE 12-20** Function generator using a ROM and a DAC.



### OUTCOME ASSESSMENT QUESTIONS

1. Describe how a computer uses a bootstrap program.
2. What is a code converter?
3. What are the main elements of a function generator?

## 12-10 SEMICONDUCTOR RAM

### OUTCOMES

Upon completion of this section, you will be able to:

- Define RAM as it is normally used.
- Describe the role of RAM in systems.

Recall that the term *RAM* stands for *random-access memory*, meaning that any memory address location is as easily accessible as any other. Many types of memory can be classified as having random access, but when the term *RAM* is used with semiconductor memories, it is usually taken to mean read/write memory (RWM) as opposed to ROM. Because it is common practice to use RAM to mean semiconductor RWM, we will do so throughout the following discussions.

RAM is used in computers for the *temporary* storage of programs and data. The contents of many RAM address locations will be read from and written to as the computer executes a program. This requires fast read and write cycle times for the RAM so as not to slow down the computer operation.

A major disadvantage of RAM is that it is volatile and will lose all stored information if power is interrupted or turned off. Some CMOS RAMs, however, use such small amounts of power in the standby mode (no read or write operations taking place) that they can be powered from batteries whenever the main power is interrupted. Of course, the main advantage of RAM is that it can be written into and read from rapidly with equal ease.

The following discussion of RAM will draw on some of the material covered in our treatment of ROM because many of the basic concepts are common to both types of memory.

### OUTCOME ASSESSMENT QUESTIONS

1. Is RAM volatile?
2. How does RAM differ from ROM?
3. *True or False:* RAM is the only memory device that is randomly accessed.

## 12-11 RAM ARCHITECTURE

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the functional blocks that make up a RAM IC.
- Describe the sequence of operations performed on the control inputs of a RAM to read and write.
- Relate configuration to the number of pins in a RAM IC.

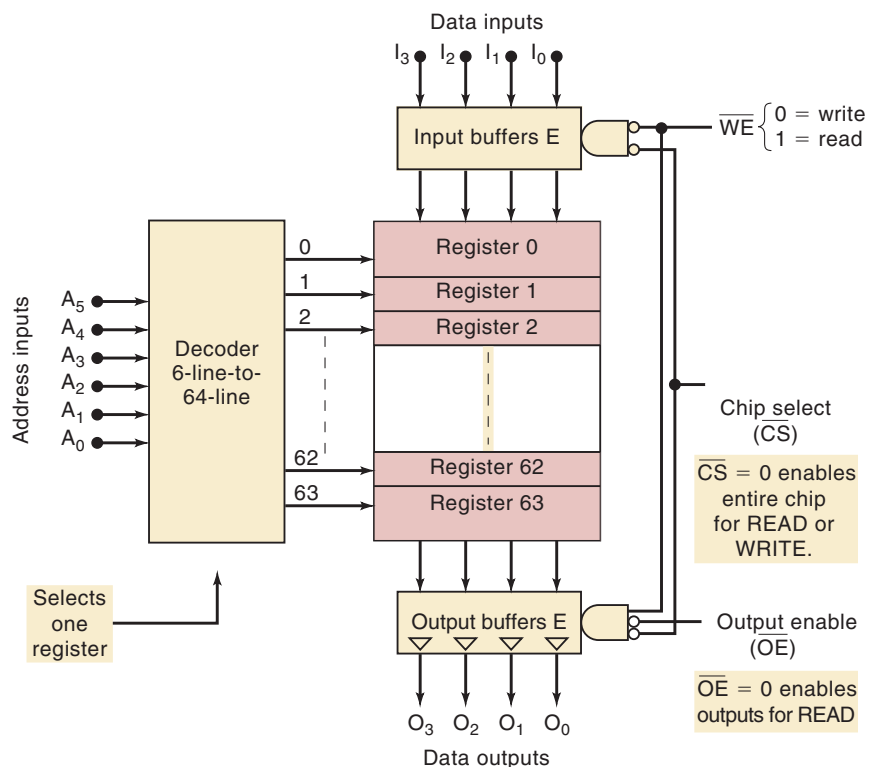
As with the ROM, it is helpful to think of the RAM as consisting of a number of registers, each storing a single data word, and each having a unique address. RAMs typically come with word capacities of 1K, 4K, 8K, 16K, 64K, 128K, 256K, and 1024K, and with word sizes of one, four, or eight bits. As we will see later, the word capacity and the word size can be expanded by combining memory chips.

Figure 12-21 shows the simplified architecture of a RAM that stores 64 words of four bits each (i.e., a  $64 \times 4$  memory). These words have addresses ranging from 0 to  $63_{10}$ . In order to select one of the 64 address locations for reading or writing, a binary address code is applied to a decoder circuit. Because  $64 = 2^6$ , the decoder requires a six-bit input code. Each address code activates one particular decoder output, which in turn enables its corresponding register. For example, assume an applied address code of

$$A_5A_4A_3A_2A_1A_0 = 011010$$

Because  $011010_2 = 26_{10}$ , decoder output 26 will go high, selecting register 26 for either a read or a write operation.

**FIGURE 12-21** Internal organization of a  $64 \times 4$  RAM.



## Read Operation

The address code picks out one register in the memory chip for reading or writing. In order to *read* the contents of the selected register, the write enable ( $\overline{WE}$ )\* input must be a 1. In addition, the CHIP SELECT ( $\overline{CS}$ ) input must be activated (a 0 in this case). The combination of  $\overline{WE} = 1$ ,  $\overline{CS} = 0$ , and  $\overline{OE} = 0$  enables the output buffers so that the contents of the selected register will appear at the four data outputs.  $\overline{WE} = 1$  also *disables* the input buffers so that the data inputs do not affect the memory during a read operation.

## Write Operation

To write a new four-bit word into the selected register requires  $\overline{WE} = 0$  and  $\overline{CS} = 0$ . This combination *enables* the input buffers so that the four-bit word applied to the data inputs will be loaded into the selected register. The  $\overline{WE} = 0$  also *disables* the output buffers, which are tristate, so that the data outputs are in their Hi-Z state during a write operation. The write operation, of course, destroys the word that was previously stored at that address.

## Chip Select

Most memory chips have one or more CS inputs that are used to enable the entire chip or disable it completely. In the disabled mode, all data inputs and data outputs are disabled (Hi-Z) so that neither a read nor a write operation can take place. In this mode, the contents of the memory are unaffected. The reason for having CS inputs will become clear when we combine memory chips to obtain larger memories. Note that many manufacturers call these inputs *chip enable* (CE). When the CS or CE inputs are in their active state, the memory chip is said to be *selected*; otherwise, it is said to be *deselected*. Many memory ICs are designed to consume much less power when they are deselected. In large memory systems, for a given memory operation, one or more memory chips will be selected while all others are deselected. More will be said on this topic later.

## Common Input/Output Pins

In order to conserve pins on an IC package, manufacturers often combine the data input and data output functions using common input/output pins. The  $\overline{OE}$  and  $\overline{WE}$  inputs control the function of these I/O pins. During a read operation, when  $\overline{WE} = 1$  and  $\overline{OE} = 0$ , the I/O pins act as data outputs that reproduce the contents of the selected address location. During a write operation, when  $\overline{WE} = 0$  and  $\overline{OE} = 1$ , the I/O pins act as data inputs to which the data to be written are applied.

We can see why this is done by considering the chip in Figure 12-21. With separate input and output pins, a total of 19 pins is required (including ground and power supply). With four common I/O pins, only 15 pins are required. The pin saving becomes even more significant for chips with larger word size.

In most applications, memory devices are used with a bidirectional data bus like we studied in Chapter 9. For this type of system, even if the

---

\*Some manufacturers use the symbol  $R/\overline{W}$  (read/write) or  $\overline{W}$  instead of  $\overline{WE}$ . In any case, the operation is the same.

memory chip had separate input and output pins, they would be connected together on the same data bus. A RAM having separate input and output pins is referred to as dual-port RAM. These are used in applications where speed is very important and the data in comes from a different device than the data out is going to. A good example is the video RAM on your PC. The RAM must be read repeatedly by the video card to refresh the screen and constantly filled with new updated information from the system bus.

### OUTCOME ASSESSMENT QUESTIONS

1. Describe the input conditions needed to read a word from a specific RAM address location.
2. Why do some RAM chips have common input/output pins?
3. How many pins are required for a  $64\text{K} \times 4$  RAM with one CS input, one  $R/\overline{W}$  control input, power, ground, and common I/O?

## 12-12 STATIC RAM (SRAM)

### OUTCOMES

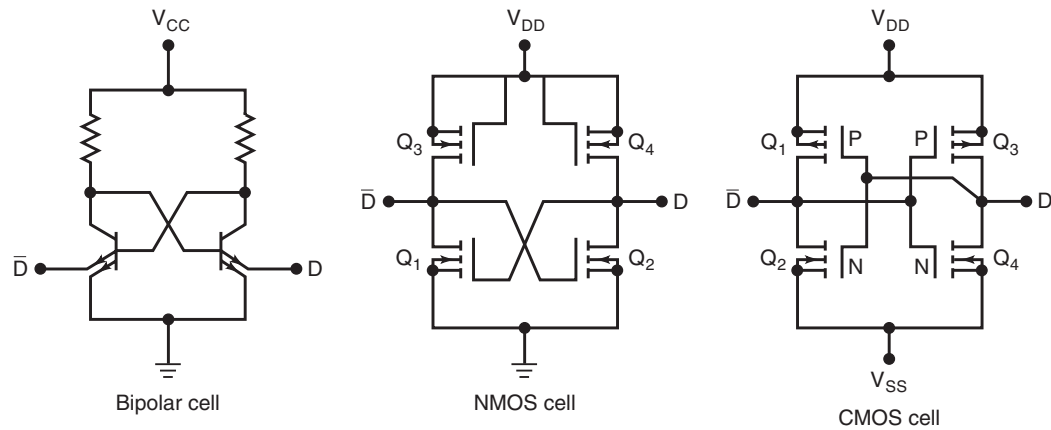
*Upon completion of this section, you will be able to:*

- Describe the means by which 1s and 0s are stored in a RAM cell.
- Distinguish between the circuits used as memory cells in bipolar, NMOS, and CMOS memory devices.
- Define Static.
- Describe the sequence of events of a read and write cycle for RAM.
- Interpret timing diagrams that define the limitations associated with control signals of RAMs.

The RAM operation that we have been discussing up to this point applies to a **static RAM**—one that can store data as long as power is applied to the chip. Static-RAM memory cells are essentially flip-flops that will stay in a given state (store a bit) indefinitely, provided that power to the circuit is not interrupted. In Section 12-13, we will describe **dynamic RAM**, which stores data as charges on capacitors. With dynamic RAMs, the stored data will gradually disappear because of capacitor discharge, so it is necessary to **refresh** the data periodically (i.e., recharge the capacitors).

Static RAMs (SRAMs) have been manufactured in bipolar, MOS, and BiCMOS technologies; the majority of applications today use CMOS RAMs. The same advantages and disadvantages that characterize these technologies in logic circuits apply to memories as well. Figure 12-22 shows a comparison of the fundamental latch circuitry used in bipolar, NMOS, and CMOS technologies. The bipolar cell is fast, power hungry, and takes up a lot of space on the silicon wafer because the bipolar transistor is more complex than a MOSFET and resistors are relatively large. The NMOS cell uses MOSFETs ( $Q_3$  and  $Q_4$ ) as the pull-up resistors, making it smaller, and the resistance values make it operate using less power. However, in both of these cells, there is always current flowing through one side of the latch circuit or the other. The CMOS cell eliminates this problem by using P-type and N-type MOSFETs. In either state of the CMOS latch, there is almost no current flowing from  $V_{DD}$  to  $V_{SS}$ . The result is the lowest power consumption, high-speed operation, but more complex circuitry, which result in a





**FIGURE 12-22** Typical bipolar, NMOS, and CMOS static-RAM cells.

larger footprint on the silicon wafer. The transistors that allow the *word* line to select the cell are not shown in this diagram for the sake of simplicity, but they also add to the size of the static RAM cell.

### Static-RAM Timing

RAM ICs are most often used as the internal memory of a computer. The CPU (central processing unit) continually performs read and write operations on this memory at a very fast rate that is determined by the limitations of the CPU. The memory chips that are interfaced to the CPU must be fast enough to respond to the CPU read and write commands, and a computer designer must be concerned with the RAM's various timing characteristics.

Not all RAMs have the same timing characteristics, but most of them are similar, and so we will use a typical set of characteristics for illustrative purposes. The nomenclature for the different timing parameters will vary from one manufacturer to another, but the meaning of each parameter is usually easy to determine from the memory timing diagrams on the RAM data sheets. Figure 12-23 shows the timing diagrams for a complete read cycle and a complete write cycle for a typical RAM chip.

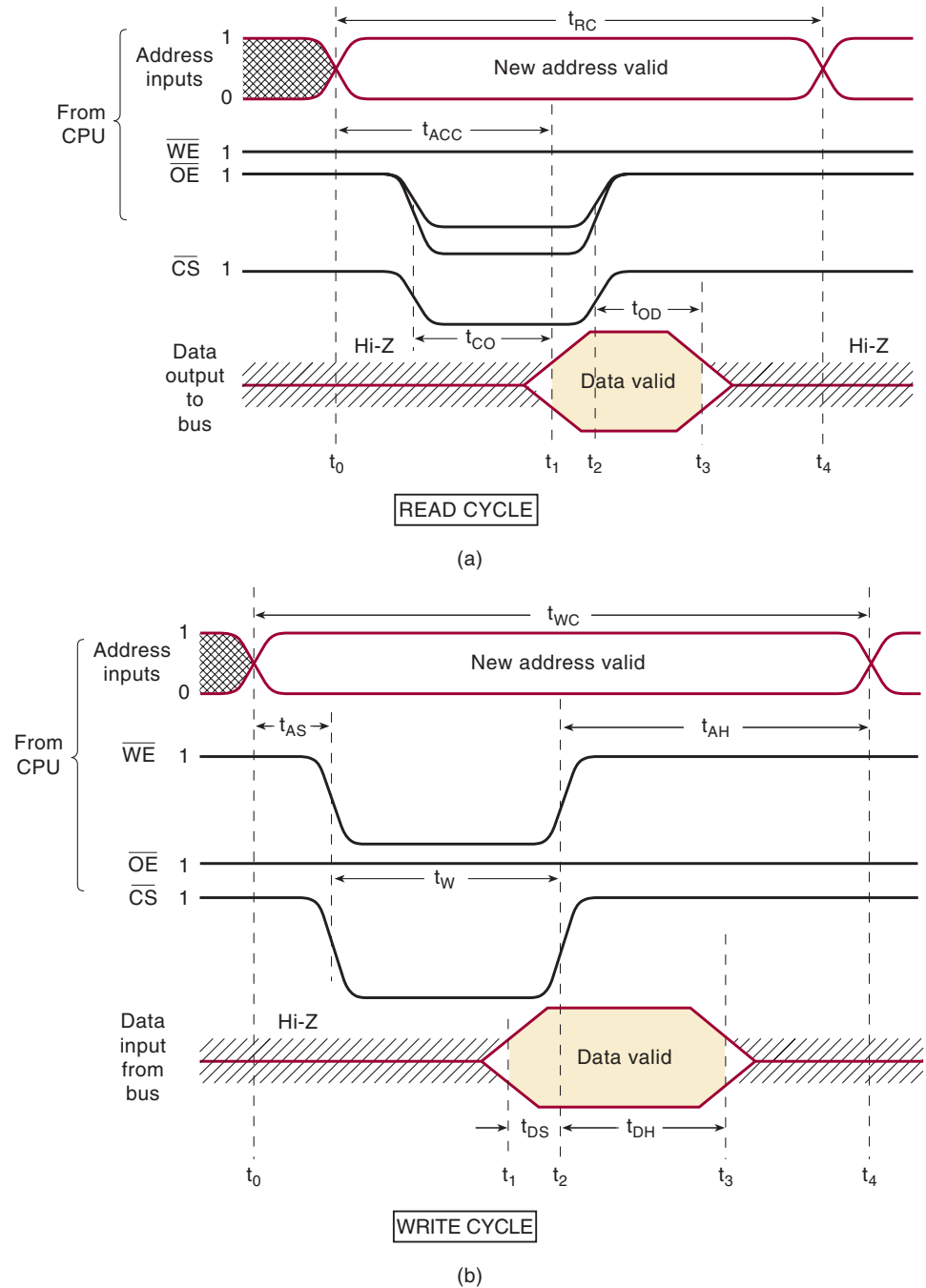
### Read Cycle

The waveforms in Figure 12-23(a) show how the address,  $\overline{WE}$ ,  $\overline{OE}$ , and chip select inputs behave during a memory read cycle. As noted, the CPU supplies these input signals to the RAM when it wants to read data from a specific RAM address location. Although a RAM may have many address inputs coming from the CPU's address bus, for clarity the diagram represents them as a bus that is changing or holding a steady value. The RAM's data output is also shown using this same method. Recall that the RAM's data output is connected to the CPU data bus (Figure 12-5).

The read cycle begins at time  $t_0$ . Prior to that time, the address inputs will be whatever address is on the address bus from the preceding operation. Because the RAM's chip select is not active, it will not respond to its "old" address. Note that the  $\overline{WE}$  line is HIGH prior to  $t_0$  and stays HIGH throughout the read cycle. In most memory systems,  $\overline{WE}$  is normally kept in the HIGH state except when it is driven LOW during a write cycle. The RAM's data output is in its Hi-Z state because  $\overline{CS} = 1$  and  $\overline{OE} = 1$ .

At  $t_0$ , the CPU applies a new address to the RAM inputs; this is the address of the location to be read. After allowing time for the address signals

**FIGURE 12-23** Typical timing for static RAM: (a) read cycle; (b) write cycle.



to stabilize, the  $\overline{CS}$  line is activated. In this diagram, the output enable is activated at the same time. Recall that both  $\overline{CS}$  and  $\overline{OE}$  must be asserted in order to access any memory location and turn on the tri-state drivers, respectively. The RAM responds by placing the data from the addressed location onto the data output line at  $t_1$ . The time between  $t_0$  and  $t_1$  is the RAM's access time,  $t_{ACC}$ , and is the time between the application of the new address and the appearance of valid output data. The timing parameter,  $t_{CO}$ , is the time it takes for the RAM output to go from Hi-Z to a valid data level once  $\overline{CS}$  and  $\overline{OE}$  or both are activated. A time may be specified for the output to become valid after  $\overline{CS}$  and a separate time from  $\overline{OE}$  until the data becomes valid. For simplicity, we are assuming they are both the same and refer to it as  $t_{CO}$ .

At time  $t_2$ , the  $\overline{CS}$  and  $\overline{OE}$  are returned HIGH, and the RAM output returns to its Hi-Z state after a time interval,  $t_{OD}$ . Thus, the RAM data will be on the data bus between  $t_1$  and  $t_3$ . The CPU can take the data from the data bus at any point during this interval and will latch these data into one of its internal registers.

The complete read cycle time,  $t_{RC}$ , extends from  $t_0$  to  $t_4$ , when the CPU changes the address inputs to a different address for the next read or write cycle.

## Write Cycle

Figure 12-23(b) shows the signal activity for a write cycle that begins when the CPU supplies a new address to the RAM at a time  $t_0$ . The CPU drives the  $\overline{WE}$  and  $\overline{CS}$  lines LOW after waiting for a time interval  $t_{AS}$ , called the *address setup time*. This gives the RAM's address decoders time to respond to the new address.  $\overline{WE}$  and  $\overline{CS}$  are held LOW for a time interval  $t_W$ , called the write time interval.

During this write time interval, at time  $t_1$ , the CPU applies valid data to the data bus to be written into the RAM. These data must be held at the RAM input for at least a time interval  $t_{DS}$  prior to, and for at least a time interval  $t_{DH}$  after, the deactivation of  $\overline{WE}$  and  $\overline{CS}$  at  $t_2$ . The  $t_{DS}$  interval is called the *data setup time*, and  $t_{DH}$  is called the *data hold time*. Similarly, the address inputs must remain stable for the address hold time interval,  $t_{AH}$ , after  $t_2$ . If any of these setup time or hold time requirements are not met, the write operation will not take place reliably.

The complete write-cycle time,  $t_{WC}$ , extends from  $t_0$  to  $t_4$ , when the CPU changes the address lines to a new address for the next read or write cycle.

The read-cycle time,  $t_{RC}$ , and write-cycle time,  $t_{WC}$ , are what essentially determine how fast a memory chip can operate. For example, in an actual application, a CPU will often be reading successive data words from memory one right after the other. If the memory has a  $t_{RC}$  of 50 ns, the CPU can read one word every 50 ns, or 20 million words per second; with  $t_{RC} = 10$  ns, the CPU can read 100 million words per second.

### OUTCOME ASSESSMENT QUESTIONS

1. How does a static-RAM cell differ from a dynamic-RAM cell?
2. Which memory technology generally uses the least power?
3. What device places data on the data bus during a read cycle?
4. What device places data on the data bus during a write cycle?
5. What RAM timing parameters determine its operating speed?

## 12-13 DYNAMIC RAM (DRAM)

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the means by which 1s and 0s are stored on DRAM.
- Identify applications of DRAM.
- Describe strengths and limitations of DRAM relative to other memory devices.

Dynamic RAM has been around since the 1960s. Since then, the technology has made huge improvements in capacity, density, and speed, yet the fundamental principles of operation have remained essentially the same. This discussion will use examples that are very small in capacity relative to the DRAM chips that are working in your computer or mobile device. The primary difference is in the number of address lines.

Dynamic RAMs are fabricated using MOS technology and are noted for their high capacity, low power requirement, and moderate operating speed. As we stated earlier, unlike static RAMs, which store information in FFs, dynamic RAMs store 1s and 0s as charges on a small MOS capacitor (typically a few picofarads). Because of the tendency for these charges to leak off after a period of time, dynamic RAMs require periodic recharging of the memory cells; this is called *refreshing* the dynamic RAM. Each RAM chip has a specific interval in which each cell must be refreshed. This ranges from 2 ms for older DRAM chips to 64 ms for modern Double Data Rate (DDR) DRAM chips.

The need for refreshing is a drawback of dynamic RAM compared to static RAM because it may require external support circuitry. Some DRAM chips have built-in refresh control circuitry that does not require extra external hardware but does require special timing of the chip's input control signals. Additionally, as we shall see, the address inputs to a DRAM must be handled in a less straightforward way than SRAM. So, all in all, designing with and using DRAM in a system is more complex than with SRAM. However, their much larger capacities and much lower power consumption make DRAMs the memory of choice in systems where the most important design considerations are keeping down size, cost, and power.

For applications where speed and reduced complexity are more critical than cost, space, and power considerations, static RAMs are still the best. SRAMs are generally faster than dynamic RAMs and require no refresh operation. They are simpler to design with, but they cannot compete with the higher capacity and lower power requirement of dynamic RAMs.

Because of their simple cell structure, DRAMs typically have four times the density of SRAMs. This increased density allows four times as much memory capacity to be placed on a single board; alternatively, it requires one-fourth as much board space for the same amount of memory. The cost per bit of dynamic RAM storage is typically one-fifth to one-fourth that of static RAMs. An additional cost saving is realized because the lower power requirements of a dynamic RAM, typically one-sixth to one-half those of a static RAM, allow the use of smaller, less expensive power supplies.

The main applications of SRAMs are in areas where only small amounts of memory are needed or where high speed is required. Many microprocessor-controlled instruments and appliances have very small memory capacity requirements. Some instruments, such as digital storage oscilloscopes and logic analyzers, require very high-speed memory. For applications such as these, SRAM is normally used.

The main internal memory of most personal computers, notebooks, tablets, and cell phones is DRAM because of its high capacity and low power consumption. These computers, however, sometimes use some small amounts of SRAM for functions requiring maximum speed, such as video graphics, look-up tables, and cache memory.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What are the main drawbacks of dynamic RAM compared with static?
2. List the advantages of dynamic RAM compared with static RAM.
3. Which type of RAM would you expect to find on the main memory modules of your PC?

## 12-14 DYNAMIC RAM STRUCTURE AND OPERATION

### OUTCOMES

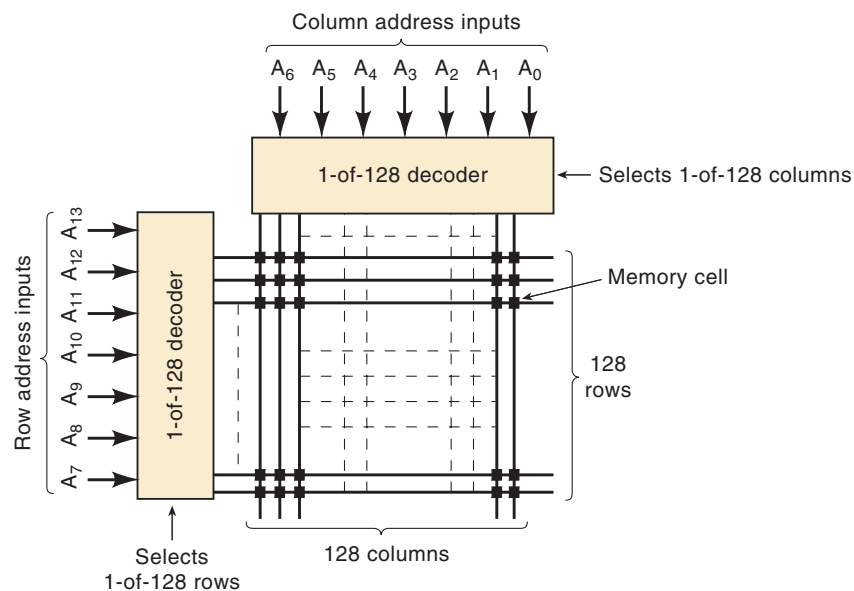
Upon completion of this section, you will be able to:

- Describe the process of reading, writing, and refreshing DRAM.
- Explain the need for address multiplexing.
- Relate configuration to the number of input pins.
- Describe the role of control bus inputs.

The dynamic RAM's internal architecture can be visualized as an array of single-bit cells, as illustrated in Figure 12-24. Here, 16,384 cells are arranged in a  $128 \times 128$  array. Each cell occupies a unique row and column position within the array. Fourteen address inputs are needed to select one of the cells ( $2^{14} = 16,384$ ); the lower address bits,  $A_0$  to  $A_6$ , select the column, and the higher-order bits,  $A_7$  to  $A_{13}$ , select the row. Each 14-bit address selects a unique cell to be written into or read from. The structure in Figure 12-24 is a  $16K \times 1$  DRAM chip. DRAM chips are currently available in various configurations. DRAMs with an eight-bit (or greater) word size have a cell arrangement similar to that of Figure 12-24 except that each position in the array contains eight cells, and each applied address selects a group of eight cells for a read or a write operation. As we will see later, larger word sizes can also be attained by combining several chips in the appropriate arrangement.

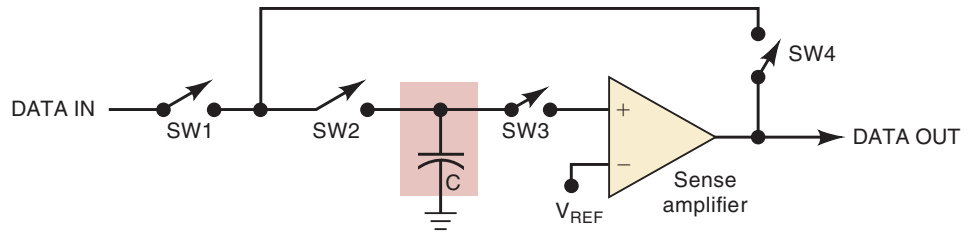
Figure 12-25 is a symbolic representation of a dynamic memory cell and its associated circuitry. Many of the circuit details are not shown, but this

**FIGURE 12-24** Cell arrangement in a  $16K \times 1$  dynamic RAM.



**FIGURE 12-25**

Symbolic representation of a dynamic memory cell. During a WRITE operation, semiconductor switches SW1 and SW2 are closed. During a read operation, all switches are closed except SW1.



simplified diagram can be used to describe the essential ideas involved in writing to and reading from a DRAM. The switches SW1 through SW4 are actually MOSFETs that are controlled by various address decoder outputs and the  $\overline{WE}$  signal. The capacitor, of course, is the actual storage cell. One sense amplifier would serve an entire column of memory cells but operate only on the bit in the selected row.

To write data to the cell, signals from the address decoding and read/write logic will close switches SW1 and SW2, while keeping SW3 and SW4 open. This connects the input data to C. A logic 1 at the data input charges C, and a logic 0 discharges it. Then the switches are open so that C is disconnected from the rest of the circuit. Ideally, C would retain its charge indefinitely, but there is always some leakage path through the off switches, so that C will gradually lose its charge.

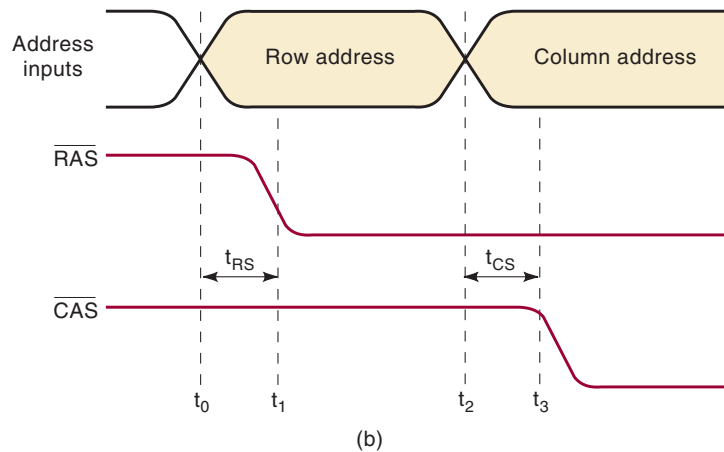
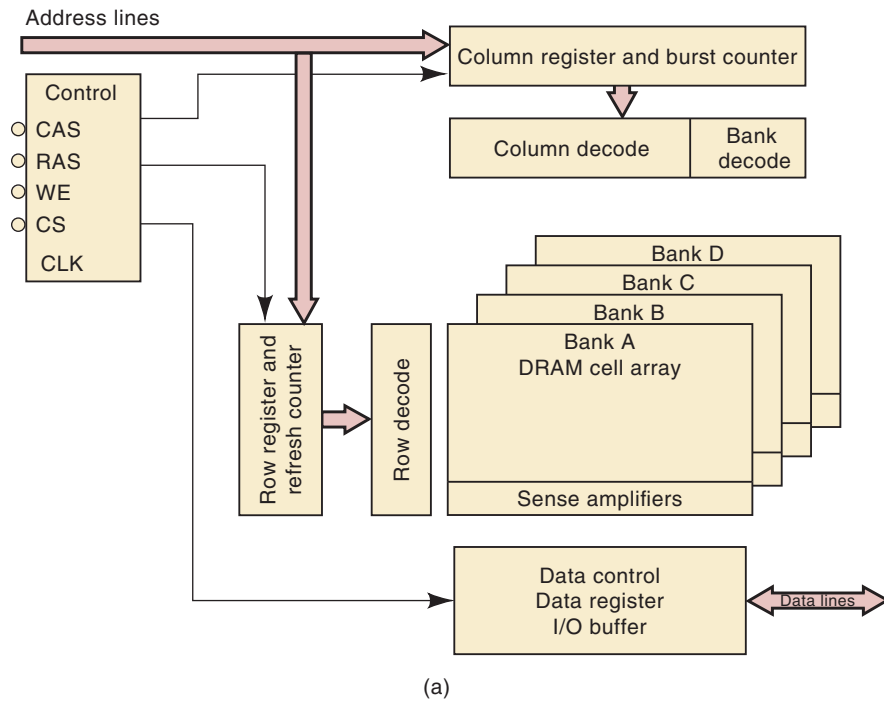
To read data from the cell, switches SW2, SW3, and SW4 are closed, and SW1 is kept open. This connects the stored capacitor voltage to the *sense amplifier*. The sense amplifier compares the voltage with some reference value to determine if it is a logic 0 or 1, and it produces a solid 0 V or 5 V for the data output. This data output is also connected to C (SW2 and SW4 are closed) and refreshes the capacitor voltage by recharging or discharging. In other words, the data bit in a memory cell is refreshed each time it is read.

## Address Multiplexing

The  $16K \times 1$  DRAM array depicted in Figure 12-24 has very limited capacity by today's standards. It has 14 address bits; a  $64K \times 1$  DRAM array would have 16 address inputs. A  $1M \times 4$  DRAM needs 20 address bits; a  $4M \times 1$  needs 22 address bits. Modern DRAMs have capacities of gigabits. High-capacity memory chips such as these would require many pins if each address bit required a separate pin. In order to reduce the number of pins on their high-capacity DRAM chips, manufacturers utilize **address multiplexing** whereby each address input pin can accommodate two different address bits. The saving in pin count translates to a significant decrease in the size of the IC packages. This is very important in large-capacity memory boards, where you want to maximize the amount of memory that can fit on one board.

A general example of the structure of a DRAM is shown in Figure 12-26. Depending on the capacity, number of data bits per location, and manufacturer, the internal organization of a particular memory IC will be slightly different; however, we will focus on aspects that will be common to all DRAMs. The memory cells are arranged in several banks of rectangular arrays. A single row (for each bank) is selected by the row decoder. The column address is decoded and used to select one of the banks and select one column for each bit in the data word. For example, if this device uses an eight-bit data word, then a given column address would enable the eight columns that make up that eight-bit memory location. The multiplexed address scheme that we described earlier requires that the entire address cannot

**FIGURE 12-26**  
 (a) Simplified architecture  
 of a typical DRAM;  
 (b)  $\overline{RAS}$ / $\overline{CAS}$  timing.



be applied at one time. Rather, it is applied in two parts: the row address and then the column address. Notice that the address lines are connected directly to both the row address register and the column address register. The row register stores the upper part of the address, and the column register stores the lower part. Two very important timing signals are used to control when the address information is latched into these registers. The **row address strobe** ( $\overline{RAS}$ ) stores the contents of the address inputs into the row address register. The **column address strobe** ( $\overline{CAS}$ ) stores the contents of the address inputs into the column address register.

A full address is applied to a typical DRAM in two steps using  $\overline{RAS}$  and  $\overline{CAS}$ . The timing is shown in Figure 12-26(b). Initially,  $\overline{RAS}$  and  $\overline{CAS}$  are both HIGH. At time  $t_0$ , the row address (i.e., the upper half of the full address) is applied to the address inputs. After allowing time for the setup

time requirement ( $t_{RS}$ ) of the row address register, the  $\overline{RAS}$  input is driven LOW at  $t_1$ . This NGT loads the row address into the row address register so that the upper address bits now appear at the row decoder inputs. The LOW at  $\overline{RAS}$  also enables this decoder so that it can decode the row address and select one row of the array.

At time  $t_2$ , the column address (i.e., the lower half of the full address) is applied to the address inputs. At  $t_3$ , the  $\overline{CAS}$  input is driven LOW to load the column address into the column address register.  $\overline{CAS}$  also enables the column decoder so that it can decode the column address and select the columns of the array.

At this point the two parts of the address are in their respective registers, the decoders have decoded them to select the data cells corresponding to the row and column address, and a read or a write operation can be performed on those cells just as in a static RAM.

As you can see, there are several operations that must be performed before the data that is stored in the DRAM can actually appear on the outputs. The term **latency** is often used to describe the time required to perform these operations. Each operation takes a certain amount of time, and this amount of time determines the maximum rate at which we can access data in the memory.

In a simple computer system, the address inputs to the memory system come from the central processing unit (CPU). When the CPU wants to access a particular memory location, it generates the complete address and places it on address lines that make up an address bus. Figure 12-27(a) shows this for a scaled-down memory system that has a capacity of 64K words and therefore requires a 16-line address bus going directly from the CPU to the memory.

This arrangement works for ROM or for static RAM, but it must be modified for DRAM that uses multiplexed addressing. If all 64K of the memory is DRAM, it will have only eight address inputs. This means that the 16 address lines from the CPU address bus must be fed into a multiplexer circuit that will transmit eight address bits at a time to the memory address inputs. This is shown symbolically in Figure 12-27(b). The multiplexer select input, labeled *MUX*, controls whether CPU address lines  $A_0$  to  $A_7$  or address lines  $A_8$  to  $A_{15}$  will be present at the DRAM address inputs.

The timing of the *MUX* signal must be synchronized with the  $\overline{RAS}$  and  $\overline{CAS}$  signals that clock the addresses into the DRAM. This is shown in Figure 12-28. *MUX* must be LOW when  $\overline{RAS}$  is pulsed LOW so that address lines  $A_8$  to  $A_{15}$  from the CPU will reach the DRAM address inputs to be loaded on the NGT of  $\overline{RAS}$ . Likewise, *MUX* must be HIGH when  $\overline{CAS}$  is pulsed LOW so that  $A_0$  to  $A_7$  from the CPU will be present at the DRAM inputs to be loaded on the NGT of  $\overline{CAS}$ .

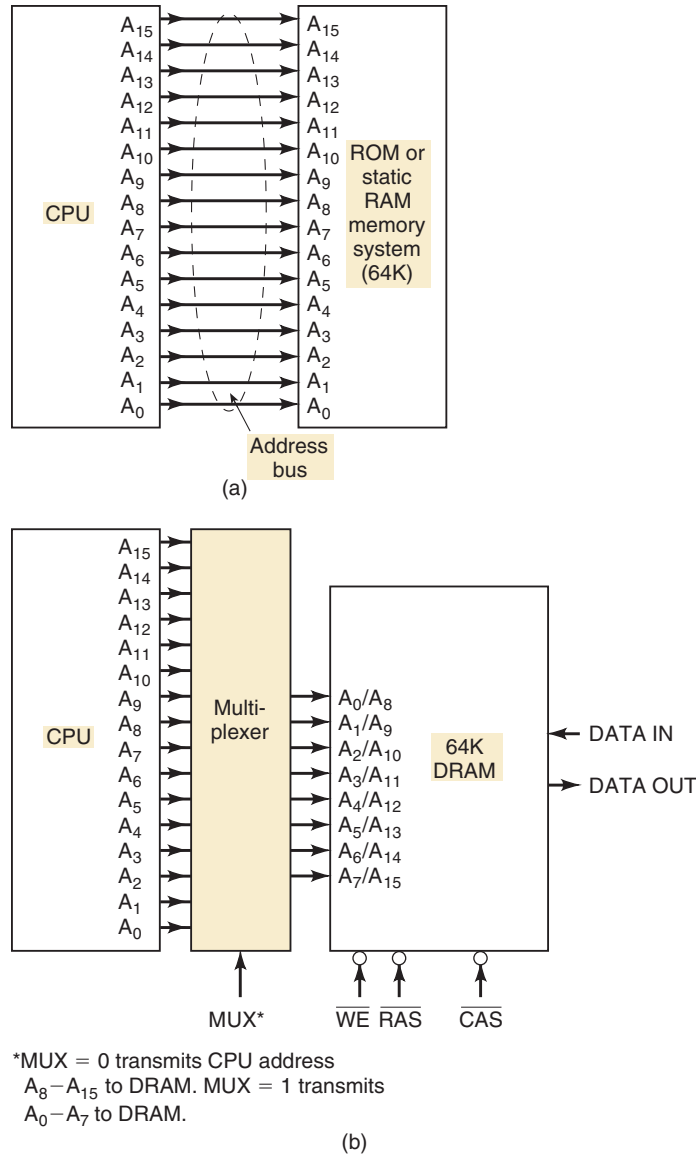
The actual multiplexing and timing circuitry will not be shown here but will be left to the end-of-chapter problems (Problems 12-26 and 12-27).

#### OUTCOME ASSESSMENT QUESTIONS

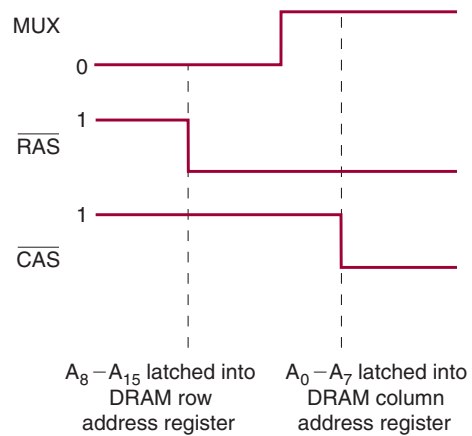
1. Describe the array structure of a  $64K \times 1$  DRAM.
2. What is the benefit of address multiplexing?
3. How many address inputs would there be on a  $1M \times 1$  DRAM chip?
4. What are the functions of the  $\overline{RAS}$  and  $\overline{CAS}$  signals?
5. What is the function of the *MUX* signal?



**FIGURE 12-27** (a) CPU address bus driving ROM or static-RAM memory; (b) CPU addresses driving a multiplexer that is used to multiplex the CPU address lines into the DRAM.



**FIGURE 12-28** Timing required for address multiplexing.



## 12-15 DRAM READ/WRITE CYCLES

### OUTCOME

Upon completion of this section, you will be able to:

- Describe the sequence of events to perform a read and write on a DRAM.

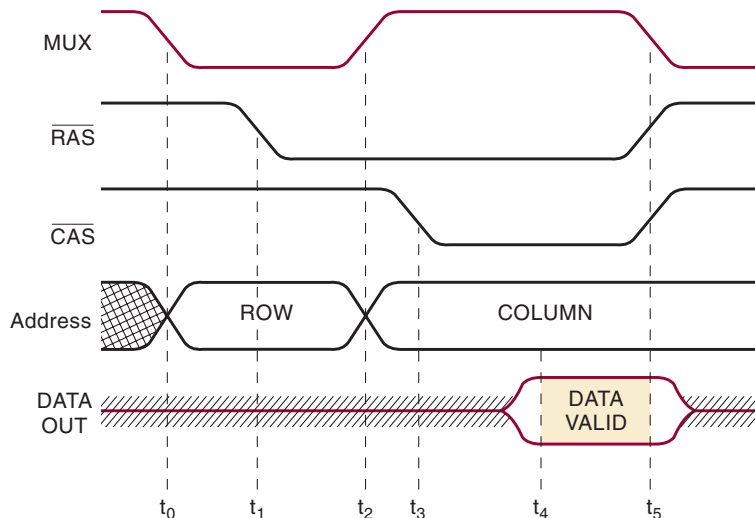
The timing of the read and write operations of a DRAM is much more complex than for a static RAM, and there are many critical timing requirements that the DRAM memory designer must consider. At this point, a detailed discussion of these requirements would probably cause more confusion than enlightenment. We will concentrate on the basic timing sequence for the read and write operations for a small DRAM system like that of Figure 12-27(b).

### DRAM Read Cycle

Figure 12-29 shows typical signal activity during the read operation. It is assumed that  $\overline{WE}$  is in its HIGH state throughout the operation. The following is a step-by-step description of the events that occur at the times indicated on the diagram.

- $t_0$ : MUX is driven LOW to apply the row address bits ( $A_8$  to  $A_{15}$ ) to the DRAM address inputs.
- $t_1$ :  $\overline{RAS}$  is driven LOW to load the row address into the DRAM.
- $t_2$ : MUX goes HIGH to place the column address ( $A_0$  to  $A_7$ ) at the DRAM address inputs.
- $t_3$ :  $\overline{CAS}$  goes LOW to load the column address into the DRAM.
- $t_4$ : The DRAM responds by placing valid data from the selected memory cell onto the DATA OUT line.
- $t_5$ : MUX,  $\overline{RAS}$ ,  $\overline{CAS}$ , and DATA OUT return to their initial states.

**FIGURE 12-29** Signal activity for a read operation on a dynamic RAM. The  $\overline{WE}$  input (not shown) is assumed to be HIGH.

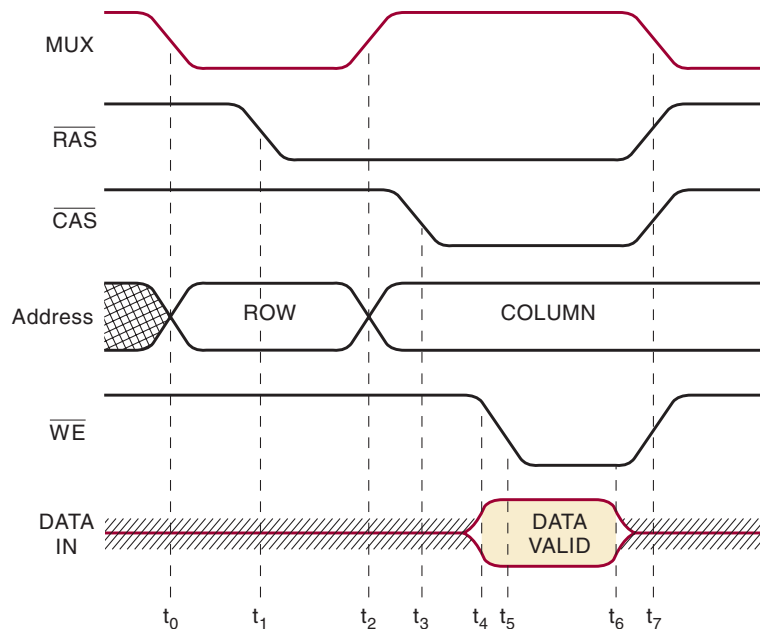


## DRAM Write Cycle

Figure 12-30 shows typical signal activity during a DRAM write operation. Here is a description of the sequence of events.

- $t_0$ : The LOW at  $MUX$  places the row address at the DRAM inputs.
- $t_1$ : The NGT at  $\overline{RAS}$  loads the row address into the DRAM.
- $t_2$ :  $MUX$  goes HIGH to place the column address at the DRAM inputs.
- $t_3$ : The NGT at  $\overline{CAS}$  loads the column address into the DRAM.
- $t_4$ : Data to be written are placed on the DATA IN line.
- $t_5$ :  $\overline{WE}$  is pulsed LOW to write the data into the selected cell.
- $t_6$ : Input data are removed from DATA IN.
- $t_7$ :  $MUX$ ,  $\overline{RAS}$ ,  $\overline{CAS}$ , and  $\overline{WE}$  are returned to their initial states.

**FIGURE 12-30** Signal activity for a write operation on a dynamic RAM.



### OUTCOME ASSESSMENT QUESTIONS

1. True or false:
  - (a) During a read cycle, the  $\overline{RAS}$  signal is activated before the  $\overline{CAS}$  signal.
  - (b) During a write operation,  $\overline{CAS}$  is activated before  $\overline{RAS}$ .
  - (c)  $\overline{WE}$  is held LOW for the entire write operation.
  - (d) The address inputs to a DRAM will change twice during a read or a write operation.
2. Which signal in Figure 12-27(b) makes sure that the correct portion of the complete address appears at the DRAM inputs?

## 12-16 DRAM REFRESHING

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe the process of refreshing cells in a DRAM.
- Describe the hardware that must surround a DRAM.

A DRAM cell is refreshed each time a read operation is performed on that cell. Each memory cell must be refreshed periodically (typically, every 2 to 8 ms, for devices of the capacity described here) or its data will be lost. This requirement would appear to be extremely difficult, if not impossible, to meet for large-capacity DRAMs. For example, a  $1\text{M} \times 1$  DRAM has  $10^{20} = 1,048,576$  cells, arranged as 1024 rows  $\times$  1024 columns. To ensure that each cell is refreshed within 4 ms, it would require that read operations be performed on successive addresses at the rate of one every 4 ns ( $4\text{ ms}/1,048,576 \approx 4\text{ ns}$ ). This is much too fast for any DRAM chip. Fortunately, manufacturers have designed DRAM chips so that

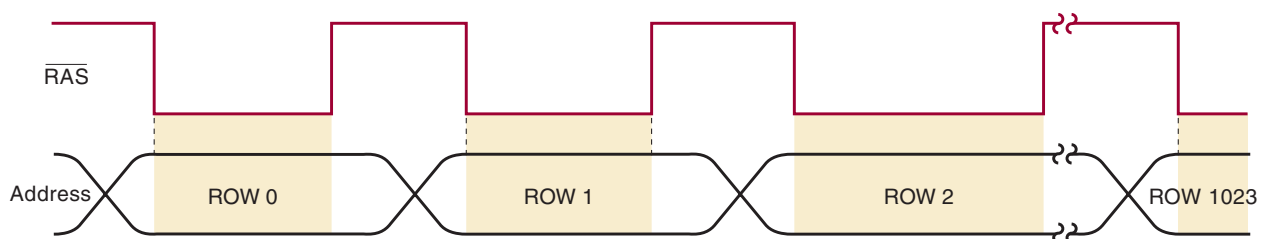
**whenever a read operation is performed on a cell, all of the cells in that row will be refreshed.**

Thus, it is necessary to do a read operation only on each *row* of a DRAM array once every 4 ms to guarantee that each *cell* of the array is refreshed. When any address is strobed into the row address register, all 1024 cells in that row are automatically refreshed.

Clearly, this row-refreshing feature makes it easier to keep all DRAM cells refreshed. However, during the normal operation of the system in which a DRAM is functioning, it is unlikely that a read operation will be performed on each row of the DRAM within the required refresh time limit. Therefore, some kind of refresh control logic is needed either external to the DRAM chip or as part of its internal circuitry. In either case, there are two refresh modes: a *burst* refresh and a *distributed* refresh.

In a burst refresh mode, the normal memory operation is suspended, and each row of the DRAM is refreshed in succession until all rows have been refreshed. With larger memories, this can take a relatively longer time, which slows down the system. In a distributed refresh mode, the row refreshing is interspersed with the normal operations of the memory. Most modern computer systems use this strategy.

The most universal method for refreshing a DRAM is the  **$\overline{\text{RAS}}$ -only refresh**. It is performed by strobing in a row address with  $\overline{\text{RAS}}$  while  $\overline{\text{CAS}}$  and  $\overline{\text{WE}}$  remain HIGH. Figure 12-31 illustrates how  $\overline{\text{RAS}}$ -only refresh is used for a burst refresh of a DRAM with 1024 rows. A **refresh counter** is used to supply 10-bit row addresses to the DRAM address inputs starting at 0000000000 (row 0).  $\overline{\text{RAS}}$  is pulsed LOW to load this address into the DRAM, and this refreshes row 0 in both banks. The counter is incremented and the process is repeated up to address 1111111111 (row 1023).



\*  $\overline{\text{R/W}}$  and  $\overline{\text{CAS}}$  lines held HIGH

**FIGURE 12-31** The  $\overline{\text{RAS}}$ -only refresh method uses only the  $\overline{\text{RAS}}$  signal to load the row address into the DRAM to refresh all cells in that row. The  $\overline{\text{RAS}}$ -only refresh can be used to perform a burst refresh as shown. A refresh counter supplies the sequential row addresses from row 0 to row 1023.

While the refresh counter idea seems easy enough, we must realize that the row addresses from the refresh counter cannot interfere with the addresses coming from the CPU during normal read/write operations. For this reason, the refresh counter addresses must be multiplexed with the CPU addresses, so that the proper source of DRAM addresses is activated at the proper times.

In order to relieve the computer's CPU from some of these burdens, a special chip called a **dynamic RAM (DRAM) controller** is often used. At a minimum, this chip will perform address multiplexing and refresh count sequence generation, leaving the generation of the timing for  $\overline{RAS}$ ,  $\overline{CAS}$ , and  $MUX$  signals up to some other logic circuitry and the person who programs the computer. Other DRAM controllers are fully automatic. Their inputs look very much like a static RAM or ROM. They automatically generate the refresh sequence often enough to maintain the memory, multiplex the address bus, generate the  $\overline{RAS}$  and  $\overline{CAS}$  signals, and arbitrate control of the DRAM between the CPU read/write cycles and local refresh operations. In current personal computers, the DRAM controller and other high-level controller circuits are integrated into a set of VLSI circuits that are referred to as a "chip set." As newer DRAM technologies are developed, new chip sets are designed to take advantage of the latest advances. In many cases, the number of existing (or anticipated) chip sets supporting a certain technology in the market determines the DRAM technology in which manufacturers invest.

Most of the DRAM chips in production today have on-chip refreshing capability that eliminates the need to supply external refresh addresses. One of these methods, for example, is called  *$\overline{CAS}$ -before- $\overline{RAS}$  refresh*. In this method, the  $\overline{CAS}$  signal is driven LOW first and is held LOW until after  $\overline{RAS}$  goes LOW. This sequence will refresh one row of the memory array and increment an internal counter that generates the row addresses. To perform a burst refresh using this feature,  $\overline{CAS}$  can be held LOW while  $\overline{RAS}$  is pulsed once for each row until all are refreshed. During this refresh cycle, all external addresses are ignored. Distributed refresh is easier with these devices because the built-in row counter keeps track of the next row that needs to be refreshed. The memory controller must only assure that enough of these row refreshes occur within the refresh interval.

### OUTCOME ASSESSMENT QUESTIONS

1. *True or false:*
  - (a) In most DRAMs, it is necessary to read only from one cell in each row in order to refresh all cells in that row.
  - (b) In the burst refresh mode, the entire array is refreshed by one  $\overline{RAS}$  pulse.
2. What is the function of a refresh counter?
3. What functions does a DRAM controller perform?
4. *True or false:*
  - (a) In the  $\overline{RAS}$ -only refresh method, the  $\overline{CAS}$  signal is held LOW.
  - (b)  $\overline{CAS}$ -before- $\overline{RAS}$  refresh can be used only by DRAMs with on-chip refresh control circuitry.

## 12-17 DRAM TECHNOLOGY\*

---

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Identify some common packaging for DRAM modules used in computers.
- Describe a few of the evolutionary changes in technology that have led to today's DRAM systems.

In selecting a particular type of RAM device for a system, a designer has some difficult decisions. The capacity (as large as possible), the speed (as fast as possible), the power needed (as little as possible), the cost (as low as possible), the footprint (as small as possible), and the convenience (as easy to change as possible) must all be kept in a reasonable balance because no single type of RAM can maximize all of these desired features. The semiconductor RAM market is constantly trying to produce the ideal mix of these characteristics in its products for various applications. This section explains some of the terms used regarding RAM technology.

### Memory Modules

With many companies manufacturing motherboards for personal computer systems, standard memory interface connectors have been adopted. These connectors receive a small printed circuit card with contact points on both sides of the edge of the card. These modular cards allow for easy installation or replacement of memory components in a computer. The single-in-line memory module (SIMM) is a circuit card with 72 functionally equivalent contacts on both sides of the card. A redundant contact point on each side of the board offers some assurance that a good, reliable contact is made. These modules use 5-V-only DRAM chips that vary in capacity from 1 to 16 Mbits in surface-mount gull-wing or J-lead packages. The memory modules vary in capacity from 1 to 32 Mbytes.

The newer, dual-in-line memory module (DIMM) has functionally unique contacts on each side of the card. DIMM cards range from 168 pins to 240 pins. The extra pins are necessary because DIMMs are connected to 64-bit data buses such as those found in modern PCs. Larger storage capacity requires more address lines as well. Both 3.3-V and 5-V versions are available. They also come in buffered and unbuffered versions. The capacity of the module depends on the DRAM chips that are mounted on it; and as DRAM capacity increases, the capacity of the DIMMs will increase. The chip set and motherboard design that is used in any given system determines which type of DIMM can be used. For compact applications, such as laptop computers, a small-outline, dual-in-line memory module is available (SODIMM).

The primary problem in the personal computer industry is providing a memory system that is fast enough to keep up with the ever-increasing clock speeds of the microprocessors while keeping the cost at an affordable level. Special features are being added to the basic DRAM devices to enhance their total bandwidth. Although these methods of improving

---

\*This topic may be omitted without affecting the continuity of the remainder of the book.

---

performance are constantly changing, the technologies described in the following sections are currently being referred to extensively in memory-related literature.

### FPM DRAM

Fast page mode (FPM) allows quicker access to random memory locations within the current “page.” A page is simply a range of memory addresses that have identical upper address bit values. In order to access data on the current page, only the lower address lines must be changed.

### EDO DRAM

Extended data output (EDO) DRAMs offer a minor improvement to FPM DRAMs. For accesses on a given page, the data value at the current memory location is sensed and latched onto the output pins. In the FPM DRAMs, the sense amplifier drives the output without a latch, requiring  $\overline{CAS}$  to remain low until data values become valid. With EDO, while these data are present on the outputs,  $\overline{CAS}$  can complete its cycle, a new address on the current page can be decoded, and the data path circuitry can be reset for the next access. This allows the memory controller to be outputting the next address at the same time that the current word is being read.

### SDRAM

Synchronous DRAM is designed to transfer data in rapid-fire *bursts* of several sequential memory locations. The first location to be accessed is the slowest due to the overhead (latency) of latching the row and column address. Thereafter, the data values are clocked out by the bus system clock (instead of the  $\overline{CAS}$  control line) in bursts of memory locations within the same page. Internally, SDRAMs are organized in two (or more) banks. This allows data to be read out at a very fast rate by alternately accessing each of the two banks. In order to provide all of the features and the flexibility needed for this type of DRAM to work with a wide variety of system requirements, the circuitry within the SDRAM has become more complex. A command sequence is necessary to tell the SDRAM which options are needed, such as burst length, sequential or interleaved data, and  $\overline{CAS}$ -before- $\overline{RAS}$  or self-refresh modes. Self-refresh mode allows the memory device to perform all of the necessary functions to keep its cells refreshed.

### DDRSDRAM

Double Data Rate SDRAM is a memory interface specification that is often referred to in computer literature. This designation refers to the memory module’s interface to the PC bus. DDR uses synchronous DRAM technology but achieves higher data rates to the system by transferring data on the rising and falling edge of the system clock. DDR essentially achieves burst transfer rates that are many times faster than older DRAMs. DDR technology has gone through four generations of improvements, with DDR4 the latest specification. Each generation has found ways of improving the speed of a computer’s operation.

---

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. Are SIMMs and DIMMs interchangeable?
2. What is a “page” of memory?
3. Why is “page mode” faster?
4. What does *EDO* stand for?
5. What term is used for accessing several consecutive memory locations?
6. How does DDR double the data rate?
7. What was the main improvement in moving from DDR to DDR2 to DDR3 to DDR4?

## 12-18 OTHER MEMORY TECHNOLOGIES

### OUTCOME

*Upon completion of this section, you will be able to:*

- Describe methods of creating memory that are not based on silicon IC technology.

The methods of storing information we have discussed so far involve fusible links, floating gate MOSFETs, capacitors, and flip-flops (latch) circuits. Other methods of storing data are in widespread use, and new methods are constantly being researched.

### Magnetic Storage

The technology of magnetic storage of digital information dates back to the first computer systems. The first method involved the use of reels of magnetic tape for long-term storage and retrieval of computer programs and data files: a technology adapted from the audio recording industry. The next improvement involved coating rigid (hard) disks with magnetic media and rotating the disks while moving a magnetic read/write head radially across the disk. This offered quicker, more random access of data at any location on the disk surface.

The original hard disk drives (from the 1950s) were built this same way, but were as big as a washing machine, had  $\frac{1}{2}$  horsepower spindle drive motors, and stored about 5 Megabytes, a very small amount of data relative to today's standards. Portable media in the form of “floppy disks” and “diskettes” came next, using essentially the same technology as hard disks, and were phased out as USB flash drives became the predominant portable storage medium. Magnetic disk technology has improved primarily in the density of storing (and reading) 1s and 0s and is now very close to the physical limits of the size of the individual magnetic domain. Many problems have been overcome that are common to handling such a huge volume of data. For example, bit error detection and correction methods used in hard drives can repair even multiple errors in a packet of data. Mechanical reliability has also been greatly improved, while the size of the machine has been reduced.

Data was stored originally using frequency modulation: 1s and 0s were represented by two different audio frequencies. Modern hard drives do not use audio tones. Instead they polarize the magnetic domains of the medium with one polarity for a 1 and the opposite polarity for a 0. The key information that is read from these drives is the transition from 0 to 1 or 1 to 0,



rather than reading each data bit itself. In order to make this work, the data must be encoded such that the number of consecutive 1s or 0s is of limited length. This is called Run Length Limited (RLL) encoding. The RLL scheme has greatly increased the density of data storage.

**MRAM** High-speed, random-access, nonvolatile magnetic storage of data was also attempted in the early days of computers using “magnetic core” technology. This involved rows and columns of little electromagnets that could be polarized in either direction. Due to size, cost, and power demands, this technology was quickly replaced by semiconductor memories. Amazingly enough, this basic technology has been brought back recently in the form of **magnetoresistive random access memory (MRAM)**. Recall the same grid of rows and columns of storage cells like we have studied in semiconductor memories. Instead of having a transistor circuit at each intersection of a row and column, imagine a magnetic nanoparticle that is polarized (spinning) in one of two possible ways. When a row is accessed and current flows through the column line, a magnetic field is created that is either the same or opposite in polarity to that of the magnetic storage cell. The interaction of these two polarities affects the resistance to the flow of current in the wire. The stored bit’s value (1 or 0) is detected based on the resistance (amount of current flow) through the column line. Data is written by altering the polarity or spin of the magnetic bit location. These devices are commercially available now but are still quite expensive. There is hope that as soon as economics allow MRAM to be mass-produced, it could become the ideal memory technology and replace mechanical hard drives, Flash, and DRAM. The fact that it is nonvolatile and yet has a fast read/write time gives it an advantage over DRAM. Flash memory offers a limited number of write cycles and MRAM offers unlimited writes. Flash write cycles are also much slower than MRAM. Hard drives are very slow and have moving parts that wear out.

## Optical Memory

The optical disc is a very significant digital storage memory technology. Digital audio compact discs (CDs) became available in the early 1980s, and the technology has spread to meet the needs of computer data storage, digital video (DVD), and most recently Blu-Ray Discs (BD). All of these optical storage formats use essentially the same technology, differing primarily in the format and density of information that can be stored/retrieved on a disc. The discs are manufactured with a highly reflective surface. To store data on the discs, a very intense laser beam is focused on a very small point in the disc. This beam alters the light diffracting properties of the surface such that it cannot reflect light as well. Digital data (1s and 0s) are stored on the disc one at a time by firing the laser on and off as the disc rotates. The information is arranged on the disc as a continuous spiral of data points that start at the center of the disc and progress toward the outer perimeter. The precision of the laser beam allows large quantities of data (up to 700 Mbytes for a CD) to be stored on a disc.

In order to read the data, a much less powerful laser beam is focused onto the surface of the disc and the reflected light is measured. At any point the reflected light is sensed as either a 1 or a 0. This optical system is mounted on a mechanical carriage and moves back and forth along the radius of the disc, following the spiral pattern of data as the disc rotates. The data retrieved from the optical system come one bit at a time in a serial data stream. Controlling the angular velocity of the disc as the radius of the spiral changes maintains a constant rate of incoming data. The data stream is decoded and grouped into data words.

Writeable CDs and DVDs allow us to store large amounts of data to backup files from a hard drive, create movie footage, and share digital pictures conveniently on very inexpensive media. The CDR discs have a coating that permanently alters its properties when the laser hits it. The CD RW discs can have previous data overwritten. This is accomplished using the laser to apply two different heat treatments to the special layer that can change its reflective/refractive characteristics back and forth between a 1 and 0.

Blu-Ray disc technology uses a shorter wavelength laser to produce a finer beam and higher bit densities than are possible using the red spectrum laser of the CD and DVD formats. Blu-Ray technology allows full length high-definition movies to be marketed on a single disc by storing up to 25 Gbytes per side.

### Phase Change Ram (PRAM)

This technology uses a material that has the ability to change phase easily. In other words, if the material was water, we could change it back and forth between liquid and steam easily. The two states of the phase change material are crystalline (ordered state), which has a very low resistance, and amorphous (disordered state), which has a very high resistance. The material becomes crystalline after a low current of relatively long pulse is applied. It changes to amorphous after a high current of relatively short pulse is applied. Different materials and methods of affecting a phase change are being researched. When the problems are overcome, many believe this technology could replace FLASH and DRAM.

### Ferroelectric RAM (FRAM)

This technology is very similar to DRAM in that it stores data in the form of an electric charge. Instead of using a conventional capacitor and storing the charge on the dielectric, it uses a ferroelectric material. The charge does not leak away like a traditional capacitor in DRAM so there is no need to refresh and it is nonvolatile.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the most common magnetic storage device in use today?
2. What type solid-state memory technology shows promise of replacing many existing technology as a “universal memory”?
3. What is a major advantage of using CDs and DVDs to store digital information?

## 12-19 EXPANDING WORD SIZE AND CAPACITY

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Build memory systems of a configuration using multiple smaller blocks.
- Explain the effect of incomplete address decoding on the memory map of a system.

In many memory applications, the required RAM or ROM memory capacity or word size cannot be satisfied by one memory chip. Instead, several memory chips must be combined to provide the capacity and/or the word

size. We will see how this is done through several examples that illustrate the important ideas that are used when memory chips are interfaced to a microprocessor. The examples that follow are intended to be instructive, and the memory chip sizes that are used were chosen to conserve space. The techniques that are presented can be extended to larger memory chips.

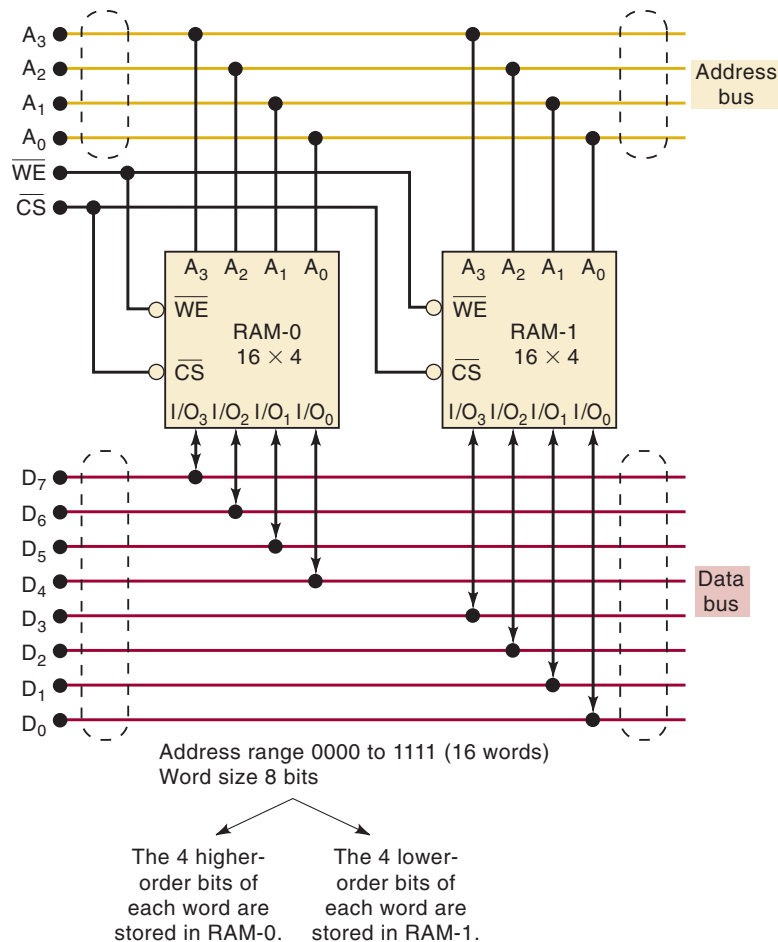
### Expanding Word Size

Suppose that we need a memory that can store 16 eight-bit words and all we have are RAM chips that are arranged as  $16 \times 4$  memories with common I/O lines. We can combine two of these  $16 \times 4$  chips to produce the desired memory. The configuration for doing so is shown in Figure 12-32. Examine this diagram carefully and see what you can find out from it before reading on.

Because each chip can store 16 four-bit words and we want to store 16 eight-bit words, we are using each chip to store *half* of each word. In other words, RAM-0 stores the four *higher-order* bits of each of the 16 words, and RAM-1 stores the four *lower-order* bits of each of the 16 words. A full eight-bit word is available at the RAM outputs connected to the data bus.

Any one of the 16 words is selected by applying the appropriate address code to the four-line *address bus* ( $A_3, A_2, A_1, A_0$ ). The address lines typically originate at the CPU. Note that each address bus line is connected to the corresponding address input of each chip. This means that once an address code is placed on the address bus, this same address code is applied to both chips so that the same location in each chip is accessed at the same time.

**FIGURE 12-32** Combining two  $16 \times 4$  RAMs for a  $16 \times 8$  module.



Once the address is selected, we can read or write at this address under control of the common  $\overline{WE}$  and  $\overline{CS}$  line. To read,  $\overline{WE}$  must be high and  $\overline{CS}$  must be low. This causes the RAM I/O lines to act as *outputs*. RAM-0 places its selected four-bit word on the upper four data bus lines, and RAM-1 places its selected four-bit word on the lower four data bus lines. The data bus then contains the full selected eight-bit word, which can now be transmitted to some other device (usually a register in the CPU).

To write,  $\overline{WE} = 0$  and  $\overline{CS} = 0$  causes the RAM I/O lines to act as *inputs*. The eight-bit word to be written is placed on the data bus (usually by the CPU). The higher four bits will be written into the selected location of RAM-0, and the lower four bits will be written into RAM-1.

In essence, the combination of the two RAM chips acts like a single  $16 \times 8$  memory chip. We refer to this combination as a  $16 \times 8$  *memory module*.

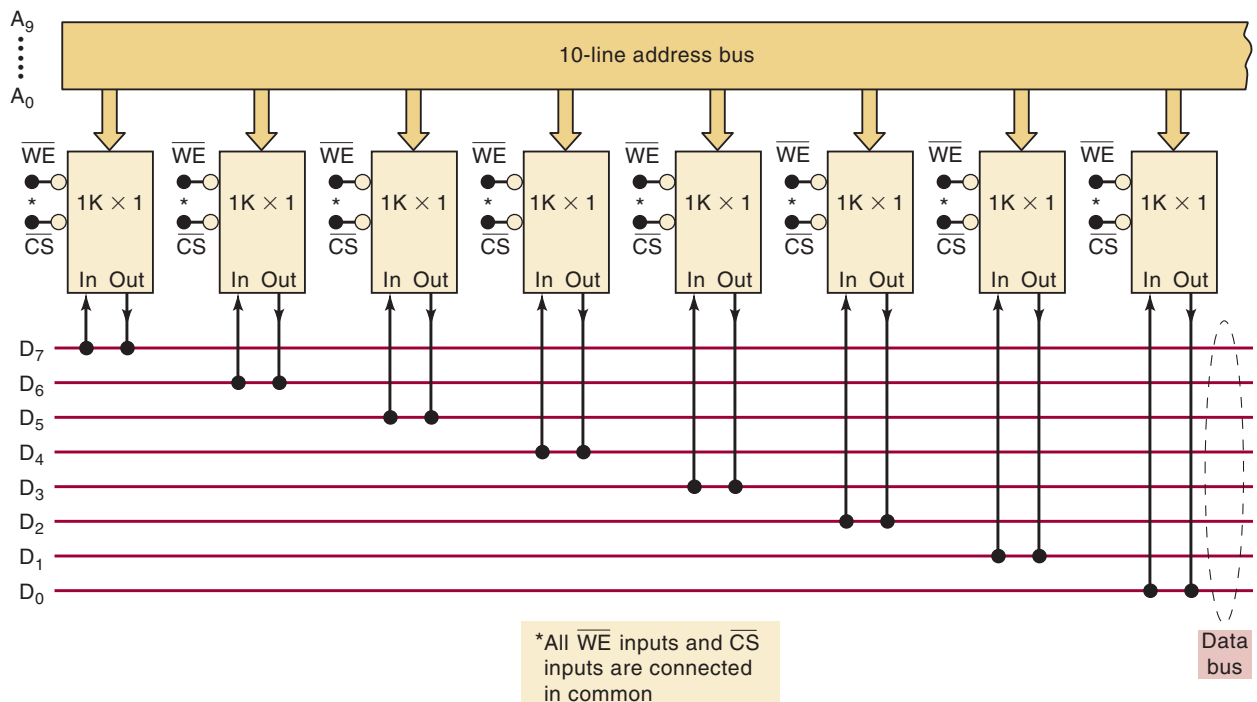
The same basic idea for expanding word size will work for many different situations. Read the following example and draw a rough diagram for what the system will look like before looking at the solution.

**EXAMPLE 12-9**

The 2125A is a static-RAM IC that has a capacity of  $1K \times 1$ , one active-LOW chip select input, and separate data input and output. Show how to combine several 2125A ICs to form a  $1K \times 8$  module.

**Solution**

The arrangement is shown in Figure 12-33, where eight 2125A chips are used for a  $1K \times 8$  module. Each chip stores one of the bits of each of the 1024 eight-bit words. Note that all of the  $\overline{WE}$  and  $\overline{CS}$  inputs are wired together, and the 10-line address bus is connected to the address inputs of each chip. Also note that because the 2125A has separate data in and data out pins, both of these pins of each chip are tied to the same data bus line.

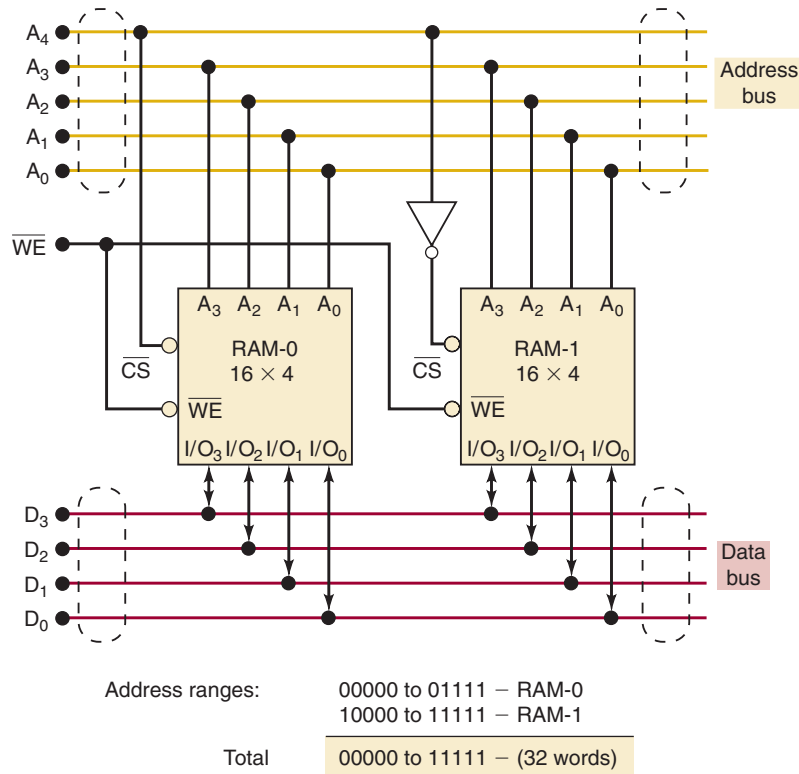


**FIGURE 12-33** Eight 2125A  $1K \times 1$  chips arranged as a  $1K \times 8$  memory.

## Expanding Capacity

Suppose that we need a memory that can store 32 four-bit words and all we have are the  $16 \times 4$  chips. By combining two  $16 \times 4$  chips as shown in Figure 12-34, we can produce the desired memory. Once again, examine this diagram and see what you can determine from it before reading on.

**FIGURE 12-34** Combining two  $16 \times 4$  chips for a  $32 \times 4$  memory.



Each RAM is used to store 16 four-bit words. The four data I/O pins of each RAM are connected to a common four-line data bus. Only one of the RAM chips can be selected (enabled) at one time so that there will be no bus-contention problems. This is ensured by driving the respective  $\overline{CS}$  inputs from different logic signals.

The total capacity of this memory module is  $32 \times 4$ , so there must be 32 different addresses. This requires *five* address bus lines. The upper address line  $A_4$  is used to select one RAM or the other (via the  $\overline{CS}$  inputs) as the one that will be read from or written into. The other four address lines  $A_0$  to  $A_3$  are used to select the one memory location out of 16 from the selected RAM chip.

To illustrate, when  $A_4 = 0$ , the  $\overline{CS}$  of RAM-0 enables this chip for read or write. Then any address location in RAM-0 can be accessed by  $A_3$  through  $A_0$ . The latter four address lines can range from 0000 to 1111 to select the desired location. Thus, the range of addresses representing locations in RAM-0 is

$$A_4A_3A_2A_1A_0 = 00000 \text{ to } 01111$$

Note that when  $A_4 = 0$ , the  $\overline{CS}$  of RAM-1 is high, so that its I/O lines are disabled (Hi-Z) and cannot communicate with (give data to or take data from) the data bus.

It should be clear that when  $A_4 = 1$ , the roles of RAM-0 and RAM-1 are reversed. RAM-1 is now enabled, and the lines  $A_3$  to  $A_0$  select one of its locations. Thus, the range of addresses located in RAM-1 is

$$A_4A_3A_2A_1A_0 = 10000 \text{ to } 11111$$

**EXAMPLE 12-10**

We want to combine several  $2K \times 8$  PROMs to produce a total capacity of  $8K \times 8$ . How many PROM chips are needed? How many address bus lines are required?

**Solution**

Four PROM chips are required, with each one storing 2K of the 8K words. Because  $8K = 8 \times 1024 = 8192 = 2^{13}$ , *thirteen* address lines are needed.

The configuration for the memory of Example 12-10 is similar to the  $32 \times 4$  memory of Figure 12-34. It is slightly more complex, however, because it requires a decoder circuit for generating the  $\overline{CS}$  input signals. The complete diagram for this  $8192 \times 8$  memory is shown in Figure 12-35(a).

The total capacity of the block of ROM is 8192 bytes. This system containing the block of memory has an address bus of 16 bits, which is typical of a small microcontroller-based system. The decoder in this system can only be enabled when  $A_{15}$  and  $A_{14}$  are LOW and  $E$  is HIGH. This means that it can only decode addresses less than 4000 hex. It is easier to understand this by looking at the memory map of Figure 12-35(b). You can see that the top two MSBs (in red) are always LOW for addresses under 4000 hex. Address lines  $A_{13}$ – $A_{11}$  (blue font) are connected to decoder inputs  $C$ – $A$ , respectively. These three bits are decoded and used to select one of the memory ICs. Notice in the bit map of Figure 12-35(b) that all the addresses that are contained in PROM-0 have  $A_{13}, A_{12}, A_{11} = 0, 0, 0$ ; PROM-1 is selected when these bits have a value of 0, 0, 1; PROM-2 when 0, 1, 0; and PROM-3 when 0, 1, 1. When any PROM is selected, the address lines  $A_{10}$ – $A_0$  can range from all 0s to all 1s. To summarize the address scheme of this system, the top two bits are used to select this decoder, the next three bits ( $A_{13}$ – $A_{11}$ ) are used to select one out of four PROM chips, and the lower 11 address lines are used to select one out of 2048 byte-sized memory locations in the enabled PROM.

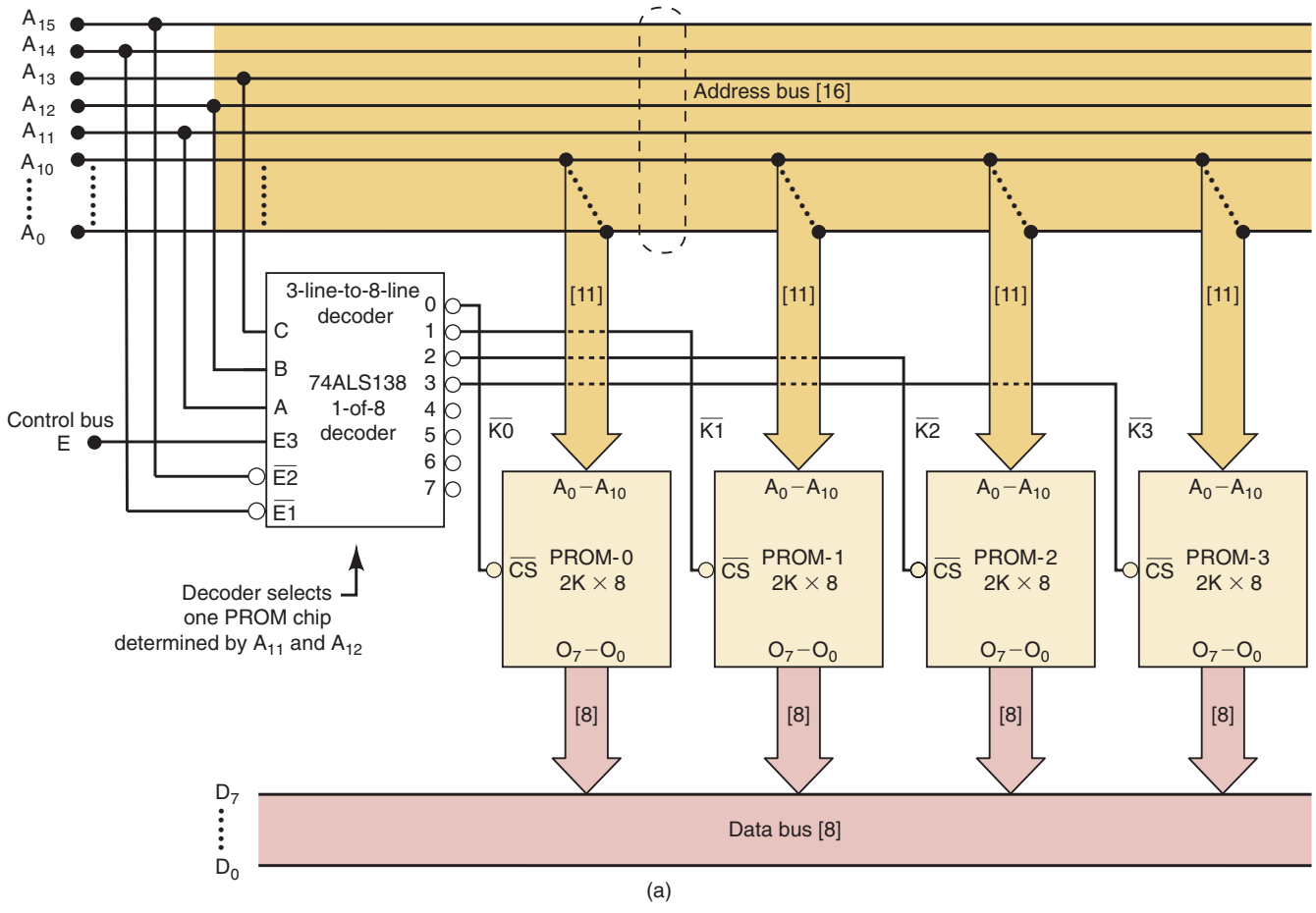
When the system address of 4000 or more is on the address bus, none of the PROMs will be enabled. However, decoder outputs 4–7 can be used to enable more memory chips if we wish to expand the capacity of the memory system. The memory map on the right side of Figure 12-35(b) shows a 48K area of the system's space that is not occupied by this memory block. In order to expand into this area of the memory map, more decoding logic would be needed.

**EXAMPLE 12-11**

What would be needed to expand the memory of Figure 12-37 to  $32K \times 8$ ? Describe what address lines are used.

**Solution**

A 32K capacity will require 16 of the 2K PROM chips. Four are already shown and four more can be connected to the  $O_4$ – $O_7$  of the existing decoder



A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	Address	System Map	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000	PROM-0	2K
0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	07FF		
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0800	PROM-1	2K
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0FFF		
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1000	PROM-2	2K
0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	17FF		
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1800	PROM-3	2K
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1FFF		
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000	O <sub>4</sub>	8K
		1	0	1													O <sub>5</sub>	
		1	1	0													O <sub>6</sub>	
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFF	O <sub>7</sub>	
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000	Available	48K
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFF		

(b)

**FIGURE 12-35** (a) Four 2K × 8 PROMs arranged to form a total capacity of 8K × 8; (b) memory map of the full system.

outputs. This accounts for half of the system. The other eight PROM chips can be selected by adding another 74ALS138 decoder and enabling it only when  $A_{15} = 0$  and  $A_{14} = 1$ . This can be accomplished by connecting an inverter between  $A_{14}$  and  $\overline{E}_1$  while connecting  $A_{15}$  directly to  $\overline{E}_2$ . The other connections are the same as in the existing decoder.

### Incomplete Address Decoding

In many instances, it is necessary to use various memory devices in the same memory system. For example, consider the requirements of a digital dashboard system on an automobile. Such a system is typically implemented using a microprocessor. Consequently, we need some nonvolatile ROM to store the program instructions. We need some read/write memory to store the digits that represent the speed, RPM, gallons of fuel, and so on. Other digitized values must be stored to represent oil pressure, engine temperature, battery voltage, and so on. We also need some nonvolatile read/write storage (EEPROM) for the odometer readout because it would not be good to have this number reset to 0 or assume a random value whenever the car battery is disconnected.

Figure 12-36 shows a memory system that could be used in a micro-computer system. Notice that the ROM portion is made up of two  $8K \times 8$  devices (PROM-0 and PROM-1). The RAM section requires a single

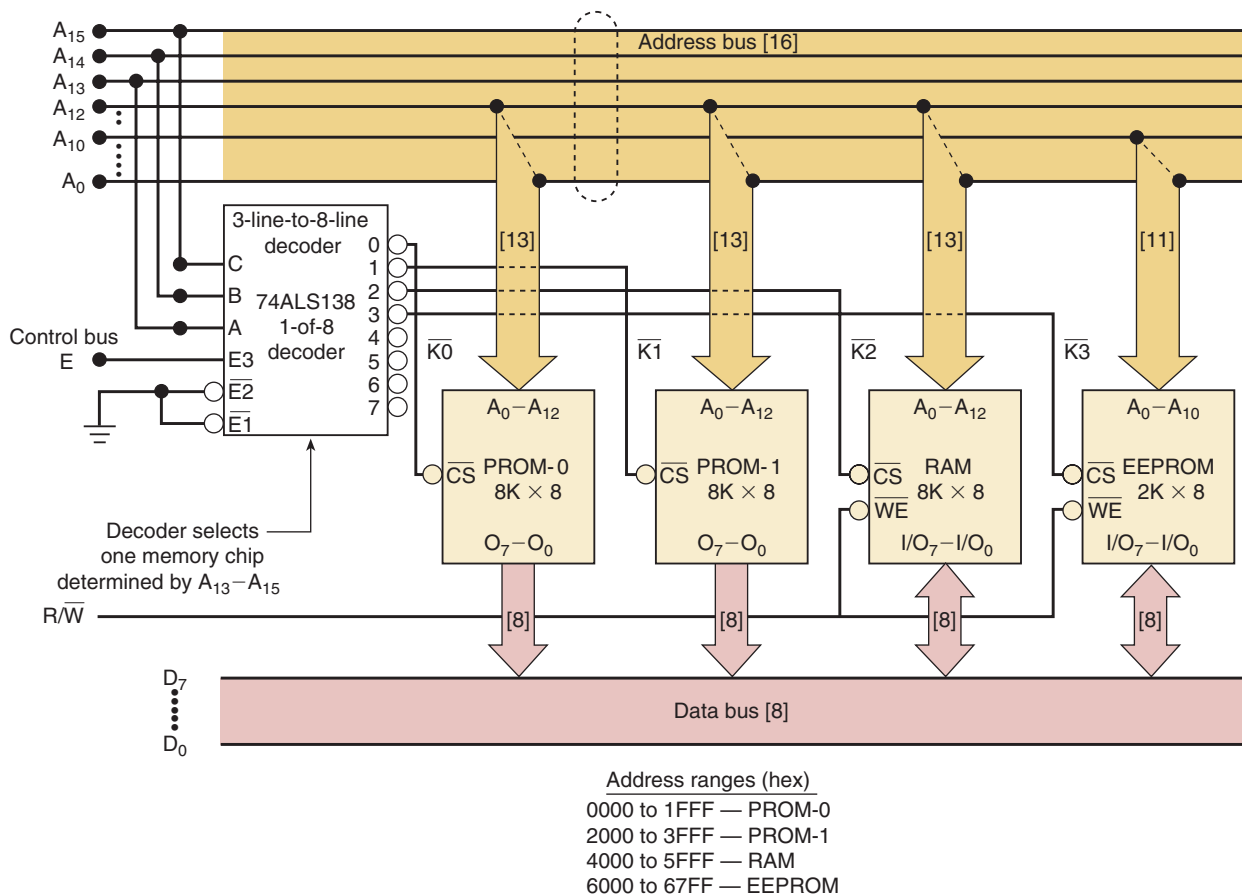
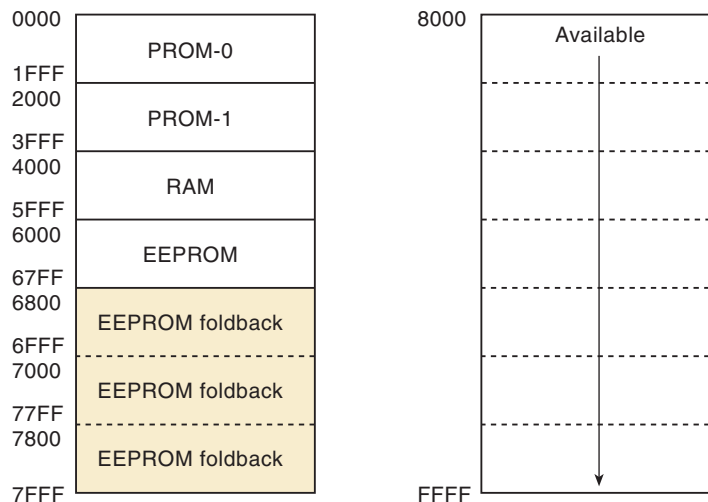


FIGURE 12-36 A system with incomplete address decoding.



8K × 8 device. The EEPROM available is only a 2K × 8 device. The memory system requires a decoder to select only one device at a time. This decoder divides the entire memory space (assuming 16 address bits) into 8K address blocks. In other words, each decoder output is activated by 8192 (8K) different addresses. Notice that the upper three address lines control the decoder. The 13 lower-order address lines are tied directly to the address inputs on the memory chips. The only exception to this is the EEPROM, which has only 11 address lines for its 2-Kbyte capacity. If the address (in hex) of this EEPROM is intended to range from 6000 to 67FF, it will respond to these addresses as intended. However, the two address lines,  $A_{11}$  and  $A_{12}$ , are not involved in the decoding scheme for this chip. The decoder output ( $\overline{K3}$ ) is active for 8K addresses, but the chip that it is connected to contains only 2K locations. As a result, the EEPROM will also respond to the other 6K of addresses in this decoded block of memory. The same contents of the EEPROM will also appear at addresses 6800–6FFF, 7000–77FF, and 7800–7FFF. These areas of memory that are redundantly occupied by a device due to incomplete address decoding are referred to as **memory foldback** areas. This occurs frequently in systems where there is an abundance of address space and a need to minimize decoding logic. A **memory map** of this system (see Figure 12-37) clearly shows the addresses that each device is assigned to as well as the memory space that is available for expansion.

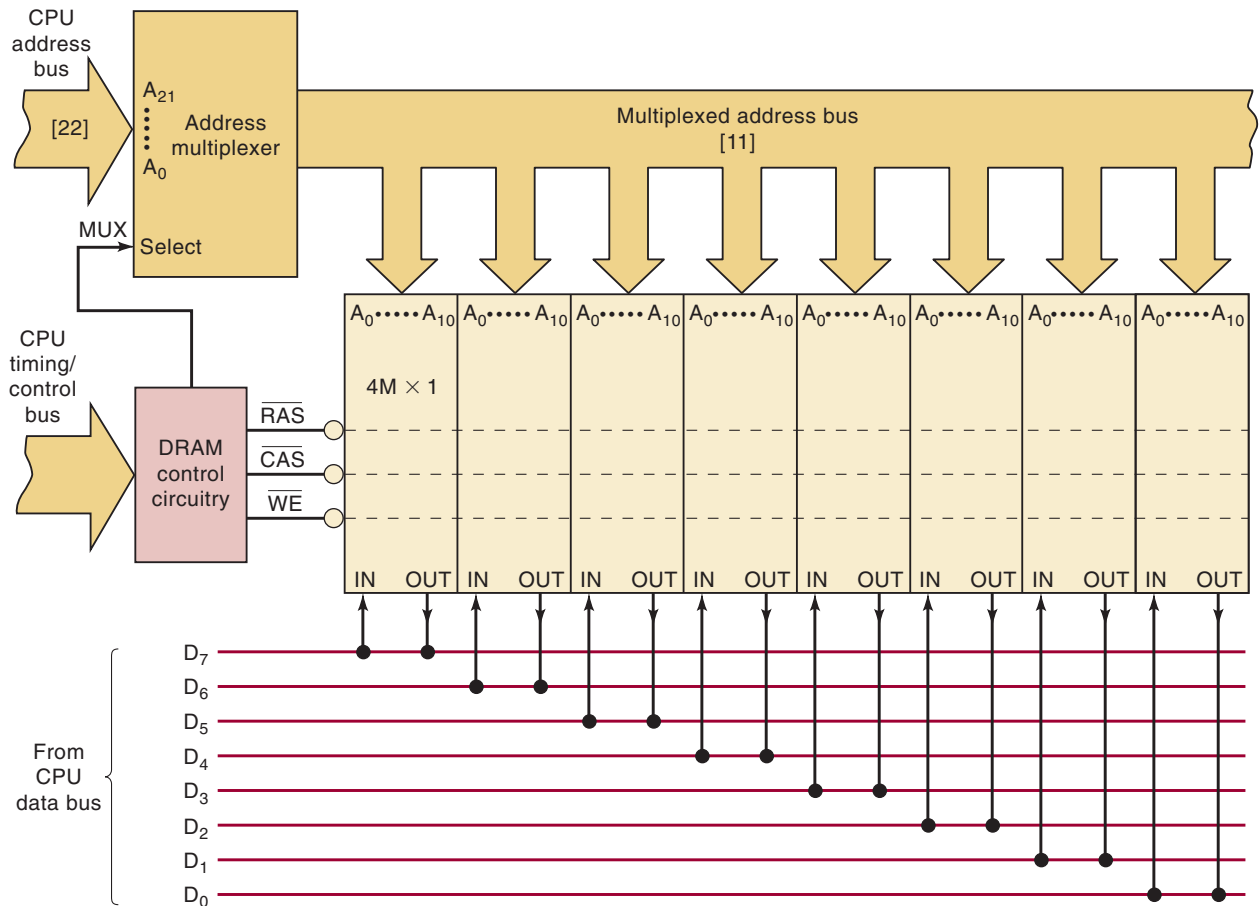
**FIGURE 12-37** A memory map of a digital dashboard system.



## Combining DRAM Chips

DRAM ICs often come with word sizes of one or four bits, so it is necessary to combine several of them to form larger word size modules. Figure 12-38 shows how to combine eight DRAM chips to form a 4M × 8 module. Each chip has a 4M × 1 capacity.

There are several important points to note. First, because  $4M = 2^{22}$ , the memory chip has *eleven* address inputs; remember, DRAMs use multiplexed address inputs. The address multiplexer takes the 22-line CPU address bus and changes it to an 11-line address bus for the DRAM chips. Second, the  $\overline{RAS}$ ,  $\overline{CAS}$ , and  $\overline{WE}$  inputs of all eight chips are connected together so that all chips are activated simultaneously for each memory operation. Finally, recall that many DRAM ICs have on-chip refresh control circuitry, so there is no need for an external refresh counter.



**FIGURE 12-38** Eight  $4\text{M} \times 1$  DRAM chips combined to form a  $4\text{M} \times 8$  memory module.

### OUTCOME ASSESSMENT QUESTIONS

1. The MCM6209C is a  $64\text{K} \times 4$  static-RAM chip. How many of these chips are needed to form a  $1\text{M} \times 4$  module?
2. How many are needed for a  $64\text{K} \times 16$  module?
3. *True or false:* When memory chips are combined to form a module with a larger word size or capacity, the CS inputs of each chip are always connected together.
4. *True or false:* When memory chips are combined for a larger capacity, each chip is connected to the same data bus lines.
5. If two of the address lines are excluded from the address decoding scheme of a memory chip, how many blocks of memory will the same chip occupy?

## 12-20 SPECIAL MEMORY FUNCTIONS

### OUTCOMES

Upon completion of this section, you will be able to:

- Define some common types of memory systems.
- Explain the purpose of each system.

We have seen that RAM and ROM devices are used as a computer's high-speed internal memory that communicates directly with the CPU (e.g., microprocessor). In this section, we briefly describe some of the special functions that semiconductor memory devices perform in computers and other digital equipment and systems. The discussion is not intended to provide details of how these functions are implemented but to introduce the basic ideas.

## Cache Memory

In order to understand the role of cache memory, let's review some facts about computers.

- Hard drives hold lots of instructions for the computer, but they are very slow.
- The instructions must travel over wires (data bus) for quite a distance. This limits the speed at which the data lines can change without distortion and errors.
- When you click on an application, a portion of code for that application (which may be many Megabytes) is loaded from the hard drive to the working memory (DRAM).
- The CPU of most computers can operate at very high speeds (over 2 GHz).
- DRAM is much faster than a hard drive, but much slower than the CPU. The data must travel over wires from DRAM to the CPU. Therefore, the bus clock rate is much slower than the CPU clock rate.

The problem that exists in computers is that the CPU can handle instructions much faster than they can be fetched from DRAM. In order to take advantage of the high speed of the CPU, the memory that it fetches from must be able to supply instructions at the same speed. This means the physical memory circuitry must be fast and it must be close to the CPU. Since it is not reasonable to put all the working memory into the CPU chip, computer architects do the next best thing. They build a small cache of memory (Kbytes) of very fast SRAM into the CPU core. This cache holds the instructions that the CPU is likely to need in the very near future. Being closest to the CPU core, it is called the level 1 (L1) cache. The contents of this cache can be accessed very quickly by the CPU.

Many CPUs today have multiple CPU cores on the same IC. Each of these cores has its own L1 cache. They share a common bus interface that is also integrated onto the same IC. Associated with the common bus interface unit is another block of memory known as the L2 cache. It may have a capacity of several Megabytes and, being on the same IC, can supply any of the L1 caches quickly if needed. The L2 cache may be supplied with instructions from the DRAM on the mother board, or the system may have an L3 cache on the motherboard (between the L2 cache and DRAM) that is faster than the DRAM.

The CPU gets instructions from the L1 cache at very high speeds. When the CPU needs an instruction that is not contained in the L1 cache (this is called a cache miss), it goes to the L2 cache to look for it. This takes a little longer. If it cannot find what it needs in the L2 cache, it will need to go to the L3 cache or even the system DRAM to reload the caches. This takes a lot longer because the system bus operates on a much slower clock due to the latency of the DRAM and the distances the data must travel.

---

Think of this process as very similar to the way a cook operates in the kitchen. The anticipated ingredients are near at hand. If something else is needed, the cook goes to the pantry where local inventory is stored. If it is not there, it must be ordered from an outside supplier, and so on. Inventories are maintained at each level to increase efficiency while managing cost. These are essentially the same reasons we use cache memory in a computer.

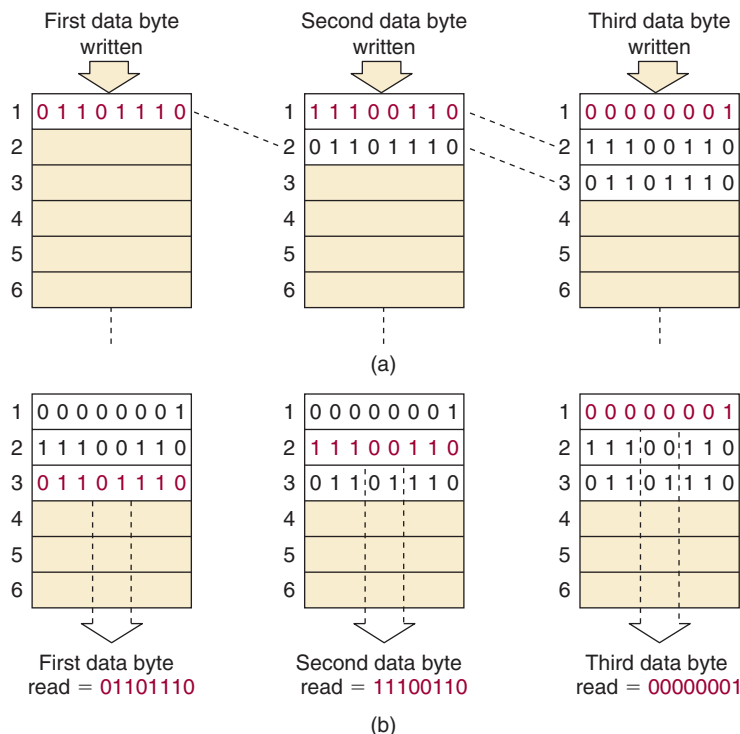
### First-In, First-Out Memory (FIFO)

In **FIFO** memory systems, data that are written into the RAM storage area are read out in the same order that they were written in. In other words, the first word written into the memory block is the first word that is read out of the memory block: hence the name FIFO. This idea is illustrated in Figure 12-39.

Figure 12-39(a) shows the sequence of writing three data bytes into the memory block. Note that as each new byte is written into location 1, the other bytes move to the next location. Figure 12-39(b) shows the sequence of reading the data out of the FIFO block. The first byte read is the same as the first byte that was written, and so on. The FIFO operation is controlled by special *address pointer registers* that keep track of where data are to be written and the location from which they are to be read.

A FIFO is useful as a **data-rate buffer** between systems that transfer data at widely different rates. One example is the transfer of data from a computer to a printer. The computer sends character data to the printer at a very high rate, say, one byte every 10  $\mu\text{s}$ . These data fill up a FIFO memory in the printer. The printer then reads out the data from the FIFO at a much slower rate, say, one byte every 5 ms, and prints out the corresponding characters in the same order as sent by the computer.

**FIGURE 12-39** In FIFO, data values are read out of memory (b) in the same order that they were written into memory (a).



A FIFO can also be used as a data-rate buffer between a slow device, such as a keyboard, and a high-speed computer. Here, the FIFO accepts keyboard data at the slow and asynchronous rate of human fingers and stores them. The computer can then read all of the recently stored keystrokes very quickly at a convenient point in its program. In this way, the computer can perform other tasks while the FIFO is slowly being filled with data.

### Circular Buffers

Data rate buffers (FIFOs) are often referred to as **linear buffers**. As soon as all the locations in the buffer are full, no more entries are made until the buffer is emptied. This way, none of the “old” information is lost. A similar memory system is called a **circular buffer**. These memory systems are used to store the last  $n$  values entered, where  $n$  is the number of memory locations in the buffer. Each time a new value is written to a circular buffer, it overwrites (replaces) the oldest value. Circular buffers are addressed by a MOD- $n$  address counter. Consequently, when the highest address is reached, the address counter will “wrap around” and the next location will be the lowest address. As you recall from Chapter 11, digital filtering and other DSP operations perform calculations using a group of recent samples. Special hardware included in a DSP allows easy implementation of circular buffers in memory.

#### OUTCOME ASSESSMENT QUESTIONS

1. What is the principal reason for using a cache memory?
2. What does *FIFO* mean?
3. What is a data-rate buffer?
4. How does a circular buffer differ from a linear buffer?

### SUMMARY

1. All memory devices store binary logic levels (1s and 0s) in an array structure. The size of each binary word (number of bits) that is stored varies depending on the memory device. These binary values are referred to as *data*.
2. The place (location) in the memory device where any data value is stored is identified by another binary number referred to as an *address*. Each memory location has a unique address.
3. All memory devices operate in the same general way. To write data in memory, the address to be accessed is placed on the address input, the data value to be stored is applied to the data inputs, and the control signals are manipulated to store the data. To read data from memory, the address is applied, the control signals are manipulated, and the data value appears on the output pins.
4. Memory devices are often used along with a microprocessor CPU that generates the addresses and control signals and either provides the data to be stored or uses the data from the memory. Reading and writing

are *always* done from the CPU’s perspective. Writing puts data into the memory, and reading gets data out of the memory.

5. Most read-only memories (ROMs) have data entered one time, and from then on their contents do not change. This storage process is called *programming*. They do not lose their data when power is removed from the device. MROMs are programmed during the manufacturing process. PROMs are programmed one time by the user. EPROMs are just like PROMs but can be erased using UV light. EEPROMs and flash memory devices are electrically erasable and can have their contents altered after programming.
6. Random access memory (RAM) is a generic term given to devices that can have data easily stored and retrieved. Data are retained in a RAM device only as long as power is applied.
7. Static RAM (SRAM) uses storage elements that are basically latch circuits. Once the data are stored, they will remain unchanged as long as power is applied to the chip. Static RAM is easier to use but more expensive per bit and consumes more power than dynamic RAM.
8. Dynamic RAM (DRAM) uses capacitors to store data by charging or discharging them. The simplicity of the storage cell allows DRAMs to store a great deal of data. Because the charge on the capacitors must be refreshed regularly, DRAMs are more complicated to use than SRAMs. Extra circuitry is often added to DRAM systems to control the reading, writing, and refreshing cycles. On many new devices, these features are being integrated into the DRAM chip itself. The goal of DRAM technology is to put more bits on a smaller piece of silicon so that it consumes less power and responds faster.
9. MRAM stores data by polarizing a very small magnetic particle in one of two possible directions. When the cell is read, its polarity affects the resistance of the column line. The resistance is sensed as a 1 or a 0.
10. Memory systems require a wide variety of different configurations. Memory chips can be combined to implement any desired configuration, whether your system needs more bits per location or more total word capacity. All of the various types of ROM and RAM can be combined within the same memory system.

## IMPORTANT TERMS

---

main memory	read-only memory (ROM)	address multiplexing
auxiliary memory	cache memory	row address strobe
memory cell	address bus	column address strobe
memory word	data bus	latency
byte	control bus	$\overline{RAS}$ -only refresh
capacity	programming	refresh counter
density	chip select	DRAM controller
address	power-down	magnetoresistive RAM
read operation	fusible link	(MRAM)
write operation	electrically erasable	memory foldback
access time	PROM (EEPROM)	memory map
volatile memory	flash memory	FIFO
random-access memory	NOR flash	data-rate buffer
(RAM)	NAND flash	linear buffer
sequential-access memory	bootstrap program	circular buffer
(SAM)	static RAM	
read/write memory	dynamic RAM	
(RWM)	refresh	

---

## PROBLEMS

### SECTIONS 12-1 TO 12-3

- B** 12-1.\*A certain memory has a capacity of  $16\text{K} \times 32$ . How many words does it store? What is the number of bits per word? How many memory cells does it contain?
- B** 12-2. How many different addresses are required by the memory of Problem 12-1?
- B** 12-3.\*What is the capacity of a memory that has 16 address inputs, four data inputs, and four data outputs?
- B** 12-4. A certain memory stores  $8\text{K}$  16-bit words. How many data input and data output lines does it have? How many address lines does it have? What is its capacity in bytes?

### DRILL QUESTIONS

- 12-5. Define each of the following terms.
- B**
- (a) RAM
  - (b) RWM
  - (c) ROM
  - (d) Internal memory
  - (e) Auxiliary memory
  - (f) Capacity
  - (g) Volatile
  - (h) Density
  - (i) Read
  - (j) Write
- 12-6. (a) What are the three buses in a computer memory system?
- B**
- (b) Which bus is used by the CPU to select the memory location?
  - (c) Which bus is used to carry data from memory to the CPU during a read operation?
  - (d) What is the source of data on the data bus during a write operation?

### SECTIONS 12-4 AND 12-5

- 12-7.\*Refer to Figure 12-6. Determine the data outputs for each of the following input conditions.
- B**
- (a)  $[A] = 1011; \overline{CS} = 1, \overline{OE} = 0$
  - (b)  $[A] = 0111; \overline{CS} = 0, \overline{OE} = 0$
- B** 12-8. Refer to Figure 12-7.
- (a) What register is enabled by input address 1011?
  - (b) What input address code selects register 4?
- B** 12-9.\*A certain ROM has a capacity of  $16\text{K} \times 4$  and an internal structure like that shown in Figure 12-7.
- (a) How many registers are in the array?
  - (b) How many bits are there per register?
  - (c) What size decoders does it require?

\*Answers to problems marked with an asterisk can be found in the back of the text.

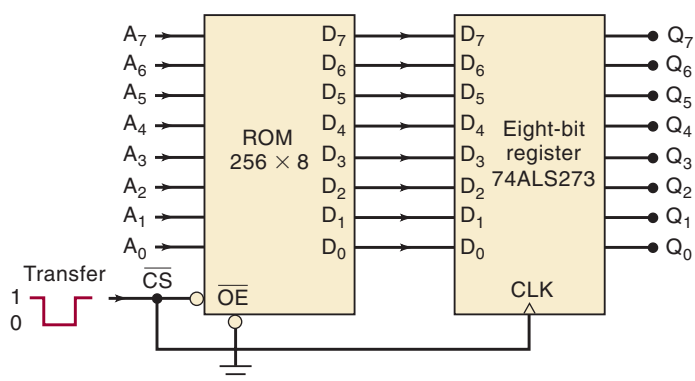
**DRILL QUESTION**

- B** 12-10. (a) *True or false:* ROMs cannot be erased.  
 (b) What is meant by *programming* or *burning* a ROM?  
 (c) Define a ROM's access time.  
 (d) How many data inputs, data outputs, and address inputs are needed for a  $1024 \times 4$  ROM?  
 (e) What is the function of the decoders on a ROM chip?

**SECTION 12-6**

- C, D** 12-11.\*Figure 12-40 shows how data from a ROM can be transferred to an external register. The ROM has the following timing parameters:  $t_{ACC} = 250$  ns and  $t_{OE} = 120$  ns. Assume that the new address inputs have been applied to the ROM 500 ns before the occurrence of the TRANSFER pulse. Determine the minimum duration of the TRANSFER pulse for reliable transfer of data.

**FIGURE 12-40** Problem 12-11.



- C, D** 12-12. Repeat Problem 12-11 if the address inputs are changed 70 ns prior to the TRANSFER pulse.

**SECTIONS 12-7 AND 12-8**

**B** 12-13. **DRILL QUESTION**

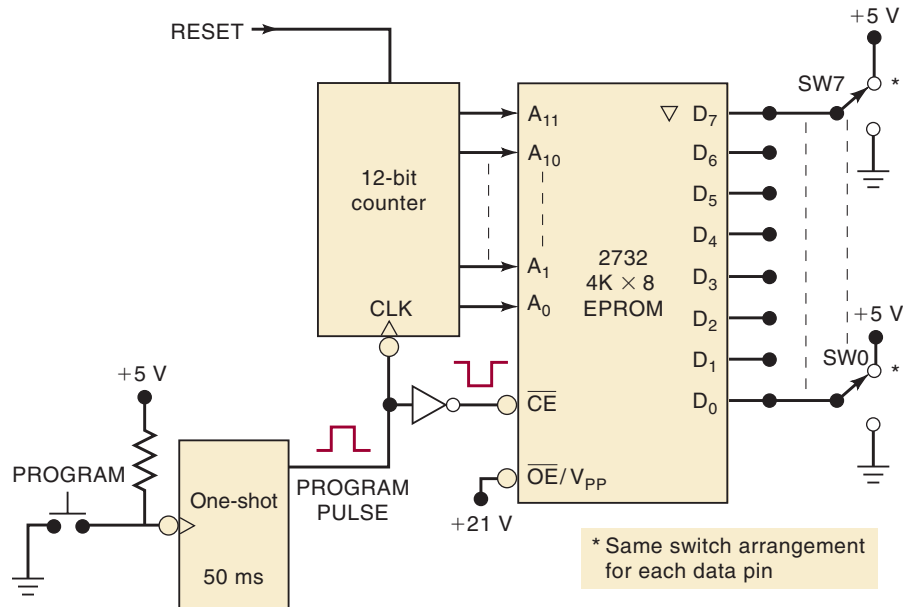
For each item below, indicate the type of memory being described: MROM, PROM, EPROM, EEPROM, flash. Some items will correspond to more than one memory type.

- Can be programmed by the user but cannot be erased.
- Is programmed by the manufacturer.
- Is volatile.
- Can be erased and reprogrammed over and over.
- Individual words can be erased and rewritten.
- Is erased with UV light.
- Is erased electrically.
- Uses fusible links.
- Can be erased in bulk or in sectors of 512 bytes.
- Does not have to be removed from the system to be erased and reprogrammed.
- Requires a special supply voltage for reprogramming.
- Erase time is about 15 to 20 min.



- B** 12-14. Which transistors in Figure 12-9 will be conducting when  $A_1 = A_0 = 1$  and  $\overline{EN} = 0$ ?
- 12-15.\*Change the MROM connections in Figure 12-9 so that the MROM stores the function  $y = 3x + 5$ .
- D** 12-16. Figure 12-41 shows a simple circuit for manually programming a 2732 EPROM. Each EPROM data pin is connected to a switch that can be set at a 1 or a 0 level. The address inputs are driven by a 12-bit counter. The 50-ms programming pulse comes from a one-shot each time the PROGRAM push button is actuated.
- Explain how this circuit can be used to program the EPROM memory locations sequentially with the desired data.
  - Show how 74293s and a 74121 can be used to implement this circuit.
  - Should switch bounce have any effect on the circuit operation?

**FIGURE 12-41** Problem 12-16.



- N** 12-17. Figure 12-42 shows a small flash memory chip connected to a CPU over a data bus and an address bus. The CPU can write to or read from the flash memory array by sending the desired memory address and generating the appropriate control signals to the chip. The CPU asserts the  $\overline{RD}$  line when it has finished outputting a stable address and wants to read data from the memory device. The CPU asserts the  $\overline{WR}$  line after it has finished outputting a stable address and has placed the data to be stored on the data bus.
- What control logic is needed to allow this flash memory array to occupy addresses between  $8000_{16}$  and  $FFFF_{16}$ ?
  - Which bus will carry the command codes from the CPU to the flash memory chip?
  - What type of bus cycle will be executed to send control codes to the flash memory chip?

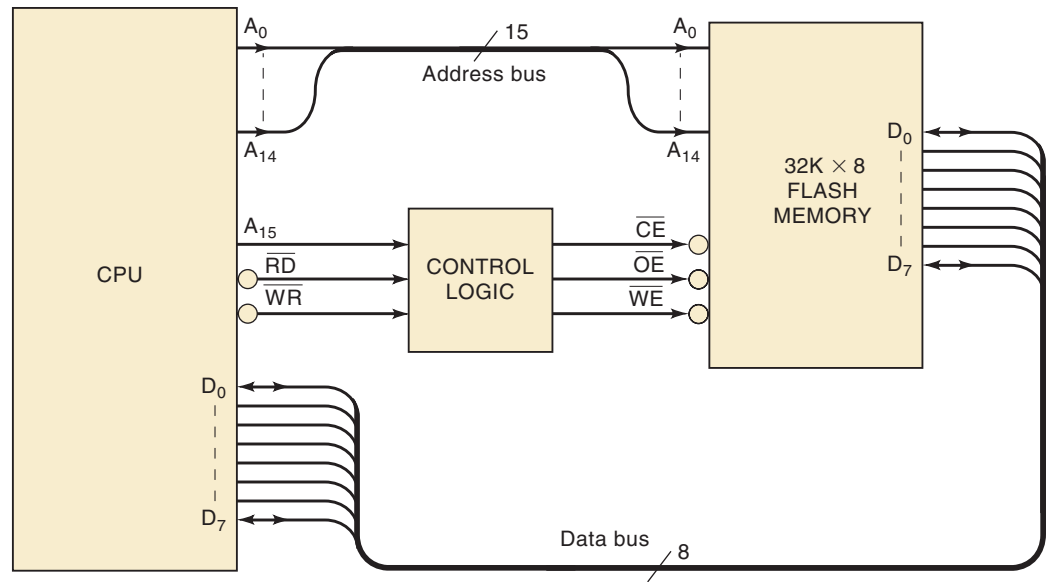
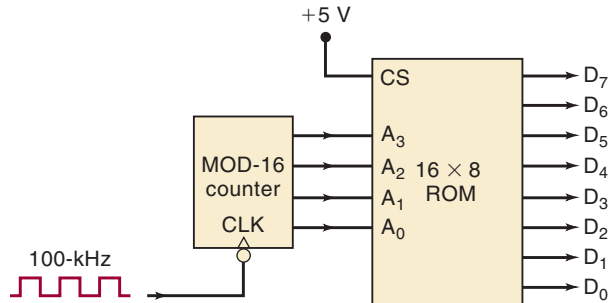


FIGURE 12-42 Problem 12-17.

SECTION 12-9

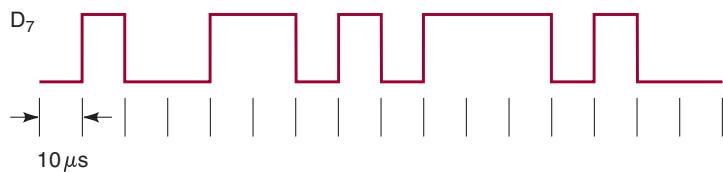
12-18. Another ROM application is the generation of timing and control signals. Figure 12-43 shows a  $16 \times 8$  ROM with its address inputs driven by a MOD-16 counter so that the ROM addresses are incremented with each input pulse. Assume that the ROM is programmed as in Figure 12-6, and sketch the waveforms at each ROM output as the pulses are applied. Ignore ROM delay times. Assume that the counter starts at 0000.

FIGURE 12-43 Problem 12-18.



D 12-19.\*Change the program stored in the ROM of Problem 12-18 to generate the  $D_7$  waveform of Figure 12-44

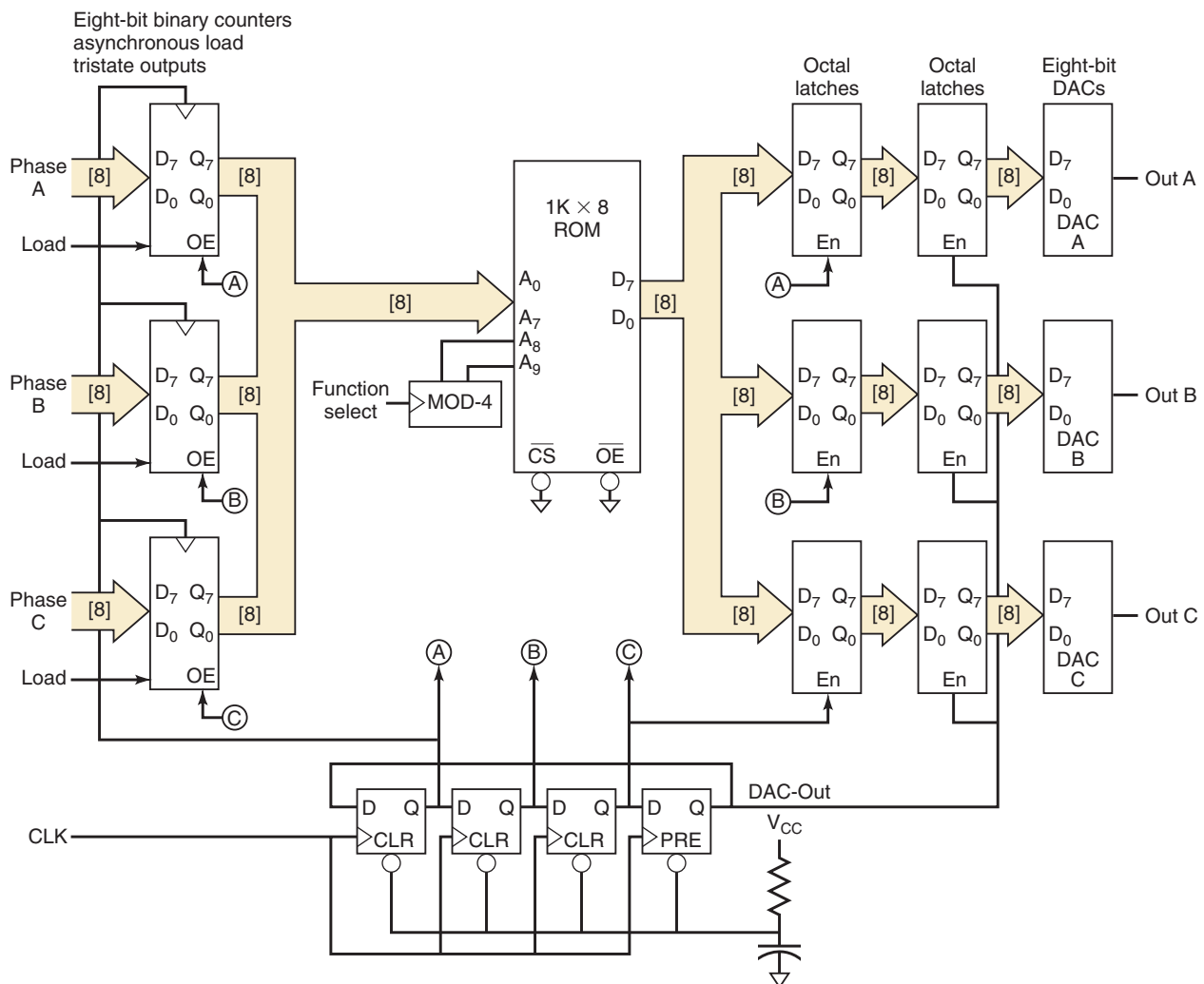
FIGURE 12-44 Problem 12-19.



D 12-20.\*Refer to the function generator of Figure 12-20.  
 (a) What clock frequency will result in a 100-Hz sine wave at the output?  
 (b) What method could be used to vary the peak-to-peak amplitude of the sine wave?

**N, C** 12-21. The system shown in Figure 12-45 is a waveform (function) generator. It uses four 256-point look-up tables in a 1-Kbyte ROM to store one cycle each of a sine wave (address 000–0FF), a positive slope ramp (address 100–1FF), a negative slope ramp (200–2FF), and a triangle wave (300–3FF). The phase relationship among the three output channels is controlled by the values initially loaded into the three counters. The critical timing parameters are  $t_{pd(ck-Q \text{ and } OE-Q \text{ max})}$ , counters = 10 ns, latches = 5 ns, and  $t_{ACC \text{ ROM}} = 20$  ns. Study the diagram until you understand how it operates and then answer the following:

- In Figure 12-45, the three latches on the left perform a function that represents one of the basic building blocks of digital systems as presented in Chapter 9. Which function is it?
- The four latches that are fed by the ROM serve to implement another basic building block of digital systems. Which function is it?
- Why are the octal latches that feed the DACs necessary?
- What number must be on the MOD 4 function select counter to produce each of the following waveforms: triangle; sine; negative ramp; positive ramp?



**FIGURE 12-45** Problems 12-21 and 12-22.

- C** 12-22.\*Refer to Problem 12-21.
- If counter A is initially loaded with 0, what values must be loaded into counters B and C so that A lags B by  $90^\circ$  and A lags C by  $180^\circ$ ?
  - If counter A is initially loaded with 0, what values must be loaded into counters B and C to generate a three-phase sine wave with  $120^\circ$  shift between each output?
  - What must the frequency of pulses on DAC\_OUT be in order to generate a 60-Hz sine wave output?
  - What is the maximum frequency of the CLK input?
  - What is the maximum frequency of the output waveforms?
  - What is the purpose of the function select counter?

### SECTION 12-11

- 12-23. (a) Draw the logic symbol for an MCM101514, a CMOS static RAM organized as a  $256\text{K} \times 4$  with separate data in and data out, and an active-LOW chip enable.
- (b) Draw the logic symbol for an MCM6249, a CMOS static RAM organized as a  $1\text{M} \times 4$  with common I/O, an active-LOW chip enable, and an active-LOW output enable.

### SECTION 12-12

- 12-24.\*A certain static RAM has the following timing parameters (in nanoseconds):

$$\begin{array}{ll}
 t_{\text{RC}} = 100 & t_{\text{AS}} = 20 \\
 t_{\text{ACC}} = 100 & t_{\text{AH}} = \text{not given} \\
 t_{\text{CO}} = 70 & t_{\text{W}} = 40 \\
 t_{\text{OD}} = 30 & t_{\text{DS}} = 10 \\
 t_{\text{WC}} = 100 & t_{\text{DH}} = 20
 \end{array}$$

- How long after the address lines stabilize will valid data appear at the outputs during a read cycle?
- How long will output data remain valid after  $\overline{\text{CS}}$  returns HIGH?
- How many read operations can be performed per second?
- How long should  $\overline{\text{WE}}$  and  $\overline{\text{CS}}$  be kept HIGH after the new address stabilizes during a write cycle?
- What is the minimum time that input data must remain valid for a reliable write operation to occur?
- How long must the address inputs remain stable after  $\overline{\text{WE}}$  and  $\overline{\text{CS}}$  return HIGH?
- How many write operations can be performed per second?

### SECTIONS 12-13 TO 12-17

- 12-25. Draw the logic symbol for the TMS4256, which is a  $256\text{K} \times 1$  DRAM. How many pins are saved by using address multiplexing for this DRAM?
- D** 12-26. Figure 12-46(a) shows a circuit that generates the  $\overline{\text{RAS}}$ ,  $\overline{\text{CAS}}$ , and  $\text{MUX}$  signals needed for proper operation of the circuit of Figure 12-27(b). The 10-MHz master clock signal provides the basic timing for the computer. The memory request signal ( $\text{MEMR}$ ) is generated by the CPU in synchronism with the master clock, as shown in part (b) of the figure.

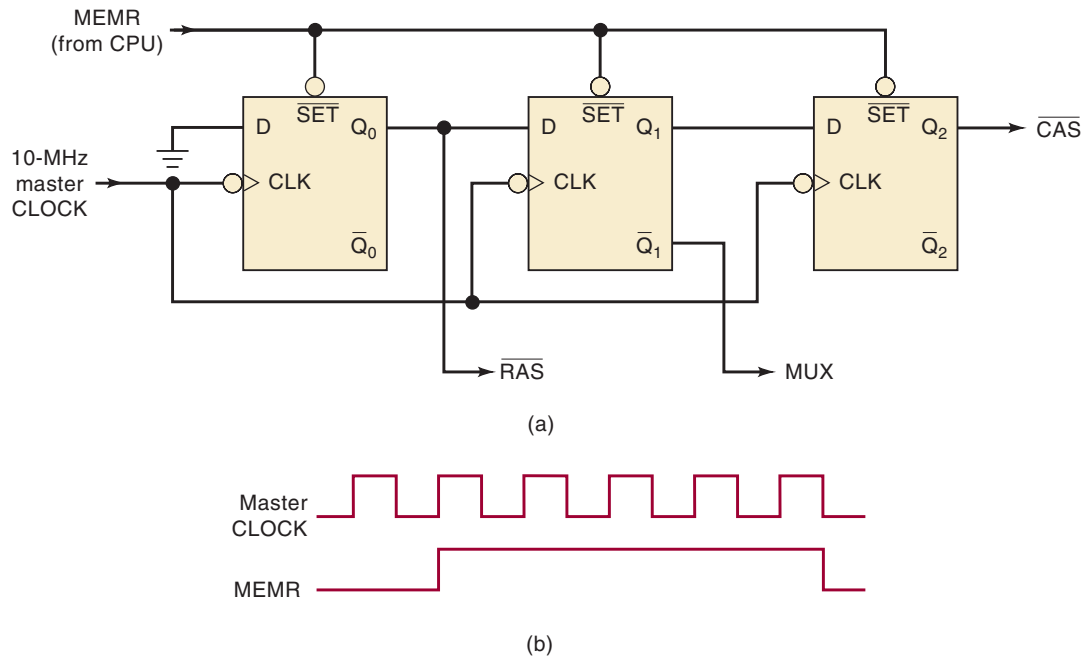


FIGURE 12-46 Problem 12-26.

*MEMR* is normally LOW and is driven HIGH whenever the CPU wants to access memory for a read or a write operation. Determine the waveforms at  $Q_0$ ,  $Q_1$ , and  $Q_2$ , and compare them with the desired waveforms of Figure 12-28.

- D** 12-27. Show how to connect two 74157 multiplexers to provide the multiplexing function required in Figure 12-27(b).
- 12-28. Refer to the signals in Figure 12-29. Describe what occurs at each of the labeled time points.
- 12-29. Repeat Problem 12-28 for Figure 12-30.
- C** 12-30.\*The 21256 is a  $256\text{K} \times 1$  DRAM that consists of a  $512 \times 512$  array of cells. The cells must be refreshed within 4 ms for data to be retained. Each time a  $\overline{\text{CAS}}$  before  $\overline{\text{RAS}}$  refresh cycle occurs, the on-chip refresh circuitry will refresh a row of the array at the row address specified by a refresh counter. The counter is incremented after each refresh. How often should  $\overline{\text{CAS}}$ -before- $\overline{\text{RAS}}$  cycles be applied in order for all of the data to be retained?
- N** 12-31. The SDRAM on the DE1 board from Terasic contains an 8 Mbyte SDRAM. This IC has 12 address ( $A_{11}-A_0$ ). All 12 address bits are latched into the row address register by  $\overline{\text{RAS}}$ . The IC has two input pins (separate from the address inputs) to specify which bank is being accessed.
- How many banks are in this IC?
  - How many bytes are in each bank?
  - How many rows must be refreshed on this SDRAM?
  - How many bits must be latched into the column address register to select one of the memory bytes stored in the selected bank?

### SECTION 12-19

- D** 12-32. Show how to combine two 6264 RAM chips (use the internet to look up the data sheet) to produce an  $8\text{K} \times 16$  module.

- D** 12-33. Show how to connect two of the 6264 RAM chips (use the internet to look up the data sheet) to produce a  $16\text{K} \times 8$  RAM module. The circuit should not require any additional logic. Draw a memory map showing the address range of each RAM chip.
- D** 12-34.\*Describe how to modify the circuit of Figure 12-35 so that it has a total capacity of  $16\text{K} \times 8$ . Use the same type of PROM chips.
- D** 12-35. Modify the decoding circuit of Figure 12-35 to operate from a 16-line address bus (i.e., add  $A_{13}$ ,  $A_{14}$ , and  $A_{15}$ ). The four PROMs are to maintain the same hex address ranges.
- C** 12-36. For the memory system of Figure 12-36, assume that the CPU is storing one byte of data at system address 4000 (hex).
- Which chip is the byte stored in?
  - Is there any other address in this system that can access this data byte?
  - Answer parts (a) and (b) by assuming that the CPU has stored a byte at address 6007. (*Hint*: Remember that the EEPROM is not completely decoded.)
  - Assume that the program is storing a sequence of data bytes in the EEPROM and that it has just completed the 2048th byte at address 67FF. If the programmer allows it to store one more byte at address 6800, what will be the effect on the first 2048 bytes?
- D** 12-37. Draw the complete diagram for a  $256\text{K} \times 8$  memory that uses RAM chips with the following specifications:  $64\text{K} \times 4$  capacity, common input/output line, and two active-LOW chip select inputs. [*Hint*: The circuit can be designed using only two inverters (plus memory chips).]

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

---

### SECTION 12-1

1. See text.    2. 16 bits per word; 8192 words; 131,072 bits or cells    3. In a read operation, a word is taken from a memory location and is transferred to another device. In a write operation, a new word is placed in a memory location and replaces the one previously stored there.    4. True    5. SAM: Access time is not constant but depends on the physical location of the word being accessed. RAM: Access time is the same for any address location.    6. RWM is memory that can be read from or written to with equal ease. ROM is memory that is mainly read from and is written into very infrequently.    7. False; its data must be periodically refreshed.

### SECTION 12-2

1. 14, 12, 12    2. Commands the memory to perform a write operation  
3. When in its active state, this input enables the memory to perform the read or the write operation by responding to inputs  $OE$  and  $\overline{WE}$ . When in its inactive state, this input disables the memory so that it cannot perform the read or the write function.

### SECTION 12-3

1. Address lines, data lines, control lines    2. See text.    3. See text.

### SECTION 12-4

1. True    2. Apply desired address inputs; activate control input(s); data appear at data outputs.    3. Process of entering data into ROM

---

**SECTION 12-5**

1.  $A_3A_2A_1A_0 = 1001$     2. The row-select decoder activates one of the enable inputs of all registers in the selected row. The column-select decoder activates one of the enable inputs of all registers in the selected column. The output buffers pass the data from the internal data bus to the ROM output pins when the  $\overline{CS}$  and  $\overline{OE}$  inputs are activated.

**SECTION 12-6**

1. RD.    2.  $t_{ACC}$

**SECTION 12-7**

1. False; by the manufacturer    2. A PROM can be programmed once by the user. It cannot be erased and reprogrammed.    3. True    4. By exposure to UV light    5. True    6. Automatically programs data into memory cells one address at a time    7. An EEPROM can be electrically erased and reprogrammed without removal from its circuit, and it is byte erasable.    8. Low density; high cost; slow erase cycle    9. EEPROM    10. A neutralized charge on the floating gate.    11. A negative charge deposited on the floating gate.

**SECTION 12-8**

1. Electrically erasable and programmable in circuit    2. Higher density; lower cost    3. Short erase and programming times    4. The contents of this register control all internal chip functions.    5. The electrical configuration of memory cells is similar to circuits that can function as either a NAND or a NOR gate.    6. NOR.    7. NAND

**SECTION 12-9**

1. On power-up, the computer executes a small bootstrap program from ROM to initialize the system hardware and to load the operating system from mass storage (disk).    2. Circuit that takes data represented in one type of code and converts it to another type of code    3. Counter, ROM, DAC, low-pass filter

**SECTION 12-10**

1. Yes.    2. Ram can be easily written.    3. False, ROMs can be randomly accessed as well.

**SECTION 12-11**

1. Desired address applied to address inputs;  $\overline{WE} = 1$ ;  $\overline{CS}$  or  $\overline{CE}$  activated;  $\overline{OE}$  activated    2. To reduce pin count    3. 24, including  $V_{CC}$  and ground

**SECTION 12-12**

1. SRAM cells are flip-flops; DRAM cells use capacitors.    2. CMOS  
3. Memory    4. CPU    5. Read- and write-cycle times.

**SECTION 12-13**

1. Generally slower speed; need to be refreshed    2. Low power; high capacity; lower cost per bit    3. DRAM

**SECTION 12-14**

1. 256 rows  $\times$  256 columns    2. It saves pins on the chip.    3.  $1M = 1024K = 1024 \times 1024$ . Thus, there are 1024 rows by 1024 columns. Because  $1024 = 2^{10}$ , the chip needs 10 address inputs.    4.  $\overline{RAS}$  is used to latch the row address into the DRAM's row address register.  $\overline{CAS}$  is used to latch the column address into the

column address register. 5. *MUX* multiplexes the full address into the row and column addresses for input to the DRAM.

**SECTION 12-15**

1. (a) True (b) False (c) False (d) True 2. *MUX*

**SECTION 12-16**

1. (a) True (b) False 2. It provides row addresses to the DRAM during refresh cycles. 3. Address multiplexing and the refresh operation 4. (a) False (b) True

**SECTION 12-17**

1. No 2. Memory locations with same upper address (same row) 3. Only the column address must be latched. 4. Extended data output 5. *Burst*  
6. Transfers data on rising and falling edge of the clock. 7. Speed of the system.

**SECTION 12-18**

1. The hard disc drive in most computers 2. Magnetoresistive RAM 3. Low cost, large capacity, portability

**SECTION 12-19**

1. Sixteen 2. Four 3. False; when expanding memory capacity, each chip is selected by a different decoder output (see Figure 12-35). 4. True 5. Four

**SECTION 12-20**

1. To optimize speed vs. cost 2. Data are read out of memory in the same order they were written in. 3. A FIFO used to transfer data between devices with widely different operating speeds 4. Circular buffers “wrap around” from the highest address to the lowest, and the newest datum always overwrites the oldest.

---





# PROGRAMMABLE LOGIC DEVICE ARCHITECTURES\*

## ■ OUTLINE

- |      |                               |      |                                    |
|------|-------------------------------|------|------------------------------------|
| 13-1 | Digital Systems Family Tree   | 13-4 | The Altera MAX and MAX II Families |
| 13-2 | Fundamentals of PLD Circuitry | 13-5 | Generations of HCPLDs              |
| 13-3 | PLD Architectures             |      |                                    |

---

\*Diagrams of the Altera devices presented in this chapter have been reproduced through the courtesy of Altera Corporation, San Jose, California.

## ■ CHAPTER OUTCOMES

*Upon completion of this chapter, you will be able to:*

- Describe the different categories of digital system devices.
- Describe the different types of PLDs.
- Interpret PLD data book information.
- Define PLD terminology.
- Compare the different programming technologies used in PLDs.
- Compare the architectures of different types of PLDs.
- Compare the features of Altera CPLDs and FPGAs.

## ■ INTRODUCTION

Throughout the chapters of this book you have been introduced to a wide variety of digital circuits. You now know how the building blocks of digital systems work and can combine them to solve a wide variety of digital problems. More complicated digital systems, such as microcomputers and digital signal processors, have also been briefly described. The defining difference between microcomputer/DSP systems and other digital systems is that the former follow a programmed sequence of instructions that the designer specifies. Many applications require faster response than a microcomputer/DSP architecture can accommodate and in these cases, a conventional digital circuit must be used. In today's rapidly advancing technology market, most conventional digital systems are not being implemented with standard logic device chips containing only simple gates or MSI-type functions. Instead, programmable logic devices, which contain the circuitry necessary to create logic functions, are being used to implement digital systems. These devices are not programmed with a list of instructions, like a computer or DSP. Instead, their internal hardware is configured by electronically connecting and disconnecting points in the circuit.

Why have PLDs taken over so much of the market? With programmable devices, the same functionality can be obtained with one IC rather than using several individual logic chips. This characteristic means less board space, less power required, greater reliability, less inventory, and overall lower cost in manufacturing.

In the previous chapters you have become familiar with the process of programming a PLD using either AHDL or VHDL. At the same time, you have learned about all the building blocks of digital systems. The PLD implementations of digital circuits up to this point have been presented as a "black box." We have not been concerned with what was going on inside the PLD to make it work. Now that you understand all the circuitry inside the black box, it is time to turn the lights on in there and look at how it works. This will allow you to make the best decisions when selecting and

applying a PLD to solve a problem. This chapter will take a look at the various types of hardware available to design digital systems. We will then introduce you to the architectures of various families of PLDs.

## 13-1 DIGITAL SYSTEMS FAMILY TREE

### OUTCOMES

Upon completion of this section, you will be able to:

- Categorize various technology solutions to digital system implementation.
- Compare technology strengths and limitations between and within categories.
- Define common terms and acronyms common to digital systems.

While the major goal of this chapter is to investigate PLD architectures, it is also useful to look at the various hardware choices available to digital system designers because it should give us a little better perception of today's digital hardware alternatives. The desired circuit functionality can generally be achieved by using several different types of digital hardware. Throughout this book, we have described both standard logic devices as well as how programmable logic devices can be used to create the same functional blocks. Microcomputers and DSP systems can also often be applied with the necessary sequence of instructions (i.e., the application's program) to produce the desired circuit function. The design engineering decisions must take into account many factors, including the necessary speed of operation for the circuit, cost of manufacturing, system power consumption, system size, amount of time available to design the product, and so on. In fact, most complex digital designs include a mix of different hardware categories. Many trade-offs between the various types of hardware have to be weighed to design a digital system.

A digital system family tree (see Figure 13-1) showing most of the hardware choices that are currently available can be useful in sorting out the many categories of digital devices. The graphical representation in the figure

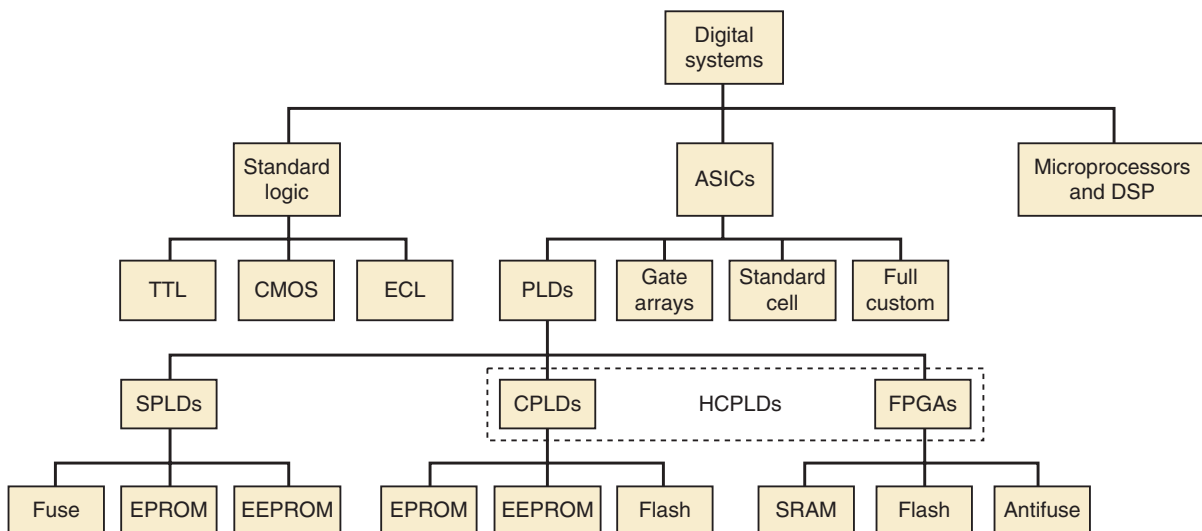


FIGURE 13-1 Digital system family tree.

does not show all the details—some of the more complex device types have many additional subcategories, and older, obsolete device types have been omitted for clarity. The major digital system categories include standard logic, application-specific integrated circuits (ASICs) and microprocessor/digital signal processing (DSP) devices.

The first category of **standard logic** devices refers to the basic functional digital components (gates, flip-flops, decoders, multiplexers, registers, counters, etc.) that are available as SSI and MSI chips. These devices have been used for many years (some more than 45 years) to design complex digital systems. An obvious drawback is that the system may literally consist of hundreds of such chips. These inexpensive devices can still be useful if our design is not very complex. As discussed in Chapter 8, there are two major technologies of logic devices: TTL, and CMOS. TTL is a mature technology consisting of numerous subfamilies that have been developed over many years of use. Very few new designs apply TTL logic, but many digital systems still contain TTL devices. CMOS is the most popular logic device technology today, primarily due to its low power consumption. Standard logic devices are still available to the digital designer, but if the application is very complex, a lot of SSI/MSI chips will be needed. That solution is not very attractive for our design needs today.

The **microprocessor/digital signal processing (DSP)** category is a much different approach to digital system design. These devices actually contain the various types of functional blocks that have been discussed throughout this text. With microcomputer/DSP systems, devices can be controlled electronically, and data can be manipulated by executing a program of instructions that has been written for the application. A great deal of flexibility can be achieved with microcomputer/DSP systems because all you have to do is change the program. The major downfall with this digital system category is speed. *Using a hardware solution for your digital system design is always faster than a software solution.*

The third major digital system category is called **application-specific integrated circuits (ASICs)**. This broad category represents the modern hardware design solution for digital systems. As the acronym implies, an integrated circuit is designed to implement a specific desired application. Four subcategories of ASIC devices are available to create digital systems: programmable logic devices, gate arrays, standard-cell, and full-custom.

**Programmable logic devices (PLDs)**, sometimes referred to as field-programmable logic devices (FPLDs), can be custom-configured to create any desired digital circuit, from simple logic gates to complex digital systems. Many examples of PLD designs have been given in earlier chapters. This ASIC choice for the designer is very different from the other three subcategories. With a relatively small capital investment, companies can purchase the necessary development software and hardware to program PLDs for their digital designs. On the other hand, to obtain a gate array, standard-cell or full-custom ASIC requires that most companies contract with an IC foundry to fabricate the desired IC chip. This option can be extremely expensive and usually requires that a company purchase a large volume of parts to be cost effective.

**Gate arrays** are ULSI circuits that offer hundreds of thousands of gates. The desired logic functions are created by the interconnections of these prefabricated gates. A custom-designed mask for the specific application determines the gate interconnections, much like the stored data in a mask-programmed ROM. For this reason, they are often referred to as mask-programmed gate arrays (MPGAs). Individually, these devices are less expensive than PLDs of comparable gate count, but the custom programming

process by the chip manufacturer is very expensive and requires a great deal of lead time.

**Standard-cell ASICs** use predefined logic function building blocks called cells to create the desired digital system. The IC layout of each cell has been designed previously, and a library of available cells is stored in a computer database. The needed cells are laid out for the desired application, and the interconnections between the cells are determined. Design costs for standard-cell ASICs are even higher than for MPGAs because all IC fabrication masks that define the components and interconnections must be custom-designed. Greater lead time is also needed for the creation of the additional masks. Standard cells do have a significant advantage over gate arrays. The cell-based functions have been designed to be much smaller than equivalent functions in gate arrays, which allows for generally higher-speed operation and cheaper manufacturing costs.

**Full-custom ASICs** are considered the ultimate ASIC choice. As the name implies, all components (transistors, resistors, and capacitors) and the interconnections between them are custom-designed by the IC designer. This design effort requires a significant amount of time and expense, but it can result in ICs that can operate at the highest possible speed and require the smallest die (individual IC chip) area. Smaller IC die sizes allow for many more die to fit on a silicon wafer, which significantly lowers the manufacturing cost for each IC.

### More on PLDs

This chapter is mainly about PLDs, so we will look a little further down that branch of the family tree. The development of PLD technology has advanced continuously since the first PLDs appeared more than 35 years ago. The early devices contained the equivalent of a few hundred gates, and now we have parts available that contain a few million gates. The old devices could handle a few inputs and a few outputs with limited logic capabilities. Now there are PLDs that can handle hundreds of inputs and outputs. Original devices could be programmed only once and, if the design changed, the old PLD would have to be removed from the circuit and a new one, programmed with the updated design, would have to be inserted in its place. With newer devices, the internal logic design can be changed on the fly, while the chip is still connected to a printed circuit board in an electronic system.

Generally, PLDs can be described as being one of three different types: **simple programmable logic devices (SPLDs)**, **complex programmable logic devices (CPLDs)**, or **field programmable gate arrays (FPGAs)**. There are several manufacturers with many different families of PLD devices, so there are many variations in architecture. We will attempt to discuss the general characteristics for each of the types, but be forewarned: The differences are not always clear-cut. The distinction between CPLDs and FPGAs is often a little fuzzy, with the manufacturers constantly designing new, improved architectures and frequently muddying the waters for marketing purposes. Together, CPLDs and FPGAs are often referred to as **high-capacity programmable logic devices (HCPLDs)**. The programming technologies for PLD devices are actually based on the various types of semiconductor memory. As new types of memory have been developed, the same technology has been applied to the creation of new types of PLD devices.

The amount of logic resources available is the major distinguishing feature between SPLDs and HCPLDs. Today, SPLDs are devices that typically contain the equivalent of 600 or fewer gates, while HCPLDs have thousands and hundreds of thousands of gates available. Internal programmable signal

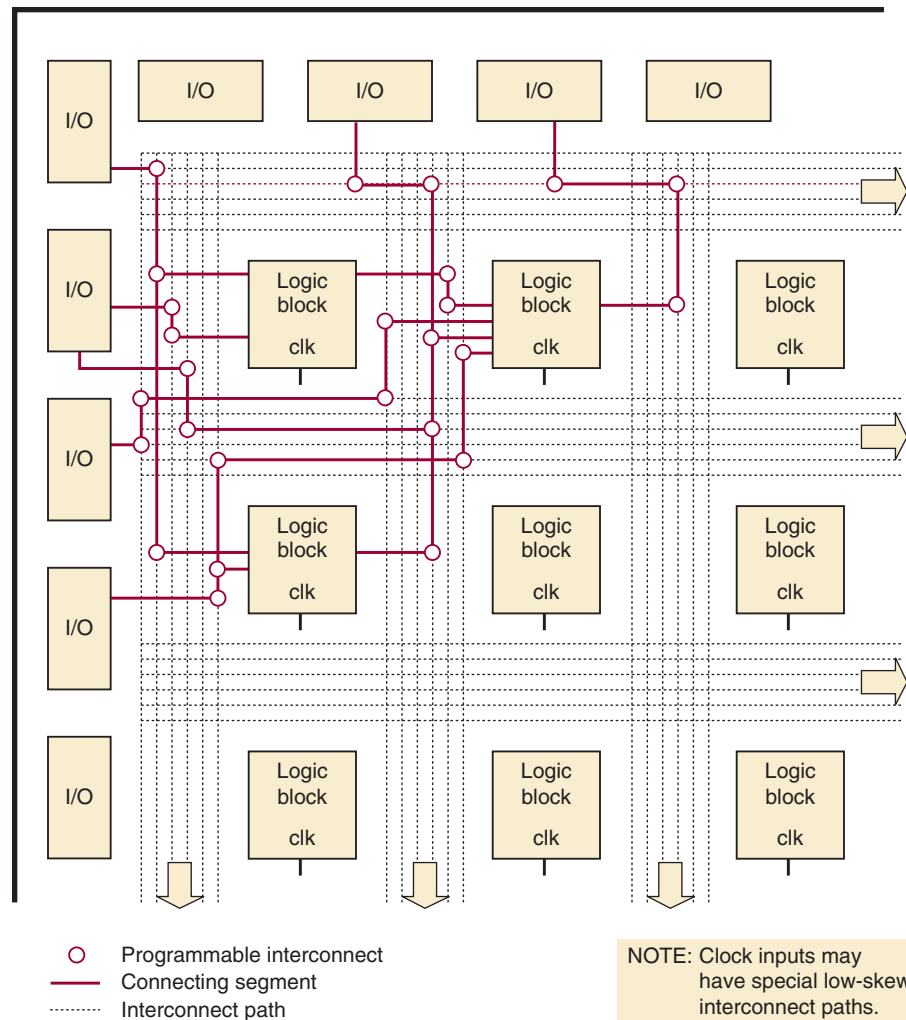
interconnect resources are much more limited with SPLDs. SPLDs are generally much less complicated and much cheaper than HCPLDs. Many small digital applications need only the resources of an SPLD. On the other hand, HCPLDs are capable of providing the circuit resources for complete complex digital systems, and larger, more sophisticated HCPLD devices are designed every year.

The SPLD classification includes the earliest PLD devices. The amount of logic resources contained in the early PLDs may be relatively small by today's standards, but they represented a significant technological step in their ability to create easily a custom IC that can replace several standard logic devices. Over the years, numerous semiconductor advances have created different SPLD types. The first PLD type to gain the interest of circuit designers was programmed by literally burning open selected fuses in the programming matrix. The fuses that were left intact in these **one-time programmable (OTP)** devices provided the electrical connections for the AND/OR circuits to produce the desired functions. This logic device was based on the fuse links in PROM memory technology (see Section 12-7) and was most commonly referred to as a programmable logic array (PLA). PLDs didn't really gain widespread acceptance with digital designers until the late 1970s, when a device called a **programmable array logic (PAL)** was introduced. The programmable fuse links in a PAL are used to determine the input connections to a set of AND gates that are wired to fixed OR gates. With the development of the ultraviolet erasable PROM came the EPROM-based PLDs in the mid 1980s, followed soon by PLDs using electrically erasable (EEPROM) technology.

CPLDs are devices that typically combine an array of PAL-type devices on the same chip. Most CPLDs contain logic blocks that have programmable AND/fixed-OR logic circuits with fewer product terms available than most PAL devices. Each logic block (often called a **macrocell**) can typically handle many input variables, and the internal programmable logic signal routing resources tend to be very uniform throughout the chip, producing consistent signal delays. When more product terms are needed, gates may be shared between logic blocks, or several logic blocks can be combined to implement the expression. The flip-flop used to implement the register in the macrocell can often be configured for D, JK, T (toggle), or SR operation. Input and output pins for some CPLD architectures are associated with a specific macrocell, and typically additional macrocells are buried (i.e., not connected to a pin). Other CPLD architectures may have independent I/O blocks with built-in registers that can be used to latch incoming or outgoing data. The programming technologies used in CPLD devices are all nonvolatile and include EPROM, EEPROM, and flash, with EEPROM being the most common. All three technologies are erasable and reprogrammable.

FPGAs also have a few fundamental characteristics that are shared. They typically consist of many relatively small and independent programmable logic modules that can be interconnected to create larger functions. Each module can usually handle only up to four or five input variables. Most FPGA logic modules utilize a **look-up table (LUT)** approach to create the desired logic functions. A LUT functions just like a truth table in which the output can be programmed to create the desired combinational function by storing the appropriate 0 or 1 for each input combination. The programmable signal routing resources within the chip tend to be quite varied, with many different path lengths available. The signal delays produced for a design depend on the actual signal routing selected by the programming software. The logic modules also contain programmable registers. The logic modules are not associated with any I/O pin. Instead, each

**FIGURE 13-2** FPGA architecture.



I/O pin is connected to a programmable input/output block that, in turn, is connected to the logic modules with selected routing lines. The I/O blocks can be configured to provide input, output, or bidirectional capability, and built-in registers can be used to latch incoming or outgoing data. A general architecture of FPGAs is shown in Figure 13-2. All of the logic blocks and input/output blocks can be programmed to implement almost any logic circuit. The programmable interconnections are accomplished via lines that run through the rows and columns in the channels between the logic blocks. Some FPGAs include large blocks of RAM memory; others do not.

The programming technologies used in FPGA devices include SRAM, flash, and antifuse, with SRAM being the most common. SRAM-based devices are volatile and therefore require the FPGA to be reconfigured (programmed) when it is powered-up. The programming information that defines how each logic block functions, which I/O blocks are inputs and outputs, and how the blocks are interconnected is stored in some type of external memory that is downloaded to the SRAM-based FPGA when power is applied. Antifuse devices are one-time programmable and are therefore nonvolatile. Antifuse memory technology is not currently used for memory devices but, as its name implies, it is the opposite of fuse technology. Instead of opening a fuse link to prevent a signal connection, an insulator layer between interconnects has an electrical short created to produce a

signal connection. Antifuse devices are programmed in a device programmer either by the end-user or by the factory or distributor.

Differences in architecture between CPLDs and FPGAs, among different HCPLD manufacturers, and among different families of devices from a single manufacturer can affect the efficiency of design implementation for a particular application. You may ask, “Does the architecture of this PLD family provide the best fit for my application?” It is very difficult, however, to predict which architecture may be the best choice to use for a complex digital system. Only a portion of the available gates can be utilized. Who knows how many equivalent gates will be needed for a large design? The basic design of the signal routing resources can affect how much of the PLD’s logic resources can be utilized. The segmented interconnects often found in FPGAs can produce shorter delays between adjacent logic blocks, but they may also produce longer delays between the blocks that are further apart than would be produced by the continuous type of interconnect found in most CPLDs. There is no easy answer to your question, but every HCPLD manufacturer will give you an answer anyway: their product is best!

Another important factor to consider when comparing FPGAs is the availability of **intellectual property (IP)**. This term refers to predefined designs of complex digital blocks that can be used together with your own design blocks to satisfy the needs of your applications. The IP blocks may be available from the specific FPGA manufacturers or from third-party sources. An example of some intellectual property designs from Altera is its family of versatile embedded processors called Nios® II. You can often evaluate the use of the intellectual property code in your design without charge, but generally there will be a licensing fee to be able to use it in your product. The types of intellectual property available for use in FPGA devices include various embedded processors, DSP building blocks, and standard core circuits for peripheral and interface functions. The life cycle for electronic products is becoming shorter and shorter due to the rapid pace of new technology developments and next-generation product features. It is becoming increasingly more important for designers to find ways to shorten the design and development cycle for new products. Intellectual property resources can significantly reduce the amount of time it takes to design a new product and, therefore, the amount of time it takes a company to get a product to the market for its customers to buy.

As you can see, the field of PLDs is quite diverse and it is constantly changing. You should now have the basic knowledge of the various types and technologies necessary to interpret PLD data sheets and learn more about them.

#### OUTCOME ASSESSMENT QUESTIONS

1. What are the three major categories of digital systems?
2. What is the major disadvantage of a microprocessor/DSP design?
3. What does ASIC stand for?
4. What are the four types of ASICs?
5. What are HCPLDs?
6. What are two major differences between CPLDs and FPGAs?
7. What does volatility refer to?



## 13-2 FUNDAMENTALS OF PLD CIRCUITRY

### OUTCOMES

Upon completion of this section, you will be able to:

- Describe architectural strategies for implementing programmable logic circuits.
- Recognize and use symbolic conventions common to PLD technical information.

A simple PLD device is shown in Figure 13-3. Each of the four OR gates can produce an output that is a function of the two input variables,  $A$  and  $B$ . Each output function is programmed with the fuses located between the AND gates and each of the OR gates.

Each of the inputs  $A$  and  $B$  feed both a noninverting buffer and an inverting buffer to produce the true and inverted forms of each variable. These are the *input lines* to the AND gate array. Each AND gate is connected to two different input lines to generate a unique product of the input variables. The AND outputs are called the *product lines*.

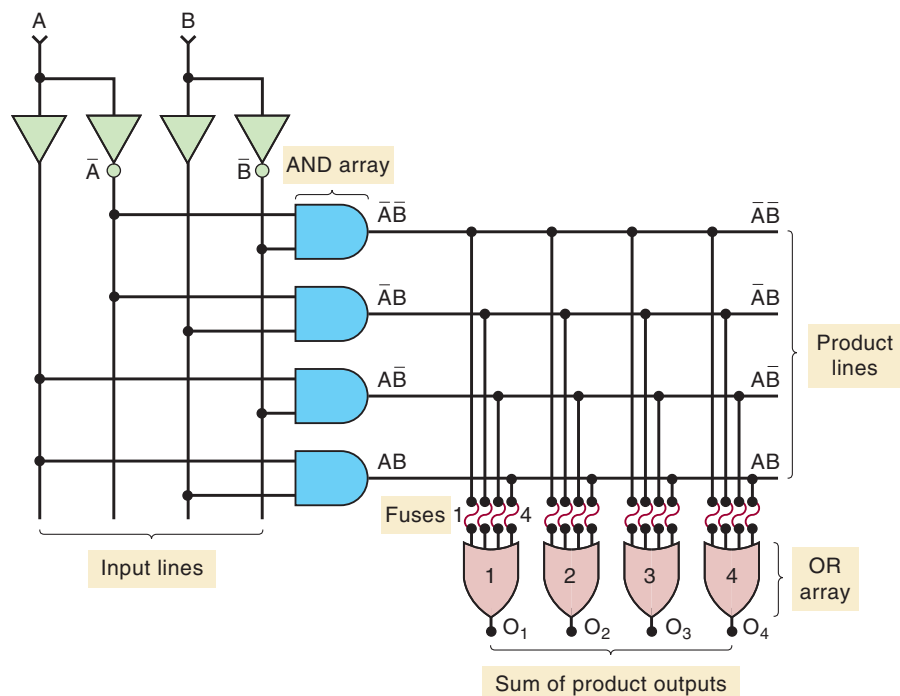
Each of the product lines is connected to one of the four inputs of each OR gate through a fusible link. With all of the links initially intact, each OR output will be a constant 1. Here's the proof:

$$\begin{aligned} O_1 &= \bar{A}\bar{B} + \bar{A}B + A\bar{B} + AB \\ &= \bar{A}(\bar{B} + B) + A(\bar{B} + B) \\ &= \bar{A} + A = 1 \end{aligned}$$

Each of the four outputs  $O_1$ ,  $O_2$ ,  $O_3$ , and  $O_4$  can be *programmed* to be any function of  $A$  and  $B$  by selectively blowing the appropriate fuses. PLDs are designed so that a blown OR input acts as a logic 0. For example, if we blow fuses 1 and 4 at OR gate 1, the  $O_1$  output becomes

$$O_1 = 0 + \bar{A}B + A\bar{B} + 0 = \bar{A}B + A\bar{B}$$

**FIGURE 13-3** Example of a programmable logic device.



We can program each of the OR outputs to any desired function in a similar manner. Once all of the outputs have been programmed, the device will permanently generate the selected output functions.

### PLD Symbology

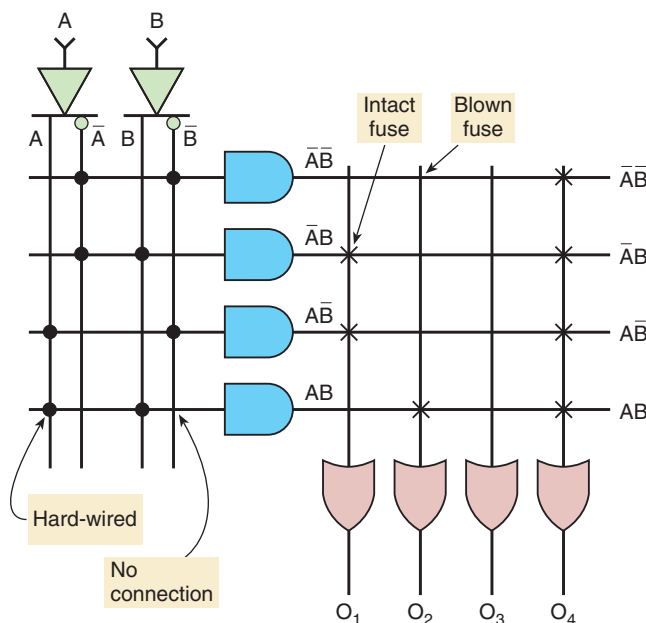
The example in Figure 13-3 has only two input variables and the circuit diagram is already quite cluttered. You can imagine how messy the diagram would be for PLDs with many more inputs. For this reason, PLD manufacturers have adopted a simplified symbolic representation of the internal circuitry of these devices.

Figure 13-4 shows the same PLD circuit as Figure 13-3 using the simplified symbols. First, notice that the input buffers are represented as a single buffer with two outputs, one inverted and one noninverted. Next, note that a *single line* is shown going into the AND gate to represent all four inputs. Each time the row line crosses a column represents a separate input to the AND gate. The connections from the input variable lines to the AND gate inputs are indicated as dots. A dot means that this connection to the AND gate input is hard-wired (i.e., one that cannot be changed). At first glance, it looks like the input variables are connected to each other. It is important to realize that this is *not* the case because the single row line represents *multiple* inputs to the AND gate.

The inputs to each of the OR gates are also designated by a single line representing all four inputs. An X represents an intact fuse connecting a product line to one input of the OR gate. The absence of an X (or a dot) at any intersection represents a blown fuse. For OR gate inputs, blown fuses (unconnected inputs) are assumed to be LOW, and for AND gate inputs, blown fuses are HIGH. In this example, the outputs are programmed as

$$\begin{aligned} O_1 &= \bar{A}B + A\bar{B} \\ O_2 &= AB \\ O_3 &= 0 \\ O_4 &= 1 \end{aligned}$$

**FIGURE 13-4** Simplified PLD symbology.



**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What is a PLD?
2. What would output  $O_1$  be in Figure 13-3 if fuses 1 and 2 were blown?
3. What does an X represent on a PLD diagram?
4. What does a dot represent on a PLD diagram?

### 13-3 PLD ARCHITECTURES

#### OUTCOMES

*Upon completion of this section, you will be able to:*

- Analyze hardware architecture of various programmable devices.
- Prescribe hardware settings for any logic function.
- Identify strengths and limitations of different devices.

The concept of PLDs has led to many different architectural designs of the inner circuitry of these devices. In this section, we will explore some of the basic differences in architecture.

#### PROMs

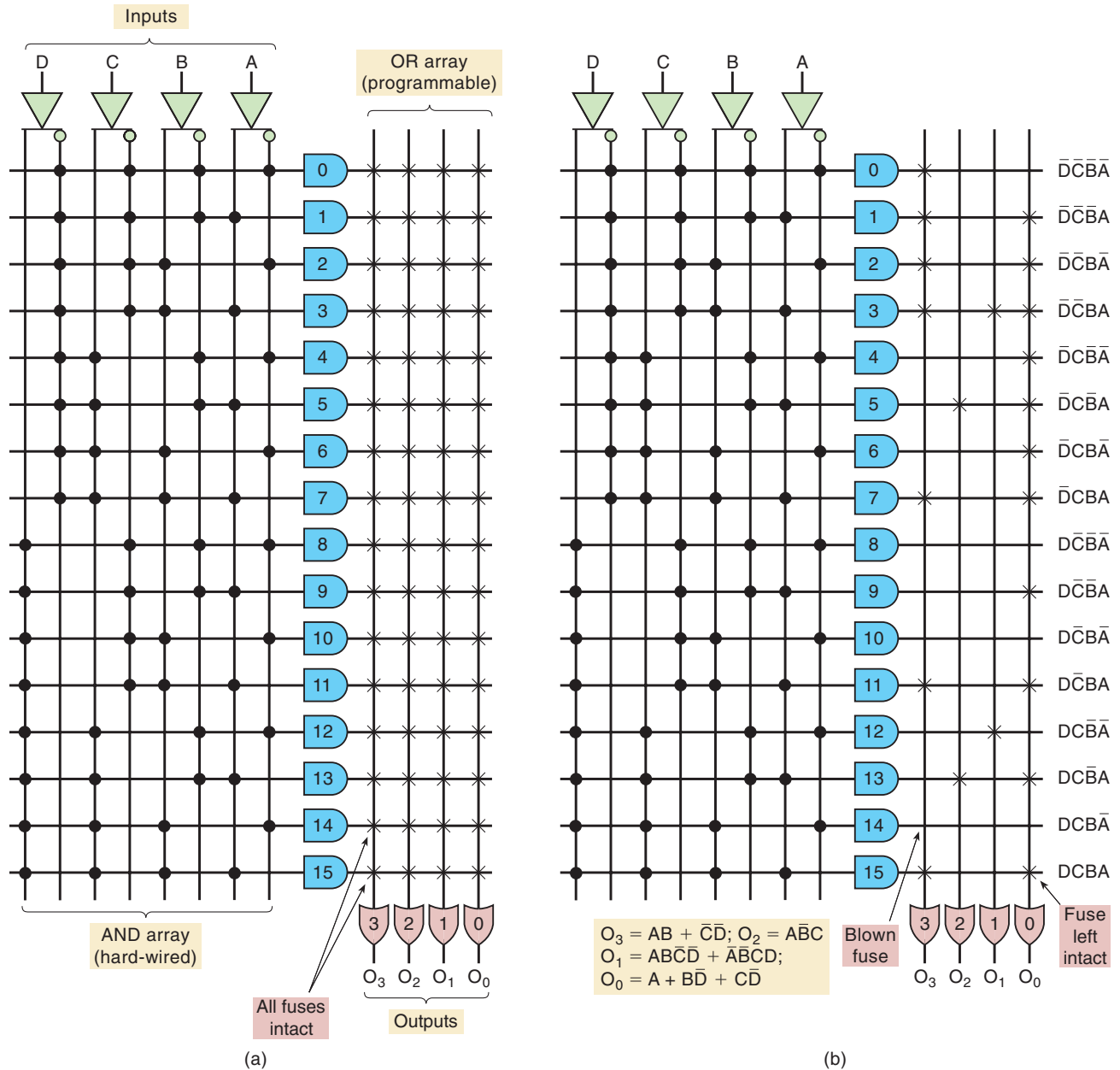
The architecture of the programmable circuits in the previous section involves programming the connections to the OR gate. The AND gates are used to decode all the possible combinations of the input variables, as shown in Figure 13-5(a). For any given input combination, the corresponding row is activated (goes HIGH). If the OR input is connected to that row, a HIGH appears at the OR output. If the input is not connected, a LOW appears at the OR output. Does this sound familiar? Refer back to Figure 12-9. If you think of the input variables as address inputs and the intact/blown fuses as stored 1s and 0s, you should recognize the architecture of a PROM.

Figure 13-5(b) shows how the PROM would be programmed to generate four specified logic functions. Let's follow the procedure for output  $O_3 = AB + \overline{C}\overline{D}$ . The first step is to construct a truth table showing the desired  $O_3$  output level for all possible input combinations (Table 13-1).

Next, write down the AND products for those cases where the output is to be a 1. The  $O_3$  output is to be the OR sum of these products. Thus, only the fuses that connect these product terms to the inputs of OR gate 3 are to be left intact. All others are to be blown, as indicated in Figure 13-5(b). This same procedure is followed to determine the status of the fuses at the other OR gate inputs.

The PROM can generate any possible logic function of the input variables because it generates every possible AND product term. In general, any application that requires every input combination to be available is a good candidate for a PROM. However, PROMs become impractical when a large number of input variables must be accommodated because the number of fuses doubles for each added input variable.

Calling a PROM a PLD is really just a semantics issue. You already knew that a PROM is programmable and it is a logic device. This is just a way of using a PROM and thinking of its purpose as implementing SOP logic expressions rather than storing data values in memory locations. The real problem is translating the logic equations into the fuse map for a given PROM. A general-purpose logic compiler designed to program SPLDs has



**FIGURE 13-5** (a) PROM architecture makes it suitable for PLDs; (b) fuses are blown to program outputs for given functions.

a list of PROM devices that it can support. If you choose to use any old scavenged EPROM as a PLD, you may need to generate your own bit map (like they used to do it), which is very tedious.

### Programmable Array Logic (PAL)

The PROM architecture is well suited for those applications where every possible input combination is required to generate the output functions. Examples are code converters and data storage (look-up) tables that we examined in Chapter 12. When implementing SOP expressions, however, they do not make very efficient use of circuitry. Each combination of address inputs must be fully decoded, and each expanded product term has

**TABLE 13-1** A PROM used to generate a logic function on  $O_3$ .

D	C	B	A	$O_3$
0	0	0	0	1 → $\overline{D}\overline{C}\overline{B}\overline{A}$
0	0	0	1	1 → $\overline{D}\overline{C}\overline{B}A$
0	0	1	0	1 → $\overline{D}\overline{C}B\overline{A}$
0	0	1	1	1 → $\overline{D}\overline{C}BA$
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1 → $\overline{D}CBA$
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1 → $D\overline{C}BA$
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1 → $DCBA$

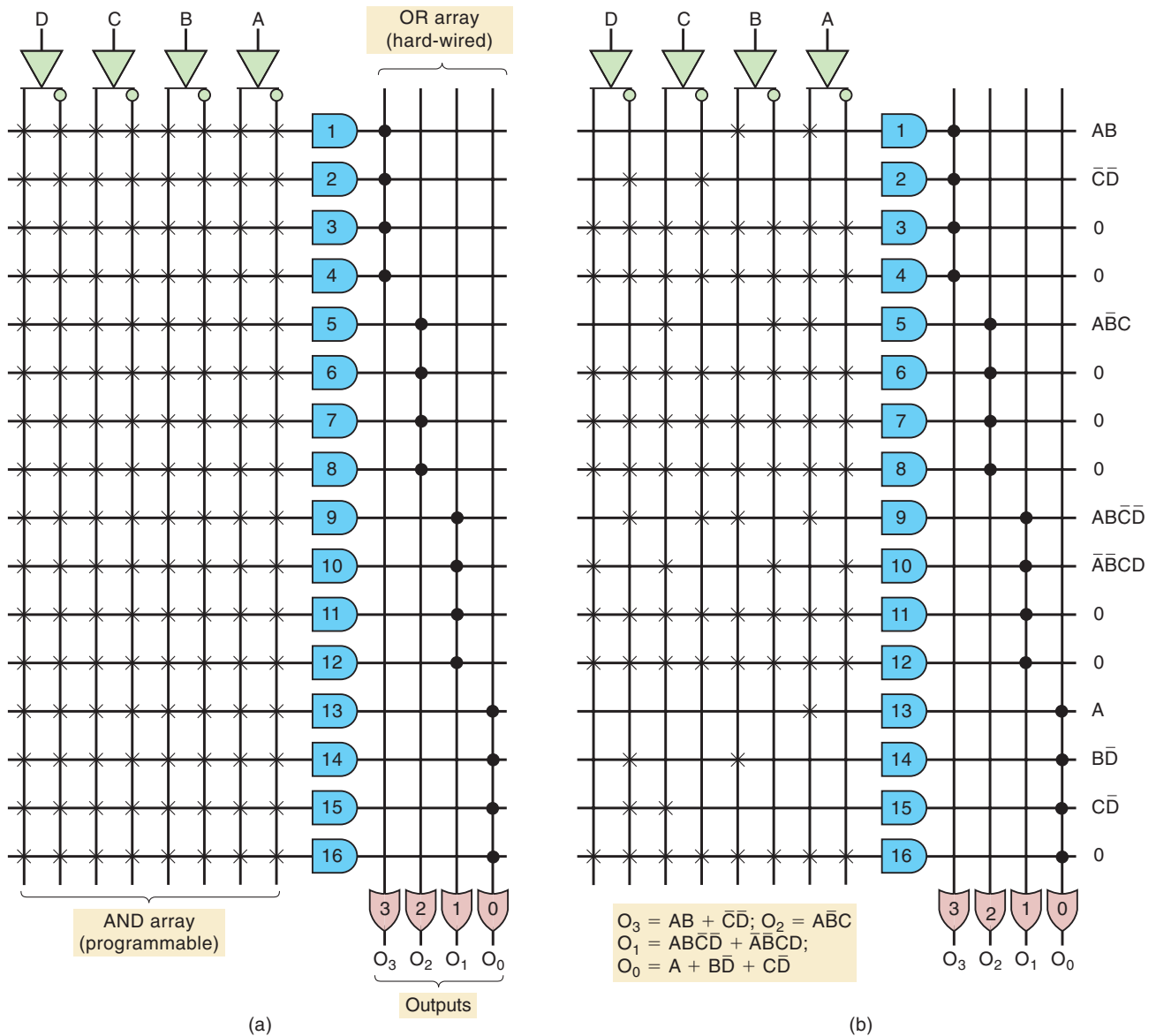
an associated fuse that is used to OR them together. For example, notice how many fuses were required in Figure 13-5 to program the simple SOP expressions and how many product terms are often not used. This has led to the development of a class of PLDs called programmable array logic (PAL). The architecture of a PAL differs slightly from that of a PROM, as shown in Figure 13-6(a).

The PAL has an AND and OR structure similar to a PROM but in the PAL, inputs to the AND gates are programmable, whereas the inputs to the OR gates are hard-wired. This means that every AND gate can be programmed to generate any desired product of the four input variables and their complements. Each OR gate is hard-wired to only four AND outputs. This limits each output function to four product terms. If a function requires more than four product terms, it cannot be implemented with this PAL; one having more OR inputs would have to be used. If fewer than four product terms are required, the unneeded ones can be made 0.

Figure 13-6(b) shows how this PAL is programmed to generate four specified logic functions. Let's follow the procedure for output  $O_3 = AB + \overline{C}\overline{D}$ . First, we must express this output as the OR sum of four terms because the OR gates have four inputs. We do this by putting in 0s. Thus, we have

$$O_3 = AB + \overline{C}\overline{D} + 0 + 0$$

Next, we must determine how to program the inputs to AND gates 1, 2, 3, and 4 so that they provide the correct product terms to OR gate 3. We do this term by term. The first term,  $AB$ , is obtained by leaving intact the fuses that connect inputs  $A$  and  $B$  to AND gate 1 and by blowing all other fuses on that line. Likewise, the second term,  $\overline{C}\overline{D}$ , is obtained by leaving intact only the fuses that connect inputs  $\overline{C}$  and  $\overline{D}$  to AND gate 2. The third term is a 0. A constant 0 is produced at the output of AND gate 3 by leaving all of its input fuses intact. This would produce an output of  $A\overline{A}B\overline{B}C\overline{C}D\overline{D}$ ,



**FIGURE 13-6** (a) Typical PAL architecture; (b) the same PAL programmed for the given functions.

which, as we know, is 0. The fourth term is also 0, so the input fuses to AND gate 4 are also left intact.

The inputs to the other AND gates are programmed similarly to generate the other output functions. Note especially that many of the AND gates have all of their input fuses intact because they need to generate 0s.

An example of an actual PAL integrated circuit is the PAL16L8, which has 10 logic inputs and eight output functions. Each output OR gate is hard-wired to seven AND gate outputs, and so it can generate functions that include up to seven terms. An added feature of this particular PAL is that six of the eight outputs are fed back into the AND array, where they can be connected as inputs to any AND gate. This makes it very useful in generating all sorts of combinational logic.

The PAL family also contains devices with variations of the basic SOP circuitry we have described. For example, most PAL devices have a tristate buffer driving the output pin. Others channel the SOP logic circuit to a D

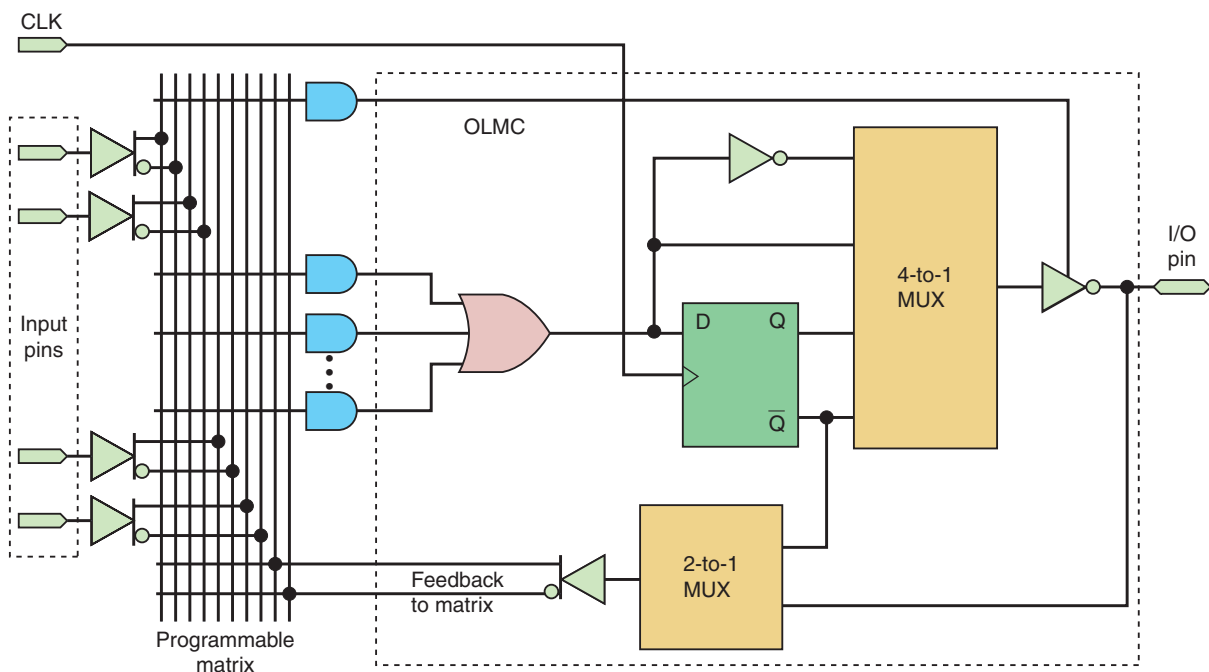
FF input and use one of the pins as a clock input to clock all of the output flip-flops synchronously. These devices are referred to as *registered PLDs* because the outputs pass through a register. An example is the PAL16R8, which has up to eight registered outputs (which can also serve as inputs) plus eight dedicated inputs.

### Field Programmable Logic Array (FPLA)

The field programmable logic array (FPLA) was developed in the mid-1970s as the first nonmemory programmable logic device. It used a programmable AND array as well as a programmable OR array. Although the FPLA is more flexible than the PAL architecture, it has not been as widely accepted by engineers. FPLAs are used mostly in state-machine design where a large number of product terms are needed in each SOP expression.

### Generic Array Logic (GAL)

Generic array logic devices have an architecture that is very similar to the PAL devices previously described. Standard, low-density PALs are one-time programmable. GAL chips, on the other hand, use an EEPROM array in the programmable matrix that determines the connections to the AND gates in an AND/OR circuit structure. The EEPROM switches can be erased and reprogrammed at least 100 times. The second feature that gives GAL chips a significant advantage over PAL devices is its programmable output logic macrocell (OLMC). In addition to the AND and OR gates used to provide the sum of product functions, GALs contain optional flip-flops for register and counter applications, tristate buffers for the outputs, and control multiplexers used to select the various modes of operation (see Figure 13-7). Consequently, GAL devices can be used as a generic, pin-compatible replacement for most PAL devices. Product terms created by



**FIGURE 13-7** Block diagram for programmable AND matrix and OLMC in GAL devices.

the AND gates that feed an OR gate in the OLMC will generate an SOP function that can be either routed to the output as a combinational function or, instead, can be clocked into a D flip-flop for a registered output. Specific locations in the EEPROM memory array are used to control the programmable connections and options for the chip. The programming software automatically takes care of all the details. GAL chips are inexpensive and versatile SPLD devices.

### OUTCOME ASSESSMENT QUESTIONS

1. Verify that the correct fuses are blown for the  $O_2$ ,  $O_1$ , and  $O_0$  functions in Figure 13-5(b).
2. A PAL has a hard-wired \_\_\_\_\_ array and a programmable \_\_\_\_\_ array.
3. A PROM has a hard-wired \_\_\_\_\_ array and a programmable \_\_\_\_\_ array.
4. How would the equation for the output of  $O_1$  in Figure 13-5(b) change if all the fuses from AND gate 14 were left intact?
5. Name two advantages of GAL devices over PAL devices.

## 13-4 THE ALTERA MAX AND MAX II FAMILIES

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe the similarities and differences between the SPLDs and the MAX7000S family of CPLDs.
- Describe the shift from AND/OR arrays to LUTs as the hardware solution to logic elements.
- Describe the shift to SRAM configuration over EEPROM.

The Altera MAX7000S family of CPLDs represents the next major evolutionary step in programmable logic after the GAL chips. One member of this family was used in Altera UP2 development boards that were widely used in educational institutions. These parts are now obsolete but help to show how technology has developed. This family used architecture for the macrocell that was very similar to that described for the GAL chips in the previous section. Groups of 16 of these macrocells are defined as **Logic Array Blocks (LABs)**. A single logic array block works very similar to a GAL chip containing 16 macrocells. However, in these CPLDs there are many LABs interconnected with each other through a **Programmable Interconnect Array (PIA)**. These devices were able to be configured while in circuit through a standard 4-wire JTAG serial interface or by removing the ICs from their circuit board and using a programmer. As modern programmable devices have become more complex and more densely packaged (surface mount technology), they are no longer removable. In-circuit programming via the JTAG interface has become the standard means of programming/configuring PLDs, memory devices, and microcontrollers. Notice that with each new family of devices, the trend is to incorporate architectural successes from previous generations, make architectural improvements, increase the density of these circuits within a chip by orders of magnitude in addition to increasing speed, improving energy efficiency, and reducing cost.

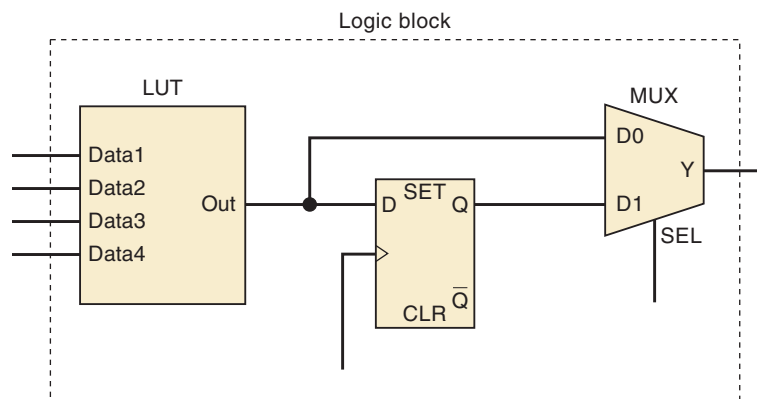


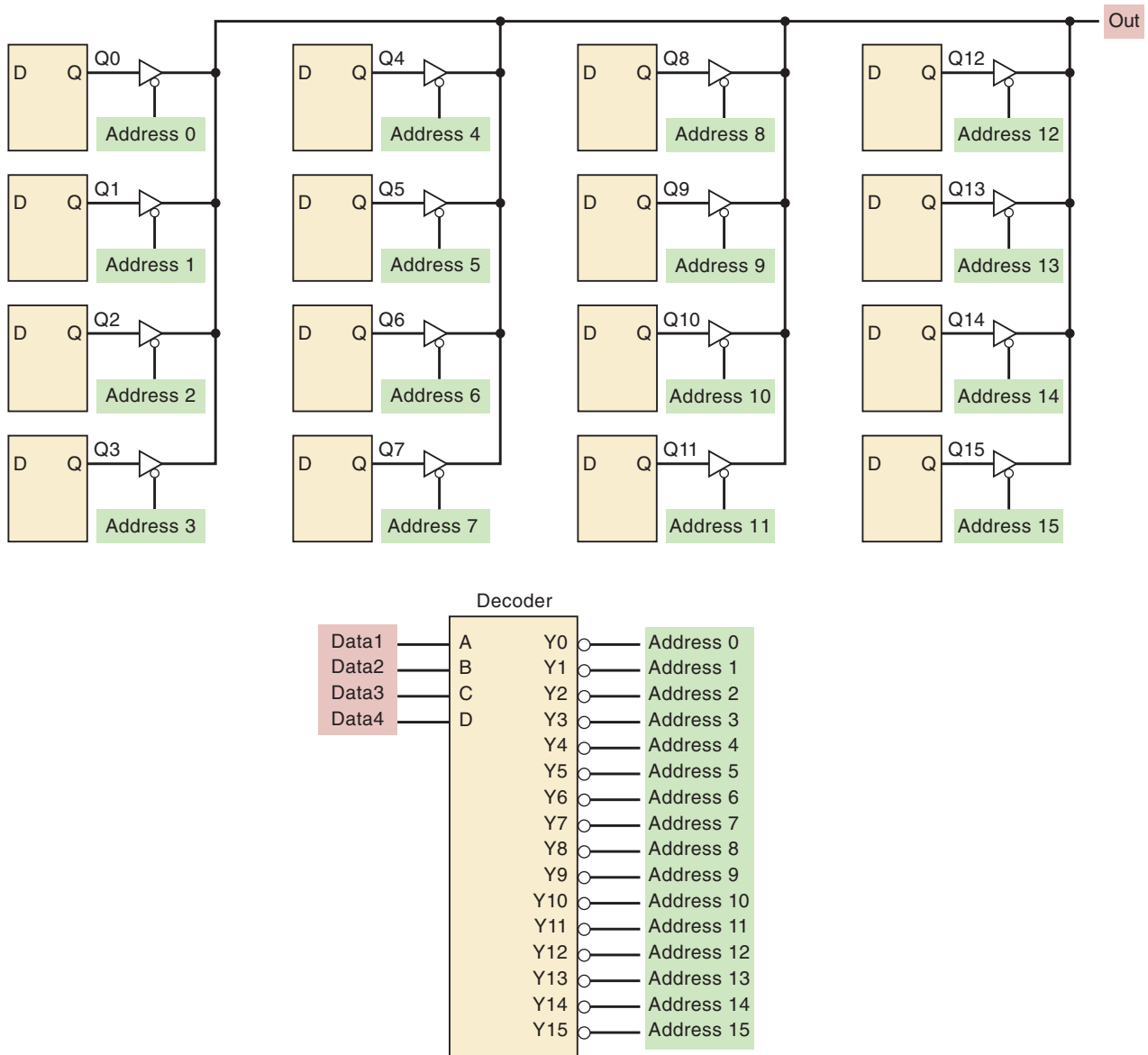
The newer Altera MAX II family of CPLDs has a much different architecture. Instead of the programmable-AND/fixed-OR gate array of macrocells and the global routing PIA used in the MAX7000S devices, this family uses a look-up table (LUT) architecture and a more die-area efficient row and column signal-routing structure. The look-up table produces logic functions by storing the desired function's output results in SRAM-based memory, acting essentially like the truth table for the logic function. SRAM technology for PLDs, although volatile, programs much faster than EEPROM-based devices and also allows for a very high density of storage cells that are used to program the larger PLD devices. In order to achieve seemingly “nonvolatile” programming of the MAX II devices, the configuration information for a design application is actually stored in a flash memory that is built into the chip, the configuration flash memory (CFM).

Let's examine the concept of a LUT. The LUT is the portion of the programmable logic block that produces a combinational function (see Figure 13-8). This function can be used as the output of the logic block or it may be registered (controlled by the internal MUX). The LUT itself consists of a set of flip-flops that store the desired truth table for our function. LUTs are usually rather small, typically handling four input variables, and so our truth table would have a total of 16 combinations. We will need a flip-flop to store each of the 16 function values (see Figure 13-9). Up to four input variables in our example LUT will be connected to the data inputs on the decoder block using programmable interconnects. The input combination that is applied will determine which of the 16 flip-flops will be selected to feed the output via the tristate buffers. The LUT is basically a  $16 \times 1$  SRAM memory block. All we have to do to create any desired function (of up to four input variables) is to store the appropriate set of 0s and 1s in the LUT's flip-flops. That is essentially what is done to program this type of PLD. Because the flip-flops are volatile (they are SRAM), we need to load the LUT memory for the desired functions whenever the PLD is powered-up. This process is called configuring the PLD. Other portions of the device are also programmed in the same fashion using other SRAM memory bits to store the programming information. This is the basic programming technique for the logic blocks, called **logic elements (LEs)**, found in the MAX II devices.

The MAX II architecture has groups of 10 LEs that are arranged together in a structure that is called a logic array block, LAB. The LABs are placed in rows and columns with the signal interconnect system, named MultiTrack, located in between the LAB rows and columns. The

**FIGURE 13-8** Simplified diagram of a programmable logic block that uses an LUT.





**FIGURE 13-9** Functional block diagram for an LUT.

I/O pins for MAX II devices are connected to input/output elements (IOEs) that are located around the periphery of the chip. The IOEs are, in turn, connected to the adjacent LABs at the ends of each row and column. The desired standard I/O characteristics are programmed into the device when it is configured.

Each MAX II device contains a flash memory block. The majority of this flash memory storage is the dedicated configuration flash memory block, which provides the nonvolatile storage for all of the SRAM configuration information. The CFM will automatically download and configure the logic and I/O at power-up, thereby providing nearly instant-on operation. The remaining portion of flash memory is called the user flash memory (UFM) block, which provides 8,192 bits of general-purpose user storage. The UFM has programmable port connections to the logic array for reading and writing data.

**OUTCOME  
ASSESSMENT  
QUESTIONS**

1. What characteristics are carried over from the SPLDs to MAX/7000 CPLDs?
2. What were the new improvements of the MAX7000S series?
3. What is a look-up table?
4. What advantage does SRAM programming technology have over EEPROM?
5. What disadvantage does SRAM programming technology have compared to EEPROM?

## 13-5 GENERATIONS OF HCPLDs

### OUTCOMES

*Upon completion of this section, you will be able to:*

- Describe the resources added with each new generation of FPGA.
- Identify the necessary characteristics for future generations of FPGAs.

The demands of the consumer product market are relentless and the most significant recent innovations have digital systems at their core. Altera and other semiconductor companies have kept up by delivering FPGAs that contain millions of logic elements. New series are constantly being developed and older series are being phased out. This section will simply provide an overview of that progression. The latest generations of FPGAs are far more powerful than any system that can be reasonably covered in a one semester course. Yet development boards with all of this capability are available for reasonable prices to educational institutions.

The MAX II was followed by the MAX V CPLD and the MAX 10 FPGA. The MAX V is still a nonvolatile part of similar architecture to the MAX II. The improvements beyond offering more logic elements were focused on including other conveniences to designers. Integrated into these FPGA parts were dedicated blocks of flash and RAM memory, oscillators, and phase-locked loops. The MAX 10 improved upon these features adding analog to digital converters, DSP blocks, and DDR3 DRAM memory controllers to go along with being smaller, faster, cooler, and cheaper.

The Cyclone family has been widely distributed on the Terasic DE0 Nano (Cyclone IV), DE1, and DE2 (Cyclone II) development boards that have been used widely in laboratory courses. These boards have all been upgraded to use more powerful recent generation FPGAs. The Cyclone V is now at the heart of the DE1 SoC, DE0 Nano SoC, and several other popular development boards. These devices have enough capacity to create microcontrollers inside the FPGA, along with the many peripheral blocks like multipliers, ADC, memory, memory controllers, Phase Lock Loops, and so on. The term SoC refers to System on Chip. We began this chapter by describing the many options of how to implement a digital system with standard logic, ASICs, or microprocessors/DSP. Now the FPGAs are able to include all of these functions within a single IC.

The basic architecture of a Cyclone series FPGA is similar to the structure of the MAX II family. In a Cyclone device, logic functions are implemented in LEs (logic elements) that contain a four-input, SRAM-based LUT (look-up table) and a programmable register (flip-flop). The LEs are grouped in LABs and signal-routing resources include MultiTrack, DirectLink, and local interconnects. Unlike the MAX II family, Cyclone devices are volatile

and must be configured at power-up. Cyclone devices can be configured using an external controller (such as a nonvolatile PLD or a microprocessor), a configuration memory device, or a download cable from a PC.

The Arria and Stratix series of FPGA continue to offer the features demanded by the explosion of digital systems and their insatiable appetite for improvements in speed, size, efficiency, and cost. The field of digital systems has never been more exciting. The opportunity to contribute to truly revolutionary, culture changing innovations has never been more open. You now have the foundation and the materials are available for you to build anything that you can imagine.

#### OUTCOME ASSESSMENT QUESTIONS

1. Name three resources that have been added to most FPGA ICs to make them more capable of implementing any digital systems.
2. Describe the improvements with each new generation of FPGA.

### SUMMARY

1. Programmable logic devices (PLDs) are the key technology in the future of digital systems.
2. PLDs can reduce parts inventory, simplify prototype circuitry, shorten the development cycle, reduce the size and power requirements of the product, and allow the hardware of a circuit to be upgraded easily.
3. The major digital system categories are standard logic, application-specific integrated circuits (ASICs), and microprocessor/digital signal processing (DSP) devices.
4. ASIC devices may be programmable logic devices (PLDs), gate arrays, standard cells, or full-custom devices.
5. PLDs are the least expensive type of ASIC to develop.
6. Simple PLDs (SPLDs) contain the equivalent of 600 or fewer gates and are programmed with fuse, EPROM, or EEPROM technology.
7. High-capacity PLDs (HCPLDs) have two major architectural categories: complex programmable logic devices (CPLDs) and field programmable gate arrays (FPGAs).
8. The most common CPLD programming technologies are EEPROM and flash, both of which are nonvolatile.
9. The most common FPGA programming technology is SRAM, which is volatile.
10. The Altera MAX7000s family of CPLDs contain multiple interconnected groups of macrocells programmed in-circuit through a JTAG interface.
11. The Altera MAX7000S family of CPLDs is nonvolatile and in-system programmable (ISP).
12. Since SRAM programming technology is volatile, it must be reconfigured at power-up; but it provides a very high density of storage cells that are used to program larger PLDs.
13. The Altera MAX II family of CPLDs uses on-chip flash memory to automatically configure the device at power-up.
14. The Altera Cyclone series of FPGAs is volatile.

## IMPORTANT TERMS

---

standard logic	standard-cell ASIC	programmable array logic (PAL)
microprocessor	full-custom ASIC	macrocell
digital signal processing (DSP)	simple PLD (SPLD)	look-up table (LUT)
application-specific integrated circuit (ASIC)	complex PLD (CPLD)	intellectual property (IP)
programmable logic device (PLD)	field programmable gate array (FPGA)	logic array block (LAB)
gate array	high-capacity PLD (HCPLD)	programmable interconnect array (PIA)
	one-time programmable (OTP)	logic element (LE)

## PROBLEMS

---

### SECTION 13-1

- 13-1. Describe each of the following major digital system categories:
- Standard logic
  - ASICs
  - Microprocessor/DSP
- 13-2.\*Name three factors that are generally considered when making design engineering decisions.
- 13-3. Why is a microprocessor/DSP system called a software solution for a design?
- 13-4.\*What major advantage does a hardware design solution have over a software solution?
- 13-5. Describe each of the following four ASIC subcategories:
- PLDs
  - Gate arrays
  - Standard-cell
  - Full-custom
- 13-6.\*What are the major advantages and disadvantages of a full-custom ASIC?
- 13-7. Name the six PLD programming technologies. Which is one-time programmable? Which is volatile?
- 13-8.\*How is the programming of SRAM-based PLDs different from other programming technologies?

### SECTION 13-4

- 13-9. Describe the functions of each of the following architectural structures found in the Altera MAX7000S family:
- LAB
  - PIA
  - Macrocell
- 13-10.\*What two ways can be used to program the MAX7000S family devices?

---

\*Answers to problems marked with an asterisk can be found in the back of the text.

- 13-11. What standard device interface is used for in-system programming in the MAX7000S family?
- 13-12.\*What is an LUT in the MAX II family?
- 13-13. Differentiate between the logic block structures that are used to produce a combinational function in the MAX7000S and MAX II families.
- 13-14. What type of programming technology is used in the LEs of the MAX II family?
- 13-15. How does a MAX II CPLD accomplish “instant on” at power-up in an application?

## ANSWERS TO OUTCOME ASSESSMENT QUESTIONS

---

### SECTION 13-1

1. Standard logic, ASICs, microprocessor    2. Speed    3. Application-specific integrated circuit    4. Programmable logic devices, gate arrays, standard cells, full custom    5. High-capacity programmable logic device    6. (1) Logic blocks: programmable AND/fixed-OR CPLD versus look-up table FPGA (2) Signal routing resources: uniform CPLD versus varied FPGA    7. Volatility refers to whether a PLD (or memory device) loses stored information when it is powered-down.

### SECTION 13-2

1. An IC that contains a large number of gates whose interconnections can be modified by the user to perform a specific function.    2.  $O_1 = A$     3. An intact fuse    4. A hard-wired connection

### SECTION 13-3

2. Hard-wired OR; programmable AND    3. Hard-wired AND; programmable OR  
 4.  $O_1 = ABCD + \bar{A}BCD + A\bar{B}CD = ABC\bar{D} + \bar{A}CD$     5. Erasable and reprogrammable; has an OLMC

### SECTION 13-4

1. The MAX family used similar AND array and macrocell architecture but grouped many of them into Logic Array Blocks (LAB) and interconnected them with a Programmable Interconnect Array (PIA).    2. Look-up tables utilize memory technology to implement truth tables of logic functions. This takes up much less space than a programmable array of connections to AND gates with OR gates in the macrocells.    3. A look-up table is typically a 16-word by one-bit SRAM array used to store the desired output logic levels for a simple logic function.    4. SRAM programs faster and has a higher logic cell density than EEPROM.    5. SRAM is volatile and must be reconfigured upon power-up of the device.

### SECTION 13-5

1. SRAM and FLASH memory blocks, phase-locked loops, multipliers, DACs, ADCs    2. FPGAs must continue to become more dense (i.e., more logic circuits in a smaller area). They must become faster, draw less power, and offer greater economy than previous generations.

---



# GLOSSARY

**Access Time** Time between the memory receiving a new input address and the output data becoming available in a read operation.

**Accumulator Register** Principal register of an arithmetic/logic unit (ALU).

**Acquisition Time** Time required for a sample-and-hold circuit to capture the analog value that is present on its input.

**Active-HIGH (LOW) Decoder** Decoder that produces a logic HIGH (LOW) at the output when detection occurs.

**Active Logic Level** Logic level at which a circuit is considered active. If the symbol for the circuit includes a bubble, the circuit is active-LOW. On the other hand, if it doesn't have a bubble, then the circuit is active-HIGH.

**Actuator** Electrically controlled device that controls a physical variable.

**Addend** Number to be added to another.

**Adder/Subtractor** An adder circuit that can subtract by complementing (negating) one of the operands. *See also* Parallel/Adder.

**Address** Number that uniquely identifies the location of a word in memory.

**Address Bus** Unidirectional lines that carry the address code from the CPU to memory and I/O devices.

**Address Multiplexing** Multiplexing used in dynamic RAMs to save IC pins. It involves latching the two halves of a complete address into the IC in separate steps.

**Alias** A digital signal that results from sampling an incoming signal at a rate less than twice the highest frequency contained in the incoming signal.

**Alphanumeric Codes** Codes that represent numbers, letters, punctuation marks, and special characters.

**Altera Hardware Description Language (AHDL)** A proprietary HDL developed by Altera Corporation for programming their programmable logic devices.

**Alternate Logic Symbol** A logically equivalent symbol that indicates the active level of the inputs and outputs.

**Analog Quantity** A value of a variable that can change continuously.

**Analog Representation** Representation of a quantity that varies over a continuous range of values.

**Analog System** Combination of devices designed to manipulate physical quantities that are represented in analog form.

**Analog-to-Digital Converter (ADC)** Circuit that converts an analog input to a corresponding digital output.

**Analog Voltage Comparator** Circuit that compares two analog input voltages and produces an output that indicates which input is greater.

**AND Gate** Digital circuit that implements the AND operation. The output of this circuit is HIGH (logic level 1) only if all of its inputs are HIGH.

**AND Operation** Boolean algebra operation in which the symbol is used to indicate the ANDing of two or more logic variables. The result of the AND operation will be HIGH (logic level 1) only if all variables are HIGH.

**Aperiodic** A waveform that does not have a constant and consistent period.

**Application-Specific Integrated Circuit (ASIC)** An IC that has been specifically designed to meet the requirements of an application. Subcategories include PLDs, gate arrays, standard cells, and full-custom ICs.

- ARCHITECTURE** Keyword in VHDL used to begin a section of code that defines the operation of a circuit block (ENTITY).
- Arithmetic/Logic Unit (ALU)** Digital circuit used in computers to perform various arithmetic and logic operations.
- ASCII Code (American Standard Code for Information Interchange)** Seven-bit alphanumeric code used by most computer manufacturers.
- Asserted** Term used to describe the state of a logic signal; synonymous with “active.”
- Astable Multivibrator** Digital circuit that oscillates between two unstable output states.
- Asynchronous Counter** Type of counter in which each flip-flop output serves as the clock input signal for the next flip-flop in the chain.
- Asynchronous Inputs** Flip-flop inputs that can affect the operation of the flip-flop independent of the synchronous and clock inputs.
- Asynchronous Transfer** Data transfer performed without the aid of the clock.
- Augend** Number to which an addend is added.
- Auxiliary Memory** The part of a computer’s memory that is separate from the computer’s main working memory. Generally has high density and high capacity, such as magnetic disk.
- Backplane** Electrical connection common to all segments of an LCD.
- Barrel Shifter** A shift register that can very efficiently shift a binary number left or right by any number of bit positions.
- BCD Counter** Binary counter that counts from 0000<sub>2</sub> to 1001<sub>2</sub> before it recycles.
- BCD-to-Decimal Decoder** Decoder that converts a BCD input into a single decimal output equivalence.
- BCD-to-7-Segment Decoder/Driver** Digital circuit that takes a four-bit BCD input and activates the required outputs to display the equivalent decimal digit on a 7-segment display.
- Behavioral Level of Abstraction** A technique of describing a digital circuit that focuses on how the circuit reacts to its inputs.
- Bidirectional Data Line** Term used when a data line functions as either an input or an output line depending on the states of the enable inputs.
- Bilateral Switch** CMOS circuit that acts like a single-pole, single-throw (SPST) switch controlled by an input logic level.
- Binary-Coded-Decimal Code (BCD Code)** Four-bit code used to represent each digit of a decimal number by its four-bit binary equivalent.
- Binary Counter** Group of flip-flops connected in a special arrangement in which the states of the flip-flops represent the binary number equivalent to the number of pulses that have occurred at the input of the counter.
- Binary Digit** Bit.
- Binary Point** Mark that separates the integer from the fractional portion of a binary quantity.
- Binary Number System** Number system in which there are only two possible digit values, 0 and 1.
- Bipolar DAC** Digital-to-analog converter that accepts signed binary numbers as input and produces the corresponding positive or negative analog output value.
- Bipolar ICs** Integrated digital circuits in which NPN and PNP transistors are the main circuit elements.
- BIT** In VHDL, the data object type representing a single binary digit (bit).
- Bit** Digit in the binary system.
- Bit Array** A way to represent a group of bits by giving it a name and assigning an element number to each bit’s position. This same structure is sometimes called a bit vector.
- BIT\_VECTOR** In VHDL, the data object type representing a bit array. *See also* Bit Array.
- Boolean Algebra** Algebraic process used as a tool in the design and analysis of digital systems. In Boolean algebra only two values are possible, 0 and 1.
- Boolean Theorems** Rules that can be applied to Boolean algebra to simplify logic expressions.
- Bootstrap Program** Program, stored in ROM, that a computer executes on power-up.
- Bubbles** Small circles on the input or output lines of logic-circuit symbols that represent inversion of a particular signal. If a bubble is present, the input or output is said to be active-LOW.
- Buffer/Driver** Circuit designed to have a greater output current and/or voltage capability than an ordinary logic circuit.
- Buffer Register** Register that holds digital data temporarily.
- Buried Node** A defined point in a circuit that is not accessible from outside that circuit.
- Bus** Group of wires that carry related bits of information.
- Bus Contention** Situation in which the outputs of two or more active devices are placed on the same bus line at the same time.
- Bus Drivers** Circuits that buffer the outputs of devices connected to a common bus; used when a large number of devices share a common bus.
- Byte** Group of eight bits.
- Cache Memory** A high-speed memory system that can be loaded from the slower system DRAM and accessed quickly by the high-speed CPU.
- Capacity** Amount of storage space in a memory expressed as the number of bits or number of words.
- Carry** Digit or bit that is generated when two numbers are added and the result is greater than the base for the number system being used.
- Carry Propagation** Intrinsic circuit delay of some parallel adders that prevents the carry bit (C<sub>OUT</sub>) and the result of the addition from appearing at the output simultaneously.
- Carry Ripple** *See* Carry Propagation.
- CAS (Column Address Strobe)** Signal used to latch the column address into a DRAM.



- CAS-before-RAS** Method for refreshing DRAMs that have built-in refresh counters. When the CAS input is driven LOW and held there as RAS is pulsed LOW, an internal refresh operation is performed at the row address given by the on-chip refresh counter.
- Cascading** Connecting logic circuits in a serial fashion with the output of one circuit driving the input of the next, and so on.
- CASE** A control structure that selects one of several options when describing a circuit's operation based on the value of a data object.
- Central Processing Unit (CPU)** Part of a computer that is composed of the arithmetic/logic unit (ALU) and the control unit.
- Chip Select** Input to a digital device that controls whether or not the device will perform its function. Also called *chip enable*.
- Circuit Excitation Table** Table showing a circuit's possible PRESENT-to-NEXT state transitions and the required *J* and *K* levels at each flip-flop.
- Circular Buffer** A memory system that always contains the last *n* data values that have been written. Whenever a new data value is stored, it overwrites the oldest value in the buffer.
- Circulating Shift Register** Shift register in which one of the outputs of the last flip-flop is connected to the input of the first flip-flop.
- CLEAR** An input to a latch or FF used to make  $Q = 0$ .
- CLEAR State** The  $Q = 0$  state of a flip-flop.
- Clock** Digital signal that controls the timing of events in a synchronous system.
- Clock Skew** Arrival of a clock signal at the clock inputs of different flip-flops at different times as a result of propagation delays.
- Clock Transition Times** Minimum rise and fall times for the clock signal transitions used by a particular IC, specified by the IC manufacturer.
- Clocked D Flip-Flop** Type of flip-flop in which the *D* (data) input is the synchronous input.
- Clocked Flip-Flops** Flip-flops that have a clock input.
- Clocked J-K Flip-Flop** Type of flip-flop in which inputs *J* and *K* are the synchronous inputs.
- Clocked S-R Flip-Flop** Type of flip-flop in which the inputs SET and RESET are the synchronous inputs.
- CMOS (Complementary Metal-Oxide-Semiconductor)** Integrated-circuit technology that uses MOSFETs as the principal circuit element. This logic family belongs to the category of unipolar digital ICs.
- Codec** Stands for Code/Decode. A device that performs analog-to-digital (coding) and digital-to-analog (decoding) conversions.
- Combinational Logic Circuits** Circuits made up of combinations of logic gates, with no feedback from outputs to inputs.
- Comments** Text added to any HDL design file or computer program to describe the purpose and operation of the code in general or of individual statements in the code. Documentation regarding author, date, revision, and so on, may also be contained in the comments.
- Common Anode** LED display that has the anodes of all of the segment LEDs tied together.
- Common Cathode** LED display that has the cathodes of all of the segment LEDs tied together.
- Compiler** A program that translates a text file written in a high-level language into a binary file that can be loaded into a programmable device such as a PLD or a computer's memory.
- Complement** See Invert.
- Complex PLD (CPLD)** Class of PLDs that contain an array of PAL-type blocks that can be interconnected.
- COMPONENT** A VHDL keyword used at the top of a design file to provide information about a library component.
- Computer Word** Group of binary bits that form the primary unit of information in a computer.
- Concatenate** A term used to describe the arrangement or linking of two or more data objects into ordered sets.
- Concurrent** Events that occur simultaneously (at the same time). In HDL, the circuits generated by concurrent statements are not affected by the order or sequence of the statements in the code.
- Concurrent Assignment Statement** A statement in AHDL or VHDL that describes a circuit that works concurrently with all other circuits that are described by concurrent statements.
- Conditional Signal Assignment** A VHDL concurrent construct that evaluates a series of conditions sequentially to determine the appropriate value to assign to a signal. The first true condition evaluated determines the assigned value.
- Conditional Transition** When two or more arrows leave a bubble in a state transition diagram. Each arrow is labelled with the "condition" of the variable that causes the transition.
- Constants** Symbolic names that can be used to represent fixed numeric (scalar) values.
- Contact Bounce** The tendency of all mechanical switches to vibrate when forced to a new position. The vibrations cause the circuit to make contact and break contact repeatedly until the vibrations settle out.
- Contention** Two (or more) output signals connected together trying to drive a common point to different voltage levels. See also Bus Contention.
- Control Bus** Set of signal lines that are used to synchronize the activities of the CPU and the separate  $\mu\text{C}$  elements.
- Control Inputs** Input signals that synchronously or asynchronously determine the output state of a flip-flop.
- Control Unit** Part of a computer that provides decoding of program instructions and the necessary timing and control signals for the execution of such instructions.
- Counter** A sequential logic circuit made up of flip-flops that are clocked through a sequence of states. The sequence is often (but not always) consecutive integers or a counting sequence.
- Count Enable** An input on a synchronous counter that controls whether the outputs respond to or ignore an active clock transition.

- Crystal-Controlled Clock Generator** Circuit that uses a quartz crystal to generate a clock signal at a precise frequency.
- Current-Sinking** The output of a logic circuit sinks current from the input of the logic circuit that it is driving.
- Current-Sinking Transistor** Name given to the output transistor ( $Q_4$ ) of a TTL circuit. This transistor is turned on when the output logic level is LOW.
- Current-Sourcing** The output of a logic circuit sources, or supplies, current to the input of the logic circuit that it is driving.
- Current-Sourcing Transistor** Name given to the output transistor ( $Q_3$ ) of most TTL circuits. This transistor is conducting when the output logic level is HIGH.
- Current Transients** Current spikes generated by the totem-pole output structure of a TTL circuit and caused when both transistors are simultaneously turned on.
- D Flip-Flop** See Clocked D Flip-Flop.
- D Latch** Circuit that contains a NAND gate latch and two steering NAND gates.
- Data** Binary representations of numerical values or nonnumerical information in a digital system. Data are used and often modified by a computer program.
- Data Acquisition** Process by which a computer acquires digitized analog data.
- Data Bus** Bidirectional lines that carry data between the CPU and the memory, or between the CPU and the I/O devices.
- Data Compression** A strategy that allows large data files to be stored in a much smaller memory space.
- Data Distributors** See Demultiplexer.
- Data-Rate Buffer** Application of FIFOs in which sequential data are written into the FIFO at one rate and read out at a different rate.
- Data Selectors** See Multiplexer.
- Data Transfer** See Parallel Data Transfer or Serial Data Transfer.
- Decade Counter** Any counter capable of going through 10 different logic states.
- Decimal System** Number system that uses 10 different digits or symbols to represent a quantity.
- Decision Control Structures** The statements and syntax that describe how to choose between two or more options in the code.
- Decoder** Digital circuit that converts an input binary code into a corresponding single active output.
- Decoding** Act of identifying a particular binary combination (code) in order to display its value or recognize its presence.
- DEFAULTS** An AHDL keyword used to establish a default value for a combinational signal for instances when the code does not explicitly specify a value.
- DeMorgan's Theorems** (1) Theorem stating that the complement of a sum (OR operation) equals the product (AND operation) of the complements, and (2) theorem stating that the complement of a product (AND operation) equals the sum (OR operation) of the complements.
- Demultiplexer (DEMUX)** Logic circuit that, depending on the status of its select inputs, will channel its data input to one of several data outputs.
- Density** A relative measure of capacity to store bits in a given amount of space.
- Differential Inputs** Method of connecting an analog signal to an analog circuit's + and - inputs, such that the analog circuit acts upon the voltage difference between the two inputs.
- Digital Computer** System of hardware that performs arithmetic and logic operations, manipulates data, and makes decisions.
- Digital Integrated Circuits** Self-contained digital circuits made by using one of several integrated-circuit fabrication technologies.
- Digital One-Shot** A one-shot that uses a counter and clock rather than an RC circuit as a time base.
- Digital Quantity** A value of a variable that can only change in discrete steps.
- Digital-Ramp ADC** Type of analog-to-digital converter in which an internal staircase waveform is generated and utilized for the purpose of accomplishing the conversion. The conversion time for this type of analog-to-digital converter varies depending on the value of the input analog signal.
- Digital Representation** Representation of a quantity that varies in discrete steps over a range of values.
- Digital Signal Processing (DSP)** Method of performing repetitive calculations on an incoming stream of digital data words to accomplish some form of signal conditioning. The data are typically digitized samples of an analog signal.
- Digital Storage Oscilloscope** Instrument that samples, digitizes, stores, and displays analog voltage waveforms.
- Digital System** Combination of devices designed to manipulate physical quantities that are represented in digital form.
- Digital-to-Analog Converter (DAC)** Circuit that converts a digital input to a corresponding analog output.
- Digitization** Process by which an analog signal is converted to digital data.
- Disable** Action in which a circuit is prevented from performing its normal function, such as passing an input signal through to its output.
- Divide-and-Conquer** Troubleshooting technique whereby tests are performed that will eliminate half of all possible remaining causes of the malfunction.
- Don't-Care Condition** Situation when a circuit's output level for a given set of input conditions can be assigned as either a 1 or a 0.
- Down Counter** Counter that counts from a maximum count downward to 0.
- Downloading** Process of transferring output files to a programming fixture.
- DRAM Controller** IC used to handle refresh and address multiplexing operations needed by DRAM systems.
- Driver** Technical term sometimes added to an IC's description to indicate that the IC's outputs can operate at higher current and/or voltage limits than a normal standard IC.

- Dual-in-Line Package (DIP)** A very common IC package with two parallel rows of pins intended to be inserted into a socket or through holes drilled in a printed circuit board.
- Dual-Slope ADC** Type of analog-to-digital converter that linearly charges a capacitor from a current proportional to  $V_A$  for a fixed time interval and then increments a counter as the capacitor is linearly discharged to 0.
- Duty Cycle** The portion of time that a periodic pulse waveform is in its active or asserted state expressed in percentage. For active-HIGH signals, it is the time HIGH divided by the period.
- Dynamic RAM (DRAM)** Type of semiconductor memory that stores data as capacitor charges that need to be refreshed periodically.
- Edge** A transition of a waveform from LOW to HIGH or from HIGH to LOW.
- Edge-Detector Circuit** Circuit that produces a narrow positive spike that occurs coincident with the active transition of a clock input pulse.
- Edge-Triggered** Manner in which a flip-flop is activated by a signal transition. A flip-flop may be either a positive- or a negative-edge-triggered flip-flop.
- Electrically Compatible** When two ICs from different logic series can be connected directly without any special measures taken to ensure proper operation.
- Electrically Erasable Programmable ROM (EEPROM)** ROM that can be electrically programmed, erased, and reprogrammed.
- Electrostatic Discharge (ESD)** The often detrimental act of the transfer of static electricity (i.e., an electrostatic charge) from one surface to another. This impulse of current can destroy electronic devices.
- ELSE** A control structure used in conjunction with IF/THEN to perform an alternate action in the case that the condition is false. An IF/THEN/ELSE always performs one of two actions.
- ELSIF** A control structure that can be used multiple times following an IF statement to select one of several options in describing a circuit's operation based on whether the associated expressions are true or false.
- Embedded Microcontroller** Microcontroller that is embedded in a marketable product such as a VCR or an appliance.
- Enable** Action in which a circuit is allowed to perform its normal function, such as passing an input signal through to its output.
- Encoder** Digital circuit that produces an output code depending on which of its inputs is activated.
- Encoding** Use of a group of symbols to represent numbers, letters, or words.
- ENTITY** Keyword in VHDL used to define the basic block structure of a circuit. This word is followed by a name for the block and the definitions of its input/output ports.
- Enumerated Type** A VHDL user-defined type for a signal or variable.
- Erasable Programmable ROM (EPROM)** ROM that can be electrically programmed by the user. It can be erased (usually with ultraviolet light) and reprogrammed as often as desired.
- EVENT** A VHDL keyword used as an attribute attached to a signal to detect a transition of that signal. Generally, an event means a signal changed state.
- Exclusive-NOR (XNOR) Circuit** Two-input logic circuit that produces a HIGH output only when the inputs are equal.
- Exclusive-OR (XOR) Circuit** Two-input logic circuit that produces a HIGH output only when the inputs are different.
- Falling Edge** See Negative Going Transition.
- Fan-Out** Maximum number of standard logic inputs that the output of a digital circuit can reliably drive.
- Feedback** A common practice of feeding output information back to the inputs of a circuit. Feedback is at the core of all sequential circuits.
- Field Programmable Gate Array (FPGA)** Class of PLDs that contain an array of more complex logic cells that can be very flexibly interconnected to implement high-level logic circuits.
- Field Programmable Logic Array (FPLA)** A PLD that uses both a programmable AND array and a programmable OR array.
- First-In, First-Out (FIFO) Memory** Semiconductor sequential-access memory in which data words are read out in the same order in which they were written in.
- 555 Timer** IC that can be wired to operate in several different modes, such as a one-shot and an astable multivibrator.
- Flash ADC** Type of analog-to-digital converter that has the highest operating speed available.
- Flash Memory** Nonvolatile memory IC that has the high-speed access and in-circuit erasability of EEPROMs but with higher densities and lower cost.
- Flip-Flop** Memory device capable of storing a logic level.
- Floating** A term used to describe a port or node in a circuit that is not driven by anything.
- Floating Bus** When all outputs connected to a data bus are in the Hi-Z state.
- Floating Input** Input signal that is left disconnected in a logic circuit.
- FOR Loop** See Iterative Loop.
- 4-to-10 Decoder** See BCD-to-Decimal Decoder.
- Frequency (F)** The number of cycles per unit time of a periodic waveform.
- Frequency Counter** Circuit that can measure and display a signal's frequency.
- Frequency Division** The use of flip-flop circuits to produce an output waveform whose frequency is equal to the input clock frequency divided by some integer value.
- Full Adder (FA)** Logic circuit with three inputs and two outputs. The inputs are a carry bit ( $C_{IN}$ ) from a

- previous stage, a bit from the augend, and a bit from the addend, respectively. The outputs are the sum bit and the carry-out bit ( $C_{OUT}$ ) produced by the addition of the bit from the addend with the bit from the augend and  $C_{IN}$ .
- Full-Custom ASIC** An application-specific integrated circuit (ASIC) that is completely designed and fabricated from fundamental elements of electronic devices such as transistors, diodes, resistors, and capacitors.
- Full-Scale Error** Term used by some digital-to-analog converter manufacturers to specify the accuracy of a digital-to-analog converter. It is defined as the maximum deviation of a digital-to-analog converter's output from its expected ideal value.
- Full-Scale Output** Maximum possible output value of a digital-to-analog converter.
- Function Generator** Circuit that produces different waveforms. It can be constructed using a ROM, a DAC, and a counter.
- Functionally Equivalent** When the logic functions performed by two different ICs are exactly the same.
- Fusible Link** Conducting material that can be made nonconducting (i.e., open) by passing too much current through it.
- Gate Array** An application-specific integrated circuit (ASIC) made up of hundreds of thousands of pre-fabricated basic gates that can be custom interconnected in the last stages of manufacture to form the desired digital circuit.
- GENERATE** A VHDL keyword used with the FOR construct to iteratively define multiple similar components and to interconnect them.
- Glitch** Momentary, narrow, spurious, and sharply defined change in voltage.
- Gray Code** A code that never has more than one bit changing when going from one state to another.
- GSI** Giga-scale integration (1,000,000 gates or more).
- Half Adder (HA)** Logic circuit with two inputs and two outputs. The inputs are a bit from the augend and a bit from the addend. The outputs are the sum bit produced by the addition of the bit from the addend with the bit from the augend and the resulting carry ( $C_{OUT}$ ) bit, which will be added to the next stage.
- Hard Disk** Rigid metal magnetic disk used for mass storage.
- Hardware Description Language (HDL)** A text-based method of describing digital hardware that follows a rigid syntax for representing data objects and control structures.
- Hexadecimal Number System** Number system that has a base of 16. Digits 0 through 9 plus letters A through F are used to express a hexadecimal number.
- Hierarchical Design** A method of designing a project by breaking it into constituent modules, each of which can be broken further into simpler constituent modules.
- Hierarchy** A group of tasks arranged in rank order of magnitude, importance, or complexity.
- High-Capacity PLD (HCPLD)** A PLD with thousands of logic gates and many programmable macrocell resources, along with very flexible interconnection resources.
- Hold Time ( $t_H$ )** Time interval immediately following the active transition of the clock signal during which the control input must be maintained at the proper level.
- Hybrid System** System that employs both analog and digital techniques.
- IEEE/ANSI** Institute of Electrical and Electronics Engineers/American National Standards Institute, both professional organizations that establish standards.
- IF/THEN** A control structure that evaluates a condition and performs an action if the condition is true or bypasses the action and continues on if the condition is false.
- Indeterminate** Of a logic voltage level, outside the required range of voltages for either logic 0 or logic 1.
- Index** Another name for the element number of any given bit in a bit array.
- Inhibit Circuits** Logic circuits that control the passage of an input signal through to the output.
- Input Term Matrix** Part of a programmable logic device that allows inputs to be selectively connected to or disconnected from internal logic circuitry.
- Input Unit** Part of a computer that facilitates the feeding of information into the computer's memory unit or ALU.
- Instructions** Binary codes that tell a computer what operation to perform. A program is made up of an orderly sequence of instructions.
- In-System Programmable (ISP)** A means by which a chip does not need to be removed from its circuit board in order to store programming information.
- INTEGER** In VHDL, the data object type representing a numeric value.
- Intellectual Property (IP)** An idea, design, or description of something that is claimed by the designers to be rightfully theirs. For example, complex circuits (like a microprocessor) described in HDL are considered intellectual property.
- Interfacing** Joining of dissimilar devices in such a way that they are able to function in a compatible and coordinated manner; connection of the output of a system to the input of a different system with different electrical characteristics.
- Interpolation Filtering** Another name for oversampling. Interpolation refers to intermediate values inserted into the digital signal to smooth out the waveform.
- Invert** Cause a logic level to go to the opposite state.
- INVERTER** Also referred to as the NOT circuit; logic circuit that implements the NOT operation. An INVERTER has only one input, and its output logic level is always the opposite of this input's logic level.
- Iterative Loop** A control structure that implies a repetitive operation and a stated number of iterations.

**Jam Transfer** See Asynchronous Transfer.

**J-K Excitation Table** Table showing the required *J* and *K* input conditions for each possible state transition for a single J-K flip-flop.

**Johnson Counter (twisted ring counter)** Shift register in which the inverted output of the last flip-flop is connected to the input of the first flip-flop.

**JTAG** Joint Test Action Group, which created a standard interface that allows access to the inner workings of an IC for testing, controlling, and programming purposes.

**Karnaugh Map (K Map)** Two-dimensional form of a truth table used to simplify a sum-of-products expression.

**Latch** Type of flip-flop; also, the action by which a logic circuit output captures and holds the value of an input.

**Latch-Up** Condition of dangerously high current in a CMOS IC caused by high-voltage spikes or ringing at device input and output pins.

**Latency** The inherent delay associated with reading data from a DRAM. It is caused by the timing requirements of supplying the row and column addresses, and the time for the data outputs to settle.

**LCD** Liquid-crystal display.

**Lead Pitch** The distance between the centers of adjacent pins on an IC.

**Least Significant Bit (LSB)** Rightmost bit (smallest weight) of a binary expressed quantity.

**Least Significant Digit (LSD)** Digit that carries the least weight in a particular number.

**Level Triggered** A digital circuit that will respond to control inputs whenever its trigger input is at its active level.

**LED** Light-emitting diode.

**Libraries** A collection of descriptions of commonly used hardware circuits that can be used as modules in a design file.

**Library of Parameterized Modules (LPM)** A group of generic functional logic blocks, with variable features (parameters) that can be included or omitted to meet a specific design requirement (e.g., number of bits, mod number, control options).

**Linear Buffer** A first-in, first-out memory system that fills at one rate and empties at another rate. After it is full, no data can be stored until data are read from the buffer. See also First In, First-Out (FIFO) Memory.

**Linearity Error** Term used by some digital-to-analog converter manufacturers to specify the device's accuracy. It is defined as the maximum deviation in step size from the ideal step size.

**Literals** In VHDL, a scalar value or bit pattern that is to be assigned to a data object.

**Load Operation** Transfer of data into a flip-flop, a register, a counter, or a memory location.

**Local Signal** See Buried Node.

**Logic Array Block (LAB)** A term Altera Corporation uses to describe building blocks of their CPLDs. Each LAB is similar in complexity to an SPLD.

**Logic Circuit** Any circuit that behaves according to a set of logic rules.

**Logic Elements** A term Altera Corporation uses to describe the PLD building blocks that are programmed as a RAM-based look-up table.

**Logic Function Generation** Implementation of a logic function directly from a truth table by means of a digital IC such as a multiplexer.

**Logic Level** State of a voltage variable. The states 1 (HIGH) and 0 (LOW) correspond to the two usable voltage ranges of a digital device.

**Logic Primitive** A circuit description of a fundamental component that is built into the Quartus II system of libraries.

**Logic Probe** Digital troubleshooting tool that senses and indicates the logic level at a particular point in a circuit.

**Logic Pulser** Testing tool that generates a short-duration pulse when actuated manually.

**Logic States** The logical condition (True/False, 1/0) of a bit or group of bits in a digital system.

**Look-Ahead Carry** Ability of some parallel adders to predict, without having to wait for the carry to propagate through the full adders, whether or not a carry bit ( $C_{OUT}$ ) will be generated as a result of the addition, thus reducing the overall propagation delays.

**Look-Up Table (LUT)** A way to implement a single logic function by storing the correct output logic state in a memory location that corresponds to each particular combination of input variables.

**Looping** Combining of adjacent squares in a Karnaugh map containing 1s for the purpose of simplification of a sum-of-products expression.

**Low-Power Schottky TTL (LS-TTL)** TTL subfamily that uses the identical Schottky TTL circuit but with larger resistor values.

**Low-Voltage Differential Signaling (LVDS)** A technology for driving high-speed data lines in low-voltage systems that uses two conductors and reverses the polarity to distinguish between HIGH and LOW.

**Low-Voltage Technology** New line of logic devices that operate from a nominal supply voltage of 3.3 V or less.

**LPM\_ADD\_SUB** A function available in the library that can add or subtract.

**LPM\_COUNTER** A counter function available in the library.

**LPM\_FF** A flip-flop function available in the library.

**LPM\_LATCH** A level triggered latch function available in the library.

**LPM\_MULT** A function available in the library that can multiply.

**LPM\_SHIFTREG** A shift register function available in the library.

**LSI** Large-scale integration (100 to 9999 gates).

- MAC** An abbreviation for Multiply Accumulate Unit, the hardware section of a DSP that multiplies a sample with a coefficient and then accumulates (sums) a running total of these products.
- MACHINE** An AHDL keyword used to create a state machine in a design file.
- Macrocell** A circuit made up of a group of basic digital components such as AND gates, OR gates, registers, and tristate control circuits that can be interconnected within a PLD via a program.
- Macrofunctions** A term used by Altera Corporation to describe the predefined hardware descriptions in their libraries that represent standard IC parts.
- Magnetic Disk Memory** Mass storage memory that stores data as magnetized spots on a rotating, flat disk surface.
- Magnetoresistive RAM (MRAM)** A memory technology that stores 1s and 0s by altering the polarity or “spin” of very small magnetic domains. The information is read back by measuring the amount of current that will flow through the magnetic domain (i.e., the spin polarity affects the resistance of the cell).
- Magnitude Comparator** Digital circuit that compares two input binary quantities and generates outputs to indicate whether the inputs are equal or, if not, which is greater.
- Main Memory** High-speed portion of a computer’s memory that holds the program and data the computer is currently working on. Also called *working memory*.
- Mask-Programmed ROM (MROM)** ROM that is programmed by the manufacturer according to the customer’s specifications. It cannot be erased or reprogrammed.
- Mass Storage** Storage of large amounts of data; not part of a computer’s internal memory.
- Maximum Clocking Frequency ( $f_{MAX}$ )** Highest frequency that may be applied to the clock input of a flip-flop and still have it trigger reliably.
- maxplus2 Function** The name Quartus II uses to describe library functions that emulate TTL standard parts from the 74XX series.
- Mealy Model** A state-machine model in which the output signals are controlled by combinational inputs as well as the state of the sequential circuit.
- Megafunction** A complex or high-level building block available in the Altera library, the name Quartus II uses to describe the versatile functions available in the Library of Parameterized Modules (LPM).
- Memory** Ability of a circuit’s output to remain at one state even after the input condition that caused that state is removed.
- Memory Cell** Device that stores a single bit.
- Memory Foldback** Redundant enabling of a memory device at more than one address range as a result of incomplete address decoding.
- Memory Map** Diagram of a memory system that shows the address range of all existing memory devices as well as available memory space for expansion.
- Memory Unit** Part of a computer that stores instructions and data received from the input unit, as well as results from the arithmetic/logic unit.
- Memory Word** Group of bits in memory that represents instructions or data of some type.
- Metastable States** An abnormal voltage that can be temporarily present on an output when timing constraints are violated.
- Microcomputer** Newest member of the computer family, consisting of microprocessor chip, memory chips, and I/O interface chips. In some cases, all of the aforementioned are in one single IC.
- Microcontroller** Small microcomputer used as a dedicated controller for a machine, a piece of equipment, or a process.
- Microprocessor (MPU)** LSI chip that contains the central processing unit (CPU).
- Minuend** Number from which the subtrahend is to be subtracted.
- MOD Number** Number of different states that a counter can sequence through; the counter’s frequency division ratio.
- Mode** The attribute of a port in a digital circuit that defines it as input, output, or bidirectional.
- Monostable Multivibrator** See One-Shot.
- Monotonicity** Property whereby the output of a digital-to-analog converter increases as the binary input is increased.
- Moore Model** A state-machine model in which the output signals are controlled only by the sequential circuit outputs.
- MOSFET** Metal-oxide-semiconductor field-effect transistor.
- Most Significant Bit (MSB)** Leftmost binary bit (largest weight) of a binary expressed quantity.
- Most Significant Digit (MSD)** Digit that carries the most weight in a particular number.
- MSI** Medium-scale integration (12 to 99 gates).
- Multiplexer (MUX)** Logic circuit that, depending on the status of its select inputs, will channel one of several data inputs to its output.
- Multiplexing** Process of selecting one of several input data sources and transmitting the selected data to a single output channel.
- Multistage Counter** Counter in which several counter stages are connected so that the output of one stage serves as the clock input of the next stage to achieve greater counting range or frequency division.
- NAND Flash Memory** A way of connecting flash memory cells (floating-gate transistors) that resembles a NAND gate circuit and results in very high density at the cost of random access.
- NAND Gate** Logic circuit that operates like an AND gate followed by an INVERTER. The output of a NAND gate is LOW (logic level 0) only if all inputs are HIGH (logic level 1).
- NAND Gate Latch** Flip-flop constructed from two cross-coupled NAND gates.

**Negation** Operation of converting a positive number to its negative equivalent, or vice versa. A signed binary number is negated by the 2's-complement operation.

**Negative Edge** See Negative Going Transition.

**Negative-Going Transition** When a clock goes from 1 to 0.

**Nested** To have one control structure embedded within another control structure.

**Nibble** A group of four bits.

**N-MOS (N-Channel Metal-Oxide-Semiconductor)** Integrated-circuit technology that uses N-channel MOSFETs as the principal circuit element.

**NODE** A keyword in AHDL used to declare an intermediate variable (data object) that is local to that subdesign.

**Noise** Spurious voltage fluctuations that may be present in the environment and cause digital circuits to malfunction.

**Noise Immunity** Circuit's ability to tolerate noise voltages on its inputs.

**Noise Margin** Quantitative measure of noise immunity.

**Nonretriggerable One-Shot** Type of one-shot that will not respond to a trigger input signal while in its quasi-stable state.

**Nonvolatile Memory** Memory that will keep storing its information without the need for electrical power.

**Nonvolatile RAM** Combination of a RAM array and an EEPROM or flash on the same IC. The EEPROM serves as a nonvolatile backup to the RAM.

**NOR Flash Memory** A way of connecting flash memory cells (floating gate transistors) that resembles a NOR gate circuit and results in high-speed random access at cost of space on the silicon wafer.

**NOR Gate** Logic circuit that operates like an OR gate followed by an INVERTER. The output of a NOR gate is LOW (logic level 0) when any or all inputs are HIGH (logic level 1).

**NOR Gate Latch** Flip-flop constructed from two cross-coupled NOR gates.

**NOT Circuit** See INVERTER.

**NOT Operation** Boolean algebra operation in which the overbar (̄) or the prime (') symbol is used to indicate the inversion of one or more logic variables.

**Objects** Various ways of representing data in the code of any HDL.

**Observation/Analysis** Process used to troubleshoot circuits or systems in order to predict the possible faults before ever picking up a troubleshooting instrument. When this process is used, the troubleshooter must understand the circuit operation, observe the symptoms of the failure, and then reason through the operation.

**Octal Number System** Number system that has a base of 8; digits from 0 to 7 are used to express an octal number.

**Octets** Groups of eight 1s that are adjacent to each other within a Karnaugh map.

**Offset Error** Deviation from the ideal 0 V at the output of a digital-to-analog converter when the input is

all 0s. In reality, there is a very small output voltage for this situation.

**1-of-10 Decoder** See BCD-to-Decimal Decoder.

**1's-Complement Form** Result obtained when each bit of a binary number is complemented.

**One-Shot (os)** Circuit that belongs to the flip-flop family but that has only one stable state (normally  $Q = 0$ ).

**One-Time Programmable (OTP)** A broad category of programmable components that are programmed by permanently altering the connections (e.g., melting a fuse element).

**Open-Collector Output** Type of output structure of some TTL circuits in which only one transistor with a floating collector is used.

**Optical Disk Memory** Class of mass memory devices that uses a laser beam to write onto and read from a specially coated disk.

**OR Gate** Digital circuit that implements the OR operation. The output of this circuit is HIGH (logic level 1) if any or all of its inputs are HIGH.

**OR Operation** Boolean algebra operation in which the symbol + is used to indicate the ORing of two or more logic variables. The result of the OR operation will be HIGH (logic level 1) if one or more variables are HIGH.

**Output Logic Macrocell (OLMC)** A group of logic elements (gates, multiplexers, flip-flops, buffers) in a PLD that can be configured in various ways.

**Output Unit** Part of a computer that receives data from the memory unit or ALU and presents it to the outside world.

**Overflow** When in the process of adding signed binary numbers, the magnitude of the sum is too large for the word size and produces an incorrect sign.

**Override Inputs** Synonymous with "asynchronous inputs."

**Oversampling** Inserting data points between sampled data in a digital signal to make it easier to filter out the rough edges of the waveform coming out of the DAC.

**PACKAGE** A VHDL keyword used to define a set of global elements that are available to other modules.

**Parallel Adder** Digital circuit made from full adders and used to add all of the bits from the addend and the augend together simultaneously.

**Parallel Counter** See Synchronous Counter.

**Parallel Data Transfer** Operation by which several bits of data are transferred simultaneously into a counter or a register.

**Parallel In/Parallel Out Register** Type of register that can be loaded with parallel data and has parallel outputs available.

**Parallel In/Serial Out Register** Type of register that can be loaded with parallel data and has only one serial output.

**Parallel Load** See Parallel Data Transfer.

**Parallel-to-Serial Conversion** Process by which all data bits are presented simultaneously to a circuit's input and then transmitted one bit at a time to its output.

- Parallel Transmission** Simultaneous transfer of all bits of a binary number from one place to another.
- Parity Bit** Additional bit that is attached to each code group so that the total number of 1s being transmitted is always even (or always odd).
- Parity Checker** Circuit that takes a set of data bits (including the parity bit) and checks to see if it has the correct parity.
- Parity Generator** Circuit that takes a set of data bits and produces the correct parity bit for the data.
- Parity Method** Scheme used for error detection during the transmission of data.
- Percentage Resolution** Ratio of the step size to the full-scale value of a digital-to-analog converter. Percentage resolution can also be defined as the reciprocal of the maximum number of steps of a digital-to-analog converter.
- Period ( $T$ )** The amount of time required for one complete cycle of a periodic event or waveform.
- Periodic** A cycle that repeats itself regularly in time and form.
- Pin-Compatible** When the corresponding pins on two different ICs have the same functions.
- Pipelined ADC** A conversion strategy that uses high-speed flash converters in two or more stages, each of which determines a portion of the binary result starting with the most significant stage.
- Pixel** Small dots of light that make up a graphical image on a display.
- P-MOS (P-channel Metal Oxide Semiconductor)** Integrated-circuit technology that uses P-channel MOSFETs as the principal circuit element.
- PORT MAP** A VHDL keyword that precedes the list of connections specified between components.
- Positional-Value System** System in which the value of a digit depends on its relative position.
- Positive Edge** Positive Going Transition.
- Positive-Going Transition (PGT)** When a clock signal changes from a logic 0 to a logic 1.
- Power-Down** Operating mode in which a chip is disabled and draws much less power than when it is fully enabled.
- Power-Supply Decoupling** Connection of a small RF capacitor between ground and  $V_{CC}$  near each TTL integrated circuit on a circuit board.
- Prescaler** A counter circuit that takes base reference frequency and scales it by dividing the frequency down to a rate required by the system.
- Present State–Next State Table** A table that lists each possible present state of a sequential (counter) circuit and identifies the corresponding next state.
- PRESET** Asynchronous input used to set  $Q = 1$  immediately.
- Presetable Counter** Counter that can be preset to any starting count either synchronously or asynchronously.
- Primitive** One of the basic functional blocks used to design circuits with Quartus II software.
- Priority Encoder** Special type of encoder that senses when two or more inputs are activated simultaneously and then generates a code corresponding to the highest-numbered input.
- PROCESS** A VHDL keyword that defines the beginning of a block of code that describes a circuit that must respond whenever certain signals (in the sensitivity list) change state. All sequential statements must occur inside a process.
- Product-of-Sums (POS) Form** Logic expression consisting of two or more OR terms (sums) that are ANDed together.
- Program** Sequence of binary-coded instructions designed to accomplish a particular task by a computer.
- Programmable Array Logic (PAL)** Class of programmable logic devices. Its AND array is programmable, whereas its OR array is hard-wired.
- Programmable Interconnect Array (PIA)** A term Altera Corporation uses to describe the resources used to connect the LABs with each other and also with the input/output modules.
- Programmable Logic Array (PLA)** Class of programmable logic devices. Both its AND and its OR arrays are programmable. Also called a *field programmable logic array (FPLA)*.
- Programmable Logic Device (PLD)** IC that contains a large number of interconnected logic functions. The user can program the IC for a specific function by selectively breaking the appropriate interconnections.
- Programmable Output Polarity** Feature of many PLDs whereby an XOR gate with a polarity fuse gives the designer the option of inverting or not inverting a device output.
- Programmable ROM (PROM)** ROM that can be electrically programmed by the user. It cannot be erased and reprogrammed.
- Programmer** A fixture used to apply the proper voltages to PLD and PROM chips in order to program them.
- Programming** The act of storing 1s and 0s in a programmable logic device to configure its behavioral characteristics.
- Propagation Delays ( $t_{PLH}/t_{PHL}$ )** Delay from the time a signal is applied to the time when the output makes its change.
- Pull-Down Transistor** See Current-Sinking Transistor.
- Pull-Up Transistor** See Current-Sourcing Transistor.
- Pulse** A momentary change of logic state that represents an event to a digital system.
- Pulse-Steering Circuit** A logic circuit that can be used to select the destination of an input pulse, depending on the logic levels present at the circuit's inputs.
- Quantization Error** Error caused by the nonzero resolution of an analog-to-digital converter. It is an inherent error of the device.
- Quasi-Stable State** State to which a one-shot is temporarily triggered (normally  $Q = 1$ ) before returning to its stable state (normally  $Q = 0$ ).
- R/2R Ladder DAC** Type of digital-to-analog converter whose internal resistance values span a range of only 2 to 1.



- Random-Access Memory (RAM)** Memory in which the access time is the same for any location.
- RAS (Row Address Strobe)** Signal used to latch the row address into a DRAM chip.
- RAS-Only Refresh** Method for refreshing DRAM in which only row addresses are strobed into the DRAM using the RAS input.
- Read** Term used to describe the condition when the CPU is receiving data from another element.
- Read-Only Memory (ROM)** Memory device designed for applications where the ratio of read operations to write operations is very high.
- Read Operation** Operation in which a word in a specific memory location is sensed and possibly transferred to another device.
- Read/Write Memory (RWM)** Any memory that can be read from and written into with equal ease.
- Refresh Counter** Counter that keeps track of row addresses during a DRAM refresh operation.
- Refreshing** Process of recharging the cells of a dynamic memory.
- Register** Group of flip-flops capable of storing data.
- RESET** Term synonymous with "CLEAR."
- RESET State** The  $Q = 0$  state of a flip-flop.
- Resolution** In a digital-to-analog converter, smallest change that can occur in the output for a change in digital input; also called *step size*. In an analog-to-digital converter, smallest amount by which the analog input must change to produce a change in the digital output.
- Retriggerable One-Shot** Type of one-shot that will respond to a trigger input signal while in its quasistable state.
- Ring Counter** Shift register in which the output of the last flip-flop is connected to the input of the first flip-flop.
- Ripple Counter** See Asynchronous Counter.
- Sample-and-Hold (S/H) Circuit** Type of circuit that utilizes a unity-gain buffer amplifier in conjunction with a capacitor to keep the input stable during an analog-to-digital conversion process.
- Sampling** Acquiring and digitizing a data point from an analog signal at a given instant of time.
- Sampling Frequency ( $F_s$ )** The rate at which an analog signal is digitized (samples per second).
- Sampling Interval** Time window during which a frequency counter samples and thereby determines the unknown frequency of a signal.
- SBD** Schottky barrier diode used in all Schottky TTL series.
- Schematic Capture** A computer program that can interpret graphic symbols and signal connections and translate them into logical relationships.
- Schmitt Trigger Circuit** Digital circuit that accepts a slow-changing input signal and produces a rapid, oscillation-free transition at the output.
- Schottky TTL** TTL subfamily that uses the basic TTL standard circuit except that it uses a Schottky barrier diode (SBD) connected between the base and the collector of each transistor for faster switching.
- Selected Signal Assignment** A VHDL statement that allows a data object to be assigned a value from one of several signal sources depending on the value of an expression.
- Self-Correcting Counter** A counter that always progresses to its intended sequence, regardless of its initial state.
- Sensitivity List** The list of signals used to invoke the sequence of statements in a PROCESS.
- Sequential** Occurring one at a time in a certain order. In HDL, the circuits that are generated by sequential statements behave differently, depending on the order of the statements in the code.
- Sequential-Access Memory (SAM)** Memory in which the access time will vary depending on the storage location of the data.
- Sequential Circuit** A logic circuit whose outputs can change states in synchronism with a periodic clock signal. The new state of an output may depend on its current state as well as the current states of other outputs and the logic levels applied to control inputs.
- Serial Data Transfer** Transfer of data from one place to another one bit at a time.
- Serial In/Parallel Out** Type of register that can be loaded with data serially and has parallel outputs available.
- Serial In/Serial Out** Type of register that can be loaded with data serially and has only one serial output.
- Serial Transmission** Transfer of binary information from one place to another a bit at a time.
- SET** An input to a latch or FF used to make  $Q = 1$ .
- Set** A grouping of concatenated variables or signals.
- SET State** The  $Q = 1$  state of a flip-flop.
- Settling Time** Amount of time that it takes for the output of a digital-to-analog converter to go from 0 to within one-half step size of its full-scale value as the input is changed from all 0s to all 1s.
- Setup Time ( $t_s$ )** Time interval immediately preceding the active transition of the clock signal during which the control input must be maintained at the proper level.
- Shift Register** Digital circuit that accepts binary data from some input source and then shifts these data through a chain of flip-flops one bit at a time.
- Sigma ( $\Sigma$ )** Greek letter that represents addition and is often used to label the sum output bits of a parallel adder.
- Sigma/Delta Modulation** Method of sampling an analog signal and converting its data points into a bit stream of serial data.
- Sign Bit** Binary bit that is added to the leftmost position of a binary number to indicate whether that number represents a positive or a negative quantity.
- Sign-Magnitude System** A system for representing signed binary numbers where the most significant bit represents the sign of the number and the remaining bits represent the true binary value (magnitude).
- Simple PLD (SPLD)** A PLD with a few hundred logic gates and possibly a few programmable macrocells available.

- Simulator** Computer program that calculates the correct output states of a logic circuit based on a description of the logic circuit and on the current inputs.
- Spike** *See* Glitch.
- S-R Latch** A circuit with SET(S) and RESET(R) inputs and Q output. The Q output remembers the last active command that was asserted on its inputs.
- SSI** Small-scale integration (fewer than 12 gates).
- Staircase Test** Process by which a digital-to-analog converter's digital input is incremented and its output monitored to determine whether or not it exhibits a staircase format.
- Staircase Waveform** Type of waveform generated at the output of a digital-to-analog converter as its digital input signal is incrementally changed.
- Standard Cell ASIC** An application-specific integrated circuit (ASIC) made of predesigned logic blocks from a library of standard cell designs that are interconnected during the system design stage and then fabricated on a single IC.
- Standard Logic** The large assortment of basic digital IC components available in various technologies as MSI, SSI chips.
- State Machines** A sequential circuit that advances through several defined states.
- State Table** A table whose entries represent the sequence of individual FF states (i.e., 0 or 1) for a sequential binary circuit.
- State Transition Diagram** A graphic representation of the operation of a sequential binary circuit, showing the sequence of individual FF states and conditions needed for transitions from one state to the next.
- Static Accuracy Test** Test in which a fixed binary value is applied to the input of a digital-to-analog converter and the analog output is accurately measured. The measured result should fall within the expected range specified by the digital-to-analog converter's manufacturer.
- Static RAM (SRAM)** Semiconductor RAM that stores information in flip-flop cells that do not have to be periodically refreshed.
- STD\_LOGIC** In VHDL, a data type defined as an IEEE standard. It is similar to the BIT type, but it offers more possible values than just 1 or 0.
- STD\_LOGIC\_VECTOR** In VHDL, a data type defined as an IEEE standard. It is similar to the BIT\_VECTOR type, but it offers more possible values than just 1 or 0 for each element.
- Step Size** *See* Resolution.
- Straight Binary Coding** Representation of a decimal number by its equivalent binary number.
- Strobe** Another name for an enable input usually used to latch a value into a register.
- Strobing** Technique often used to eliminate decoding spikes.
- Structural Level of Abstraction** A technique for describing a digital circuit that focuses on connecting ports of modules with signals.
- SUBDESIGN** Keyword in AHDL used to begin a circuit description.
- Substrate** Piece of semiconductor material that is part of the building block of any digital IC.
- Subtrahend** Number that is to be subtracted from a minuend.
- Successive-Approximation ADC** Type of analog-to-digital converter in which an internal parallel register and complex control logic are used to perform the conversion. The conversion time for this type of analog-to-digital converter is always the same regardless of the value of the input analog signal.
- Sum-of-Products (SOP) Form** Logic expression consisting of two or more AND terms (products) that are ORed together.
- Supercomputers** Computers with the greatest speed and computational power.
- Surface Mount** A method of manufacturing circuit boards whereby ICs are soldered to conductive pads on the surface of the board.
- Synchronous Control Inputs** *See* Control Inputs.
- Synchronous Counter** Counter in which all of the flip-flops are clocked simultaneously.
- Synchronous Systems** Systems in which the circuit outputs can change states only on the transitions of a clock.
- Synchronous Transfer** Data transfer performed by using the synchronous and clock inputs of a flip-flop.
- Syntax** The rules defining keywords and their arrangement, usage, punctuation, and format for a given language.
- Test Vector** Sets of inputs used to test a PLD design before the PLD is programmed.
- Timing Diagram** Depiction of logic levels as related to time.
- Toggle Mode** Mode in which a flip-flop changes states for each clock pulse.
- Toggling** Process of changing from one binary state to the other.
- Top-Down** A design method that starts at the overall system level and then defines a hierarchy of modules.
- Totem-Pole Output** Term used to describe the way in which two bipolar transistors are arranged at the output of most TTL circuits.
- Transducer** Device that converts a physical variable to an electrical variable (for example, a photocell or a thermocouple).
- Transient State** Bit combinations that appear for a very short time as signals propagate through a digital system during a change from one steady state to another.
- Transmission Gate** *See* Bilateral Switch.
- Transparent** Of a D latch, operating so that the Q output follows the D input.
- Trigger** Input signal to a flip-flop or one-shot that causes the output to change states depending on the conditions of the control signals.
- Tristate** Type of output structure that allows three types of output states: HIGH, LOW, and high-impedance (Hi-Z).

**Truth Table** Logic table that depicts a circuit's output response to the various combinations of the logic levels at its inputs.

**TTL (Transistor/Transistor Logic)** Integrated-circuit technology that uses the bipolar transistor as the principal circuit element.

**2's-Complement Form** Result obtained when a 1 is added to the least significant bit position of a binary number in the 1's-complement form.

**Type** The attribute of a variable in a computer-based language that defines its size and how it can be used.

**ULSI** Ultra-large-scale integration (100,000 or more gates).

**Unasserted** Term used to describe the state of a logic signal; synonymous with "inactive."

**Undersampling** Acquiring samples of a signal at a rate less than twice the highest frequency contained in the signal.

**Unipolar ICs** Integrated digital circuits in which unipolar field-effect transistors (MOSFETs) are the main circuit elements.

**Up Counter** Counter that counts upward from 0 to a maximum count.

**Up/Down Counter** Counter that can count up or down depending on how its inputs are activated.

**VARIABLE** A keyword in AHDL used to begin a section of the code that defines the names and types of data objects and library primitives. A keyword used in VHDL to declare a local data object within a PROCESS.

**Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL)** A hardware description language developed by the Department of

Defense to document, simulate, and synthesize complex digital systems.

**VLSI** Very large-scale integration (10,000 to 99,999 gates).

**Volatile Memory** Memory requiring electrical power to keep information stored.

**Voltage-Controlled Oscillator (VCO)** Circuit that produces an output signal with a frequency proportional to the voltage applied to its input.

**Voltage-Level Translator** Circuit that takes one set of input voltage levels and translates it to a different set of output levels.

**Voltage-to-Frequency ADC** Type of analog-to-digital converter that converts the analog voltage to a pulse frequency that is then counted to produce a digital output.

**Weighted Average** An average calculation of a group of samples that assigns a different weight (between 0.0 and 1.0) to each sample.

**Wired-AND** Term used to describe the logic function created when open-collector outputs are tied together.

**Word** Group of bits that represent a certain unit of information.

**Word Size** Number of bits in the binary words that a digital system operates on.

**WRITE** Term used to describe the condition when the CPU is sending data to another element.

**Write Operation** Operation in which a new word is placed into a specific memory location.

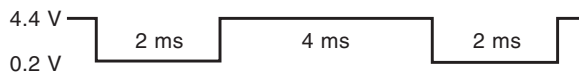
**ZIF Socket** Zero-insertion-force IC socket.



# ANSWERS TO SELECTED PROBLEMS\*

## CHAPTER 1

1-5. (a) Door\_open (c) Passenger\_seated  
1-6.



1-7.  $T = 7 \text{ ms}$ ;  $F = 143 \text{ Hz}$ ;  $DC = 71.4\%$   
1-11. (a) and (e) are digital; (b), (c), and (d) are analog  
1-13. (a) 25 (b) 9.5625 (c) 1241.6875  
1-15. 000, 001, 010, 011, 100, 101, 110, 111  
1-17. 1023  
1-19. Nine bits  
1-22. (a) and  $2^N - 1 = 15$  and  $N = 4$ ; therefore, four lines are required for parallel transmission.  
(b) Only one line is required for serial transmission.

## CHAPTER 2

2-1. (a) 22 (c) 2313 (e) 255 (g) 983  
(i) 38 (k) 59  
2-2. (a) 100101 (c) 10111101 (e) 1001101  
(g) 11001101 (i) 111111111  
2-3. (a) 255  
2-4. (a) 1859 (c) 14333 (e) 357 (g) 2047  
2-5. (a) 3B (c) 397 (e) 303 (g) 10000  
2-6. (a) 11101000011 (c) 1101111111101  
(e) 101100101 (g) 01111111111  
2-7. (a) 16 (c) 909 (e) FF (g) 3D7  
2-9.  $2133_{10} = 855_{16} = 100001010101_2$   
2-11. (a) 146 (c) 12,634 (e) 15 (g) 704  
2-12. (a) 4B (c) 800 (e) 1C4D (g) 6413

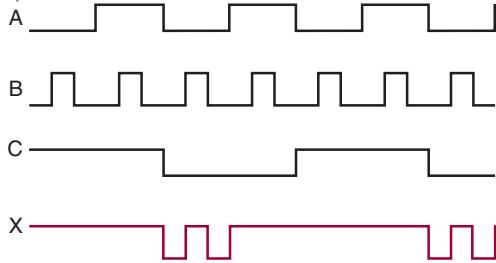
2-15.  $4095_{10}$   
2-16. (a) 10010010 (c) 001101111111101  
(e) 1111 (g) 1011000000  
2-17. 280, 281, 282, 283, 284, 285, 286, 287, 288, 289,  
28A, 28B, 28C, 28D, 28E, 28F, 290, 291, 292, 293,  
294, 295, 296, 297, 298, 299, 29A, 29B, 29C, 29D,  
29E, 29F, 2A0  
2-19. (a) 01000111 (c) 000110000111  
(e) 00010011 (g) 10001001011000100111  
(i) 01110010 (k) 01100001  
2-21. (a) 9752 (c) 695 (e) 492  
2-22. (a) 64 (b) FFFFFFFF (c) 999,999  
2-25. 78, A0, BD, A0, 33, AA, F9  
2-26. (a) BEN SMITH  
2-27. (a) 101110100 (parity bit on the left)  
(c) 11000100010000100 (e) 0000101100101  
2-28. (a) No single-bit error (b) Single-bit error  
(c) Double error (d) No single-bit error  
2-30. (a) 10110001001 (b) 11111111 (c) 209  
(d) 59,943 (e) 9C1 (f) 010100010001 (g) 565  
(h) 10DC (i) 1961 (j) 15,900 (k) 640  
(l) 952B (m) 100001100101 (n) 947  
(o) 10001100101 (p) 101100110100 (q) 1001010  
(r) 01011000 (BCD)  
2-31. (a) 100101 (b) 00110111 (c) 25  
(d) 0110011 0110111  
2-32. (a) Hex (b) 2 (c) Digit (d) Gray  
(e) Parity; single-bit errors (f) ASCII  
(g) Hex (h) Byte  
2-33. (a) 1000  
2-34. (a)  $1011_2$   
2-35. (a) 777A (c) 1000 (e) A00  
2-36. (a) 7778 (c) OFFE (e) 9FE

\*Note: Solutions to some problems are provided on the website, which can be accessed at [www.pearsonhighered.com/electronics](http://www.pearsonhighered.com/electronics).

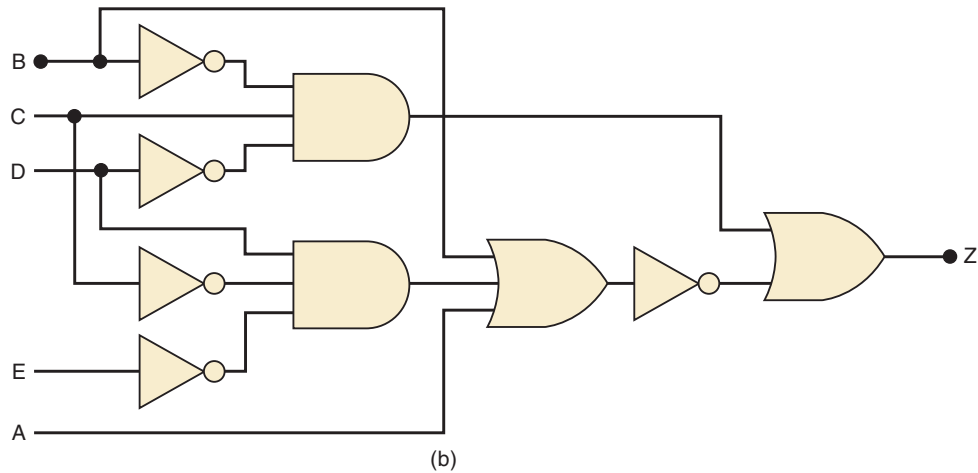
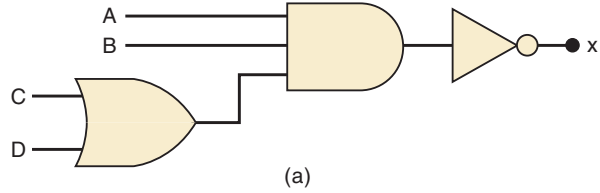
- 2-37. (a) 1,048,576 (b) Five (c) 000FF  
 (d)  $2k = 0-2047_{10} = 0-7FF_{16}$   
 2-39. Eight

**CHAPTER 3**

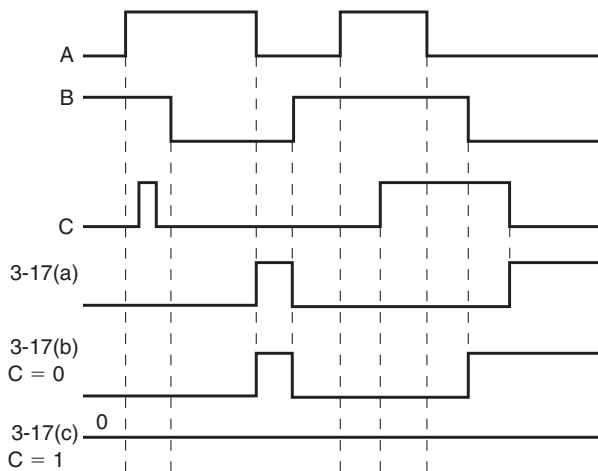
3-1. (a)



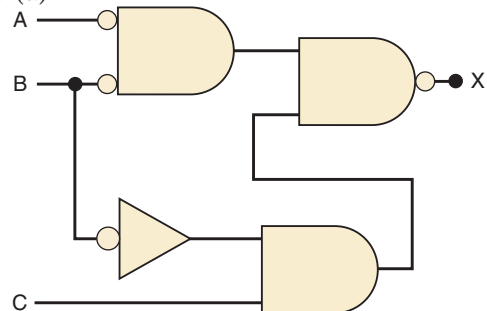
- 3-1. (c)  $x$  will be a constant HIGH.  
 3-6. (a)  $x$  is HIGH only when  $A$ ,  $B$ , and  $C$  are all HIGH.  
 3-7. Change the OR gate to an AND gate.  
 3-8. OUT is always LOW.  
 3-12. (a)  $x = (\overline{A} + \overline{B})BC$ .  $x$  is HIGH only when  $ABC = 111$   
 3-13.  $X$  is HIGH for all cases where  $E = 1$  except for  $EDCBA = 10101, 10110, \text{ and } 10111$ .  
 3-14. (a)  $x = D \cdot (\overline{AB} + \overline{C}) + E$   
 3-16.



3-17.



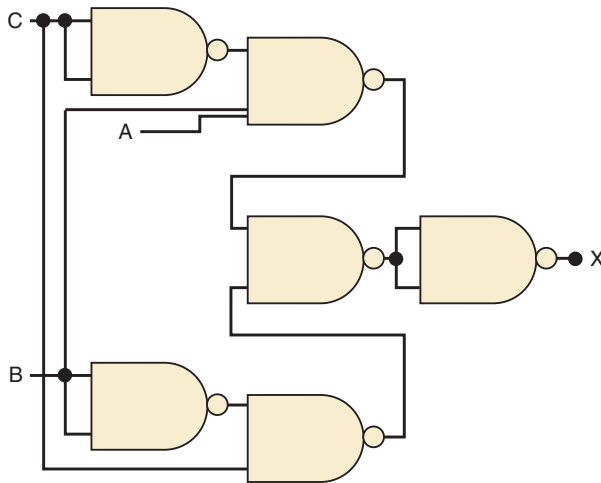
- 3-19.  $x = \overline{(A + B)} \cdot (B + C)$   
 $x = 0$  only when  $A = B = 0, C = 1$ .  
 3-23. (a) 1 (b)  $A$  (c) 0 (d)  $C$  (e) 0  
 (f)  $D$  (g)  $\overline{D}$  (h) 1 (i)  $G$  (j)  $y$   
 3-24. (a)  $MP\overline{N} + \overline{M}PN$   
 3-26. (a)  $A + \overline{B} + C$  (c)  $\overline{A} + \overline{B} + CD$  (e)  $A + B$   
 (g)  $\overline{A} + B + \overline{C} + \overline{D}$   
 3-27.  $A + B + \overline{C}$   
 3-32. (a)  $W = 1$  when  $T = 1$  and either  $P = 1$  or  $R = 0$ .  
 3-35. (a) NOR (b) AND (c) NAND  
 3-37. (a)



- 3-40.  $X$  will go HIGH when  $E = 1$ , or  $D = 0$ , or  $B = C = 0$ , or when  $B = 1$  and  $A = 0$ .  
 3-41. (a) HIGH (b) LOW  
 3-43. LIGHT = 0 when  $A = B = 0$  or  $A = B = 1$ .  
 3-45. (a) False (b) True (c) False (d) True  
 (e) False (f) False (g) True (h) False  
 (i) True (j) True  
 3-47. See website for solution.  
 3-49. Put INVERTERS on the  $A_7, A_5, A_4, A_2$  inputs to the 74HC30.  
 3-51. Requires six 2-input NAND gates.

**CHAPTER 4**

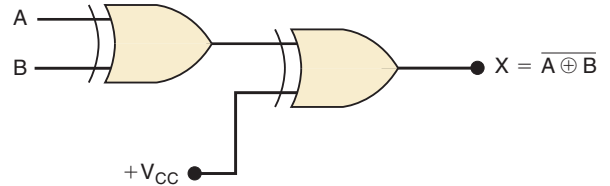
- 4-1. (a)  $C\bar{A} + CB$  (b)  $\bar{Q}R + Q\bar{R}$  (c)  $C + \bar{A}$   
 (d)  $\bar{R}\bar{S}\bar{T}$  (e)  $BC + \bar{B}(\bar{C} + A)$   
 (f)  $BC + \bar{B}(\bar{C} + A)$  or  $BC + \bar{B}\bar{C} + \bar{A}\bar{B}$   
 (g)  $\bar{D} + \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C$   
 (h)  $x = ABC + AB\bar{D} + \bar{A}BD + \bar{B}\bar{C}\bar{D}$   
 4-3.  $MN + Q$   
 4-4. One solution:  $\bar{x} = \bar{B}C + \bar{A}BC$ . Another:  $x = \bar{A}B + \bar{B}\bar{C}$ . Another:  $BC + \bar{B}\bar{C} + \bar{A}\bar{C}$   
 4-7.  $x = \bar{A}_3(A_2 + A_1A_0)$   
 4-9.



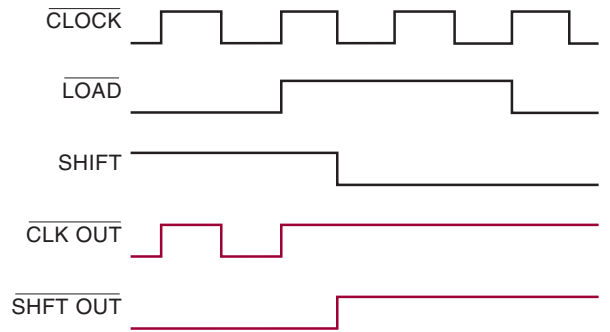
- 4-11. (a)  $x = \bar{A}\bar{C} + \bar{B}C + AC\bar{D}$

	$\bar{C}\bar{D}$	$\bar{C}D$	$CD$	$C\bar{D}$
$\bar{A}\bar{B}$	1	1	1	1
$\bar{A}B$	1	1		
$AB$				1
$A\bar{B}$			1	1

- 4-14. (a)  $x = BC + \bar{B}\bar{C} + AC$ ; or  $x = BC + \bar{B}\bar{C} + \bar{A}\bar{B}$   
 (c) One possible looping:  
 $x = \bar{A}BD + \bar{A}BC + \bar{A}BD + \bar{B}\bar{C}\bar{D}$ , another one is:  
 $x = \bar{A}BC + \bar{A}BD + \bar{A}C\bar{D} + \bar{B}\bar{C}\bar{D}$   
 4-15.  $x = \bar{A}_3A_2 + \bar{A}_3A_1A_0$   
 4-16. (a) Best solution:  $x = \bar{B}\bar{C} + AD$   
 4-17.  $x = \bar{S}_1\bar{S}_2 + \bar{S}_1\bar{S}_3 + \bar{S}_3\bar{S}_4 + \bar{S}_2\bar{S}_3 + \bar{S}_2\bar{S}_4$   
 4-18.  $z = \bar{B}C + \bar{A}B\bar{D}$   
 4-21.  $A = 0, B = C = 1$   
 4-23. One possibility is shown below.



- 4-24. Four XNORs feeding an AND gate  
 4-26. Four outputs where  $z_3$  is the MSB  
 $z_3 = y_1y_0x_1x_0$   $z_2 = y_1x_1(\bar{y}_0 + \bar{x}_0)$   
 $z_1 = y_0x_1(\bar{y}_1 + \bar{x}_0) + y_1x_0(\bar{y}_0 + \bar{x}_1)$   
 $z_0 = y_0x_0$   
 4-28.  $x = \bar{A}B(\bar{C} \oplus \bar{D})$   
 4-30.  $N-S = \bar{C}\bar{D}(A + B) + \bar{A}B(\bar{C} + \bar{D})$ ;  $E-W = \bar{N}-\bar{S}$   
 4-33. (a) No (b) No  
 4-35.  $x = A + BCD$   
 4-38.  $z = x_1x_0y_1y_0 + x_1\bar{x}_0y_1\bar{y}_0 + \bar{x}_1x_0\bar{y}_1y_0 + \bar{x}_1\bar{x}_0\bar{y}_1\bar{y}_0$   
 No pairs, quads, or octets  
 4-40. (a) Indeterminate (b) 1.4–1.8 V (c) See below.



- 4-43. Possible faults: faulty  $V_{CC}$  or ground on Z2; Z2-1 or Z2-2 open internally or externally; Z2-3 internally open  
 4-44. Yes: (c), (e), (f). No: (a), (b), (d), (g).  
 4-46. Z2-6 and Z2-11 shorted together  
 4-48. Most likely faults: faulty ground or  $V_{CC}$  on Z1; Z1 plugged in backward; Z1 internally damaged  
 4-49. Possible faults: Z2-13 shorted to  $V_{CC}$ ; Z2-8 shorted to  $V_{CC}$ ; broken connection to Z2-13; Z2-3, Z2-6, Z2-9, or Z2-10 shorted to ground  
 4-50. (a) T (b) T (c) F (d) F (e) T  
 4-54. Boolean equation; truth table; schematic diagram  
 4-56. (a) AHDL: gadgets[7..0] :OUTPUT;  
 VHDL: gadgets :OUT\_BIT\_VECTOR  
 (7 DOWNT0 0);

```

4-57. (a) AHDL: H"98"    B"10011000" 152
           VHDL: X"98"    B"10011000" 152
4-58.   AHDL: outbits[3] = inbits[1];
           outbits[2]   = inbits[3];
           outbits[1]   = inbits[0];
           outbits[0]   = inbits[2];
           VHDL: outbits(3) <= inbits(1);
           outbits(2) <= inbits(3);
           outbits(1) <= inbits(0);
           outbits(0) <= inbits(2);
    
```

```

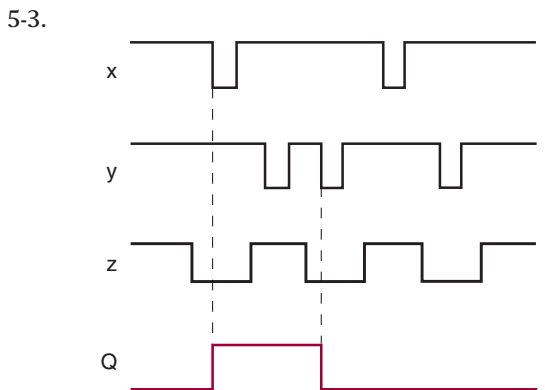
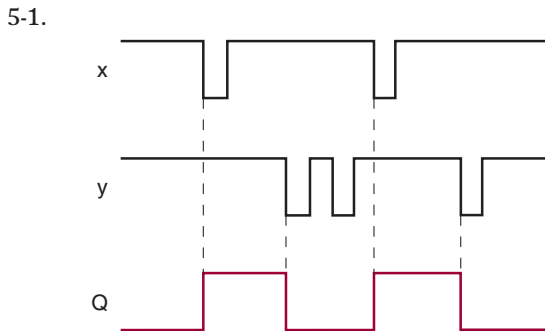
4-60.
BEGIN
  IF digital_value[] < 10 THEN
    z = VCC; --output a 1
  ELSE z = GND; --output a 0
  END IF;
END;
    
```

```

4-62.
PROCESS (digital_value)
  BEGIN
    IF (digital_value < 10) THEN
      z <= '1';
    ELSE
      z <= '0';
    END IF;
  END PROCESS
    
```

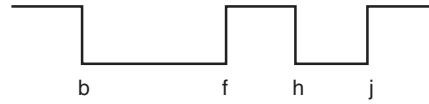
END PROCESS  
 4-65. S=!P#Q&R  
 4-68. (a) 00 to EF

**CHAPTER 5**

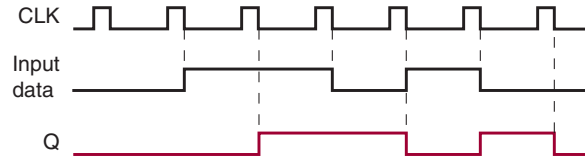


5-6. Z1-4 stuck HIGH

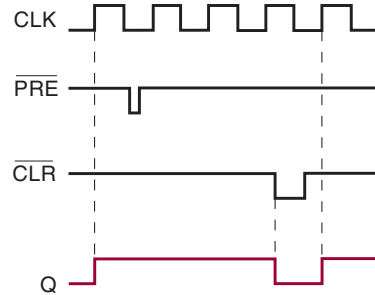
5-9. Assume  $Q = 0$  initially.  
 For PGT FF:  $Q$  will go HIGH on first PGT of CLK.  
 For NGT FF:  $Q$  will go HIGH on first NGT of CLK,  
 LOW on second NGT, and HIGH again on fourth NGT.  
 5-11.



5-12. (a) 5-kHz square wave  
 5-14. (a)

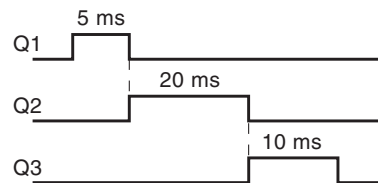


5-16. 500-Hz square wave  
 5-21.



5-23. (a)  $T_{plh\ max} = 16\ ns$  (b)  $T_{su} = 15\ ns$

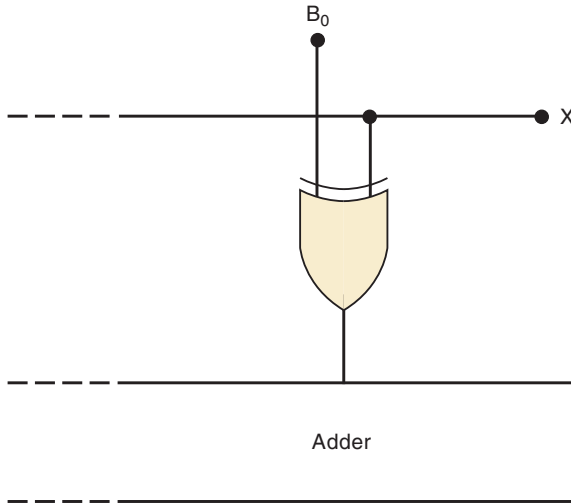
5-25. Connect A to J,  $\bar{A}$  to K.  
 5-27. (a) Connect X to J,  $\bar{X}$  to K. (b) Use arrangement of Figure 5-39.  
 5-29. Connect  $X_0$  to D input of  $X_2$ .  
 5-30. (a) 101; 011; 000  
 5-33. (a) 10 (b) 1953 Hz (c) 1024 (d) 12  
 5-36. Put INVERTERS on  $A_8, A_{11},$  and  $A_{14}$ .  
 5-41.



5-43. (a)  $A_1$  or  $A_2$  must be LOW when a PGT occurs at B.  
 5-45. One possibility is  $R = 1\ k\Omega$  and  $C = 80\ nF$ .  
 5-50. (a) No (b) Yes  
 5-51. (a) Yes  
 5-53. (a) No (b) No  
 5-55. (a) No (b) No (c) Yes  
 5-56. (a) NAND and NOR latch (b) J-K (c) D latch  
 (d) D flip-flop  
 5-59. See website for solution.  
 5-61. See website for solution.  
 5-66. See website for solution.

**CHAPTER 6**

- 6-1. (a) 10101 (b) 10010 (c) 1111.0101 (j) 11  
 (k) 101 (l) 111.001  
 6-2. (a) 00100000 (including sign bit) (b) 11110010  
 (c) 00111111 (d) 10011000 (e) 01111111  
 (f) 10000001 (g) 01011001 (h) 11001001  
 6-3. (a) +13 (b) -3 (c) +123 (d) -103  
 (e) +127  
 6-5.  $-16_{10}$  to  $15_{10}$   
 6-6. (a) 01001001; 10110111 (b) 11110100; 00001100  
 6-7. (a) 0 to 1023; -512 to +511  
 6-9. (a) 00001111 (b) 11111101 (c) 11111011  
 (d) 10000000 (e) 00000001  
 6-11. (a) 100011 (b) 1111001  
 6-12. (a) 11 (b) 111  
 6-13. (a) 10010111 (BCD) (b) 10010101 (BCD)  
 (c) 010100100111 (BCD)  
 6-14. (a) 6E24 (b) 100D (c) 18AB  
 6-15. (a) 0EFE (b) 229 (c) 02A6  
 6-17. (a) 119 (b) +119  
 6-19. SUM =  $A \oplus B$ ; CARRY =  $AB$   
 6-21.  $[A] = 1111$ , or  $[A] = 000$  (if  $C_0 = 1$ )  
 6-25.  $C_3 = A_2B_2 + (A_2 + B_2) \{A_1B_1 + (A_1 + B_1)[A_0B_0 + A_0C_0 + B_0C_0]\}$   
 6-27. (a) SUM = 0111  
 6-32.



6-33.

	[F]	$C_{N+4}$	OVR
(a)	1001	0	1

- 6-35. (a) 00001100  
 6-37. (a) 0001 (b) 1010  
 6-39. (a) 1111 (b) HIGH (c) No change (d) HIGH  
 6-43. (a) 00000100 (b) 10111111  
 6-45. (a) 0 (b) 1 (c) 0010110  
 6-46. **AHDL**  
 $z[6..0] = a[7..1];$   
 $z[7] = a[0];$   
**VHDL**  
 $z(6..0) < = a(7..1);$   
 $z(7) < = a(0);$

- 6-53. Use D flip-flops. Connect  $(\overline{S_3 + S_2 + S_1 + S_0})$  to the D input of the 0 FF;  $C_4$  to the D input of the carry FF; and  $S_3$  to the D input of the sign FF.  
 6-54. 0000000001001001; 1111111110101110

**CHAPTER 7**

- 7-1. (a) 250 kHz; 50% (b) Same as (a) (c) 1 MHz  
 (d) 32  
 7-3.  $10000_2$   
 7-5. 1000 and 0000 states never occur  
 7-7. (a) See website for solution. (b) 33 MHz  
 7-11. Replace four-input NAND with a three-input NAND driving all FF CLR's whose inputs are Q5, Q4, and Q1  
 7-13. See website for solution.  
 7-15. Counter switches states between 000 and 111 on each clock pulse  
 7-17. See website for solution.  
 7-19. See website for solution.  
 7-21. (a) 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, and repeat (b) MOD-12  
 (c) Frequency at QD (MSB) is 1/12 of CLDK frequency  
 (d) 33.3%  
 7-23. (a) See website for solution. (b) MOD-10  
 (c) 10 down to 1 (d) Can produce MOD-10, but not same sequence  
 7-25. See website for solution.  
 7-27. See website for solution.  
 7-29.

Output:	QA	QB	QC	QD	RCO
Frequency:	3 MHz	1.5 MHz	750 kHz	375 kHz	375 kHz
Duty cycle:	50%	50%	50%	50%	6.25%

- 7-31. Frequency at  $f_{out1} = 500$  kHz, at  $f_{out2} = 100$  kHz  
 7-33.  $12M/8 = 1.5M$ ,  $1.5M/10 = 150k$ ,  $1.5M/15 = 100k$ . Also see website for solution.  
 7-35. See website for solution.  
 7-37. See website for solution.  
 7-39. See website for solution.  
 7-41. See website for solution.  
 7-43. (a)  $J_A = B \overline{C}$ ,  $K_A = 1$ ,  $J_B = C A + \overline{C} \overline{A}$ ,  $K_B = 1$ ,  $J_C = B \overline{A}$ ,  $K_C = B + \overline{A}$   
 (b)  $J_A = B \overline{C}$ ,  $K_A = 1$ ,  $J_B = K_B = 1$ ,  $J_C = K_C = B$   
 7-45.  $J_A = K_A = 1$ ,  $J_B = C \overline{A} + D \overline{A}$ ,  $K_B = \overline{A}$ ,  $J_C = D \overline{A}$ ,  $K_C = B \overline{A}$ ,  $J_D = \overline{C} B \overline{A}$ ,  $K_D = \overline{A}$   
 7-47.  $D_A = \overline{A}$ ,  $D_B = B A + \overline{B} \overline{A}$ ,  $D_C = C A + C B + \overline{C} \overline{B} \overline{A}$   
 7-49. See website for solution.  
 7-51. See website for solution.  
 7-53. See website for solution.  
 7-55. See website for solution.  
 7-57. See website for solution.  
 7-59. See website for solution.  
 7-61. See website for solution.  
 7-63. See website for solution.  
 7-65. See website for solution.



7-67. Eight clock pulses are needed to serially load a 74166, since there are eight FFs in the chip.

7-69. See website for solution.

7-71. See website for solution.

7-73. See website for solution.

7-75. See website for solution.

7-77. Output of 3-in AND or J, K inputs to FF D shorted to ground, FF D output shorted to ground, CLK input on FF D open, B input to NAND is open

7-79. See website for solution.

7-81. See website for solution.

7-83. See website for solution.

7-85. See website for solution.

7-87. See website for solution.

7-89. (a) Parallel (b) Binary (c) MOD-8 down (d) MOD-10, BCD, decade (e) Asynchronous, ripple (f) Ring (g) Johnson (h) All (i) Presettable (j) Up/down (k) Asynchronous, ripple (l) MOD-10, BCD, decade (m) Synchronous, parallel

## CHAPTER 8

8-1. (a) A; B (b) A (c) A

8-2. (a) 39.4 mW, 18.5 ns (b) 65.6 mW, 7.0 ns

8-3. (a) 0.9 V

8-4. (a)  $I_{IH}$  (b)  $I_{CCL}$  (c)  $t_{PHL}$  (d)  $V_{NH}$  (e) Surface-mount (f) Current sinking (g) Fan-out (h) Totem-pole (i) Sinking transistor (j) 4.75 to 5.25 V (k) 2.5 V; 2.0 V (l) 0.8 V; 0.5 V (m) Sourcing

8-5. (a) 0.7 V; 0.3 V (b) 0.5 V; 0.4 V (c) 0.5 V; 0.3 V

8-6. (b) AND, NAND (c) Unconnected inputs

8-7. (a) 40 (b) 33

8-8. (a) 20  $\mu$ A/0.4 mA

8-9. (a) 30/15 (b) 24 mA

8-11. Fan-out is not exceeded in either case.

8-13. 60 ns; 38 ns

8-14. (a) 2 k $\Omega$

8-15. (b) 4.7-k $\Omega$  resistor is too large.

8-19. a, c, e, f, g, h

8-21. 12.6 mW

8-27.  $AB + CD + FG$

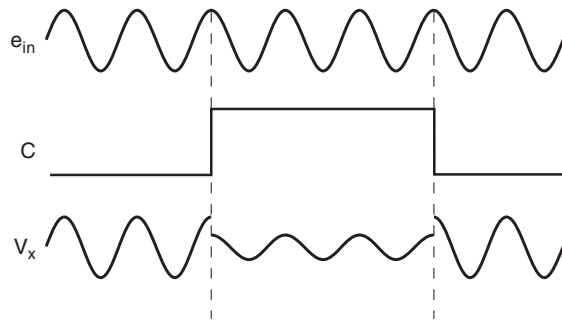
8-29. (a) 5 V (b)  $R_S = 110 \Omega$  for LED current of 20 mA

8-30. (a) 12 V (b) 40 mA

8-33. Ring counter

8-36. 1.22 V; 0 V

8-37.



8-38. -1 and -2

8-39. (a) 74HCT (b) Converts logic voltages (c) CMOS cannot sink TTL current. (d) False

8-41. (a) None

8-44. Fan-out of 74HC00 is exceeded; disconnect pin 3 of 7402 and tie it to ground.

8-46.  $R_2 = 1.5 \text{ k}\Omega$ ,  $R_1 = 18 \text{ k}\Omega$

8-49. (b) is a possible fault.

8-50. 0 V to -11.25 V and back up to -6 V

## CHAPTER 9

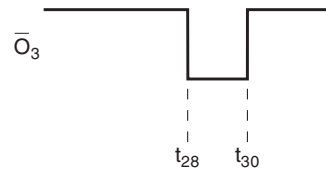
9-1. (a) All HIGH (b)  $\bar{O}_0 = \text{LOW}$

9-2. Six inputs, 64 outputs

9-3. (a)  $E_3\bar{E}_2\bar{E}_1 = 100$ ; [A] = 110

(b)  $E_3\bar{E}_2\bar{E}_1 = 100$ ; [A] = 011

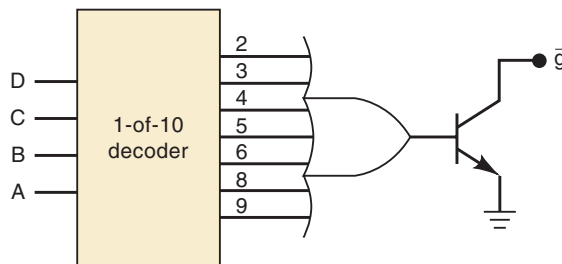
9-5.



9-7. (a) Enabled when  $D = 0$

9-10. Resistors are 250  $\Omega$ .

9-12.



9-13. (a), (b) Encoder (c), (d), (e) Decoder

9-17. The fourth key actuation would be entered into the MSD register.

9-18. Choice (b)

9-20. (a) Yes (b) No (c) No

9-21.  $A_2$  bus line is open between Z2 and Z3.

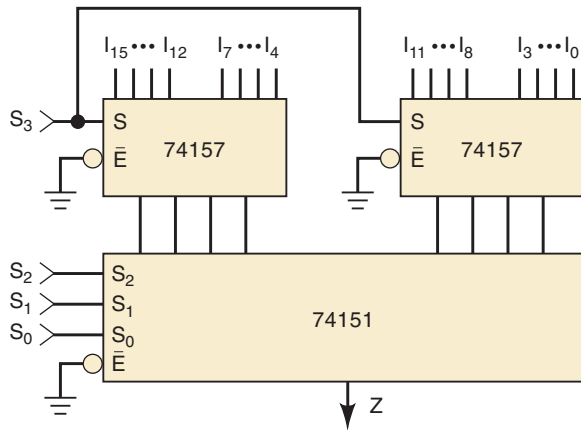
9-23.  $g$  segment too bright: segment LED or decoder output transistor would burn out.

9-25. Decoder outputs:  $a$  and  $b$  are shorted together.

9-26. Connection 'f' from decoder/driver to XOR gate is open.

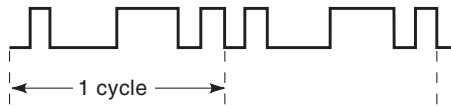
9-29. A 4-to-1 MUX

9-31. (a)



9-32. (b) The total number of connections in the circuit using MUXes is 63, not including  $V_{CC}$  and GND, and not including the connections to counter clock inputs. The total number for the circuit using separate decoder/drivers is 66.

9-33.



9-35.

A	B	C	
0	0	0	$0 \Rightarrow I_0$
0	0	1	$0 \Rightarrow I_1$
0	1	0	$0 \Rightarrow I_2$
0	1	1	$1 \Rightarrow I_3$
1	0	0	$0 \Rightarrow I_4$
1	0	1	$1 \Rightarrow I_5$
1	1	0	$1 \Rightarrow I_6$
1	1	1	$1 \Rightarrow I_7$

9-37.  $Z = \text{HIGH}$  for  $DCBA = 0010, 0100, 1001, 1010$ .

9-39. (a) Encoder, MUX (b) MUX, DEMUX

(c) MUX (d) Encoder (e) Decoder, DEMUX

(f) DEMUX (g) MUX

9-41. Each DEMUX output goes LOW, one at a time in sequence.

9-43. Five lines

9-46. (a) Sequencing stops after actuator 3 is activated.

9-47. Probable fault is short to ground at MSB of tens MUX.

9-48.  $Q_0$  and  $Q_1$  are probably reversed.

9-49. Inputs 6 and 7 of MUX are probably shorted together.

9-50.  $S_1$  stuck LOW

9-53. (a) Use three 74HC85s

9-55.  $A_0$  and  $B_0$  are probably reversed.

9-57.  $\overline{OE}_C = 0, \overline{IE}_C = 1; \overline{OE}_B = \overline{OE}_A = 1; \overline{IE}_B = \overline{IE}_A = 0$ ; apply a clock pulse.

9-61. (a) At  $t_3$ , each register holds 1001.

9-63. (a) 57FA (b) 5000 to 57FF (c) 9000 to 97FF

(d) No

9-65. See website for solution.

### CHAPTER 10

10-1. (d) 20 Hz (e) Only one LED will be lit at any time.

10-2. 24

10-3. Four states = four steps \*  $15^\circ/\text{step} = 60^\circ$  of rotation

10-5. Three state transitions \*  $15^\circ/\text{step} = 45^\circ$  of rotation

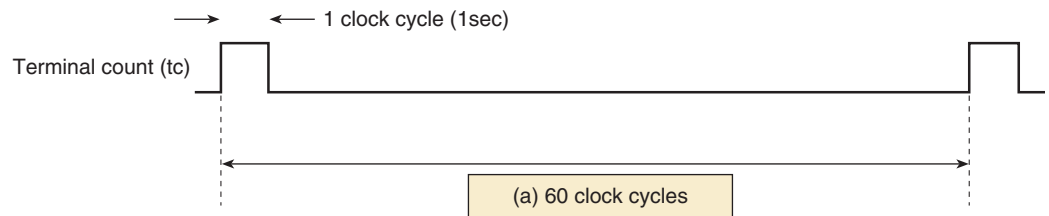
10-10. 1111

10-12. (a) 1011

10-13. No

10-15. The data go away (Hi-Z) before the DAV goes LOW. The Hi-Z state is latched.

10-16.



10-17. 60 cycles/sec \* 60 sec/min \* 60 min/hr \* 24 hr/day = 5,184,000 cycles/day. This takes a long time to generate a simulation file.

10-18. When the set input is active, bypass the prescaler and feed the 60-Hz clock directly into the units of seconds counter.

10-26. See website for solution.

### CHAPTER 11

11-1. (f), (g) False

11-3. LSB = 20 mV

11-5. Approximately 5 mV

11-7. 14.3 percent, 0.286 V

11-9. 250.06 rpm

11-11. The eight MSBs: PORT[7..0] ⇒ DAC[9..2]

11-13. 800 Ω; no

11-15. Uses fewer different  $R$  values

11-17. (a) Seven

11-19. 242.5 mV is not within specifications.

11-21. Bit 1 of DAC is open or stuck HIGH.

11-22. Bits 0 and 1 are reversed.

11-24. (a) 10010111

11-27. (a) 1.2 mV (b) 2.7 mV

11-28. (a) 0111110110

11-31. Reconstructed waveform frequency is 3.33 kHz.

11-32. (a) 5 kHz (b) 9.9 kHz

11-33. Digital ramp:  $a, d, e, f, h$ . SAC:  $b, c, d, e, g, h$

11-36. 80 μs

11-38. 2.276 V

11-40. (a) 00000000 (b) 500 mV (c) 510 mV

(d) 255 mV (e) 01101110 (f) 0.199°F; 1.99 mV

11-45. Switch is stuck closed, switch is stuck open, or capacitor is shorted.

11-47. (a) Address is EAxx.

11-52. False: a, e, g; True: b, c, d, f, h

### CHAPTER 12

12-1. 16,384; 32; 524,288

12-3. 64K × 4

12-7. (a) Hi-Z (b) 11101101

12-9. (a) 16,384 (b) Four (c) Two 1-of-128 decoders

12-11. 120 ns

12-15. The following transistors will have open source connections:  $Q_0, Q_2, Q_5, Q_6, Q_7, Q_9, Q_{15}$ .

12-19. Hex data: 5E, BA, 05, 2F, 99, FB, 00, ED, 3C, FF, B8, C7, 27, EA, 52, 5B

12-20. (a) 25.6 kHz (b) Adjust  $V_{ref}$ .

12-22. (a) [B] = 40 (hex); [C] = 80 (hex) (b) [B] = 55 (hex); [C] = AA (hex) (c) 15,360 Hz (d) 28.6 MHz

(e) 27.9 kHz

12-24. (a) 100 ns (b) 30 ns (c) 10 million (d) 20 ns

(e) 30 ns (f) 40 ns (g) 10 million

12-30. Every 7.8 μs

12-34. Add four more PROMs (PROM-4 through PROM-7) to the circuit. Connect their data outputs and address inputs to data and address bus, respectively. Connect  $AB_{13}$  to C input of decoder, and connect decoder outputs 4 through 7 to CS inputs of PROMs 4 through 7, respectively.

### CHAPTER 13

13-2. The necessary speed of operation for the circuit, cost of manufacturing, system power consumption, system size, amount of time available to design the product, and so on.

13-4. Speed of operation

13-6. Advantages: highest speed and smallest die area; Disadvantages: design/development time and expense

13-8. SRAM-based PLDs must be configured (programmed) upon power-up.

13-10. In a PLD programmer or in-system (via JTAG interface)

13-12. An LUT is a look-up table, used to define logic functions using SRAM memory



## INDEX OF ICs

### 74XX

- 7400 Quad two-input NAND gates, 632  
7402 Quad two-input NOR gates, 566, 633  
7404 Hex inverters, 176  
7406 Hex inverters buffer/driver (w/OC High-Voltage Output), 600–601, 613, 628, 738  
7407 Hex buffer/driver (with open-collector High-Voltage Output), 601  
7408 Quad two-input AND gates, 387  
7413 Dual four-input NAND gates with Schmitt-trigger inputs, 296  
7414 Hex Schmitt-trigger Inverters, 296, 300  
7442 BCD-to-decimal decoder, 644, 727  
7445 BCD-to-decimal decoder/driver, 645  
7446 BCD-to-seven-segment decoder/driver, 648  
7447 BCD-to-seven-segment decoder/driver, 648, 668, 709  
7483 Four-bit full adder (parallel-adder), 373, 721  
7485 Four-bit magnitude comparator, 687, 720  
7486 Quad two-input EX-OR gates, 632  
74112 Dual edge-triggered J-K flip-flop with preset and clear, 307  
74121 Single nonretriggerable one-shot, 298–299, 632  
74122 Single retriggerable one-shot, 298  
74123 Dual retriggerable one-shot, 298  
74138 1-of-8 decoder/demultiplexer, 705–706  
74147 Decimal-to-BCD priority encoder, 655–657, 713  
74148 Octal-to-binary priority encoder, 655  
74160 BCD counter, 456, 763–764  
74161 Synchronous MOD-16 counter, 456  
74162 Synchronous decade counter, 456  
74163 Synchronous MOD-16 counter, 456  
74173 Four-bit D-type registers with three-state outputs, 702  
74175 Quadruple D-type flip-flops with clear, 308  
74185 Six-bit binary-to-BCD code converter, 895  
74190 Four-bit synchronous up/down counter, 456  
74191 Four-bit synchronous up/down counter, 456, 508  
74221 Dual nonretriggerable one-shot, 298  
74283 Four-bit parallel-adder, 386, 389, 394  
74375 Quad bistable latches, 308  
74382 ALU, 384–385
- ### 74ACXX
- 74AC02 Quad two-input NOR gates, 176  
74AC11004 Hex inverters, 588
- ### 74ACTXX
- 74ACT02 Quad two-input NOR gates, 176  
74ACT11293 Four-bit binary counter, 625
- ### 74AHCXX
- 74AHC74 Dual edge-triggered D flip-flop, 601  
74AHC126 Quad noninverting tristate buffers, 604–605
- ### 74ALSXX
- 74ALS00 Quad two-input NAND gates, 507, 566, 567, 574–575, 577  
74ALS04 Hex inverters, 175  
74ALS14 Hex Schmitt-trigger inverters, 504  
74ALS138 1-of-8 decoder/demultiplexer, 641–642, 675–676, 739, 921, 923

- 74ALS151 Eight-input multiplexer, 664–665  
 74ALS157 Quadruple 2-line to 1-line data selectors/multiplexers, 666–668  
 74ALS160 Synchronous decade counter, 418, 430–434, 447  
 74ALS161 Synchronous MOD-16 counter, 418, 430–434, 447, 507, 532, 534–535  
 74ALS162 Synchronous decade counter, 418, 430–434, 447  
 74ALS163 Synchronous MOD-16 counter, 418, 430–434, 436–439, 447, 532, 645  
 74ALS164 Eight-bit serial in/parallel out shift register, 500–501  
 74ALS165 Eight-bit parallel in/serial out shift register, 498–500  
 74ALS166 Eight-bit serial in/serial out shift register, 496–498, 544  
 74ALS173 Four-bit D-type registers with three-state outputs, 696–698  
 74ALS174 Six-bit parallel in/parallel out register, 494–496, 541  
 74ALS190 Four-bit synchronous up/down counter, 430, 434–438, 447  
 74ALS191 Four-bit synchronous up/down counter, 430, 434–438, 447  
 74ALS192 Four-bit synchronous up/down counter, 430  
 74ALS193 Four-bit synchronous up/down counter, 430  
 74ALS273 Eight-bit register, 931  
 74ALS299 Eight-bit register w/common I/O lines, 705
- 74ASXX**  
 74AS04 Hex inverters, 175, 577  
 74AS20 Dual four-input positive-NAND gates, 575–576  
 74AS74 Dual edge-triggered D flip-flop, 577
- 74AUCXX**  
 74AUC08 Quad two-input AND gate, 615
- 74AVCXX**  
 74AVC08 Quad two-input AND gate, 614  
 74AVC1T45 Dual-supply-level translator, 614
- 74CXX**  
 74C02 Quad two-input NOR gates, 176  
 74C86 Quad two-input EX-OR gates, 164  
 74C266 Quad EX-NOR, 166
- 74FXX**  
 74F04 Hex inverters, 572
- 74HCXX**  
 74HC00 Quad two-input NAND gates, 151, 178, 613, 632–633  
 74HC02 Quad two-input NOR gates, 176, 187  
 74HC04 Hex inverters, 178, 589, 609  
 74HC05 Hex inverters with open-drain, 600, 601, 633  
 74HC08 Quad two-input AND gate, 187, 614–615  
 74HC13 Dual four-input NAND gates with Schmitt-trigger inputs, 296  
 74HC14 Hex Schmitt-trigger inverters, 296, 299  
 74HC42 BCD-to-decimal decoder, 644  
 74HC83 Four-bit full adder, 692  
 74HC85 Four-bit magnitude comparator, 687–690  
 74HC86 Quad two-input EX-OR gates, 164, 650–651  
 74HC123 Dual retriggerable one-shot, 298  
 74HC125 Quad noninverting tristate buffers, 613, 738  
 74HC126 Quad noninverting tristate buffers, 695  
 74HC138 1-of-8 decoder/demultiplexer, 672, 675, 734  
 74HC139 Dual 1-of-4 decoder w/active-LOW enable, 738  
 74HC147 Decimal-to-BCD priority encoder, 655  
 74HC148 Octal-to-binary priority encoder, 655  
 74HC151 Eight-input multiplexer, 664–766, 670–671, 675, 732  
 74HC157 Quadruple 2-line to 1-line data selectors/multiplexers, 666–667  
 74HC160 Synchronous decade counter, 418, 430–434  
 74HC161 Synchronous MOD-16 counter, 418, 430–434  
 74HC162 Synchronous decade counter, 418, 430–434  
 74HC163 Synchronous MOD-16 counter, 418, 430–434  
 74HC164 Eight-bit serial in/parallel out shift register, 500–501  
 74HC165 Eight-bit parallel in/serial out shift register, 498–500  
 74HC166 Eight-bit serial in/serial out shift register, 496–498  
 74HC173 Four-bit D-type registers with three-state outputs, 696–698  
 74HC174 Six-bit parallel in/parallel out register, 494–496, 738  
 74HC181 ALU (Arithmetic Logic Unit), 382  
 74HC190 Four-bit synchronous up/down counter, 430, 434–440, 533  
 74HC191 Four-bit synchronous up/down counter, 430, 434–440, 533  
 74HC192 Four-bit synchronous up/down counter, 430  
 74HC193 Four-bit synchronous up/down counter, 430  
 74HC221 Dual nonretriggerable one-shot, 298  
 74HC266 Quad EX-NOR, 166  
 74HC283 Four-bit full adder (parallel-adder), 373–374  
 74HC382 ALU (Arithmetic Logic Unit), 379–381  
 74HC541 Octal bus driver, 703–705  
 74HC881 ALU (Arithmetic Logic Unit), 382  
 74HC4016 Quad bilateral switch, 608–609, 630, 635  
 74HC4017 Johnson-counter, 506  
 74HC4022 Johnson-counter, 506  
 74HC4316 Quad bilateral switch, 609  
 74HC4511 BCD-to-7-segment decoder/driver, 650–651  
 74HC4543 LCD numerical display decoder/driver, 651

**74HCTXX**

74HCT02 Quad two-input NOR gates, 176  
 74HCT04 Hex inverters, 589  
 74HCT74 Dual edge-triggered D flip-flop, 601  
 74HCT293 Four-bit binary counter, 589

**74LSXX**

74LS00 Quad two-input NAND gates, 100–101, 184, 188, 229, 304, 334  
 74LS01 Quad two-input NAND gates open-collector, 601, 628  
 74LS04 Hex inverters, 175, 229, 589  
 74LS05 Hex inverters with open-drain, 599  
 74LS08 Quad two-input AND gate, 100–101  
 74LS13 Dual four-input NAND gates with Schmitt-trigger inputs, 296  
 74LS14 Hex Schmitt-trigger Inverters, 296, 299, 626, 661  
 74LS20 Dual four-input positive-NAND gates, 625  
 74LS32 Quad two-input OR gate, 100–101  
 74LS37 Quad two-input NAND gates (buffer), 625  
 74LS42 BCD-to-decimal decoder, 644  
 74LS74 Dual edge-triggered D flip-flop, 304  
 74LS83A Four-bit full adder (parallel-adder), 373  
 74LS85 Four-bit magnitude comparator, 687  
 74LS86 Quad two-input EX-OR gates, 164, 188, 625  
 74LS112 Dual edge-triggered J-K flip-flop, 334, 600, 625, 628, 632  
 74LS122 Single retriggerable one-shot, 298  
 74LS123 Dual retriggerable one-shot, 298  
 74LS125 Quad noninverting tristate buffers, 603, 605, 629  
 74LS126 Quad noninverting tristate buffers, 603, 605  
 74LS138 1-of-8 decoder/demultiplexer, 726, 862  
 74LS147 Decimal-to-BCD priority encoder, 655–658  
 74LS148 Octal-to-binary priority encoder, 655, 659  
 74LS181 ALU (Arithmetic Logic Unit), 382

74LS193

74LS221  
 74LS266  
 74LS283

74LS374  
 74LS382  
 74LS881

**74LVCXX**

74LVC07

**74SXX**

74S00  
 74S04  
 74S112

**OTHER ICs**

555  
 2125A  
 27C64  
 2732  
 4001B  
  
 4016  
 4049B  
 4316  
 AD781  
  
 AD7524  
 ADC0804  
  
 ADC0808  
 EMP7128SLC84  
 LM34  
 LM339  
 MCM101514  
 MCM6209C  
 MCM6249  
 PAL16R8  
 TMS4256

Four-bit synchronous up/down counter, 625  
 Dual nonretriggerable one-shot, 298  
 Quad EX-NOR, 166  
 Four-bit full adder (parallel-adder), 373, 375, 378, 383  
 Octal D-type FF register, 605  
 ALU (Arithmetic Logic Unit), 379–381  
 ALU (Arithmetic Logic Unit), 382

Hex buffer/driver (with open-collector), 614–615

Quad two-input NAND gates, 571  
 Hex inverters, 175  
 Dual edge-triggered J-K flip-flop, 632

Timer, 299–301  
 1K × 1 SRAM, 919–920  
 8K × 8 MOS ROM, 881  
 4K × 8 EPROM, 932  
 Quad two-input NOR gates, 176, 613, 632  
 Quad bilateral switch, 608  
 Hex inverters, 632  
 Quad bilateral switch, 609  
 Integrated sample-and-hold circuit, 843  
 Eight-bit DAC, 812  
 Successive-approximation ADC, 829–834  
 Successive-approximation ADC, 845  
 ALTERA PLD, 174  
 Temperature measuring device, 616  
 Quad Analog voltage comparator, 616  
 256K × 4 CMOS SRAM, 935  
 64K × 4 SRAM, 925  
 1M × 4 CMOS SRAM, 935  
 PLD with 8 registered outputs, 954  
 256K × 1 DRAM, 935



# INDEX

## A

- Access time
  - defined, 870
  - ROM, 881
- Accumulator register, 363
- Acquisition time, sample-and-hold circuits, 843
- Active (*see also* Asserted levels)
  - HIGH decoding, 441–442
  - logic levels, 104–105
  - LOW decoding, 442
- Actuator, 797
- Addend, 351, 365
- Adders
  - four-bit parallel, 375–378
  - full, 365
  - parallel, 364–366
- Addition in
  - BCD, 357–359
  - binary, 342–343
  - hexadecimal, 360–361
  - 2'-complement system, 351–352
    - equal and opposite numbers, 352
    - positive number and larger negative number, 351
    - positive number and smaller negative number, 351
    - two negative numbers, 352
    - two positive numbers, 351
- Address, 870
  - bus, 876, 912
  - decoders, ROM, 880
  - incomplete decoding, 923–924
  - inputs, 662, 874
  - multiplexing (in DRAM), 905–908
  - pointer registers, 927
  - setup time, 902
  - unidirectional, 876
- Address code, 293
- Advanced
  - CMOS 74AC/ACT, 589
  - high-speed CMOS 74AHC, 589
  - low-power Schottky TTL 74ALS series (ALS-TTL), 571–572
  - low-voltage BiCMOS (74ALVT), 596
  - low-voltage CMOS (74ALVC), 595
  - Schottky TTL 74AS series (AS-TTL), 571
  - ultra-low-power (74AUP), 596
  - ultra-low-voltage CMOS (74AUC), 595
  - very-low-voltage CMOS (74AVC), 595
- Advantages of digital techniques, 17
- AHDL, 114–117, 465–466
  - adder/subtractor, 394–395
  - BCD-to-binary code converter, 722–723
  - BEGIN, 120
  - behavioral description of a counter in, 466
  - bit array declarations, 205
  - Boolean description using, 119
  - buried nodes, 122–123
  - cascading BCD counters, 477–478
  - CASE, 217–218, 473, 518, 706, 717
  - code converter, 722–723
  - comments, 122–123
  - comparator, 720
  - concurrent assignment statement, 120
  - CONSTANT, 394
  - counter, 321
  - D latch, 314
  - decoder(s), 705–707
    - driver, 709–712
    - full-step sequence, 751
  - decoding the MOD-5 counter, 473–474
  - DEFAULTS, 706, 717
  - demultiplexers, 717–718
  - design file, 208

- digital clock project (HDL), 761–778 (*see also* HDL)
  - eight-bit adder, 392
  - ELSE, 469
  - ELSIF, 216, 469
  - Encoder, 713–715
  - END, 120
  - essential elements in, 120
  - flip-flops, 315–316
  - frequency counter project, 785–789 (*see also* HDL)
  - full-featured counter, 467–468
  - IF/THEN/ELSE, 212–213, 712–713, 769
  - INCLUDE, 775
  - INPUT, 120
  - intermediate variables in, 123
  - J-K flip-flop, 315
  - keypad encoder project, 755–761 (*see also* HDL)
    - scanning, 755
    - simulation, 761
    - solution, 758–759
  - MACHINE, 482–483
  - magnitude comparator, 720
  - MOD-5 counter, 461–462, 474
  - MOD-6 counter, 766, 775
  - MOD-8 counter, 750
  - MOD-10 counter, 477–478, 768, 775
    - graphic block symbols, 773
  - MOD-12 counter, 770–771
  - MOD-60 counter, 774
  - MOD-100 BCD counter, 477
  - module integration, 775–776
  - multiplexers, 717–718
  - NAND latch, 313
  - NODE, AHDL, 122, 390
  - nonretriggerable one-shot, 521
  - one-shots, simple, 521
  - OUTPUT, 120
  - PISO register, 515–516
  - primitive port identifiers, Altera, 314
  - retriggerable, edge-triggered one-shot, 525–526
  - ring counter, 519–520
  - ripple-up counter (MOD-8), 321
  - shift register, universal, 517–518
  - SISO register, 514–515
  - state descriptions in, 461–462
  - state machines, simple, 482–483
  - stepper driver, 752
    - simulation testing, 752
  - stepper motor driver project (*see also* HDL)
  - SUBDESIGN, 120, 122–123, 204–205, 316, 461, 469, 473–474
  - TABLE, 706, 713
  - traffic light controller, 486–488
  - truth tables, 208–209
  - VARIABLE, 122–123, 314, 390, 461
- Aliasing, 825–826
- Alphanumeric codes, 53–55
- ALTERA
- binary up counter, 389
  - block diagram file of an eight-bit ALU, 385
  - cyclone
    - II and III series device features, 958
    - IV E family, 958
    - series, 958
  - eight-bit parallel adder using 74283 macrofunction, 386–388
  - eight-bit parallel adder using LPM\_ADD\_SUB macrofunction, 386–388
  - full-featured MOD-16 counter, 456
  - hardware description language, 114–117
  - LEs, 956
  - library functions, using, 384–390
    - for counters, 456–460
  - logic array blocks (LABs), 955
  - logic elements (LEs), 956
  - LPM\_COUNTER, 456
  - LPM\_SHIFTREG, 509
  - macrofunction, 384
  - MAX II family, 955–959
    - block diagram, 956–957
    - configuration flash memory (CFM), 957
    - features, 958
    - LAB (logic array block) structure, 958–959
    - LE (logic elements) in normal mode, 956
    - logic array block (LAB), 958–959
  - MAX7000S family, 955–958
    - macrocell, 955–956
  - megafunction LPMs for arithmetic circuits, 385–386
  - megaWizard settings, 509–511
  - Mod-8 down counter block diagram and functional simulation results, 390
  - parallel adder to count, using, 389–390
  - primitive port identifiers, 314
  - programmable interconnect array (PIA), 955
  - Quartus II, 115
    - maxplus2 library, 509
    - megafunctions
      - decoders, 641–642
      - magnitude comparator, 687–690
      - mux, 662–663
    - schematic capture, 179
  - SER\_IN, 509
  - SER\_OUT, 509
- Alternate logic-gate representation, 102–105
- ALU integrated circuits, 363–364, 378–382
- expanding the ALU, 381
  - operations
    - ADD, 379
    - AND, 380
    - CLEAR, 379
    - EX-OR, 380
    - OR, 380
    - PRESET, 380
    - SUBTRACT, 379
- American Standard Code for Information Interchange (ASCII), 54–55
- Amplitude, 17
- Analog quantity, 795
- Analog voltage comparators, 614–616
  - counter decoding, 440–443
- Analog-to-digital (ADC)
- accuracy, 821–821
  - conversion, 813, 815–816
  - conversion time, 821–822, 827
  - converter (ADC), 19, 796
  - data acquisition, 822–826
  - digital amplitude control, 813–814
  - digital-ramp, 817–819
  - dual-slope, 837–838
  - flash, 834–836
  - IC, eight-bit successive approximation (ADC0804), 829–834
    - application, 832
    - Chip Select ( $\overline{CS}$ ), 831



Analog-to-digital (*continued*)

- CLK IN, 831
- CLK OUT, 831
- differential inputs, 829
- INTERRUPT ( $\overline{INTR}$ ), 831
- READ ( $\overline{RD}$ ), 831
- $V_{ref}/2$ , 831
- WRITE ( $\overline{WR}$ ), 831
- multiplexing, 844–845
- other conversion methods, 836–842
- pipeline, 840–842
- quantization error, 820–821
- resolution, 820–822
- sample-and-hold circuit, 843–844
- serial, 826
- sigma/delta modulation, 838–840
- successive approximation, 826–834 (*see also* Digital-to-analog converter)
- typical architectures for applications, 842
- voltage-to-frequency, 838

Analyzing synchronous counters, 444–447

- representation, 15
- systems, 17

AND gate, 77–80 (*see also* Combinational logic circuits)

- alternate logic-gate representation, 102–105
- Boolean description, 77
- Boolean theorems, 92–95
- defined, 78
- implementing from Boolean expressions, 87–88
- summary of operation, 79
- symbol, 78
- which representation to use, 105–111

AND operation, 71, 77–80

- summary, 79

Aperiodic digital signals, 11

Application-specific integrated circuits (ASICs), 943–944

Applications of analog interfacing, 849–851

- data acquisition systems, 849–850
- digital camera, 850
- digital cellular telephone, 850–851

ARCHITECTURE, 121, 205

Arithmetic circuits, 363–364

Arithmetic/logic unit (ALU), 29, 363–364, 378–382

- functional parts of an, 363

Arithmetic overflow, 353–354

Array, register, 880

ASCII code, 54–55

- padding, 55

ASICs, 943–944

Asserted levels, 110

Associative laws, 93

Astable multivibrators, 299–302

- 555 timer used as, 299–301

Asynchronous (ripple) counters, 410–414

- MOD number, 412
- propagation delay, 414–416

Asynchronous active pulse width, 269

Asynchronous inputs, 264–267

- designations for, 266–267

Asynchronous systems, 251

Asynchronous transfer, 276–277

Augend, 351, 364

Automatic circuit testing (using DACs), 813

Auxiliary memory, 868

**B**

Backlit LCDs, 649

Backplane, LCD, 650

Ball grid array, 560

Barrel shifter, 847

Base-10 system, 19

Basic characteristics of digital ICs, 173–180

BCD addition, 357–359

- sum equals 9 or less, 358
- sum greater than 9, 358–359

BCD (binary-coded-decimal) code, 46–48

- advantage, 48
- BCD-to-decimal decoder, 645
- BCD-to-decimal decoder/driver, 644
- BCD-to-7 segment decoder/driver, 647–648
- comparison with binary, 47–48
- forbidden codes, 47
- subtraction, 361

BCD counters, 425

- decoding, 442
- displaying two multidigit, 668

Behavioral

- description, 465
- level of abstraction, 464

BiCMOS

- 5-volt logic, 589
- family, 595–596

Bidirectional

- busing, 704–705
- data lines, 705

Bilateral switch, 607–609

Binarily weighted, 806

Binary

- addition, 342–343
- arithmetic and number circles, 354–355
- BCD, 46–48
- counter, 283
- counting sequence, 22
- digit, 22
- division, 357
- multiplication, 355–356
- parity method for error detection, 56–58
- point, 21
- quantities, representation of, 23–25
- subtraction, 342

Binary system, 21–22

- binary-to-decimal conversion, 38
- binary-to-gray conversion, 49
- binary-to-hex conversion, 44
- conversions, summary, 45–46
- decimal-to-binary conversion, 39–41
- gray-to-binary conversion, 49
- hex-to-binary conversion, 43–44
- negation, 347–348
- parallel and serial transmission, 25–27
- representing quantities, 23–25
- signed numbers, representing, 343–350

Bipolar DACs, 806

Bipolar digital ICs, 174–175

Bistable multivibrators, 239, 299–302

(*see also* Flip-flops)

Bit, 22

- arrays, 203–204
- carry, 365
- sum, 365
- vectors, 203–204

- Block diagram (digital clock using HDL), 762
  - Blu-ray technology, 916–917
  - Boolean
    - algebra, 71
    - alternate logic-gate representation, 102–105
    - AND operation, 77–80
    - constants and variables, 71
    - DeMorgan's theorems, 95–98
    - description of logic circuits, 82–84
    - evaluation of logic-circuit outputs, 84–86
    - implementing circuits from expressions, 87–88
    - NAND gate, 88–92
    - NOR gate, 88–92
    - NOT operation, 71, 80–81
      - summary, 81
    - OR operation, 73–77
      - summary, 75–76
    - simplifying logic circuits, 139–140
    - theorems, 92–95
    - truth tables, 72–73
    - which representation to use, 105–111
  - Bootstrap
    - memory, 894
    - program, 894
  - B register, 364
  - Bubbles, 104
    - placement of, 107–108
  - Buffer(s)
    - circular, 928
    - driver, 600–601
    - inverting, 603
    - linear, 928
    - noninverting, 603
    - open-collector, 600–601
    - output, ROM, 879
    - tristate, 603–604
  - Building the blocks from the bottom up (digital clock using HDL), 766
  - Bulk erase, 889
  - Bundle method, 703–704
  - Buried nodes, AHDL, 122–123
  - Bus
    - address, 876
    - contention, 604
    - control, 876
    - data, 876
    - drivers, 702
    - expanding, 701–702
    - high-speed interface logic, 605–606
    - representation, simplified, 703
    - signals, 700–701
    - termination techniques, 606
  - Busing, bidirectional, 704–705, 876
  - Byte, 52–53, 869
- C**
- Cache memory, 871, 926–927
  - Capacity, memory
    - defined, 869
    - expansion, 917–925
  - Carry, 342, 364
    - bit, 365
    - look-ahead, 372
    - propagation, 372–373
    - ripple, 372
  - Cascading
    - parallel adders, 373–375
  - CASE using
    - AHDL, 217
    - VHDL, 218
  - Cell phone, 30, 850–851
  - Central processing unit (CPU), 29 (*see also* Microprocessors)
  - Characteristics of an FPGA, 619–621
  - Checker, parity, 169–170
  - Chip, 173
  - Chip select, 877, 898
  - Choosing HDL coding techniques, 491
  - Circuit excitation table, 451
  - Circuits, digital, 8 (*see also* Logic circuits)
    - adder/subtractor, 377
    - clock generator, 302
    - enable/disable, 170–172
    - 2's-complement, 375–378
  - Circular buffers, 928
  - Circulating shift register, 502
  - CLEAR, 285–286
  - Cleared state, latch, 241
  - Clock
    - crystal-controlled, 302
    - defined, 251
    - duty cycle, 413–414
    - edges, 251
    - frequency, 251
    - generator circuits, 299–302
      - crystal-controlled, 302
    - period, 251
    - pulse HIGH  $t_w(H)$ , 268
    - pulse LOW  $t_w(L)$ , 268
    - signals, 251–254
    - skew, 305–307
    - transition times, 269
  - Clocked flip-flops, 251–254
    - asynchronous inputs, 264–267
  - D, 260–262
  - D latch (transparent latch), 262–264, 314
  - J-K, 258–260
  - S-R, 254–257
  - CMOS logic family, 14, 174–175, 585–594
    - 4000/14000 series, 176
    - 74AC series, 176
    - 74ACT series, 176
    - 74C series, 176
    - 74HC series, 176
    - 74HCT series, 176
    - 4000/14000 series, 588–589
    - 74AC series, 551
    - 74ACT series, 589
    - 74AHC series, 589
    - 74AHCT series, 590
    - 74ALB, 596
    - 74ALVC series, 595
    - 74ALVT series, 596
    - 74AUP series, 596
    - 74AVC series, 595
    - 74HC series, 589
    - 74HCT series, 589
    - 74LV series, 595
    - 74LVC series, 595–596
    - 74LVT series, 596
    - 74VME series, 596

- CMOS logic family (*continued*)
  - advanced low-voltage, 596
  - BiCMOS 5-volt, 589
  - bilateral switch, 607–609
  - characteristics, 588–594
  - driving TTL, 611
    - in the HIGH state, 611
    - in the LOW state, 612–613
  - electrically compatible, 176, 589
  - electrostatic discharge (ESD), 593
  - fan-out, 592–593
  - flash memory IC, typical, 889–893
  - functionally equivalent, 589
  - ground, 176
  - input voltages, 590
  - INVERTER circuit, 175, 586
  - Karnaugh map method, 368
  - latch-up, 594
  - logic level voltage ranges, 177
  - low-voltage BiCMOS, 596
  - low-voltage levels, 590
  - NAND gate, 586–587
  - noise margins, 590
  - NOR gate, 587–588
  - open-drain outputs, 597–602
  - output voltages, 590
  - outputs shorted together, 597
  - PD increases with frequency, 591–592
  - pin compatible, 589
  - power dissipation, 591
  - power-supply voltage, 176–177, 590
  - series characteristics, 588–594
  - SET-RESET FF, 588
  - static RAM cell, 858, 899–900
  - static sensitivity, 593–595
  - switching speed, 592
  - transmission gate, 607–609
  - tristate outputs, 602–605
  - TS switch, 596
  - unconnected inputs, 177–178
  - unused inputs, 593
  - voltage levels, 590
- Code
  - alphanumeric, 53–55
  - BCD, 46–48
  - defined, 46
  - gray, 48–50
  - putting it all together, 51
- Code converters, 690–694, 721–724
  - basic idea, 691
  - circuit implementation, 692–693
  - conversion process, 691–692
  - other implementations, 694
- Codec, 851
- Column address strobe ( $\overline{CAS}$ ), 906
- Combinational logic circuits, 136–234
  - algebraic simplification, 140–145
  - complete design procedure, 147–152
  - complete simplification process, 157–160
  - designing, 145–152
  - exclusive-NOR, 163–168
  - exclusive-OR, 163–168
  - Karnaugh map method, 152–163
  - parity generator and checker, 169–170
  - product-of-sums, 138–139
  - simplifying, 140–145
  - sum-of-products form, 138–139
  - summary, 220–221
- Combined addition and subtraction, 377
- Combining DRAM chips, 924–925
- Command register, 890
- Common input/output pins (in RAM), 898–899
- Commutative laws, 93
- Complementation, 80 (*see also* NOT operation)
- Complete hierarchy of the project (digital clock using HDL), 766
- Complex programmable logic devices (CPLDs), 944
- Computers
  - data acquisition system, 822–824
  - decision process of a program, 116
  - digital, 28–31
  - embedded controller, 30
  - functional diagram of, 29
  - major parts of, 28–29
  - microcomputer, 29
  - microcontroller, 29
  - microprocessor, 29
  - programming languages, 114–117
  - types of, 29–30
- Concatenating, 514
- Conditional signal assignment statement, 715
- Constants, 394
- Contact bounce, 243
- Control
  - bus, 876
  - inputs, 253, 265
    - synchronous, 253
  - unit, 29
- Controlled inverter, 166
- Conversion time, ADC, 821–822, 827
- Converter, data, 895
- Count enable, 431
- Counters and registers, 408–528
  - asynchronous (ripple), 410–414
    - propagation delay, 414–416
  - basic idea, 447
  - BCD, decoding, 442
  - cascading, 439–440
  - decade, 425
  - decoding, 431, 440–443, 505
  - design procedure, 449–452
  - displaying states, 421
  - feedback, with, 502
  - glitches, 416, 421
  - HDL, 460–472
  - J-K excitation table, 449
  - Johnson, 503–505
    - decoding, 505
    - with MOD numbers  $_2N$ , 419–420
  - multistage arrangement, 439–440
  - NEXT state, 444–447
  - parallel in/parallel out (74ALS174/74HC174), 494–496
  - parallel in/serial out (74ALS165/74HC165), 498–500
  - PRESENT state, 444–447
  - presettable, 428–430
  - recycle, 410
  - ring, 502–503, 658
  - ripple, 321–323, 410–414
  - self-correcting, 445
  - serial in/parallel out (74ALS164/74HC164), 500–501
  - serial in/serial out (74ALS166/74HC166), 496–498

- 74ALS160-163/74HC160-163 series, 430–434
- 74ALS190-191/74HC190-191 series, 434–438
- shift register, 502–506
- spike, 421
- summary, 492, 528–529
- synchronous (parallel), 416–419, 430–434
- synchronous (parallel) down and up/down, 426–428
- synchronous design, 447–455
- synchronous design with D FFs, 454–455
- synchronous presetting, 429
- synchronous, analyzing, 444–447
- transition states, 421
- troubleshooting, 506–509
- undesired states, 449
- with MOD numbers  $_2N$ , 419–420
- Counting
  - binary, 22–23
  - decimal, 20–21
  - hexadecimal, 44
  - operation, 283–284
- CPU, 29
- Cross Bar Technology (74CBT), 596
  - low voltage (74CBTLV), 596
- Crystal-controlled clock generators, 302
- Current
  - parameters for digital ICs, 552–553
  - sinking action, TTL, 557, 564
  - sinking transistor, TTL, 564
  - sourcing action, 557, 564
  - sourcing transistor, TTL, 564
  - transients, TTL, 581–582
- D**
- DAC (see Digital-to-analog converter)
- Data
  - acquisition, 822–826, 849
  - bus
    - bundle method, 703
    - defined, 694
    - floating, 696
    - operation, 699–705
  - compression, 850
  - converter, 895
  - distributors, 673–683
  - hold time, 902
  - lines, 293
  - rate buffer, 928
  - routing, by MUXs, 668–669
  - sampling, 822
  - selectors, 662–663
  - setup time, 902
  - storage and transfer, 276–278
  - tables, 895
  - transfer operation, 276
  - word, 702
- Data transfer, 276–278
  - asynchronous, 276
  - data busing, 699–700
  - demultiplexers, 673–683
  - hold time requirement, 279–280
  - operation, 699–700
  - parallel, 262, 277–278
  - parallel *versus* serial transfer, 281
    - economy and simplicity of, 281
    - speed, 281
  - and portability, 894
  - and storage, 276–278
  - registers, between, 699
  - serial, 277
  - shift registers, 278–281
  - simulation, full-step, (HDL), 751
  - simultaneous, 277
  - synchronous, 277
- Decade counters, 425
- Decimal
  - counting, 20–21
  - point, 19
- Decimal system, 19–20
  - binary-to-decimal conversion, 38
  - conversions, summary, 45–46
  - decimal-to-binary conversion, 39–41
    - counting range, 41
  - decimal-to-hex conversion, 42–43
  - hex-to-decimal conversion, 42
- Decimal-to-BCD priority encoder (74147), 655–656
- Decision control structures in HDL, 210–220
- Decoders, 639–647
  - address, 880
  - applications, 645–646
  - BCD-to-7 segment drivers, 647–649
  - BCD-to-decimal, 640
  - Binary-to-octal, 640
  - column, 880
  - demultiplexer, 673–683
  - ENABLE inputs, 640
  - 4-to-10, 644
  - liquid crystal displays (LCDs), 648–649
  - 1-of-10, 644
  - 1-of-8, 640–642
  - row, 880
  - 3-line-to-8-line, 644
  - using HDL, 705–708
- Decoding
  - counters, 440–443
  - Johnson, counter, 503–505
- Decoupling, power-supply TTL, 582
- DeMorgan's theorems, 95–98
  - implications of, 97–98
- Demultiplexers (DEMUXs), 673–683, 716–719
  - 1-line-to-8-line, 674–675
  - security monitoring system, applications, 675–677
- Dependency notation
  - &, 698
  - ◇, 601
  - ▽, 605
- Depletion MOSFET, 583
- Describing logic circuits, 68–135
  - summary, 124–125
- Description languages *versus* programming languages, 114–117
- Designing combinational logic circuits, 145–152
- Detecting a transition or event, 275
- Detecting an input sequence, 273–274
- Development software (for PLDs), 197–199
- Diagrams
  - logic circuit connections, 178–179
  - simplified bus timing, 701
  - state transition, 284, 421, 445
  - timing, 445
- Differential inputs, 829

- Digital, techniques
  - amplitude control, 813–814
  - and analog systems, 17–19
  - arithmetic, 340–407
    - 2's-complement system addition, 351–352
    - 2's-complement system, multiplication, 357
    - 2's-complement system, subtraction, 352–355
    - BCD addition, 357–359
    - binary addition, 342–343
    - binary division, 357
    - binary multiplication, 355–356
    - carry propagation, 372–373
    - circuits, 363–364
    - full adder, 365
    - hexadecimal addition, 360–361
    - hexadecimal representation of signed numbers, 362–363
    - hexadecimal subtraction, 361–362
    - Integrated-circuit parallel adder, 373–375
    - number circles and binary arithmetic, 354–355
    - operations and circuits, 340–407
    - parallel binary adder, 364–366
    - signed number representation, 343–350
    - summary, 397–398
  - camera, 850
  - cellular telephone, 850–851
  - circuits, 8, 14 (*see also* Logic circuits)
  - computers, 28–31 (*see also* Microcomputers)
  - family tree, 942–947
  - integrated circuits, 14
  - multiplexer, 662–663
  - number systems, 19–23
  - one-shots, HDL, 521–528
  - pulses, 249–250
  - quantity, 795–796
  - ramp ADC, 817–819
  - representation, 15–16
  - signal processing (DSP), 845–848
    - architecture, 847
    - arithmetic logic unit (ALU), 847
    - barrel shifter, 847
    - filtering, 846–848
    - interpolation filtering, 848
    - multiply and accumulate section (MAC), 847
    - oversampling, 847
    - weighted average, 847
  - techniques
    - advantages, 17
    - limitations, 18–19
  - temperature control system, 18
  - vs.* analog, review, 795–797
- Digital signals, 9–13
  - duty cycle, 12
  - edges/events, 12
  - highs and lows over time, 11
  - period/frequency, 11–12
  - periodic/apperiodic, 11
  - timing, 10
  - transitions, 12
- Digital systems, 17
  - input internally shorted to ground or supply, 182–183
  - introductory concepts, 2–35
    - summary, 31–32
  - malfunction in internal circuitry, 182
  - open signal lines, 186–187
  - open-circuited input or output, 183–185
    - output internally shorted to ground or supply, 183
    - output loading, 188
    - power supply, faulty, 187–188
    - short between two pins, 185–186
    - shorted signal lines, 187
    - tree diagram, 685
    - troubleshooting, 180–181, 617–618, 659–662, 683–686
      - prototyped circuits, 190–194
- Digital-to-analog converter (DAC), 19, 796–825
  - accuracy, 810–811
  - analog output, 798
  - analog-to-digital conversion, used in, 816–817
  - applications, 813–814
  - bipolar, 804
  - circuitry, 804–809
  - control, used in, 813
  - conversion, 797–804
  - conversion accuracy, 806
  - current output, with, 806–808
  - digitizing a signal, 825
  - full-scale output, 798, 800
  - input weights, 799–800
  - integrated circuit (AD7524), 812–813
  - monotonicity, 811
  - offset error, 811
  - output waveform, 800
  - percentage resolution, 801–802
  - perfect, 810
  - R/2R ladder, 808–809
  - resolution, 800–802
  - serial, 814
  - settling time, 811
  - signal reconstruction, 824
  - specifications, 809–811
  - staircase, 800
  - staircase test, 815
  - static accuracy test, 815
  - step size, 800–801
  - troubleshooting, 814–815
- Digitize
  - reconstructing a signal, 822–824
  - signal, 813, 824–826
- Digits, 15, 19
- DIMM, 913
- Diode, Schottky barrier (SBD), 570
- DIP (dual-in-line package), 173
- Discrete steps, 15
- Displaying counter states, 421
- Displays
  - LCD, 648–653
    - backlit, 649
    - passive matrix panel, 652
    - reflective, 649
    - Super Twisted Nematic (STN), 652
    - TFT (Thin Film Transistor), 653
    - Twisted Nematic (TN), 652
  - LED, 648–649
    - common-anode, 648–649
    - common-cathode, 648–649
- Distributive law, 93
- Divide and conquer, troubleshooting process, 660
- Dividend, 357
- Division, binary, 357
- Divisor, 357
- D latch (transparent latch), 262–264, 314

- D latch (see Flip-flops)
- Don't-care conditions, 161–162
- Double Data Rate SDRAM (DDRSDRAM), 914
- Driver, decoder, 645
- DSP (Digital Signal Processing), 845–848
- Dual
  - in-line package (DIP), 560
- Dual-in-line package (DIP), 173
- Dual-slope ADC, 837–838
- Duty cycle, 12, 413–414
- DVD player, block diagram, 199
- Dynamic RAM (DRAM), 902–903
  - address multiplexing, 905–908
  - architecture, simplified, 906
  - combining chips, 924–925
  - controller, 912
  - DDRSDRAM, 914
  - DIMM, 913
  - EDO, 914
  - FPM (Fast Page Mode), 914
  - memory modules, 913–914
  - read/write cycles, DRAM, 909–910
    - read cycle, 909
    - write cycle, 910
  - refresh counter, 912
  - refreshing, 903, 910–912
    - burst, 911
    - CAS-before  $\overline{\text{RAS}}$  refresh, 911, 912
    - distributed, 911
    - RAS-only refresh, 911
  - SDRAM (Synchronous DRAM), 914
  - SIMM, 913
  - simplified architecture of a typical, 906
  - SODIMM, 913
  - structure and operation, 904–908
  - technology, 913–915
- E**
- Edge-detector circuit, 257
- Edge
  - falling, 9
  - negative, 9
- Edges, of a clock signal, 251–252
- Edges/events, 12
- Edge-triggered devices, 315–320
  - event, 315
  - logic primitive, 315
- EDO (Extended Data Output) DRAM, 914
- EEPROMs (electrically erasable PROMs), 887–888
- Eight-input multiplexers, 664–666
- Electrical noise, 56
- Electrically Erasable PROMs (EEPROMs), 887–888
- Electrostatic discharge (ESD), 593
  - microcontroller program memory, 894
- ELSIF, 215–216
  - using AHDL, 216
  - using VHDL, 217
- Embedded controller, 30
- ENABLE inputs, decoders, 640–644
- Enable/Disable circuits, 170–172
- Encoders, 652–659
  - decimal-to-BCD priority, 655–656
  - 8-line-to-3-line, 654
  - octal-to-binary, 654
  - priority, 655–656
  - switch, 656–659
- Encoding, 653, 916
- Enhancement MOSFET, 583
- EPROMs (erasable programmable ROMs), 886–887
- Erasable Programmable ROMs (EPROMs), 886–887
- Error detection, parity method for, 56–58
- Etching, incomplete, 187
- Even-parity method, 57
- Event, 315
- Excitation table, J-K, 448
- Exclusive
  - NOR circuit, 165–168
  - OR circuit, 163–164
- Extension, sign, 347–348
- External faults, 186–187
- F**
- Falling edge, 9
- Fan-out, 553
  - CMOS, 592
  - determining, 574–577
  - TTL, 573–578
- Fast page mode (FPM) DRAM, 914
- Fast TTL (74F), 571
- FGMOSFET, 891
- Field-programmable gate arrays (FPGA), 619–620, 944–947
  - Altera Cyclone II characteristics using general purpose I/O standards, 620
  - Altera Cyclone II comparison of counter performance, 621
  - architecture, 914
  - characteristics of, 619–621
  - logic voltage levels, 619–620
  - maximum input voltage and output current ratings, 621
  - power dissipation, 620
  - power-supply voltage, 619
  - switching speed, 621
- Filling K map from output expression, 160–161
- First-in, first-out memory (FIFO), 927–928
- Flash
  - ADC, 836
  - conversion time of, 836
  - memory, 889–893
  - NAND, 891–892
  - NOR, 891–892
  - technology, 891–892
  - typical CMOS memory IC, typical, 89–891
- Flip-flops, 27, 236–339
  - ambiguous output, 258
  - applications, 271
    - with timing constraints, 286–292
  - asynchronous inputs, 264–267
  - bistable multivibrator, 239
  - clearing, 239
  - clock signals, 251–254
  - clocked, 251–254
  - clocked D, 260–262
    - implementation of, 260
  - D (data), 260–261
    - implementation of, 250
  - D latch (transparent latch), 262–264, 314
  - defined, 238
  - edge-triggered, 252
  - feedback, 238
  - frequency division and counting, 282–285
  - input sequence detection, 273–274

- Flip-flops (*continued*)
    - J-K, 258–260
    - latches, 27
    - memory characteristics, 239
    - NAND gate latch, 239–244
      - alternate representations, 242
      - summary of, 241–242
      - troubleshooting case study, 247–249
      - using AHDL, 313
    - NOR gate latch, 245–247
    - override inputs, 265
    - propagation delays, 268
    - resetting, 239
    - S-R, 254–257
    - setting, 239
    - setup and hold times, 252–254
    - shift registers, 278–281
    - state on power-up, 247
    - summary, 323–324
    - synchronization, 272–273
    - terminology, 242
    - timing considerations, 267–269
    - timing problems, 269–271, 290–292
    - transition or event detection, 275
    - troubleshooting circuits, 302–307
  - Floating, 177
    - bus, 696
    - gate, MOSFETs, 886–888, 891–892
    - inputs (*see also* Unconnected inputs)
  - Floating inputs, 177–178 (*see also* Unconnected inputs)
  - Four-input multiplexers, 664
  - Free-running multivibrator, 299–302
  - Frequency, 11–12, 12, 251
    - counter project (HDL), 785–789 (*see also* HDL)
    - division, 282–285, 412–413
      - and counting, 282–285
  - Full adder, 365
    - design of, 366–369
    - K-map simplification, 368
  - Full-featured counters in HDL, 467–468
  - Full-scale error (of a DAC), 810
  - Full-scale output (of a DAC), 798, 800
  - Full-step sequence (HDL stepper-motor), 748
  - Function generator, 895
  - Fusible-link, PROMs, 885
- G**
- GAL (Generic array logic), 954–955
    - output logic macro cells (OLMCs), 954–955
  - Gate(s)
    - AND, 77–80
      - arrays, 943–944
    - NAND, 88–92
    - NOR, 88–92
    - NOT (inverter), 80–81
    - OR, 73–77
      - propagation delay, 111–112
      - which representation to use, 105–111
    - XNOR, 165–168
    - XOR, 163–164
  - Generator
    - parity, 169–170
    - function, 895
  - Giga-scale integration (GSI), 174, 551
  - Glitches, 416, 421
  - Gray Code, 48–50
  - Gunning Transceivers Logic Plus, 607
    - Plus (74GTLP), 596
- H**
- Half-adder, 369
  - Half-step sequence (HDL stepper-motor), 748
  - HDL (hardware description language), 114, 197
    - adders, 392–394
      - behavioral description, 464
      - behavioral level of abstraction, 464
    - bit arrays, 203–204
    - bit vectors, 203–204
    - CASE, 518, 706, 751, 769–770
      - choosing coding techniques, 491
    - concatenation, 513
    - counters, 460–472
      - circuits with multiple components, 320–323
    - code converters, 721–724
    - combining blocks using only, 774–775
    - comparator, 720–721
    - decision control structures, 210–220
      - concurrent, 210
      - sequential, 210
    - decoder/driver, 7-segment, 709–712
    - decoders, using, 705–708
    - demultiplexers, 716–719
    - designing number systems in, 202
    - digital clock project, 761–778
      - block diagram, 762
        - building the blocks from the bottom up, 766
        - combining blocks graphically, 773–774
        - complete hierarchy of the project, 766
        - hours section circuit, 762
        - MOD-6 counter simulation, 767–768
        - MOD-60 section, 765
        - prescaler, 765
        - top-down hierarchical design, 764–766
    - encoders, 712–716
    - format and syntax, 118–121
    - frequency counter project, 785–789
      - block diagram, 787
      - sampling interval, 786
      - timing and control block, 787
      - timing diagram, 788
    - full-featured counters in, 467–468
    - hierarchical design, 473
    - IF/ELSE, 211–212, 520
    - IF/ELSIF, 769
    - IF/THEN, 211
    - index, 203
    - keypad encoder project, 755–761
      - block diagram, 756
      - encoder operation, 756
      - problem analysis, 755
      - problem decomposition, 757
      - simulation, 761
      - strategic planning, 757
    - literals, 202
    - magnitude comparator, 720–721
    - microwave oven project, 778–785
      - definition of the project, 779–789
      - encoder/timer input control block, 783
      - hierarchy showing blocks and signals, 784
      - integration and testing, 784

- magnetron control block, 783
  - minutes/seconds counter, 781
  - problem decomposition, 780–784
  - strategic planning, 780–784
  - synthesis, 784
  - system block diagram, 779
  - TC (terminal count), 782
  - the 3-digit BCD down counter for minutes and seconds, 782
  - MOD-12 design, 770–771
  - MOD-60 graphic block symbol, 774
  - mode, 119
  - multiplexers, 716–719
  - nesting, 754
  - NEXT, 518–519
  - one-shots, 521–528
  - PRESENT, 519
  - projects using, 744–792
  - registers, 513–519
  - representing data, 202–207
  - retriggerable, edge-triggered one-shots in, 524–525
  - ring counters, 519–520
  - scalars, 202
  - schematic diagram, 119
    - D latch, 314
    - NAND latch, 313
  - simulation of basic counter, 467
  - simulation of full-featured counter, 471–472
  - small-project management, 746–747
    - definition, 746, 779–780
    - problem decomposition, 746–747, 749–750, 757, 780–784
    - strategic planning, 746, 749–750, 757, 780–784
    - synthesis and testing, 747, 750–751, 784
    - system integration and testing, 747, 784
  - state transition description methods, 461
  - stepper motor driver project, 747–755
    - full-step sequence, 751
    - half-step sequence, 748–749
    - problem decomposition, 749–750
    - problem definition, 748–749
    - strategic planning, 749–750
    - synthesis and testing, 750–754
    - wave-drive sequence, 748
  - structural level of abstraction, 323
  - syntax and format, 118–121
  - TABLE, 706
  - timing simulation, 201
  - truth tables, 207–210
  - type, 119
  - wiring modules together, 473–480
- Hertz, 251
- Hexadecimal
- addition, 360–361
  - arithmetic, 360–362
  - number system, 41–46
  - representation of signed numbers, 362–363
  - subtraction, 361–362
- Hierarchical design, 199
- Hierarchy, 764–766
- High capacity programmable logic devices (HCPLDs), 944
- generations, 958–959
- High-speed bus interface logic, 605–606
- High-speed CMOS 74HC/HCT, 589
- High-state noise margin (VNH), 556
- Hold time ( $t_H$ ), 253–254, 267, 279
- I**
- IC synchronous counters, 430–440
- IEEE/ANSI symbol for
- open-collector/drain outputs, 601
  - tristate outputs, 605
- IF/ELSE, 211–212
- IF/THEN, 211
- IF/THEN/ELSE using AHDL, 212–213
- Implementing logic circuits with PLDs, 117–118
- Implications of DeMorgan's theorems, 97–98
- Incomplete address decoding, 923–924
- Indeterminate
- logic level, 181
  - voltages, 177
- Inhibit circuits, 79
- Input
- currents for standard devices with a supply voltage of 5 V, 611
  - sequence detection, 273–274
  - unit, 28
- Integrated-circuit logic families, 551–637
- ALU(s), 378–382
- add operation, 379
  - AND operation, 380
  - CLEAR operation, 379
  - EX-OR operation, 380
  - expanding the, 381
  - OR operation, 380
  - PRESET operation, 380
  - SUBTRACT operation, 380
- basic characteristics, 173–180
- bipolar, 174–175
- defined, 551–552
- interfacing, 609–613
- summary, 621
- terminology, 552–561
- unipolar, 174–175
- Integrated-circuit packages, 558–561
- ball grid array, 560
  - common, 558
  - dual-in-line (DIP), 558, 560
  - gull-wing, 560
  - J-shaped leads, 560
  - land grid array (LGA), 560
  - lead pitch, 558
  - low-profile five-pitch ball grid array (LFBGA), 560
  - plastic leaded chip carrier (PLCC), 560
  - quad flat pack (QFP), 560
  - shrink small outline package (SSOP), 560
  - small outline IC (SOIC), 560
  - surface-mount technology, 558
  - thin quad flat pack (TQFP), 560
  - thin shrink small outline package (TSSOP), 560
  - thin very small outline package (TVSOP), 560
- Integrated-circuit parallel adder, 373–375
- Integrated-circuit registers, 494–501
- parallel in/parallel out (74ALS174/74HC174), 494–495
  - parallel in/serial out (74ALS165/74HC165), 498–500
  - serial in/parallel out (74ALS164/74HC164), 500–501
  - serial in/serial out (74ALS166/74HC166), 496–498
- Integrated-circuit shift-register counters, 506
- Intellectual property (IP), 947



- Interfacing
    - 5-V TTL and CMOS, 611
    - high-voltage outputs driving low-voltage loads, 614
    - integrated circuit, 609–613
    - logic ICs, 610
    - low-voltage outputs driving high-voltage loads, 614
    - mixed-voltage, 614–615
    - not needed, 610
    - required, 610
    - with the analog world, 794–865
      - summary, 852–853
  - Intermediate signals, 112–113
  - Interpolation filtering, 848
  - Introduction to digital 1s and 0s, 4–8
  - Introductory concepts, 2–35
  - Invalid voltage levels, 557
  - Inversion, 80–81 (*see also* NOT operation)
  - Inverted flip-flop output, 238
  - Inverter, 81
    - circuits containing, 83–84
    - controlled inverter, 166
    - response to slow noisy inverter, 295
  - Inverting tristate buffer, 603
- J**
- J-K excitation table, 448
  - J-K flip-flops, 258–260
  - J-lead packages, standard memory, 913
  - Jam transfer, 277, 429
  - JEDEC standard, 197
  - JTAG interface, 197, 955
  - Johnson counter, 503–505
    - decoding, 505
- K**
- Karnaugh map
    - complete simplification process, 157–160
    - don't-care conditions, 161–162
    - filling from output expression, 160–161
    - format, 152–154
    - looping, 154–157
    - method, 152–163
    - simplification, 368
    - summary, 163
- L**
- Labeling
    - active-LOW signals, 110
    - bistate signals, 110–111
  - Ladder, R/2R, 808–809
  - Land grid array (LGA), 560
  - Large-scale integration (LSI), 174, 551
  - Latches, 27, 239–249, 262–264 (*see also* Flip-flops)
    - resetting, 241
    - S-R, 242
    - setting, 241
  - Latch-up, 594
  - Latency, 907
  - Least significant bit (LSB), 22
  - Least-significant digit (LSD), 19
  - Level triggered, 12
  - Libraries of parameterized modules (LPMs), 308
  - Light-emitting diodes (LEDs), 648–649
    - common-anode *vs.* common-cathode, 648–649
  - Limitations of digital techniques, 18–19
  - Linear buffers, 928
  - Linearity error (of a DAC), 810
  - Liquid crystal displays (LCDs), 649–652
    - backplane, 650
    - driving a, 650–651
    - types, 651–653
  - Loading
    - factor, 553
    - TTL, 573–578
  - Local signals, VHDL, 123
  - Logic
    - diagram using Quartus II schematic capture, 179
    - function generation, 672–673
    - level, 71
    - primitive, 315
    - probe, how to use, 181, 617–618
    - pulser, how to use, 181, 617–618
    - standard, (PLD), 944
    - states, 5
  - Logic circuits
    - analysis using a table, 85–86
    - analyzing, 108–110
    - and technology, 13–14
    - arithmetic, 363–364
    - connection diagrams, 178–179
    - defined, 13
    - describing, 68–135
      - algebraically, 82–84
    - disabled, 170–172
    - enabled, 170–172
    - evaluation of outputs, 84–86
    - implementing from Boolean expressions, 87–88
    - implementing with PLDs, 117–118
    - interface, 609–613
    - pulse-shaping circuit, 413
    - pulse-steering, 172
  - Logic gates, 71–135
    - alternate representation, 102–105
    - AND, 78
    - Boolean theorems, 92–95
    - DeMorgan's theorems, 95–98
    - evaluation of outputs, 84–86
    - NAND, 88–92
    - NOR, 88–92
    - NOT circuit (INVERTER), 81
    - OR, 74–77
    - summary of methods to describe, 112–114
    - truth tables, 72–73
    - which representation to use, 105–111
    - XNOR, 165–168
    - XOR, 163–164
  - Logic operations, 71
    - on bit arrays with HDLs, 390–391
  - Logic signals
    - labeling active-LOW, 110
    - labeling bistate, 110–111
  - Logic symbol interpretation, 104
    - summary, 105
  - Logical complementation or inversion
    - (NOT operation), 80–81
  - Look-ahead carry, 373
  - Look-up table (LUT), 945
  - Looping, 154–157
    - octets, 156–157
    - pairs, 154–155
    - quads, 155–156

- Low-voltage (74LV), 596
    - BiCMOS Technology (74LVT), 596
    - CMOS (74LVC), 595–596
    - series characteristics, 596
    - voltage technology, 594–597
  - Low-power Schottky TTL 74LS series (LS-TTL), 571
  - Low-state noise margin (VNL), 556
  - Low-voltage differential signaling (LVDS), 607
  - LPMs, 308
  - LUT (Look-up table), 945
    - functional block diagram, 957
  - LVDS (low-voltage differential signaling), 607
- M**
- Machines, state, 481–492
  - Macrofunction, 384
  - Magnetic storage, 915–916
  - Magnetoresistive RAM, 916
  - Magnitude comparator, 687–690, 720–721
    - applications, 689
    - cascading inputs, 688–689
    - data inputs, 688
    - outputs, 688
  - Magnitude of binary numbers, 344
  - Major parts of a computer, 28–29
  - Mask-Programmed
    - gate arrays (MPGAs), 943
    - ROM (MROM), 885
  - Maximum clocking frequency ( $f_{MAX}$ ), 268
  - Maxplus2 functions, VHDL, 206
  - Mealy model, 481
  - Medium-scale integration (MSI), 174, 551
  - Megafunction LPMs for arithmetic circuits, 385–386
  - Megafunction registers, 509–512
  - MegaWizard settings, 509–510
  - Memory, 27, 864–939
    - auxiliary, 868, 871
    - bipolar NMOS and CMOS static RAM cells, 899–900
    - bootstrap, 894
    - cache, 871, 926–927
    - capacity, 869
    - cell, 869
    - CMOS static RAM cell, 900
    - compact disk, 868
    - connections, CPU, 875–877
    - density, 869
    - devices, 866–939
    - dynamic, devices, 871
    - embedded microcontroller program, 894
    - enable, 873
    - expanding capacity, 917–925
    - expanding word size, 917–925
    - fetch operation, 870
    - first-in, first-out (FIFO), 927–928
    - flash, 889–893
      - architecture, NAND, 893
      - bulk erase, 889
      - command register, 890
      - functional diagram, 891
      - IC, typical CMOS flash memory, 890–891
      - NAND, 891–892
      - NOR, 891–892
      - sector erase, 889
      - tradeoffs, 889
    - fold-back, 924
    - general operation, 872–875
    - magnetoresistive random access (MRAM), 916
    - main, 868, 871
    - map, 924
    - mass, 868, 894
    - module, 925
    - modules, DRAM, 913–914
    - NMOS static RAM cell, 900
    - nonvolatile, 870–871, 877
    - other technologies
      - blu-ray, 916–917
      - magnetic storage, 915–916
      - optical, 916–917
    - random-access (RAM), 870
    - read-only (ROM), 871
    - read/write, 871
    - sequential-access, 870–871
    - special functions, 925–928
      - cache memory, 926–927
      - first-in, first-out, 927–928
    - static, devices, 871
    - store operation, 870
    - summary, 928–929
    - terminology, 868–871
    - unit, 28
    - volatile, 870
    - word, 870
    - working, 868, 871
  - Metastable states, 253, 292
  - Microcomputer
    - application, 293–294
    - defined, 29
    - input unit, 28
    - memory unit, 28
    - output unit, 29
  - Microcontroller, 29
  - Microprocessor, 29
    - Digital signal processing (DSP), 942–943
    - READ operation, 876
    - WRITE operation, 876
  - Minuend, 353
  - Mixed systems, digital and analog, 17–19
  - Mixed-voltage interfacing, 614–615
    - high-voltage outputs driving low-voltage loads, 614–615
    - low-voltage outputs driving high-voltage loads, 614
    - voltage-level translator, 614
  - MOD number, 284, 412
    - changing, 421
    - general procedure, 423–424
    - Johnson counter, 503–505
    - ring counter, 502–503
  - Monostable multivibrator, 296–299 (*see also* One-shot)
  - Monotonicity (of a DAC), 811
  - Moore model, 481
    - simulation of, 484–485
    - traffic light controller, 485–488
  - MOS
    - electrostatic discharge (ESD), 593
    - FETs, 582–585
    - logic family, 582–585
    - NMOS, 583–585
    - static sensitivity, 593–594
  - MOS Floating-gate, 891
  - MOSFET, 582–585
    - basic switch, 583–584
    - CMOS, 586–588
    - digital circuits, 583–584

MOSFET (*continued*)

- FGMOSFET, 891
  - N-MOS, 583
  - P-MOS, 585
  - Most significant bit (MSB), 22
  - Most significant digit (MSD), 19
  - MSI logic circuits
    - BCD-to-7 segment decoder/drivers, 647–648
    - BCD-to-decimal decoder, 644
    - data busing, 692–693
    - decoders, 639–647
    - demultiplexers (DEMUXs), 673–683
    - encoders, 653–659
    - liquid crystal displays (LCDs), 649–652
    - multiplexers (MUX), 662–673
    - summary, 724–725
    - tristate registers, 698–699
  - Multiple-emitter input transistor, 561
  - Multiplexers (MUX), 662–673, 716–719
    - applications, 668–673, 689–690
    - control sequence, seven-step, 671
    - eight-input, 664–665
    - four-input, 664
    - operation sequencing, using, 669–672
    - quad two-input, 666–667
    - time division, 679–683
    - two-input, basic, 663–664
  - Multiplexing
    - ADC, 844–845
    - address (in DRAM), 905–908
  - Multiplication
    - in the 2<sup>n</sup>-complement system, 357
    - of binary numbers, 355–356
- N**
- NAND gate, 88–92
    - alternate representation, 102–105
    - CMOS, 587
    - counter decoding, 440–443
    - defined, 90
    - internal circuitry of the edge-triggered J-K FF, 259–260
    - internal circuitry of the edge-triggered S-R FF, 256–257
    - latch flip-flop, 239–244
    - TTL, 561
    - universality of, 99–102
    - which representation to use, 105–111
  - Negation, 347–348
  - Negative edge, 9
  - Negative-going threshold voltage ( $V_{T-}$ ), 294–296
  - Negative-going transition (NGT), 251
  - NEXT state, 444–447
  - Nibble, 52–53
  - Nios<sup>®</sup> II, 947
  - NMOS
    - logic circuits, 583
  - NMOS static RAM cell, 900
  - Noise, 17, 303
    - immunity, 555
  - Noise margin, 555
    - CMOS, 591
    - DC, 555
  - Nonretriggerable one-shot, 296–297
  - Nonvolatile memory, 886–887, 889, 894
  - NOR gate, 88–92
    - alternate representation, 102–105

- CMOS, 587–588
  - defined, 88
  - latch, 245–247
  - NMOS, 583
  - universality of, 99–102
  - which gate representation to use, 105–111
- Normal flip-flop output, 238
- NOT circuit (INVERTER), 80–81
  - alternate representation, 102–105
  - circuits containing, 83–84
  - controlled inverter, 166
  - defined, 81
  - DeMorgan's theorems, 95–98
  - implementing from Boolean expressions, 87–88
  - symbol, 81
  - which representation to use, 105–111
- NOT operation, 71, 80–81
- Number circles and binary arithmetic, 354–355
- Number systems, 19–23 (*see also* Binary system)
  - and codes, 36–67
  - applications, 59–61
  - binary, 21–22
  - decimal, 19–20 (*see also* Decimal system)
  - digital, 19–23
  - hexadecimal, 41–46
  - summary, 61
- Numerical representations, 14–16

**O**

- Observation/analysis, troubleshooting
  - process, 660
- Octal-to-binary encoders, 654
- Octets, looping, 156–157
- Odd-parity method, 57
- Offset error, 811
- 1's-complement form, 344
- One-shot (monostable multivibrator), 296–299
  - actual devices, 298
  - AHDL, 522, 525–526
  - HDL, 521–528
  - retriggerable, edge-triggered in HDL, 524–525
  - VHDL, 523–525, 526–528
- One-time programmable ROM (OTP), 886, 945
- Open-collector
  - buffer/drivers, 600–601
  - outputs, 597–601
- Open-collector buffer/drivers, 600–601
- Open-drain
  - buffer/drivers, 600–601
  - outputs, 597–601
- Operation
  - fetch, 870
  - refresh, 871
- Operational amplifier (in a DAC), 805
- Optical memory, 916–917
- OR gate, 74–77
  - alternate logic-gate representation, 102–105
  - Boolean theorems, 92–95
  - defined, 74
  - implementing from Boolean expressions, 87–88
  - symbol, 74
- OR operation, 74–77
  - summary, 75–76
  - which representation to use, 105–111
- Organizational hierarchical chart, 200
- Oscillator, Schmitt-trigger, 299

- OTP (one-time programmable ROM), 886, 945
- Output
- buffers, ROM, 880
  - currents for standard devices with a supply voltage of 5 V, 611
  - enable (*OE*), 874
  - enable time ( $t_{OE}$ ), 882
  - loading, 188
  - unit, 29
- Overflow bit, 369
- Override inputs, 265
- P**
- Pairs, looping, 154–155
- Parallel
- loading, 428
  - parallel in/parallel out 74ALS174/74HC174, 494–496
  - parallel in/serial out 74ALS165/74HC165, 498–500
  - parallel-to-serial conversion, 669
  - transmission, 25–27
- Parallel binary adder, 364–366
- 2's-complement circuits, 375–378
  - carry propagation, 372–373
  - complete, with registers, 369–371
  - integrated circuits, 373–375
  - troubleshooting case study, 382–384
- Parallel data transfer, 262, 277–278
- versus* serial transfer, 281
- Parallel and serial transmission, 25–27
- trade-offs between, 26
- Parameterizing the bit capacity of a circuit, 394–397
- Parasitic, 594
- Parity
- bit, 57–58
  - checker, 169–170
  - checking, 57, 169
  - errors
    - single-bit, 57
    - two-bit, 57
  - generation, 169
  - generator, 169–170
  - method for error detection, 56–58
- Percentage resolution, 801–802
- Period, 11–12, 12, 251
- periodic digital signals, 11
- Pipeline ADC, 840–842
- PIPO (parallel in/parallel out), 494
- PISO (parallel in/serial out), 493
- PISO register, AHDL, 515–516
- PISO register, VHDL, 515–518
- Pixels, 652
- Plastic leaded chip carrier (PLCC), 560
- Positional-value system, 19
- Positive-going threshold voltage ( $V_{T+}$ ), 294
- Positive-going transition (PGT), 251
- Power
- down (in MROM), 885
  - requirements for digital ICs, 554–555
  - supply decoupling, TTL, 582
- Precision reference supply, 806
- Prescaler (digital clock using HDL), 765
- PRESENT state, 444–447
- PRESET, 265
- Presettable counters, 428–430
- Priority encoders, 655
- Problem decomposition, (using HDL), 746, 749–750, 757
- Product-of-sums, 138–139
- Program, defined, 17, 28
- Programmable Logic Devices (PLDs), 115, 194–202, 307–308, 940–961
- architecture/s, 950–955
    - FPGA (see Field-programmable gate arrays)
    - FPLA (field-programmable logic array), 954
    - programmable array logic (PAL), 945, 951–954
    - PROMs, 950–951
      - summary, 959
  - CPLD, 944
  - design and development process, 200–201
    - test vectors, 200
    - top-down, 200
  - development cycle flowchart, 201
  - development software, 197–199
    - AHDL, 198
    - compilers, 117
    - timing simulation, circuit described in HDL, 201
    - VHDL, 198
  - examples of, 943–947
  - FPGA (see Field-programmable gate arrays)
  - fundamentals of PLD circuitry, 948–950
  - generic array logic (GAL), 954–955
  - hardware, 195–196
  - HCPLD, 944
  - hierarchical design, 199
  - look-up table (LUT), 951
  - macrocell, 945
  - mask-programmed gate arrays (MPGAs), 941
  - maxplus2
  - megafunction, 307–308
  - more on, 944–947
  - one-time programmable (OTP), 945
  - organizational hierarchical chart, 200
  - primitives, 307
  - product lines, 948
  - programmable array logic (PAL), 945
  - programmer, 197
  - programming, 196–197, JEDEC standard 197
    - development board, 197
    - JTAG, 197
    - zero insertion force socket (ZIF), 196–197
  - sequential circuits using schematic entry, 307–310
  - SPLD, 944–945
  - standard J lead memory packaging, 913
  - symbolology, 949
  - universal, 197
- Programmable ROMs (PROMs), 885–886
- Programmer, 197
- Programming languages, computer, 115–116
- Projects, using HDL, 744–792
- digital clock, 761–778
  - frequency counter, 785–779
  - keypad encoder, 755–761
  - management, 746–747
  - microwave oven, 778–785
  - stepper motor driver, 747–755
- Propagation delay(s), 111
- flip-flop, 268
  - in asynchronous counters, 414–416
  - integrated circuits, 554–555
  - TTL NAND gate, 568
- Pull-down transistor, TTL, 564
- Pull-up transistor, TTL, 564

- Pulse(s), 249–250
  - leading edge, 249
  - negative, 249–250
  - positive, 249–250
  - shaping circuit, 413
  - steering circuit, 172, 257
  - trailing edge, 249
- Q**
- Quad
  - flat pack (QFP), 560
  - two-input multiplexers, 666
- Quad looping, 155–156
- Quadrature encoders, 50–51, 274, 286–292
- Quantization error, 820–821
- Quartus II, 115
  - decoders, 641–642
  - magnitude comparator, 690
  - maxplus2 library, 509
  - mux, 662–663
  - schematic capture, 179
- Quartz
  - crystal, 302
  - watch, 283
- Quasi-stable state, 296
- R**
- R/2R ladder digital-to-analog converters, 808–809
- RAMs (random-access memories)
  - architecture, 897–899
  - capacity expansion, 920–923
  - defined, 870
  - dynamic devices, 871
  - magnetoresistive, 916
  - power-up self-test, 957–958
  - semiconductor, 896
  - static (SRAM), 899–902
  - word size expansion, 917–925
- Read operation
  - CPU, 876
  - defined, 898
  - RAM, 896
- Read/Write memory (RWM), 871
- Reconstructing a digitized signal, 824–825
- Reflective LCDs, 649–650
- Refresh counter, 912
- Refreshing, DRAM, 903, 910–912
- Register array, 880
- Registers, 276, 408–528
  - accumulator, 364
  - address pointer, 927
  - and counters, 408–528
  - complete parallel adder with, 369–371
  - data transfer, 493
  - HDL, 513–519
  - megafunction, 509–512
  - notation, 370–371
  - sequence of operations, 371
  - shift left operation, 281
  - tristate (74ALS173/HC173), 698–699
- Repeated division method, 39–40
- Representing
  - binary quantities, 23–25
  - data in HDL, 202–207
  - signed numbers, 343–350
  - using 2's-complement, 345–346
- RESET, 265
- Resetting a flip-flop, defined, 241
- Resolution, 800–803
  - ADC, 820–822
  - DAC, 800–801, 810
- Resolution, percentage, 801–802
- Retriggerable one-shot, 297
- Ring counter, 502–503, 658
  - in circuit, 658
  - starting a, 502
  - state diagram, of, 502, 757
- ROM (read-only memory), 877–879
  - applications, 894–896
  - architecture, 879
  - output buffers, 880–881
    - block diagram, 877–878
    - burning-in, 877
    - column decoder, 879
    - defined, 880–881
    - erased, 877
    - mask-programmed, 883–885
    - one-time programmable (OTP), 886
  - programming, 877
  - READ operation, the, 878
  - row decoder, 879
  - timing, 881–882
  - types of, 882–888
- Row address strobe (RAS), 906
- Run length limited (RLL) encoding, 916
- S**
- SAM (sequential-access memory), 870–871
- Sample-and-hold circuits, 843–844
- Sampling, 822
  - frequency, 825
- SBD (Schottky Barrier Diode), 571
- Schmitt-trigger
  - devices, 294–296
  - oscillator, 299
  - response to slow noisy input, 295
- Schottky
  - barrier diode (SBD), 571
  - TTL 74S series, 571
- SDRAM (synchronous DRAM), 914
- Sector erase, 889
- Security monitoring system, 675–677
- Select inputs, (in MUXs), 662–663
- Sense amplifier (in DRAM), 905
- Sequential-access memory (SAM), 870–871
- Sequential circuit design, 447–455
- Sequential circuits, 271
  - design, 447–455
  - in PLDs using schematic entry, 307–310
  - using HDL, 311–314
  - using schematic entry, 307–310
- Sequential logic systems, troubleshooting, 506
- Serial
  - ADCs, 826
  - serial in/serial out, 74ALS166/74HC166, 496–498
  - transmission, 25–27
- Serial data transfer, 278–281
  - between registers, 280
- Sets, HDL, 390
- Setting the flip-flop
  - and resetting simultaneously, 241
  - latch, 241

- Setup time ( $t_s$ ), 252–254, 267
- 74 TTL series, 570, 571
- 74AC series, 176, 589
- 74ACT series, 176, 589
- 74AHC series, 589
- 74AHCT series, 589
- 74ALB series, 596
- 74ALS TTL series, 571–572
- 74ALVC series, 595
- 74ALVT series, 596
- 74AS TTL series, 571–572
- 74AUC series, 595
- 74AVC series, 595
- 74C series, 176
- 74CBT series, 596
- 74CBTLV series, 596
- 74F-Fast TTL series, 571
- 74GTLP series, 596
- 74HC series, 176, 589
- 74HCT series, 176, 589
- 74LS TTL series, 176, 571
- 74LV series, 596
- 74LVC series, 595–596
- 74LVT series, 596
- 74S TTL series, 571
- 74SSTV series, 596
- 74 TTL series, 176
- 74TVC series, 596
- Shift-register counters, 502–506
- Shift registers, 278–281, 494–496, 496–498, parallel in/parallel out 74ALS174/74HC174, serial in/serial out 74ALS166/74HC166
  - left, 281
  - Shrink small outline package (SSOP), 560
- Sigma
  - delta modulation, (ADC), 838–840
- Sigma ( $\Sigma$ ), 373
- Sign
  - bit, 343
  - extension, 347–348
  - magnitude system, 344
- Signal
  - alias, 826
  - contention, 185
  - duty cycle, 413–414
  - flow, 411
- Signed numbers, 343–350
  - in sign-magnitude form, 344
- SIMM, 913–914
- Simple programmable logic devices (SPLDs), 944–945
- Simulation of state machines, 484–485
- SIPO (serial in/parallel out register), 493
- SISO (serial in/serial out register), 493
- SISO register, AHDL, 514–515
- SISO register, VHDL, 514
- Skew, clock, 305–307
- Small outline integrated circuit (SOIC), 560
- Small-project management (using HDL), 746–747
  - definition, 746, 779–780
  - problem decomposition, 746–747
  - strategic planning, 746–747
  - synthesis and testing, 747
  - system integration and testing, 747
- Small-scale integration (SSI), 174, 551
- SODIMM, 913
- Solder bridges, 187
- Special memory functions, 925–928
- Spike, 421
- SPLDs, 944–945
- S-R latches, 242
- Staircase
  - test, of a DAC, 815
  - waveform, of a DAC, 800
- Standard
  - cell ASICs, 944
  - logic, (PLD), 944
- State
  - descriptions in AHDL, 461–462
  - descriptions in VHDL, 463–466
  - machines, 481–492
    - simulation of, 484–485
    - traffic light controller, 485–488
  - table, 283
  - transition description methods, HDL, 461
  - transition diagram, 284, 421
    - Mod-6 counter, 421
    - synchronous counter, 449
- Static accuracy test, of a DAC, 815
- Static RAM (SRAM), 899–904
  - actual chip (MCM6264C), 902–904
  - read cycle, 900–902
  - timing, 900
  - write cycle, 902
- Step-size, 800–801
- Stepper motor
  - control, 452–454
  - driver project (HDL), 747–755 (*see also* HDL)
  - universal, interface circuit, 750
- Straight binary coding, 46
- Strategic planning (using HDL), 746–747, 749–750
- Stub Series Terminated Logic (74SSTV), 596
- SUBDESIGN, 120, 204–205
- Subpixels, 652
- Substrate, 173
- Subtraction
  - BCD, 359
  - binary, 343
  - hexadecimal, 361–362
  - 2's-complement circuits, 375–378
  - 2's-complement system, 352–355
- Subtrahend, 353
- Successive-approximation ADC, 826–834
- Sum bit, 365
- Sum-of-products form, 138–139
- Switch
  - bilateral, 607–609
  - debouncing, 243
  - encoders, 656–659
- Synchronization, flip-flop, 272–273
- Synchronous
  - control inputs, 252
  - counter design with D FF, 454–455
  - inputs, 252, 265
  - presetting, 430
  - systems, 231, 251
  - transfer, 276
- Synchronous (parallel) counters, 416–419
  - actual ICs, 418–419
  - advantages over asynchronous, 419
  - design, 447–455
    - stepper motor control, 452–454
  - down and up/down, 426–428

- Synchronous (parallel) counters (*continued*)
    - operation, 418–419
    - presetable, 428–430
  - Synchronous data transmission system, 677–679
    - receiver operation, 679
    - system timing, 679
    - transmitter operation, 678–679
  - Synthesis and testing (using HDL), 747, 750–751
  - System integration and testing (using HDL), 747
- T**
- Table
    - analysis using, 85–86
    - circuit excitation, 448–419, 451
    - J-K excitation, 448–449
    - look-up (LUT), 945
    - state, 283
  - Temperature-limit detector using an LM339, 616
  - Temporary storage, RAM, 896
  - Test vectors, 200
  - Theorems
    - Boolean, 92–95
    - DeMorgan's, 95–98
    - multivariable, 93–94
  - Thin Film Transistor (TFT) LCD, 653
  - Thin quad flat pack (TQFP), 560
  - Thin shrink small outline package (TSSOP), 560
  - Thin very small outline package (TVSOP), 560
  - 3-line-to-8-line decoder, 644
  - TI signal switch (TS switch), 596
  - Tied-together inputs, TTL, 579
  - Time Division Multiplexing, 679–683
  - Timer 555 used as an astable multivibrator, 299–302
    - (*see also* Astable multivibrator)
  - Timing diagrams, 7, 445
    - simplified bus, 701
  - Timing problems in flip-flop circuits, 269–271
  - Toggle mode, 258
  - Toggles, 22
  - Top-down hierarchical design (digital clock using HDL), 764–766
  - Totem-pole output circuit, 564–565
  - Tradeoffs (for nonvolatile memories), 889
  - Transducer, 796
  - Transfer data, register, 493
  - Transfer operation, data, 276
  - Transition diagram, state, 284
  - Transition states, 421
  - Translation Voltage Clamp (74TVC), 596
  - Transmission gate, CMOS, 607–609
  - Transparent latch, (D-Latch), 262–264
  - Trigger input, 256
  - Tristate
    - data bus, 605
    - outputs, 602–605
    - registers (74ALS173/HC173), 698–699
      - connected to data bus, 698
  - Troubleshooting, case study, fault
    - basic steps, 180–181
    - case study
      - parallel binary adder/subtractor, 382–384
    - counters, 506–509
    - decoders, circuit with, 653–654
    - digital systems, 180–181, 617–618 (*see also* Digital systems)
    - digital-to-analog converters, 814–815
    - divide-and-conquer, 660
    - fault
      - correction, 180
      - detection, 180
      - external IC, 186–190
      - isolation, 180
    - finding shorted nodes, 618
    - flip-flop circuits, 302–307
      - internal IC faults, 182–186
      - logic pulser and probe to test a circuit, using, 618
      - observation/analysis, 660
      - open inputs, 303–304
      - shorted outputs, 304–305
    - parallel binary adder/subtractor, 382–384
    - prototyped circuits, 190–194
    - security monitoring system, 675–677
    - sequential logic systems, 506
    - synchronous data transmission system, 677–678
    - tools used in, 181, 617–618
    - tree diagram, 685
  - Truth tables, 72–73
    - using AHDL, 208–209
    - using HDL, 207–210
    - using VHDL, 209–210
  - TTL logic family, 14, 175–176, 561–566
    - active pull-up action, 565
    - ALS series, 176
    - AS series, 176
    - biasing inputs LOW, 580
    - characteristics, 569–573
    - circuit operation-HIGH state, 562
    - circuit operation-LOW state, 561–562
    - comparison of series characteristics, 572
    - current ratings, 576
    - current transients, TTL, 581–582
    - current-sinking action, TTL, 564
    - data sheets, 566–569
    - defined, 175
    - fan-out, 573–578
    - fast series (74F), 571–572
    - ground, 176–177
    - input voltages, 590
    - interfacing 5 V and CMOS, 611
    - INVERTER circuit, 175
    - loading, 573–578
    - logic-level voltage ranges, 177
    - low-power Schottky 74LS series (LS-TTL), 571
    - LS series, 176
    - maximum voltage ratings, 568
    - NAND gate, basic, 561–562
    - NOR gate, basic, 566
    - open-collector outputs, 597–602
    - other characteristics, 578–582
    - output voltages, 590
    - power, 176–177
    - power dissipation, 568
    - propagation delays, 568
    - S series, 176
    - Schottky 74S series, 571
    - series characteristics, 572–573
    - standard 74 series, 572
    - subfamilies, 175, 569–573
    - summary, 565
    - supply (power) voltage, 176–177, 567
    - temperature range, 567

- tied-together inputs, 579
  - totem-pole output circuit, 561, 565
  - tristate, 602–605
  - Twisted-ring counters, 504
  - unconnected inputs (floating), 177–178, 578
  - unused inputs, 578–579
  - voltage levels, 567–568
  - Two-input multiplexer, basic, 663–664
  - 2's-complement
    - addition, 351–352
    - circuits, 375–378
      - addition, 375–378
      - subtraction, 375–378
    - form, 344
    - special case representation, 348–350
    - subtraction, 352–355
    - system, 344
      - addition and subtraction, combined, 377–378
      - multiplication, 355–356
  - Types of computers, 29–30
    - embedded controller, 30
    - microcomputer, 29
    - microcontroller, 29
    - microprocessor, 29
  - Types of LCDs, 651–653
  - Typical ADC architectures for applications, 842–843
  - Typical CMOS flash memory IC, 890–891
- U**
- Unasserted levels, 110
  - Unconnected inputs
    - CMOS, 177–178, 593
    - TTL, 177–178, 578–579
  - Unipolar digital ICs, 174–175 (*see also* CMOS logic family)
  - Universal programmers, 197
  - Universal shift register, AHDL, VHDL, 517–518
  - Universality of NAND gates and NOR gates, 99–102
  - Usefulness of hex, 44–45
  - UV light, EPROMs, 887
- V**
- VERSA Module Eurocard (74VME), 596
  - VHDL, 115–116, 466–467, n-bit adder/subtractor.
    - 396–397
    - adder/subtractor, 396–397
    - ARCHITECTURE, 121, 205, 476, 479
    - BCD-to-binary code converter, 723
    - BEGIN, 121, 463, 466
    - behavioral description of a counter in, 466
    - BIT, 121, 206, 318
    - BIT\_VECTOR declarations, 205–207, 209, 390, 392, 466, 475
    - Boolean description using, 120
    - BUFFER, 475
    - cascading BCD counters, 477–478
    - CASE, 218, 463–464, 474, 484, 488, 491, 518, 759
    - code converter, 723
    - comments, 123–124
    - comparator, 723
    - complete clock, 777
    - COMPONENT(s), 316–318, 475, 480, 488, 776
      - conditional signal assignment statement, 715
      - declaration, 317
      - graphic representation using, 317
      - HDL circuits with multiple, 320–323
      - library, 316–318
    - concurrent assignment statement, 120
    - CONSTANT, 396
    - converter, 723
    - D latch, 314
    - data types, common, 206
    - decoding the MOD-5 counter, 474–475
    - decoder(s), 708
      - drivers, 710–711
      - full-step sequence, 751
    - demultiplexers, 718–719
    - design file, 209
    - digital clock project, 761–778 (*see also* HDL)
    - DOWNT0, 390, 759
    - driver, 710–712
    - eight-bit adder, 392–393
    - ELSFIF, 215–217
    - encoder, 715–716
    - END, 121, 464
    - ENTITY, 121, 205, 209, 322, 463, 474–475, 479–480
    - enumerated type, 484
    - essential elements in, 121
    - EVENT, 319, 322, 464
    - flip-flops, 318–320
      - J-K flip-flop, 318–320
      - MOD-8 ripple counter, 321–322
      - simulation, 319
    - frequency counter project, 785–789
    - full-featured counter, 469–470
    - full-step sequence, 751
    - IF/THEN/ELSE, 213–214, 484, 491, 713, 721
    - IN, 205
    - INTEGER, 206, 213, 392, 466, 469, 712–713
    - intermediate signals in, 124
    - J-K flip-flop, 318–320
      - simulation, 319
    - keypad encoder project (HDL), 755–761
      - (*see also* HDL)
      - simulation, 761
      - solution, 759–761
    - libraries, 206, 715
      - components, 316–318
      - of parameterized modules, 307
    - local signals, 123–124
    - LPMs, 308
    - macrofunctions, 206
    - magnitude comparator, 721
    - MAP, 475, 480, 488
    - maxplus2 functions, 206
    - MOD-5 counter, 464
    - MOD-6 counter, 766, 776
      - graphic block symbols, 773
      - simulation, 768
    - MOD-8 counter, 750
      - simulation, 750
    - MOD-10 counter, 768–769, 776
    - MOD-12 counter, 771–774
      - graphic block symbols, 774
      - simulation, 773
    - MOD-60 counter, 776
    - MOD-100 BCD counter, 479–480
    - module integration, 776–778
    - multiplexers, 718–719
    - NAND latch, 313
    - nonretriggerable one-shot, 523–528
    - objects, 206
    - one-shots, simple, 523–525



- VHDL (*continued*)
    - PACKAGE, 396
    - PISO register, 516–518
    - PORT, 121
      - MAP, 318, 323
    - PROCESS, 214, 318, 322, 463–464, 466, 469–470, 474, 491, 518, 526, 712, 721, 759, 772
    - RANGE, 213, 466
    - retriggerable, edge-triggered one-shot, 526–528
    - ripple-up counter (MOD-8), 321–323
    - SELECT, 209
    - sensitivity list, 214
    - shift register, universal, 517–518
    - SIGNAL, 123–124, 209–210, 319, 322, 390, 463, 710, 715, 772
    - simulation, 523–524, 527–528
    - simulation of full-featured counter, 469–471
    - simulation testing, 754
    - SISO register, 514
    - state descriptions in, 463–466
    - state machine, simple, 483
    - STD\_LOGIC, 206, 317
    - STD\_LOGIC\_VECTOR, 206
    - stepper driver, 753
    - stepper motor driver project, 747–755
    - traffic light controller, 488–491
    - truth tables, 209–210
      - concatenating, 209–210
      - selected signal assignments, 209–210
    - TYPE, 483
    - values, table, 209
    - VARIABLE, 319, 463, 526, 710, 715, 772
    - WHEN, 708
    - WITH, 209
  - VHDL (very-high-speed integrated circuit hardware description language), 115–116
  - VLSI (very-large scale integration), 174, 551
  - Volatile memory, 870–871
  - Voltage
    - comparators, 614–616
    - controlled oscillator, linear (VCO), 838
    - level translator, 614
    - levels, invalid, 556–557
    - parameters for digital ICs, 552–553
    - to frequency ADC, 838
- W**
- Wave-drive sequence (HDL stepper-motor), 748
  - Wired-AND connection, 599–600
  - Word, 52–53
    - size, 53
    - Write cycle, 293
  - Write cycle
    - address setup time, 902
    - data hold time, 902
    - data setup time, 902
    - time, 902
  - Write enable ( $\overline{WE}$ ) input, 873–874
  - Write operation
    - CPU, 876
    - defined, 870
    - RAM, 898
- Z**
- Zero count, 22
  - Zero insertion force socket (ZIF), 196–197

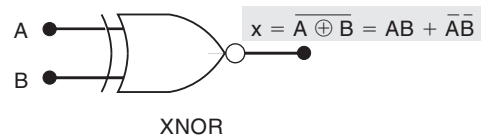
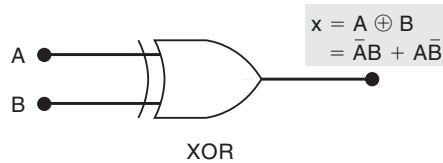
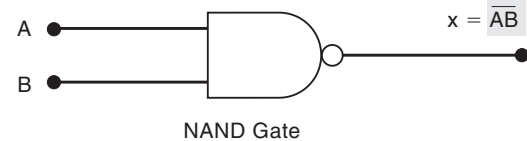
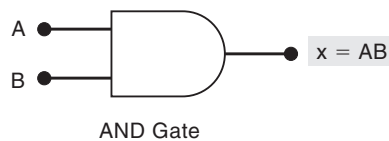
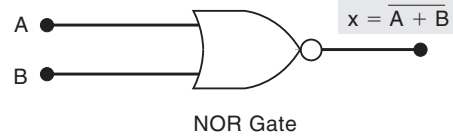
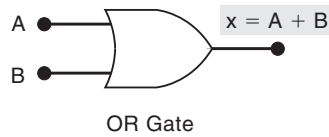
## BOOLEAN THEOREMS

- |   |   |  |
|---|---|--|
| 1. $x \cdot 0 = 0$                            | 2. $x \cdot 1 = x$                          | 3. $x \cdot x = x$                       |
| 4. $x \cdot \bar{x} = 0$                      | 5. $x + 0 = x$                              | 6. $x + 1 = 1$                           |
| 7. $x + x = x$                                | 8. $x + \bar{x} = 1$                        | 9. $x + y = y + x$                       |
| 10. $x \cdot y = y \cdot x$                   | 11. $x + (y + z) = (x + y) + z = x + y + z$ | 12. $x(yz) = (xy)z = xyz$                |
| 13a. $x(y + z) = xy + xz$                     | 13b. $(w + x)(y + z) = wy + xy + wz + xz$   | 14. $x + xy = x$                         |
| 15a. $x + \bar{x}y = x + y$                   | 15b. $\bar{x} + xy = \bar{x} + y$           | 16. $\overline{x + y} = \bar{x} \bar{y}$ |
| 17. $\overline{\bar{x}y} = \bar{x} + \bar{y}$ |   |  |

## LOGIC GATE TRUTH TABLES

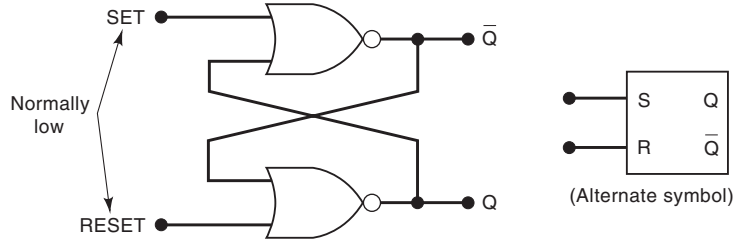
A	B	OR $A + B$	NOR $\overline{A + B}$	AND $A \cdot B$	NAND $\overline{A \cdot B}$	XOR $A \oplus B$	XNOR $\overline{A \oplus B}$
0	0	0	1	0	1	0	1
0	1	1	0	0	1	1	0
1	0	1	0	0	1	1	0
1	1	1	0	1	0	0	1

## LOGIC GATE SYMBOLS



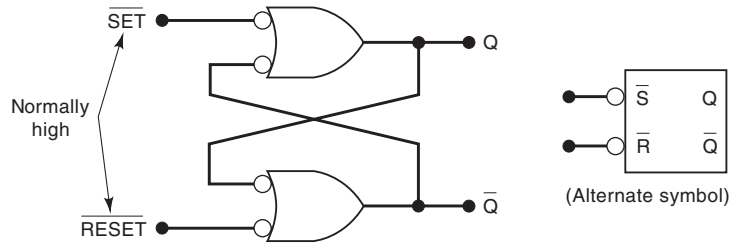
## FLIP-FLOPS

NOR Latch



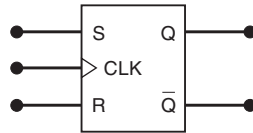
S	R	Q
0	0	No change
1	0	Q = 1
0	1	Q = 0
1	1	Invalid

NAND Latch



$\bar{S}$	$\bar{R}$	Q
0	0	Invalid
1	0	Q = 0
0	1	Q = 1
1	1	No change

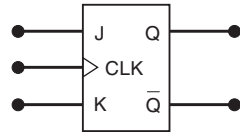
Clocked S-R



S	R	CLK	Q
0	0	↑	$Q_0$ (no change)
1	0	↑	1
0	1	↑	0
1	1	↑	Ambiguous

↓ of CLK has no effect on Q

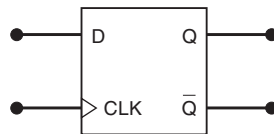
Clocked J-K



J	K	CLK	Q
0	0	↑	$Q_0$ (no change)
1	0	↑	1
0	1	↑	0
1	1	↑	$Q_0$ (toggles)

↓ of CLK has no effect on Q

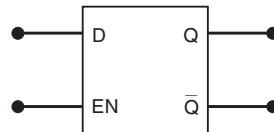
Clocked D



D	CLK	Q
0	↑	0
1	↑	1

↓ of CLK has no effect on Q

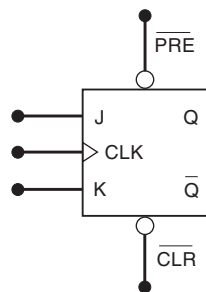
D Latch



EN	D	$Q^*$
0	X	No change
1	0	0
1	1	1

\*Q follows D input while EN is HIGH

Asynchronous Inputs



$\bar{PRE}$	$\bar{CLR}$	$Q^*$
1	1	No effect; FF can respond to J, K and CLK
1	0	Q = 0 independent of J, K, CLK
0	1	Q = 1 independent of J, K, CLK
0	0	Ambiguous (not used)

\*CLK can be in any state