

microprocessors and Interfacing

DOUGLAS V HALL



Tata McGraw-Hill

Special Indian Edition 2006

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

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

**MICROPROCESSORS AND INTERFACING: Programming & Hardware,
Revised Second Edition**

Eighth reprint

DZXACDDXRYQCY

Copyright © 1992 by the Glencoe Division of Macmillan/McGraw-Hill School Publishing Company. Copyright © 1986 by McGraw-Hill, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

IBM PC, IBM PC/XT, IBM PC/AT, PS/2, and MicroChannel Architecture are registered trademarks of IBM Corporation. The following are registered trademarks of Intel Corporation: i486TM, i860TM, ICE, iRMX, Borland, Sidekick, Turbo Assembler, TASM, Turbo Debugger, and Turbo C++ are registered trademarks of Borland International, Inc. Microsoft, MS, MS DOS, Windows 3.0, Codeview, and MASM are registered trademarks of Microsoft Corporation. Other product names are registered trademarks of the companies associated with the product name reference in the text of figure.

ISBN-13: 978-0-07-060167-3

ISBN-10: 0-07-060167-4

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, and printed at
Pashupati Printers (P) Ltd, 1/429/16, Gali No. 1,
Friends Colony, GT Road Shahdara, Delhi 110 095

Cover : SDR Printers

Contents

| | |
|--|-----------------|
| <i>Preface</i> | <i>xi</i> |
| <i>Acknowledgements</i> | <i>xiv</i> |
| 1. Computer Number Systems, Codes, and Digital Devices | 1.1–1.20 |
| Computer Number Systems and Codes 1.1 | |
| Arithmetic Operations on Binary, HEX, and BCD Numbers 1.7 | |
| Basic Digital Devices 1.12 | |
| Review Questions and Problems 1.19 | |
| 2. Computers, Microcomputers, and Microprocessors | 2.1–2.21 |
| —An Introduction | |
| Types of Computers 2.1 | |
| How Computers and Microcomputers are Used—An Example 2.2 | |
| Overview of Microcomputer Structure and Operation 2.6 | |
| Execution of a Three-instruction Program 2.7 | |
| Microprocessor Evolution and Types 2.9 | |
| The 8086 Microprocessor Family—Overview 2.11 | |
| 8086 Internal Architecture 2.12 | |
| Introduction to Programming the 8086 2.16 | |
| Review Questions and Problems 2.20 | |
| 3. 8086 Family Assembly Language Programming—Introduction | 3.1–3.32 |
| Program Development Steps 3.1 | |
| Constructing the Machine Codes for 8086 Instructions 3.12 | |
| Writing Programs for Use with an Assembler 3.19 | |
| Assembly Language Program Development Tools 3.26 | |
| Review Questions and Problems 3.29 | |
| 4. Implementing Standard Program Structures in 8086 Assembly Language | 4.1–4.36 |
| Simple Sequence Programs 4.1 | |
| Jumps, Flags, and Conditional Jumps 4.8 | |
| If-Then, If-Then-Else, and Multiple If-Then-Else Programs 4.15 | |
| While-Do Programs 4.20 | |
| Repeat-Until Programs 4.22 | |
| Instruction Timing and Delay Loops 4.31 | |
| Review Questions and Problems 4.33 | |

| | |
|--|-----------------|
| 5. Strings, Procedures, and Macros | 5.1–5.40 |
| The 8086 String Instructions 5.1 | |
| Writing and Using Procedures 5.6 | |
| Writing and Using Assembler Macros 5.37 | |
| Review Questions and Problems 5.39 | |
| 6. 8086 Instruction Descriptions and Assembler Directives | 6.1–6.36 |
| Instruction Descriptions 6.1 | |
| Assembler Directives 6.30 | |
| Assume 6.31 | |
| DB—Define Byte 6.31 | |
| DD—Define Doubleword 6.31 | |
| DQ—Define Quadword 6.31 | |
| DT—Define Ten Bytes 6.31 | |
| DW—Define Word 6.32 | |
| END—End Program 6.32 | |
| ENDP—End Procedure 6.32 | |
| ENDS—End Segment 6.32 | |
| EQU—Equate 6.32 | |
| EVEN—Align on Even Memory Address 6.32 | |
| EXTRN 6.33 | |
| GLOBAL—Declare Symbols as Public or Extern 6.33 | |
| GROUP—Group-related Segments 6.33 | |
| INCLUDE—Include Source Code From File 6.33 | |
| LABEL 6.34 | |
| LENGTH—Not Implemented in IBM MASM 6.34 | |
| NAME 6.34 | |
| OFFSET 6.34 | |
| ORG—Originate 6.34 | |
| PROC—Procedure 6.35 | |
| PTR—Pointer 6.35 | |
| Segment 6.35 | |
| Short 6.35 | |
| Type 6.36 | |
| 7. 8086 System Connections Timing, and Troubleshooting | 7.1–4.48 |
| A Basic 8086 Microcomputer System 7.1 | |
| Using a Logic Analyzer to Observe Microprocessor Bus Signals 7.7 | |
| An Example Minimum-mode System, The SDK-86 7.12 | |
| Troubleshooting a Simple 8086-based Microcomputer 7.42 | |
| Review Questions and Problems 7.47 | |
| 8. 8086 Interrupts and Interrupt Applications | 8.1–8.44 |
| 8086 Interrupts and Interrupt Responses 8.1 | |
| Hardware Interrupt Applications 8.12 | |
| 8254 Software-Programmable Timer/Counter 8.17 | |

| | | |
|--|--------------|-------------------|
| <u>8259A Priority Interrupt Controller</u> | <u>8.30</u> | |
| <u>Software Interrupt Applications</u> | <u>8.39</u> | |
| <u>Review Questions and Problems</u> | <u>8.41</u> | |
| 9. Digital Interfacing | | 9.1-9.50 |
| Programmable Parallel Ports and Handshake Input/Output | <u>9.1</u> | |
| Interfacing a Microprocessor to Keyboards | <u>9.17</u> | |
| Interfacing to Alphanumeric Displays | <u>9.25</u> | |
| 8279 Circuit Connections and Operation Overview | <u>9.28</u> | |
| Interfacing to 18-segment and Dot-matrix Led Displays | <u>9.35</u> | |
| Interfacing a Microcomputer to Nonmultiplexed Lcd Displays | <u>9.36</u> | |
| Interfacing Microcomputer Ports to High-power Devices | <u>9.37</u> | |
| Optical Motor Shaft Encoders | <u>9.44</u> | |
| Review Questions and Problems | <u>9.47</u> | |
| 10. Analog Interfacing and Industrial Control | | 10.1-10.60 |
| Review of Operational-amplifier Characteristics and Circuits | <u>10.2</u> | |
| Sensors and Transducers | <u>10.7</u> | |
| D/A Converter Operation, Interfacing, and Applications | <u>10.13</u> | |
| A/D Converter Specifications, Types, and Interfacing | <u>10.17</u> | |
| A Microcomputer-based Scale | <u>10.21</u> | |
| A Microcomputer-based Industrial Process-control System | <u>10.31</u> | |
| An 8086-based Process-control System | <u>10.35</u> | |
| Developing the Prototype of a Microcomputer-based Instrument | <u>10.46</u> | |
| Robotics and Embedded Control | <u>10.47</u> | |
| Digital Signal Processing and Digital Filters | <u>10.52</u> | |
| Review Questions and Problems | <u>10.59</u> | |
| 11. Dma, Drams, Cache Memories, Coprocessors, and Eda Tools | | 11.1-11.49 |
| Introduction | <u>11.2</u> | |
| The 8086 Maximum Mode | <u>11.4</u> | |
| Direct Memory Access (DMA) Data Transfer | <u>11.5</u> | |
| Interfacing and Refreshing Dynamic RAMs | <u>11.10</u> | |
| A Coprocessor—The 8087 Math Coprocessor | <u>11.23</u> | |
| Computer-based Design and Development Tools | <u>11.39</u> | |
| Review Questions and Problems | <u>11.48</u> | |
| 12. C, a High-level Language for System Programming | | 12.1-12.52 |
| Introduction—A Simple C Program Example | <u>12.2</u> | |
| Program Development Tools for C | <u>12.3</u> | |
| Programming in C | <u>12.7</u> | |
| Review Questions and Problems | <u>12.50</u> | |
| 13. Microcomputer System Peripherals | | 13.1-13.59 |
| System-level Keyboard Interfacing | <u>13.1</u> | |
| Microcomputer Displays | <u>13.5</u> | |
| Computer Mice and Trackballs | <u>13.31</u> | |

| | |
|---|--------------|
| Computer Vision | 13.33 |
| <u>Magnetic-disk Data-storage Systems</u> | <u>13.34</u> |
| Optical Disk Data Storage | 13.49 |
| Printer Mechanisms and Interfacing | 13.50 |
| <u>Speech Synthesis and Recognition with a Computer</u> | <u>13.53</u> |
| <u>Digital Video Interactive</u> | <u>13.55</u> |
| <i>Review Questions and Problems</i> | 13.57 |

14. Data Communication and Networks**14.1–14.52**

| | |
|---|--------------|
| <u>Introduction to Asynchronous Serial Data Communication</u> | <u>14.1</u> |
| Serial-data Transmission Methods and Standards | 14.8 |
| 20- and 60-ma Current Loops | 14.9 |
| Asynchronous Communication Software on the IBM PC | 14.23 |
| <u>Synchronous Serial-data Communication and Protocols</u> | <u>14.35</u> |
| <u>Local Area Networks</u> | <u>14.40</u> |
| <u>The GPIB, HPIB, IEEE488 Bus</u> | <u>14.48</u> |
| <i>Review Questions and Problems</i> | 14.50 |

15. The 80286, 80386 and 80486 Microprocessors**15.1–15.45**

| | |
|---|-------------|
| <u>Multiuser/Multitasking Operating System Concepts</u> | <u>15.2</u> |
| The Intel 80286 Microprocessor | 15.11 |
| The Intel 80386 32-bit Microprocessor | 15.16 |
| The Intel 80486 Microprocessor | 15.41 |
| <i>Review Questions and Problems</i> | 15.44 |

16. An Introduction to the Pentium Processors**16.1–16.19**

| | |
|--------------------------------------|-------------|
| <u>The Pentium Processor</u> | <u>16.2</u> |
| Epilogue | 16.18 |
| <i>Review Questions and Problems</i> | 16.19 |

| | |
|---------------------|--------------------|
| <u>Appendix A</u> | <u>A.1–A.13</u> |
| <u>Appendix B</u> | <u>B.1–B.16</u> |
| <u>Appendix C</u> | <u>C.1–C.2</u> |
| <u>Bibliography</u> | <u>Bib.1–Bib.3</u> |
| <u>Index</u> | <u>I.1–I.25</u> |

Preface

This book is written for a wide variety of introductory microprocessor courses. The only prerequisite for this book is some knowledge of diodes, transistors, and simple digital devices.

My experience as an engineer and as a teacher indicates that it is much more productive to first learn one microprocessor family very thoroughly and from that strong base learn others as needed. For this book I chose the Intel 8086/80186/80286/80386/80486 and Pentium family of microprocessors. Devices in this family are used in millions and millions of personal computers, including the IBM PC/AT, the IBM PS/2 models, and many "clones." The 8086 was the first member of this family, and although it has been superseded by newer processors, the 8086 is still an excellent entry point for learning about microprocessors. You don't need to know about the advanced features of the newer processors until you learn about multiuser/multitasking systems. Therefore, the 8086 is used for most of the hardware and programming examples until Chapter 15, which discusses the features of the higher processors and how these features are used in multiuser/multitasking systems. Chapter 16 further discusses Pentium Processors and their architecture.

CONTENT AND ORGANIZATION

All chapters begin with fundamental objectives and conclude with a review of important terms and concepts. Each chapter also concludes with a generous supply of questions and problems that reinforce both the theory and applications presented in the chapters.

To help refresh your memory, Chapter 1 contains a brief review of the digital concepts needed for the rest of the book. It also includes an overview of basic computer mathematics and arithmetic operations on binary, HEX, and BCD numbers.

Chapters 2–10

Chapters 2–10 provide you with a comprehensive introduction to microprocessors, including interrupt applications, digital and analog interfacing, and industrial controls. These chapters include an overview of the 8086 microprocessor family and its architecture, programming language, and systems connections and troubleshooting.

Because I came into the world of electronics through the route of vacuum tubes, my first tendency in teaching microprocessors was to approach them from a hardware direction. However, the more I designed with microprocessors and taught microprocessor classes, the more I became aware that the real essence of a microprocessor is what you can program it to do. Therefore, Chapters 2–5 introduce you to writing structured assembly language programs for the 8086 microprocessor. The approach taken in this programming section is to solve the problem, write an algorithm for the solution, and then simply translate the algorithm to assembly language. Experience has shown that this approach is much more likely to produce a working program than just writing down assembly language instructions. The 8086 instruction set is introduced in Chapter 2–5 as needed to solve simple programming problems, but for reference Chapter 6 contains a dictionary of all 8086 instructions with examples for each.

Chapter 7 discusses the signals, timing, and system connections for a simple 8086-based microcomputer. Also discussed in Chapter 7 is a systematic method for troubleshooting a malfunctioning 8086-based microcomputer system and the use of a logic analyzer to observe microcomputer bus signals. Chapter 8 discusses how the 8086 responds to interrupts, how interrupt-service procedures are written, and the operation of a peripheral device called a priority-interrupt controller.

Chapters 9 and 10 show how a microprocessor is interfaced with a wide variety of low-level input and output devices. Chapter 9 shows how a microprocessor is interfaced with digital devices such as keyboards, displays, and relays. Chapter 10 shows how a microprocessor is interfaced with analog input/output devices such as A/Ds,

D/As, and a variety of sensors. It also shows how all the “pieces” are put together to produce a microprocessor-based scale and a simple microprocessor-based process control system. Chapter 10 concludes with a discussion of how microprocessors can be used to implement digital filters.

Chapters 11-16

Chapters 11-15 are devoted to the hardware, software, and peripheral interfacing for a microcomputer such as those in the IBM PC and the IBM PS/2 families. Chapter 11 discusses motherboard circuitry, including DRAM systems, caches, math coprocessors, and peripheral interface buses. It also shows how to use a schematic capture program to draw the schematic, a simulator program to verify the logic and timing of the design, and a layout program to design a printed-circuit board for the system. Knowledge of these electronic design automation tools is essential for anyone developing high-speed microprocessor systems.

At the request of many advisors from industry, Chapter 12 introduces you to the C programming language, which is used to write a large number of system-level programs. This chapter takes advantage of the fact that it is very easy to learn C if you are already familiar with 8086-type assembly language. A section in this chapter also shows you how to write simple programs which contain both C and assembly language modules.

Chapter 13 describes the operation and interfacing of common peripherals such as CRT displays, magnetic disks, and printers. Chapter 14 shows how a microcomputer is interfaced with communication systems such as modems and networks.

Chapter 15 starts with a discussion of the needs that must be met by a multiuser/multitasking operating system and then describes how the protected-mode features of the 80286, 80386, and 80486 processors meet these needs. This section of the book also includes discussions of how to develop programs for the 386 in a variety of environments. The chapter concludes with introductions to parallel processors, neural networks, and fuzzy logic. I think you will find these newly developing areas as fascinating as I have.

Finally, Chapter 16 is about the Pentium processors and their architecture. Data transfer between processor and memory or I/O is also discussed. The chapter ends with a brief description of the Hyper Thread or HT Technology.

SUGGESTIONS FOR ASSIGNMENTS

Flexible Organization

The text is comprehensive, yet flexible in its organization. Chapter 1 could be easily omitted if students have a solid background in basic binary mathematics and digital fundamentals.

Chapters 2-10

I suggest following Chapters 2-10 as an instructional block as each chapter builds on the preceding chapter. These nine chapters represent ideal coverage for a “short course” in microprocessors. The remaining chapters represent an opportunity for the instructor to tailor assignments for the students’ needs or perhaps to give an individual student added study in recent developments in the architecture of microprocessors.

Chapter 11

Individual topics from Chapter 11 could be selected for study as students gain knowledge of the “tools” available for designing computer-based systems. The DRAM section is very important.

Chapter 12

You may wish to assign or leave for outside reading Chapter 12 on programming in C, a new chapter. At the very least you should take a careful look at the simple programming examples and the development of tools for C. If class time does not permit assigning this chapter, you may wish to use selected examples and programs in your lecture presentations. This chapter should be included in any course sequence which does not have a separate class in C programming.

Chapter 13

Portions of the peripherals chapter may be assigned as required, depending upon the course syllabus. The CRT, disk, and printer sections are highly recommended.

Chapter 14

This is an important chapter, given the ever-expanding use of data communications. It should be assigned, if at all possible, unless the curriculum includes a separate course in data communications. Of primary importance are the sections on modems and LANs.

Chapters 15 and 16

The last two chapters discuss the cutting edge of the development of higher microprocessors up to 80486 and the Pentium Processors.

SALIENT FEATURES OF THIS EDITION

In response to feedback from a variety of electronics instructors and from the industry, the revised second edition of *Microprocessors and Interfacing: Programming and Hardware* contains the following salient features.

1. Focused coverage of 8086 microprocessor.
2. Examples of programs with C and assembly language modules.
3. Interfacing illustrated via real life systems.
4. Introduction to Neural Networks and Fuzzy Logic.
5. Inclusion of a new chapter on Pentium Processors.
6. Focused discussion on Software Keyboard Interfacing.
7. Multimedia Technologies: MMX, SSE, SSE2 and SSE3 are also discussed.

SPECIAL FEATURES AND SUPPLEMENTS

This book contains many hardware and software exercises students can do to solidify their knowledge of microprocessors. A dedicated website now supports the book. This website contains the Instructor's Manual and Experiments Manual for instructors and students.

The Experiments Manual contains 40 laboratory exercises that are directly coordinated to the text. Each experiment includes chapter references, required equipment, objectives, and experimental procedures. An IBM PC or IBM PC-compatible computer can be used to edit, assemble, link/locate, run, and debug many of the 8086 assembly language programs.

The Instructor's Manual contains answers to the review questions. It also includes experimental notes and answers to selected questions for the Experiments Manual.

ADDITIONAL GOALS

One of the main goals of this book is to teach you how to decipher manufacturers' data sheets for microprocessor and peripheral devices, so the book contains relevant parts of many data sheets. Because of the large number of devices discussed, however, it was not possible to include complete data sheets. If you are doing an in-depth study, it is suggested that you acquire or gain access to the latest editions of Intel Microprocessors and Peripherals handbooks. These are available free of charge to colleges and universities from the Academic Relations Department of Intel. The bibliography at the end of the book contains a list of other books and periodicals you can refer to for further details on the topics discussed in the book.

If you have suggestions for improving the book or ideas that might clarify a point for someone else, please communicate with me through the publisher.

Douglas V. Hall

Acknowledgements

I wish to express my profound thanks to the people around me who helped make this book a reality. Firstly, I would like to thank Dr. K.M. Hebbar, Director, Computer Engineering, NMAM Institute of Technology, NITTE, Karnataka, for the effort put into adapting this edition of the book. His efforts have indeed updated the book to current standards. Thanks to Pat Hunter, whose cheerful encouragement helped me through seemingly endless details. She proofread and coded the manuscript, worked out the answers to the end-of-chapter to verify that they are solvable, and made suggestions and contributions too numerous to mention. Thanks to Richard Cihkey of New England Technical Institute in New Britain, Connecticut, who meticulously worked his way through the manuscript and made many valuable suggestions. Thanks to Mike Olisewski of Instant Information, Inc., who helped me “C the light” in Chapter 12 and contributed his industry perspective on the topics that should be included in the book. Thanks to Dr. Michael A. Driscoll of Portland State University, who helped me fine-tune Chapter 15. Thanks to Intel Corporation for letting me use many drawings from their data books so that this book could lead readers into the real world of data books. Finally, thanks to my wife, Rosemary, my children Linda, Brad, Mark, Lee, and Kathryn, and to the rest of my family for their patience and support during the long effort of rewriting this book.

Douglas V. Hall

1

Computer Number Systems, Codes, and Digital Devices

Before starting our discussion of microprocessors and microcomputers, we need to make sure that some key concepts of the number systems, codes, and digital devices used in microcomputers are fresh in your mind. If the short summaries of these concepts in this chapter are not enough to refresh your memory, then you may want to consult some of the chapters in *Digital Circuits and Systems*, McGraw-Hill, 1989, before going on in this book.

OBJECTIVES

At the conclusion of this chapter you should be able to:

1. Convert numbers between the following codes: binary, hexadecimal, and BCD.
2. Define the terms bit, nibble, byte, word, most significant bit, and least significant bit.
3. Use a table to find the ASCII or EBCDIC code for a given alphanumeric character.
4. Perform addition and subtraction of binary, hexadecimal, and BCD numbers.
5. Describe the operation of gates, flip-flops, latches, registers, ROMs, PALs, dynamic RAMs, static RAMs, and buses.
6. Describe how an arithmetic logic unit can be instructed to perform arithmetic or logical operations on binary words.

COMPUTER NUMBER SYSTEMS AND CODES

Review of Decimal System

To understand the structure of the binary number system, the first step is to review the familiar decimal or base-10 number system. Here is a decimal number with the value of each place holder or digit expressed as a power of 10.

| | | | | | |
|--------|--------|--------|--------|-----------|-----------|
| 5 | 3 | 4 | 6 | 7 | 2 |
| 10^3 | 10^2 | 10^1 | 10^0 | 10^{-1} | 10^{-2} |

The digits in the decimal number 5346.72 thus tell you that you have 5 thousands, 3 hundreds, 4 tens, 6 ones, 7 tenths, and 2 hundredths. The number of symbols needed in any number system is equal to the base number. In the decimal number system, then, there are 10 symbols, 0 through 9. When the count in any digit position passes that of the highest-value symbol, the digit rolls back to 0 and the next higher digit is incremented by 1. A car odometer is a good example of this.

A number system can be built using powers of any number as place holders or digits, but some bases are more useful than others. It is difficult to build electronic circuits which can store and manipulate 10 different voltage levels but relatively easy to build circuits which can handle two levels. Therefore, a binary, or *base-2*, number system is used to represent numbers in digital systems.

The Binary Number System

Figure 1.1a, shows the value of each digit in a binary number. Each binary digit represents a power of 2. A binary digit is often called a *bit*. Note that digits to the right of the *binary point* represent fractions used for numbers less than 1. The binary system uses only two symbols, zero (0) and one (1), so in binary you count as follows: 0, 1, 10, 11, 100, 101, 110, 111, 1000, etc. For reference, Fig. 1.1b shows the powers of 2 from 2^1 to 2^{32} .

Binary numbers are often called *binary words* or just *words*. Binary words with certain numbers of bits have

| | | | | | | | |
|-------|-------|-------|-------|-------|---------------|---------------|-------|
| | 1 | 0 | 1 | 1 | 0. | 1 | 1 |
| 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| | | | | | $\frac{1}{2}$ | $\frac{1}{2}$ | |

(a)

| | | | |
|-------------|-------------------|-----------------------|--------------------------|
| $2^1 = 2$ | $2^9 = 512$ | $2^{17} = 131,072$ | $2^{25} = 33,554,432$ |
| $2^2 = 4$ | $2^{10} = 1,024$ | $2^{18} = 262,144$ | $2^{26} = 67,108,864$ |
| $2^3 = 8$ | $2^{11} = 2,048$ | $2^{19} = 524,288$ | $2^{27} = 134,217,728$ |
| $2^4 = 16$ | $2^{12} = 4,096$ | $2^{20} = 1,048,576$ | $2^{28} = 268,435,456$ |
| $2^5 = 32$ | $2^{13} = 8,192$ | $2^{21} = 2,097,152$ | $2^{29} = 536,870,912$ |
| $2^6 = 64$ | $2^{14} = 16,384$ | $2^{22} = 4,194,304$ | $2^{30} = 1,073,741,824$ |
| $2^7 = 128$ | $2^{15} = 32,768$ | $2^{23} = 8,388,608$ | $2^{31} = 2,147,483,648$ |
| | $2^{16} = 65,536$ | $2^{24} = 16,777,216$ | $2^{32} = 4,294,967,296$ |

(b)

Fig. 1.1 (a) Digit values in binary. (b) Powers of 2.

or $4 + 0 + 1 = \text{decimal } 5$. For the binary number 10110.11, you have:

$$(1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \\ + (1 \times 2^{-1}) + (1 \times 2^{-2}) = \\ 16 + 0 + 4 + 2 + 0 + 0.5 + 0.25 = \text{decimal } 22.75$$

To convert a decimal number to binary, there are two common methods. The first (Fig. 1.2a) is simply a reverse of the binary-to-decimal method. For example, to convert the decimal number 21 (sometimes written as 21_{10}) to binary, first subtract the largest power of 2 that will fit in the number. For 21_{10} the largest power of 2 that will fit is 16 or 2^4 . Subtracting 16 from 21 gives a remainder of 5. Put a 1 in the 2^4 digit position and see if the next lower power of 2 will fit in the remainder. Since 2^3 is 8 and 8 will not fit in the remainder of 5, put a 0 in the 2^3 digit position. Then try the next lower power of 2. In this case the next is 2^2 or 4, which will fit in the remainder of 5. A 1 is therefore put in the 2^2 digit position. When 2^2 or 4 is subtracted from the old remainder of 5, a new remainder of 1 is left. Since 2^1 or 2 will not fit into this remainder, a 0 is put in that position. A 1 is put in the 2^0 position because 2^0 is equal to 1 and this fits exactly into the remainder of 1. The result shows that 21_{10} is equal to 10101 in binary. This conversion process is somewhat messy to describe

also acquired special names. A 4-bit binary word is called a *nibble*, and an 8-bit binary word is called a *byte*. A 16-bit binary word is often referred to just as a *word*, and a 32-bit binary word is referred to as a *doubleword*. The rightmost or *least significant bit* of a binary word is usually referred to as the *LSB*. The leftmost or *most significant bit* of a binary word is usually called the *MSB*.

To convert a binary number to its equivalent decimal number, multiply each digit times the decimal value of the digit and just add these up. The binary number 101, for example, represents: $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$,

but easy to do. Try converting 46_{10} to binary. You should get 101110.

Another method of converting a decimal number to binary is shown in Fig. 1.2b. Divide the decimal number by 2 and write the quotient and remainder as shown. Divide this quotient and following quotients by 2 until the quotient reaches 0. The column of remainders will be the binary equivalent of the given decimal number. Note that the *MSD* is on the bottom of the column and the *LSD* is on the top of the column if you perform the divisions in order from the top to the bottom of the page. You can demonstrate that the binary number is correct by reconverting from binary to decimal, as shown in the right-hand side of Fig. 1.2b.

You can convert decimal numbers less than 1 to binary by successive multiplication by 2, recording carries until the quantity to the right of the decimal point becomes zero, as shown in Fig. 1.2c. The carries represent the binary equivalent of the decimal number, with the *most significant bit* at the top of the column. Decimal 0.625 equals 0.101 in binary. For decimal values that do not convert exactly the way this one did (the quantity to the right of the decimal never becomes zero), you can continue the conversion process until you get the number of binary digits desired.

At this point it is interesting to compare the number of digits required to express numbers in decimal with the number required to express them in binary. In decimal, one digit can represent 10^1 numbers, 0 through 9; two digits can represent 10^2 or 100 numbers, 0 through 99; and three digits can represent 10^3 or 1000 numbers, 0 through 999. In binary, a similar pattern exists. One binary digit can represent 2 numbers, 0 and 1; two binary digits can represent 2^2 or 4 numbers, 0 through 11; and three binary digits can represent 2^3 or 8 numbers, 0 through 111. The pattern, then, is that N decimal digits can represent 10^N numbers and N binary digits can represent 2^N numbers. Eight binary digits can represent 2^8 or 256 numbers, 0 through 255 in decimal.

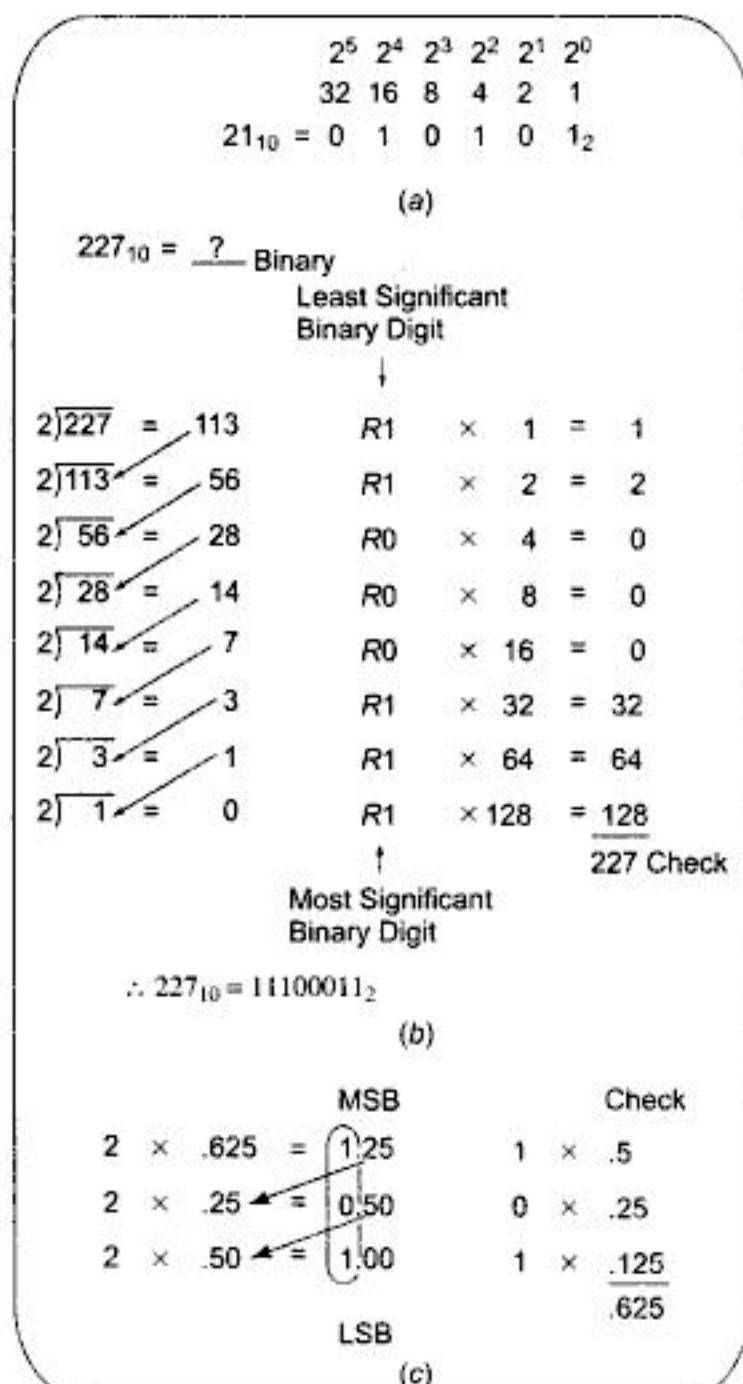


Fig. 1.2 Converting decimal to binary. (a) Digit value method, (b) Divide by 2 method, (c) Decimal fraction conversion.

Hexadecimal

Binary is not a very compact code. This means that it requires many more digits to express a number than does, for example, decimal. Twelve binary digits can only describe a number up to 4095_{10} . Computers require binary data, but people working with computers have trouble remembering long binary words. One solution to the problem is to use the *hexadecimal* or base-16 number system.

Figure 1.3a shows the digit values for hexadecimal, which is often just called *hex*. Since hex is base 16, you have to have 16 possible symbols, one for each digit. The table of Fig. 1.3b shows the symbols for hex code.

$$16^3 \quad 16^2 \quad 16^1 \quad 16^0 \quad 16^{-1} \quad 16^{-2} \quad 16^{-3}$$

(a)

| Dec | Hex | Dec | Hex |
|-----|-----|-----|-----|
| 0 | 0 | 8 | 8 |
| 1 | 1 | 9 | 9 |
| 2 | 2 | 10 | A |
| 3 | 3 | 11 | B |
| 4 | 4 | 12 | C |
| 5 | 5 | 13 | D |
| 6 | 6 | 14 | E |
| 7 | 7 | 15 | F |

(b)

$$\begin{array}{r} 227_D = \underline{\quad ? \quad} \text{ Hex } \quad \text{LSD} \\ 16) \overline{227} \quad = \quad 14 \qquad R_3 \times 1 \quad = \quad 3 \\ 16) \overline{14} \quad = \quad 0 \qquad R_E \times 16 \quad = \quad \underline{224} \\ \qquad \qquad \qquad \text{MSD} \qquad \qquad \qquad \underline{227} \end{array}$$

$$227_{10} = E3_{16}$$

(c)

Fig. 1.3 Hexadecimal numbers, (a) Value of place holders, (b) Symbols, (c) Decimal-to-hexadecimal conversion

After the decimal symbols 0 through 9 are used up, you use the letters A through F for values 10 through 15.

As mentioned above, each hex digit is equal to four binary digits. To convert the binary number 11010110 to hex, mark off the binary bits in groups of 4, moving to the left from the binary point. Then write the hex symbol for the value of each group of 4.

| | | |
|--------|------|------|
| Binary | 1101 | 0110 |
| Hex | D | 6 |

The 0110 group is equal to 6 and the 1101 group is equal to 13. Since 13 is D in hex, 11010110 binary is equal to D6 in hex. "H" is usually used after a number to indicate that it is a hexadecimal number. For example, D6 hex is usually written D6H. As you can see, 8 bits can be represented with only 2 hex digits.

If you want to convert a number from decimal to hexadecimal, Fig. 1.3c shows a familiar trick for doing this. The result shows that 227_{10} is equal to E3H. As you can see, hex is an even more compact code than decimal. Two hexadecimal digits can represent a decimal number up to 255. Four hex digits can represent a decimal number up to 65,535.

To illustrate how hexadecimal numbers are used in digital logic, a service manual tells you that the 8-bitwide data bus of an 8088A microprocessor should contain 3FH during a certain operation. Converting 3FH to binary gives the pattern of 1's and 0's (0011 1111) you would expect to find with your oscilloscope or logic analyzer on the parallel lines. The 3FH is simply a shorthand which is easier to remember and less prone to errors than the binary equivalent.

BCD Codes

STANDARD BCD

In applications such as frequency counters, digital voltmeters, or calculators, where the output is a decimal

display, a *binary-coded decimal* or *BCD* code is often used. BCD uses a 4-bit binary code to individually represent each decimal digit in a number. As you can see in Table 1.1, the simplest BCD code uses the first 10 numbers of standard binary code for the BCD numbers 0 through 9. The hex codes A through F are invalid BCD codes. To convert a decimal number to its BCD equivalent, just represent each decimal digit by its 4-bit binary equivalent, as shown here.

| | | | |
|---------|------|------|------|
| Decimal | 5 | 2 | 9 |
| BCD | 0101 | 0010 | 1001 |

To convert a BCD number to its decimal equivalent, reverse the process.

GRAY CODE

Gray code is another important binary code; it is often used for encoding shaft position data from machines such as computer-controlled lathes. This code has the same possible combinations as standard binary, but as you can see in the 4-bit example in Table 1.1, they are arranged in a different order. Notice that only one binary digit changes at a time as you count up in this code.

If you need to construct a Gray-code table larger than that in Table 1.1, a handy way to do so is to observe the pattern of 1's and 0's and just extend it. The least significant digit column starts with one 0 and then has alternating groups of two 1's and two 0's as you go down the column. The second most significant digit column starts with two 0's and then has alternating groups of four 1's and four

Table 1.1 Common Number Codes

| Decimal | Binary | Octal | Hex | Binary-Coded Decimal | | | Reflected Gray Code | 7-Segment Display (1 = on) | |
|---------|--------|-------|-----|----------------------|------|-----------|---------------------------|----------------------------|---------|
| | | | | 8421 | BCD | EXCESS-3 | | a b c d e f g | Display |
| 0 | 0000 | 0 | 0 | | 0000 | 0011 0011 | 0000 | 1111110 | 0 |
| 1 | 0001 | 1 | 1 | | 0001 | 0011 0100 | 0001 | 0110000 | 1 |
| 2 | 0010 | 2 | 2 | | 0010 | 0011 0101 | 0011 | 1101101 | 2 |
| 3 | 0011 | 3 | 3 | | 0011 | 0011 0110 | 0010 | 1111001 | 3 |
| 4 | 0100 | 4 | 4 | | 0100 | 0011 0111 | 0110 | 0110011 | 4 |
| 5 | 0101 | 5 | 5 | | 0101 | 0011 1000 | 0111 | 1011011 | 5 |
| 6 | 0110 | 6 | 6 | | 0110 | 0011 1001 | 0101 | 1011111 | 6 |
| 7 | 0111 | 7 | 7 | | 0111 | 0011 1010 | 0100 | 1110000 | 7 |
| 8 | 1000 | 10 | 8 | | 1000 | 0011 1011 | 1100 | 1111111 | 8 |
| 9 | 1001 | 11 | 9 | | 1001 | 0011 1100 | 1101 | 1110011 | 9 |
| 10 | 1010 | 12 | A | 0001 | 0000 | 0100 0011 | 1111 | 1111101 | A |
| 11 | 1011 | 13 | B | 0001 | 0001 | 0100 0100 | 1110 | 0011111 | B |
| 12 | 1100 | 14 | C | 0001 | 0010 | 0100 0101 | 1010 | 0001101 | C |
| 13 | 1101 | 15 | D | 0001 | 0011 | 0100 0110 | 1011 | 0111101 | D |
| 14 | 1110 | 16 | E | 0001 | 0100 | 0100 0111 | 1001 | 1101111 | E |
| 15 | 1111 | 17 | F | 0001 | 0101 | 0100 1000 | 1000 | 1000111 | F |

0's. The third column starts with four 0's, then has alternating groups of eight 1's and eight 0's. By now you should see the pattern. Try to Fig. out the Gray code for the decimal number 16. You should get 11000.

7-Segment Display Code

Figure 1.4a shows the segment identifiers for a 7-segment display such as those commonly used in digital instruments. Table 1.1 shows the logic levels required to display 0 to 9 and A to F on a common-cathode LED display such as that shown in Fig. 1.4b. For a common-anode LED display such as that in Fig. 1.4c, simply invert the segment codes shown in Table 1.1.

Alphanumeric Codes

When communicating with or between computers, you need a binary-based code which can represent letters of the alphabet as well as numbers. Common codes used for this have 7 or 8 bits per word and are referred to as *alphanumeric codes*. To detect possible errors in these codes, an additional bit, called a *parity bit*, is often added as the most significant bit.

Parity is a term used to identify whether a data word has an odd or even number of 1's. If a data word contains an odd number of 1's, the word is said to have *odd parity*. The binary word 0110111 with five 1's has odd parity. The binary word 0110000 has an even number of 1's (two), so it has *even parity*.

In practice the parity bit is used as follows. The system that is sending a data word checks the parity of the word. If the parity of the data word is odd, the system will set the parity bit to a 1. This makes the parity of the data word plus parity bit even. If the parity of the data word is even, the sending system will reset the parity bit to a 0. This

again makes the parity of the data word plus parity even. The receiving system checks the parity of the data word plus parity bit that it receives. If the receiving system detects odd parity in the received data word plus parity, it assumes an error has occurred and tells the sending system to send the data again. The system is then said to be using even parity. The system could have been set up to use (maintain) odd parity in a similar manner.

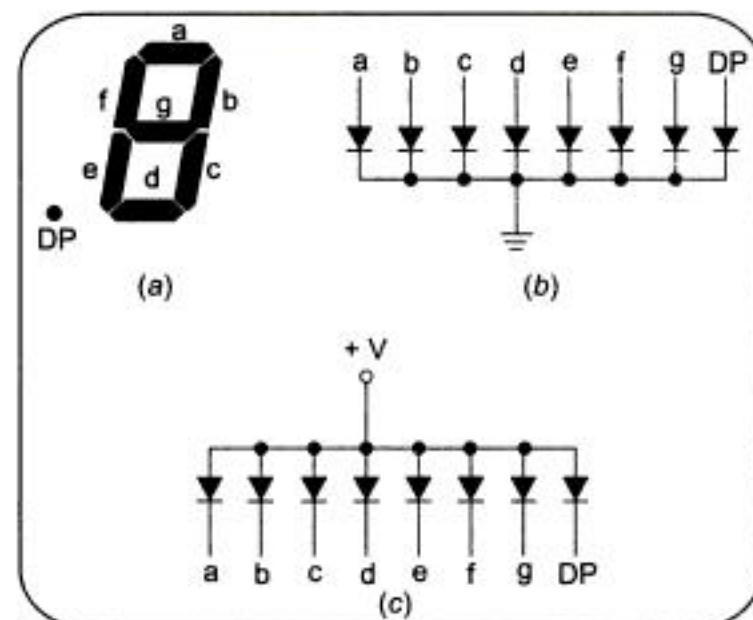


Fig. 1.4 7-segment LED display. (a) Segment labels. (b) Schematic of common-cathode type. (c) Schematic of common-anode type.

ASCII

Table 1.2 shows several alphanumeric codes. The first of these is *ASCII*, or American Standard Code for Information Interchange. This is shown in the table as a 7-bit code. With 7 bits you can code up to 128 characters, which is enough for the full upper- and lowercase alphabet,

Table 1.2 Common Alphanumeric Codes

| ASCII Symbol | HEX Code for 7-Bit ASCII | EBCDIC Symbol | HEX Code for EBCDIC | ASCII Symbol | HEX Code for 7-Bit ASCII | EBCDIC Symbol | HEX Code for EBCDIC | ASCII Symbol | HEX Code for 7-Bit ASCII | EBCDIC Symbol | HEX Code for EBCDIC |
|--------------|--------------------------|---------------|---------------------|--------------|--------------------------|---------------|---------------------|--------------|--------------------------|---------------|---------------------|
| NUL | 00 | NUL | 00 | * | 2A | * | 5C | T | 54 | T | E3 |
| SOH | 01 | SOH | 01 | + | 2B | + | 4E | U | 55 | U | E4 |
| STX | 02 | STX | 02 | , | 2C | , | 6B | V | 56 | V | E5 |
| ETX | 03 | ETX | 03 | - | 2D | - | 60 | W | 57 | W | E6 |
| EOT | 04 | EOT | 37 | . | 2E | . | 4B | X | 58 | X | E7 |
| ENQ | 05 | ENQ | 2D | / | 2F | / | 61 | Y | 59 | Y | E8 |
| ACK | 06 | ACK | 2E | 0 | 30 | 0 | F0 | Z | 5A | Z | E9 |
| BEL | 07 | BEL | 2F | 1 | 31 | 1 | F1 | ! | 5B | ! | AD |

(Contd.)

(Contd.)

| ASCII Symbol | HEX Code for 7-Bit ASCII | EBCDIC Symbol | HEX Code for EBCDIC | ASCII Symbol | HEX Code for 7-Bit ASCII | EBCDIC Symbol | HEX Code for EBCDIC | ASCII Symbol | HEX Code for 7-Bit ASCII | EBCDIC Symbol | HEX Code for EBCDIC |
|-----------------|-----------------------------------|------------------|------------------------------|-----------------|-----------------------------------|------------------|------------------------------|-----------------|-----------------------------------|------------------|------------------------------|
| BS | 08 | BS | 16 | 2 | 32 | 2 | F2 | λ | 5C | NL | 15 |
| HT | 09 | HT | 05 | 3 | 33 | 3 | F3 |] | 5D | [| DD |
| LF | 0A | LF | 25 | 4 | 34 | 4 | F4 | ^ | 5E | ˥ | 5F |
| VT | 0B | VT | OB | 5 | 35 | 5 | F5 | — | 5F | — | 6D |
| FF | 0C | FF | OC | 6 | 36 | 6 | F6 | ‘ | 60 | RES | 14 |
| CR | 0D | CR | OD | 7 | 37 | 7 | F7 | a | 61 | a | 81 |
| SO | 0E | SO | OE | 8 | 38 | 8 | F8 | b | 62 | b | 82 |
| SI | 0F | SI | OF | 9 | 39 | 9 | F9 | c | 63 | c | 83 |
| DLE | 10 | DLE | 10 | : | 3A | : | 7A | d | 64 | d | 84 |
| DC1 | 11 | DC1 | 11 | ; | 3B | : | 5E | e | 65 | e | 85 |
| DC2 | 12 | DC2 | 12 | — | 3C | — | 4C | f | 66 | f | 86 |
| DC3 | 13 | DC3 | 13 | = | 3D | = | 7E | g | 67 | g | 87 |
| DC4 | 14 | DC4 | 35 | \ | 3E | \ | 6E | h | 68 | h | 88 |
| NAK | 15 | NAK | 3D | ? | 3F | ? | 6F | i | 69 | i | 89 |
| SYN | 16 | SYN | 32 | @ | 40 | @ | 7C | j | 6A | j | 91 |
| ETB | 17 | EOB | 26 | A | 41 | A | C1 | k | 6B | k | 92 |
| CAN | 18 | CAN | 18 | B | 42 | B | C2 | l | 6C | l | 93 |
| EM | 19 | EM | 19 | C | 43 | C | C3 | m | 6D | m | 94 |
| SUB | 1A | SUB | 3F | D | 44 | D | C4 | n | 6E | n | 95 |
| ESC | IB | BYP | 24 | E | 45 | E | C5 | o | 6F | o | 96 |
| FS | 1C | FLS | 1C | F | 46 | F | C6 | P | 70 | P | 97 |
| GS | ID | GS | ID | G | 47 | G | C7 | q | 71 | q | 98 |
| RS | IE | RDS | IE | H | 48 | H | C8 | r | 72 | r | 99 |
| US | IF | US | IF | I | 49 | I | C9 | s | 73 | s | A2 |
| SP | 20 | SP | 40 | J | 4A | J | D1 | t | 74 | t | A3 |
| ! | 21 | ! | 5A | K | 4B | K | D2 | u | 75 | u | A4 |
| " | 22 | " | 7F | L | 4C | L | D3 | v | 76 | v | A5 |
| # | 23 | # | 7B | M | 4D | M | D4 | w | 77 | w | A6 |
| \$ | 24 | \$ | 5B | N | 4E | N | D5 | X | 78 | X | A7 |
| % | 25 | % | 6C | O | 4F | O | D6 | y | 79 | y | A8 |
| & | 26 | & | 50 | P | 50 | P | D7 | z | 7A | z | A9 |
| ' | 27 | ' | 7D | Q | 51 | Q | D8 | { | 7B | { | 8B |
| (| 28 | (| 4D | R | 52 | R | D9 | | 7C | | 4F |
|) | 29 |) | 5D | S | 53 | S | E2 | } | 7D | } | 9B |
| | | | | | | | | - | 7E | € | 4A |
| | | | | | | | DEL | 7F | DEL | 07 | |

numbers, punctuation marks, and control characters. The code is arranged so that if only uppercase letters, numbers, and a few control characters are needed, the lower 6 bits are all that are required. If a parity check is wanted, a parity bit is added to the basic 7-bit code in the MSB position. The binary word 1100 0100, for example, is the ASCII code for uppercase D with odd parity. Table 1.3 gives the meanings of the control character symbols used in the ASCII code table.

EBCDIC

Another alphanumeric code commonly encountered in IBM equipment is the Extended Binary-Coded Decimal Interchange Code or *EBCDIC*. This is an 8-bit code without parity. A ninth bit can be added for parity. To save space in Table 1.2, the eight binary digits of EBCDIC are represented by their 2-digit hex equivalent.

Table 1.3 Definitions of Control Characters

| | | | |
|------|---------------------|-----|------------------------|
| NULL | Null | DC1 | Direct control 1 |
| SOH | Start of heading | DC2 | Direct control 2 |
| STX | Start text | DC3 | Direct control 3 |
| ETX | End text | DC4 | Direct control 4 |
| EOT | End of transmission | NAK | Negative acknowledge |
| ENQ | Enquiry | SYN | Synchronous idle |
| ACK | Acknowledge | ETB | End transmission block |
| BELL | BS | CAN | Cancel |
| BS | Backspace | EM | End of medium |
| HT | Horizontal tab | SUB | Substitute |
| LF | Line feed | ESC | Escape |
| VT | Vertical tab | FS | Form separator |
| FF | Form feed | GS | Group separator |
| CR | Carriage return | RS | Record separator |
| SO | Shift out | US | Unit separator |
| SI | Shift in | | |
| DLE | Data link escape | | |

ARITHMETIC OPERATIONS ON BINARY, HEX, AND BCD NUMBERS

Binary Arithmetic

ADDITION

Figure 1.5a shows the truth table for addition of two binary digits and a carry in (C_{IN}) from addition of previous digits. Fig. 1.5b shows the result of adding two 8-bit binary

| INPUTS | | | OUTPUTS | |
|--------|---|----------|---------|-----------|
| 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 |

(a)

$$S = A \oplus B \oplus C_{IN}$$

$$C_{OUT} = A \cdot B + C_{IN} (A \oplus B)$$

| | | |
|----------|---|----------|
| 10011010 | + | 11011100 |
| | | 01110110 |
| | | ↑ Carry |

(b)

Fig. 1.5 Binary addition, (a) Truth table for 2 bits plus carry. (b) Addition of two 8-bit words.

numbers together using these rules. Assuming that $C_{IN} = 1$, $1 + 0 + C_{IN} =$ a sum of 0 and a carry into the next digit, and $1 + 1 + C_{IN} =$ a sum of 1 and a carry into the next digit because the result in any digit position can only be a 1 or a 0.

2'S-COMPLEMENT SIGNS-AND-MAGNITUDE BINARY

When you handwrite a number that represents some physical quantity such as temperature, you can simply put a + sign in front of the number to indicate that the number is positive, or you can write a - sign to indicate that the number is negative. However, if you want to store values such as temperatures, which can be positive or negative, in a computer memory, there is a problem. Since the computer memory can store only 1's and 0's, some way must be established to represent the sign of the number with a 1 or a 0.

A common way to represent signed numbers is to reserve the most significant bit of the data word as a *sign bit* and to use the rest of the bits of the data word to represent the size (magnitude) of the quantity. A computer that works with 8-bit words will use the MSB (bit 7) as the sign bit and the lower 7 bits to represent the magnitude of the numbers. The usual convention is to represent a positive number with a 0 sign bit and a negative number with a 1 sign bit.

To make computations with signed numbers easier, the magnitude of negative numbers is represented in a special form called 2's complement. The 2's complement of a binary number is formed by inverting each bit of the data word and adding 1 to the result. Some examples should help clarify all of this.

The number $+7_{10}$ is represented in 8-bit sign-and-magnitude form as 00000111. The sign bit is 0, which indicates a positive number. The magnitude of positive numbers is represented in straight binary, so 00000 111 in the least significant bits represents 7_{10} .

To represent -7_{10} in 8-bit 2's-complement sign-and-magnitude form, start with the 8-bit code for $+7$, 00000111. Invert each bit, including the MSB, to get 11111000. Then add 1 to get 11111001. This result is the correct representation of -7_{10} . Fig. 1.6 shows some more examples of positive and negative numbers expressed in 8-bit sign-and-magnitude form. For practice, try generating each of these yourself to see if you get the same result.

To reverse this procedure and find the magnitude of a number expressed in sign-and-magnitude form, proceed as follows. If the number is positive, as indicated by the

| | Sign bit | |
|-------|----------|---------|
| + 7 | 0 | 0000111 |
| + 46 | 0 | 0101110 |
| + 105 | 0 | 1101001 |
| - 12 | 1 | 1110100 |
| - 54 | 1 | 1001010 |
| - 117 | 1 | 0001011 |
| - 46 | 1 | 1010010 |

} Sign and two's complement of magnitude

Fig. 1.6 Positive and negative numbers represented with a sign bit and 2's complement.

sign bit being a 0, then the least significant 7 bits represent the magnitude directly in binary. If the number is negative, as indicated by the sign bit being a 1, then the magnitude is expressed in 2's complement. To get the magnitude of this negative number expressed in standard binary, invert each bit of the data word, including the sign bit, and add 1 to the result. For example, given the word 11101011, invert each bit to get 00010100. Then add 1 to get 00010101. This equals 21₁₀, so you know that the original numbers represent -21₁₀. Again, try reconverting a few of the numbers in Fig. 1.6 for practice.

Figure 1.7 shows some examples of addition of signed binary numbers of this type. Sign bits are added together just as the other bits are. Fig. 1.7a shows the results of adding two positive numbers. The sign bit of the result is zero, so the result is positive. The second example, in Fig. 1.7b, adds a -9 to a +13 or, in effect, subtracts 9 from 13. As indicated by the zero sign bit, the result of 4 is positive and in true binary form.

Figure 1.7c shows the result of adding a -13 to a smaller positive number, + 9. The sign bit of the result is a 1. This indicates that the result is negative and the magnitude is in 2's-complement form. To reconvert a 2's complement result to a signed number in true binary form:

1. Invert each bit to produce the 1's complement.
2. Add 1.
3. Put a minus sign in front to indicate that the result is negative.

The final example, in Fig. 1.7d, shows the result of adding two negative numbers. The sign bit of the result is a 1, so the result is negative and in 2's-complement form. Again, inverting each bit, adding 1, and prefixing a minus sign will put the result in a more recognizable form.

Now let's consider the range of numbers that can be represented with 8 bits in sign-and-magnitude form. Eight

bits can represent a maximum of 2⁸ or 256 numbers. Since we are representing both positive and negative numbers, half of this range will be positive and half negative. Therefore, the range is -128 to +127. Here are the sign-and-magnitude binary representations for these values:

| | |
|-----------------|------|
| 0 1 1 1 1 1 1 1 | +127 |
| ⋮ | |
| 0 0 0 0 0 0 0 1 | +1 |
| 0 0 0 0 0 0 0 0 | zero |
| 1 1 1 1 1 1 1 1 | -1 |
| ⋮ | |
| 1 0 0 0 0 0 0 1 | -127 |
| 1 0 0 0 0 0 0 0 | -128 |

| | |
|---|---|
| $\begin{array}{r} +13 \\ + 9 \\ \hline +22 \end{array}$ | 00001101 00001001 00010110 ← Sign bit is 0 so result is positive (a) |
| $\begin{array}{r} +13 \\ - 9 \\ + 4 \\ \hline \end{array}$ | 00001101 11110111 2's complement for -9 with sign bit 00000100 ← Sign bit is 0 so result is positive Ignore carry (b) |
| $\begin{array}{r} + 9 \\ -13 \\ - 4 \\ \hline \end{array}$ | 00001001 11110011 2's complement for -13 with sign bit 11111100 Sign bit is 1 00000011 so invert each bit ← equals + 1 Add 1 -00000100 Prefix with minus sign (c) |
| $\begin{array}{r} - 9 \\ -13 \\ - 22 \\ \hline \end{array}$ | 11110111 2's complement for -9 with sign bit 11110011 Sign-and-magnitude form 11101010 Sign bit is 1 so invert each bit ← equals + 1 Add 1 -00010110 Prefix with minus sign (d) |

Fig. 1.7 Addition of signed binary numbers, (a) +9 and +13. (a) -9 and +13. (c) +9 and -13. (d) -9 and -13.

If you like number patterns, you might notice that this scheme shifts the normal codes for 128 to 255 downward to represent -128 to -1.

If a computer is storing signed numbers as 16-bit words, then a much larger range of numbers can be represented. Since 16 bits gives 2^{16} or 65,536 possible values, the range for 16-bit sign-and-magnitude numbers is -32,768 to +32,767. Operations with 16-bit sign-and-magnitude numbers are done the same way as operations with 8-bit sign-and-magnitude numbers.

SUBTRACTION

There are two common methods for doing binary subtraction. These are the pencil method and the 2's-complement add method. Fig. 1.8a shows the truth table for binary subtraction of two binary digits A and B. Also included in the truth table is the effect of a borrow-in, B_{IN} , from subtracting previous digits. Fig. 1.8b shows an example of the "pencil" method of subtracting two 8-bit numbers. Using the truth table, this method is done the same way that you do decimal subtraction.

A second method of performing binary subtraction is by adding the 2's-complement representation of the bottom number (subtrahend) to the top number (minuend). Fig. 1.8c shows how this is done. First represent the top number in sign-and-magnitude form. Then form the 2's-complement sign-and-magnitude representation for the negative of the bottom number. Finally, add the two parts formed. For the example in Fig. 1.8c, the sign of the result

is a 0, which indicates that the result is positive and in true form. The final carry produced by the addition can be ignored. Fig. 1.8d shows another example of this method of subtraction. In this case the bottom number is larger than the top number. Again, represent the top number in sign-and-magnitude form, produce the 2's-complement sign-and-magnitude form for the negative of the bottom number, and add the two together. The sign bit of the result is a 1 for this example. This indicates that the result is negative and its magnitude is represented in 2's-complement form. To get the result into a form that is more recognizable to you, invert each bit of the result, add 1 to it, and put a minus sign in front of it as shown in Fig. 1.8d.

Problems that may occur when doing signed addition or subtraction are *overflow* and *underflow*. If the magnitude of the number produced by adding two signed numbers is larger than the number of bits available to represent the magnitude, the result will "overflow" into the sign bit position and give an incorrect result. For example, if the signed positive number 01001001 is added to the signed positive number 01101101, the result is 10110110. The 1 in the MSB of this result indicates that it is negative, which is obviously incorrect for the sum of two positive numbers. In a similar manner, doing an 8-bit signed subtraction that produces a magnitude greater than

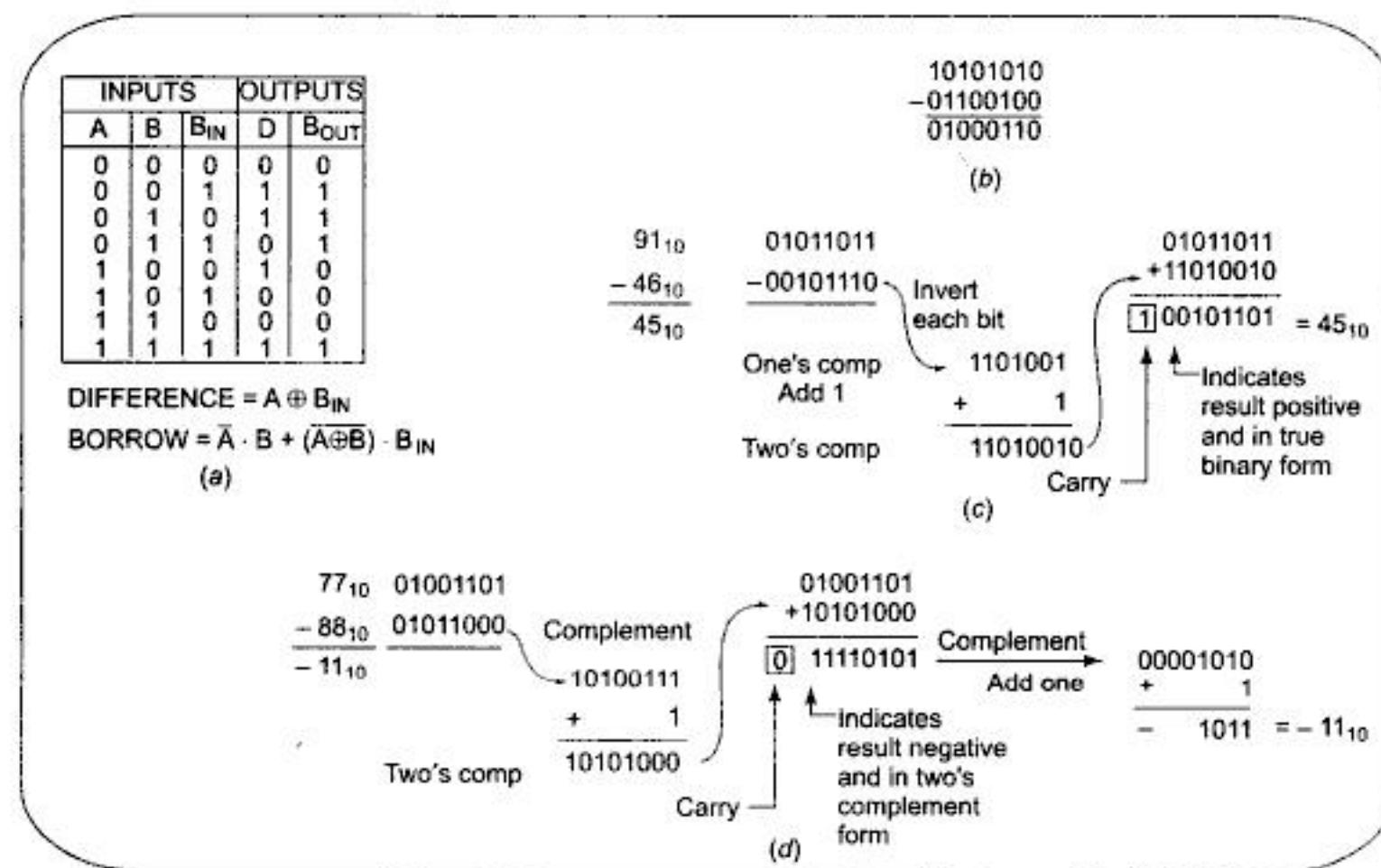


Fig. 1.8 Binary subtraction. (a) Truth table for 2 bits and borrow, (b) Pencil method, (c) 2's-complement positive result, (d) 2's-complement negative result.

-128 will cause an "underflow" into the sign bit and produce an incorrect result.

For simplicity the examples shown use 8 bits, but the method works for any number of bits. This method may seem awkward, but it is easy to do in a computer or microprocessor because it requires only the simple operations of inverting and adding.

MULTIPLICATION

There are several methods of doing binary multiplication. Fig. 1.9 shows what is called the *pencil method* because it is the same way you learned to multiply decimal numbers. The top number, or multiplicand, is multiplied by the least significant digit of the bottom number, or multiplier. The partial product is written down. The top number is then multiplied by the next digit of the multiplier. The resultant partial product is written down under the last, but shifted one place to the left. Adding all the partial products gives the total product. This method works well when doing multiplication by hand, but it is not practical for a computer because the type of shifts required makes it awkward to implement.

| | | |
|---|---|---------------------------------------|
| $\begin{array}{r} 11 \\ \times 9 \\ \hline \end{array}$ | $\begin{array}{r} 1011 \\ \times 1001 \\ \hline \end{array}$ | MULTICAND MULTIPLIER |
| | | PARTIAL PRODUCTS |
| | $\begin{array}{r} 1011 \\ 0000 \\ 0000 \\ 1011 \\ \hline \end{array}$ | |
| | 1100011 | PRODUCT |

Fig. 1.9 Binary multiplication.

One of the multiplication methods used by computers is repeated addition. To multiply 7×55 , for example, the computer can just add up seven 55's. For large numbers, however, this method is slow. To multiply 786×253 , for example, requires 252 add operations.

Most computers use an add-and-shift-right method. This method takes advantage of the fact that for binary multiplication, the partial product can only be either the top number exactly if the multiplier digit is a 1 or a 0 if the multiplier digit is a 0. The method does the same thing as the pencil method, except that the partial products are added as they are produced and the sum of the partial products is shifted right rather than each partial product being shifted left.

A point to note about multiplying numbers is the number of bits the product requires. For example, multiplying two 4-bit numbers can give a product with as many as 8-bits, and two 8-bit numbers can give a 16-bit product.

DIVISION

Binary division can also be performed in several ways. Fig. 1.10 shows two examples of the pencil method. This is the same process as decimal long division. However, it is much simpler than decimal long division because the digits of the result (quotient) can only be 0 or 1. A division is attempted on part of the dividend. If this is not possible because the divisor is larger than that part of the dividend, a 0 is entered in the quotient. Another attempt is then made to divide using one more digit of the dividend. When a division is possible, a 1 is entered in the quotient. The divisor is then subtracted from the portion of the dividend used. As with standard long division, the process is continued until all the dividend is used. As shown in Fig. 1.10b, 0's can be added to the right of the binary point and division continued to convert a remainder to a binary equivalent.

| | | |
|--|---|--|
| $\begin{array}{r} 01100 \\ \hline \end{array}$ | QUOTIENT | |
| $\begin{array}{r} 110 \\ \hline \end{array}$ | DIVISOR | $\begin{array}{r} 1001000 \\ \hline \end{array}$ |
| $\begin{array}{r} -110 \\ \hline \end{array}$ | | $\begin{array}{r} 12 \\ \hline \end{array}$ |
| $\begin{array}{r} 110 \\ -110 \\ \hline \end{array}$ | | $\begin{array}{r} 6 \\ \hline \end{array} 72$ |
| | | $\begin{array}{r} 0 \\ \hline \end{array}$ |
| | | (a) |
| $\begin{array}{r} 010.01 \\ \hline \end{array}$ | $\begin{array}{r} 6.25 \\ \hline \end{array}$ | |
| $\begin{array}{r} 110 \\ \hline \end{array}$ | $\begin{array}{r} 25 \\ \hline \end{array}$ | |
| $\begin{array}{r} -100 \\ \hline \end{array}$ | | |
| $\begin{array}{r} 100 \\ -100 \\ \hline \end{array}$ | | |
| | | $\begin{array}{r} 0100 \\ \hline \end{array}$ |
| | | (b) |

Fig. 1.10 Binary division.

Another method of division that is easier for computers and microprocessors to perform uses successive subtractions. The divisor is subtracted from the dividend and from each successive remainder until a borrow is produced. The desired quotient is 1 less than the number of subtractions needed to produce a borrow. This method is simple, but for large numbers it is slow.

For faster division of large numbers, computers use a subtract-and-shift-left method that is essentially the same process you go through with a pencil long division.

Hexadecimal Addition and Subtraction

People working with computers or microprocessors often use hexadecimal as a shorthand way of representing long binary numbers such as memory addresses. It is therefore useful to be able to add and subtract hexadecimal numbers.

ADDITION

As shown in Fig. 1.11a, one way to add two hexadecimal numbers is to convert each hexadecimal number to its binary equivalent, add the two binary numbers, and convert the binary result back to its hex equivalent. For converting to binary, remember that each hex digit represents 4 binary digits.

A second method, shown in Fig. 1.11b, works directly with the hex numbers. When adding hex digits, a carry is produced whenever the sum is 16 decimal or greater. Another way of saying this is that the value of a carry in hex is 16 decimal. For the least significant digits in Fig. 1.11b, an A in hex is 10 in decimal and an F is 15 in decimal. These add to give 25 decimal. This is greater than 16, so mentally subtract 16 from the 25 to give a carry and a remainder of 9. The 9 is written down and the carry is added to the next digit column. In this column 7 plus 3 plus a carry gives a decimal 11, or B in hex.

| | | | | Carry ↓ | |
|------|--------|------|-----|------------------|------------------|
| 7A | 0111 | 1010 | | 7 1 | A ₁₆ |
| + 3F | + 0011 | 1111 | | + 3 | F ₁₆ |
| B9 | 1011 | 1001 | | 11 ₁₀ | 25 ₁₀ |
| | B | 9 | | B ₁₆ | 9 ₁₆ |
| (a) | | | (b) | | |

Fig. 1.11 Hexadecimal addition.

You may use whichever method seems easier to you and gives you consistently right answers. If you are doing a great deal of hexadecimal arithmetic, you might buy an electronic calculator specifically designed to do decimal, binary, and hexadecimal arithmetic.

SUBTRACTION

Hexadecimal subtraction is similar to decimal subtraction except that when a borrow is needed, 16 is borrowed from the next most significant digit. Fig. 1.12 shows an example of this. It may help you to follow the example if you do partial conversions to decimal in your head. For example, 7 plus a borrowed 16 is 23. Subtracting B or 11 leaves 12 or C in hexadecimal. Then 3 from the 6 left after a borrow leaves 3, so the result is 3CH.

$$\begin{array}{r} 77_{16} \\ - 3B_{16} \\ \hline 3C_{16} \end{array} = \begin{array}{r} 119_{10} \\ - 59_{10} \\ \hline 60_{10} \end{array}$$

Fig. 1.12 Hexadecimal subtraction.

BCD Addition and Subtraction

In systems where the final result of a calculation is to be displayed, such as a calculator, it may be easier to work with numbers in a BCD format. These codes, as shown in Table 1.1, represent each decimal digit, 0 through 9, by its 4-bit binary equivalent.

ADDITION

BCD can have no digit-word with a value greater than 9. Therefore, a carry must be generated if the result of a BCD addition is greater than 1001 or 9. Fig. 1.13 shows three examples of BCD addition. The first, in Fig. 1.13a, is very straightforward because the sum for each BCD digit is less than 9. The result is the same as it would be for adding standard binary.

For the second example, in Fig. 1.13b, adding BCD 7 to BCD 5 produces 1100. This is a correct binary result of 12, but it is an illegal BCD code. To convert the result to BCD format, a correction factor of 6 is added. The result of adding 6 is 0001 0010, which is the legal BCD code for 12.

Figure 1.13c shows another case where a correction factor must be added. The initial addition of 9 and 8

| | | BCD | |
|----|------|-------------|----------------|
| 35 | + 23 | 0011 0101 | |
| | | + 0010 0011 | |
| | | <hr/> | |
| 58 | | 0101 1000 | |
| | | (a) | |
| | | | BCD |
| 7 | + 5 | 0111 | |
| | | + 0101 | |
| | | <hr/> | |
| 12 | | 1100 | INCORRECT BCD |
| | | + 0110 | ADD 6 |
| | | <hr/> | |
| | | 0001 0010 | CORRECT BCD 12 |
| | | (b) | |
| | | | BCD |
| 9 | + 8 | 1001 | |
| | | + 1000 | |
| | | <hr/> | |
| 17 | | 0001 0001 | INCORRECT BCD |
| | | 0000 0110 | ADD 6 |
| | | <hr/> | |
| | | 0001 0111 | CORRECT BCD 17 |
| | | (c) | |

Fig. 1.13 BCD addition. (a) No correction needed, (b) Correction needed because of illegal BCD result, (c) Correction needed because of carry-out of BCD digit.

produces 0001 0001. Even though the lower four digits are less than 9, this is an incorrect BCD result because a carry out of bit 3 of the BCD digit-word was produced.

This carry out of bit 3 is often called an *auxiliary carry*. Adding the correction factor of 6 gives the correct BCD result of 0001 0111 or 17.

To summarize, a correction factor of 6 must be added if the result in the lower 4 bits is greater than 9 or if the initial addition produces a carry out of bit 3 of any BCD digit-word. This correction is sometimes called a *decimal adjust operation*.

The reason for the correction factor of 6 is that in BCD we want a carry into the next digit after 1001 or 9, but in binary a carry out of the lower 4 bits does not occur until after 1111 or 15. The difference between the two carry points is 6, so you have to add 6 to produce the desired carry if the result of an addition in any BCD digit is more than 1001.

SUBTRACTION

Figure 1.14 shows a subtraction, BCD 17 (0001 0111) minus BCD 9 (0000 1001). The initial result, 0000 1110, is not a legal BCD number. Whenever this occurs in BCD subtraction, 6 must be *subtracted* from the initial result to produce the correct BCD result. For the example shown in Fig. 1.14, subtracting 6 gives a correct BCD result of 0000 1000 or 8.

| | | |
|-----|-----------|-------------|
| 17 | 0001 0111 | |
| - 9 | 0000 1001 | |
| 8 | 0000 1110 | ILLEGAL BCD |
| | - 0110 | SUBTRACT 6 |
| | 0000 1000 | CORRECT BCD |

Fig. 1.14 BCD subtraction.

The correction factor of 6 must be subtracted from any BCD digit-word if that digit-word is greater than 1001, or if a borrow from the next higher digit was required to do the subtraction.

BASIC DIGITAL DEVICES

Microcomputers such as those we discuss throughout this book often contain basic logic gates as “glue” between LSI (large-scale integration) devices. For troubleshooting these systems, it is important to be able to predict logic levels at any point directly from the schematic rather than having to work your way through a truth table for each gate. This section should help refresh your memory of basic logic functions and help you remember how to quickly analyze logic gate circuits.

Inverting and Noninverting Buffers

Figure 1.15 shows the schematic symbols and truth tables for simple buffers and logic gates. The first thing to remember about these symbols is that the shape of the symbol indicates the logic function performed by the device. The second thing to remember about these symbols is that a bubble or no bubble indicates the assertion level for an input or output signal. Let’s review how modern logic designers use these symbols.

The first symbol for a *buffer* in Fig. 1.15a has no bubbles on the input or output. Therefore, the input is active high and the output is active high. We read this symbol as follows: If the input A is asserted high, then the output Y will be asserted high. The rest of the truth table is covered by the assumption that if the A input is not asserted high, then the Y output will not be asserted high.

The next two symbols for a buffer each contain a bubble. The bubble on the output of the first of these indicates that the output is active low. The input has no

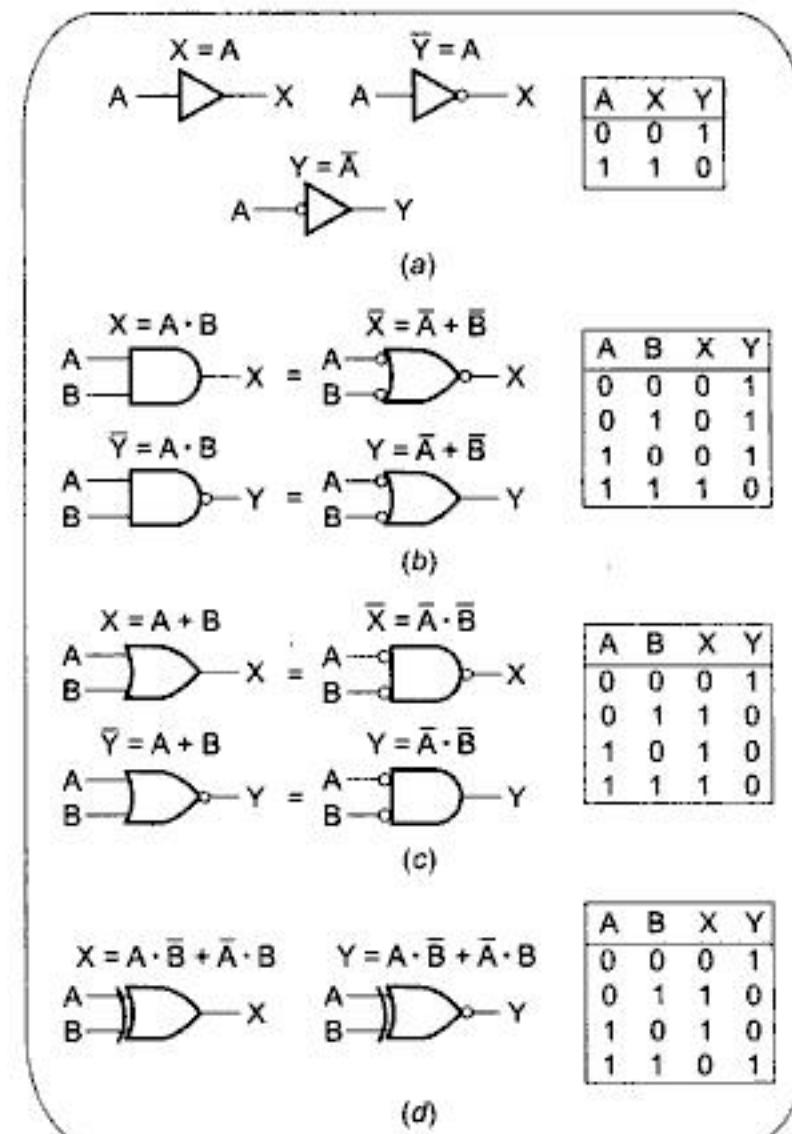


Fig. 1.15 Buffers and logic gates. (a) Buffers. (b) AND-NAND. (c) OR-NOR. (d) Exclusive OR.

bubble, so it is active high. You can read the function of the device directly from the schematic symbol as follows. If the A input is asserted high, then the Y output will be asserted low. This device simply changes the assertion level of a signal. The output Y will always have a logic state which is the complement or inverse of that on the input, so the device is usually referred to as an *inverter*.

The second schematic symbol for an inverter in Fig. 1.15a has the bubble on the input. We draw the symbol this way when we want to indicate that we are using the device to change an asserted-low signal to an asserted-high signal. For example, if we pass the signal CS through this device, it becomes CS. The symbol tells you directly that if the input is asserted low, then the output will be asserted high. Now let's review how you express the functions of logic gates using this approach.

Logic Gates

Figure 1.15b shows the symbols and truth tables for simple logic gates. A symbol with a flat back and a round front indicates that the device performs the logical *AND* function. This means that the output will be asserted if the A input is asserted *and* the B input is asserted. Again, bubbles or no bubbles are used to indicate the assertion level of each input and output. The first AND symbol in Fig. 1.15b has no bubbles, so the inputs and the output are active high. The output then will be asserted high if the A input is asserted high *and* the B input is asserted high. The bubble on the output of the second AND symbol in Fig. 1.15b indicates that this device, commonly called a *NAND* gate, has an active low output. If the A input is asserted high and the B input is asserted high, then the Y output will be asserted low. Look at the truth table in Fig. 1.15b to see if you agree with this.

Figure 1.15c shows the other two possible cases for the AND symbol. The first of these has bubbles on the inputs and on the output. If you see this symbol in a schematic, you should immediately see that the output will be asserted low if the A input is asserted low *and* the B input is asserted low. The second AND symbol in Fig. 1.15c has no bubble on the output, so the output will be asserted high if the A and B inputs are both asserted low.

A logic symbol with a curved back indicates that the output of the device will be asserted if the A input is asserted *or* the B input of the device is asserted. Again, bubbles or no bubbles are used to indicate the assertion level for inputs and outputs. Note in Fig. 1.15b and c that each of the AND symbol forms has an equivalent OR symbol form. An AND symbol with active high inputs and an active high output, for example, represents the same

device (a 74LS08 perhaps) as an OR symbol with active low inputs and an active low output. Use the truth table in Fig. 1.15b to convince yourself of this. The bubbled-OR representation tells you that if one input is asserted low, the output will be low, regardless of the state of the other input. As we will show later in this chapter, this is often a useful way to think of the operation of an AND gate.

Figure 1.15d shows the symbol and truth table for an *exclusive OR* gate and for an *exclusive NOR* gate. The output of an exclusive OR gate will be high if the logic levels on the two inputs are different. The output of an exclusive NOR gate will be high if the logic levels on the two inputs are the same.

You need to be familiar with all these symbols, because most logic designers will use the symbol that best describes the function they want a device to perform in a particular circuit.

Programmable Logic Devices

Instead of using discrete gates, modern microcomputer systems usually use *programmable logic devices* such as PLAs, PROMs, or PALs to implement the "glue" logic between LSI devices. To refresh your memory. Figure 1.16 shows the internal structure of each of these devices. As you can see, they all consist of a programmable AND-OR matrix, so they can easily implement any sum-of-products logic expression. Each AND gate in these figures has up to four inputs, but to simplify the drawing only a single input line is shown. Likewise, the OR gates have several inputs, but are shown with a single input line to simplify the drawing. These devices are programmed by blowing out fuses, which are represented in the figure by Xs. An X in the figure indicates that the fuse is intact and makes a connection between, for example, the output of an AND gate and one of the inputs of an OR gate. A dot at the intersection of two wires indicates a hard-wired connection implemented during manufacture.

In a *programmable logic array* (PLA) or *field programmable logic array* (FPLA), both the AND matrix and the OR matrix are programmable by leaving in fuses or blowing them out. The two programmable matrixes make FPLAs very flexible, but difficult to program.

In a *programmable read-only memory* or PROM, the AND matrix is fixed and just the OR matrix is programmable by leaving in fuses or blowing them out. PROMs implement all the possible product terms for the input variables, so they are useful as code converters.

In a *programmable array logic* device or PAL, the connections in the OR matrix are fixed and the AND matrix connections are programmable. PALs are often used to

implement combinational logic and address decoders in microcomputer systems.

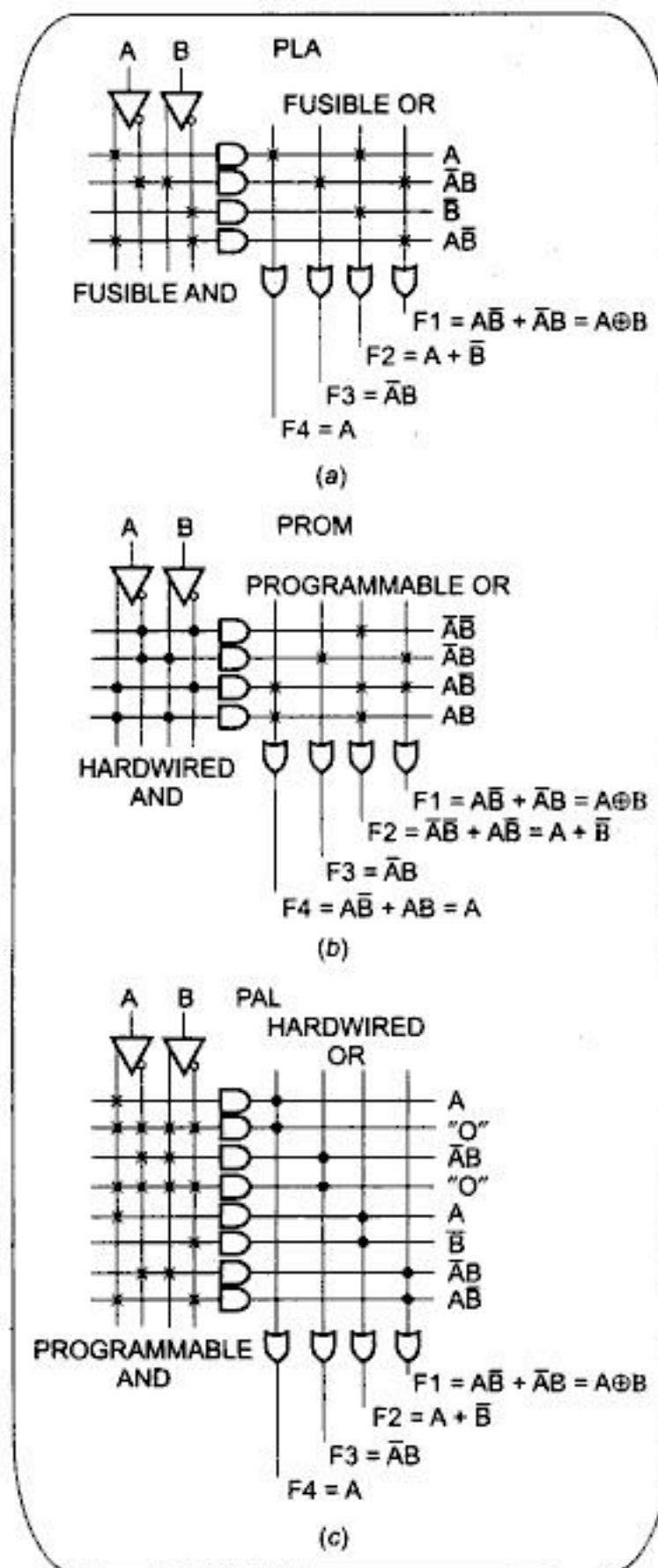


Fig. 1.16 FPLA, PROM, and PAL programmed to implement some simple logic functions. (a) FPLA, (b) PROM, (c) PAL.

A computer program is usually used to develop the fuse map for an FPLA, PROM, or PAL. Once developed, the fuse-map file is downloaded to a programmer which blows fuses or stores charges to actually program the device.

Latches, Flip-Flops, Registers, and Counters

THE D LATCH

A *latch* is a digital device that stores a 1 or a 0 on its output. Fig. 1.17a shows the schematic symbol and truth table for a D latch. The device functions as follows. If the *enable* input CK is low, the logic level present on the D input will have no effect on the Q and \bar{Q} outputs.

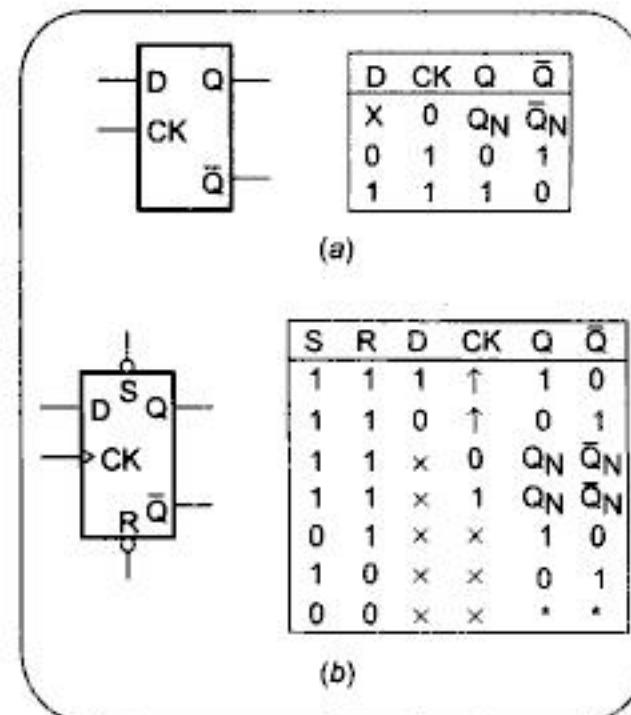


Fig. 1.17 Latches and flip-flops. (a) D latch. (b) D flip-flop.

This is indicated in the truth table by an X in the D column. If the enable input is high, a high or a low on the D input will be passed to the Q output. In other words, the Q output will follow the D input as long as the enable input is high. The \bar{Q} output will contain the complement of the logic state on Q. When the enable input is made low again, the state on Q at that time will be latched there. Any changes on D will have no effect on Q until the enable input is made high again. When the enable input goes low, then, the state present on D just before the enable goes low will be stored on the Q output. Keep this operation in mind as you read about the D flip-flop in the next section.

THE D FLIP-FLOP

Figure 1.17b shows the schematic symbol and the truth table for a typical D flip-flop. The small triangle next to the CK input of this device tells you that the Q and \bar{Q} outputs are updated when a rising signal edge is applied to the CK input. The up arrows in the clock column of the truth table also indicate that a 1 or 0 on the D input will be copied to the Q output when the clock input goes from low to high. In other words, the D flip-flop takes a

snapshot of whatever state is on the D input when the clock goes high, and displays the “photo” on the Q output. If the clock input is low, a change on D will have no effect on the output. Likewise, if the clock input is high, a change on D will have no effect on the Q output. Contrast this operation with that of the D latch to make sure you understand the difference between the two devices.

The D flip-flop in Fig. 1.17b also has direct *set* (S) and *reset* (R) inputs. A flip-flop is considered set if its Q output is a 1. It is *reset* if its Q output is a 0. The bubbles on the set and reset inputs tell you that these inputs are active low. The truth table for the D flip-flop in Fig. 1.17b indicates that the set and reset inputs are *asynchronous*. This means that if the set input is asserted low, the output will be set, regardless of the states on the D and the clock inputs. Likewise, if the reset input is asserted low, the Q output will be reset, regardless of the state of the D and clock inputs. The Xs in the D and CK columns of the truth table remind you that these inputs are “don’t cares” if set or reset is asserted. The condition indicated by the asterisks (*) is a nonstable condition; that is, it will not persist when reset or clear inputs return to their inactive (high) level.

REGISTERS

Flip-flops can be used individually or in groups to store binary data. A *register* is a group of D flip-flops connected in parallel, as shown in Fig. 1.18a. A binary word applied to the data inputs of this register will be transferred to the Q outputs when the clock input is made high. The binary word will remain stored on the Q outputs until a new binary word is applied to the D inputs and a low-to-high signal is applied to the clock input. Other circuitry can read the stored binary word from the Q outputs at any time without changing its value.

If the Q output of each flip-flop in the register is connected to the D input of the next as shown in Fig. 1.18b, then the register will function as a *shift register*. A 1 applied to the first D input will be shifted to the first Q output by a clock pulse. The next clock pulse will shift this 1 to the output of the second flip-flop. Each additional clock pulse will shift the 1 to the next flip-flop in the register. Some shift registers allow you to load a binary word into the register and shift the loaded word left or right when the register is clocked. As we will show later, the ability to shift binary numbers is very useful.

COUNTERS

Flip-flops can also be connected to make devices whose outputs step through a binary or other count sequence when they are clocked. Fig. 1.19a shows a schematic

symbol and count sequence for a presettable 4-bit binary counter. The main point we want to review here is how a presettable counter functions, so there is no need to go into the internal circuitry of the device. If the reset input is asserted, the Q outputs will all be made 0’s. After the reset signal is unasserted, each clock pulse will cause the

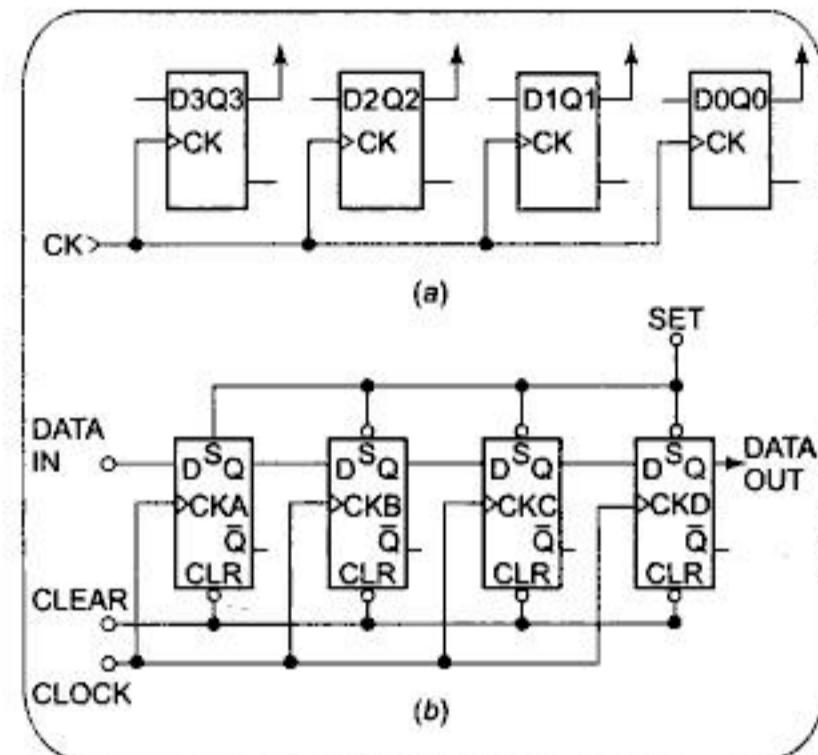


Fig. 1.18 Registers, (a) Simple data storage, (b) Shift register.

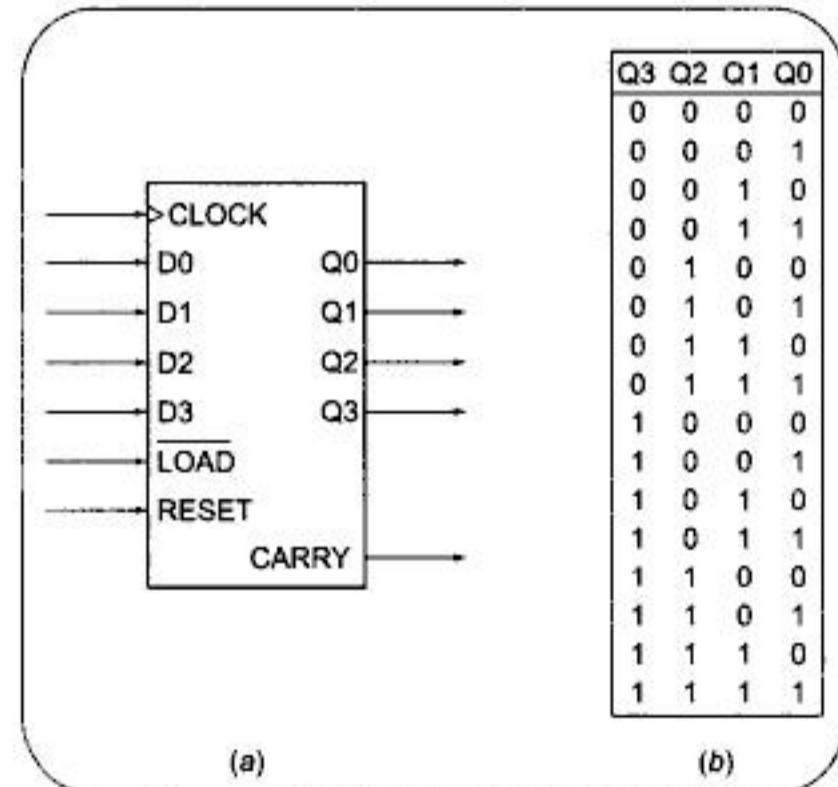


Fig. 1.19 Four-bit, presettable binary counter, (a) Schematic symbol, (b) Count sequence.

binary count on the outputs to be incremented by 1. As shown in Fig. 1.19b, the count sequence will go from 0000 to 1111. If the outputs are at 1111, then the next clock

pulse will cause the outputs to “roll over” to 0000 and a carry pulse to be sent out the carry output. This carry pulse can be used as the clock input for another counter. Counters can be cascaded to produce as large a count sequence as is needed for a particular application. The maximum count for a binary counter is $2^N - 1$, where N is the number of flip-flops.

Now, suppose that we want the counter to start counting from some number other than 0000. We can do this by applying the desired number to the four data inputs and asserting the load input. For example, if we apply a binary 6, 0110, to the data inputs and assert the load input, this value will be transferred to the Q outputs. After the load signal is unasserted, the next clock signal will increment the Q outputs to 0111 or 7.

ROMs, RAMs, and Buses

The next topics we need to review are the devices that store large numbers of binary words and how several of these devices can be connected on common data lines.

ROMS

The term *ROM* stands for *read-only memory*. There are several types of ROM that can be written to, read, erased, and written to with new data, but the main feature of ROMs is that they are *nonvolatile*. This means that the information stored in them is not lost when the power is removed from them.

Figure 1.20a shows the schematic symbol of a common ROM. As indicated by the eight *data* outputs, D0 to D7, this ROM stores 8-bit data words. The data outputs are *three-state* outputs. This means that each output can be at a logic low state, a logic high state, or a high-impedance floating state. In the high-impedance state an output is essentially disconnected from anything connected to it. If the CE input of the ROM is not asserted, then all the outputs will be in the high-impedance state. Most ROMs also switch to a lower-power-consumption standby mode if CE is not asserted. If the \bar{CE} input is asserted, the device will be powered up, and the output buffers will be enabled. Therefore, the outputs will be at a normal logic low or logic high state. If you don't happen to remember, you will soon see why this is important.

You can think of the binary words stored in the ROM as being in a long, numbered list. The number that identifies the location of each stored word in the list is called its *address*. You can tell the number of binary words stored in the ROM by the number of address inputs. The number of words is equal to 2^N , where N is the number of address lines. The device in Fig. 1.20a has 15 address lines, A0 to A14, so the number of words is 2^{15} or 32,768. In a data sheet this device would be referred to as a 32K \times 8 ROM. This means it has 32K addresses with 8 bits per address.

In order to get a particular word onto the outputs of the ROM, you have to do two things. You have to apply

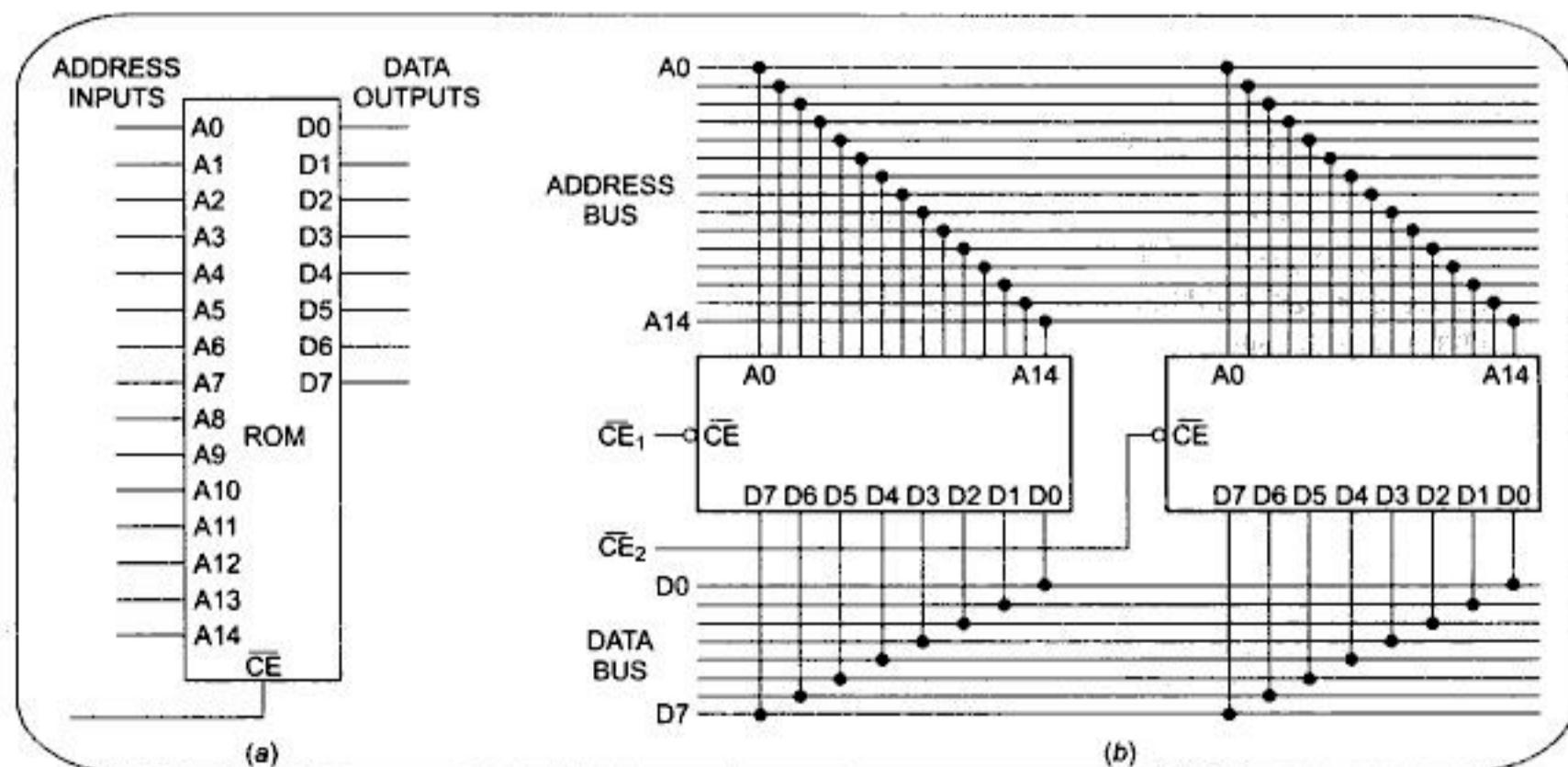


Fig. 1.20 ROMs. (a) Schematic symbol, (b) Connection in parallel.

the address of that word to the address inputs, A0 to A14, and you have to assert the \overline{CE} input to power up the device and to enable the three-state outputs.

Now, let's see why we want three-state outputs on this ROM. Suppose that we want to store more than 32K data words. We can do this by connecting two or more ROMs in parallel, as shown in Fig. 1.20b. The address lines connect to each device in parallel, so we can address one of the 32,768 words in each. A set of parallel lines used to send addresses or data to several devices in this way is called a *bus*. The data outputs of the ROMs are likewise connected in parallel so that any one of the ROMs can output data on the common data bus. If these ROMs had standard two-state outputs, a serious problem would occur when both ROMs tried to output data words on the bus. The resulting argument between data outputs would probably destroy some of the outputs and give meaningless information on the data bus. Since the ROMs have three-state outputs, however, we can use external circuitry to make sure that only one ROM at a time has its outputs enabled. The very important principle here is that whenever several outputs are connected on a bus, the outputs should all be three-state, and only one set of outputs should be enabled at a time.

At the beginning of this section we mentioned that some ROMs can be erased and rewritten or reprogrammed with new data. Here's a summary of the different types of ROMs.

Mask-programmed ROM—Programmed during manufacture; cannot be altered.

PROM—User programs by blowing fuses; cannot be altered except to blow additional fuses.

EPROM—Electrically programmable by user; erased by shining ultraviolet light on quartz window in package.

EEPROM—Electrically programmable by user; erased with electrical signals, so it can be reprogrammed in circuit.

Flash EPROM—Electrically programmable by user; erased electrically, so it can be reprogrammed in circuit.

STATIC AND DYNAMIC RAMS

The name RAM stands for *random-access memory*, but since ROMs are also random access, the name probably should be *read-write memory*. RAMs are also used to store binary words. A *static RAM* is essentially a matrix of flip-flops. Therefore, we can write a new data word in a RAM location at any time by applying the word to the flip-flop data inputs and clocking the flip-flops. The stored data

word will remain on the flip-flop outputs as long as the power is left on. This type of memory is *volatile* because data is lost when the power is turned off.

Figure 1.21 shows the schematic symbol for a common RAM. This RAM has 12 address lines, A0 to A11, so it stores 2^{12} (4096) binary words. The eight data lines tell you that the RAM stores 8-bit words. When we are reading a word from the RAM, these lines function as outputs. When we are writing a word to the RAM, these lines function as inputs. The *chip enable* input, \overline{CE} , is used to enable the device for a read or for a write. The R/W input will be asserted high if we want to read from the RAM or asserted low if we want to write a word to the RAM. Here's how all these lines work for reading from and writing to the device.

To write to the RAM, we apply the desired address to the address inputs, assert the \overline{CE} input low to turn on the device, and assert the R/W input low to tell the RAM we want to write to it. We then apply the data word we want to store to the data lines of the RAM for a specified time. To read a word from the RAM, we address the desired word, assert \overline{CE} low to turn on the device, and assert R/W high to tell the RAM we want to read from it. For a read operation the output buffers on the data lines will be enabled and the addressed data word will be present on the outputs.

The static RAMs we have just reviewed store binary words in a matrix of flip-flops. In *dynamic RAMs* (DRAMs), binary 1's and 0's are stored as an electric charge or no charge on a tiny capacitor. Since these tiny capacitors take up less space on a chip than a flip-flop

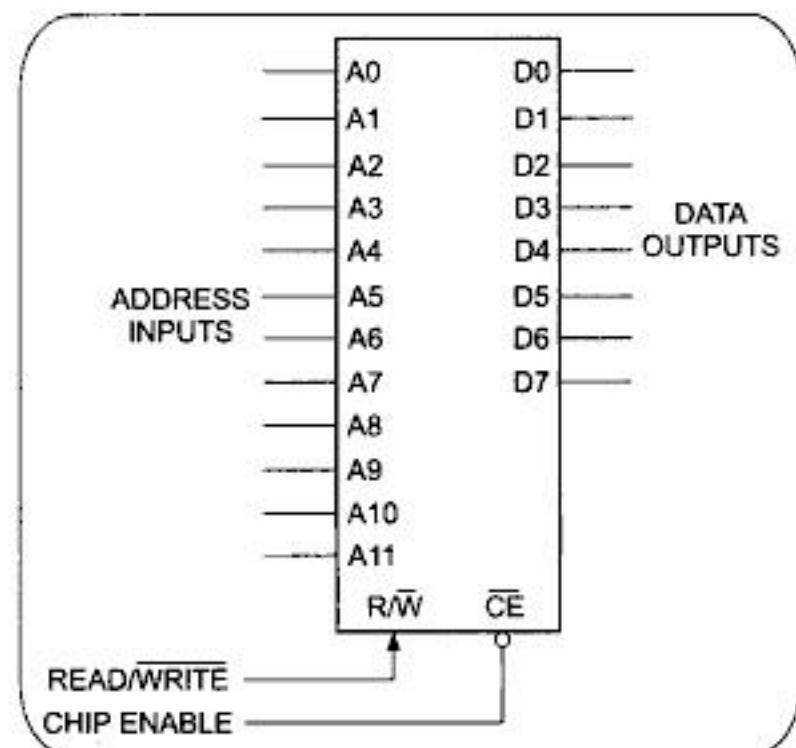


Fig. 1.21 RAM schematic symbol.

would, a dynamic RAM chip can store many more bits than the same size static RAM chip. The disadvantage of dynamic RAMs is that the charge leaks off the tiny capacitors. The logic state stored in each capacitor must be *refreshed* every 2 milliseconds (ms) or so. A device called a *dynamic RAM refresh controller* can be used to refresh a large number of dynamic RAMs in a system. Some newer dynamic RAM devices contain built-in refresh circuitry, so they appear static to external circuitry.

Arithmetic Logic Units

An *arithmetic logic unit*, or *ALU*, is a device that can AND, OR, add, subtract, and perform a variety of other operations on binary words. Fig. 1.22a shows a block diagram for the 74LS181, which is a 4-bit ALU. This device can perform any one of 16 logic functions or any one of 16 arithmetic functions on two 4-bit binary words. The function performed on the two words is determined by the logic level applied to the mode input M and by the 4-bit binary code applied to the select inputs S0 to S3.

Figure 1.22b shows the truth table for the 74LS181. In this truth table, A represents the 4-bit binary word applied to the A0 to A3 inputs, and B represents the 4-bit

binary word applied to the B0 to B3 inputs. F represents the 4-bit binary word that will be produced on the F0 to F3 outputs. If the mode input M is high, the device will perform one of 16 logic functions on the two words applied to the A and B inputs. For example, if M is high and we make S3 high, S2 low, S1 high, and S0 high, the 4-bit word on the A inputs will be ANDed with the 4-bit word on the B inputs. The result of this ANDing will appear on the F outputs. Each bit of the A word is ANDed with the corresponding bit of the B word to produce the result on F. Fig. 1.22c shows an example of ANDing two words with this device. As you can see in this example, an output bit is high only if the corresponding bit is high in both the A word and the B word.

For another example of the operation of the 74LS181, suppose that the M input is high, S3 is high, S2 is high, S1 is high, and S0 is low. According to the truth table, the device will now OR each bit in the A word with the corresponding bit in the B word and give the result on the corresponding F output. Fig. 1.22c shows the result that will be produced by ORing two 4-bit words. Fig. 1.22c also shows for your reference the result that would be produced by exclusive ORing these two 4-bit words together.

| 74LS181 | | SELECTION | ACTIVE-HIGH DATA | | | |
|-------------|-----------|-------------|-----------------------------|------------------------------|-------------------------|--|
| | | | M = H LOGIC FUNCTIONS | M = L; ARITHMETIC OPERATIONS | | |
| | | S3 S2 S1 S0 | $\bar{C}_N = H$ (NO CARRY) | $\bar{C}_N = L$ (WITH CARRY) | | |
| A0 | | L L L L | $F = \bar{A}$ | $F = A$ | $F = A + 1$ | |
| A1 | | L L L H | $F = \bar{A} \oplus B$ | $F = A + B$ | $F = (A + B) + 1$ | |
| A2 | | L L H L | $F = \bar{A}B$ | $F = A + \bar{B}$ | $F = (A + \bar{B}) + 1$ | |
| A3 | F0 | L L H H | $F = 0$ | $F = MINUS 1$ (2's COMPL) | $F = 0$ | |
| B0 | F1 | L H L L | $F = AB$ | $F = A + B$ | $F = (A + \bar{B}) + 1$ | |
| B1 | F2 | L H L H | $F = \bar{A}B$ | $F = A + \bar{B}$ | $F = (A + \bar{B}) + 1$ | |
| B2 | F3 | L H H L | $F = \bar{A}\bar{B}$ | $F = A + B$ | $F = (A + B) + 1$ | |
| B3 | P | L H H H | $F = A \oplus B$ | $F = A + B$ | $F = (A + B) + 1$ | |
| \bar{C}_N | C_{N+4} | L H H L | $F = \bar{A}B$ | $F = A + B$ | $F = (A + B) + 1$ | |
| M | A = B | L H H H | $F = \bar{A}\bar{B}$ | $F = A + B$ | $F = (A + B) + 1$ | |
| S3 S2 S1 S0 | | H L L L | $F = A + B$ | $F = A + B$ | $F = A + B$ | |
| | | H L L H | $F = \bar{A} \oplus B$ | $F = A + B$ | $F = A + B$ | |
| | | H L H L | $F = B$ | $F = A + B$ | $F = A + B$ | |
| | | H L H H | $F = AB$ | $F = AB$ | $F = AB$ | |
| | | H H L L | $F = 1$ | $F = AB$ | $F = AB$ | |
| | | H H L H | $F = A + \bar{B}$ | $F = A + B$ | $F = A + B$ | |
| | | H H H L | $F = A + B$ | $F = A + B$ | $F = A + B$ | |
| | | H H H H | $F = A$ | $F = A$ | $F = A$ | |

*EACH BIT IS SHIFTED TO THE NEXT MORE SIGNIFICANT BIT POSITION

| | | |
|--------------------------|--------------------------|-------------------------------|
| (a) | (b) | (c) |
| $A = A_3\ A_2\ A_1\ A_0$ | $A = 1\ 0\ 1\ 0$ | $A = 1\ 0\ 1\ 0$ |
| $B = B_3\ B_2\ B_1\ B_0$ | $B = 0\ 1\ 1\ 0$ | $B = 0\ 1\ 1\ 0$ |
| $F = F_3\ F_2\ F_1\ F_0$ | $F = A + B = 1\ 1\ 1\ 0$ | $F = A \times B = 0\ 0\ 1\ 0$ |
| | | $F = A \oplus B = 1\ 1\ 0\ 0$ |

Fig. 1.22 Arithmetic logic unit (ALU). (a) Schematic symbol, (b) Truth table, (c) Sample AND, OR, and XOR operations.

If the M input of the 74LS181 is low, then the device will perform one of 16 arithmetic functions on the A and B words. Again, the result of the operation will be put on the F outputs. Several 74LS181s can be cascaded to operate on words longer than 4 bits. The ripple-carry input, C_N , allows a carry from an operation on previous words to be included in the current operation. If the \bar{C}_N input is asserted low, then a carry will be added to the results of the operation on A and B. For example, if the M input is low, S3 is high, S2 is low, S1 is low, S0 is high, and C_N is low, the F outputs will have the sum of A plus B plus a carry.

The real importance of an ALU such as the 74LS181 is that it can be programmed with a binary instruction applied to its mode and select inputs to perform many different functions on two binary words applied to its data inputs. In other words, instead of having to build a different circuit to perform each of these functions, we have one programmable device. We can perform any of the operations that we want in a computer with a sequence of simple operations such as those of the 74LS181. Therefore, an ALU is a very important part of the microprocessors and microcomputers that we discuss in the next chapter.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms or concepts in this list, use the index to find them in the chapter.

Binary, bit, nibble, byte, word, doubleword
 LSB, MSB, LSD, MSD
 Hexadecimal, standard BCD, Gray code
 7-segment display code
 Alphanumeric codes: ASCII, EBCDIC
 Parity bit, odd parity, even parity
 Converting between binary, decimal, hexadecimal, BCD
 Arithmetic with binary, hexadecimal, BCD
 BCD decimal adjust operation
 Signed numbers, sign bit
 2's complement sign-and-magnitude form
 Signal assertion level
 Inverting and noninverting buffers
 Symbols and truth tables for AND, NAND, OR, NOR, XOR logic gates
 PLA, PROM, PAL
 D latch, D flip-flop
 Register, shift register, binary counter
 ROM: address lines, data lines, bus lines, three-state outputs and enable input
 PROM, EPROM, EEPROM, flash EPROM
 RAM: static, dynamic
 ALU

REVIEW QUESTIONS AND PROBLEMS

- Write the decimal equivalent for each integral power of 2 from 2^0 to 2^{20} .
- Convert the following decimal numbers to binary:
 - 22
 - 76
 - 500
- Convert the following binary numbers to decimal:
 - 1011
 - 11010001
 - 1110111001011001
- Convert to hexadecimal:
 - 53 decimal
 - 756 decimal
 - 01101100010 binary
 - 11000010111 binary
- Convert to decimal:
 - D3H
 - 3FEH
 - 44H
- Convert the following decimal numbers to BCD:
 - 86
 - 62
 - 33
- The L key is depressed on an ASCII-encoded keyboard. What pattern of 1's and 0's would you expect to find on the seven parallel data lines coming from the keyboard? What pattern would a carriage return, CR, give?
- Define *parity* and describe how it is used to detect an error in transmitted data.
- Show addition of:
 - 10011_2 and 1011_2 in binary
 - 37_{10} and 25_{10} in BCD
 - 4AH and 77H
- Express the following decimal numbers in 8-bit sign-and-magnitude form:
 - +26
 - 7
 - 26
 - 125
- Show the subtraction, in binary, of the following decimal numbers using both the pencil method and the 2's-complement addition method:
 - $7 - 4$
 - $37 - 26$
 - $125 - 93$

12. Show the multiplication of 1001 and 011 by the pencil method. Do the same for 11010 and 101.
13. Show the division of 1100100 by 1010 using the pencil method.
14. Perform the indicated operations on the following numbers:
 - a. $3AH + 94H$
 - b. $17AH - 4CH$
 - c. $0101\ 1001$ BCD
+ $0100\ 0010$ BCD
 - d. $0111\ 1001$ BCD
+ $0100\ 1001$ BCD
 - e. $0101\ 1001$ BCD
- $0010\ 0110$ BCD
 - f. - $0110\ 0111$ BCD
- $0011\ 1001$ BCD
15. For the circuit in Fig. 1.23:
 - a. Is the Y output active high or active low?
 - b. Is the C signal active high or active low?
 - c. What input conditions on A, B, and C will cause the Y output to be asserted?
16. Describe how a D latch responds to a positive pulse on its CK input and how a D flip-flop responds to a positive pulse on its CK input.
17. The National Semiconductor INS8298 is a 65,536-bit ROM organized as 8192 words or bytes of 8 bits. How many address lines are required to address one of the 8192 bytes?
18. Why do most ROMs and RAMs have three-state outputs?

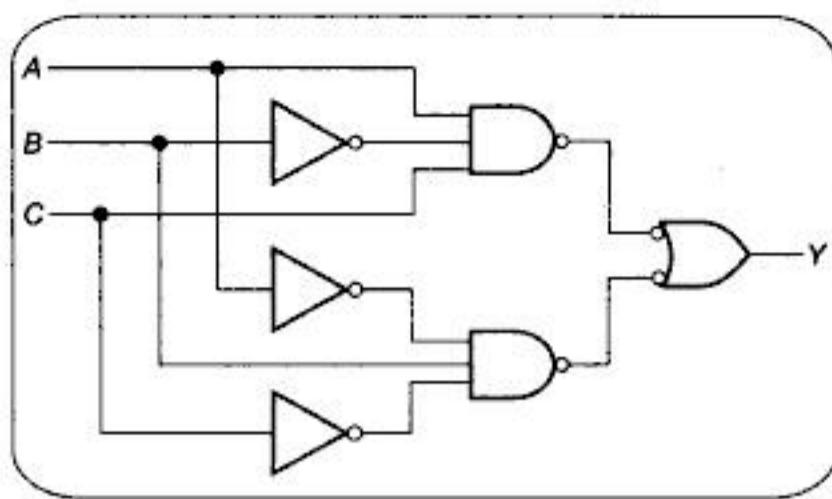


Fig. 1.23 Circuit for problem 15

19. Using Fig. 1.22b, show the programming of the select and mode inputs the 74181 requires to perform the following arithmetic functions:
 - a. $A + B$
 - b. $A - B - 1$
 - c. $AB + A$
20. Show the output word produced when the following binary words are ANDed with each other and when they are ORed with each other:
 - a. 1010 and 0111
 - b. 1011 and 1100
 - c. 11010111 and 111000
 - d. ANDing an 8-bit binary number with 1111 0000 is sometimes referred to as "masking" the lower 4 bits. Why?

Computers, Microcomputers, and Microprocessors—An Introduction

We live in a computer-oriented society, and we are constantly bombarded with a multitude of terms relating to computers. Before getting started with the main flow of the book, we will try to clarify some of these terms and to give an overview of computers and computer systems.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Define the terms *microcomputer, microprocessor, hardware, software, firmware, timesharing, multitasking, distributed processing, and multiprocessing*.
2. Describe how a microcomputer fetches and executes an instruction.
3. List the registers and other parts in the 8086/8088 execution unit and bus interface unit.
4. Describe the function of the 8086/8088 queue.
5. Demonstrate how the 8086/8088 calculates memory addresses.

TYPES OF COMPUTERS

Mainframes

Computers come in a wide variety of sizes and capabilities. The largest and the most powerful ones are often called *mainframes*. Mainframe computers may fill an entire room. They are designed to work at very high speeds with large data words, typically 64 bits or greater, and they have massive amounts of memory. Computers of this type are used for military defence control, for business data processing (in an insurance company, for example), and for creating computer graphics displays for science fiction movies. Examples of this type of computer are the IBM 4381, the Honeywell DPS8, and the Cray Y-MP/832. The fastest and the most powerful mainframes are called *supercomputers*. Fig. 2.1a, shows a photograph of a Cray Y-MP/832 supercomputer, which contains eight central processors and 32 million 64-bit words of memory.

Minicomputers

Scaled-down versions of mainframe computers are often called *minicomputers*. The main unit of a minicomputer usually fits in a single rack or box. A minicomputer runs more slowly, works directly with smaller data words

(often 32-bit words), and does not have as much memory as a mainframe. Computers of this type are used for business data processing, industrial control (for an oil refinery, for example), and scientific research. Examples of this type of computer are the Digital Equipment Corporation VAX 6360 and the Data General MV/8000II. Fig. 2.1b shows a photograph of a Digital Equipment Corporation's VAX 6360 minicomputer.

Microcomputers

As the name implies, *microcomputers* are small computers. They range from small controllers that work directly with 4-bit words and can address a few thousand bytes of memory to larger units that work directly with 32-bit words and can address billions of bytes of memory. Some of the more powerful microcomputers have all or most of the features of earlier minicomputers. Therefore, it has become very hard to draw a sharp line between these two types. One distinguishing feature of a microcomputer is that the CPU is usually a single integrated circuit called a *microprocessor*. Older books often used the terms *microprocessor* and *microcomputer* interchangeably, but actually the microprocessor is the CPU to which you add ROM, RAM, and ports to make a microcomputer. A later section in this chapter discusses the evolution of different types of microprocessors. Microcomputers are used in everything from smart sewing machines to computer-aided design systems. Examples of microcomputers are the Intel 8051 single-chip controller; the SDK-86, a single-board computer design kit; the IBM Personal Computer (PC); and the Apple Macintosh computer. The Intel 8051 microcontroller is contained in a single 40-pin chip. Fig. 2.2a, shows the SDK-86 board, and Fig. 2.2b shows the Compaq 386/25 system.

HOW COMPUTERS AND MICROCOMPUTERS ARE USED—AN EXAMPLE

The following sections are intended to give you an overview of how computers are interfaced with users to do useful work. These sections should help you understand many of the features designed into current microprocessors and where this book is heading.

Computerizing an Electronics Factory—Problem

Now, suppose that we want to “computerize” an electronics company. By this we mean that we want to make computer-use available to as many people in the company

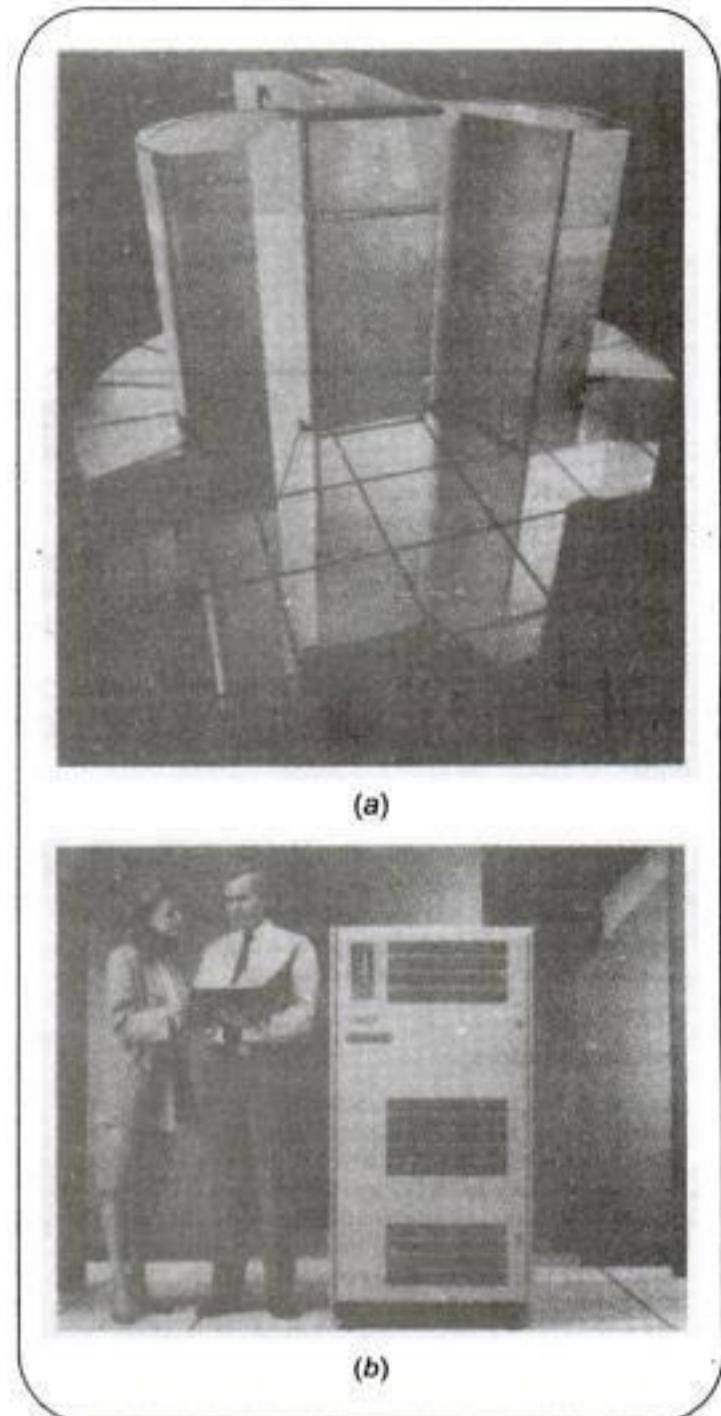


Fig. 2.1 (a) Photograph of Cray Y-MP/832 computer. (Courtesy Cray Research, Inc., and photographer, Paul Shambroom.) (b) Photograph of VAX 6360 minicomputer. (Courtesy Digital Equipment Corp.)

as cheaply as possible. We want the engineers to have access to a computer which can help them design circuits. People in the drafting department should have access to a computer which can be used for computer-aided drafting. The accounting department should have access to a computer for doing all the financial book-keeping. The warehouse should have access to a computer to help with inventory control. The manufacturing department should have access to a computer for controlling machines and testing finished products. The president, vice-presidents, and supervisors should have access to a computer to help

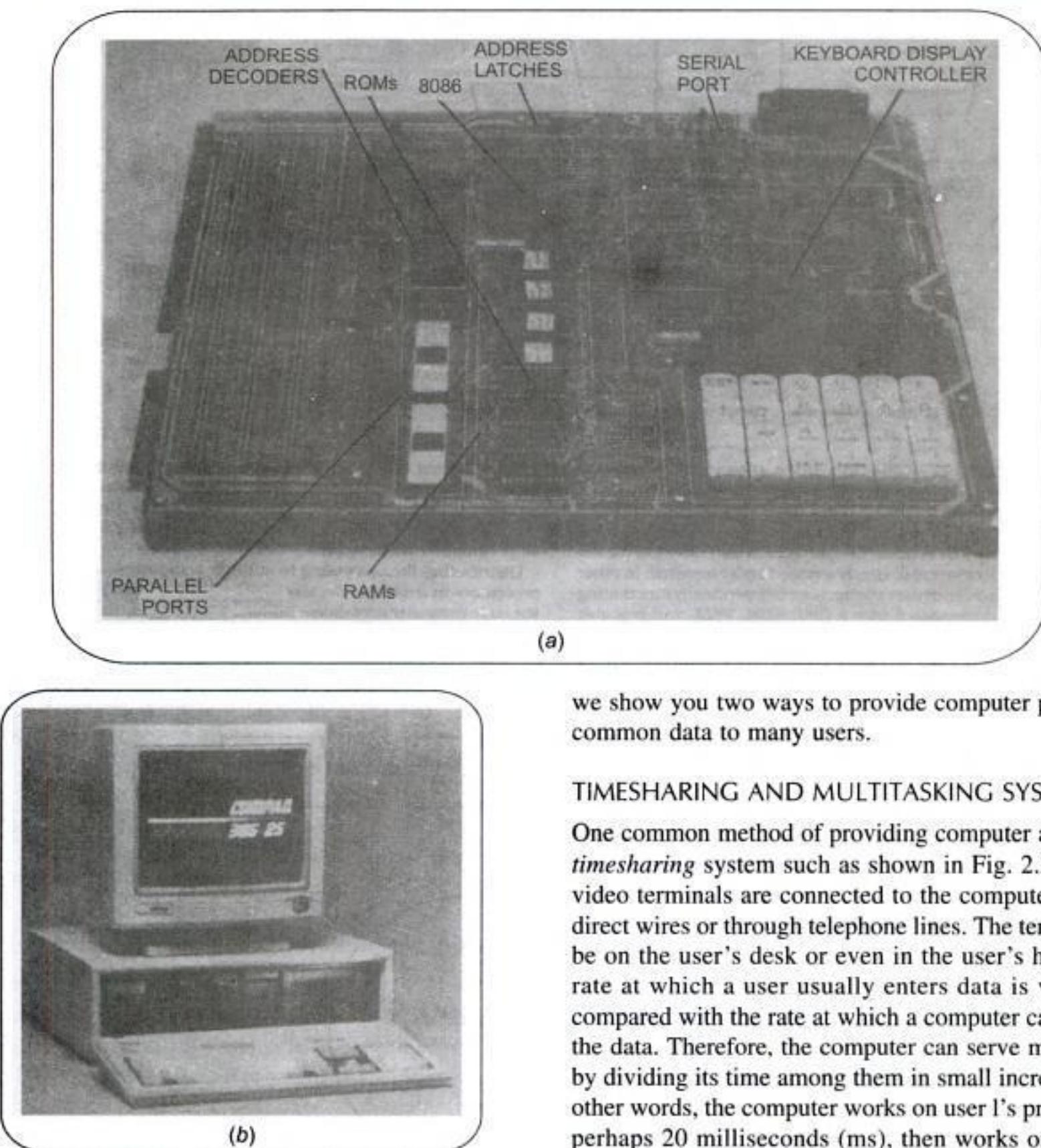


Fig. 2.2 (a) Photograph of Intel SDK-86 board. (Intel Corp.) (b) Photograph of Compaq 386/25. (Compaq Corp.)

them with long-range planning. Secretaries should have access to a computer for word processing. Salespeople should have access to a computer to help them keep track of current pricing, product availability, and commissions. There are several ways to provide all the needed computer power. One solution is to simply give everyone an individual personal computer. The problem with this approach is that it makes it difficult for different people to access commonly needed data. In the next sections,

we show you two ways to provide computer power and common data to many users.

TIMESHARING AND MULTITASKING SYSTEMS

One common method of providing computer access is a *timesharing* system such as shown in Fig. 2.3. Several video terminals are connected to the computer through direct wires or through telephone lines. The terminal can be on the user's desk or even in the user's home. The rate at which a user usually enters data is very slow compared with the rate at which a computer can process the data. Therefore, the computer can serve many users by dividing its time among them in small increments. In other words, the computer works on user 1's program for perhaps 20 milliseconds (ms), then works on user 2's program for 20 ms, then works on user 3's program for 20 ms, and so on, until all the users have had a turn. In a few milliseconds the computer will get back to user 1 again and repeat the cycle. To each user it will appear as if he or she has exclusive use of the computer because the computer processes data as fast as the user enters it. A timesharing system such as this allows several users to interact with the computer at the same time. Each user can get information from or store information in the large memory attached to the computer. Each user can have an inexpensive printer attached to the terminal or can direct program or data output to a high-speed printer attached directly to the computer.

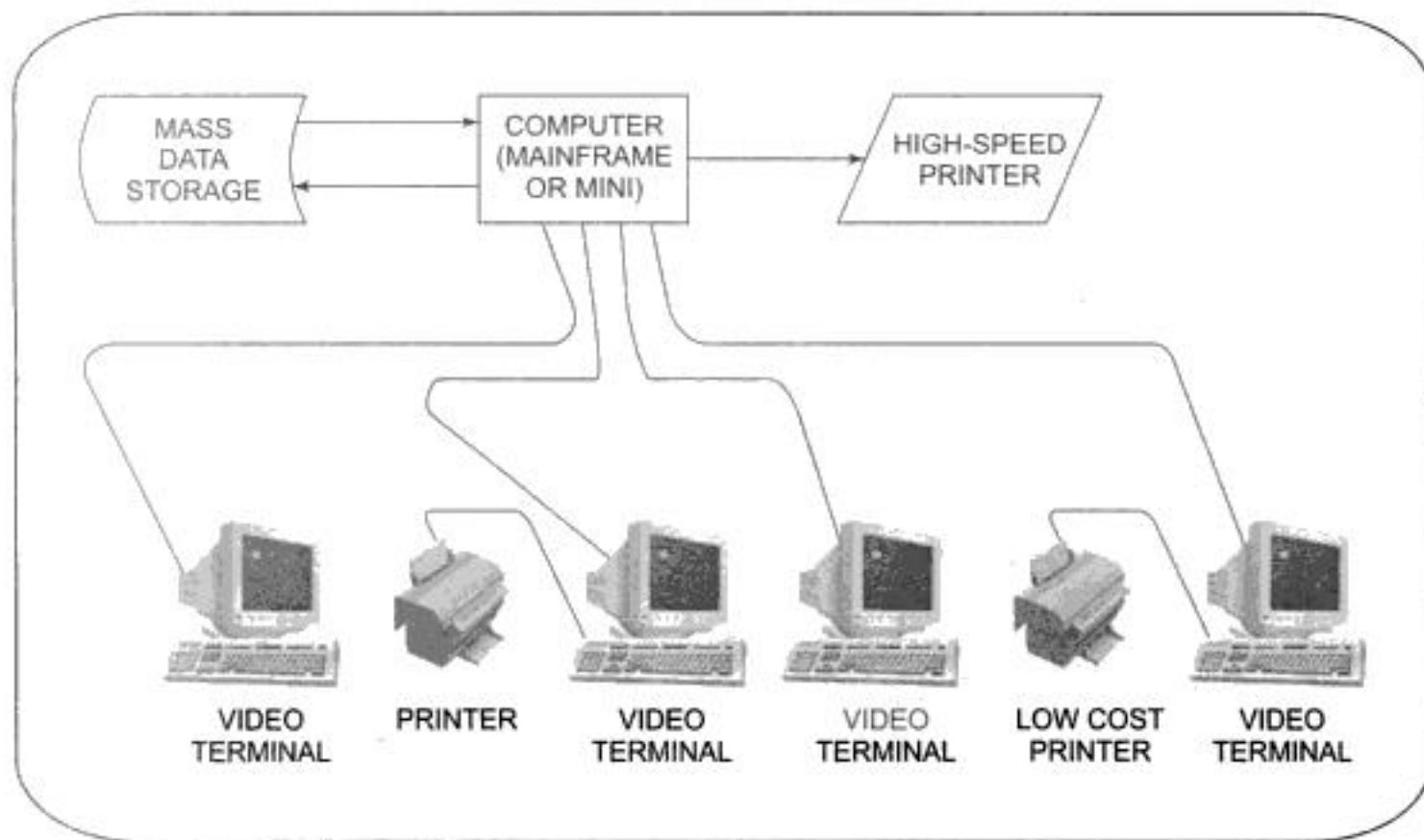


Fig. 2.3 Block diagram of a computer timesharing system.

An airline ticket reservation computer might use a timesharing system such as this to allow users from all over the country to access flight information and make reservations. A time-multiplexed or time-sliced system such as this can also allow a computer to control many machines or processes in a factory. A computer is much faster than the machines or processes. Therefore, it can check and adjust many pressures, temperatures, motor speeds, etc. before it needs to get back and recheck the first one. A system such as this is often called a *multitasking system* because it appears to be doing many tasks at the same time.

Now let's take another look at our problem of computerizing the electronics company. We could put a powerful computer in some central location and run wires from it to video display terminals on users' desks. Each user could then run the program needed to do a particular task. The accountant could run a ledger program, the secretary could run a word processing program, etc. Each user could access the computer's large data memory. Incidentally, a large collection of data stored in a computer's memory is often referred to as a *database*. For a small company, a system such as this might be adequate. However, there are at least two potential problems.

The first potential problem is, "What happens if the computer is not working?" The answer to this question is that everything grinds to a halt. In a situation where people have become dependent on the computer, not much gets done until the computer is up and running again. The

old saying about putting all your eggs in one basket comes to mind here.

The second potential problem of the simple timesharing system is saturation. As the number of users increases, the time it takes the computer to do each user's task increases also. Eventually, the computer's response time to each user becomes unreasonably long. People get very upset about the time they have to wait.

DISTRIBUTED PROCESSING OR MULTIPROCESSING

A partial solution for the two potential problems of a simple timesharing system is to use a *distributed processing* system. Fig. 2.4 shows a block diagram for such a system. The system has a powerful central computer with a large memory and a high-speed printer, as does the simple timesharing system described previously. However, in this system each user has a microcomputer instead of simply a video display terminal. In other words, each user station is an independently functioning microcomputer with a CPU, ROM, RAM, and probably magnetic or optical disk memory. This means that a person can do many tasks locally on the microcomputer without having to use the large computer at all. Since the microcomputers are connected to the large computer through a network, a user can access the computing power, memory, or other resources of the large computer when needed.

Distributing the processing to multiple computers or processors in a system has several advantages. First, if the large computer goes down, the local microcomputers

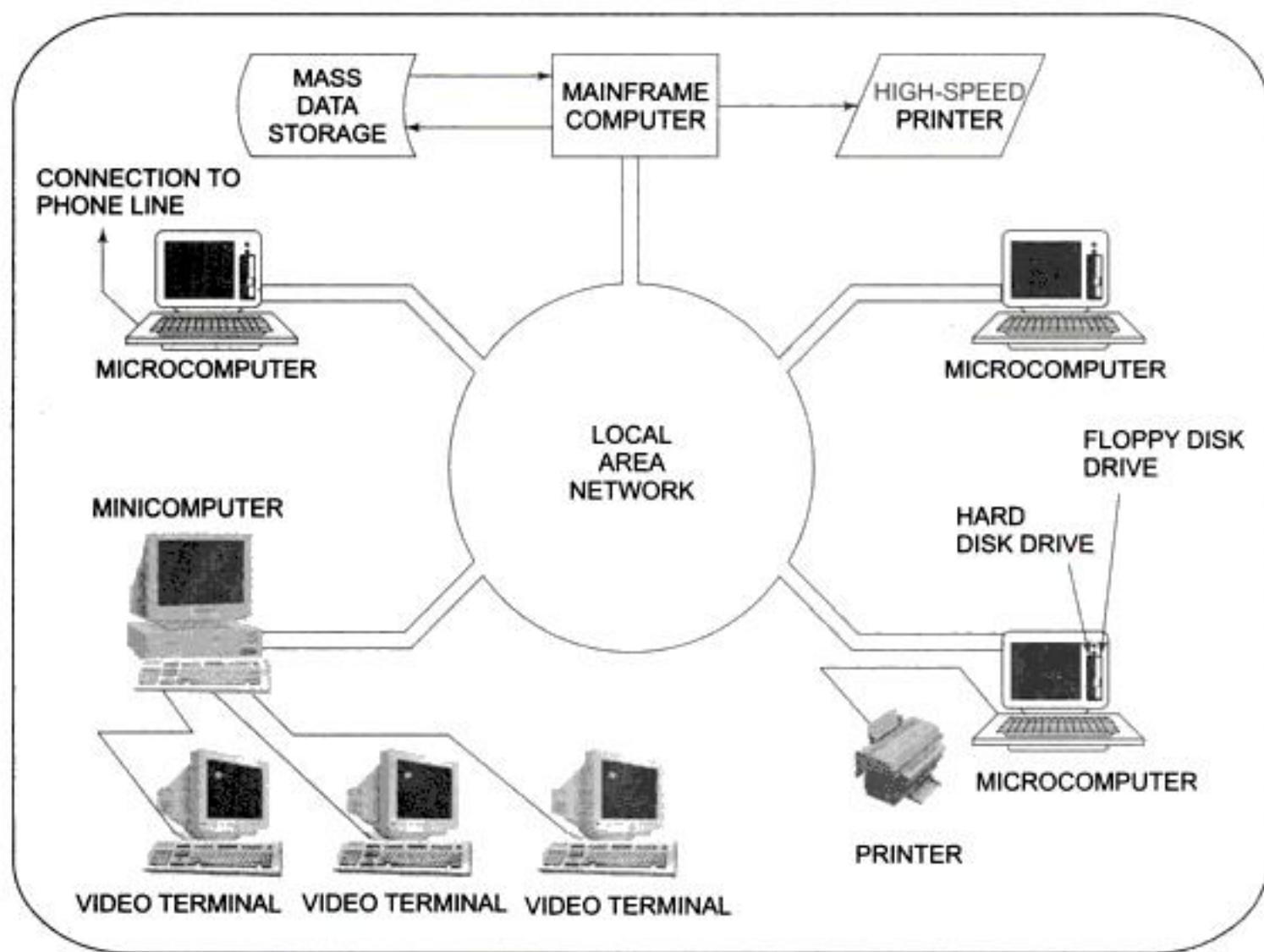


Fig. 2.4 Block diagram of a distributed processing computer system.

can continue working until they need to access the large computer for something. Second, the burden on the large computer is reduced greatly, because much of the computing is done by the local microcomputers. Finally, the distributed processing approach allows the system designer to use a local microcomputer that is best suited to the task it has to do.

COMPUTERIZED ELECTRONICS COMPANY OVERVIEW

Distributed processing seems to be the best way to go about computerizing our electronics factory. Engineers can have personal computers or engineering works-stations on their desks. With these they can use available programs to design and test circuits. They can access the large computer if they need data from its memory. Through the telephone lines, the engineer with a personal computer can access data in the memory of other computers all over the world. The drafting people can have personal computers for simple work, or large computer-aided design systems for more complex work. Completed work can be stored in the memory of the large computer. The production department can have networked computers to keep track of product flow and to control the machines which actually mount components on circuit boards, etc.

The accounting department can use personal computers with spreadsheet programs to work with financial data kept in the memory of the large computer. The warehouse supervisor can likewise, use a personal computer with an inventory program to keep personal records and the records the large computer's memory updated. Corporate officers can have personal computers tied into the network. They then can interact with any of the other systems on the network. Salespeople can have portable personal computers that they can carry with them in the field. They can communicate with the main computer over the telephone lines using a modem. Secretaries doing word processing can use individual word processing units or personal computers. Users can also send messages to one another over the network. The specifics of a computer system such as this will obviously depend on the needs of the individual company for which the system is designed.

SUMMARY AND DIRECTION FROM HERE

The main concepts that you should take with you from this section are timesharing or multitasking and distributed processing or multiprocessing. As you work your way through the rest of this book, keep an overview of the

computerized electronics company in the back of your mind. The goal of this book is to teach you how the microcomputers and other parts of a system such as this work, how the parts are connected together, and how the system is programmed at different levels.

OVERVIEW OF MICROCOMPUTER STRUCTURE AND OPERATION

Figure 2.5 shows a block diagram for a simple microcomputer. The major parts are the *central processing unit* or CPU, *memory*, and the *input and output* circuitry or I/O. Connecting these parts are three sets of parallel lines called *buses*. The three buses are the *address bus*, the *data bus*, and the *control bus*. Let's take a brief look at each of these parts.

Memory

The memory section usually consists of a mixture of RAM and ROM. It may also have magnetic floppy disks, magnetic hard disks, or optical disks. Memory has two purposes. The first purpose is to store the binary codes for the sequences of instructions you want the computer to carry out. When you write a computer program, what you are really doing is writing a sequential list of instructions for the computer. The second purpose of the memory is to store the binary-coded data with which the computer is going to be working. This data might be, for example, the inventory records of a supermarket.

Input/Output

The input/output or I/O section allows the computer to take in data from the outside world or send data to the outside world. Peripherals such as keyboards, video display terminals, printers, and modems are connected to the I/O section. These allow the user and the computer

to communicate with each other. The actual physical devices used to interface the computer buses to external systems are often called *ports*. Ports in a computer function just as shipping ports do for a country. An *input port* allows data from a keyboard, an A/D converter, or some other source to be read into the computer under control of the CPU. An *output port* is used to send data from the computer to some peripheral, such as a video display terminal, a printer, or a D/A converter. Physically, the simplest type of input or output port is just a set of parallel D flip-flops. If they are being used as an input port, the D inputs are connected to the external device, and the Q outputs are connected to the data bus which runs to the CPU. Data will then be transferred through the latches when they are enabled by a control signal from the CPU. In a system where they are being used as an output port, the D inputs of the latches are connected to the data bus, and the Q outputs are connected to some external device. Data sent out on the data bus by the CPU will be transferred to the external device when the latches are enabled by a control signal from the CPU.

Central Processing Unit

The central processing unit or CPU controls the operation of the computer. In a microcomputer the CPU is a microprocessor, as we discussed in an earlier section of the chapter. The CPU fetches binary-coded instructions from memory, decodes the instructions into a series of simple actions, and carries out these actions in a sequence of steps.

The CPU also contains an *address counter* or *instruction pointer* register, which holds the address of the next instruction or data item to be fetched from memory; general-purpose registers, which are used for temporary storage of binary data; and circuitry, which generates the control bus signals.

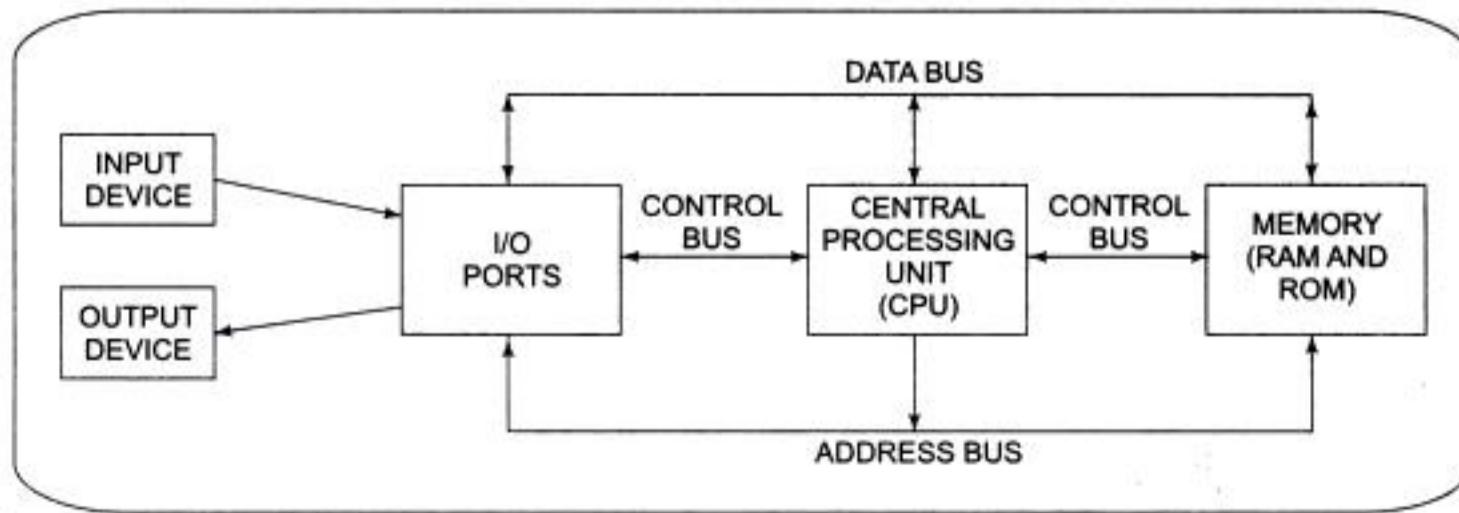


Fig. 2.5 Block diagram of a simple microcomputer.

Address Bus

The address bus consists of 16, 20, 24, or 32 parallel signal lines. On these lines the CPU sends out the address of the memory location that is to be written to or read from. The number of memory locations that the CPU can address is determined by the number of address lines. If the CPU has N address lines, then it can directly address 2^N memory locations. For example, a CPU with 16 address lines can address 2^{16} or 65,536 memory locations, a CPU with 20 address lines can address 2^{20} or 1,048,576 locations, and a CPU with 24 address lines can address 2^{24} or 16,777,216 locations. When the CPU reads data from or writes data to a port, it sends the port address out on the address bus.

Data Bus

The data bus consists of 8, 16, or 32 parallel signal lines. As indicated by the double-ended arrows on the data bus line in Fig. 2.5, the data bus lines are *bidirectional*. This means that the CPU can read data in from memory or from a port on these lines, or it can send data out to memory or to a port on these lines. Many devices in a system will have their outputs connected to the data bus, but only one device at a time will have its outputs enabled. Any device connected on the data bus must have *three-state outputs* so that its outputs can be disabled when it is not being used to put data on the bus.

Control Bus

The control bus consists of 4 to 10 parallel signal lines. The CPU sends out signals on the control bus to enable the outputs of addressed memory devices or port devices. Typical control bus signals are *Memory Read*, *Memory Write*, *I/O Read*, and *I/O Write*. To read a byte of data from a memory location, for example, the CPU sends out the memory address of the desired byte on the address bus and then sends out a Memory Read signal on the control bus. The Memory Read signal enables the addressed memory device to output a data word onto the data bus. The data word from memory travels along the data bus to the CPU.

Hardware, Software, and Firmware

When working around computers, you hear the terms hardware, software, and firmware almost constantly. *Hardware* is the name given to the physical devices and circuitry of the computer. *Software* refers to the programs written for the computer. *Firmware* is the term given to programs stored in ROMs or in other devices which permanently keep their stored information.

Summary of Important Points So Far

- A computer or microcomputer consists of memory, a CPU, and some input/output circuitry.
- These three parts are connected by the address bus, the data bus, and the control bus.
- The sequence of instructions or program for a computer is stored as binary numbers in successive memory locations.
- The CPU fetches an instruction from memory, decodes the instruction to determine what actions must be done for the instruction, and carries out these actions.

EXECUTION OF A THREE-INSTRUCTION PROGRAM

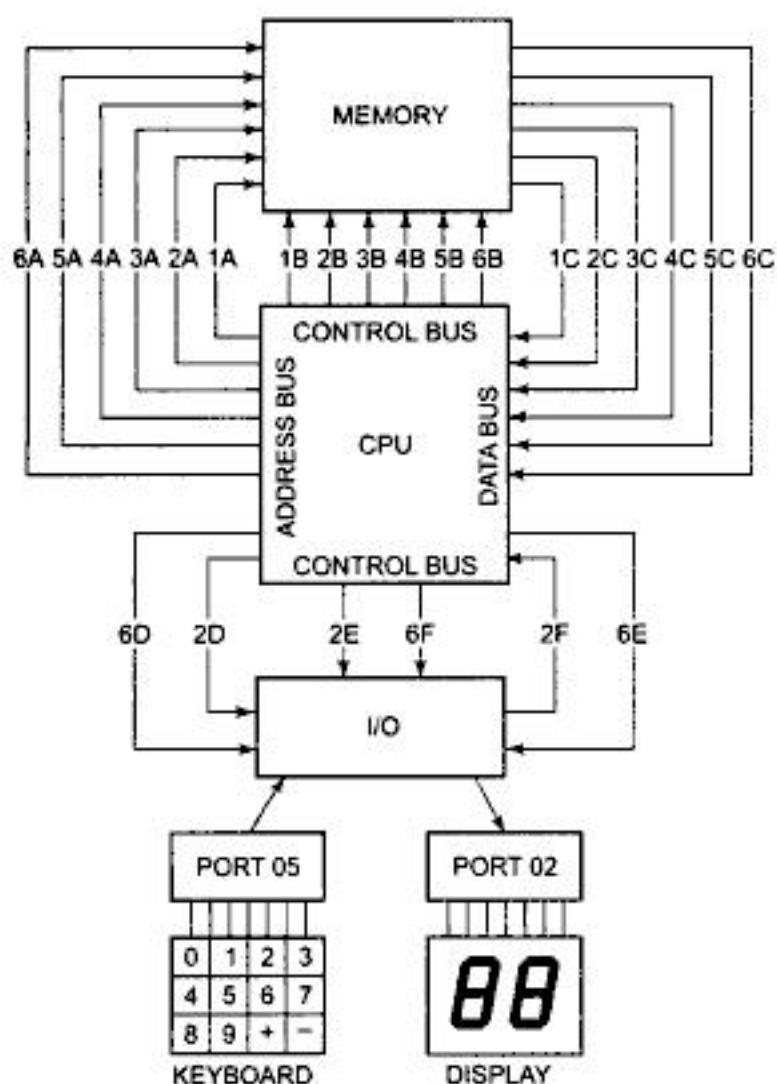
To give you a better idea of how the parts of a microcomputer function together, we will now describe the actions a simple microcomputer might go through to carry out (execute) a simple program. The three instructions of the program are

1. Input a value from a keyboard connected to the port at address 05H.
2. Add 7 to the value.
3. Output the result to a display connected to the port at address 02H.

Figure 2.6 shows in diagram form and sequential list form the actions that the computer will perform to execute these three instructions.

For this example, assume that the CPU fetches instructions and data from memory 1 byte at a time, as is done in the original IBM PC and its clones. Also assume that the binary codes for the instructions are in sequential memory locations starting at address 00100H. Fig. 2.6b shows the actual binary codes that would be required in successive memory locations to execute this program on an IBM PC-type microcomputer.

The CPU needs an instruction before it can do anything, so its first action is to fetch an instruction byte from memory. To do this, the CPU sends out the address of the first instruction byte, in this case 00100H, to memory on the address bus. This action is represented by line 1A in Fig. 2.6a. The CPU then sends out a Memory Read signal on the control bus (line 1B in the figure). The Memory Read signal enables the memory to output the addressed byte on the data bus. This action is represented by line 1C in the figure. The CPU reads in this first instruction byte (E4H) from the data bus and *decodes* it. By *decode* we mean that the CPU determines from the

**PROGRAM**

1. INPUT A VALUE FROM PORT 05.
2. ADD 7 TO THIS VALUE.
3. OUTPUT THE RESULT TO PORT 02.

SEQUENCE

- 1A CPU SENDS OUT ADDRESS OF FIRST INSTRUCTION TO MEMORY.
- 1B CPU SENDS OUT MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
- 1C INSTRUCTION BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
- 2A ADDRESS NEXT MEMORY LOCATION TO GET REST OF INSTRUCTION.
- 2B SEND MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
- 2C PORT ADDRESS BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
- 2D CPU SENDS OUT PORT ADDRESS ON ADDRESS BUS.
- 2E CPU SENDS OUT INPUT READ CONTROL SIGNAL TO ENABLE PORT.
- 2F DATA FROM PORT SENT TO CPU ON DATA BUS.
- 3A CPU SENDS ADDRESS OF NEXT INSTRUCTION TO MEMORY.
- 3B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
- 3C INSTRUCTION BYTE FROM MEMORY SENT TO CPU ON DATA BUS.
- 4A CPU SENDS NEXT ADDRESS TO MEMORY TO GET REST OF INSTRUCTION.
- 4B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
- 4C NUMBER 07H SENT FROM MEMORY TO CPU ON DATA BUS.
- 5A CPU SENDS ADDRESS OF NEXT INSTRUCTION TO MEMORY.
- 5B CPU SENDS MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
- 5C INSTRUCTION BYTE FROM MEMORY SENT TO CPU ON DATA BUS.
- 6A CPU SENDS OUT NEXT ADDRESS TO GET REST OF INSTRUCTION.
- 6B CPU SENDS OUT MEMORY READ CONTROL SIGNAL TO ENABLE MEMORY.
- 6C PORT ADDRESS BYTE SENT FROM MEMORY TO CPU ON DATA BUS.
- 6D CPU SENDS OUT PORT ADDRESS ON ADDRESS BUS.
- 6E CPU SENDS OUT DATA TO PORT ON DATA BUS.
- 6F CPU SENDS OUT OUTPUT WRITE SIGNAL TO ENABLE PORT.

(a)

| MEMORY ADDRESS | CONTENTS (BINARY) | CONTENTS (HEX) | OPERATION |
|----------------|-------------------|----------------|---------------------|
| 00100H | 11100100 | E4 | INPUT FROM PORT 05H |
| 00101H | 00000101 | 05 | |
| 00102H | 00000100 | 04 | ADD |
| 00103H | 00000111 | 07 | 07H |
| 00104H | 11100110 | E6 | OUTPUT TO PORT 02 |
| 00105H | 00000010 | 02 | |

(b)

Fig. 2.6 (a) Execution of a three-step computer program. (b) Memory addresses and memory contents for a three-step program.

binary code read in, what actions it is supposed to take. If the CPU is a microprocessor, it selects the sequence of microinstructions needed to carry out the instruction read from memory. For the example instruction here, the CPU determines that the code read in represents an Input instruction. From decoding this instruction byte, the CPU also determines that it needs more information before it can carry out the instruction. The additional information the CPU needs is the address of the port that the data is to be input from. This port address part of the instruction is stored in the next memory location after the code for the Input instruction.

To fetch this second byte of the instruction, the CPU sends out the next sequential address (00101H) to memory, as shown by line 2A in the figure. To enable the addressed memory device, the CPU also sends out another Memory

Read signal on the control bus (line 2B). The memory then outputs the addressed byte on the data bus (line 2C). When the CPU has read in this second byte, 05H in this case, it has all the information it needs to execute the instruction.

To execute the Input instruction, the CPU sends out the port address (05H) on the address bus (line 2D) and sends out an I/O Read signal on the control bus (line 2E). The I/O Read signal enables the addressed port device to put a byte of data on the data bus (line 2F). The CPU reads in the byte of data and stores it in an internal register. This completes the fetching and execution of the first instruction.

Having completed the first instruction, the CPU must now fetch its next instruction from memory. To do this, it sends out the next sequential address (00102H) on the address bus (line 3A) and sends out a Memory Read signal

on the control bus (line 3B). The Memory Read signal enables the memory device to put the addressed byte (04H) on the data bus (line 3C). The CPU reads in this instruction byte from the data bus and decodes it. From this instruction byte the CPU determines that it is supposed to add some number to the number stored in the internal register. The CPU also determines from decoding this instruction byte that it must go to memory again to get the next byte of the instruction, which contains the number that it is supposed to add. To get the required byte, the CPU will send out the next sequential address (00103H) on the address bus (line 4A) and another Memory Read signal on the control bus (line 4B). The memory will then output the contents of the addressed byte (the number 07H) on the data bus (line 4C). When the CPU receives this number, it will add it to the contents of the internal register. The result of the addition will be left in the internal register. This completes the fetching and executing of the second instruction.

The CPU must now fetch the third instruction. To do this, it sends out the next sequential address (00104H) on the address bus (line 5A) and sends out a Memory Read signal on the control bus (line 5B). The memory then outputs the addressed byte (E6H) on the data bus (line 5C). From decoding this byte, the CPU determines that it is now supposed to do an Output operation to a port. The CPU also determines from decoding this byte that it must go to memory again to get the address of the output port. To do this, it sends out the next sequential address (00105H) on the address bus (line 6A), sends out a Memory Read signal on the control bus (line 6B), and reads in the byte (02H) put on the data bus by the memory (line 6C). The CPU now has all the information that it needs to execute the Output instruction.

To output a data byte to a port, the CPU first sends out the address of the desired port on the address bus (line 6D). Next it outputs the data byte from the internal register on the data bus (line 6E). The CPU then sends out an I/O Write signal on the control bus (line 6F). This signal enables the addressed output port device so that the data from the data bus lines can pass through it to the LED displays. When the CPU removes the I/O Write signal to proceed with the next instruction, the data will remain latched on the output pins of the port device. The data will remain latched on the port until the power is turned off or until a new data word is output to the port. This is important because it means that the computer does not have to keep outputting a value over and over in order for it to remain on the output.

All the steps described above may seem like a great deal of work just to input a value from a keyboard, add 7 to it, and output the result to a display. Even a simple microcomputer, however, can run through all these steps in a few microseconds.

Summary of Simple Microcomputer Bus Operation

1. A microcomputer fetches each program instruction in sequence, decodes the instruction, and executes it.
2. The CPU in a microcomputer fetches instructions or reads data from memory by sending out an address on the address bus and a Memory Read signal on the control bus. The memory outputs the addressed instruction or data word to the CPU on the data bus.
3. The CPU writes a data word to memory by sending out an address on the address bus, sending out the data word on the data bus, and sending a Memory Write signal to memory on the control bus.
4. To read data from a port, the CPU sends out the port address on the address bus and sends an I/O Read signal to the port device on the control bus. Data from the port comes into the CPU on the data bus.
5. To write data to a port, the CPU sends out the port address on the address bus, sends out the data to be written to the port on the data bus, and sends an I/O Write signal to the port device on the control bus.

MICROPROCESSOR EVOLUTION AND TYPES

As we told you in the preceding section, a microprocessor is used as the CPU in a microcomputer. There are now many different microprocessors available, so before we dig into the details of a specific device, we will give you a short microprocessor history lesson and an overview of the different types.

Microprocessor Evolution

A common way of categorizing microprocessors is by the number of bits that their ALU can work with at a time. In other words, a microprocessor with a 4-bit ALU will be referred to as a 4-bit microprocessor, regardless of the number of address lines or the number of data bus lines that it has. The first commercially available microprocessor was the Intel 4004, produced in 1971. It contained 2300 PMOS transistors. The 4004 was a 4-bit

device intended to be used with some other devices in making a calculator. Some logic designers, however, saw that this device could be used to replace PC boards full of combinational and sequential logic devices. Also, the ability to change the function of a system by just changing the programming, rather than redesigning the hardware, is very appealing. It was these factors that pushed the evolution of microprocessors.

In 1972 Intel came out with the 8008, which was capable of working with 8-bit words. The 8008, however, required 20 or more, additional devices to form a functional CPU. In 1974, Intel announced the 8080, which had a much larger instruction set than the 8008 and required only two additional devices to form a functional CPU. Also, the 8080 used NMOS transistors, so it operated much faster than the 8008. The 8080 is referred to as a *second-generation microprocessor*.

Soon after Intel produced the 8080, Motorola came out with the MC6800, another 8-bit general-purpose CPU. The 6800 had the advantage that it required only a +5-V supply rather than the -5-V, +5-V, and +12-V supplies required by the 8080. For several years, the 8080 and the 6800 were the top-selling 8-bit microprocessors. Some of their competitors were the MOS Technology 6502, used as the CPU in the Apple II microcomputer, and the Zilog Z80, used as the CPU in the Radio Shack TRS-80 microcomputer.

As designers found more and more applications for microprocessors, they pressured microprocessor manufacturers to develop devices with architectures and features optimized for doing certain types of tasks. In response to the expressed needs, microprocessors have evolved in three major directions during the last 15 years.

Dedicated or Embedded Controllers

One direction in the *dedicated or embedded controllers*. These devices are used to control "smart" machines, such as microwave ovens, clothes washers, sewing machines, auto ignition systems, and metal lathes. Texas Instruments has produced millions of their TMS-1000 family of 4-bit microprocessors for this type of application. In 1976 Intel introduced the 8048, which contains an 8-bit CPU, RAM, ROM, and some I/O ports all in one 40-pin package. Other manufacturers have followed with similar products. These devices are often referred to as *microcontrollers*. Some currently available devices in this category—the Intel 8051 and the Motorola MC6801, for example—contain programmable counters and a serial port (UART) as well as a CPU, ROM, RAM, and parallel I/O ports. A more recently introduced single-chip microcontroller, the Intel 8096, contains a 16-bit CPU, ROM, RAM, a UART, ports, timers, and a 10-bit analog-to-digital converter.

Bit-Slice Processors

A second direction of microprocessor evolution has been *bit-slice processors*. For some applications, general-purpose CPUs such as the 8080 and 6800 are not fast enough or do not have suitable instruction sets. For these applications, several manufacturers produce *devices* which can be used to build a custom CPU. An example is the Advanced Micro Devices 2900 family of devices. This family includes 4-bit ALUs, multiplexers, sequencers, and other parts needed for custom-building a CPU. The term *slice* comes from the fact that these parts can be connected in parallel to work with 8-bit words, 16-bit words, or 32-bit words. In other words, a designer can add as many slices as needed for a particular application. The designer not only custom-designs the hardware of the CPU, but also custom-makes the instruction set for it, using "microcode."

General-Purpose CPUs

The third major direction of microprocessor evolution has been towards general-purpose CPUs which give a microcomputer most or all of the computing power of earlier minicomputers. After Motorola came out with the MC6800, Intel produced the 8085, an upgrade of the 8080 that required only a +5-V supply. Motorola then produced the MC6809, which has a few 16-bit instructions, but is still basically an 8-bit processor. In 1978 Intel came out with the 8086, which is a full 16-bit processor. Some 16-bit microprocessors, such as the National PACE and the Texas Instruments 9900 family of devices, had been available previously, but the market apparently wasn't ready. Soon after Intel came out with the 8086, Motorola came out with the 16-bit MC68000, and the 16-bit race was off and running. The 8086 and the 68000 work directly with 16-bit words instead of with 8-bit words, they can address a million or more bytes of memory instead of the 64 Kbytes addressable by the 8-bit processors, and they execute instructions much faster than the 8-bit processors. Also, these 16-bit processors have single instructions for functions such as *multiply* and *divide*, which required a lengthy sequence of instructions on the 8-bit processors.

The evolution along this last path has continued on to 32-bit processors that work with gigabytes (10^9 bytes) or terabytes (10^{12} bytes) of memory. Examples of these devices are the Intel 80386, the Motorola MC68020, and the National 32032.

Since we could not possibly describe in this book the operation and programming of even a few of the available processors, we confine our discussions primarily to one

group of related microprocessors. The family we have chosen is the Intel 8086, 8088, 80186, 80188, 80286, 80386, 80486 family. Members of this family are very widely used in personal computers, business computer systems, and industrial control systems. Our experience has shown that learning the programming and operation of one family of microcomputers very thoroughly is much more useful than looking at many processors superficially. If you learn one processor family well, you will most likely find it quite easy to learn another when you have to.

THE 8086 MICROPROCESSOR FAMILY—OVERVIEW

The Intel 8086 is a 16-bit microprocessor that is intended to be used as the CPU in a microcomputer. The term *16-bit* means that its arithmetic logic unit, its internal registers, and most of its instructions are designed to work with 16-bit binary words. The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time. The 8086 has a 20-bit address bus, so it can address any one of 2^{20} , or 1,048,576, memory locations.

Each of the 1,048,576 memory addresses of the 8086 represents a byte-wide location. Sixteen-bit words will be stored in two consecutive memory locations. If the first byte of a word is at an even address, the 8086 can read the entire word in one operation. If the first byte of the word is at an odd address, the 8086 will read the first byte with one bus operation and the second byte with another bus operation. Later we will discuss this in detail. The main point here is that if the first byte of a 16-bit word is at an even address, the 8086 can read the entire word in one operation.

The Intel 8088 has the same ALU, the same registers, and the same instruction set as the 8086. The 8088 also has a 20-bit address bus, so it can address any one of 1,048,576 bytes in memory. The 8088, however, has an 8-bit data bus, so it can only read data from or write data to memory and ports, 8 bits at a time. The 8086, remember, can read or write either 8 or 16 bits at a time. To read a 16-bit word from two successive memory locations, the 8088 will always have to do two read operations. Since the 8086 and the 8088 are almost identical, any reference we make to the 8086 in the rest of the book will also pertain to the 8088 unless we specifically indicate otherwise. This is done to make reading easier. The Intel 8088, incidentally, is used as the CPU in the original IBM Personal Computer, the IBM PC/XT, and several compatible personal computers.

The Intel 80186 is an improved version of the 8086, and the 80188 is an improved version of the 8088. In addition to a 16-bit CPU, the 80186 and 80188 each have programmable peripheral devices integrated in the same package. In a later chapter we will discuss these integrated peripherals. The instruction set of the 80186 and 80188 is a *superset* of the instruction set of the 8086. The term *superset* means that all the 8086 and 8088 instructions will execute properly on an 80186 or an 80188, but the 80186 and the 80188 have a few additional instructions. In other words, a program written for an 8086 or an 8088 is *upward-compatible* to an 80186 or an 80188, but a program written for an 80186 or an 80188 may not execute correctly on an 8086 or an 8088. In the instruction set descriptions in Chapter 6, we specifically indicate which instructions work only with the 80186 or 80188.

The Intel 80286 is a 16 bit, advanced version of the 8086 which was specifically designed for use as the CPU in a multiuser or multitasking microcomputer. When operating in its *real address mode*, the 80286 functions mostly as a fast 8086. Most programs written for an 8086 can be run on an 80286 operating in its real address mode. When operating in its *virtual address mode*, an 80286 has features which make it easy to keep users' programs separate from one another and to protect the system program from destruction by users' programs. In Chapter 15, we discuss the operation and use of the 80286. The 80286 is the CPU used in the IBM PC/AT personal computer.

With the 80386 processor, Intel started the 32-bit processor architecture, known as the IA-32 architecture. This architecture extended all the address and general purpose registers to 32-bits, which gave the processor the capability to handle 32-bit address (4 GB of memory addressing), with 32-bit data, and yet accommodating all the software designed for the earlier 16-bit processors, 8086, 8088, 80186, 80188 and 80286. It contains more sophisticated features for use in multiuser and multitasking environments.

Intel 80486 is the next member of the IA-32 architecture. This processor has the floating point processor (80387) integrated into the CPU chip itself. These processors are then followed by different versions of the Pentium Processors, with different additional capabilities such as multimedia (MMX, SSE, SSE2 etc.), system power saving modes, hyper thread technology etc.

In Chapter 15, we will discuss the 80286, 386, and 486 processors, and in Chapter 16, we shall take up some major versions of the Pentium processors for discussion.

8086 INTERNAL ARCHITECTURE

Before we can talk about how to write programs for the 8086, we need to discuss its specific internal features, such as its ALU, flags, registers, instruction byte queue, and segment registers.

As shown by the block diagram in Fig. 2.7, the 8086 CPU is divided into two independent functional parts, the *bus interface unit* or BIU, and the *execution unit* or EU. Dividing the work between these two units speeds up processing.

The BIU sends out addresses, fetches instructions from memory, reads data from ports and memory, and writes data to ports and memory. In other words, the BIU handles all transfers of data and addresses on the buses for the execution unit.

The execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions. Let's take a look at some of the parts of the execution unit.

The Execution Unit

CONTROL CIRCUITRY, INSTRUCTION DECODER, AND ALU

As shown in Fig. 2.7, the EU contains *control circuitry* which directs internal operations. A *decoder* in the EU translates instructions fetched from memory into a series of actions which the EU carries out. The EU has a 16-bit *arithmetic logic unit* which can add, subtract, AND, OR, XOR, increment, decrement, complement, or shift binary numbers.

FLAG REGISTER

A *flag* is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU. A 16-bit *flag register* in the EU contains nine active flags. Fig. 2.8 shows the location of the nine flags in the flag register. Six of the nine flags are used to indicate some *condition* produced by an instruction. For example, a flip-flop called the *carry flag* will be set to a 1 if the addition of two 16-bit binary

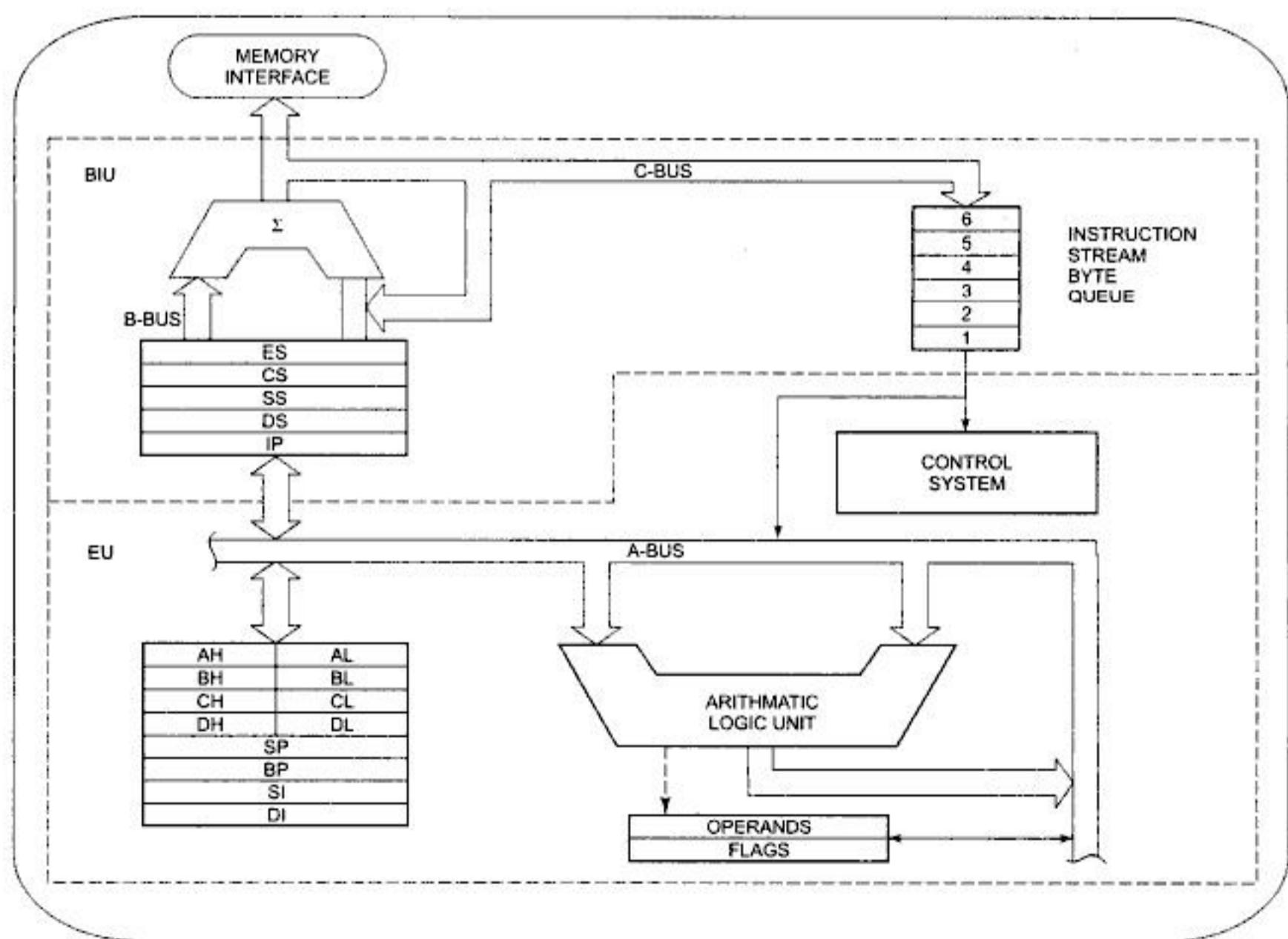


Fig. 2.7 8086 internal block diagram. (Intel Corp.)

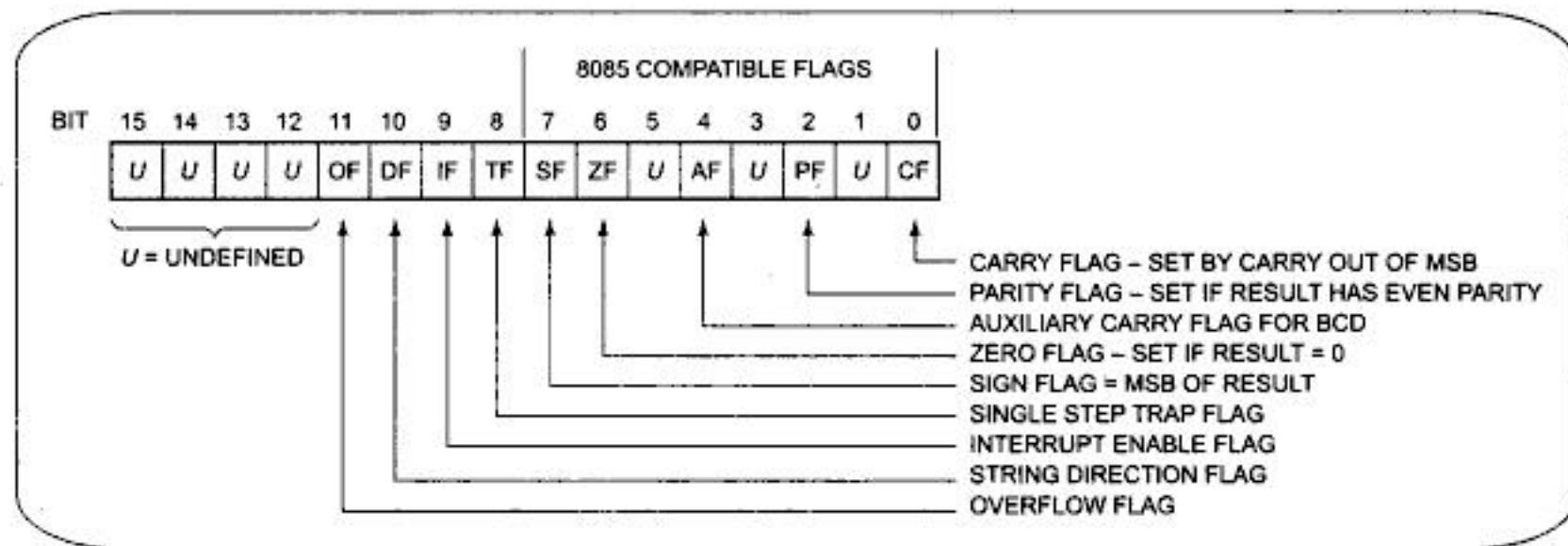


Fig. 2.8 8086 flag register format. (Intel Corp.)

numbers produces a carry out of the most significant bit position. If no carry out of the MSB is produced by the addition, then the carry flag will be a 0. The EU, thus effectively runs up a "flag" to tell you that a carry was produced.

The six conditional flags in this group are the *carry flag* (CF), the *parity flag* (PF), the *auxiliary carry flag* (AF), the *zero flag* (ZF), the *sign flag* (SF), and the *overflow flag* (OF). The names of these flags should give you hints as to what conditions affect them. Certain 8086 instructions check these flags to determine which of two alternative actions should be done in executing the instruction.

The three remaining flags in the flag register are used to *control* certain operations of the processor. These flags are different from the six conditional flags described above in the way they are set or reset. The six conditional flags are set or reset by the EU on the basis of the results of some arithmetic or logic operation. The *control flags* are deliberately set or reset with specific instructions you put in your program. The three control flags are the *trap flag* (TF), which is used for single stepping through a program; the *interrupt flag* (IF), which is used to allow or prohibit the interruption of a program; and the *direction flag* (DF), which is used with string instructions.

Later we will discuss in detail the operation and use of the nine flags.

GENERAL-PURPOSE REGISTERS

Observe in Fig. 2.7 that the EU has eight *general-purpose registers*, labeled AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually for temporary storage of 8-bit data. The AL register is also called the *accumulator*. It has some features that the other general-purpose registers do not have.

Certain pairs of these general-purpose registers can be used together to store 16-bit data words. The acceptable register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. The AH–AL pair is referred to as the *AX register*, the BH–BL pair is referred to as the *BX register*, the CH–CL pair is referred to as the *CX register*, and the DH–DL pair is referred to as the *DX register*.

The 8086 general-purpose register set is very similar to those of the earlier generation 8080 and 8085 microprocessors. It was designed this way so that the many programs written for the 8080 and 8085 could easily be translated to run on the 8086 or the 8088. The advantage of using internal registers for the temporary storage of data is that, since the data is already in the EU, it can be accessed much more quickly than it could be accessed in external memory. Now let's look at the features of the BIU.

The BIU

THE QUEUE

While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instruction bytes for the following instructions. The BIU stores these prefetched bytes in a first-in-first-out register set called a *queue*. When the EU is ready for its next instruction, it simply reads the instruction byte(s) for the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes. The process is analogous to the way a bricklayer's assistant fetches bricks ahead of time and keeps a queue of bricks lined up so that the bricklayer can just reach out and grab a brick when necessary. Except in the cases of JMP and CALL,

instructions, where the queue must be dumped and then reloaded starting from a new address, this prefetch-and-queue scheme greatly speeds up processing. Fetching the next instruction while the current instruction executes is called *pipelining*.

SEGMENT REGISTERS

The 8086 BIU sends out 20-bit addresses, so it can address any of 2^{20} or 1,048,576 bytes in memory. However, at any given time the 8086 works with only four 65,536-byte (64-Kbyte) segments within this 1,048,576-byte (1-Mbyte) range. Four *segment registers* in the BIU are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time. The four segment registers are the *code segment* (CS) register, the *stack segment* (SS) register, the *extra segment* (ES) register, and the *data segment* (DS) register.

Figure 2.9 shows how these four segments might be positioned in memory at a given time. The four segments can be separated as shown, or, for small programs which do not need all 64 Kbytes in each segment, they can overlap.

To repeat, then, a segment register is used to hold the upper 16 bits of the starting address for each of the segments. The code segment register, for example, holds the upper 16 bits of the starting address for the segment from which the BIU is currently fetching instruction code bytes. The BIU always inserts zeros for the lowest 4 bits (nibble) of the 20-bit starting address for a segment. If the code segment register contains 348AH, for example, then the code segment will start at address 348A0H. In other words, a 64-Kbyte segment can be located anywhere within the 1-Mbyte address space, but the segment will always start at an address with zeros in the lowest 4 bits. This constraint was put on the location of segments so that it is only necessary to store and manipulate 16-bit numbers when working with the starting address of a segment. The part of a segment starting address stored in a segment register is often called the *segment base*.

A *stack* is a section of memory set aside to store addresses and data while a *subprogram* executes. The stack segment register is used to hold the upper 16 bits of the starting address for the program stack. We will discuss the use and operation of a stack in detail later.

The extra segment register and the data segment register are used to hold the upper 16 bits of the starting addresses of two memory segments that are used for data.

INSTRUCTION POINTER

The next feature to look at in the BIU is the *instruction pointer* (IP) register. As discussed previously, the code

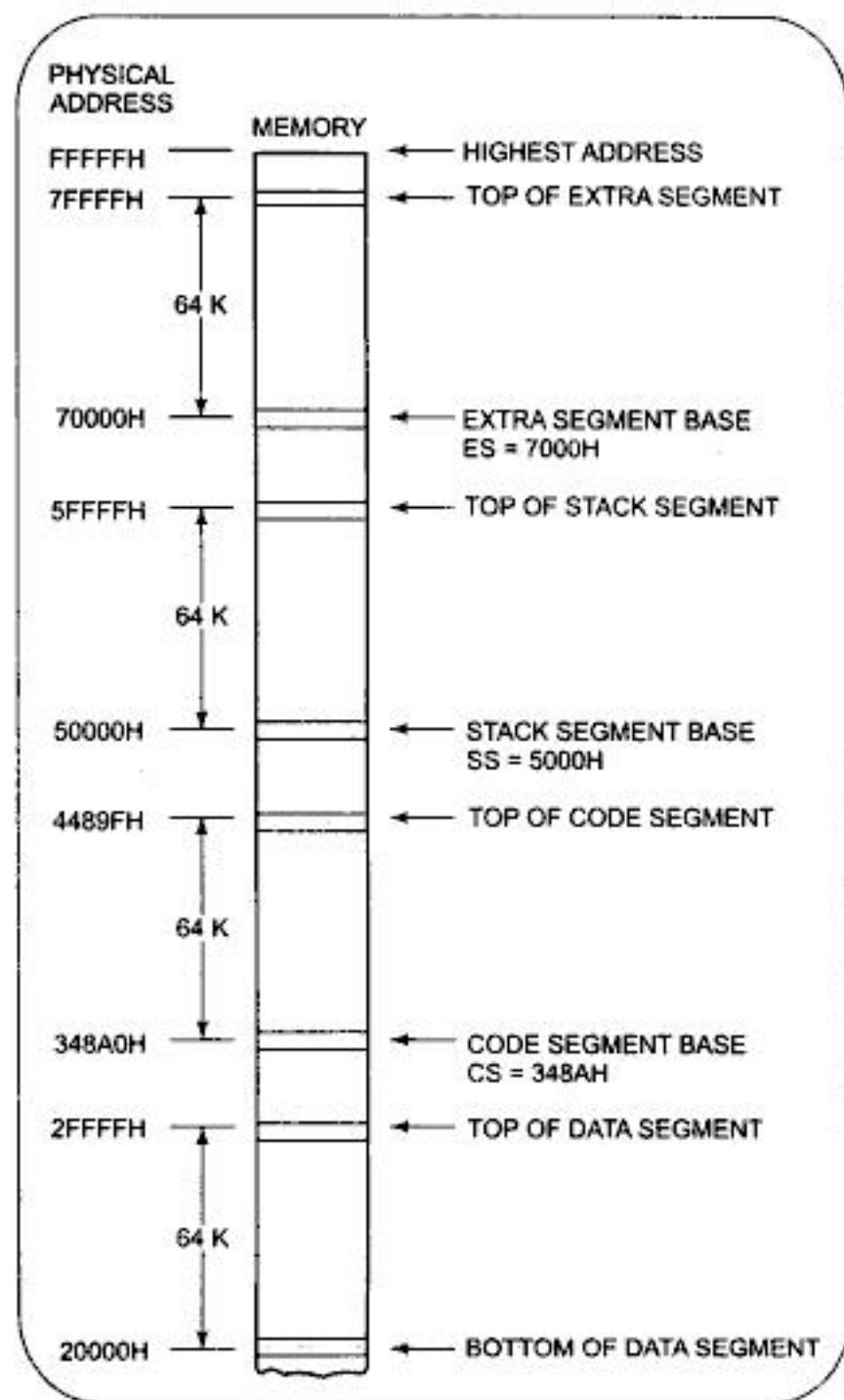


Fig. 2.9 One way four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086.

segment register holds the upper 16 bits of the starting address of the segment from which the BIU is currently fetching instruction code bytes. The instruction pointer register holds the 16-bit address, or *offset*, of the next code byte *within* this code segment. The value contained in the IP is referred to as an *offset* because this value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address sent out by the BIU. Fig. 2.10a shows in diagram form how this works. The CS register points to the *base* or start of the current code segment. The IP contains the distance or offset from this base address to the next instruction byte to be fetched. Fig. 2.10b shows how the 16-bit offset in IP is added to the 16-bit segment base address in CS to produce the 20-bit *physical address*. Notice that the two

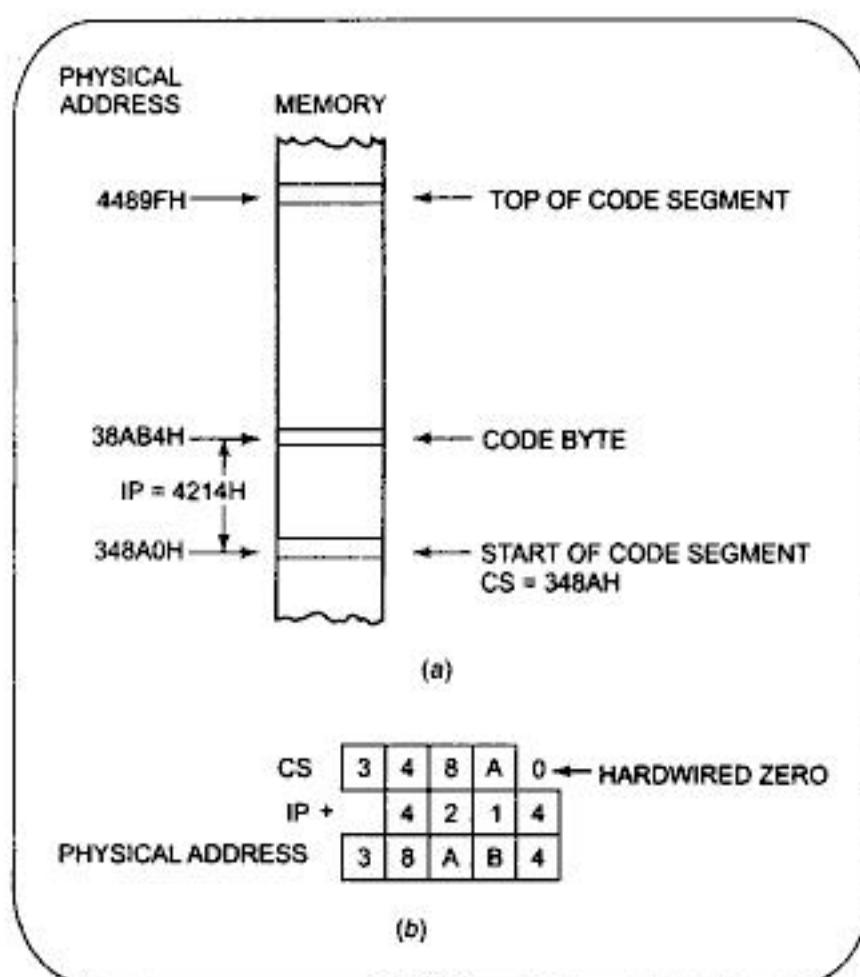


Fig. 2.10 Addition of IP to CS to produce the physical address of the code byte, (a) Diagram, (b) Computation.

16-bit numbers are not added directly in line, because the CS register contains only the upper 16 bits of the base address for the code segment. As we said before, the BIU automatically inserts zeros for the lowest 4 bits of the segment base address.

If the CS register, for example, contains 348AH, you know that the starting address for the code segment is 348A0H. When the BIU adds the offset of 4214H in the IP to this segment base address, the result is a 20-bit physical address of 38AB4H.

An alternative way of representing a 20-bit physical address is the *segment base:offset form*. For the address of a code byte, the format for this alternative form will be CS:IP. As an example of this, the address constructed in the preceding paragraph, 38AB4H, can also be represented as 348A:4214.

To summarize, then, the CS register contains the upper 16 bits of the starting address of the code segment in the 1-Mbyte address range of the 8086. The instruction pointer register contains a 16-bit offset which tells, where in that 64-Kbyte code segment the next instruction byte is to be fetched from. The actual physical address sent to memory is produced by adding the offset contained in the IP register to the segment base represented by the upper 16 bits in the CS register.

Any time the 8086 accesses memory, the BIU produces the required 20-bit physical address by adding an offset to a segment base value represented by the contents of one of the segment registers. As another example of this, let's look at how the 8086 uses the contents of the stack segment register and the contents of the stack pointer register to produce a physical address.

STACK SEGMENT REGISTER AND STACK POINTER REGISTER

A stack, remember, is a section of memory set aside to store addresses and data while a subprogram is executing. The 8086 allows you to set aside an entire 64-Kbyte segment as a stack. The upper 16 bits of the starting address for this segment are kept in the stack segment register. The *stack pointer* (SP) register in the execution unit holds the 16-bit offset from the start of the segment to the memory location where a word was most recently stored on the stack. The memory location where a word was most recently stored is called the *top of stack*. Fig. 2.11a shows this in diagram form.

The physical address for a stack read or a stack write is produced by adding the contents of the stack pointer register to the segment base address represented by the upper 16 bits of the base address in SS. Fig. 2.11b shows

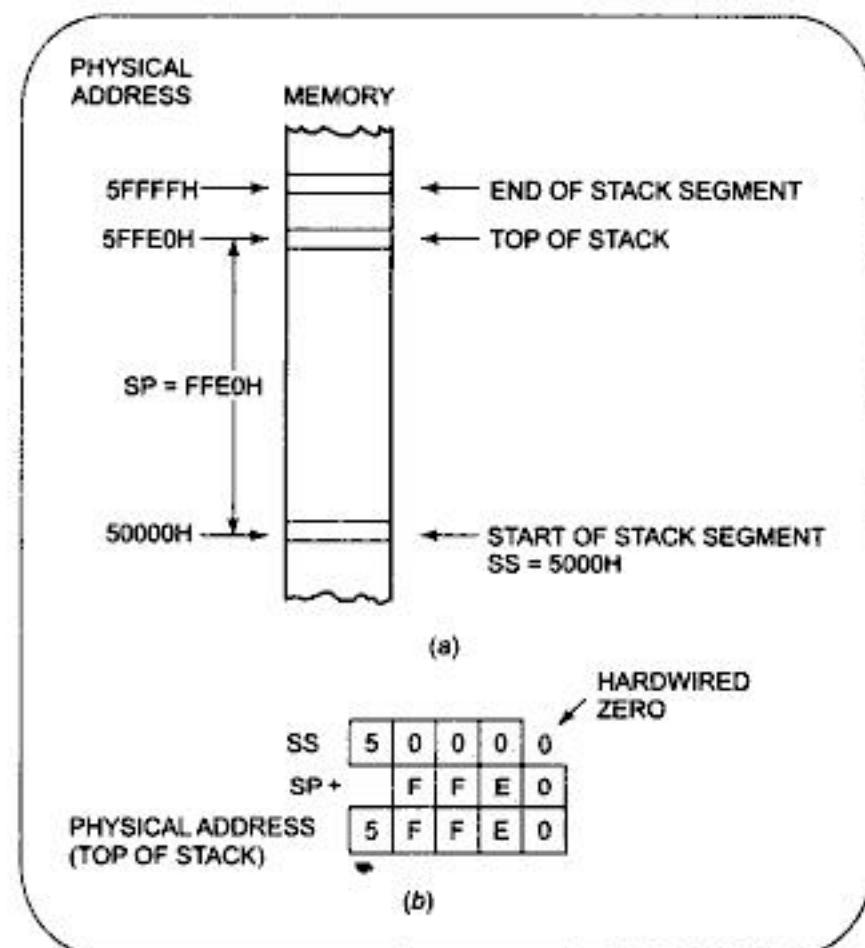


Fig. 2.11 Addition of SS and SP to produce the physical address of the top of the stack, (a) Diagram, (b) Computation.

an example. The 5000H in SS represents a segment base address of 50000H. When the FFE0H in the SP is added to this, the resultant physical address for the top of the stack will be 5FFE0H. The physical address can be represented either as a single number, 5FFE0H, or in SS:SP form as 5000:FFE0H.

The operation and use of the stack will be discussed in detail later as need arises.

POINTER AND INDEX REGISTERS IN THE EXECUTION UNIT

In addition to the stack pointer register (SP), the EU contains a 16-bit *base pointer* (BP) register. It also contains a 16-bit *source index* (SI) register and a 16-bit *destination index* (DI) register. These three registers can be used for temporary storage of data just as the general-purpose registers described above. However, their main use is to hold the 16-bit offset of a data word in one of the segments. SI, for example, can be used to hold the offset of a data word in the data segment. The physical address of the data in memory will be generated in this case by adding the contents of SI to the segment base address represented by the 16-bit number in the DS register. After we give you an overview of the different levels of languages used to program a microcomputer, we will show you some examples of how we tell the 8086 to read data from or write data to a desired memory location.

INTRODUCTION TO PROGRAMMING THE 8086

Programming Languages

Now that you have an overview of the 8086 CPU, it is time to start thinking about how it is programmed. To run a program, a microcomputer must have the program stored in binary form in successive memory locations, as shown in Fig. 2.12. There are three language levels that can be used to write a program for a microcomputer.

MACHINE LANGUAGE

You can write programs as simply a sequence of the binary codes for the instructions you want the microcomputer to execute. The three-instruction program in Fig. 2.6b is an example. This binary form of the program is referred to as *machine language* because it is the form required by the machine. However, it is difficult, if not impossible, for a programmer to memorize the thousands of binary instruction codes for a CPU such as the 8086. Also, it is very easy for an error to occur when working with long

series of 1's and 0's. Using hexadecimal representation for the binary codes might help some, but there are still thousands of instruction codes to cope with.

| LABEL FIELD | OPCODE FIELD | OPERAND FIELD | COMMENT FIELD |
|-------------|--------------|---------------|------------------------|
| NEXT: | ADD | AL, 07H | ;ADD CORRECTION FACTOR |

Fig. 2.12 Assembly language program statement format.

ASSEMBLY LANGUAGE

To make programming easier, many programmers write programs in *assembly language*. They then translate the assembly language program to machine language so that it can be loaded into memory and run. Assembly language uses two-, three-, or four-letter *mnemonics* to represent each instruction type. A mnemonic is just a device to help you remember something. The letters in an assembly language mnemonic are usually initials or a shortened form of the English word(s) for the operation performed by the instruction. For example, the mnemonic for subtract is SUB, the mnemonic for Exclusive OR is XOR, and the mnemonic for the instruction to copy data from one location to another is MOV.

Assembly language statements are usually written in a standard form that has *four fields*, as shown in Fig. 2.12. The first field in an assembly language statement is the *label field*. A *label* is a symbol or group of symbols used to represent an address which is not specifically known at the time the statement is written. Labels are usually followed by a colon. Labels are not required in a statement, they are just inserted where they are needed. We will show later many uses of labels.

The *opcode field* of the instruction contains the mnemonic for the instruction to be performed. Instruction mnemonics are sometimes called *operation codes*, or *opcodes*. The ADD mnemonic in the example statement in Fig. 2.12 indicates that we want the instruction to do an addition.

The *operand field* of the statement contains the data, the memory address, the port address, or the name of the register on which the instruction is to be performed. *Operand* is just another name for the data item(s) acted on by an instruction. In the example instruction in Fig. 2.12, there are two operands, AL and 07H, specified in the operand field. AL represents the AL register, and 07H represents the number 07H. This assembly language statement thus says, "Add the number 07H to the contents

of the AL register." By Intel convention, the result of the addition will be put in the register or the memory location specified *before* the comma in the operand field. For the example statement in Fig. 2.12, then, the result will be left in the AL register. As another example, the assembly language statement ADD BH, AL, when converted to machine language and run, will add the contents of the AL register to the contents of the BH register. The results will be left in the BH register.

The final field in an assembly language statement such as that in Fig. 2.12 is the *commentfield*, which starts with a semicolon. Comments do not become part of the machine language program, but they are very important. You write *comments* in a program to remind you of the function that an instruction or group of instructions performs in the program.

To summarize why assembly language is easier to use than machine language, let's look a little more closely at the assembly language ADD statement. The general format of the 8086 ADD instruction is

ADD Destination, Source

The *source* can be a number written in the instruction, the contents of a specified register, or the contents of a memory location. The *destination* can be a specified register or a specified memory location. However, the source and the destination in an instruction cannot both be memory locations.

A later section on 8086 addressing modes will show all the ways in which the source of an operand and the destination of the result can be specified. The point here is that the single mnemonic ADD, together with a specified source and a specified destination, can represent a great many 8086 instructions in an easily understandable form.

The question that may occur to you at this point is, "If I write a program in assembly language, how do I get it translated into machine language which can be loaded into the microcomputer and executed?" There are two answers to this question. The first method of doing the translation is to work out the binary code for each instruction, a bit at a time using the templates given in the manufacturer's data books. We will show you how to do this in the next chapter, but it is a tedious and error-prone task. The second method of doing the translation is with an assembler. An assembler is a program which can be run on a personal computer or *microcomputer development system*. It reads the file of assembly language instructions you write and generates the correct binary code for each. For developing all but the simplest assembly language programs, an assembler and other program development tools are essential. We will introduce you to these program development tools in the next chapter and describe their use throughout the rest of this book.

HIGH-LEVEL LANGUAGES

Another way of writing a program for a microcomputer is with a *high-level language*, such as BASIC, Pascal, or C. These languages use program statements which are even more English-like than those of assembly language. Each high-level statement may represent many machine code instructions. An *interpreter program* or a *compiler program* is used to translate higher-level language statements to machine codes which can be loaded into memory and executed. Programs can usually be written faster in high-level languages than in assembly language because a high-level language works with bigger building blocks. However, programs written in a high-level language and interpreted or compiled almost always execute more slowly and require more memory than the same programs written in assembly language. Programs that involve a lot of hardware control, such as robots and factory control systems, or programs that must run as quickly as possible are usually best written in assembly language. Complex data processing programs that manipulate massive amounts of data, such as insurance company records, are usually best written in a high-level language. The decision concerning which language to use has recently been made more difficult by the fact that current assemblers allow the use of many high-level language features, and the fact that some current high-level languages provide assembly language features.

OUR CHOICE

For most of this book we work very closely with hardware, so assembly language is the best choice. In later chapters, however, we do show you how to write programs which contain modules written in assembly language and modules written in the high-level language C. In the next chapter we introduce you to assembly language programming techniques. Before we go on to that, however, we will use a few simple 8086 instructions to show you more about accessing data in registers and memory locations.

How the 8086 Accesses Immediate and Register Data

In a previous discussion of the 8086 BIU, we described how the 8086 accesses code bytes using the contents of the CS and IP registers. We also described how the 8086 accesses the stack using the contents of the SS and SP registers. Before we can teach you assembly language programming techniques, we need to discuss some of the different ways in which an 8086 can access the data that it operates on. The different ways in which a processor can access data are referred to as its *addressing modes*.

In assembly language statements, the addressing mode is indicated in the instruction. We will use the 8086 MOV instruction to illustrate some of the 8086 addressing modes.

The MOV instruction has the format

MOV Destination, Source

When executed, this instruction *copies* a word or a byte from the specified source location to the specified destination location. The source can be a number written directly in the instruction, a specified register, or a memory location specified in 1 of 24 different ways. The destination can be a specified register or a memory location specified in any 1 of 24 different ways. The source and the destination cannot both be memory locations in an instruction.

IMMEDIATE ADDRESSING MODE

Suppose that in a program you need to put the number 437BH in the CX register. The MOV CX, 437BH instruction can be used to do this. When it executes, this instruction will put the *immediate* hexadecimal number 437BH in the 16-bit CX register. This is referred to as *immediate addressing mode* because the number to be loaded into the CX register will be put in the two memory locations immediately following the code for the MOV instruction. This is similar to the way the port address was put in memory immediately after the code for the input instruction in the three-instruction program in Fig. 2.6b.

A similar instruction, MOV CL, 48H, could be used to load the 8-bit immediate number 48H into the 8-bit CL register. You can also write instructions to load an 8-bit immediate number into an 8-bit memory location or to load a 16-bit number into two consecutive memory locations, but we are not yet ready to show you how to specify these.

REGISTER ADDRESSING MODE

Register addressing mode means that a register is the source of an operand for an instruction. The instruction MOV CX, AX, for example, copies the contents of the 16-bit AX register into the 16-bit CX register. Remember that the destination location is specified in the instruction before the comma, and the source is specified after the comma. Also note that the contents of AX are just *copied* to CX, not actually moved. In other words, the previous contents of CX are written over, but the contents of AX are not changed. For example, if CX contains 2A84H and AX contains 4971H before the MOV CX, AX instruction executes, then after the instruction executes, CX will contain 4971H and AX will still contain 4971H. You can MOV any 16-bit register to any 16-bit register, or you can MOV any 8-bit register to any 8-bit register. However, you

cannot use an instruction such as MOV CX, AL because this is an attempt to copy a *byte-type* operand (AL) into a *word-type* destination (CX). The byte in AL would fit in CX, but the 8086 would not know which half of CX to put it in. If you try to write an instruction like this and you are using a good assembler, the assembler will tell you that the instruction contains a *type error*. To copy the byte from AL to the high byte of CX, you can use the instruction MOV CH, AL. To copy the byte from AL to the low byte of CX, you can use the instruction MOV CL, AL.

Accessing Data in Memory

OVERVIEW OF MEMORY ADDRESSING MODES

The addressing modes described in the following sections are used to specify the location of an operand in memory. To access data in memory, the 8086 must also produce a 20-bit physical address. It does this by adding a 16-bit value called the *effective address* to a segment base address represented by the 16-bit number in one of the four segment registers. The effective address (EA) represents the *displacement* or *offset* of the desired operand from the segment base. In most cases, any of the segment bases can be specified, but the data segment

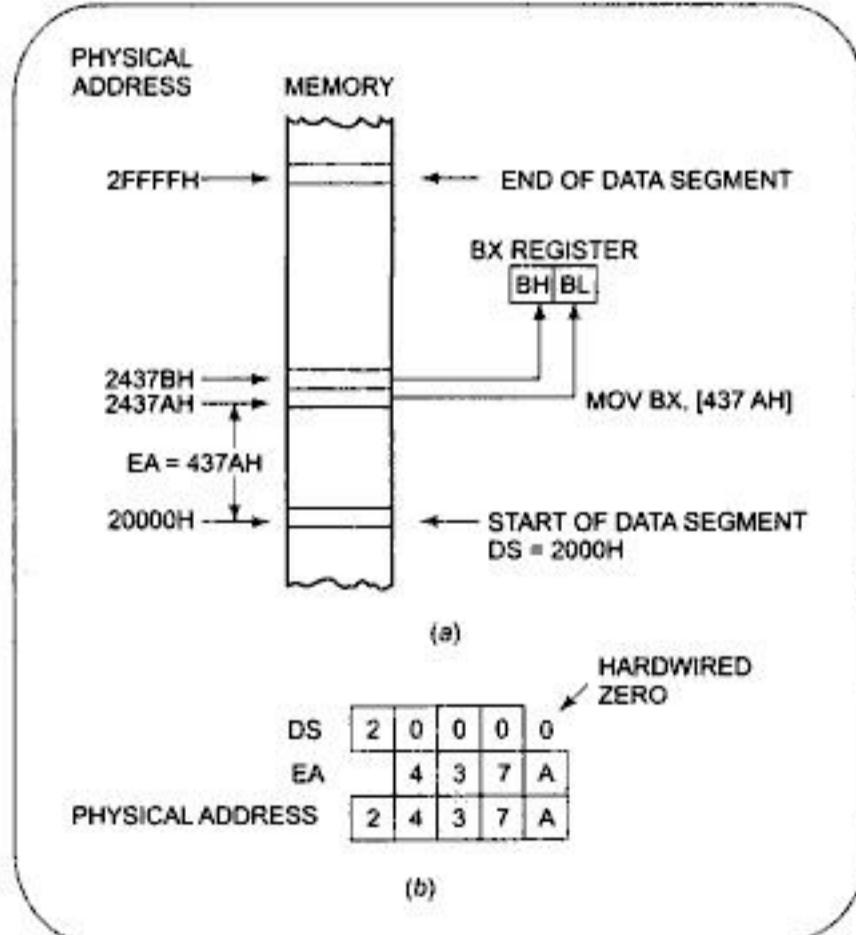


Fig. 2.13 Addition of data segment register and effective address to produce the physical address of the data byte. (a) Diagram. (b) Computation.

is the one most often used. Fig. 2.13a shows in graphic form how the EA is added to the data segment base to point to an operand in memory. Fig. 2.13b shows how the 20-bit physical address is generated by the BIU. The starting address for the data segment in Fig. 2.13b is 20000H, so the data segment register will contain 2000H. The BIU adds the effective address, 437AH, to the data segment base address of 20000H to produce the physical address sent out to memory. The 20-bit physical address sent out to memory by the BIU will then be 2437AH. The physical address can be represented either as a single number 2437AH or in the segment base:offset form as 2000:437AH.

The execution unit calculates the effective address for an operand using information you specify in the instruction. You can tell the EU to use a number in the instruction as the effective address, to use the contents of a specified register as the effective address, or to compute the effective address by adding a number in the instruction to the contents of one or two specified registers. The following section describes one way you can tell the execution unit to calculate an effective address. In later chapters we show other ways of specifying the effective address. Later we also show how the addressing modes this provides, are used to solve some common programming problems.

DIRECT ADDRESSING MODE

For the simplest memory addressing mode, the effective address is just a 16-bit number written directly in the instruction. The instruction `MOV BL, [437AH]` is an example. The square brackets around the 437AH are shorthand for “the contents of the memory location(s) at a displacement from the segment base of.” When executed, this instruction will copy “the contents of the memory location at a displacement from the data segment base of” 437AH into the BL register, as shown by the rightmost arrow in Fig. 2.13a. The BIU calculates the 20-bit physical memory address by adding the effective address 437AH to the data segment base, as shown in Fig. 2.13b. This addressing mode is called *direct* because the displacement of the operand from the segment base is specified directly in the instruction. The displacement in the instruction will be added to the data segment base in DS unless you tell the BIU to add it to some other segment base. Later we will show you how to do this.

Another example of the direct addressing mode is the instruction `MOV BX, [437AH]`. When executed, this instruction copies a 16-bit word from memory into the

BX register. Since each memory address of the 8086 represents a byte of storage, the word must come from two memory locations. The byte at a displacement of 437AH from the data segment base will be copied into BL, as shown by the right arrow in Fig. 2.13a. The contents of the next higher address, displacement 437BH, will be copied into the BH register, as shown by the left arrow in Fig. 2.13a. From the instruction coding, the 8086 will automatically determine the number of bytes that it must access in memory.

An important point here is that an 8086 always stores the low byte of a word in the lower of the two addresses and stores the high byte of a word in the higher address. To stick this in your mind, remember:

Low byte—Low address, High byte—High address

The previous two examples showed how the direct addressing mode can be used to specify the source of an operand. Direct addressing can also be used to specify the destination of an operand in memory. The instruction `MOV [437AH], BX`, for example, will copy the contents of the BX register to two memory locations in the data segment. The contents of BL will be copied to the memory location at a displacement of 437AH. The contents of BH will be copied to the memory location at a displacement of 437BH. This operation is represented by simply reversing the direction of the arrows in Fig. 2.13a.

NOTE: When you are hand-coding programs using direct addressing of the form shown above, make sure to put in the square brackets to remind you how to code the instruction. If you leave the brackets out of an instruction such as `MOV BX, [437AH]`, you will code it as if it were the instruction `MOV BX, 437AH`. This second instruction will load the immediate number 437AH into BX, rather than loading a word from memory at a displacement of 437AH into BX. Also note that if you are writing an instruction using direct addressing such as this for an assembler, you must write the instruction in the form `MOV BL, DS:BYTE PTR [437AH]` to give the assembler all the information it needs. As we will show you in the next chapter, when you are using an assembler, you usually use a name to represent the direct address rather than the actual numerical value.

A FEW WORDS ABOUT SEGMENTATION

At this point you may be wondering why Intel designed the 8086 family devices to access memory using the segment:offset approach rather than accessing memory directly with 20-bit addresses. The segment:offset scheme requires only a 16-bit number to represent the base address

for a segment, and only a 16-bit offset to access any location in a segment. This means that the 8086 has to manipulate and store only 16-bit quantities instead of 20-bit quantities. This makes for an easier interface with 8- and 16-bit-wide memory boards and with the 16-bit registers in the 8086.

The second reason for segmentation has to do with the type of microcomputer in which an 8086-family CPU is likely to be used. A previous section of this chapter described briefly the operation of a timesharing microcomputer system. In a timesharing system, several users share a CPU. The CPU works on one user's program for perhaps 20 ms, then works on the next user's program for 20 ms. After working 20 ms for each of the other users, the CPU comes back to the first user's program again. Each time the CPU switches from one user's program to the next, it must access a new section of code and new sections of data. Segmentation makes this switching quite easy. Each user's program can be assigned a separate set of logical segments for its code and data. The user's program will contain offsets or displacements from these segment bases. To change from one user's program to a second user's program, all that the CPU has to do is to reload the four segment registers with the segment base addresses assigned to the second user's program. In other words, segmentation makes it easy to keep users' programs and data separate from one another, and segmentation makes it easy to switch from one user's program to another user's program. In Chapter 15 we tell

you much more about the use of segmentation in multiuser systems.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms or concepts in the following list, use the index to find them in the chapter.

Microcomputer, microprocessor

Hardware, software, firmware

Timesharing computer system

Multitasking computer system

Distributed processing system

Multiprocessing

CPU

Memory, RAM, ROM

I/O ports

Address, data, and control buses

Control bus signals

ALU

Segmentation

Bus interface unit (BIU)

Instruction byte queue, pipelining, ES, CS, SS, DS registers, IP register

Execution unit (EU) AX, BX, CX, DX registers, flag register, ALU, SP, BP, SI, DI registers

Machine language, assembly language, high-level language

Mnemonic, opcode, operand, label, comment Assembler, compiler

Immediate address mode, register address mode, direct address mode

Effective address

REVIEW QUESTIONS AND PROBLEMS

1. Describe the main advantages of a distributed processing computer system over a simple timesharing system.
2. Describe the sequence of signals that occurs on the address bus, the control bus, and the data bus when a simple microcomputer fetches an instruction.
3. What determines whether a microprocessor is considered an 8-bit, a 16-bit, or a 32-bit device?
4.
 - a. How many address lines does an 8086 have?
 - b. How many memory addresses does this number of address lines allow the 8086 to access directly?
 - c. At any given time, the 8086 works with four segments in this address space. How many bytes are contained in each segment?
5. What is the main difference between the 8086 and the 8088?
6.
 - a. Describe the function of the 8086 queue.
 - b. How does the queue speed up processing?
7.
 - a. If the code segment for an 8086 program starts at address 70400H, what number will be in the CS register?
 - b. Assuming this same code segment base, what physical address will a code byte be fetched from if the instruction pointer contains 539CH?
8. What physical address is represented by:
 - a. 4370:561EH
 - b. 7A32:0028H
9. What is the advantage of using a CPU register for temporary data storage over using a memory location?
10. If the stack segment register contains 3000H and the stack pointer register contains 8434H, what is the physical address of the top of the stack?

- 11.**
 - a.** What is the advantage of using assembly language instead of writing a program directly in machine language?
 - b.** Describe the operation an 8086 will perform when it executes ADD AX, BX.
- 12.** What types of programs are usually written in assembly language?
- 13.** Describe the operation that an 8086 will perform when it executes each of the following instructions:
 - a.** MOV BX, 03FFH
 - b.** MOVAL, 0DBH
 - c.** MOV DH, CL
 - d.** MOVBX, AX
- 14.** Write the 8086 assembly language statement which will perform the following operations:
 - a.** Load the number 7986H into the BP register.
 - b.** Copy the BP register contents to the SP register.
 - c.** Copy the contents of the AX register to the DS register.
 - d.** Load the number F3H into the AL register.
- 15.** If the 8086 execution unit calculates an effective address of 14A3H and DS contains 7000H, what physical address will the BIU produce?
- 16.** If the data segment register (DS) contains 4000H, what physical address will the instruction MOV AL, [234BH] read?
- 17.** If the 8086 data segment register contains 7000H, write the instruction that will copy the contents of DL to address 74B2CH.
- 18.** Describe the difference between the instructions MOV AX, 2437H and MOV AX, [2437H].

8086 Family Assembly Language Programming—Introduction

The last chapter showed you the format for assembly language instructions and introduced you to a few 8086 instructions. Developing a program, however, requires more than just writing down a series of instructions. When you want to build a house, it is a good idea to first develop a complete set of plans for the house. From the plans you can see whether the house has the rooms you need, whether the rooms are efficiently placed, and whether the house is structured so that you can easily add on to it if you have more kids. You have probably seen examples of what happens when someone attempts to build a house by just putting pieces together without a plan.

Likewise, when you write a computer program, it is a good idea to start by developing a detailed plan or outline for the entire program. A good outline helps you to break down a large and seemingly overwhelming programming job into small modules which can easily be written, tested, and debugged. The more time you spend organizing your programs, the less time it will take you to write and debug them. You should *never* start writing an assembly language program by just writing down instructions! In this chapter we show you how to develop assembly language programs in a systematic way.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Write a task list, flowchart, or pseudocode for a simple programming problem.

2. Write, code or assemble, and run a very simple assembly language program.
3. Describe the use of program development tools such as editors, assemblers, linkers, locators, debuggers, and emulators.
4. Properly document assembly language programs.

PROGRAM DEVELOPMENT STEPS

Defining the Problem

The first step in writing a program is to think very carefully about the problem that you want the program to solve. In other words, ask yourself many times, “What do I really want this program to do?” If you don’t do this, you may write a program that works great but does not do what you need it to do. As you think about the problem, it is a good idea to write down exactly what you want the program to do and the order in which you want the program to do it. At this point you do not write down program statements, you just write the operations you want in general terms. An example for a simple programming problem might be

1. Read temperature from sensor.
2. Add correction factor of + 7.
3. Save result in a memory location.

For a program as simple as this, the three actions desired are very close to the eventual assembly language

statements. For more complex problems, however, we develop a more extensive outline before writing the assembly language statements. The next section shows you some of the common ways of representing program operations in a program outline.

Representing Program Operations

The formula or sequence of operations used to solve a programming problem is often called the *algorithm* of the program. The following sections show you two common ways of representing the algorithm for a program or program segment.

FLOWCHARTS

If you have done any previous programming in BASIC or in FORTRAN, you are probably familiar with *flowcharts*. Flowcharts use graphic shapes to represent different types of program operations. The specific operation desired is written in the graphic symbol. Fig. 3.1 shows some of the common flowchart symbols. Plastic templates are available to help you draw these symbols if you decide to use them for your programs.

Figure 3.2, shows a flowchart for a program to read in 24 data samples from a temperature sensor at 1-hour intervals, add 7 to each, and store each result in a memory location. A racetrack- or circular-shaped symbol labeled START is used to indicate the *beginning* of the program. A parallelogram is used to represent an input or an *output* operation. In the example, we use it to indicate reading data from the temperature sensor. A rectangular box symbol is used to represent *simple operations* other than input and output operations. The box containing "add 7" in Fig. 3.2 is an example.

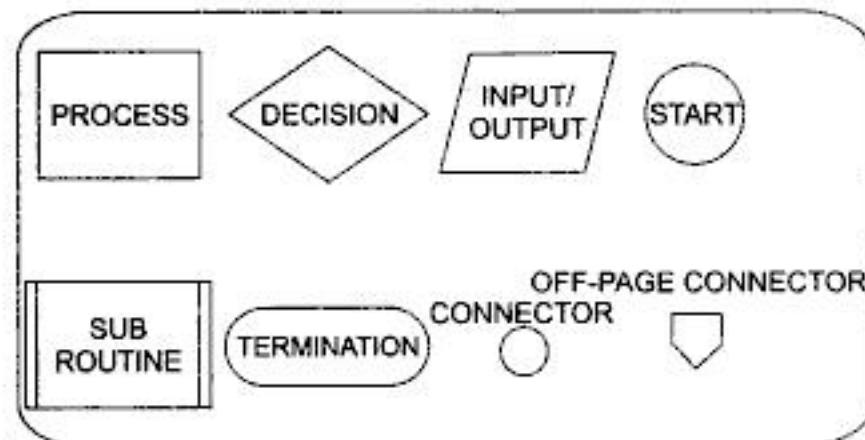


Fig. 3.1 Flowchart symbols.

A rectangular box with double lines at each end is often used to represent a *subroutine* or *procedure* that will be written separately from the main program. When a set of operations must be done several times during a program, it is usually more efficient to write the series of operations

once as a separate subroutine, then just "call" this subroutine each time it is needed. For example, suppose that there are several places in a program where you need to compute the square root of a number. Instead of writing the series of instructions for computing a square root each time you need it in the program, you can write the instruction sequence once as a separate procedure and put it in memory after the main program. A special instruction allows you to call this procedure each time you need to compute a square root. Another special instruction at the end of the procedure program returns execution to the main program. In the flowchart in Fig. 3.2, we use the double-ended box to indicate that the "wait 1 hour" operation will be programmed as a procedure. Incidentally, the terms *subprogram*, *subroutine*, and *procedure* all have the same meaning. Chapter 5 shows how procedures are written and used.

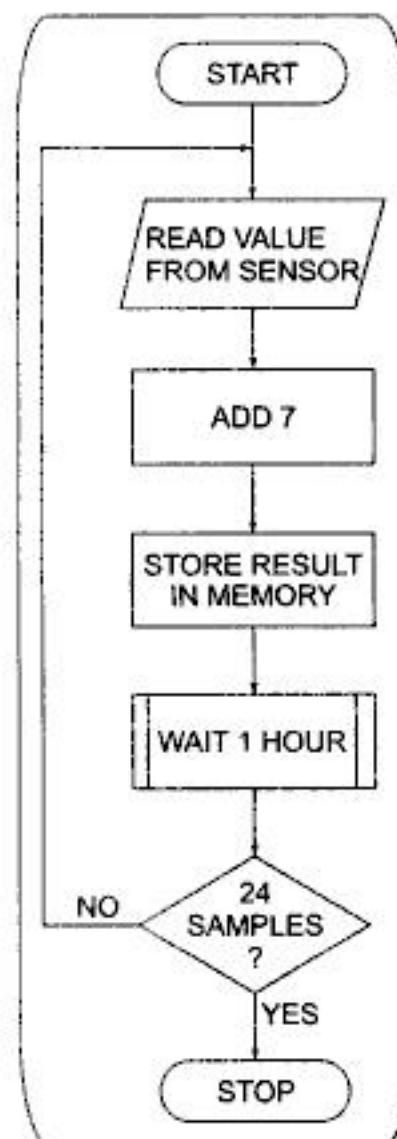


Fig. 3.2 Flowchart for program to read in 24 data samples from a port, correct each value, and store each in a memory location.

A diamond-shaped box is used in flowcharts to represent a *decision point* or crossroad. Usually it indicates that some condition is to be checked at this point in the

program. If the condition is found to be *true*, one set of actions is to be done; if the condition is found to be *false*, another set of actions is to be done. In the example flowchart in Fig. 3.2, the condition to be checked is whether 24 samples have been read in and processed. If 24 samples have not been read in and processed, the arrow labeled NO in the flowchart indicates that we want the computer to jump back and execute the read, add, store, and wait steps again. If 24 samples have been read in, the arrow labeled YES in the flowchart of Fig. 3.2 indicates that all the desired operations have been done. The racetrack-shaped symbol at the bottom of the flowchart indicates the *end* of the program.

The two additional flowchart symbols in Fig. 3.1 are *connectors*. If a flowchart column gets to the bottom of the paper, but not all the program has been represented, you can put a small circle with a letter in it at the bottom of the column. You then start the next column at the top of the same paper with a small circle containing the same letter. If you need to continue a flowchart to another page, you can end the flowchart on the first page with the five-sided off-page connector symbol containing a letter or number. You then start the flowchart on the next page with an off-page connector symbol containing the same letter or number.

For simple programs and program sections, flowcharts are a graphic way of showing the operational flow of the program. We will show flowcharts for many of the program examples throughout this book. Flowcharts, however, have several disadvantages. First, you can't write much information in the little boxes. Second, flowcharts do not present information in a very compact form. For more complex problems, flowcharts tend to spread out over many pages. They are very hard to follow back and forth between pages. Third, and most important, with flowcharts the overall structure of the program tends to get lost in the details. The following section describes a more clearly *structured* and *compact* method of representing the algorithm of a program or program segment.

STRUCTURED PROGRAMMING AND PSEUDOCODE OVERVIEW

In the early days of computers, a single brilliant person might write even a large program single-handedly. The main concerns in this case were, "Does the program work?" and "What do we do if this person leaves the company?" As the number of computers increased and the complexity of the programs being written increased, large programming jobs were usually turned over to a

team of programmers. In this case the compatibility of parts written by different programmers became an important concern. During the 1970s it became obvious to many professional programmers that in order for team programming to work, a systematic approach and standardized tools were absolutely necessary.

One suggested systematic approach is called *top-down design*. In this approach, a large programming problem is first divided into major *modules*. The top level of the outline shows the relationship and function of these modules. This top level then presents a one-page overview of the entire program. Each of the major modules is broken down into still smaller modules on following pages. The division is continued until the steps in each module are clearly understandable. Each programmer can then be assigned a module or set of modules to write for the program. Another advantage of this approach is that people who later want to learn about the program can start with the overview and work their way down to the level of detail they need. This approach is the same as drawing the complete plans for a house before starting to build it.

The opposite of top-down design is *bottom-up design*. In this approach, each programmer starts writing low-level modules and hopes that all the pieces will eventually fit together. When completed, the result should be similar to that produced by the top-down design. Most modern programming teams use a combination of the two techniques. They do the top-down design first, then build, test, and link modules starting from the smallest and working upward.

The development of standard programming methods was helped by the discovery that any desired program operation could be represented by three basic types of operation. The first type of operation is *sequence*, which means simply doing a series of actions. The second basic type of operation is *decision*, or *selection*, which means choosing between two alternative actions. The third basic type of operation is *repetition*, or *iteration*, which means repeating a series of actions until some condition is or is not present.

On the basis of this observation, the suggestion was made that programmers use a set of three to seven standard *structures* to represent all the operations in their programs. Actually, only three structures, SEQUENCE, IF-THEN-ELSE, and WHILE-DO, are required to represent any desired program action, but three or four more structures derived from these often make programs clearer. If you have previously written programs in a structured language such as Pascal, then these structures are probably already familiar to you. Fig. 3.3, uses

flowchart symbols to represent the commonly used structures so that you can more easily visualize their operation. In actual program documentation, however, English-like statements called *pseudocode* are used rather than the space-consuming flowchart symbols. Fig. 3.3 also shows the pseudocode format and an example for each structure.

Each structure has only *one entry point* and *one exit point*. As you will see later, this feature makes debugging the final program much easier. The output of one structure is connected to the input of the next structure. Program execution then proceeds through a series of these structures.

Any structure can be used within another. An IF-THEN-ELSE structure, for example, can contain a sequence of statements. Any place that the term *statement(s)* appears in Fig. 3.3, one of the other structures could be substituted for it. The term *statement(s)* can also represent a subprogram or procedure that is called to do a series of actions. Now, let's look more closely at these structures.

STANDARD PROGRAMMING STRUCTURES

The structure shown in Fig. 3.3a is an example of a simple sequence. In this structure, the actions are simply written down in the desired order. An example is

Read temperature from sensor.

Add correction factor of + 7.

Store corrected value in memory.

Figure 3.3b shows an IF-THEN-ELSE example of the decision operation. This structure is used to direct operation to one of two different actions based on some condition. An example is

IF temperature less than 70 degrees THEN

 Turn on heater

ELSE

 Turn off heater

The example says that if the temperature is below the thermostat setting, we want to turn the heater on. If the temperature is equal to or above the thermostat setting, we want to turn the heater off.

The IF-THEN structure shown in Fig. 3.3c is the same as the IF-THEN-ELSE except that one of the paths contains no action. An example of this is

IF hungry THEN

 Get food

The assumption for this example is that if you are not hungry, you will just continue on with your next task.

To represent a situation in which you want to select one of several actions based on some condition, you can use a nested IF-THEN-ELSE structure such as that shown in Fig. 3.3d. This everyday example describes the thinking a soup cook might go through. Note that in this example the last IF-THEN has no ELSE after it because all the possible days have been checked. You can, if you want, add the final ELSE to the IF-THEN-ELSE chain to send an error message if the data does not match any of the choices.

The CASE structure shown in Fig. 3.3e is really just a compact way to represent a complex IF-THEN-ELSE structure. The choice of action is determined by testing some quantity. The cook or the computer checks the value of the variable called "day" and selects the appropriate actions for that day. Each of the indicated actions, such as "Make celery soup," is itself a sequence of actions which could be represented by the structures we have described. Note that the CASE structure does not contain the final ELSE for an error.

The CASE form is more compact for documentation purposes, and some high-level languages such as Pascal allow you to implement it directly. However, the nested IF-THEN-ELSE structure gives you a much better idea of how you write an assembly language program section to choose between several alternative actions.

The WHILE-DO structure in Fig. 3.3f is one form of repetition. It is used to indicate that you want to do some action or sequence of actions as long as some condition is present. This structure represents a *program loop*. The example in Fig. 3.3f is

WHILE money lasts DO

 Eat supper out.

 Go to movie.

 Take a taxi home.

This example shows a sequence of actions you might do each evening until you ran out of money. Note that in this structure, the condition is checked *before* the action is done the first time. You certainly want to check how much money you have before eating out.

Another useful repetition structure is the REPEAT-UNTIL structure shown in Fig. 3.3g. You use this structure to indicate that you want the program to repeat some action or series of actions until some condition is present. A good example of the use of this structure is the programming problem we used in the discussion of flowcharts. The example is

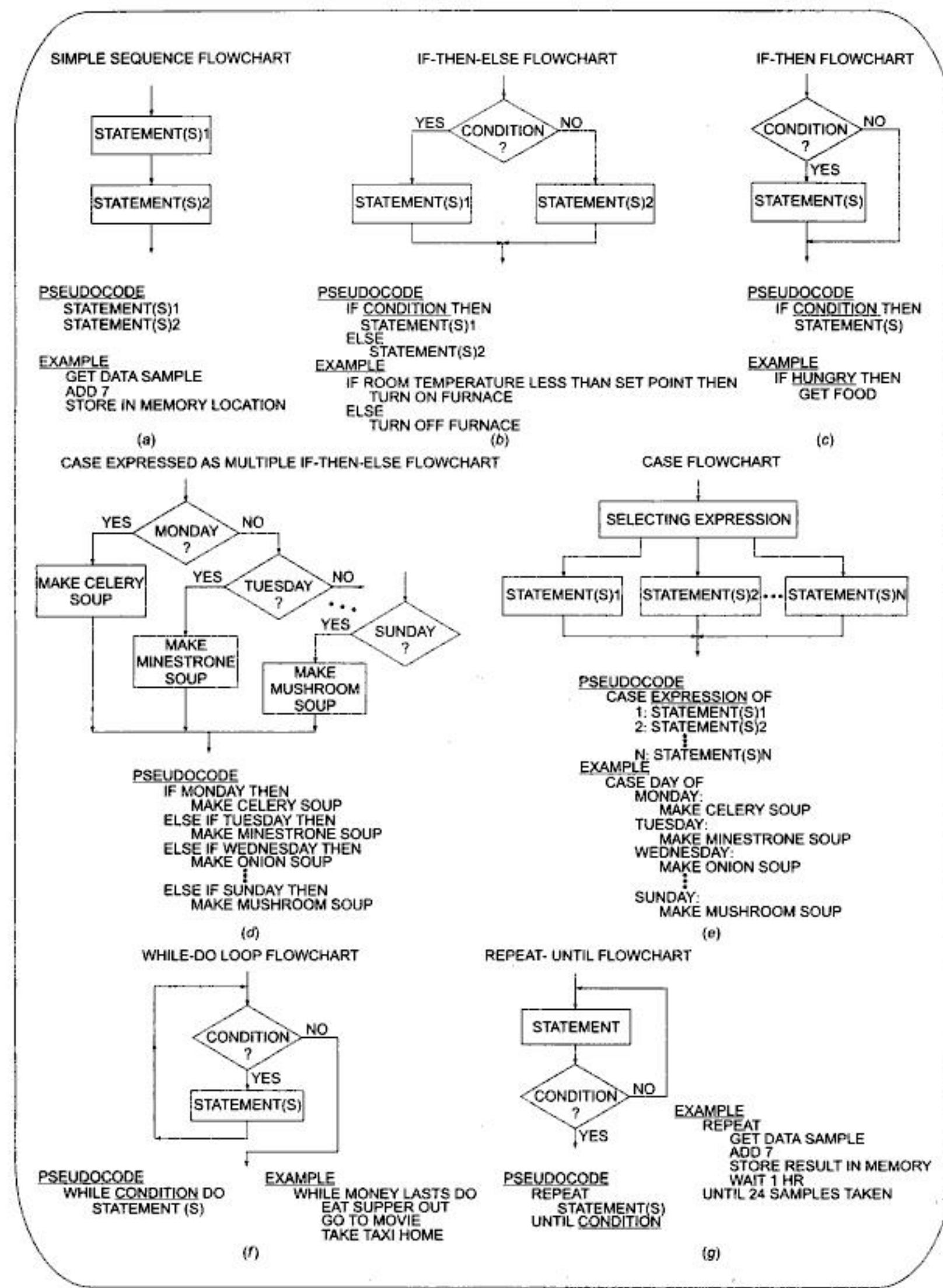


Fig. 3.3 Standard program structures. (a) Sequence, (b) IF-THEN-ELSE, (c) IF-THEN, (d) CASE expressed as nested IF-THEN-ELSE, (e) CASE, (f) WHILE-DO, (g) REPEAT-UNTIL.

REPEAT

Get data sample from sensor.
Add correction of + 7.
Store result in a memory location.
Wait 1 hour.
UNTIL 24 samples taken.

Note that in a REPEAT-UNTIL structure, the action(s) is done once before the condition is checked. If you want the condition to be checked before any action is done, then you can write the algorithm with a WHILE-DO structure as follows:

WHILE NOT 24 samples DO

Read data sample from temperature sensor.
Add correction factor of + 7.
Store result in memory location.
Wait 1 hour.

Remember, a REPEAT-UNTIL structure indicates that the condition is first checked *after* the statement(s) is performed, so the action or series of actions will always be done at least once. If you don't want this to happen, then use the WHILE-DO, which indicates that the condition is checked *before* any action is taken. As we will show later, the structure you use makes a difference in the actual assembly language program you write to implement it.

The WHILE-DO and REPEAT-UNTIL structures contain a simple IF-THEN-ELSE decision operation. However, since this decision is an implied part of these two structures, we don't indicate the decision separately in them.

Another form of the repetition operation that you might see in high-level language programs is the FOR-DO loop. This structure has the form

FOR count = 1 TO n DO

statement
statement

This FOR DO loop, as it is often called, simply repeats the sequence of actions n times, so for assembly language algorithms we usually implement this type of operation with a REPEAT-UNTIL structure.

Incidentally, if you compare the space required by the pseudocode representation for a program structure with the space required by the flowchart representation for the same structure, the space advantage of pseudocode should be obvious.

Throughout the rest of this book, we show you how to use these structures to represent program actions and how to implement these structures in assembly language.

**SUMMARY OF PROGRAM STRUCTURE
REPRESENTATION FORMS**

Writing a successful program does not consist of just writing down a series of instructions. You must first think carefully about what you want the program to do and how you want the program to do it. Then you must represent the structure of the program in some way that is very clear both to you and to anyone else who might have to work on the program.

One way of representing program operations is with flowcharts. Flowcharts are a very graphic representation, and they are useful for short program segments, especially those that deal directly with hardware. However, flowcharts use a great deal of space. Consequently, the flowchart for even a moderately complex program may take up several pages. It often becomes difficult to follow program flow back and forth between pages. Also, since there are no agreed-upon structures, a poor programmer can write a flowchart which jumps all over the place and is even more difficult to follow. The term "logical spaghetti" comes to mind here.

A second way of representing the operations you want in a program is with a top-down design approach and standard program structures. The overall program problem is first broken down into major functional modules. Each of these modules is broken down into smaller and smaller modules until the steps in each module are obvious. The algorithms for the whole program and for each module are expressed with a standard structure. Only three basic structures, SEQUENCE, IF-THEN-ELSE, and WHILE-DO, are needed to represent any needed program action or series of actions. However, other useful structures such as IF-THEN, REPEAT-UNTIL, FOR-DO, and CASE can be derived from these basic three. A structure can contain another structure of the same type or one of the other types. Each structure has only one entry point and one exit point. These programming structures may seem restrictive, but using them usually results in algorithms which are easy to follow. Also, as we will show you soon, if you write the algorithm for a program carefully with these standard structures, it is relatively easy to translate the algorithm to the equivalent assembly language instructions.

Finding the Right Instruction

After you get the structure of a program worked out and written down, the next step is to determine the instruction statements required to do each part of the program. Since the examples in this book are based on the 8086 family of

microprocessors, now is a good time to give you an overview of the instructions the 8086 has for you to use. First, however, is a hint about how to approach these instructions.

You do not usually learn a new language by memorizing an entire dictionary of the language. A better way is to learn a few useful words and practice putting these words together in simple sentences. You can then learn more words as you need them to express more complex thoughts. Likewise, you should not try to memorize all the instructions for a microprocessor at once.

For future reference, Chapter 6 contains a dictionary of all the 8086 instructions with detailed descriptions and examples of each. As an introduction, however, the few pages here contain a list of all the 8086 instructions with a short explanation of each. Skim through the list and pick out a dozen or so instructions that seem useful and understandable. As a start, look for move, input, output, logical, and arithmetic instructions. Then look through the list again to see if you can find the instructions that you might use to do the “read temperature sensor value from a port, add + 7, and store result in memory” example program.

You can use Chapter 6 as a reference as you write programs. Here we simply list the 8086 instructions in *functional* groups with single-sentence descriptions so that you can see the types of instructions that are available to you. As you read through this section, do not expect to understand all the instructions. When you start writing programs, you will probably use this section to determine the type of instruction and Chapter 6 to get the instruction details as you need them. After you have written a few programs, you will remember most of the basic instruction types and will be able to simply look up an instruction in Chapter 6 to get any additional details you need. Chapter 4 shows you in detail how to use the move, arithmetic, logical, jump, and string instructions. Chapter 5 shows how to use the call instructions and the stack.

DATA TRANSFER INSTRUCTIONS

General-purpose byte or word transfer instructions:

| MNEMONIC | DESCRIPTION |
|----------|---|
| MOV | Copy byte or word from specified source to specified destination. |
| PUSH | Copy specified word to top of stack. |
| POP | Copy word from top of stack to specified location. |
| PUSHA | (80186/80188 only) Copy all registers to stack. |
| POPA | (80186/80188 only) Copy words from stack to all registers. |

XCHG Exchange bytes or exchange words.
 XLAT Translate a byte in AL using a table in memory.

Simple input and output port transfer instructions:

IN Copy a byte or word from specified port to accumulator.
 OUT Copy a byte or word from accumulator to specified port.

Special address transfer instructions:

LEA Load effective address of operand into specified register.
 LDS Load DS register and other specified register from memory.
 LES Load ES register and other specified register from memory.

Flag transfer instructions:

LAHF Load (copy to) AH with the low byte of the flag register.
 SAHF Store (copy) AH register to low byte of flag register.
 PUSHF Copy flag register to top of stack.
 POPF Copy word at top of stack to flag register.

ARITHMETIC INSTRUCTIONS

Addition instructions:

ADD Add specified byte to byte or specified word to word.
 ADC Add byte + byte + carry flag or word + word + carry flag.
 INC Increment specified byte or specified word by 1.
 AAA ASCII adjust after addition.
 DAA Decimal (BCD) adjust after addition.

Subtraction instructions:

SUB Subtract byte from byte or word from word.
 SBB Subtract byte and carry flag from byte or word and carry flag from word.
 DEC Decrement specified byte or specified word by 1.
 NEG Negate — invert each bit of a specified byte or word and add 1 (form 2's complement).
 CMP Compare two specified bytes or two specified words.
 AAS ASCII adjust after subtraction.
 DAS Decimal (BCD) adjust after subtraction.

Multiplication instructions:

MUL Multiply unsigned byte by byte or unsigned word by word.

| | | |
|--|---|--|
| IMUL | Multiply signed byte by byte or signed word by word. | |
| AAM | ASCII adjust after multiplication. | |
| <i>Division instructions:</i> | | |
| DIV | Divide unsigned word by byte or unsigned double word by word. | |
| IDIV | Divide signed word by byte or signed double word by word. | |
| AAD | ASCII adjust before division. | |
| CBW | Fill upper byte of word with copies of sign bit of lower byte. | |
| CWD | Fill upper word of double word with sign bit of lower word. | |
| BIT MANIPULATION INSTRUCTIONS | | |
| <i>Logical instructions:</i> | | |
| NOT | Invert each bit of a byte or word. | |
| AND | AND each bit in a byte or word with the corresponding bit in another byte or word. | |
| OR | OR each bit in a byte or word with the corresponding bit in another byte or word. | |
| XOR | Exclusive OR each bit in a byte or word with the corresponding bit in another byte or word. | |
| TEST | AND operands to update flags, but don't change operands. | |
| <i>Shift instructions:</i> | | |
| SHL/SAL | Shift bits of word or byte left, put zero(s) in LSB(s). | |
| SHR | Shift bits of word or byte right, put zero(s) in MSB(s). | |
| SAR | Shift bits of word or byte right, copy old MSB into new MSB. | |
| <i>Rotate instructions:</i> | | |
| ROL | Rotate bits of byte or word left, MSB to LSB and to CF. | |
| ROR | Rotate bits of byte or word right, LSB to MSB and to CF. | |
| RCL | Rotate bits of byte or word left, MSB to CF and CF to LSB. | |
| RCR | Rotate bits of byte or word right, LSB to CF and CF to MSB. | |
| PROGRAM EXECUTION TRANSFER INSTRUCTIONS | | |
| These instructions are used to tell the 8086 to start fetching instructions from some new address, rather than continuing in sequence. | | |
| <i>Unconditional transfer instructions:</i> | | |
| CALL | Call a procedure (subprogram), save return address on stack. | |
| RET | Return from procedure to calling | |

STRING INSTRUCTIONS

A *string* is a series of bytes or a series of words in sequential memory locations. A string often consists of ASCII character codes. In the list, a “/” is used to separate different mnemonics for the same instruction. Use the mnemonic which most clearly describes the function of the instruction in a specific application. A “B” in a mnemonic

PROGRAM EXECUTION TRANSFER INSTRUCTIONS

These instructions are used to tell the 8086 to start fetching instructions from some new address, rather than continuing in sequence.

Unconditional transfer instructions:

| | |
|------|--|
| CALL | Call a procedure (subprogram), save return address on stack. |
| RET | Return from procedure to calling program. |
| JMP | Go to specified address to get next instruction. |

Conditional transfer instructions:

A “/” is used to separate two mnemonics which represent the same instruction. Use the mnemonic which most clearly describes the decision condition in a specific program. These instructions are often used after a compare instruction. The terms *below* and *above* refer to unsigned

binary numbers. *Above* means larger in magnitude. The terms *greater than* or *less than* refer to signed binary numbers. *Greater than* means more positive.

| | |
|---------|--|
| JA/JNBE | Jump if above/Jump if not below or equal. |
| JAE/JNB | Jump if above or equal/Jump if not below. |
| JB/JNAE | Jump if below/Jump if not above or equal. |
| JBE/JNA | Jump if below or equal/Jump if not above. |
| JC | Jump if carry flag CF = 1. |
| JE/JZ | Jump if equal/Jump if zero flag ZF = 1. |
| JG/JNLE | Jump if greater/Jump if not less than or equal. |
| JGE/JNL | Jump if greater than or equal/Jump if not less than. |
| JL/JNGE | Jump if less than/Jump if not greater than or equal. |
| JLE/JNG | Jump if less than or equal/Jump if not greater than. |
| JNC | Jump if no carry (CF = 0). |
| JNE/JNZ | Jump if not equal/Jump if not zero (ZF = 0). |
| JNO | Jump if no overflow (overflow flag OF = 0). |
| JNP/JPO | Jump if not parity/Jump if parity odd (PF = 0). |
| JNS | Jump if not sign (sign flag SF = 0). |
| JO | Jump if overflow flag OF = 1. |
| JP/JPE | Jump if parity/Jump if parity even (PF = 1). |
| JS | Jump if sign (SF = 1). |

Iteration control instructions:

These instructions can be used to execute a series of instructions some number of times. Here mnemonics separated by a “/” represent the same instruction. Use the one that best fits the specific application.

| | |
|---------------|--|
| LOOP | Loop through a sequence of instructions until CX = 0. |
| LOOPE/LOOPZ | Loop through a sequence of instructions while ZF = 1 and CX ≠ 0. |
| LOOPNE/LOOPNZ | Loop through a sequence of instructions while ZF = 0 and CX ≠ 0. |
| JCXZ | Jump to specified address if CX = 0. |

If you aren't tired of instructions, continue skimming through the rest of the list. Don't worry if the explanation

is not clear to you because we will explain these instructions in detail in later chapters.

Interrupt instructions:

| | |
|--|--|
| INT | Interrupt program execution, call service procedure. |
| INTO | Interrupt program execution if OF = 1. |
| IRET | Return from interrupt service procedure to main program. |
| <i>High-level language interface instructions:</i> | |
| ENTER | (80186/80188 only) Enter procedure. |
| LEAVE | (80186/80188 only) Leave procedure. |
| BOUND | (80186/80188 only) Check if effective address within specified array bounds. |

PROCESSOR CONTROL INSTRUCTIONS

Flag set/clear instructions:

| | |
|-----|---|
| STC | Set carry flag CF to 1. |
| CLC | Clear carry flag CF to 0. |
| CMC | Complement the state of the carry flag CF. |
| STD | Set direction flag DF to 1 (decrement string pointers). |
| CLD | Clear direction flag DF to 0. |
| STI | Set interrupt enable flag to 1 (enable INTR input). |
| CLI | Clear interrupt enable flag to 0 (disable INTR input). |

External hardware synchronization instructions:

| | |
|------|--|
| HLT | Halt (do nothing) until interrupt or reset. |
| WAIT | Wait (do nothing) until signal on the TEST pin is low. |
| ESC | Escape to external coprocessor such as 8087 or 8089. |
| LOCK | An instruction prefix. Prevents another processor from taking the bus while the adjacent instruction executes. |

No operation instruction:

| | |
|-----|------------------------------------|
| NOP | No action except fetch and decode. |
|-----|------------------------------------|

Now that you have skimmed through an overview of the 8086 instruction set, let's see whether you found the instructions needed to implement the “read sensor, add + 7, and store result in memory” example program. The IN instruction can be used to read the temperature value

from an A/D converter connected to a port. The ADD instruction can be used to add the correction factor of +7 to the value read in. Finally, the MOV instruction can be used to copy the result of the addition to a memory location. A major point here is that breaking down the programming problem into a sequence of steps makes it easy to find the instruction or small group of instructions that will perform each step. The next section shows you how to write the actual program using the 8086 instructions.

Writing a Program

INITIALIZATION INSTRUCTIONS

After finding the instructions you need to do the main part of your program, there are a few additional instructions that you need to determine before you actually write your program. The purpose of these additional instructions is to *initialize* various parts of the system, such as segment registers, flags, and programmable port devices. Segment registers, for example, must be loaded with the upper 16 bits of the address in memory where you want the segment to begin. For our “read temperature sensor, add + 7, and store result in memory” example program, the only part we need to initialize is the data segment register. The data segment register must be initialized so that we can copy the result of the addition to a location in memory. If, for example, we want to store data in memory starting at address 00100H, then we want the data segment register to contain the upper 16 bits of this address, 0010H. The 8086 does not have an instruction to move a number directly into a segment register. Therefore, we move the desired number into one of the 16-bit general-purpose registers, then copy it to the desired segment register. Two MOV instructions will do this.

If you are using the stack in your program, then you must include instructions to load the stack segment register and an instruction to load the stack pointer register with the offset of the top of the stack. Most microcomputer systems contain several programmable peripheral devices, such as ports, timers, and controllers. You must include instructions which send control words to these devices to tell them the function you want them to perform. Also, you usually want to include instructions which set or clear the control flags, such as the interrupt enable flag and the direction flag.

The best way to approach the initialization task is to make a checklist of all the registers, programmable devices, and flags in the system you are working on. Then you can mark the ones you need for a specific program and

determine the instructions needed to initialize each part. An initialization list for an 8086-based system, such as the SDK-86 prototyping board, might look like the following.

INITIALIZATION LIST

- Data segment register DS
- Stack segment register SS
- Extra segment register ES
- Stack pointer register SP
- 8255 programmable parallel port
- 8259A priority interrupt controller
- 8254 programmable counter
- 8251A programmable serial port
- Initialize data variables
- Set interrupt enable flag

As you can see, the list can become quite lengthy even though we have not included all the devices a system might commonly have. Note that initializing the code segment register CS is absent from this list. The code segment register is loaded with the correct starting value by the system command you use to run the program. Now let's see how you put all these parts together to make a program.

A STANDARD PROGRAM FORMAT

In this section we show you how to format your programs if you are going to construct the machine codes for each instruction by *hand*. A later section of this chapter will show you the additional parts you need to add to the program if you are going to use a computer program called an *assembler* to produce the binary codes for the instructions.

To help you write your programs in the correct format, *assembly language coding sheets* such as that shown in Fig. 3.4 are available. The ADDRESS column is used for the address or the offset of a code byte or data byte. The actual code bytes or data bytes are put in the DATA/CODE column. A *label* is a name which represents an address referred to in a jump or call instruction; labels are put in the LABELS column. A label is followed by a colon (:) if it is used by a jump or call instruction in the same code segment. The MNEM column contains the opcode mnemonics for the instructions. The OPERAND(S) column contains the registers, memory locations, or data acted upon by the instructions. A COMMENTS column gives you space to describe the function of the instruction for future reference.

Figure 3.4, shows how instructions for the “read temperature, add +7, store result in memory” program can be written in sequence on a coding sheet. We will discuss

| PROGRAMMER | D. V. HALL | | | SHEET / | OF / |
|-----------------|--|--------|-------|------------|---|
| PROGRAM TITLE | READ TEMPERATURE AND CORRECT | | | DATE: | 1/1/XX |
| ABSTRACT: | This program reads in a temperature value from a sensor connected to port 05h, adds a correction factor of + 7 to the value read in, and then stores the result in a reserved memory location. | | | | |
| PROCEDURES: | None called. | | | | |
| REGISTERS USED: | Ax | | | | |
| FLAGS AFFECTED: | All conditional | | | | |
| PORTS: | uses 05 as input port | | | | |
| MEMORY: | 00100H - DATA: 00200H-0020CH, CODE | | | | |
| ADDRESS | DATA or CODE | LABELS | MNEM. | OPERAND(S) | COMMENTS |
| 00100 | XX | | | | Reserve memory location to store result. This location will be loaded with a data byte as read in & corrected by the program. |
| 00101 | | | | | |
| 00102 | | | | | |
| 00103 | | | | | |
| 00104 | | | | | XX means "don't care" about contents of location. |
| 00105 | | | | | |
| 00106 | | | | | |
| 00107 | | | | | |
| 00108 | | | | | |
| 00109 | | | | | |
| 0010A | | | | | |
| 0010B | | | | | |
| 0010C | | | | | |
| 0010D | | | | | |
| 0010E | | | | | Code starts here |
| 0010F | | | | | Note break in address |
| 200 | B8 | | MOV | AX, 0010H | Initialize DS to point to start of memory set aside for storing data |
| 01 | 10 | | | | |
| 02 | 00 | | | | |
| 03 | 8E | | MOV | DS, AX | |
| 04 | D8 | | | | |
| 05 | E4 | | IN | AL, 05H | Read temperature from port 05H |
| 06 | 05 | | | | |
| 07 | 04 | | ADD | AL, 07H | Add correction factor of +07 |
| 08 | 07 | | | | |
| 09 | A2 | | MOV | [0000], AH | Store result in reserved memory |
| 0A | 00 | | | | |
| 0B | 00 | | | | |
| 0C | EE | | INT | 3 | Stop, wait for command from user |
| 0D | | | | | |
| 0E | | | | | |
| 0F | | | | | |

Fig. 3.4 Assembly language program on standard coding form.

here the operation of these instructions to the extent needed. If you want more information, detailed descriptions of the *syntax* (assembly language grammar) and operation of each of these instructions can be found in Chapter 6.

The first line at the top of the coding form in Fig. 3.4 does not represent an instruction. It simply indicates that we want to set aside a memory location to store the result. This location must be in available RAM so that we can write to it. Address 00100H is an available RAM location on an SDK-86 prototyping board, so we chose it for this example. Next, we decide where in memory we want to start putting the code bytes for the instructions of the program. Again, on an SDK-86 prototyping board, address 00200H and above is available RAM, so we chose to start the program at address 00200H.

The first operation we want to do in the program is to initialize the data segment register. As discussed previously, two MOV instructions are used to do this. The MOV AX, 0010H instruction, when executed, will load the upper 16 bits of the address we chose for data storage into the AX register. The MOV DS, AX instruction will copy this number from the AX register to the data segment register. Now we get to the instructions that do the input, add, and store operations. The IN AL, 05H instruction will copy a data byte from the port 05H to the AL register. The ADD AL, 07 instruction will add 07H to the AL register and leave the result in the AL register. The MOV [0000], AL instruction will copy the byte in AL to a memory location at a displacement of 0000H from the data segment base. In other words, AL will be copied to a physical address computed by adding 0000 to the segment base address represented by the 0010H in the DS register. The result of this addition is a physical address of 00100H, so the result in AL will be copied to physical address 00100H in memory. This is an example of the direct addressing mode described near the end of the previous chapter.

The INT 3 instruction at the end of the program functions as a *breakpoint*. When the 8086 on an SDK-86 board executes this instruction, it will cause the 8086 to stop executing the instructions of your program and return control to the *monitor* or *system program*. You can then use *system commands* to look at the contents of registers and memory locations, or you can run another program. Without an instruction such as this at the end of the program, the 8086 would fetch and execute the code bytes for your program, then go on fetching meaningless bytes from memory and trying to execute them as if they were code bytes.

The next major section of this chapter will show you how to construct the binary codes for these and other 8086

instructions so that you can assemble and run the programs on a development board such as the SDK-86. First, however, we want to use Fig. 3.4 to make an important point about writing assembly language programs.

DOCUMENTATION

In a previous section of this chapter, we stressed the point that you should do a lot of thinking and carefully write down the algorithm for a program before you start writing instruction statements. You should also document the program itself so that its operation is clear to you and to anyone else who needs to understand it.

Each page of the program should contain the name of the program, the page number, the name of the programmer, and perhaps a version number. Each program or procedure should have a heading block containing an *abstract* describing what the program is supposed to do, which procedures it calls, which registers it uses, which ports it uses, which flags it affects, the memory used, and any other information which will make it easier for another programmer to interface with the program.

Comments should be used generously to describe the specific *function* of an instruction or group of instructions in this particular program. Comments should not be just an expansion of the instruction mnemonic. A comment of ";add 7 to AL" after the instruction ADD AL, 07H, for example, would not tell you much about the function of the instruction in a particular program. A more enlightening comment might be ";Add altitude correction factor to temperature." Incidentally, not every statement needs an individual comment. It is often more useful to write a comment which explains the function of a group of instructions.

We cannot overemphasize the importance of clear, concise documentation in your programs. Experience has shown that even a short program you wrote without comments a month ago may not be at all understandable to you now.

CONSTRUCTING THE MACHINE CODES FOR 8086 INSTRUCTIONS

This section shows you how to construct the binary codes for 8086 instructions. Most of the time you will probably use an assembler program to do this for you, but it is useful to understand how the codes are constructed. If you have an 8086-based prototyping board such as the Intel SDK-86 available, knowing how to hand code instructions will enable you to code, enter, and run simple programs.

Instruction Templates

To code the instructions for 8-bit processors such as the 8085, all you have to do is look up the hexadecimal code for each instruction on a one-page chart. For the 8086, the process is not quite as simple. Here's why. There are 32 ways to specify the source of the operand in an instruction such as `MOV CX, source`. The source of the operand can be any one of eight 16-bit registers, or a memory location specified by any one of 24 memory addressing modes. Each of the 32 possible instructions requires a different binary code. If CX is made the source rather than the destination, then there are 32 ways of specifying the destination. Each of these 32 possible instructions requires a different binary code. There are thus 64 different codes for `MOV` instructions using CX as a source or as a destination. Likewise, another 64 codes are required to specify all the possible `MOVs` using CL as a source or a destination, and 64 more are required to specify all the possible `MOVs` using CH as a source or a destination. The point here is that, because there is such a large number of possible codes for the 8086 instructions, it is impractical to list them all in a simple table. Instead, we use a *template* for each basic instruction type and fill in bits within this template to indicate the desired addressing mode, data type, etc. In other words, we build up the instruction codes on a bit- by-bit basis.

Different Intel literature shows two slightly different formats for coding 8086 instructions. One format is shown at the end of the 8086 data sheet in Appendix A. The second format is shown along with the 8086 instruction timings in Appendix B. We will start by showing you how to use the templates shown in the 8086 data sheet.

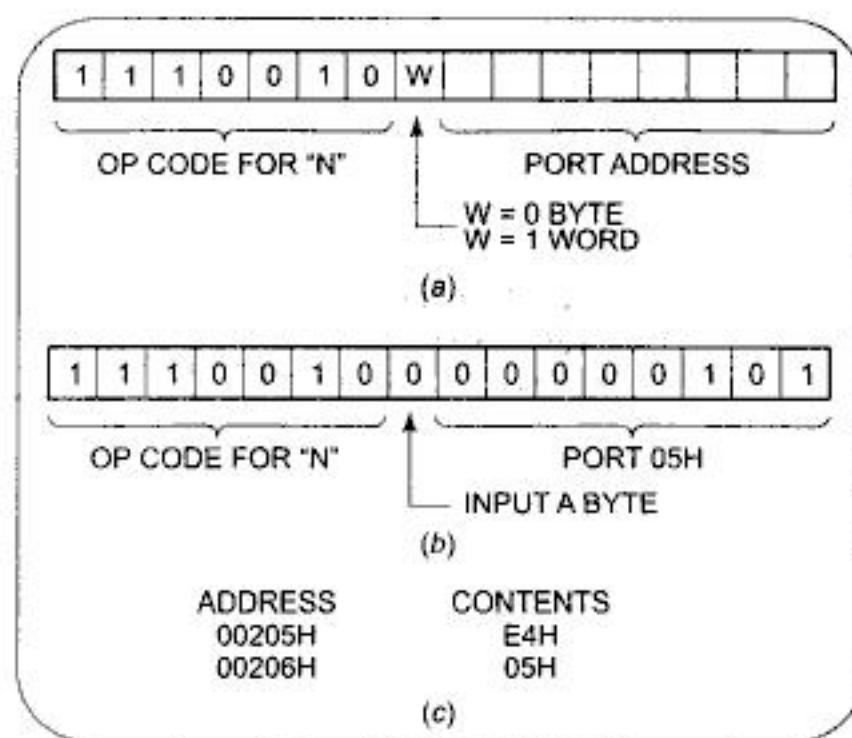


Fig. 3.5 Coding template for 8086 IN (fixed port) instruction. (a) Template. (b) Example 1 (c) Hex codes in sequential memory locations.

As a first example of how to use these templates, we will build the code for the `IN AL, 05H` instruction from our example program. To start, look at the template for this instruction in Fig. 3.5a. Note that two bytes are required for the instruction. The upper 7 bits of the first byte tell the 8086 that this is an “input from a fixed port” instruction. The bit labeled “W” in the template is used to tell the 8086 whether it should input a byte to AL or a word to AX. If you want the 8086 to input a byte from an 8-bit port to AL, then make the W bit a 0. If you want the 8086 to input a word from a 16-bit port to the AX register, then make the W bit a 1. The 8-bit port address, 05H or

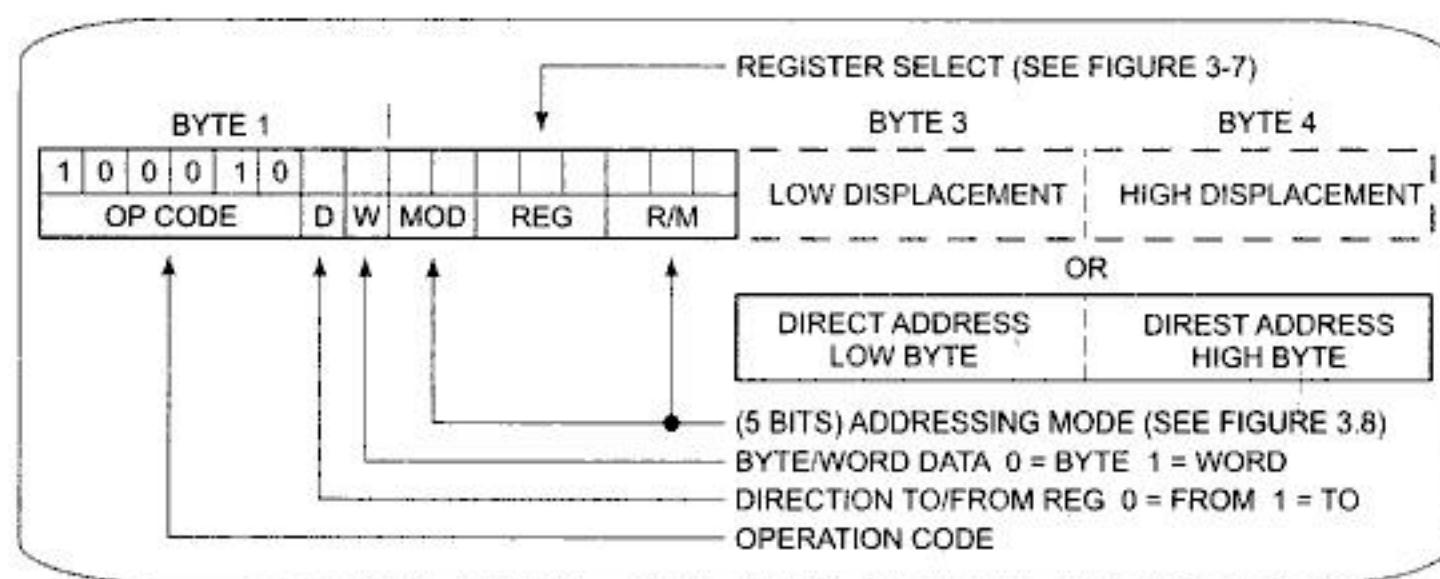


Fig. 3.6 Coding template for 8086 instructions which `MOV` data between registers or between a register and a memory location.

00000101 binary, is put in the second byte of the instruction. When the program is loaded into memory to be run, the first instruction byte will be put in one memory location, and the second instruction byte will be put in the next. Fig. 3.5c shows this in hexadecimal form as E4H, 05H.

To further illustrate how these templates are used, we will show here several examples with the simple MOV instruction. We will then show you how to construct the rest of the codes for the example program in Fig. 3.4. Other examples will be shown as needed in the following chapters.

MOV Instruction Coding Format and Examples

FORMAT

Figure 3.6 shows the coding template or format for 8086 instructions which MOV data from a register to a register, from a register to a memory location, or from a memory location to a register. Note that at least two code bytes are required for the instruction.

The upper 6 bits of the first byte are an opcode which indicates the general type of instruction. Look in the table in Appendix A to find the 6-bit opcode for this MOV register/memory to/from register instruction. You should find it to be 100010.

The W bit in the first word is used to indicate whether a byte or a word is being moved. If you are moving a byte, make W = 0. If you are moving a word, make W = 1.

In this instruction, one operand must always be a register, so 3 bits in the second byte are used to indicate which register is involved. The 3-bit codes for each register are shown in the table at the end of Appendix A and in Fig. 3.7. Look in one of these places to find the code for the CL register. You should get 001.

| REGISTER | CODE |
|----------|------|
| W=1 | W=0 |
| AL | AX |
| BL | BX |
| CL | CX |
| DL | DX |
| AH | SP |
| BH | DI |
| CH | BP |
| DH | SI |
| SEGREG | CODE |
| CS | 01 |
| DS | 11 |
| ES | 00 |
| SS | 10 |

Fig. 3.7 Instruction codes for 8086 registers.

The D bit in the first byte of the instruction code is used to indicate whether the data is being moved to the register identified in the REG field of the second byte or *from* that register. If the instruction is moving data to the register identified in the REG field, make D = 1. If the instruction is moving data *from* that register, make D = 0.

Now remember that in a MOV instruction, one operand must be a register and the other operand may be a register or a memory location. The 2-bit field labeled MOD and the 3-bit field labeled R/M in the second byte of the instruction code are used to specify the desired addressing mode for the other operand. Fig. 3.8 shows the MOD and R/M bit patterns for each of the 32 possible addressing modes. Here's an overview of how you use this table.

1. If the other operand in the instruction is also one of the eight registers, then put in 11 for the MOD bits in the instruction code. In the R/M bit positions in the instruction code, put the 3-bit code for the other register.
2. If the other operand is a memory location, there are 24 ways of specifying how the execution unit should compute the effective address of the operand in memory. Remember from Chapter 2 that the effective address can be specified directly in the instruction, it can be contained in a register, or it can be the sum of one or two registers and a displacement. The MOD bits are used to indicate whether the address specification in the instruction contains a displacement. The R/M code indicates which register(s) contain part(s) of the effective address. Here's how it works:

If the specified effective address contains no displacement, as in the instruction MOV CX, [BX] or in the instruction MOV [BX][SI], DX, then make the MOD bits 00 and choose the R/M bits which correspond to the register(s) containing the effective address. For example, if an instruction contains just [BX], the 3-bit R/M code is 111. For an instruction which contains [BX][SI], the R/M code is 000. Note that for direct addressing, where the displacement of the operand from the segment base is specified directly in the instruction, MOD is 00 and R/M is 110. For an instruction using direct addressing, the low byte of the direct address is put in as a third instruction code byte of the instruction, and the high byte of the direct address is put in as a fourth instruction code byte.

3. If the effective address specified in the instruction contains a displacement less than 256 along with a reference to the contents of a register, as in the

| MOD RM \ | 00 | 01 | 10 | 11 | | |
|-------------|-------------------------|------------------|-------------------|----|-------|-------|
| | | | | | W = 0 | W = 1 |
| 000 | [BX] + [SI] | [BX] + [SI] + d8 | [BX] + [SI] + d16 | | AL | AX |
| 001 | [BXI] + [DI] | [BX] + [DI] + d8 | [BX] + [DI] + d16 | | CL | CX |
| 010 | [BP] + [SI] | [BP] + [SI] + d8 | [BP] + [SI] + d16 | | DL | DX |
| 011 | [BP] + [DI] | [BP] + [DI] + d8 | [BP] + [DI] + d16 | | BL | BX |
| 100 | [SI] | [SI] + d8 | [SI] + d16 | | AH | SP |
| 101 | [DI] | [DI] + d8 | [DI] + d16 | | CH | BP |
| 110 | d16 (direct address) | [BP] + d8 | [BP] + d16 | | DH | SI |
| 111 | [BX] | [BX] + d8 | [BX] + d16 | | BH | DI |

MEMORY MODE

REGISTER MODE

d8 = 8-bit displacement d16 = 16-bit displacement

Fig. 3.8 MOD and R/M bit patterns for 8086 instructions. The effective address (EA) produced by these addressing modes will be added to the data segment base to form the physical address, except for those cases where BP is used as part of the EA. In that case the EA will be added to the stack segment base to form the physical address. You can use a segment-override prefix to indicate that you want the EA to be added to some other segment base.

instruction MOV CX, 43H[BX], then code in MOD as 01 and choose the R/M bits which correspond to the register(s) which contain the part(s) for the effective address. For the instruction MOV CX, 43H[BX], MOD will be 01 and R/M will be 111. Put the 8-bit value of the displacement in as the third byte of the instruction.

- If the expression for the effective address contains a displacement which is too large to fit in 8 bits, as in the instruction MOV DX, 4527H[BX], then put in 10 for MOD and choose the R/M bits which correspond to the register(s) which contain the part(s) for the effective address. For the instruction MOV DX, 4527H[BX], the R/M bits are 111. The low byte of the displacement is put in as a third byte of the instruction. The high byte of the displacement is put in as a fourth byte of the instruction. The examples which follow should help clarify all this for you.

MOV Instruction Coding Examples

All the examples in this section use the MOV instruction template in Fig. 3.6. As you read through these examples, it is a good idea to keep track of the bit-by-bit development on a separate piece of paper for practice.

CODING MOV SP, BX

This instruction will copy a word from the BX register to the SP register. Consulting the table in Appendix A, you find that the 6-bit opcode for this instruction is 100010.

Because you are moving a word, W = 1. The D bit for this instruction may be somewhat confusing, however. Since two registers are involved, you can think of the move as either to SP or *from* BX. It actually does not matter which you assume as long as you are consistent in coding the rest of the instruction. If you think of the instruction as moving a word *to* SP, then make D = 1 and put 100 in the REG field to represent the SP register. The MOD field will be 11 to represent register addressing mode. Make the R/M field 011 to represent the other register, BX. The resultant code for the instruction MOV SP, BX will be 10001011 11100011. Fig. 3.9a shows the meaning of all these bits.

If you change the D bit to a 0 and swap the codes in the REG and R/M fields, you will get 10001001 11011100, which is another equally valid code for the instruction. Fig. 3.9b shows the meaning of the bits in this form. This second form, incidentally, is the form that the Intel 8086 Macroassembler produces.

CODING MOV CL, [BX]

This instruction will copy a byte to CL from the memory location whose effective address is contained in BX. The effective address will be added to the data segment base in DS to produce the physical address.

To find the 6-bit opcode for byte 1 of the instruction, consult the table in Appendix A. You should find that this code is 100010. Make D = 1 because data is being moved to register CL. Make W = 0 because the instruction is moving a byte into CL. Next you need to put the 3-bit code

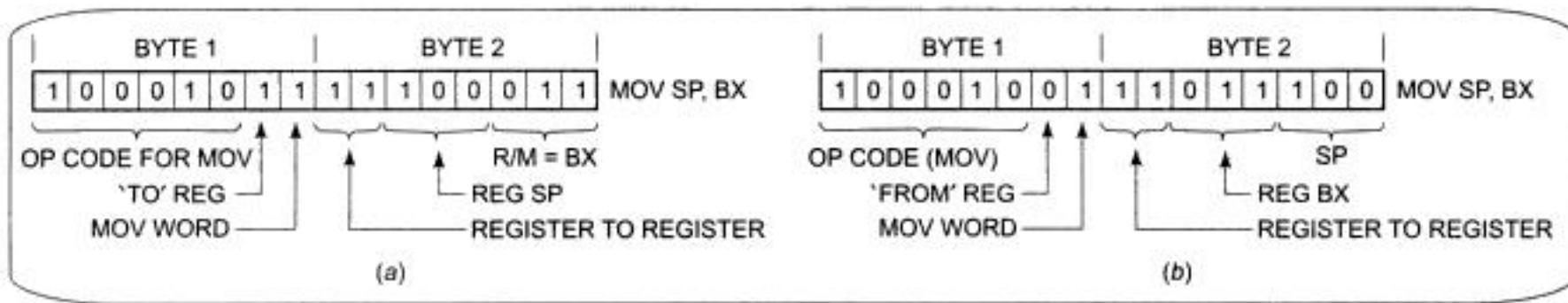


Fig. 3.9 MOV instruction coding examples, (a) MOV SP, BX. (b) MOV SP, BX alternative.

which represents register CL in the REG field of the second byte of the instruction code. The codes for each register are shown in Fig. 3.7. In this figure you should find that the code for CL is 001. Now, all you need to determine is the bit patterns for the MOD and R/M fields. Again use the table in Fig. 3.8 to do this. In the table, first find the box containing the desired addressing mode. The box containing [BX], for example, is in the lower left corner of the table. Read the required MOD-bit pattern from the top of the column. In this case, MOD is 00. Then read the required R/M-bit pattern at the left of the box. For this instruction you should find R/M to be 111. Assembling all these bits together should give you 10001010 00001111 as the binary code for the instruction MOV CL, [BX]. Fig. 3.10 summarizes the meaning of all the bits in this result.

CODING MOV 43H[SI], DH

This instruction will copy a byte from the DH register to a memory location. The BIU will compute the effective address of the memory location by adding the indicated displacement of 43H to the contents of the SI register. As we showed you in the last chapter, the BIU then produces the actual physical address by adding this effective address to the data segment base represented by the 16-bit number in the DS register.

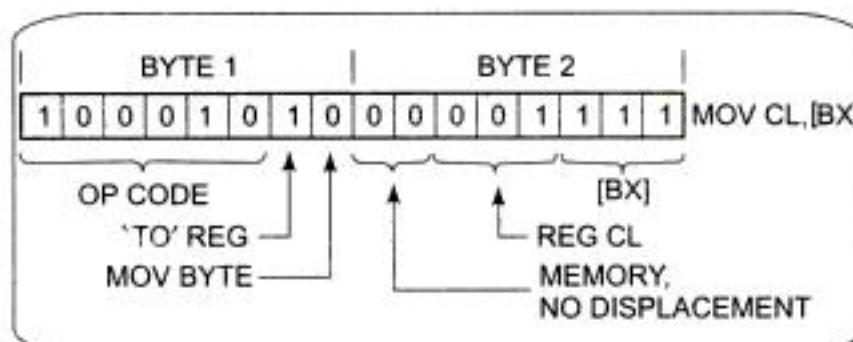


Fig. 3.10 MOV CL, [BX].

The 6-bit opcode for this instruction is again 100010. Put 110 in the REG field to represent the DH register. D = 0 because you are moving data *from* the DH register.

W = 0 because you are moving a byte. The R/M field will be 100 because SI contains part of the effective address. The MOD field will be 01 because the displacement contained in the instruction, 43H, will fit in 1 byte. If the specified displacement had been a number larger than FFH, then MOD would be 10. Putting all these pieces together gives 10001000 01110100 for the first two bytes of the instruction code. The specified displacement, 43H or 01000011 binary, is put after these two as a third instruction byte. Fig. 3.11 shows this. If an instruction specifies a 16-bit displacement, then the low byte of the displacement is put in as byte 3 of the instruction code, and the high byte of the displacement is put in as byte 4 of the instruction code.

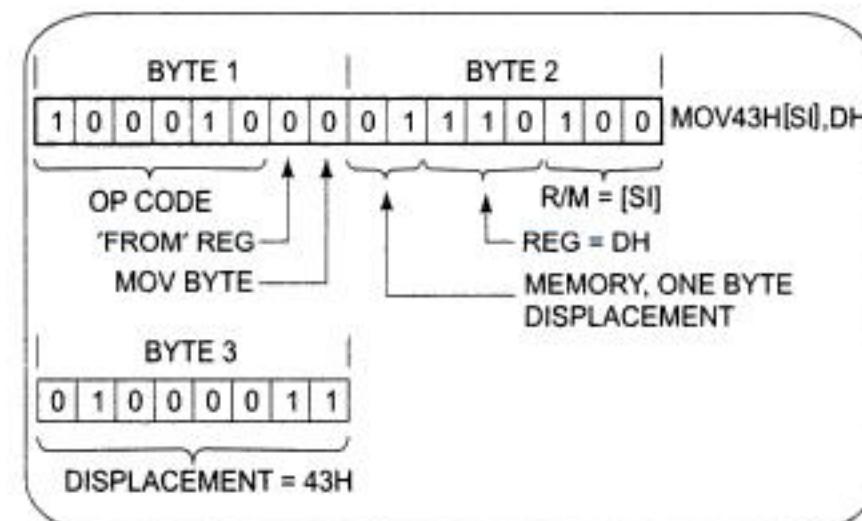


Fig. 3.11 MOV 43H[SI], DH.

CODING MOV CX, [437AH]

This instruction copies the contents of two memory locations into the CX register. The direct address or displacement of the first memory location from the start of the data segment is 437AH. As we showed you in the last chapter, the BIU will produce the physical memory address by adding this displacement to the data segment base represented by the 16-bit number in the DS register.

The 6-bit opcode for this instruction is again 100010. Make D = 1 because you are moving data to the CX

register, and make W = 1 because the data being moved is a word. Put 001 in the REG field to represent the CX register, then consult Fig. 3.8 to find the MOD and R/M codes. In the first column of the figure, you should find a box labeled “direct address,” which is the name given to the addressing mode used in this instruction. For direct addressing, you should find MOD to be 00 and R/M to be 110. The first two code bytes for the instruction, then, are 10001011 00001110. These two bytes will be followed by the low byte of the direct address, 7AH (01111010 binary), and the high byte of the direct address, 43H (01000011 binary). The instruction will be coded into four successive memory addresses as 8BH, 0EH, 7AH, and 43H. Fig. 3.12 spells this out in detail.

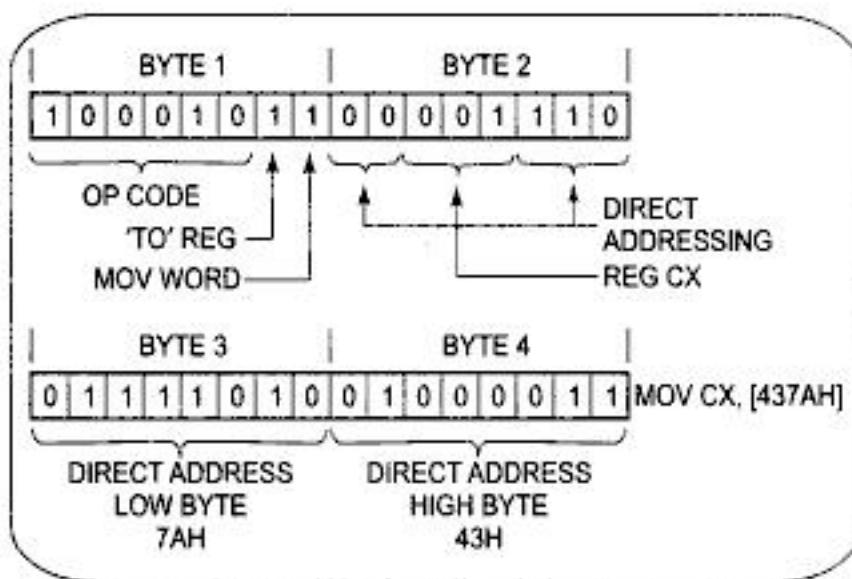


Fig. 3.12 MOV CX, [437AH].

CODING MOV CS:[BX], DL

This instruction copies a byte from the DL register to a memory location. The effective address for the memory location is contained in the BX register. Normally an effective address in BX will be added to the data segment base in DS to produce the physical memory address. In this instruction, the CS: in front of [BX] indicates that we want the BIU to add the effective address to the code segment base in CS to produce the physical address. The CS: is called a *segment override prefix*.

When an instruction containing a segment override prefix is coded, an 8-bit code for the segment override prefix is put in memory *before* the code for the rest of the instruction. The code byte for the segment override prefix has the format 001XX110. You insert a 2-bit code in place of the X's to indicate which segment base you want the effective address to be added to. As shown in Fig. 3.7, the codes for these 2 bits are as follows: ES = 00, CS = 01, SS = 10, and DS = 11. The segment override prefix byte for CS, then, is 00101110. For practice, code out

the rest of this instruction. Fig. 3.13 shows the result you should get and how the code for the segment override prefix is put before the other code bytes for the instruction.

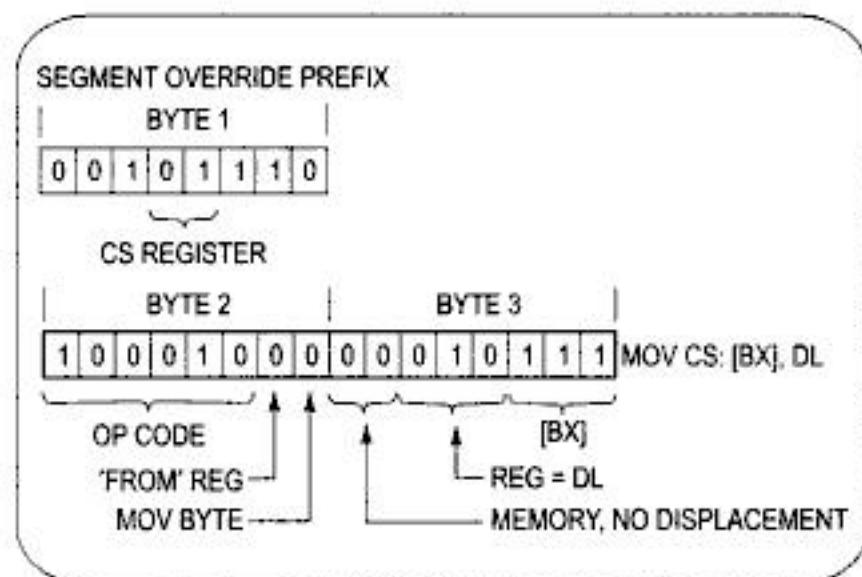


Fig. 3.13 MOV CS:[BX], DL.

Coding the Example Program in Fig. 3.4

Again, as you read through this section, follow the bit-by-bit development of the instruction codes on a separate piece of paper for practice.

MOV AX, 0010H This instruction will load the immediate word 0010H into the AX register. The simplest code template to use for this instruction is listed in the table in Appendix A under the “MOV — Immediate to register” heading. The format for this instruction is 1011 W REG, data byte low, data byte high. W = 1 because you are moving a word. Consult Fig. 3.7 to find the code for the AX register. You should find this to be 000. Put this 3-bit code in the REG field of the instruction code. The completed instruction code byte is 10111000. Put the low byte of the immediate number, 10H, in as the second code byte. Then put the high byte of the immediate data, 00H, in as the third code byte. The resultant sequence of code bytes, then, will be B8H, 10H, 00H.

MOV DS, AX This instruction copies the contents of the AX register into the data segment register. The template to use for coding this instruction is found in the table in Appendix A under the heading “MOV — Register/memory to segment register.” The format for this template is 10001110 MOD 0 segreg R/M. Segreg represents the 2-bit code for the desired segment register, as shown in Fig. 3.7. These codes are also found in the table at the end of Appendix A. The segreg code for the DS register is 11. Since the other operand is a register, MOD should be 11. Put the 3-bit code for the AX register, 000, in the R/M

field. The resultant codes for the two code bytes should then be 10001110 11011000, or 8EH D8H.

IN AL, 05H This instruction copies a byte of data from port 05H to the AL register. The coding for this instruction was described in a previous section. The code for the instruction is 11100100 00000101 or E4H 05H.

ADD AL, 07H This instruction adds the immediate number 07H to the AL register and puts the result in the AL register. The simplest template to use for coding this instruction is found in the table in Appendix A under the heading "ADD— Immediate to accumulator." The format is 0000010 W, data byte, data byte. Since you are adding a byte, W = 0. The immediate data byte you are adding will be put in the second code byte. The third code byte will not be needed because you are adding only a byte. The resultant codes, then, are 00000100 00000111 or 04H 07H.

MOV [0000], AL This instruction copies the contents of the AL register to a memory location. The direct address or displacement of the memory location from the start of the data segment is 0000H. The code template for this instruction is found in the table in Appendix A under the heading "MOV — Accumulator to memory." The format for the instruction is 1010001 W, address low byte, address high byte. Since the instruction moves a byte, W = 0. The low byte of the direct address is written in as the second instruction code byte, and the high byte of the direct address is written in as the third instruction code byte. The codes for these 3 bytes, then, will be 10100010 00000000 00000000 or A2H 00H 00H.

INT 3 In some 8086 systems this instruction causes the 8086 to stop executing your program instructions, return to the monitor program, and wait for your next command. According to the format table in Appendix A, the code for a type 3 interrupt is the single byte 11001100 or CCH.

SUMMARY OF HAND CODING THE EXAMPLE PROGRAM

Figure 3.4 shows the example program with all the instruction codes in sequential order as you would write them so that you could load the program into memory and run it. Codes are in HEX to save space.

A Look at Another Coding Template Format

As we mentioned previously, Intel literature shows the 8086 instruction coding templates in two different forms. The preceding sections have shown you how to use the templates found at the end of the 8086 data sheet in

Appendix A. Now let's take a brief look at the second form, which is shown along with the instruction clock cycles in Appendix B.

The only difference between the second form for the templates and the form we discussed previously is that the D and W bits are not individually identified. Instead, the complete opcode bytes are shown for each version of an instruction. For example, in Appendix B, the opcode byte for the MOV memory 8, register 8 instruction is shown as 88H, and the opcode byte for the MOV memory 16, register 16 instruction is shown as 89H. If you compare these codes with those derived from Appendix A, you will see that the only difference between the two codes is the W bit. For the 8-bit move, W = 0, and for the 16-bit move, W = 1.

One important point to make about using the templates in Appendix B is that for operations involving two registers, the register identified in the REG field is not consistent from instruction to instruction. For the MOV instructions, the templates in Appendix B assume that the 3-bit code for the source register is put in the REG field of the MOD/RM instruction byte, and the 3-bit code for the destination register is put in the R/M field of the MOD/RM instruction byte. According to Appendix B, the template for a 16-bit register-to-register move is 89H followed by the MOD reg R/M byte. In this template, D = 0, so the 3-bit code for the source register will be put in the reg field. Using this template, then, the instruction MOV BX, CX is coded as 1001001 11001011 or 89H CBH.

For the ADD, ADC, SUB, SBB, AND, OR, and XOR instructions which involve two registers, the templates in Appendix B show D = 1. To be consistent with these templates, then, you have to put the 3-bit code for the destination register in the reg field in the instruction.

It really doesn't matter whether you use the templates in Appendix A or those in Appendix B, as long as you are consistent in coding each instruction.

A Few Words about Hand Coding

If you have to hand code 8086 assembly language programs, here are a few tips to make your life easier. First, check your algorithm very carefully to make sure that it really does what it is supposed to do. Second, initially write down just the assembly language statements and comments for your program. You can check the table in the appendix to determine how many bytes each instruction takes so that you know how many blank lines to leave between instruction statements. You may find it helpful to insert three or four NOP instructions after every nine or ten instructions. The NOP instruction doesn't do anything but kill time. However, if you accidentally leave

out an instruction in your program, you can replace the NOPs with the needed instruction(s). This way you don't have to rewrite the entire program after the missing instruction.

After you have written down the instruction statements, recheck very carefully to make sure you have the right instructions to implement your algorithm. Then work out the binary codes for each instruction and write them in the appropriate places on the coding form.

Hand coding is laborious for long programs. When writing long programs, it is much more efficient to use an assembler. The next section of this chapter shows you how to write your programs so that you can use an assembler to produce the machine codes for the instructions.

WRITING PROGRAMS FOR USE WITH AN ASSEMBLER

If you have an 8086 assembler available, you should learn to use it as soon as possible. Besides doing the tedious task of producing the binary codes for your instruction statements, an assembler also allows you to refer to data items by name rather than by their numerical offsets. As you should soon see, this greatly reduces the work you have to do and makes your programs much more readable. In this section we show you how to write your programs so that you can use an assembler on them.

NOTE: The assembly language programs in the rest of this book were assembled with TASM 1.0 from Borland International or MASM 5.1 from Microsoft Corp. TASM is faster, but the program format for these two assemblers is essentially the same. If you are using some other assembler, check the manual for it to determine any differences in syntax from the examples in this book.

Program Format

The best way to approach this section seems to be to show you a simple, but complete, program written for an assembler and explain the function of the various parts of the program. By now you are probably tired of the "read temperature, add +7, and store result in memory" program, so we will use another example.

Figure 3.14, shows an 8086 assembly language program which multiplies two 16-bit binary numbers to give a 32-bit binary result. If you have a microcomputer development system or a microcomputer with an 8086 assembler to work on, this is a good program for you to key in, assemble, and run to become familiar with the operation of your system. (A sequence of exercises in the

accompanying lab manual explains how to do this.) In any case, you can use the structure of this example program as a model for your own programs.

In addition to program instructions, the example program in Fig. 3.14 contains directions to the assembler. These directions to the assembler are commonly called *assembler directives* or *pseudo operations*. A section at the end of Chapter 6 lists and describes for your reference a large number of the available assembler directives. Here we will discuss the basic assembler directives you need to get started writing programs. We will introduce more of these directives as we need them in the next two chapters.

SEGMENT and ENDS Directives

The SEGMENT and ENDS directives are used to identify a group of data items or a group of instructions that you want to be put together in a particular segment. These directives are used in the same way that parentheses are used to group like terms in algebra. A group of data statements or a group of instruction statements contained between SEGMENT and ENDS directives is called a *logical segment*. When you set up a logical segment, you give it a name of your choosing. In the example program, the statements DATA_HERE SEGMENT and DATA_HERE ENDS set up a logical segment named DATA_HERE. There is nothing sacred about the name DATA_HERE. We simply chose this name to help us remember that this logical segment contains data statements. The statements CODE_HERE SEGMENT and CODE_HERE ENDS in the example program set up a logical segment named CODE_HERE which contains instruction statements. Most 8086 assemblers, incidentally, allow you to use names and labels of up to 31 characters. You can't use spaces in a name, but you can use an underscore as shown to separate words in a name. Also, you can't use instruction mnemonics as segment names or labels. Throughout the rest of the program you will refer to a logical segment by the name that you give it when you define it.

A logical segment is not usually given a physical starting address when it is declared. After the program is assembled and perhaps linked with other assembled program modules, it is then assigned the physical address where it will be loaded in memory to be run.

Naming Data and Addresses — EQU, DB, DW, and DD Directives

Programs work with three general categories of data: constants, variables, and addresses. The value of a

```

; 8086 PROGRAM F3-14.ASM
;ABSTRACT : This program multiplies the two 16-bit words in the memory
;           ; locations called MULTIPLICAND and MULTIPLIER. The result
;           ; is stored in the memory location, PRODUCT
;REGISTERS : Uses CS, DS, AX, DX
;PORTS     : None used

DATA_HERE SEGMENT
    MULTIPLICAND DW 204AH      ; First word here
    MULTIPLIER   DW 3B2AH      ; Second word here
    PRODUCT      DW 2 DUP(0)   ; Result of multiplication here
DATA_HERE ENDS

CODE_HERE SEGMENT
    ASSUME CS:CODE_HERE, DS:DATA_HERE
START:  MOV AX, DATA_HERE       ; Initialize DS register
        MOV DS, AX
        MOV AX, MULTIPLICAND    ; Get one word
        MUL MULTIPLIER          ; Multiply by second word
        MOV PRODUCT, AX         ; Store low word of result
        MOV PRODUCT+2, DX       ; Store high word of result
        INT 3                  ; Wait for command from user
CODE_HERE ENDS
END START

; Programs to be run using a debugger in DOS must include the START: label and the
; START after the END followed by a carriage return. Programs to be downloaded and run need
; only the END directive followed by a carriage return.

```

Fig. 3.14 Assembly language source program to multiply two 16-bit binary numbers to give a 32-bit result.

constant does not change during the execution of the program. The number 7 is an example of a constant you might use in a program. A variable is the name given to a data item which can change during the execution of a program. The current temperature of an oven is an example of a variable. Addresses are referred to in many instructions. You may, for example, load an address into a register or jump to an address.

Constants, variables, and addresses used in your programs can be given names. This allows you to refer to them by name rather than having to remember or calculate their value each time you refer to them in an instruction. In other words, if you give names to constants, variables, and addresses, the assembler can use these names to find a desired data item or address when you refer to it in an instruction. Specific directives are used to give names to constants and variables in your programs. Labels are used to give names to addresses in your programs.

THE EQU DIRECTIVE

The EQU, or *equate*, directive is used to assign names to constants used in your programs. The statement CORRECTION_FACTOR EQU 07H, in a program such

as our previous example, would tell the assembler to insert the value 07H every time it finds the name CORRECTION_FACTOR in a program statement. In other words, when the assembler reads the statement ADD AL,CORRECTION_FACTOR, it will automatically code the instruction as if you had written it ADD AL,07H. Here's the advantage of using an EQU directive to declare constants at the start of your program. Suppose you use the correction factor of + 07H 23 times in your program. Now the company you work for changes the brand of temperature sensor it buys, and the new correction factor is + 09H. If you used the number 07H directly in the 23 instructions which contain this correction factor, then you have to go through the entire program, find each instruction that uses the correction factor, and update the value. Murphy's law being what it is, you are likely to miss one or two of these, and the program won't work correctly. If you used an EQU at the start of your program and then referred to CORRECTION_FACTOR by name in the 23 instructions, then all you do is change the value in the EQU statement from 07H to 09H and reassemble the program. The assembler automatically inserts the new value of 09H in all 23 instructions.

DB, DW, AND DD DIRECTIVES

The DB, DW, and DD directives are used to assign names to variables in your programs. The DB directive after a name specifies that the data is of *type byte*. The program statement OVEN_TEMPERATURE DB 27H, for example, declares a variable of type byte, gives it the name OVEN_TEMPERATURE, and gives it an initial value of 27H. When the binary code for the program is loaded into memory to be run, the value 27H will be loaded into the memory location identified by the name OVEN_TEMPERATURE DB 27H.

As another example, the statement CONVERSION_FACTORS DB 27H, 48H, 32H, 69H will declare a data structure (array) of 4 bytes and initialize the 4 bytes with the specified 4 values. If you don't care what value a data item is initialized to, then you can indicate this with a "?" as in the statement TARE_WEIGHT DB ?

NOTE: Variables which are changed during the operation of a program should also be initialized with program instructions so that the program can be rerun from the start without reloading it to initialize the variables.

DW is used to specify that the data is of *type word* (16 bits), and DD is used to specify that the data is of *type doubleword* (32 bits). The example program in Fig. 3.14 shows three examples of naming and initializing word-type data items.

The first example, MULTIPLICAND DW 204AH, declares a data word named MULTIPLICAND and initializes that data word with the value 204AH. What this means is that the assembler will set aside two successive memory locations and assign the name MULTIPLICAND to the first location. As you will see, this allows us to access the data in these memory locations by name. The MULTIPLICAND DW 204AH statement also indicates that when the final program is loaded into memory to be run, these memory locations will be loaded with (initialized to 204AH). Actually, since this is an Intel microprocessor, the first address in memory will contain the low byte of the word, 4AH, and the second memory address will contain the high byte of the word, 20H.

The second data declaration example in Fig. 3.14, MULTIPLIER DW 3B2AH, sets aside storage for a word in memory and gives the starting address of this word the name MULTIPLIER. When the program is loaded, the first memory address will be initialized with 2AH, and the second memory location with 3BH.

The third data declaration example in Fig. 3.14,

PRODUCT DW 2 DUP(0), sets aside storage for two words in memory and gives the starting address of the first word the name PRODUCT. The DUP(0) part of the statement tells the assembler to initialize the two words to all zeros. When we multiply two 16-bit binary numbers, the product can be as large as 32 bits, so we must set aside this much space to store the product. We could have used the DD directive to declare PRODUCT a doubleword, but since in the program we move the result to PRODUCT one word at a time, it is more convenient to declare PRODUCT 2 words.

Figure 3.15 shows how the data for MULTIPLICAND, MULTIPLIER, and PRODUCT will actually be arranged in memory starting from the base of the DATA_HERE segment. The first byte of MULTIPLICAND, 4AH, will be at a displacement of zero from the segment base, because MULTIPLICAND is the first data item declared in the logical segment DATA_HERE. The displacement of the second byte of MULTIPLICAND is 0001. The displacement of the first byte of MULTIPLIER from the segment base is 0002H, and the displacement of the second byte of MULTIPLIER is 0003H. These are the displacements that we would have to figure out for each data item if we were not using names to refer to them.

If the logical segment DATA_HERE is eventually put in ROM or EPROM, then MULTIPLICAND will function as a constant, because it cannot be changed during program execution. However, if DATA_HERE is eventually put in RAM, then MULTIPLICAND can function as a variable because a new value could be written in those memory locations during program execution.

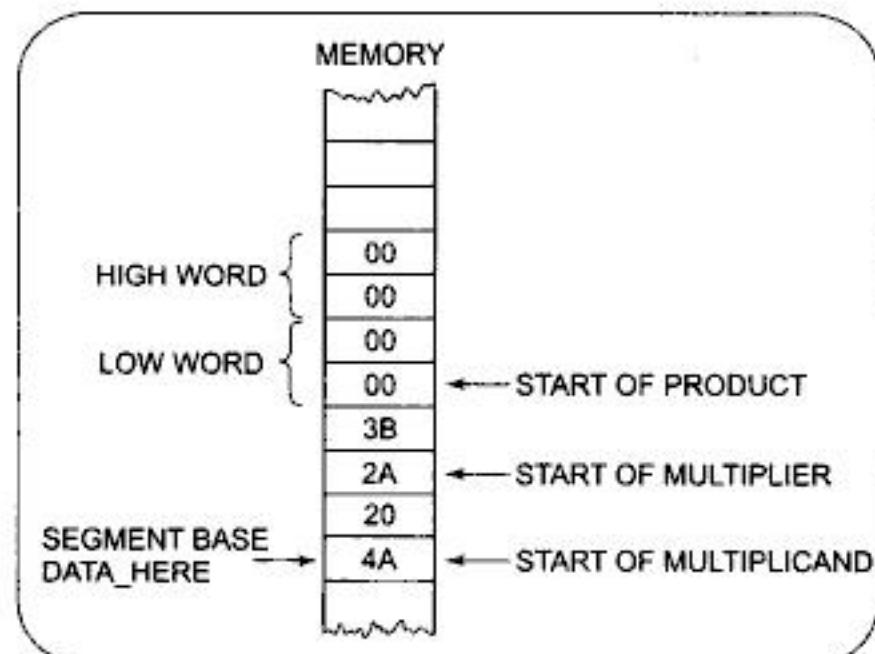


Fig. 3.15 Data arrangement in memory for multiply program.

Types of Numbers Used in Data Statements

All the previous examples of DB, DW, and DD declarations use hexadecimal numbers, as indicated by an "H" after the number. You can, however, put in a number in any one of several other forms. For each form you must tell the assembler which form you are using.

BINARY

For example, when you use a binary number in a statement, you put a "B" after the string of 1's and 0's to let the assembler know that you want the number to be treated as a binary number. The statement TEMP_MAX DB 01111001B is an example. If you want to put in a negative binary number, write the number in its 2's complement sign-and-magnitude form.

DECIMAL

The assembler treats a number with no identifying letter after it as a decimal number. The assembler automatically converts a decimal number in a statement to binary so that the value can be loaded into memory. Given the statement TEMP_MAX DB 49, for example, the assembler will automatically convert the 49 decimal to its binary equivalent, 00110001. If you indicate a negative number in a data declaration statement, the assembler will convert the number to its 2's complement sign-and-magnitude form. For example, given the statement TEMP_MIN DB -20, the assembler will insert the value 11101100, which is the 2's complement representation for -20 decimal.

NOTE: If you forget to put an H after a number that you want the assembler to treat as hexadecimal, the assembler will treat it as a decimal number. You can put a D after the decimal values if you want to indicate more clearly that the value is decimal.

HEXADECIMAL

As shown in several previous examples, a hexadecimal number is indicated by an H after the hexadecimal digits. The statement MULTIPLIER DW 3B2AH is an example. A zero must be placed in front of a hex number that starts with a letter; for example, the number AH must be written 0AH.

BCD

Remember from Chapter 1 that in BCD each decimal digit is represented by its 4-bit binary equivalent. The decimal number 37, for example, is represented in BCD as 00110111. As you can see, this number is equal to 37H. The only way you can tell whether the number 00110111

represents BCD 37 or hexadecimal 37 is by how it is used in the program! The point here is that if you want the assembler to initialize a variable with the value 37 BCD, you put an H after the number. The statement SECONDS DB 59H, for example, will initialize the variable SECONDS with 01011001, the BCD representation of 59.

ASCII

You can declare a data structure (array) containing a sequence of ASCII codes by enclosing the letters or numbers after a DB in single quotation marks. The statement BOY1 DB 'ALBERT', for example, tells the assembler to declare a data item named BOY1 that has six memory locations. It also tells the assembler to put the ASCII code for A in the first memory location, the ASCII code for L in the second, the ASCII code for B in the third, etc. The assembler will automatically determine the ASCII codes for the letters or numbers within the quotes. Note that this ASCII trick can be used only with the DB directive.

Accessing Named Data with Program Instructions

Now that we have shown you how a data structure can be set up, let's look at how program instructions access this data. Temporarily skipping over the first two instructions in the CODE_HERE section of the program in Fig. 3.16, find the instruction MOV AX, MULTIPLICAND. This instruction, when executed, will copy a word from the memory location named MULTIPLICAND to the AX register. Here's how this works.

When the assembler reads through this program the first time, it automatically calculates the offset of each of the named data items from the segment base DATA_HERE. In Fig. 3.15 you can see that the displacement of MULTIPLICAND from the segment base is 0000. This is because MULTIPLICAND is the first data item declared in the segment. The assembler, then, will find that the displacement of MULTIPLICAND is 0000H. When the assembler reads the program the second time to produce the binary codes for the instructions, it will insert this displacement as part of the binary code for the instruction MOV AX, MULTIPLICAND. Since we know that the displacement of MULTIPLICAND is 0000, we could have written the instruction as MOV AX, [0000]. However, there would be a problem if we later changed the program by adding another data item before MULTIPLICAND in DATA_HERE. The displacement of MULTIPLICAND would be changed. Therefore, we would have to remember to go through the entire program and correct the displacement in all instructions that access MULTIPLICAND. If you use a name to refer to each data item as shown, the assembler will automatically calculate

Turbo Assembler Version 1.0

Page 1

```

1 ; 8086 PROGRAM F3-14.ASM
2 ;ABSTRACT : This program multiplies the two 16-bit words in the memory
3 ; locations called MULTIPLICAND and MULTIPLIER. The result
4 ; is stored in the memory location, PRODUCT
5 ;REGISTERS : Uses CS, DS, AX, DX
6 ;PORTS : None used
7
8 0000      DATA_HERE   SEGMENT
9 0000 204A    MULTIPLICAND DW 204AH      ; First word here
10 0002 382A   MULTIPLIER   DW 382AH      ; Second word here
11 0004 02*(0000) PRODUCT     DW 2 DUP(0)  ; Result of multiplication here
12 0008      DATA_HERE   ENDS
13
14 0000      CODE_HERE  SEGMENT
15           ASSUME   CS:CODE_HERE, DS:DATA_HERE
16 0000 B8 0000s  START:    MOV AX, DATA_HERE      ; Initialize DS register
17 0003 BE D8    MOV DS, AX
18 0005 A1 0000r  MOV AX, MULTIPLICAND  ; Get one word
19 0008 F7 26 0002r MUL MULTIPLIER      ; Multiply by second word
20 000C A3 0004r  MOV PRODUCT, AX      ; Store low word of result
21 000F 89 16 0006r MOV PRODUCT+2, DX  ; Store high word of result
22 0013 CC       INT 3                 ; Wait for command from user
23 0014          CODE_HERE  ENDS
24           END START

```

Turbo Assembler Version 1.0

Page 2

Symbol Table

| Symbol Name | Type | Value |
|-------------------------------------|--------|----------------|
| ??DATE | Text | "04-06-89" |
| ??FILENAME | Text | "F3-14 " |
| ??TIME | Text | "07:41:58" |
| ??VERSION | Number | 0100 |
| @CPU | Text | 0101H |
| @CURSEG | Text | CODE_HERE |
| @FILENAME | Text | F3-14 |
| @MORDSIZE | Text | 2 |
| MULTIPLICAND | Word | DATA_HERE:0000 |
| MULTIPLIER | Word | DATA_HERE:0002 |
| PRODUCT | Word | DATA_HERE:0004 |
| START | Near | CODE_HERE:0000 |
| Groups & Segments | | |
| Bit Size Align Combine Class | | |
| CODE_HERE | 16 | 0014 Para none |
| DATA_HERE | 16 | 0008 Para none |

Fig. 3.16 Assembler listing for example program in Figure 3.14.

the correct displacement of that data item for you and insert this displacement each time you refer to it in an instruction.

To summarize how this works, then, the instruction **MOV AX, MULTIPLICAND** is an example of direct

addressing where the direct address or displacement of the desired data word in the data segment is represented by the name MULTIPLICAND. For instructions such as this, the assembler will automatically calculate the displacement of the named data item from the start of the

segment and insert this value as part of the binary code for the instruction. This can be seen on line 18 of the assembler listing shown in Fig. 3.16. When the instruction executes, the BIU will add the displacement contained in the instruction to the data segment base in DS to produce the 20-bit physical address of the data word named MULTIPLICAND.

The next instruction in the program in Fig. 3.16 is another example of direct addressing using a named data item. The instruction MUL MULTIPLIER multiplies the word from the memory location named MULTIPLIER in DATA_HERE by the word in the AX register. When the assembler reads through this program the first time, it will find that the displacement of MULTIPLIER in DATA_HERE is 0002H. When it reads through the program the second time, it inserts this displacement as part of the binary code for the MUL instruction, as shown on line 19 in Fig. 3.16. When the MUL MULTIPLIER instruction executes, the BIU will add the displacement contained in the instruction to the data segment base in DS to address MULTIPLIER in memory. After the multiplication, the low word of the result is left in the AX register, and the high word of the result is left in the DX register.

The next instruction, MOV PRODUCT, AX, in the program in Fig. 3.16 copies the low word of the result from AX to memory. The low byte of AX will be copied to a memory location named PRODUCT. The high byte of AX will be copied to the next higher address, which we can refer to as PRODUCT + 1. As you can see on line 20 in Fig. 3.16, the displacement of PRODUCT, 0004H, is inserted in the code for the MOV PRODUCT, AX instruction.

The following instruction in the program, MOV PRODUCT + 2, DX, copies the high word of the multiplication result from DX to memory. When the assembler reads this instruction, it will add the indicated "2" to the displacement it calculated for PRODUCT and insert the result as part of the binary code for the instruction, as shown on line 21 in Fig. 3.16. Therefore, when the instruction executes, the low byte of DX will be copied to memory at a displacement of PRODUCT + 2. The high byte of DX will be copied to a memory location which we can refer to as PRODUCT + 3. Fig. 3.15 shows how the two words of the product are put in memory. Note that the lower byte of a word is always put in the lower memory address.

This example program should show you that if you are using an assembler, names are a very convenient way of specifying the direct address of data in memory. In the

next section we show you how to refer to addresses by name.

Naming Addresses — Labels

One type of name used to represent addresses is called a *label*. Labels are written in the label field of an instruction statement or a directive statement. One major use of labels is to represent the destination for jump and call instructions. Suppose, for example, we want the 8086 to jump back to some previous instruction over and over. Instead of computing the numerical address that we want the 8086 to jump to, we put a label in front of the destination instruction and write the jump instruction as JMP label:. Here is a specific example.

NEXT: IN AL, 05H ; Get data sample from port 05H;
; Process data value read in

JMP NEXT ; Get next data value and process

If you use a label to represent an address, as shown in this example, the assembler will automatically calculate the address that needs to be put in the code for the jump instruction. The next two chapters show many examples of the use of labels with jump and call instructions.

Another example of using a name to represent an address is in the SEGMENT directive statement. The name DATA_HERE in the statement DATA_HERE SEG_MENT, for example, represents the starting address of a segment named DATA_HERE. Later we show you how we use this name to initialize the data segment register, but first we will discuss some other parts you need to know about in the example program in Fig. 3.14.

The ASSUME Directive

An 8086 program may have several logical segments that contain code and several that contain data. However, at any given time the 8086 works directly with only four physical segments: a *code segment*, a *data segment*, a *stack segment*, and an *extra segment*. The ASSUME directive tells the assembler which logical segment to use for each of these physical segments at a given time.

In Fig. 3.14, for example, the statement ASSUME CS:CODE_HERE, DS:DATA_HERE tells the assembler that the logical segment named CODE_HERE contains the instruction statements for the program and should be treated as a code segment. It also tells the assembler that it should treat the logical segment DATA_HERE as the data segment for this program. In other words, the DS:DATA_HERE part of the statement tells the assembler that for any instruction which refers to data in the data

segment, data will be found in the logical segment **DATA_HERE**. The **ASSUME . . . DS:DATA_HERE**, for example, tells the assembler that a named data item such as **MULTIPLICAND** is contained in the logical segment called **DATA_HERE**. Given this information, the assembler can construct the binary codes for the instruction. As we explained before, the displacement of **MULTIPLICAND** from the start of the **DATA_HERE** segment will be inserted as part of the instruction by the assembler.

If you are using the stack segment and the extra segment in your program, you must include terms in the **ASSUME** statement to tell the assembler which logical segments to use for each of these. To do this, you might add terms such as **SS:STACK_HERE**, **ES:EXTRA_HERE**. As we will show later, you can put another **ASSUME** directive later in the program to tell the assembler to use different logical segments from that point on.

If the **ASSUME** directive is not completely clear to you at this point, don't worry. We show many more examples of its use throughout the rest of the book. We introduced the **ASSUME** directive here because you need to put it in your programs for most 8086 assemblers. You can use the **ASSUME** statement in Fig. 3.14 as a model of how to write this directive for your programs.

Initializing Segment Registers

The **ASSUME** directive tells the assembler the names of the logical segments to use as the code segment, data segment, stack segment, and extra segment. The assembler uses displacements from the start of the specified logical segment to code out instructions. When the instructions are executed, the displacements in the instructions will be added to the segment base addresses represented by the 16-bit numbers in the segment registers to produce the actual physical addresses. The assembler, however, cannot directly load the segment registers with the upper 16 bits of the segment starting addresses as needed.

The segment registers other than the code segment register must be initialized by program instructions before they can be used to access data. The first two instructions of the example program in Fig. 3.14 show how you initialize the data segment register. The name **DATA_HERE** in the first instruction represents the upper 16 bits of the starting address you give the segment **DATA_HERE**. Since the 8086 does not allow us to move this immediate number directly into the data segment register, we must first load it into one of the general-purpose registers, then copy it into the data segment register. **MOV AX, DATA-HERE** loads the upper 16 bits of the segment starting address

into the **AX** register. **MOV DS, AX** copies this value from **AX** to the data segment register. This is the same operation we described for hand coding the example program in Fig. 3.4, except that here we use the segment name instead of a number to refer to the segment base address. In this example we used the **AX** register to pass the value, but any 16-bit register other than a segment register can be used. If you are hand coding your program, you can just insert the upper 16 bits of the 20-bit segment starting address in place of **DATA_HERE** in the instruction. For example, if in your particular system you decide to locate **DATA_HERE** at address **00300H**, **DS** should be loaded with **0030H**. If you are using an assembler, you can use the segment name to refer to the segment base address, as shown in the example.

If you use the stack segment and the extra segment in a program, the stack segment register and the extra segment register must be initialized by program instructions in the same way.

When the assembler reads through your assembly language program, it calculates the displacement of each named variable from the start of the logical segment that contains it. The assembler also keeps track of the displacement of each instruction code byte from the start of a logical segment. The **CS:CODE_HERE** part of the **ASSUME** statement in Fig. 3.14 tells the assembler to calculate the displacements of the following instructions from the start of the logical segment **CODE_HERE**. In other words, it tells the assembler that when this program is run, the code segment register will contain the upper 16 bits of the address where the logical segment **CODE_HERE** was located in memory. The instruction byte displacements that the assembler is keeping track of are the values that the 8086 will put in the instruction pointer (IP) to fetch each instruction byte.

There are several ways in which the **CS** register can be loaded with the code segment base address and the instruction pointer can be loaded with the offset of the instruction byte to be fetched next. The first way is with the command you give your system to execute a program starting at a given address. A typical command of this sort is **G = 0010:0000 <CR>**. (**<CR>** means "press the return key.") This command will load **CS** with **0010** and load **IP** with **0000**. The 8086 will then fetch and execute instructions starting from address **00100**, the address produced when the BIU adds **IP** to the code segment base in the **CS** register.

As we will show you in the next two chapters, jump and call instructions load new values in **IP**, and in some cases they load new values in the **CS** register.

The END Directive

The END directive, as the name implies, tells the assembler to stop reading. Any instructions or statements that you write after an END directive will be ignored.

ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT TOOLS

Introduction

For all but the very simplest assembly language programs, you will probably want to use some type of *microcomputer development system* and *program development tools* to make your work easier. A typical system might consist of an IBM PC-type microcomputer with at least several hundred kilobytes of RAM, a keyboard and video display, floppy and/or hard disk drives, a printer, and an emulator. Fig. 3.17 shows an Applied Microsystems ES 1800 16-bit emulator which can be added to an IBM PC/AT or compatible computer to produce a complete 8086/80186/80286 development system.

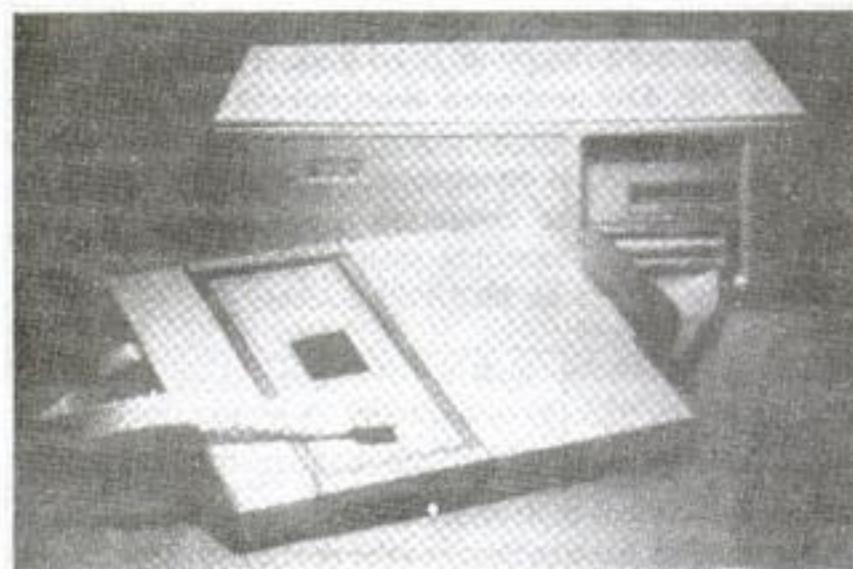


Fig. 3.17 Applied Microsystems ES 1800 16-bit emulator. (Applied Microsystems Corp.)

The following sections give you an introduction to several common program development tools which you use with a system such as this. Most of these tools are programs which you run to perform some function on the program you are writing. You will have to consult the manuals for your system to get the specific details, but this section should give you an overview of the steps involved in developing an assembly language program. An accompanying lab manual takes you through the use of all these tools with the SDK-86 board and an IBM PC-type computer.

Editor

An *editor* is a program which allows you to create a file containing the assembly language statements for your program. Examples of suitable editors are PC Write, Wordstar, and the editor that comes with some assemblers.

Figure 3.14 shows an example of the format you should use when typing in your program. The actual position of each field on a line is not important, but you must put the fields of each statement in the correct order, and you must leave at least one blank between fields. Whenever possible, we like to line the fields up in columns so that it is easier to read the program.

As you type in your program, the editor stores the ASCII codes for the letters and numbers in successive RAM locations. If you make a typing error, the editor will let you back up and correct it. If you leave out a program statement, the editor will let you move everything down and insert the line. This is much easier than working with pencil and paper, even if you type as slowly as I do.

When you have typed in all of your program, you then save the file on a floppy or hard disk. This file is called a *source file*. The next step is to process the source file with an assembler. Incidentally, if you are going to use the TASM or MASM assembler, you should give your source file name the extension .ASM. You might, for instance, give the example source program in Fig. 3.14 a name such as MULTIPLY.ASM.

Assembler

As we told you earlier in the chapter, an *assembler* program is used to translate the assembly language mnemonics for instructions to the corresponding binary codes. When you run the assembler, it reads the source file of your program from the disk where you saved it after editing. On the first pass through the source program, the assembler determines the displacement of named data items, the offset of labels, etc., and puts this information in a *symbol table*. On the second pass through the source program, the assembler produces the binary code for each instruction and inserts the offsets, etc., that it calculated during the first pass.

The assembler generates two files on the floppy or hard disk. The first file, called the *object file*, is given the extension .OBJ. The object file contains the binary codes for the instructions and information about the addresses of the instructions. After further processing, the contents of this file will be loaded into memory and run. The second file generated by the assembler is called the *assembler list file* and is given the extension .LST.

Figure 3.16 shows the assembler list file for the source program in Fig. 3.14. The list file contains your assembly language statements, the binary codes for each instruction, and the offset for each instruction. You usually send this file to a printer so that you will have a printout of the entire program to work with when you are testing and troubleshooting the program. The assembler listing will also indicate any typing or syntax (assembly language grammar) errors you made in your source program.

To correct the errors indicated on the listing, you use the editor to reedit your source program and save the corrected source program on disk. You then reassemble the corrected source program. It may take several times through the edit-assemble loop before you get all the syntax errors out of your source program.

NOTE: The assembler only finds syntax errors; it will not tell you whether your program does what it is supposed to do. To determine whether your program works, you have to run the program and test it.

Now let's take a closer look at some of the information given on the assembler listing in Fig. 3.16. The leftmost column in the listing gives the offsets of data items from the start of the data segment and the offsets of code bytes from the start of the code segment. Note that the assembler generates only offsets, not absolute physical addresses. A linker or locator will be used to assign the physical starting addresses for the segments.

As evidence of this, note that the MOV AX, DATA_HERE statement is assembled with some blanks after the basic instruction code because the start of DS is not known at the time the program is assembled.

The trailer section of the listing in Fig. 3.16 gives some additional information about the segments and names used in the program. The statement CODE_HERE 160014 Para none, for example, tells you that the segment CODE_HERE is 14H bytes long. The statement MULTIPLIER Word DATA_HERE:0002 tells you that MULTIPLIER is a variable of type word and that it is located at an offset of 0002 in the segment DATA_HERE.

Linker

A *linker* is a program used to join several object files into one large object file. When writing large programs, it is usually much more efficient to divide the large program into smaller *modules*. Each module can be individually written, tested, and debugged. Then, when all the modules work, their object modules can be linked together to form a large, functioning program. Also, the object modules for useful programs — a square root program, for example — can be kept in a *library file* and linked into other programs as needed.

NOTE: On IBM PC-type computers, you must run the LINK program on your .OBJ file, even if it contains only one assembly module.

The linker produces a *link file* which contains the binary codes for all the combined modules. The linker also produces a *link map* file which contains the address information about the linked files. The linker, however, does not assign absolute addresses to the program; it assigns only relative addresses starting from zero. This form of the program is said to be *relocatable* because it can be put anywhere in memory to be run. The linkers which come with the TASM or MASM assemblers produce link files with the .EXE extension.

If your program does not require any external hardware, you can use a program called a *debugger* to load and run the .EXE file. We will tell you more about debuggers later. The debugger program which loads your program into memory automatically assigns physical starting addresses to the segments.

If you are going to run your program on a system such as an SDK-86 board, then you must use a *locator program* to assign physical addresses to the segments in the .EXE file.

Locator

A *locator* is a program used to assign the specific addresses of where the segments of object code are to be loaded into memory. A locator program called EXE2BIN comes with the IBM PC Disk Operating System (DOS). EXE2BIN converts a .EXE file to a .BIN file which has physical addresses. You can then use the SDKCOM1 program from Chapter 13 to download the .BIN file to the SDK-86 board. The SDKCOM 1 program can also be used to run the program and debug it on the SDK-86 board.

Debugger

If your program requires no external hardware or requires only hardware accessible directly from your microcomputer, then you can use a *debugger* to run and debug your program. A debugger is a program which allows you to load your object code program into system memory, execute the program, and troubleshoot or “debug” it. The debugger allows you to look at the contents of registers and memory locations after your program runs. It allows you to change the contents of registers and memory locations and rerun the program. Some debuggers allow you to stop execution after each instruction so that you can check or alter memory and register contents. A debugger also allows you to set a *breakpoint* at any point in your program. If you insert a breakpoint, the debugger will run the program up to the instruction where you put

the breakpoint and then stop execution. You can then examine register and memory contents to see whether the results are correct at that point. If the results are correct, you can move the breakpoint to a later point in the program. If the results are not correct, you can check the program up to that point to find out why they are not correct.

The point here is that the debugger commands help you to quickly find the source of a problem in your program. Once you find the problem, you can then cycle back and correct the algorithm if necessary, use the editor to correct your source program, reassemble the corrected source program, relink, and run the program again.

A basic debugger comes with the DOS for most IBM PC-type computers, but more powerful debuggers such as Borland's Turbo Debugger and Microsoft's Codeview debugger make debugging much easier because they allow you to directly see the contents of registers and memory locations change as a program executes. In a later chapter we show you how to use one of these debuggers.

Microprocessor prototyping boards such as the SDK-86 contain a debugger program in ROM. On boards such as this, the debugger is commonly called a *monitor program* because it lets you monitor program activity. The SDK-86 monitor program, for example, lets you enter and run programs, single-step through programs, examine register and memory contents, and insert breakpoints.

Emulator

Another way to run your program is with an *emulator*, such as that shown in Fig. 3.17. An emulator is a mixture of hardware and software. It is usually used to test and debug the hardware and software of an external system, such as the prototype of a microprocessor-based instrument. Part of the hardware of an emulator is a multiwire cable which connects the host system to the system being developed. A plug at the end of the cable is plugged into the prototype system in place of its microprocessor. Through this connection the software of the emulator allows you to download your object code program into RAM in the system being tested and run it. Like a debugger, an emulator allows you to load and run programs, examine and change the contents of registers, examine and change the contents of memory locations, and insert breakpoints in the program. The emulator also takes a "snapshot" of the contents of registers, activity on the address and data bus, and the state of the flags as each instruction executes. The emulator stores this *trace data*, as it is called, in a large RAM. You can do a printout of the trace data to see the results that your program produced on a step-by-step basis.

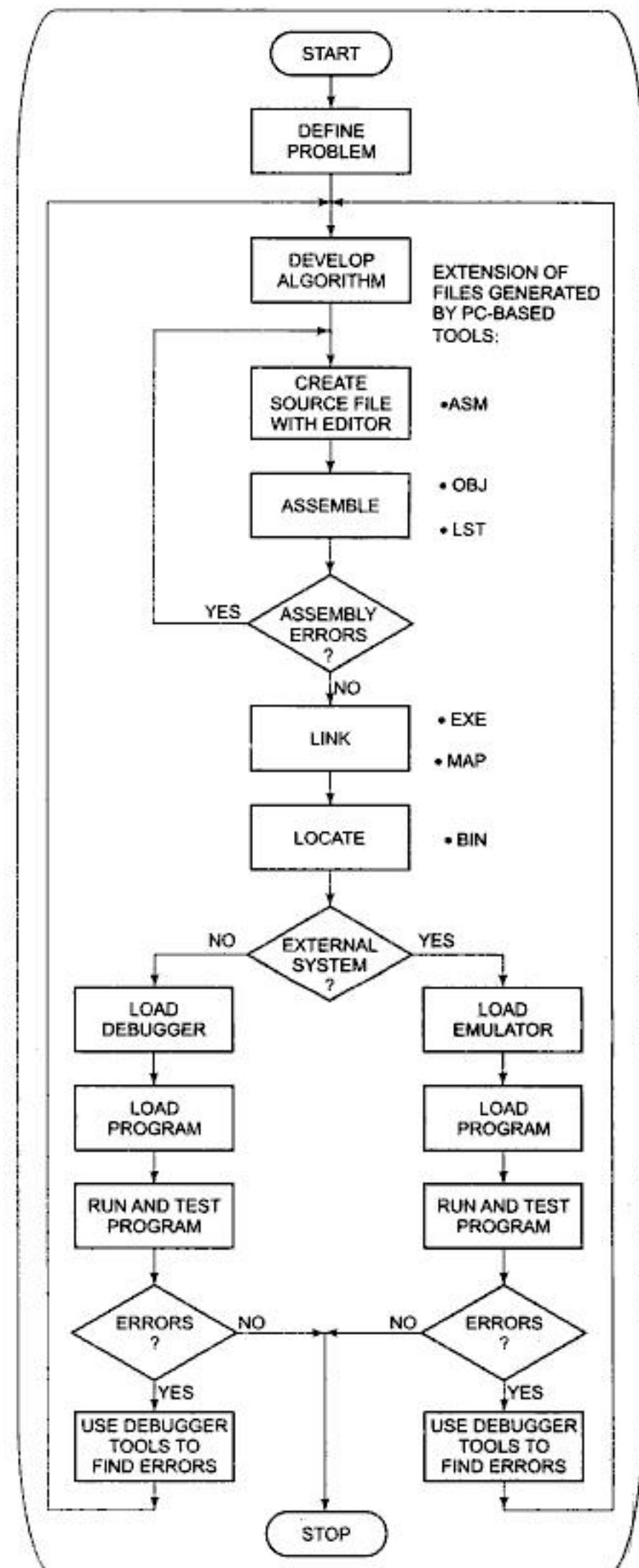


Fig. 3.18 Program development algorithm

Another powerful feature of an emulator is the ability to use either system memory or the memory on the prototype for the program you are debugging. In a later chapter we discuss in detail the use of an emulator in developing a microprocessor-based product.

Summary of the Use of Program Development Tools

Figure 3.18 summarizes the steps in developing a working program. This may seem complicated, but if you use the accompanying lab manual to go through the process a couple of times, you will find that it is quite easy.

The first and most important step is to think out very carefully what you want the program to do and how you want the program to do it. Next, use an editor to create the source file for your program. Assemble the source file. If the assembler list file indicates any errors in your program, use the editor to correct these errors. Cycle through the edit-assemble loop until the assembler tells you on the listing that it found no errors. If your program consists of several modules, then use the linker to join their object modules into one large object module. If your system requires it, use a locate program to specify where you want your program to be put in memory. Your program is now ready to be loaded into memory and run. Note that Fig. 3.18 also shows the extensions for the files produced by each of the development programs.

If your program does not interact with any external hardware other than that connected directly to the system, then you can use the system debugger to run and debug your program. If your program is intended to work with external hardware, such as the prototype of a microprocessor-based instrument, then you will probably use an emulator to run and debug your program. We will

be discussing and showing the use of these program development tools throughout the rest of this book.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms or concepts in the following list, use the index to find them in the chapter.

Algorithm

Flowcharts and flowchart symbols

Structured programming

Pseudocode

Top-down and bottom-up design methods

Sequence, repetition, and decision operations

SEQUENCE, IF-THEN-ELSE, IF-THEN, nested IF-THEN-ELSE, CASE, WHILE-DO, REPEAT-UNTIL programming structures

8086 instructions: MOV, IN, OUT, ADD, ADC, SUB, SBB, AND, OR, XOR, MUL, DIV

Instruction mnemonics

Initialization list

Assembly language program format

Instruction template: W bit, MOD, R/M, D bit

Segment-override prefix

Assembler directives: SEGMENT, ENDS, END, DB, DW, DD, EQU, ASSUME

Accessing named data items

Editor

Assembler

Linker: library file, link files, link map, relocatable

Locator

Debugger, monitor program

Emulator, trace data

REVIEW QUESTIONS AND PROBLEMS

- List the major steps in developing an assembly language program.
- What is the main advantage of a top-down design approach to solving a programming problem?
- Why should you develop a detailed algorithm for a program before writing down any assembly language instructions?
- a. What are the three basic structure types used to write the algorithm for a program?
b. What is the advantage of using only these structures when writing the algorithm for a program?
- A program is like a recipe. Use a flowchart or pseudocode to show the algorithm for the following recipe. The operations in it are sequence and repetition. Instead of implementing the resulting algorithm in assembly language, implement it in your microwave and use the result to help you get through the rest of the book.
Peanut Brittle:
1 cup sugar 1 teaspoon butter
0.5 cup white corn syrup 1 teaspoon vanilla
1 cup unsalted peanuts 1 teaspoon baking soda
i. Put sugar and syrup in 1.5-quart casserole (with handle) and stir until thoroughly mixed.

- ii. Microwave at HIGH setting for 4 minutes.
 - iii. Add peanuts and stir until thoroughly mixed.
 - iv. Microwave at HIGH setting for 4 minutes. Add butter and vanilla, stir until well mixed, and microwave at HIGH setting for 2 more minutes.
 - v. Add baking soda and gently stir until light and foamy. Pour mixture onto nonstick cookie sheet and let cool for 1 hour. When cool, break into pieces. Makes 1 pound.
6. Use a flowchart or pseudocode to show the algorithm for a program which gets a number from a memory location, subtracts 20H from it, and outputs 01H to port 3AH if the result of the subtraction is greater than 25H.
7. Given the register contents in Fig. 3.19, answer the following questions:
- What physical address will the next instruction be fetched from?
 - What is the physical address for the top of the stack?

| DATA SEGMENT | | | | | | | | | |
|--------------|------|----|--------|----|----|--|--|--|--|
| ES | 6000 | | 5000CH | 07 | | | | | |
| CS | 4000 | | 5000BH | 9A | | | | | |
| SS | 7000 | | 5000AH | 7C | | | | | |
| DS | 5000 | | 50009H | DB | | | | | |
| IP | 43E8 | | 50008H | C3 | | | | | |
| SP | 0000 | | 50007H | B2 | | | | | |
| BP | 2468 | | 50006H | 49 | | | | | |
| SI | 4000 | | 50005H | 21 | | | | | |
| DI | 7000 | | 50004H | 89 | | | | | |
| | | | 50003H | 71 | | | | | |
| | | | 50002H | 22 | | | | | |
| | | | 50001H | 4A | | | | | |
| | | | 50000H | 3B | | | | | |
| AH | AL | | BH | BL | | | | | |
| AX | 42 | 35 | BX | 07 | 5A | | | | |
| CH | CL | | DH | DL | | | | | |
| CX | 00 | 04 | DX | 33 | 02 | | | | |

Fig. 3.19 8086 register and memory contents for Problems 7, 8, and 10.

8. Describe the operation and results of each of the following instructions, given the register contents shown in Fig. 3.19. Include in your answer the physical address or register that each instruction will get its operands from and the physical address or register in which each instruction will put the result. Use the instruction descriptions in Chapter 6 to help you. Assume that the following instructions are independent, not sequential, unless listed together under a letter.

- a. MOV AX, BX
 - b. MOV CL, 37H
 - c. INC BX
 - d. MOV CX, [246BH]
 - e. MOV CX, 246BH
 - f. ADD AL, DH
 - g. MUL BX
 - h. DEC BP
 - i. DIV BL
 - j. SUB AX, DX
 - k. OR CL, BL
 - l. NOT AH
 - m. ROL BX, 1
 - n. AND AL, CH
 - o. MOV DS, AX
 - p. ROR BX, CL
 - q. AND AL, OFH
 - r. MOV, AX, [BX]
 - s. MOV [BX] [SI], CL
9. See if you can spot the grammatical (syntax) errors in the following instructions (use Chapter 6 to help you):
- a. MOV BH, AX
 - b. MOV DX, CL
 - c. ADD AL, 2073H
 - d. MOV 7632H, CX
 - e. IN BL, 04H
10. Show the results that will be in the affected registers or memory locations after each of the following groups of instructions executes. Assume that each group of instructions starts with the register and memory contents shown in Fig. 3.19. (Use Chapter 6.)
- a. ADD BL, AL
 - b. MOV CL, 04
 - c. ADD AL, BH
 - d. MOV BX, 000AH
 - e. MOV [00041, BL]
 - f. ROR DI, 04
 - g. SUB AL, CL
 - h. INC BX
 - i. DAA
 - j. MOV [BX], AL
11. Write the 8086 instruction which will perform the indicated operation. Use the instruction overview in this chapter and the detailed descriptions in Chapter 6 to help you.
- a. Copy AL to BL.
 - b. Load 43H into CL.
 - c. Increment the contents of CX by 1.
 - d. Copy SP to BP.
 - e. Add 07H to DL.
 - f. Multiply AL times BL.
 - g. Copy AX to a memory location at offset 245AH in the data segment.
 - h. Decrement SP by 1.
 - i. Rotate the most significant bit of AL into the least significant bit position.
 - j. Copy DL to a memory location whose offset is in BX.
 - k. Mask the lower 4 bits of BL.
 - l. Set the most significant bit of AX to a 1, but do not affect the other bits.
 - m. Invert the lower 4 bits of BL, but do not affect the other bits.

12. Construct the binary code for each of the following 8086 instructions.
- MOV BL, AL
 - MOV [BX], CX
 - ADD BX, 59H[DI]
 - SUB [2048], DH
 - XCHG CH, ES:[BX]
 - ROR AX, 1
 - OUT DX, AL
 - AND AL, OFH
 - NOP
 - IN AL, DX
13. Describe the function of each assembler directive and instruction statement in the short program shown in Fig. 3.20.
14. Describe how an assembly language program is developed and debugged using system tools such as editors, assemblers, linkers, locators, emulators, and debuggers.
15. Write the pseudocode representation for the flowchart in Fig. 3.18.

```
;PRESSURE READ PROGRAM

DATA_HERE SEGMENT
    PRESSURE DB 0      ;storage for pressure
DATA_HERE ENDS

PRESSURE_PORT EQU 04H ;Pressure sensor connected
                      ;to port 04H
CORRECTION_FACTOR EQU 07H ;Current correction factor
                           ;of 07

CODE_HERE SEGMENT
    ASSUME CS:CODE_HERE, DS:DATA_HERE
    MOV AX, DATA_HERE
    MOV DS, AX
    IN AL, PRESSURE_PORT
    ADD AL, CORRECTION_FACTOR
    MOV PRESSURE, AL
CODE_HERE ENDS
END
```

Fig. 3.20 Program for Problem 13.

4

Implementing Standard Program Structures In 8086 Assembly Language

In Chapter 3 we worked very hard to convince you that you should not try to write programs directly in assembly language. The analogy of building a house without a plan should come to mind here. When faced with a programming problem, you should solve the problem and write the algorithm for the solution using the standard program structures we described. Then you simply translate each step in the flowchart or pseudocode to a group of one to four assembly language instructions which will implement that step. The comments in the assembly language program should describe the functions of each instruction or group of instructions, so you essentially write the comments for the program, then write the assembly language instructions which implement those comments. Once you learn how to implement each of the standard programming structures, you should find it quite easy to translate algorithms to assembly language. Also, as we will show you, the standard structure approach makes debugging relatively easy.

The purposes of this chapter are to show you how to write the algorithms for some common programming problems, how to implement these algorithms in 8086 assembly language, and how to systematically debug assembly language programs. In the process you will also learn more about how some of the 8086 instructions work.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Write flowcharts or pseudocode for simple programming problems.
2. Implement SEQUENCE, IF-THEN-ELSE, WHILE-DO, and REPEAT-UNTIL program structures in 8086 assembly language.
3. Describe the operation of selected data transfer, arithmetic, logical, jump, and loop instructions.
4. Use based and indexed addressing modes to access data in your programs.
5. Describe a systematic approach to debugging a simple assembly language program using debugger, monitor, or emulator tools.
6. Write a delay loop which produces a desired amount of delay on a specific 8086 system.

SIMPLE SEQUENCE PROGRAMS

Finding the Average of Two Numbers

DEFINING THE PROBLEM AND WRITING THE ALGORITHM

A common need in programming is to find the average of two numbers. Suppose, for example, we know the maximum temperature and the minimum temperature for

a given day, and we want to determine the average temperature. The sequence of steps we go through to do this might look something like the following.

Add maximum temperature and minimum temperature.

Divide sum by 2 to get average temperature.

This sequence doesn't look much like an assembly language program, and it shouldn't. The algorithm at this point should be general enough that it could be implemented in any programming language, or on any machine. Once you are reasonably sure of your algorithm, then you can start thinking about the architecture and instructions of the specific microcomputer on which you plan to run the program. Now let's show you how we get from the algorithm to the assembly language program for it.

SETTING UP THE DATA STRUCTURE

One of the first things for you to think about in this process is the data that the program will be working with. You need to ask yourself questions such as:

1. Will the data be in memory or in registers?
2. Is the data of type byte, type word, or perhaps type doubleword?
3. How many data items are there?
4. Does the data represent only positive numbers, or does it represent positive and negative (signed) numbers?
5. For more complex problems, you might ask how the data is structured. For example, is the data in an array or in a record?

Let's assume for this example that the data is all in memory, that the data is of type byte, and that the data represents only positive numbers in the range 0 to 0FFH. The top part of Fig. 4.1, between the DATA SEGMENT and the DATA ENDS directives, shows how you might set up the data structure for this program. It is very similar to the data structure for the multiplication example in the last chapter. In the logical segment called DATA, HI_TEMP is declared as a variable of type byte and initialized with a value of 92H. In an actual application, the value in HI_TEMP would probably be put there by another program which reads the output from a temperature sensor. The statement LO_TEMP DB 52H declares a variable of type byte and initializes it with the value 52H. The statement AV_TEMP DB ? sets aside a byte location to store the average temperature, but does not initialize the location to any value. When the program executes, it will write a value to this location.

INITIALIZATION CHECKLIST

Although it does not show in the algorithm, you know from the discussion in Chapter 3 that most programs start with a series of initialization instructions. For this example program, all you have to initialize is the data segment register. The MOV AX,DATA and MOV DS,AX instructions at the start of the program in Fig. 4.1 do this.

These instructions load the DS register with the upper 16 bits of the starting address for the data segment. If you are using an assembler, you can use the name DATA in the instruction to refer to this address. If you are not using an assembler, then just put the hex for the upper 16 bits of the address in the MOV AX,DATA instruction in place of the name.

CHOOSING INSTRUCTIONS TO IMPLEMENT THE ALGORITHM

The next step is to look at the algorithm to determine the major actions that you want the program to perform. If you have written the algorithm correctly, then all you should have to do is translate each step in the algorithm to one to four assembly language instructions which will implement that step.

You want the program to add two byte-type numbers together, so scan through the instruction groups in Chapter 3 to determine which 8086 instruction will do this for you. The ADD instruction is the obvious choice in this case.

Next, find and read the detailed discussion of the ADD instruction in Chapter 6. From the discussion there, you can determine how the instruction works and see if it will do the necessary job. From the discussion of the ADD instruction, you should find that the ADD instruction has the format ADD destination, source. A byte from the specified source is added to a byte in the specified destination, or a word from the specified source is added to a word in the specified destination. (Note that you cannot directly add a byte to a word.) The result in either case is put in the specified destination. The source can be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, in a single instruction the source and the destination cannot both be memory locations. This means that you have to move one of the operands from memory to a register before you can do the ADD.

Another point to consider here is that if you add two 8-bit numbers, the sum can be larger than 8 bits. Adding F0H and 40H, for example, gives 130H. The 8-bit destination will contain 30H, and the carry will be held in

the carry flag. This means that to have the complete sum, you must collect the parts of the result in a location large enough to hold all 9 bits. A 16-bit register is a good choice.

To summarize, then, you need to move one of the numbers you want to add into a register, such as AL, add the other number from memory to it, and move any carry produced by the addition to the upper half of the 16-bit register which contains the sum in its lower 8 bits. Now let's take another look at Fig. 4.1 to see how you implement this step in the algorithm with 8086 instructions.

The instruction `MOV AL,HI_TEMP` copies one of the temperatures from a memory location to the AL register. The name `HI_TEMP` in the instruction represents the direct address or displacement of the variable in the logical segment DATA. The `ADD AL,LO_TEMP` instruction adds the specified byte from memory to the contents of the AL register. The lower 8 bits of the sum are left in the AL register. If the addition produces a result greater than FFH, the carry flag will be set to a 1. If the addition produces a result less than or equal to FFH, the carry flag will be a 0. In either case, we want to get the contents of the carry flag into the least significant bit of the AH register, so that the entire sum is in the AX register.

The `MOV AH,00H` instruction clears all the bits of AH to 0's. The `ADC AH,00H` instruction adds the immediate

number 00H plus the contents of the carry flag to the contents of the AH register. The result will be left in the AH register. Since we cleared AH to all 0's before the add, what we are really adding is 00H + 00H + CF. The result of all this is that the carry bit ends up in the least significant bit of AH, which is what we set out to do.

The next major action in our algorithm is to divide the sum of the two temperatures by 2. To determine how this step can be translated to assembly language instructions, look at the instruction groups in the last chapter to see if the 8086 has a Divide instruction. You should find that it has two Divide instructions, DIV and IDIV. DIV is for dividing unsigned numbers, and IDIV is used for dividing signed binary numbers. Since in this example we are dividing unsigned binary numbers, look up the DIV instruction in Chapter 6 to find out how it works.

The DIV instruction can be used to divide a 16-bit number in AX by a specified byte in a register or in a memory location. After the division, an 8-bit quotient is left in the AL register, and an 8-bit remainder is left in the AH register. The DIV instruction can also be used to divide a 32-bit number in the DX and AX registers by a 16-bit number from a specified register or memory location. In this case, a 16-bit quotient is left in the AX register, and a

```

1 ; 8086 PROGRAM      F4-01.ASM
2 ;ABSTRACT : This program averages two temperatures
3 ;           named HI_TEMP and LO_TEMP and puts the
4 ;           result in the memory location AV_TEMP.
5 ;REGISTERS : Uses DS, CS, AX, BL
6 ;PORTS    : None used
7
8 0000          DATA   SEGMENT
9 0000 92        HI_TEMP DB 92H    ; Max temp storage
10 0001 52       LO_TEMP DB 52H    ; Low temp storage
11 0002 ??       AV_TEMP DB ?     ; Store average here
12 0003          DATA   ENDS
13
14 0000          CODE   SEGMENT
15
16 0000 B8 0000s ASSUME CS:CODE, DS:DATA
17 0003 8E D8
18 0005 A0 0000r START: MOV AX, DATA      ; Initialize data segment
19 0008 02 06 0001r MOV DS, AX
20 000C B4 00     MOV AL, HI_TEMP   ; Get first temperature
21 000E 80 D4 00   ADD AL, LO_TEMP   ; Add second to it
22 0011 B3 02     MOV AH, 00H      ; Clear all of AH register
23 0013 F6 F3     ADC AH, 00H      ; Put carry in LSB of AH
24
25 0015 A2 0002r MOV BL, 02H      ; Load divisor in BL register
26 0018           DIV BL          ; Divide AX by BL. Quotient in AL,
27                         ; and remainder in AH
                           MOV AV_TEMP, AL   ; Copy result to memory
                           CODE   ENDS
                           END   START

```

Fig. 4.1 8086 program to average two temperatures.

16-bit remainder is left in the DX register. In either case, there is a problem if the quotient is too large to fit in AX for a 32-bit divide or AL for a 16-bit divide. Fortunately, the data in the example here is such that the problem will not arise. In a later chapter we discuss what to do about this problem.

Remember from the previous discussion that the sum of the two temperatures is already positioned in the AX register as required by the DIV operation. Before we can do the DIV operation, however, we have to get the divisor, 02H, into a register or memory location to satisfy the requirements of the DIV instruction. A simple way to do this is with the MOV BL,02H instruction, which loads the immediate number 02H into the BL register. Now you can do the divide operation with the instruction DIV BL. The 8-bit quotient from the division will be left in the AL register.

The algorithm doesn't show it, but in our discussion of the data structure we said that the minimum, maximum, and average temperatures were all in memory locations. Therefore, to complete the program, you have to copy the quotient in AL to the memory location we set aside for the average temperature. As shown in Fig. 4.1, the instruction MOV AV_TEMP,AL will copy AL to this memory location.

NOTE: We could have used the remainder from the division in AH to round off the average temperature to the nearest degree, but that would have made the program more complex than we wanted for this example.

SUMMARY OF CONVERTING AN ALGORITHM TO ASSEMBLY LANGUAGE

The first step in converting an algorithm to assembly language is to set up the data structure that the algorithm will be working with. The next step is to write at the start of the code segment any instructions required to initialize variables, segment registers, peripheral devices, etc. Then determine the instructions required to implement each of the major actions in the algorithm, and decide how the data must be positioned for these instructions. Finally, insert the MOV or other instructions required to get the data into the correct position for these instructions.

A Few Comments about the 8086 Arithmetic Instructions

The 8086 has instructions to add, subtract, multiply, and divide. It can operate on signed or unsigned binary numbers, BCD numbers, or numbers represented in ASCII. Rather than put a lot of arithmetic examples at this

point in the book, we show arithmetic examples with each arithmetic instruction description in Chapter 6. The description of the MUL instruction in Chapter 6, for example, shows how unsigned binary numbers are multiplied. Also we show other arithmetic examples as needed throughout the rest of the book. If you need to do some arithmetic operations with an 8086, there are a few instructions in addition to the basic add, subtract, multiply, and divide instructions that you need to look up in Chapter 6.

If you are adding BCD numbers, you need to also look up the Decimal Adjust for Addition (DAA) instruction. If you are subtracting BCD numbers, then you need to look up the Decimal Adjust for Subtraction (DAS) instruction. If you are working with ASCII numbers, then you need to look up the ASCII Adjust after Addition (AAA) instruction, the ASCII Adjust after Subtraction (AAS) instruction, the ASCII Adjust after Multiply (AAM) instruction, and the ASCII Adjust before Division (AAD) instruction.

Debugging Assembly Language Programs

By now you should be writing some programs of your own, so we need to give you a few hints on how to debug them if they don't work correctly the first time you try to run them.

The first technique you use when you hit a difficult-to-find problem in either hardware or software is the *5-minute rule*. This rule says, "You get 5 minutes to freak out and mumble about changing vocations, then you have to cope with the problem in a systematic manner." What this means is step back from the problem, collect your wits, and think out a systematic series of steps to find the solution. Random poking and probing wastes a lot of valuable time and seldom finds the problem. Here is a list of additional techniques you may find useful in writing and debugging your programs.

1. Very carefully define the problem you are trying to solve with the program and work out the best algorithm you can.
2. Write and test each section of a program as you go, instead of writing a large program all at once.
3. If a program or program section does not work, first recheck the algorithm to make sure it really does what you want it to. You might have someone else look at it also. Another person may quickly spot an error you have overlooked 17 times.
4. If the algorithm seems correct, check to make sure that you have used the correct instructions to

- implement the algorithm. It is very easy to accidentally switch the operands in an instruction. You might, for example, write down the instruction MOV AX.DX when the instruction you really want is MOV DX, AX. Sometimes it helps to work out on paper the effect that a series of instructions will have on some sample numbers. These predictions can later be compared with the actual results produced when the program section runs.
5. If you are hand coding your programs, this is the next place to check. It is very easy to get a bit wrong when you construct the 8086 instruction codes. Also remember, when constructing instruction codes which contain addresses or displacements, that the low byte of the address or displacement is coded in before the high byte.
 6. If you don't find a problem in the algorithm, instructions, or coding, now is the time to use debugger, monitor, or emulator tools to help you localize the problem. You could use these tools right from the start, but if you do, it is easy to get lost in chasing bits and not see the bigger picture of what is causing the program to fail. When debugging short program sections on an SDK-86 board, for example, you might use the *single-step* command to help you determine why the program is not doing what you want it to do. The SDK-86 board's single-step command executes one instruction and then stops execution. You can then use the Examine Register and Examine Memory commands to see if registers and memory contain the correct data. If the results are correct at that point, you can use the single-step command to execute the next instruction. You keep stepping through the program until you reach a point where the results are not what you predicted they should be at that point. Once you have localized the problem to one or two instructions, it is usually not too hard to find the error. An exercise in the accompanying lab manual shows you how to use the single-step command on an SDK-86 board.
 7. For longer programs, the single-step approach can be somewhat tedious. *Breakpoints* are often a faster technique to narrow the source of a problem down to a small region. Most debuggers, monitors, and emulators allow you to specify both a starting address and an ending address in their GO command. The SDK-86 monitor GO command, for example, has the format GO address, breakpoint address. When you enter one of these commands, execution will start at the address specified first in the

command and stop when it reaches the address specified in the second position in the command. After the program runs to a breakpoint, you can use the Examine Register and Examine Memory commands to check the results at that point.

Here's how you use breakpoints. Instead of running the entire program, specify a breakpoint so that execution stops some distance into the program. You can then check to see if the results are correct at this point. If they are, you can run the program again with the breakpoint at a later address and check the results at that point. If the results are not correct, you can move the breakpoint to an earlier point in the program, run it again, and check whether the results in registers and memory are correct.

Suppose, for example, you write a program such as the averaging program in Fig. 4.1, and it does not give the correct results. The first place to put a breakpoint might be at the address of the MOV AH,00 instruction. Incidentally, in most systems the instruction at the address where you put the breakpoint does not get executed. After the program runs to this breakpoint, you check to see if the data segment register was initialized correctly and if the basic addition was performed correctly. If the program works correctly to this point, you can run it again with the breakpoint at the address of the MOV AV_TEMP,AL instruction. After the program executes to this breakpoint, you can check AL to see if the division produced the results you predicted. If the 8086 is working at all, it will almost always do operations such as this correctly, so recheck your predictions if you disagree with it.

It helps your frustration level if you make a game of thinking where to put breakpoints to track down the little bug that is messing up your program. With a little practice you should soon develop an efficient debugging algorithm of your own using the specific tools available on your system. In the next chapter we show you how to use a more powerful debugger to run and debug programs in an IBM PC-type computer.

Converting Two ASCII Codes to Packed BCD

DEFINING THE PROBLEM AND WRITING THE ALGORITHM

Computer data is often transferred as a series of 8-bit ASCII codes. If, for example, you have a microcomputer connected to an SDK-86 board and you type a 9 on an ASCII-encoded computer terminal keyboard, the 8-bit ASCII code sent to the SDK-86 will be 00111001 binary, or 39H. If you type a 5 on the keyboard, the code sent to

the computer will be 00110101 binary or 35H, the ASCII code for 5. As shown in Table 1.2, the ASCII codes for the numbers 0 through 9 are 30H through 39H. The lower nibble of the ASCII codes contains the 4-bit BCD code for the decimal number represented by the ASCII code.

For many applications, we want to convert the ASCII code to its simple BCD equivalent. We can do this by simply replacing the 3 in the upper nibble of the byte with four 0's. For example, suppose we read in 00111001 binary or 39H, the ASCII code for 9. If we replace the upper 4 bits with 0's, we are left with 00001001 binary or 09H. The lower 4 bits then contain 1001 binary, the BCD code for 9. Numbers represented as one BCD digit per byte are called *unpacked BCD*.

For applications in which we are going to perform mathematical operations on the BCD numbers, we usually combine two BCD digits in a single byte. This form is called *packed BCD*. Fig. 4.2 shows examples of ASCII, unpacked BCD, and packed BCD. The problem we are going to work on here is how to convert two numbers from ASCII code form to unpacked BCD form and then pack the two BCD digits into one byte. Fig. 4.2 shows in numerical form the steps we want the program to perform. When you are writing a program which manipulates data such as this, a numerical example will help you visualize the algorithm.

| | | | |
|-----------------------|----|------|------------|
| ASCII | 5 | 0011 | 0101 = 35H |
| ASCII | 9 | 0011 | 1001 = 39H |
| UNPACKED BCD | 5 | 0000 | 0101 = 05H |
| UNPACKED BCD | 9 | 0000 | 1001 = 09H |
| UNPACKED BCD | 5 | 0101 | 0000 = 50H |
| MOVED TO UPPER NIBBLE | | | |
| PACKED BCD | 59 | 0101 | 1001 = 59H |

Fig. 4.2 ASCII, unpacked BCD, and packed BCD examples.

The algorithm for this problem can be stated simply as Convert first ASCII number to unpacked BCD.

Convert second ASCII number to unpacked BCD.

Move first BCD nibble to upper nibble position in byte.

Pack two BCD nibbles in one byte.

Now let's see how you can implement this algorithm in 8086 assembly language.

THE DATA STRUCTURE AND INITIALIZATION LIST

For this example program, let's assume that the ASCII

code for 5 was received and put in the BL register, and the second ASCII code was received and left in the AL register. Since we are not using memory for data in this program, we do not need to declare a data segment or initialize the data segment register. Incidentally, in a real application this program would probably be a procedure or a part of a larger program.

MASKING WITH THE AND INSTRUCTION

The first operation in the algorithm is to convert a number in ASCII form to its unpacked BCD equivalent. This is done by replacing the upper 4 bits of the ASCII byte with four 0's. The 8086 AND instruction can be used to do this operation. Remember from basic logic or from the review in Chapter 1 that when a 1 or a 0 is ANDed with a 0, the result is always a zero. ANDing a bit with a 0 is called *masking* that bit because the previous state of the bit is hidden or masked. To mask 4 bits in a word, then, all you do is AND each bit you want to mask with a 0. A bit ANDed with a 1, remember, is not changed.

According to the description of the AND instruction in Chapter 6, the instruction has the format AND destination source. The instruction ANDs each bit of the specified source with the corresponding bit of the specified destination and puts the result in the specified destination. The source can be an immediate number, a register, or a memory location specified in one of those 24 different ways. The destination can be a register or a memory location. The source and the destination must both be bytes, or they must both be words. The source and the destination cannot both be memory locations in an instruction.

For this example the first ASCII number is in the BL register, so we can just AND an immediate number with this register to mask the desired bits. The upper 4 bits of the immediate number should be 0's because these correspond to the bits we want to mask in BL. The lower 4 bits of the immediate number should be 1's because we want to leave these bits unchanged. The immediate number, then, should be 00001111 binary or 0FH. The instruction to convert the first ASCII number is AND BL,0FH. When this instruction executes, it will leave the desired unpacked BCD in BL. Fig. 4.3 shows how this will work for an ASCII number of 35H initially in BL.

| | | |
|---------|------|------|
| ASCII 5 | 0011 | 0101 |
| MASK | 0000 | 1111 |
| RESULT | 0000 | 0101 |

Fig. 4.3 Effects of ANDing with 1's and 0's.

For the next action in the algorithm, we want to perform the same operation on a second ASCII number

in the AL register. The instruction AND AL, OFH will do this for us. After this instruction executes, AL will contain the unpacked BCD for the second ASCII number.

MOVING A NIBBLE WITH THE ROTATE INSTRUCTION

The next action in the algorithm is to move the 4 BCD bits in the first unpacked BCD byte to the upper nibble position in the byte. We need to do this so that the 4 BCD bits are in the correct position for packing with the second BCD nibble. Take another look at Fig. 4.2 to help you visualize this. What we are effectively doing here is swapping or exchanging the top nibble with the bottom nibble of the byte. If you check the instruction groups in Chapter 3, you will find that the 8086 has an Exchange instruction, XCHG, which can be used to swap two bytes or to swap two words. The 8086 does not have a specific instruction to swap the nibbles in a byte. However, if you think of the operation that we need to do as shifting or rotating the BCD bits 4 bit positions to the left, this will give you a good idea which instruction will do the job for you. The 8086 has a wide variety of rotate and, shift instructions. For now, let's look at the rotate instructions. There are two instructions, ROL and RCL, which rotate the bits of a specified operand to the left. Figure 4.4 shows in diagram form how these two instructions work. For ROL, each bit in the specified register or memory location is rotated 1 bit position to the left. The bit that was the MSB is rotated around into the LSB position. The old MSB is also copied to the carry flag. For the RCL instruction,

each bit of the specified register or memory location is also rotated 1 bit position to the left. However, the bit that was in the MSB position is moved to the carry flag, and the bit that was in the carry flag is moved into the LSB position. The C in the middle of the mnemonic should help you remember that the carry flag is included in the rotated loop when the RCL instruction executes.

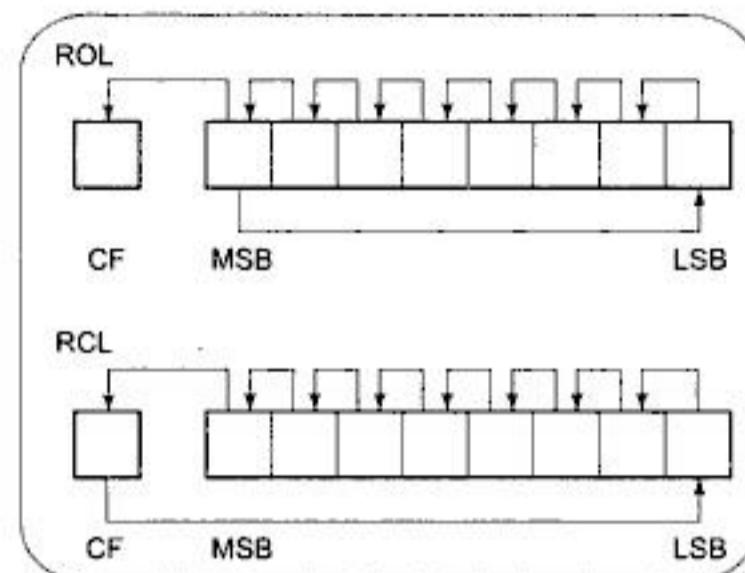


Fig. 4.4 ROL instruction and RCL instruction operations for byte operands.

In the example program we really don't want the contents of the carry flag rotated into the operand, so the ROL instruction seems to be the one we want. If you consult the ROL instruction description in Chapter 6, you will find that the instruction has the format ROL destination, count. The destination can be a register or a memory location. It can be a byte location or a word

```

1 ; 8086 PROGRAM F4-05.ASM
2 ;ABSTRACT : Program produces a packed BCD byte from 2 ASCII-encoded digits
3 ; The first ASCII digit (5) is loaded in BL.
4 ; The second ASCII digit (9) is loaded in AL.
5 ; The result (packed BCD) is left in AL
6 ;REGISTERS ; Uses CS, AL, BL, CL
7 ;PORTS : None used
8
9 0000           CODE   SEGMENT
10              ASSUME CS:CODE
11 0000  B3 35    START: MOV   BL, '5'   ; Load first ASCII digit into BL
12 0002  B0 39    MOV   AL, '9'   ; Load second ASCII digit into AL
13 0004  80 E3 0F  AND   BL, OFH   ; Mask upper 4 bits of first digit
14 0007  24 0F    AND   AL, OFH   ; Mask upper 4 bits of second digit
15 0009  B1 04    MOV   CL, 04H   ; Load CL for 4 rotates required
16 000B  D2 C3    ROL   BL, CL    ; Rotate BL 4 bit positions
17 000D  0A C3    OR    AL, BL    ; Combine nibbles, result in AL
18 000F           CODE   ENDS
19             END START

```

Fig. 4.5 List file of 8086 assembly language program to produce packed BCD from two ASCII characters.

location. The count can be the immediate number 1 specified directly in the instruction, or it can be a number previously loaded into the CL register. The instruction ROL AL, 1, for example, will rotate the contents of AL 1 bit position to the left. We could repeat this instruction four times to produce the shift of 4 bit positions that we need for our BCD packing problem. However, there is an easier way to do it. We first load the CL register with the number of times we want to rotate AL. The instruction MOV CL,04H will do this. Then we use the instruction ROL BL,CL to do the rotation. When it executes, this instruction will automatically rotate BL the number of bit positions loaded into CL. Note that for the 80186 you can write the single instruction ROL BL,04H to do this job.

Now that we have determined the instructions needed to mask the upper nibbles and the instructions needed to move the first BCD digit into position, the only thing left is to pack the upper nibble from BL and the lower nibble from AL into a single byte.

COMBINING BYTES OR WORDS WITH THE ADD OR THE OR INSTRUCTION

You can't use a standard MOV instruction to combine two bytes into one as we need to do here. The reason is that the MOV instruction copies an operand from a specified source to a specified destination. The previous contents of the destination are lost. You can, however, use an ADD or an OR instruction to pack the two BCD nibbles.

As described in the previous program example, the ADD instruction adds the contents of a specified source to the contents of a specified destination and leaves the result in the specified destination. For the example program here, the instruction ADD AL,BL can be used to combine the two BCD nibbles. Take a look at Fig. 4.2 to help you visualize this addition.

Another way to combine the two nibbles is with the OR instruction. If you look up the OR instruction in Chapter 6, you will find that it has the format OR destination, source. This instruction ORs each bit in the specified source with the corresponding bit in the specified destination. The result of the ORing is left in the specified destination. Remember from basic logic or the review in Chapter 1 that ORing a bit with a 1 always produces a result of 1. ORing a bit with a 0 leaves the bit unchanged. To set a bit in a word to a 1, then, all you have to do is OR that bit with a word which has a 1 in that bit position and 0's in all the other bit positions. This is similar to the way the AND instruction is used to clear bits in a word to 0's. See the OR instruction description in Chapter 6 for examples of this.

For the example program here, we use the instruction OR AL,BL to pack the two BCD nibbles. Bits ORed with 0s will not be changed. Bits ORed with 1s will become 1's. Again look at Fig. 4.2 to help you visualize this operation.

SUMMARY OF BCD PACKING PROGRAM

If you compare the algorithm for this program with the finished program in Fig. 4.5, you should see that each step in the algorithm translates to one or two assembly language instructions. As we told you before, developing the assembly language program from a good algorithm is really quite easy because you are simply translating one step at a time to its equivalent assembly language instructions. Also, debugging a program developed in this way is quite easy because you simply single-step or breakpoint your way through it and check the results after each step. In the next section we discuss the 8086 JMP instructions and flags so we can show you how you implement some of the other programming structures in assembly language.

JUMPS, FLAGS, AND CONDITIONAL JUMPS

Introduction

The real power of a computer comes from its ability to choose between two or more sequences of actions based on some condition, repeat a sequence of instructions *as long as* some condition exists, or repeat a sequence of instructions *until* some condition exists. *Flags* indicate whether some condition is present or not. *Jump* instructions are used to tell the computer the address to fetch its next instruction from. Fig. 4.6 shows in diagram form the different ways a Jump instruction can direct the 8086 to fetch its next instruction from some place in memory other than the next sequential location.

The 8086 has two types of Jump instructions, conditional and unconditional. When the 8086 fetches and decodes an Unconditional Jump instruction, it always goes to the specified jump destination. You might use this type of Jump instruction at the end of a program so that the entire program runs over and over, as shown in Fig. 4.6.

When the 8086 fetches and decodes a Conditional Jump instruction, it evaluates the state of a specified flag to determine whether to fetch its next instruction from the jump destination location or to fetch its next instruction from the next sequential memory location.

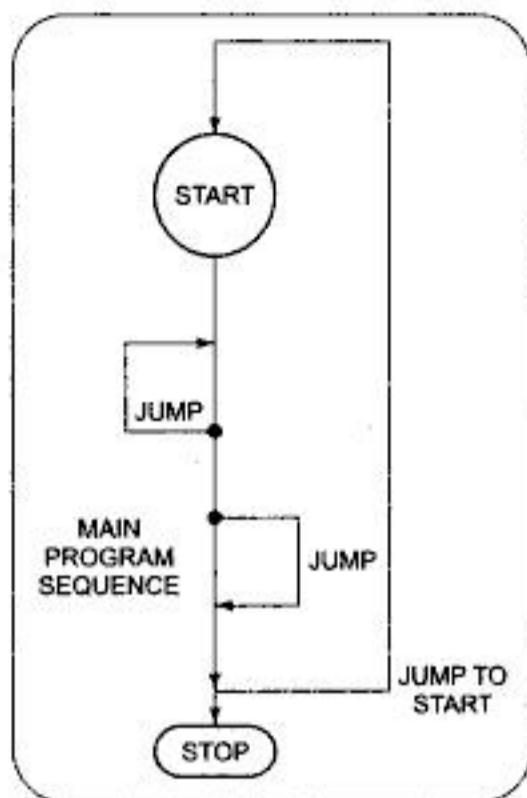


Fig. 4.6 Change in program flow that can be caused by jump instructions.

Let's start by taking a look at how the 8086 Unconditional Jump instruction works.

The 8086 Unconditional Jump Instruction

INTRODUCTION

As we said before, Jump instructions can be used to tell the 8086 to start fetching its instructions from some new location rather than from the next sequential location. The 8086 JMP instruction always causes a jump to occur, so this is referred to as an *unconditional* jump.

Remember from previous discussions that the 8086 computes the physical address from which to fetch its next code byte by adding the offset in the instruction pointer register to the code segment base represented by the 16-bit number in the CS register. When the 8086 executes a JMP instruction, it loads a new number into the instruction pointer register, and in some cases it also loads a new number into the code segment register.

If the JMP destination is in the same code segment, the 8086 only has to change the contents of the instruction pointer. This type of jump is referred to as a *near*, or *intrasegment*, jump.

If the JMP destination is in a code segment which has a different name from the segment in which the JMP instruction is located, the 8086 has to change the contents of both CS and IP to make the jump. This type of jump is referred to as a *far*, or *intersegment*, jump.

Near and far jumps are further described as either *direct* or *indirect*. If the destination address for the jump is specified directly as part of the instruction, then the jump is described as *direct*. You can have a direct near jump or a direct far jump. If the destination address for the jump is contained in a register or memory location, the jump is referred to as *indirect*, because the 8086 has to go to the specified register or memory location to get the required destination address. You can have an indirect near jump or an indirect far jump.

Figure 4.7 shows the coding templates for the four basic types of unconditional jumps. As you can see, for the direct types, the destination offset, and, if necessary, the segment base are included directly in the instruction. The indirect types of jumps use the second byte of the instruction to tell the 8086 whether the destination offset (and segment base, if necessary) is contained in a register or in memory locations specified with one of the 24 address modes we introduced you to in the last chapter.

The JMP instruction description in Chapter 6 shows examples of each type of jump instruction, but in most of your programs you will use a direct near-type JMP instruction, so in the next section we will discuss in detail how this type works.

UNCONDITIONAL JUMP INSTRUCTION TYPES—OVERVIEW

The 8086 Unconditional Jump instruction, JMP, has five different types. Figure 4.7 shows the names and instruction coding templates for these five types. We will first summarize how these five types work to give you an overview; then we will describe in detail the two types you need for your programs at this point. The JMP instruction description in Chapter 6 shows examples of each of the five types.

THE DIRECT NEAR- AND SHORT-TYPE JMP INSTRUCTIONS

As we described previously, a near-type jump instruction can cause the next instruction to be fetched from anywhere in the current code segment. To produce the new instruction fetch address, this instruction adds a 16-bit signed displacement contained in the instruction to the contents of the instruction pointer register. A 16-bit signed displacement means that the jump can be to a location anywhere from +32,767 to -32,768 bytes from the current instruction pointer location. A positive displacement usually means you are jumping ahead in the program, and a negative displacement usually means that you are jumping "backward" in the program.

| JMP = JMP | | | | |
|---|-------------|--|----------|----------|
| Within segment or group, IP relative—near and short | | | | |
| Opcode | DispL | DispH | | |
| E9 | 15 | IP ← IP + Disp16 | | |
| EB | 15 | IP ← IP + Disp8 (Disp8 sign-extended) | | |
| Within segment or group, Indirect | | | | |
| Opcode | mod 100 r/m | mem-low | mem-high | |
| FF | 11 | IP ← Reg16 | | |
| FF | 18 + EA | IP ← Mem16 | | |
| Inter-segment or group, Direct | | | | |
| Opcode | offset-low | offset-high | seg-low | seg-high |
| EA | 15 | CS ← segbase IP ← offset | | |
| Inter-segment or group, Indirect | | | | |
| Opcode | mod 101 r/m | — | — | — |
| FF | 24 + EA | CS ← segbase IP ← offset | | |

Fig. 4.7 8086 Unconditional Jump instructions.
(Intel Corporation)

A special case of the direct near-type jump instruction is the direct short-type jump. If the destination for the jump is within a displacement range of +127 to -128 bytes from the current instruction pointer location, the destination can be reached with just an 8-bit displacement. The coding for this type of jump is shown on the second line of the coding template for the direct near JMP in Fig. 4.7. Only one byte is required for the displacement in this case. Again the 8086 produces the new instruction fetch address by adding the signed 8-bit displacement, contained in the instruction, to the contents of the instruction pointer register. Here are some examples of how you use these JMP instructions in programs.

DIRECT WITHIN-SEGMENT NEAR AND DIRECT WITHIN-SEGMENT SHORT JMP EXAMPLES

Suppose that we want an 8086 to execute the instructions in a program over and over. Fig. 4.8 shows how the JMP instruction can be used to do this. In this program, the label BACK followed by a colon is used to give a name to

the address we want to jump back to. When the assembler reads this label, it will make an entry in its symbol table indicating where it found the label. Then, when the assembler reads the JMP instruction and finds the name BACK in the instruction, it will be able to calculate the displacement from the jump instruction to the label. This displacement will be inserted as part of the code for the instruction. Even if you are not using an assembler, you should use labels to indicate jump destinations so that you can easily see them. The NOP instructions used in the program in Fig. 4.8 do nothing except fill space. We used them in this example to represent the instructions that we want to loop through over and over. Once the 8086 gets into the JMP-BACK loop, the only ways it can get out are if the power is turned off, an interrupt occurs, or the system is reset.

Now let's see how the binary code for the JMP instruction in Fig. 4.8 is constructed. The jump is to a label in the same segment, so this narrows our choices down to the first three types of JMP instruction shown in Fig. 4.7. For several reasons, it is best to use the direct-type JMP instruction whenever possible. This narrows our choices down to the first two types in Fig. 4.7. The choice between these two is determined by whether you need a 1-byte or a 2-byte displacement to reach the JMP destination address. Since for our example program the destination address is within the range of -128 to +127 bytes from the instruction after the JMP instruction, we can use the direct within-segment short type of JMP. According to Fig. 4.7, the instruction template for this instruction is 11101011 (EBH) followed by a displacement. Here's how you calculate the displacement to put in the instruction.

NOTE: An assembler does this for you automatically, but you should still learn how it is done to help you in troubleshooting.

The numbers in the left column of Fig. 4.8 represent the offset of each code byte from the code segment base. These are the numbers that will be in the instruction pointer as the program executes. After the 8086 fetches an instruction byte, it automatically increments the instruction pointer to point to the next instruction byte. The displacement in the JMP instruction will then be added to the offset of the next in-line instruction after the JMP instruction. For the example program in Fig. 4.8, the displacement in the JMP instruction will be added to offset 0006H, which is in the instruction pointer after the JMP instruction executes. What this means is that when you are counting the number of bytes of displacement, you

```

1 ; 8086 PROGRAM F4-08.ASM
2 ;ABSTRACT : This program illustrates a "backwards" jump
3 ;REGISTERS : Uses CS, AL
4 ;PORTS    : None used
5
6 0000      CODE SEGMENT
7          ASSUME CS:CODE
8 0000 04 03  BACK: ADD AL, 03H ; Add 3 to total
9 0002 90      NOP      ; Dummy instructions to represent those
10 0003 90     NOP      ; Instructions jumped back over
11 0004 EB FA   JMP BACK ; Jump back over instructions to BACK label
12 0006
13 0006      CODE ENDS
14 0006      END

```

Fig. 4.8 List file of program demonstrating "backward" JMP.

```

1 ; 8086 PROGRAM F4-09.ASM
2 ;ABSTRACT : This program illustrates a "forwards" jump
3 ;REGISTERS : Uses CS, AX
4 ;PORTS    : None used
5
6 0000      CODE SEGMENT
7          ASSUME CS:CODE
8 0000 EB 03 90  JMP THERE ; Skip over a series of instructions
9 0003 90      NOP      ; Dummy instructions to represent those
10 0004 90     NOP      ; Instructions skipped over
11 0005 BB 0000 THERE: MOV AX, 0000H ; Zero accumulator before addition instructions
12 0008 90      NOP      ; Dummy instruction to represent continuation of execution
13 0009
14 0009      CODE ENDS
15 0009      END

```

Fig. 4.9 List file of program demonstrating "forward" JMP.

always start counting from the address of the instruction immediately after the JMP instruction. For the example program, we want to jump from offset 0006H back to offset 0000H. This is a displacement of -6H.

You can't, however, write the displacement in the instruction as -6H. Negative displacements must be expressed in 2's complement, sign-and-magnitude form. We showed how to do this in Chapter 1. First, write the number as an 8-bit positive binary number. In this case, that is 00000110. Then, invert each bit of this, including the sign bit, to give 11111001. Finally, add 1 to that result to give 11111010 binary or FAH, which is the correct 2's complement representation for -6H. As shown on line 11 in the assembler listing for the program in Fig. 4.8, the two code bytes for this JMP instruction then are EBH and FAH.

To summarize this example, then, a label is used to give a name to the destination address for the jump. This name is used to refer to the destination address in the JMP instruction. Since the destination in this example is within the range of -128 to +127 bytes from the address after the JMP instruction, the instruction can be coded as a

direct within-segment short-type JMP. The displacement is calculated by counting the number of bytes from the next address after the JMP instruction to the destination. If the displacement is negative (backward in the program), then it must be expressed in 2's complement form before it can be written in the instruction code template.

Now let's look at another simple example program, in Fig. 4.9, to see how you can jump ahead over a group of instructions in a program. Here again we use a label to give a name to the address that we want to JMP to. We also use NOP instructions to represent the instructions that we want to skip over and the instructions that continue after the JMP. Let's see how this JMP instruction is coded.

When the assembler reads through the source file for this program, it will find the label "THERE" after the JMP mnemonic. At this point the assembler has no way of knowing whether it will need 1 or 2 bytes to represent the displacement to the destination address. The assembler plays it safe by reserving 2 bytes for the displacement. Then the assembler reads on through the rest of the program. When the assembler finds the specified label, it calculates the displacement from the instruction after



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The 8086 Conditional Jump Instructions

As we stated previously, much of the real power of a computer comes from its ability to choose between two courses of action depending on whether some condition is present or not. In the 8086 the six conditional flags indicate the conditions that are present after an instruction. The 8086 Conditional Jump instructions look at the state of a specified flag(s) to determine whether the jump should be made or not.

Figure 4.10 shows the mnemonics for the 8086 Conditional Jump instructions. Next to each mnemonic is a brief explanation of the mnemonic. Note that the terms *above* and *below* are used when you are working with unsigned binary numbers. The 8-bit unsigned number 11000110 is above the 8-bit unsigned number 00111001, for example. The terms *greater* and *less* are used when you are working with signed binary numbers. The 8-bit signed number 00111001 is greater (more positive) than the 8-bit signed number 11000110, which represents a negative number. Also shown in Fig. 4.10 is an indication of the flag conditions that will cause the 8086 to do the jump. If the specified flag conditions are not present, the 8086 will just continue on to the next instruction in sequence. In other words, if the jump condition is not met, the Conditional Jump instruction will effectively function as a NOP. Suppose, for example, we have the instruction JC SAVE, where SAVE is the label at the destination address. If the carry flag is set, this instruction will cause

the 8086 to jump to the instruction at the SAVE: label. If the carry flag is not set, the instruction will have no effect other than taking up a little processor time.

All conditional jumps are *short-type* jumps. This means that the destination label must be in the same code segment as the jump instruction. Also, the destination address must be in the range of -128 bytes to +127 bytes from the address of the instruction after the Jump instruction. As we show in later examples, it is important to be aware of this limit on the range of conditional jumps as you write your programs.

The Conditional Jump instructions are usually used after arithmetic or logic instructions. They are very commonly used after Compare instructions. For this case, the Compare instruction syntax and the Conditional Jump instruction syntax are such that a little trick makes it very easy to see what will cause a jump to occur. Here's the trick. Suppose that you see the instruction sequence

```
CMP BL, DH
JAE HEATER_OFF
```

in a program, and you want to determine what these instructions do. The CMP instruction compares the byte in the DH register with the byte in the BL register and sets flags according to the result. A previous section showed you how the carry and zero flags are affected by a Compare instruction. According to Fig. 4.10, the JAE instruction says, "Jump if above or equal" to the label

| MNEMONIC | CONDITION TESTED | "JUMP IF..." |
|----------|-------------------------|----------------------------|
| JAV/JNBE | (CF or ZF) = 0 | above/not below nor equal |
| JAE/JNB | CF = 0 | above or equal/not below |
| JB/JNAE | CF = 1 | below/not above nor equal |
| JBE/JNA | (CFor ZF) = 1 | below or equal/not above |
| JC | CF = 1 | carry |
| JE/JZ | ZF = 1 | equal/zero |
| JG/JNLE | ((SF xor OF) or ZF) = 0 | greater/not less nor equal |
| JGE/JNL | (SF xor OF) = 0 | greater or equal/not less |
| JL/JNGE | (SF xor OF) = 1 | less/not greater nor equal |
| JLE/JNG | ((SF xor OF) or ZF) = 1 | less or equal/not greater |
| JNC | CF = 0 | not carry |
| JNE/JNZ | ZF = 0 | not equal/not zero |
| JNO | OF = 0 | not overflow |
| JNP/JPO | PF = 0 | not parity/parity odd |
| JNS | SF = 0 | not sign |
| JO | OF = 1 | overflow |
| JPUPE | PF = 1 | parity/parity equal |
| JS | SF = 1 | sign |

Note: "above" and "below" refer to the relationship of two unsigned values;
"greater" and "less" refer to the relationship of two signed values.

Fig. 4.10 8086 Conditional Jump instructions.

HEATER_OFF. The question now is, will it jump if BL is above DH, or will it jump if DH is above BL? You could determine how the flags will be affected by the comparison and use Fig. 4.10 to answer the question, but an easier way is to mentally read parts of the Compare instruction between parts of the Jump instruction. If you read the example sequence as "Jump if BL is above or equal to DH," the meaning of the sequence is immediately clear. As you write your own programs, thinking of a conditional sequence in this way should help you to choose the right Conditional Jump instruction. The next sections show you how we use Conditional and Unconditional Jump instructions to implement some of the standard program structures and solve some common programming problems.

IF-THEN, IF-THEN-ELSE, AND MULTIPLE IF-THEN-ELSE PROGRAMS

IF-THEN Programs

Remember from Chapter 2 that the IF-THEN structure has the format

IF condition THEN

action
action

This structure says that IF the stated condition is found to be true, the series of actions following THEN will be executed. If the condition is false, execution will skip over the actions after the THEN and proceed with the next mainline instruction.

The simple IF-THEN is implemented with a Conditional Jump instruction. In some cases an instruction to set flags is needed before the Conditional Jump instruction. Fig. 4.11a shows, with a program fragment, one way to implement the simple IF-THEN structure. In this program we first compare BX with AX to set the required flags. If the zero flag is set after the comparison, indicating that $AX = BX$, the JE instruction will cause execution to jump to the MOV CL,07H instruction labeled THERE. If $AX \neq BX$, then the ADD AX,0002H instruction after the JE instruction will be executed before the MOV CL,07H instruction.

The implementation in Fig. 4.11a will work well for a short sequence of instructions after the Conditional Jump instruction. However, if the sequence of instructions is lengthy, there is a potential problem. Remember from the discussion of conditional jumps in the last section that a conditional jump can only be to a location in the range of

```
CMP AX, BX ; Compare to set_flags
JE THERE ; If equal then skip correction
ADD AX, 0002H ; Add correction factor
THERE: MOV CL, 07H ; Load count
```

(a)

```
CMP AX, BX ; Compare to set flags
JNE FIX ; If not equal do correction
JMP THERE ; If equal then skip correction
FIX: ADD AX, 0002H ; Add correction factor
THERE: MOV CL, 07H ; Load count
```

(b)

Fig. 4.11 Programming conditional jumps.
(a) Destinations closer than ± 128 bytes,
(b) Destinations further than ± 128 bytes.

-128 bytes to $+127$ bytes from the address after the Conditional Jump instruction. A long sequence of instructions after the Conditional Jump instruction may put the label out of range of the instruction. If you are absolutely sure that the destination label will not be out of range, then use the instruction sequence shown in Fig. 4.11a to implement an IF-THEN structure. If you are not sure whether the destination will be in range, the instruction sequence shown in Fig. 4.11b will always work. In this sequence, the Conditional Jump instruction only has to jump over the JMP instruction. The JMP instruction used to get to the label THERE can jump to anywhere in the code segment, or even to another code segment. Note that you have to change the Conditional Jump instruction from JE to JNE for this second version. The price you pay for not having to worry whether the destination is in range is an extra jump instruction. Incidentally, some assemblers now automatically code Conditional Jump instructions in this way if necessary.

IF-THEN-ELSE Programs

OVERVIEW

The IF-THEN-ELSE structure is used to indicate a choice between two alternative courses of action. Fig. 3.3b shows the flowchart and pseudocode for this structure. Basically the structure has the format

IF condition THEN

action

ELSE

action

This is a different situation from the simple IF-THEN, because here either one series of actions or another series



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

port in the DX register. If, for example, you load DX with FFF8H and then do an IN AL,DX, the 8086 will copy a byte of data from port FFF8H to the AL register. The variable-port input instruction has two major advantages. First, up to 65,536 different input ports can be specified with the 16-bit port address in DX. Second, the port address can be changed as a program executes by simply putting a different number in DX. This is handy in a case where you want the computer to be able to input from 15 different terminals, for example. Instead of writing 15 different input programs, you can write one input program which simply changes the contents of DX to input from each of the different terminals.

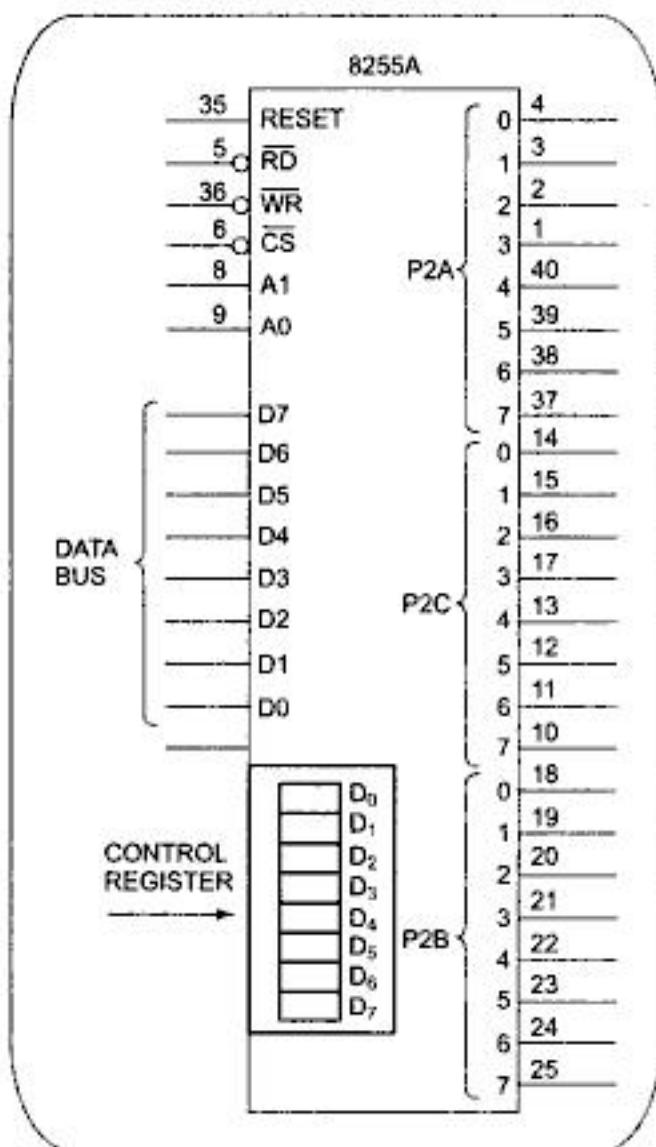


Fig. 4.13 Block diagram of SDK-86 board's 8255A port.

The 8086 also has a fixed-port output instruction and a variable-port output instruction. The fixed-port output instruction has the form OUT port,AL or OUT port,AX. Here again the term port represents an 8-bit port address written in the instruction. OUT 0AH,AL, for example, will copy the contents of the AL register to port 0AH.

The format for the variable-port output instruction is OUT DX,AL or OUT DX,AX. To use this type of

instruction, you have to first put the 16-bit port address in the DX register. If, for example, you load DX with FFFAH and then do an OUT DX,AL instruction, the 8086 will copy the contents of the AL register to port FFFAH.

The device used for parallel input and output ports on the SDK-86 board and in many microcomputers is the Intel 8255. As shown in the block diagram in Fig. 4.13, the 8255 basically contains three 8-bit ports and a control register. Each of the ports and the control register will have a separate address, so you can write to them or read from them. The addresses for the ports and control registers for the two 8255s on an SDK-86 board, for example, are as follows:

| | | | |
|----------|-------|-----------|-------|
| PORt 2A | FFF8H | PORT 1A | FFF9H |
| PORt 2B | FFF9H | PORT 1B | FFFBH |
| PORt 2C | FFFCH | PORT 1C | FFFDH |
| CONTROL2 | FFFEH | CONTROL 1 | FFFFH |

The ports in an 8255 can be individually programmed to operate as input or output ports. When the power is first applied to an 8255, the ports are all configured as input ports. If you want to use any of the ports as an output port, you must write a control word to the control register to initialize that port for operation as an output. Chapter 9 and later chapters describe in detail how to initialize an 8255 for a variety of applications, but we show you here how to initialize one of the ports in an 8255 device on an SDK-86 microcomputer for use as an output port.

You initialize an 8255 by sending a control word to the control register address for that device. As we showed above, the control register address for one of the 8255s on an SDK-86 board is FFFEH. In order to write a control word to this address, you first point DX at the address with the instruction MOV DX,OFFFEH.

The control word needed to make port P2B of this 8255 an output, and P2A and P2C inputs, is 99H. (In Chapter 9 we show how we determined this control word.) You load this control word into AL with MOV AL,99H and send it to the 8255 control register with OUT DX,AL. Now that port 2B is initialized as an output, you can output a byte to that port of the device any time you need to in the program.

IF-THEN-ELSE ASSEMBLY LANGUAGE PROGRAM EXAMPLE

Figure 4.14a, shows the list file of the 8086 assembly language implementation of the algorithm in Fig. 4.12a. The first three instructions in this program initialize port 2B at address FFFAH as an output port, so we can output values to it to turn on LEDs. Assume that the driver for the yellow lamp is connected to bit 0 of port FFFAH, and the driver for the green lamp is connected to bit 1 of port

```

1 ; 8086 PROGRAM F4-14A.ASM
2 ;ABSTRACT : Program section for PC board making machine.
3 ; This program section reads the temperature of a cleaning bath
4 ; solution and lights one of two lamps according to the
5 ; temperature read. If the temp <30°C, a yellow lamp will be
6 ; turned on. If the temp is ≥30°C, a green lamp will be turned on.
7 ;REGISTERS: Uses CS, AL, DX
8 ;PORTS : Uses FFF8H - temperature input
9 ; FFFAH - lamp control output (yellow=bit 0, green=bit 1)
10
11 0000      CODE   SEGMENT
12           ASSUME CS:CODE
13           ;initialize SDK-86 port FFFAH as output port, FFF8H as input port
14 0000 BA FFFE    MOV DX, OFFFEH    ; Point DX to port control register
15 0003 B0 99     MOV AL, 99H      ; Load control word to initialize ports
16 0005 EE        OUT DX, AL     ; Send control word to port control register
17
18 0006 BA FFFB    MOV DX, OFFFBH    ; Point DX at input port
19 0009 EC        IN AL, DX      ; Read temp from sensor on input port
20 000A 3C 1E     CMP AL, 30      ; Compare temp with 30°C
21 000C 72 03     JB YELLOW      ; IF temp <30 THEN light yellow lamp
22 000E EB 0A 90     JMP GREEN      ; ELSE light green lamp
23 0011 B0 01     YELLOW: MOV AL, 01H
24 0013 BA FFFA    MOV DX, OFFFAH    ; Point DX at output port
25 0016 EE        OUT DX, AL     ; Send code to light yellow lamp
26 0017 EB 07 90     JMP EXIT       ; Go to next mainline instruction
27 001A B0 02     GREEN: MOV AL, 02H
28 001C BA FFFA    MOV DX, OFFFAH    ; Point DX at output port
29 001F EE        OUT DX, AL     ; Send code to light green lamp
30 0020 BA FFFC     EXIT:  MOV DX, OFFFCH    ; Next mainline instruction
31 0023 EC        IN AL, DX      ; Read ph sensor
32 0024
33           CODE   ENDS
               END

```

(a)

```

20 000A 3C 1E     CMP AL, 30      ; Compare temp with 30°C
21 000C 73 03     JAE GREEN      ; IF temp ≥30 THEN light green lamp
22 000E EB 0A 90     JMP YELLOW      ; ELSE light yellow lamp
23 0011 B0 02     GREEN: MOV AL, 02H
24 0013 BA FFFA    MOV DX, OFFFAH    ; Point DX at output port
25 0016 EE        OUT DX, AL     ; Send code to light green lamp
26 0017 EB 07 90     JMP EXIT       ; Go to next mainline instruction
27 001A B0 01     YELLOW: MOV AL, 01H
28 001C BA FFFA    MOV DX, OFFFAH    ; Point DX at output port
29 001F EE        OUT DX, AL     ; Send code to light yellow lamp
30 0020 BA FFFC     EXIT:  MOV DX, OFFFCH    ; Next mainline instruction
31 0023 EC        IN AL, DX      ; Read ph sensor
32 0024
33           CODE   ENDS
               END

```

(b)

Fig. 4.14 List file for printed-circuit-board-making machine program. (a) Below 30° version. (b) Program section for above 30° version.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the CMP AL,40 instruction to see whether the temperature is below the second set point, 40°. The JB GREEN instruction will cause a jump to the label GREEN if the temperature is less than 40°. If the jump is not taken, we know that the temperature must be at or above 40°C, so we just go ahead and turn on the red lamp.

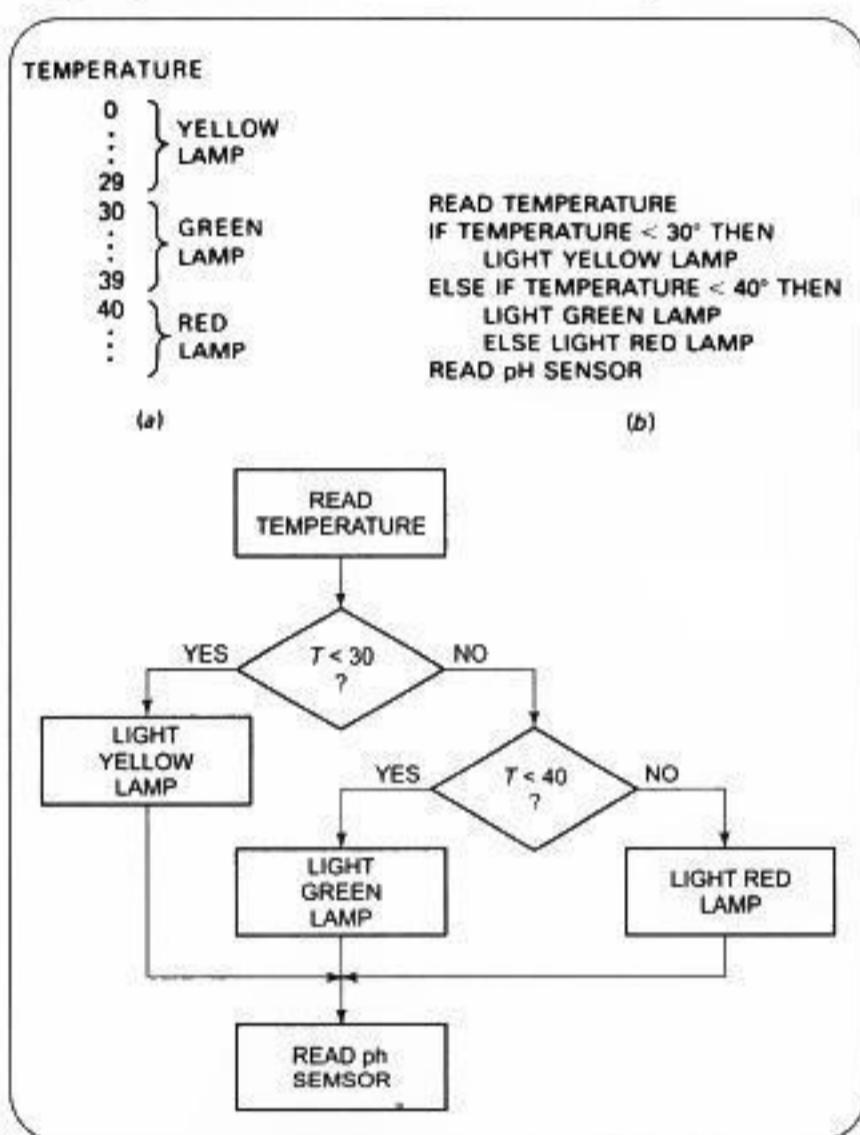


Fig. 4.15 Algorithm for three-lamp printed-circuit-board-making machine. (a) Condition list, (b) Pseudocode, (c) Flowchart.

For this program, we assume that the lines which control the three lamps are connected to port FFFAH. The yellow lamp is connected to bit 0, the green is connected to bit 1, and the red is connected to bit 2. We turn on a lamp by outputting a 1 to the appropriate bit of port FFFAH. The instruction sequence MOV AL,04H—OUT DX,AL, for example, will turn on the red lamp by sending a 1 to bit 2 of port FFFAH.

Summary of IF-THEN-ELSE Implementation

From the preceding examples, you should see that you can implement IF-THEN-ELSE structures in your

programs by using Compare or other instructions to set the appropriate flag(s) and Conditional Jump instructions to go to the desired sequence of actions.

A single IF-THEN-ELSE structure is used to choose one of two alternative series of actions. IF-THEN-ELSE structures can be linked to choose one of three or more alternative series of actions. As shown in Fig. 3.3d, linked IF-THEN-ELSE structures are one way to implement the CASE structure. The algorithm for the printed-circuit-board-making machine lamps program in the preceding section's example could have been expressed as

CASE temperature OF

| | | |
|---------------|---|-------------------|
| < 30 | : | light yellow lamp |
| ≥ 30 and < 40 | : | light green lamp |
| ≥ 40 | : | light red lamp |

This CASE structure would be implemented in the same way as the program in Fig. 4.16. However, expressing the algorithm for the problem as linked IF-THEN-ELSE structures makes it much easier to see how to implement the algorithm in assembly language. In Chapter 10 we show you another way to implement a CASE situation using a *jump table*.

WHILE-DO PROGRAMS

Overview

Remember from the discussion in Chapter 3 that the WHILE-DO structure has the form

WHILE some condition is present DO

action
action

An important point about this structure is that the condition is checked *before* any action is done. In industrial control applications of microprocessors, there are many cases where we want to do this. The following very simple example will show you how to implement this structure in 8086 assembly language.

Defining the Problem and Writing the Algorithm

Suppose that, in controlling a chemical process, we want to bring the temperature of a solution up to 100°C before going on to the next step in the process. If the solution temperature is below 100°, we want to turn on a heater and wait for the temperature to reach 100°. If the solution

```

1 ; 8086 PROGRAM F4-16.ASM
2 ;ABSTRACT : This program section reads the temperature of a cleaning bath
3 ; solution and lights one of three lamps according to the
4 ; temperature read. If the temp < 30°C, a yellow lamp will be
5 ; turned on. If the temp ≥ 30° and < 40°, a green lamp will be
6 ; turned on. Temperatures ≥ 40° will turn on a red lamp.
7 ;REGISTERS : Uses CS, AL, DX
8 ;PORTS : Uses FFF8H - temperature input
9 ; FFFAH - lamp control output, yellow=bit 0, green=bit 1, red=bit 2
10 0000      CODE SEGMENT
11          ASSUME CS:CODE
12          ;initialize port FFFAH for output and port FFF8H for input
13 0000 BA FFFE    MOV DX, OFFFEH   ; Point DX to port control register
14 0003 B0 99    MOV AL, 99H     ; Load control word to set up output port
15 0005 EE        OUT DX, AL    ; Send control word to control register
16
17 0006 BA FFF8    MOV DX, OFFF8H   ; Point DX at input port
18 0009 EC        IN AL, DX     ; Read temp from sensor on input port
19 000A BA FFFA    MOV DX, OFFFAH   ; Point DX at output port
20 000D 3C 1E    CMP AL, 30      ; Compare temp with 30°C
21 000F 72 DA    JB YELLOW    ; IF temp < 30 THEN light yellow lamp
22 0011 3C 28    CMP AL, 40      ; ELSE compare with 40°
23 0013 72 0C    JB GREEN     ; IF temp < 40 THEN light green lamp
24 0015 B0 04    RED: MOV AL, 04H   ; ELSE temp ≥ 40 so light red lamp
25 0017 EE        OUT DX, AL    ; Send code to light red lamp
26 0018 EB 0A 90    JMP EXIT     ; Go to next mainline instruction
27 001B B0 01    YELLOW: MOV AL, 01H   ; Load code to light yellow lamp
28 001D EE        OUT DX, AL    ; Send code to light yellow lamp
29 001E EB 04 90    JMP EXIT     ; Go to next mainline instruction
30 0021 B0 02    GREEN: MOV AL, 02H   ; Load code to light green lamp
31 0023 EE        OUT DX, AL    ; Send code to light green lamp
32 0024 BA FFFC    EXIT: MOV DX, OFFFCH  ; Next mainline instruction
33 0027 EC        IN AL, DX     ; Read ph sensor
34 0028
35          CODE ENDS
            END

```

Fig. 4.16 List file for three-lamp printed-circuit-board-making machine program.

temperature is at or above 100°, then we want to go on with the next step in the process. The WHILE-DO structure fits this problem because we want to check the condition (temperature) before we turn on the heater. We don't want to turn on the heater if the temperature is already high enough because we might overheat the solution.

Figure 4.17 shows a flowchart and the pseudocode of an algorithm for this problem. The first step in the algorithm is to read in the temperature from a sensor connected to a port. The temperature read in is then compared with 100°. These two parts represent the condition-checking part of the structure. If the temperature is at or above 100°, execution will exit the structure and do the next mainline action, turn off the heater. If the temperature is less than 100°, the heater is turned on and the temperature rechecked. Execution will stay in this loop while the temperature is below 100°. Incidentally, it will not do any harm to turn the heater on if it is already on.

When the temperature reaches 100°, execution will exit the structure and go on to the next mainline action, turn off the heater.

Implementing the Algorithm in Assembly Language

We have assumed for this example that the temperature sensor inputs an 8-bit binary value for the Celsius temperature to port FFF8H. We have also assumed that the heater control output is connected to the most significant bit of port FFFAH. As we showed previously, the actual address of port P2B on the SDK-86 board is FFFAH. It is to this address that we will output a byte to turn the heater on or off.

Figure 4.18a, shows one way to implement our algorithm. After initializing the heater control port for output, we read in the temperature, and compare the value read with 100. The JAE instruction after the compare can

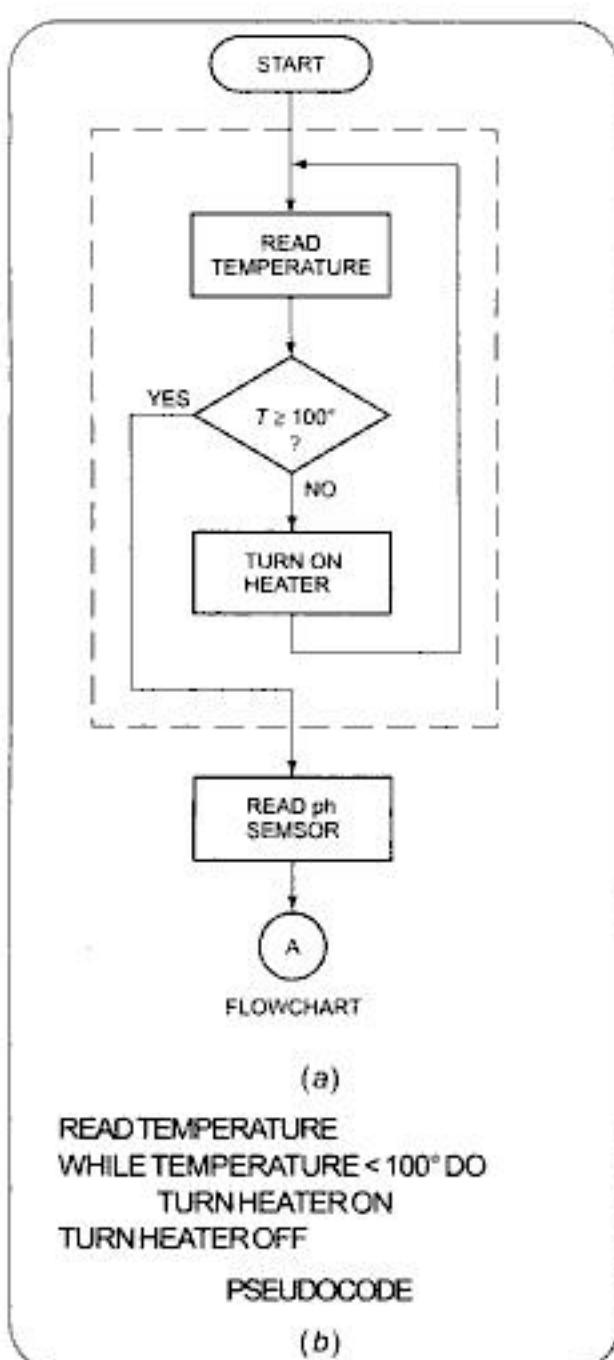


Fig. 4.17 Flowchart and pseudocode for heater control program.

be read as "jump to the label HEATER_OFF if AL is above or equal to 100." Note that we used the Jump if Above or Equal instruction rather than a Jump if Equal instruction. Can you see why? To see the answer, visualize what would happen if we had used a JE instruction and the temperature of the solution were 101°. On the first check, the temperature would not be equal to 100°, so the 8086 would turn on the heater. The heater would not get turned off until meltdown.

If the heater temperature is below 100°, we turn on the heater by loading a 1 in the most significant bit of AL and outputting this value to the most significant bit of port FFFAH. Then we do an unconditional JMP to loop back and check the temperature again.

When the temperature is at or above 100°, we load a 0 in the most significant bit of AL and output this to port

FFFFAH to turn off the heater. Note that the action of turning off the heater is outside the basic WHILE-DO structure. The WHILE-DO structure is shown by the dotted box in the flowchart in Fig. 4.17a and by the indentation in the pseudocode in Fig. 4.17b.

Solving a Potential Problem of Conditional Jump Instructions

In the example program in Fig. 4.18a, we used the Conditional Jump instruction JAE to implement the WHILE-DO structure. Remember that all the Conditional Jump instructions are short-type jumps. This means that a conditional jump can only be to a location within the range of -128 to +127 bytes from the instruction after the Conditional Jump instruction. This limit on the range of the jump posed no problem for the example program in Fig. 4.18a because we were only jumping to a location 8 bytes ahead in the program. Suppose, however, that the instructions for turning on the heater required 220 bytes of memory. The HEATER_OFF label would then be outside the range of the JAE instruction.

We showed you how to solve this problem in Fig. 4.11. To refresh your memory, Fig. 4.18b shows how you can change the instructions in this program slightly to solve the problem without changing the basic WHILE-DO overall structure. In this example, we read the temperature in as before and compare it to 100. We then use the Jump if Below instruction to jump to the program section which turns on the heater. This instruction, together with the CMP instruction, says, "Jump to the label HEATER_ON if AL is below 100." If the temperature is at or above 100, the JB instruction will act like a NOP, and the 8086 will go on to the JMP HEATER_OFF instruction. Changing the Conditional Jump instruction and writing the program in this way means that the destination for the Conditional Jump instruction is always just two instructions away. Therefore, you know that the destination will always be reachable. Except for very time-critical program sections, you should always write Conditional Jump instruction sequences in this way so that you don't have to worry about the potential problem. The disadvantages of this approach are the time and memory space required by the extra JMP instruction.

REPEAT-UNTIL PROGRAMS

Overview

Remember from the discussion in Chapter 3 that the REPEAT-UNTIL structure has the form

REPEAT
action

UNTIL some condition is present

An important point about this structure is that the action or series of actions is done once *before* the condition is

checked. This is different from the WHILE-DO structure, where the condition is checked before any action(s).

The following examples will show you how you can implement the REPEAT-UNTIL with 8086 assembly language and introduce you to some more assembly language programming techniques.

```

1 ; 8086 PROGRAM F4-1BA.ASM
2 ;ABSTRACT : Program turns heater off if temperature ≥ 100°C
3 ; and turns heater on if temperature < 100°C.
4 ;REGISTERS : Uses CS, DX, AL
5 ;PORTS      : Uses FFF8H - temperature data input
6 ;           ; FFFAH - MSB for heater control output, 0=off, 1=on
7 0000 CODE SEGMENT
8 ASSUME CS:CODE
9 ; Initialize port FFFAH for output, and port FFF8H for input
10 0000 BA FFFE MOV DX, OFFFEH ; Point DX to port control register
11 0003 80 99 MOV AL, 99H ; Control word to set up output port
12 0005 EE OUT DX, AL ; Send control word to port
13
14 0006 BA FFF8 TEMP_IN: MOV DX, OFFF8H ; Point at input port
15 0009 EC IN AL, DX ; Input temperature data
16 000A 3C 64 CMP AL, 100 ; If temp ≥ 100 then
17 000C 73 08 JAE HEATER_OFF ; turn heater off
18 000E B0 80 MOV AL, 80H ; else load code for heater on
19 0010 BA FFFA MOV DX, OFFFAH ; Point DX to output port
20 0013 EE OUT DX, AL ; Turn heater on
21 0014 EB F0 JMP TEMP_IN ; WHILE temp < 100 read temp again
22 0016 B0 00 HEATER_OFF:MOV AL, 00 ; Load code for heater off
23 0018 BA FFFA MOV DX, OFFFAH ; Point DX to output port
24 001B EE OUT DX, AL ; Turn heater off
25 001C
26 CODE ENDS
27 END

```

(a)

```

14 0006 BA FFF8 TEMP_IN: MOV DX, OFFF8H ; Point DX at input port
15 0009 EC IN AL, DX ; Read in temperature data
16 000A 3C 64 CMP AL, 100 ; If temp < 100° then
17 000C 72 03 JB HEATER_ON ; turn heater on
18 000E EB 09 90 JMP HEATER_OFF ; else temp ≥ 100° so turn heater off
19 0011 B0 80 HEATER_ON:MOV AL, 80H ; Load code for heater on
20 0013 BA FFFA MOV DX, OFFFAH ; Point DX at output port
21 0016 EE OUT DX, AL ; Turn heater on
22 0017 EB ED JMP TEMP_IN ; WHILE temp < 100° read temp again
23 0019 B0 00 HEATER_OFF:MOV AL, 00 ; Load code for heater off
24 001B BA FFFA MOV DX, OFFFAH ; Point DX at output port
25 001E EE OUT DX, AL ; Turn heater off
26 001F
27 CODE ENDS

```

(b)

Fig. 4.18 List file for heater control program. (a) First approach, (b) Improved version of WHILE-DO section of program.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

prices we have operated on, we use another register as a *counter*. The example program in Fig. 4.21c shows one way in which the algorithm for this problem can be implemented in assembly language.

The example program in Fig. 4.21c uses several assembler directives. Let's review the function of these before describing the operation of the program instructions. The ARRAYS SEGMENT and ARRAYS ENDS directives are used to set up a logical segment containing the data definitions. The CODE SEGMENT and CODE ENDS directives are used to set up a logical segment which contains the program instructions. The ASSUME CS:CODE,DS:ARRAYS directive tells the assembler to use CODE as the code segment and use ARRAYS for all references to the data segment. The END directive lets the assembler know that it has reached the end of the program. Now let's discuss the data structure for the program.

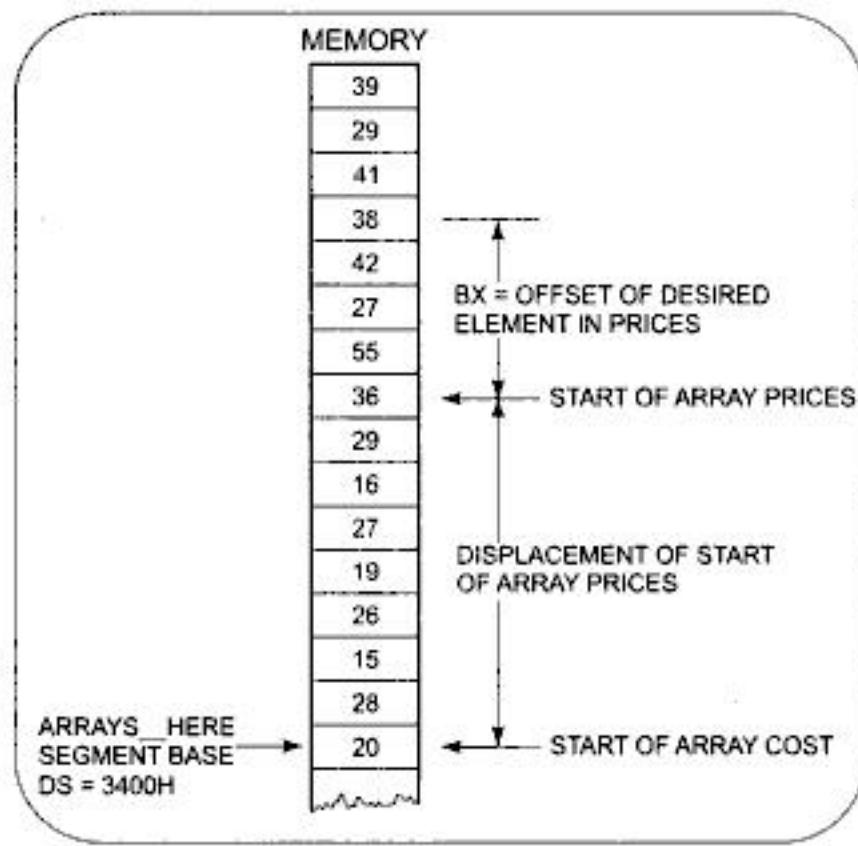


Fig. 4.22 Data arrangement in memory for "inflate prices" program.

The statement COSTDB20H,28H,15H,26H,19H,27H,16H,29H in the program tells the assembler to set aside successive memory locations for an eight-element array of bytes. The array is given the name COST. When the assembled program is loaded into memory to be run, the eight memory locations will be loaded with the eight values specified in the DB statement. The statement PRICES DB 36H,55H,27H,42H,38H,41H,29H,39H sets up another eight-element array of bytes and gives it the name PRICES. The eight memory locations will be loaded with

the specified values when the assembled program is loaded into memory. Figure 4.22, shows how these two arrays will be arranged in memory. Note that the name of the array represents the displacement or offset of the first element of the array from the start of the data segment.

The first two instructions, MOV AX, ARRAYS and MOV DS,AX, initialize the data segment register as was described for the example program in Fig. 3.14. The LEA mnemonic in the next instruction stands for Load Effective Address. An effective address, remember, is the number of bytes from the start of a segment to the desired data item. The instruction LEA BX,PRICES loads the displacement of the first element of PRICES into the BX register. A displacement contained in a register is usually referred to as an *offset*. If you take another look at the data structure for this program in Fig. 4.22, you should see that the offset of PRICES is 0008H. Therefore, the LEA BX,PRICES instruction will load BX with 0008H. We are using BX as a *pointer* to an element in PRICES. We will soon show you how this pointer is used to indicate which price we want to operate on at a given time in the program.

The next instruction, MOV CX,0008H, loads the CX register with the number of prices in the array. We use this register as a *counter* to keep track of how many prices we have operated on. After we operate on each price, we decrement the counter by 1. When the counter reaches 0, we know that we have operated on all the prices.

The MOV AL,[BX] instruction copies one of the prices from memory to the AL register. Here's how it works. Remember, the 8086 produces the physical address for accessing data in memory by adding an effective address to the segment base represented by the 16-bit number in a segment register. A section in Chapter 3 showed you how the effective address could be specified directly in the instruction with either a name or a number. The instructions MOV AX,MULTPLICAND and MOV AX,DS:WORD PTR[0000H] are examples of this addressing mode. We also showed you that the effective address can be contained in a register. The square brackets around BX in the instruction MOV AL,[BX] indicate that the effective address is contained in the BX register. In our example program, we used the LEA BX,PRICES instruction to load the BX register with the offset of the first element in the array PRICES. The first time the MOV AL,[BX] instruction executes, BX will contain 0008H, the effective address or offset of the first price in the array. Therefore, the first price will be copied into AL.

The next instruction, ADD AL,03H, adds the immediate number 03H to the contents of the AL register. The binary



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

As with the previously described Conditional Jump instructions, the LOOP instructions can do only short jumps. This means that the destination label must be in the range of -128 bytes to +127 bytes from the instruction after the LOOP instruction.

As shown in Fig. 4.26, there are two additional forms of LOOP instructions. These instructions check the state of the zero flag as well as the value in the CX register to determine whether to take the jump or not. Shown in Fig. 4.26 are the condition(s) checked by each instruction to determine whether it should do the jump. NE in the mnemonics stands for "not equal," and NZ in the mnemonics stands for "not zero." Instruction mnemonics separated by a "/" in Fig. 4.26 represent the same instruction.

The LOOP instructions decrement the CX register but do not affect the zero flag. This leaves the zero flag available for other tests. The LOOPE/LOOPZ label instruction will decrement the CX register by 1 and jump to the specified label if $CX \neq 0$ and $ZF = 1$. In other words, program execution will exit from the repeat loop if CX has been decremented to zero *or* the zero flag is not set. This instruction might be used after a Compare instruction, for example, to continue a sequence of operations for a specified number of times or until compared values were no longer equal.

The LOOPNE/LOOPNZ label instruction decrements the CX register by 1. If $CX \neq 0$ and $ZF = 0$, this instruction will cause a jump to the specified label. In other words, execution will exit from the loop if CX is equal to zero *or* the zero flag is set. This instruction is useful when you want to execute a sequence of instructions a fixed number of times or until two values are equal. An example might be a program to read data from a disk. We typically write this type of program so that it attempts to read the data until the checksums are equal or until 10 unsuccessful attempts have been made to read the disk. Consult the descriptions for these instructions in Chapter 6 for specific examples of how the LOOPE and LOOPNE instructions are used.

In summary, then, the LOOP instructions are useful for implementing the REPEAT-UNTIL structure for those special cases where we want to do a series of actions a fixed number of times *or* until the zero flag changes state. LOOP instructions incorporate two operations in each instruction; therefore, they are somewhat more efficient than single instructions to do the same job. In the next section we introduce you to instruction timing and show you how the LOOP instruction can be used to produce a delay between the execution of two instructions.

INSTRUCTION TIMING AND DELAY LOOPS

The rate at which 8086 instructions are executed is determined by a crystal-controlled clock with a frequency of a few megahertz. Each instruction takes a certain number of clock cycles to execute. The MOV register, register instruction, for example, requires 2 clock cycles to execute, and the DAA instruction requires 4 clock cycles. The JNZ instruction requires 16 clock cycles if it does the jump, but it requires only 4 clock cycles if it doesn't do the jump. A table in Appendix B shows the number of clock cycles required by each instruction. Using the numbers in this table, you can calculate how long it takes to execute an instruction or series of instructions. For example, if you are running an 8086 with a 5-MHz clock, then each clock cycle takes $1/(5\text{ MHz})$ or $0.2\text{ }\mu\text{s}$. An instruction which takes 4 clock cycles, then, will take $4\text{ clock cycles} \times 0.2\text{ }\mu\text{s}/\text{clock cycle} = 0.8\text{ }\mu\text{s}$ to execute.

A common programming problem is the need to introduce a delay between the execution of two instructions. For example, we might want to read a data value from a port, wait 1 ms, and then read the port again. A later chapter will show how you can use interrupts to mark off time intervals such as this, but for now we will show you how to use a program loop to do it.

The basic principle is to execute an instruction or series of instructions over and over until the desired time has elapsed. Figure 4.27a shows a program we might use to do this. The MOV CX,N instruction loads the CX register with the number of times we want to repeat the delay loop. The NOP instructions next in the program are not required; the KILL_TIME label could be right in front of the LOOP instruction. In this case, only the LOOP instruction would be repeated. However, we put the NOPs in to show you how you can get more delay by extending the time it takes to execute the loop.

| | Clock Cycles |
|------------------|--------------------|
| MOV CX, N ; | $4 = C_o$ |
| KILL_TIME: NOP ; | 3 |
| NOP ; | $3 = C_L$ |
| LOOP KILL_TIME ; | $17 \text{ or } 5$ |

(a)

$$C_T = C_o + N(C_L) - 12$$

$$N = \frac{C_T - C_o + 12}{C_L} = \frac{5000 - 4 + 12}{23} = 218 = 0DAH$$

(b)

Fig. 4.27 Delay loop program and calculations.
(a) Program, (b) Calculations.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

(D7), and store the ASCII code in an array in memory. Next, you want to poll the strobe over and over again until you find it low. When you find the strobe has gone low, check to see if you have read in 10 characters yet. If not, then go back and wait for the strobe to go high again. If 10 characters have been read in, stop.

14. a. Write a delay loop which produces a delay of 500 μ s on an 8086 with a 5-MHz clock.
b. Write a short program which outputs a 1-kHz

square wave on D0 of port FFFAH. The basic principle here is to output a high, wait 500 μ s (0.5 ms), output a low, wait 500 μ s, output a high, etc. Remember that, before you can output to a port device, you must first initialize it as in Fig. 4.18a. If you connect a buffer such as that shown in Fig. 8.23 and a speaker to D0 of the port, you will be able to hear the tone produced.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

by 2 after each word move. If the REP prefix is used, CX will be decremented by 1 after each word move, so CX should be initialized with the number of words in the string.

As you can see from this example, a single MOVS B instruction can cause the 8086 to move up to 65,536 bytes from one location in memory to another. The string instruction is much more efficient than using a sequence of standard instructions, because the 8086 only has to fetch and decode the REP MOVS B instruction once! A standard instruction sequence such as MOV, MOV, INC, INC, LOOP, etc., would have to be fetched and decoded each time around the loop.

USING THE COMPARE STRING BYTE TO CHECK A PASSWORD

For this program example, suppose that we want to compare a user-entered password with the correct password stored in memory. If the passwords do not match, we want to sound an alarm. If the passwords match, we want to allow the user access to the computer and continue with the mainline program. Figure 5.2, shows how we might represent the algorithm for this with a flowchart and with pseudocode. Note that we want to terminate the REPEAT-UNTIL when either the compared bytes do not match or we are at the end of the string. We then use an IF-THEN-ELSE structure to sound the alarm if the compared strings were not equal at any point. If the strings match, the IF-THEN-ELSE just directs execution on to the main program.

To implement this algorithm in assembly language, we probably would first expand the basic structures as shown in Fig. 5.2c. The first action in the expanded algorithm is to initialize the port device for output. We need to have an output port because we will turn on the alarm by outputting a 1 to the alarm control circuit. Next we need to initialize a pointer to each string and a counter to keep track of how many string elements have been compared. The REPEAT-UNTIL shows how we will use the pointer and counter to do the compare.

Figure 5.3, shows how the Compare String instruction, CMPS, can be used to help translate this algorithm to assembly language. As a review, first let's look at the data structure for this program. The statement PASSWORD DB 'FAIL-SAFE' sets aside 8 bytes of memory and gives the first memory location the name PASSWORD. This statement also initializes the eight memory locations with the ASCII codes for the letters FAILSAFE. The ASCII codes will be 46H, 41H, 49H, 4CH, 53H, 41H, 46H, 45H.

When an assembler reads through the source code for a program, it uses a *location counter* to keep track of the

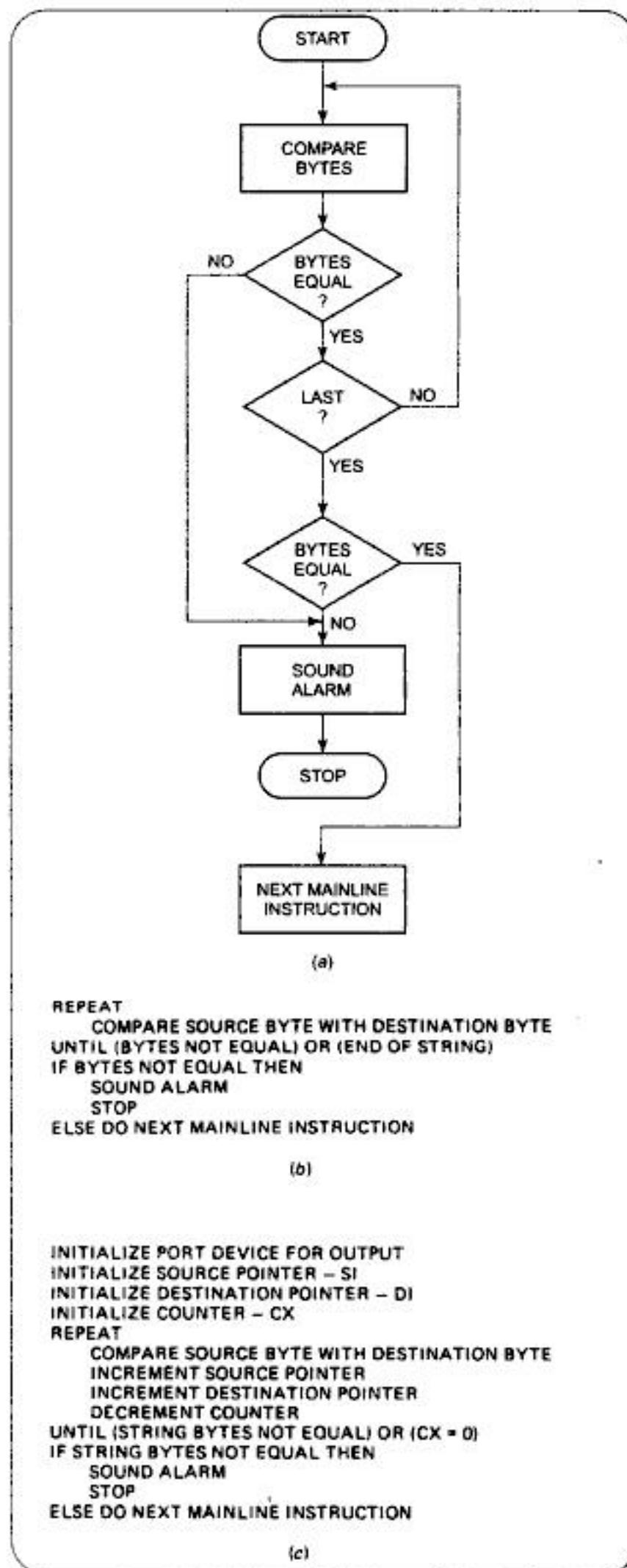


Fig. 5.2 Flowchart and pseudocode for comparing strings program. (a) Flowchart, (b) Initial pseudocode, (c) Expanded pseudocode.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

For most of your programs, you will simply call procedures by name with an instruction such as CALL DELAY. The DELAY in this instruction represents a label you put next to the first instruction of the procedure. This form of CALL instruction is referred to as *direct* because the destination address is specified directly in the instruction. As with the JMP instructions, however, the destination address for a CALL can be specified in several different ways. For reference, Fig. 5.6a shows the coding formats for the four forms of the 8086 CALL instruction. The differences among these four forms are in the way they tell the 8086 to get the starting address for the procedure.

DIRECT WITHIN-SEGMENT NEAR CALL

The first form, direct within-segment near call, tells the 8086 to produce the starting address of the procedure by adding a 16-bit signed displacement contained in the instruction to the contents of the instruction pointer. This is the same process as we described for the direct within-segment near JMP instruction in Chapter 4. With this instruction, the starting address of the procedure can be anywhere in the range of -32,768 bytes to +32,767 bytes from the address of the instruction after the CALL. If you are hand coding a program, you calculate the displacement by counting from the address of the instruction after the CALL to the starting address of the procedure. If the procedure is in memory before the CALL instruction, then the displacement will be negative. In this case you represent the displacement in 16-bit, 2's complement sign-and-magnitude form just as you do for backward JMP instructions. If you are using an assembler, the assembler will automatically calculate the displacement from the instruction after the CALL to a label you put at the start of the procedure.

THE INDIRECT WITHIN-SEGMENT NEAR CALL

The indirect within-segment CALL instruction is also a near call. When this form of CALL executes, the instruction pointer is replaced with a 16-bit value from a specified register or memory location. As indicated by the MOD-R/M byte in the coding template, the source of the value can be any of the eight 16-bit registers or a memory location specified by any one of the 24 addressing modes shown in Fig. 3.8. This form of CALL instruction can be used to choose one of several procedures based on a computed value. The instruction CALL BP, for example, will do a near call to the offset contained in BP. In other words, the value in BP will be put in the instruction pointer.

The instruction CALL WORD PTR [BX] will get the new value for the instruction pointer from a memory location pointed to by BX.

CALL = CALL

Within segment or group, IP relative

| Opcode | DispLow | DispHigh |
|--------|---------|---------------------------------------|
| E8 | 19 | IP ← IP + Disp16 — (SP) ← return link |
| Opcode | Clocks | Operation |
| FF | 16 | IP ← Reg16 — (SP) ← return link |

Within segment or group, Indirect

| Opcode | mod 010 r/m | | | | |
|--------|-------------|---------------------------------|--|--|--|
| Opcode | Clocks | Operation | | | |
| FF | 16 | IP ← Reg16 — (SP) ← return link | | | |
| FF | 21 + EA | IP ← Mem16 — (SP) ← return link | | | |

Inter-segment or group, Direct

| Opcode | offset-low | offset-high | seg-low | seg-high |
|--------|------------|-----------------------------|---------|----------|
| Opcode | Clocks | Operation | | |
| 9A | 28 | CS ← segbase IP ← offset | | |
| | | | | |

Inter-segment or group, Indirect

| Opcode | mod 100 r/m | mem-low | mem-high |
|--------|-------------|-----------------------------|----------|
| Opcode | Clocks | Operation | |
| FF | 37 + EA | CS ← Segbase IP ← Offset | |
| | | | |

(a)

RET = Return from Subroutine

| Opcode | | |
|--------|--------|----------------------|
| Opcode | Clocks | Operation |
| C3 | 8 | intra-segment return |
| CB | 18 | intra-segment return |

Return and add constant to SP

| Opcode | DataL | DataH |
|--------|--------|---------------------------|
| Opcode | Clocks | Operation |
| C2 | 12 | intra-segment ret and add |
| CA | 17 | intra-segment ret and add |

(b)

Fig. 5.6 8086 CALL and RET instruction formats.
(a) CALL. (b) RET. (Intel Corporation)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

in the array. After each value is read in and put in the array, the delay procedure is called to produce the desired interval between samples. When execution returns to the mainline, the pointer is incremented so that it points to the next location in the array. Finally, the counter is decremented to determine whether the desired number of samples have been taken. If not, the read, store, and delay series of instructions is repeated.

Note that the algorithm for the procedure is done separately from that for the main program. As we discussed in the introduction to procedures, the flow of the mainline program is clearer if much of the detail is put in separate procedures. For the delay procedure, you simply load a number in a register or memory location and decrement the number until it is zero.

```

1 ; 8086 PROGRAM F5-10.ASM
2 ;ABSTRACT : This program takes in data samples from a port at 1 ms
3 ; intervals, masks the upper 4 bits of each sample, and
4 ; puts each masked sample in successive locations in an array.
5 ;REGISTERS : Uses CS, SS, DS, AX, BX, CX, DX, SI, SP
6 ;PORTS : Uses OFFF8H - data samples input from port P2A on SDK-86
7 ;PROCEDURES: Uses WAIT_1MS
8
9     = FFFF8             PRESSURE_PORT EQU OFFF8H
10
11 0000           DATA      SEGMENT
12 0000 64*(0000)    PRESSURES   DW 100 DUP(0)      ; Set up array of 100 words
13     = 0064          NBR_OF_SAMPLES EQU ((-$-PRESSURES)/2)
14 00C8           DATA      ENDS
15
16 0000           STACK_SEG SEGMENT
17 0000 28*(0000)    DW 40 DUP(0)      ; set stack length of 40 words
18           STACK_TOP LABEL WORD
19 0050           STACK_SEG ENDS
20
21 0000           CODE      SEGMENT
22           ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
23 0000 BB 0000s     START:    MOV AX, DATA        ; Initialize data segment register
24 0003 8E D8         MOV DS, AX
25 0005 BB 0000s     MOV AX, STACK_SEG    ; Initialize stack segment register
26 0008 8E D0         MOV SS, AX
27 000A BC 0050r     MOV SP, OFFSET STACK_TOP ; Initialize stack pointer to top of stack
28
29 000D 8D 36 0000r   LEA SI, PRESSURES   ; Point SI to start of array
30 0011 BB 0064       MOV BX, NBR_OF_SAMPLES ; Load BX with number of samples
31 0014 BA FFFF8     MOV DX, PRESSURE_PORT ; Point DX at input port
32 0017 ED           NEXT_VALUE:IN AX, DX      ; Read data from port
33 0018 25 0FFF       AND AX, OFFFH        ; Mask upper 4 bits
34 001B 89 04         MOV [SI], AX        ; Store data word in array
35 001D E8 0006       CALL WAIT_1MS      ; Delay 1 ms
36 0020 46           INC SI            ; Point SI at next location in array
37 0021 46
38 0022 48           DEC BX            ; Decrement sample counter
39 0023 75 F2         JNZ NEXT_VALUE    ; Repeat until 100 samples done
40 0025 90           STOP:    NOP
41
42 0026             WAIT_1MS PROC NEAR
43 0026 B9 23F2H     MOV CX, 23F2H        ; Load delay constant into CX
44 0029 E2 FE         HERE:    LOOP HERE      ; Loop until CX = 0
45 002B C3           RET
46 002C             WAIT_1MS ENDP
47
48 002C             CODE      ENDS
49             END START

```

Fig. 5.10 Assembly language program to read in 100 samples of data at 1-ms intervals.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

decrements the stack pointer by 2 and copies the flag word to the stack. The 80186, 80286, and 80386, incidentally, have a single instruction, **PUSHA**, which pushes AX, CX, DX, BX, SP, BP, SI, and DI on the stack.

You can **POP** a word from the stack to any of the registers except CS, and you can **POP** a word from the stack to a memory location specified in any one of those 24 ways. The **POPF** instruction copies a word from the stack to the flag register and increments the stack pointer by 2. The 80186, 80286, and 80386 **POPA** instruction copies words from the stack to the DI, SI, BP, BX, DX, CX, and AX registers. Note that the **POPA** instruction does not return a value to the SP register.

When you **PUSH** several registers on the stack, you have to remember to **POP** them off in the reverse order that you pushed them on. This is because the stack functions in a *last-in-first-out* manner. An everyday example of this type of operation is the spring-loaded plate stacks seen in some restaurants. The last plate pushed onto the stack is the first one popped off. Fig. 5.12a should help you visualize how this works for the 8086.

The first four instructions show a sequence of **PUSH** instructions you might use to save registers and flags at the start of a near procedure called **MULTO**. Figure 5.12b shows the contents of the stack after the **CALL** and **PUSH** instructions execute. The first entry in the stack is the copy of the instruction pointer put there by the **CALL** instruction that called the procedure. Following this are the flag word and the words from registers AX, BX, and CX. After all of these are pushed on the stack, the stack pointer is left pointing at the location in the stack where CX was pushed.

At the end of the procedure, you want to restore the

saved values to the registers and flags. You first **POP CX** because it was the last register pushed on the stack. After CX is popped, the stack pointer will be left pointing at the location where BX is stored. Therefore, you **POP BX** next. You continue popping until all the registers and flags are restored. The **RET** instruction then copies the return address from the stack to the instruction pointer to return execution to the main program. It is very important to keep the number of pushes equal to the number of pops or in some other way keep the stack balanced so that the **RET** instruction finds the correct word to put in the instruction pointer.

Some programmers like to push and pop registers in the mainline or calling program rather than in the procedure as we did in Fig. 5.12a. This approach has the advantage that you can push only those registers that you care about saving each time you call the procedure. The disadvantages of this approach are that the pushes and pops clutter up the mainline program, and that you may decide to use another register at some point in the program and forget to add a push for it. We like to push the flags and any registers used in a procedure directly in the procedure. This way we always know that the procedure can be called from anywhere in the program without losing the contents of any registers. Another advantage of this approach is that you only have to write the pushes and pops once. A disadvantage is that in a situation in which not all the pushes are needed, the procedure may take a little longer to run.

Passing Parameters to and from Procedures

Often when we call a procedure, we want to make some data values or addresses available to the procedure.

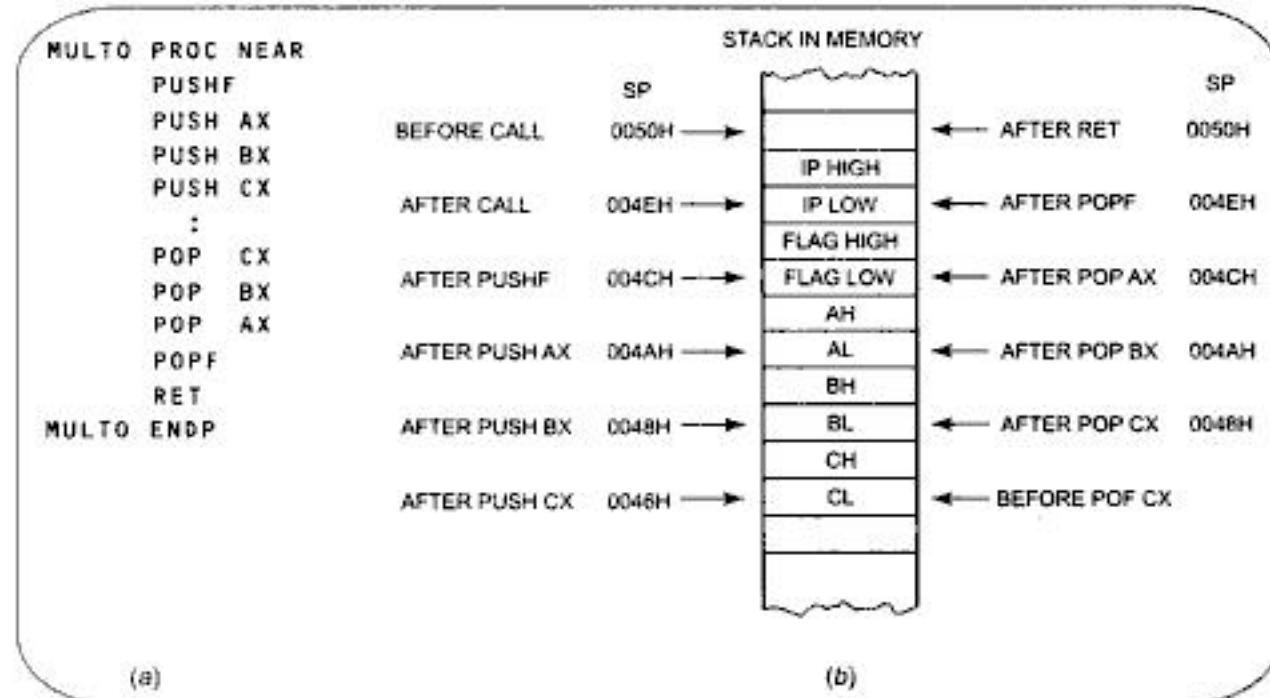


Fig. 5.12 Using **PUSH** and **POP** instructions. (a) Instruction sequence, (b) Effect on stack and stack pointer.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

instruction. From here on, the procedure is the same as the previous version until we reach the point where we want to pass the binary result back to the calling program. Here we use the `MOV BIN_VALUE,AL` instruction to copy the result directly to the dedicated memory location we set aside for it. To complete the procedure, we pop the flags and registers, and return to the main program.

The approach used in Fig. 5.15 works in this case, but it has a severe limitation. Can you see what it is? The limitation is that this procedure will always look to the memory location named `BCD_INPUT` to get its data and will always put its result in the memory location called `BIN_VALUE`. In other words, the way it is written, we can't easily use this procedure to convert a BCD number in some other memory location. As we explain in detail later, this method has the further problem that it makes the procedure *nonreentrant*.

PASSING PARAMETERS USING POINTERS

A parameter-passing method which overcomes the disadvantage of using data item names directly in a procedure is to use registers to pass the procedure pointers to the desired data. Fig. 5.16, shows one way to do this. In the main program, before we call the procedure, we use the `MOV SI,OFFSET BCD_INPUT` instruction to set up the `SI` register as a pointer to the memory location `BCD_INPUT`. We also use the `MOV DI,OFFSET BIN_VALUE` instruction to set up the `DI` register as a pointer to the memory location named `BIN_VALUE`.

In the procedure, the `MOV AL,[SI]` instruction will copy the byte pointed to by `SI` into `AL`. Likewise, the `MOV [DI],AL` instruction later in the procedure will copy the byte from `AL` to the memory location pointed to by `DI`.

This pointer approach is more versatile because you can pass the procedure pointers to data anywhere in memory. You can pass pointers to individual values or pointers to arrays or strings. To access complex data structures, you can use registers to pass the segment base and the offset of a table of pointers in memory. The procedure then can read in a pointer from the table and use the pointer to access the desired data.

For many of your programs, you will probably use registers to pass data parameters or pointers to procedures. As we show you in Chapter 8, this is the method you use when you call procedures in the *Basic Input/Output System* or *BIOS* of a computer. However, as we show you in later chapters, for programs which allow several users to timeshare a system or those which consist of a mixture of high-level languages and assembly

language, we usually use the stack to pass parameters to and from procedures.

PASSING PARAMETERS USING THE STACK

To pass parameters to a procedure using the stack, we push the parameters on the stack somewhere in the mainline program before we call the procedure. Instructions in the procedure then read the parameters from the stack as needed. Likewise, parameters to be passed back to the calling program are written to the stack by instructions in the procedure and read off the stack by instructions in the mainline program. A simple example will best show you how this works.

Figure 5.17, shows a version of our `BCD_BIN` procedure which uses the stack for passing the BCD number to the procedure and for passing the binary value back to the calling program. To save space here, we assume that previous instructions in the mainline program set up a stack segment, initialized the stack segment register, and initialized the stack pointer. Now in the mainline fragment in Fig. 5.17, we copy the BCD number into `AL`. We then copy `AX` to the stack with the `PUSH AX` instruction. In a more complex example, the BCD number or a pointer to it would probably be put on the stack by a different mechanism, but the important point for now is that the BCD value is on the stack for the procedure to access.

The `CALL` instruction in the mainline program decrements the stack pointer by 2, copies the return address onto the stack, and loads the instruction pointer with the starting address of the procedure. `PUSH` instructions at the start of the procedure save the flags and all the registers used in the procedure on the stack. Before discussing any more instructions, let's take a look at the contents of the stack after these pushes.

Figure 5.18, shows how the values pushed on the stack will be arranged. Note that the BCD value is in the stack at a higher address than the return address. After the registers are pushed onto the stack, the stack pointer is left pointing to the stack location where `BP` is stored. Now, the question is, how can we easily access the parameter that seems buried in the stack? One way is to add 12 to the stack pointer with an `ADD SP,12` instruction so that the stack pointer points to the word we want from the stack. A `POP AX` instruction could then be used to copy the desired word from the stack to `AX`. However, for a variety of reasons, which we will explain later, we would like to be able to access the parameter without changing the contents of the stack pointer.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

and CS, and increments the stack pointer by 4. The other two forms of RET instruction perform the same functions, but they also add a number specified in the instruction to the stack pointer. The near RET 6 instruction, for example, will first copy a word from the stack to the instruction pointer and increment the stack pointer by 2. It will then add 6 more to the stack pointer. This is a quick way to skip the stack pointer up over some old parameters on the stack.

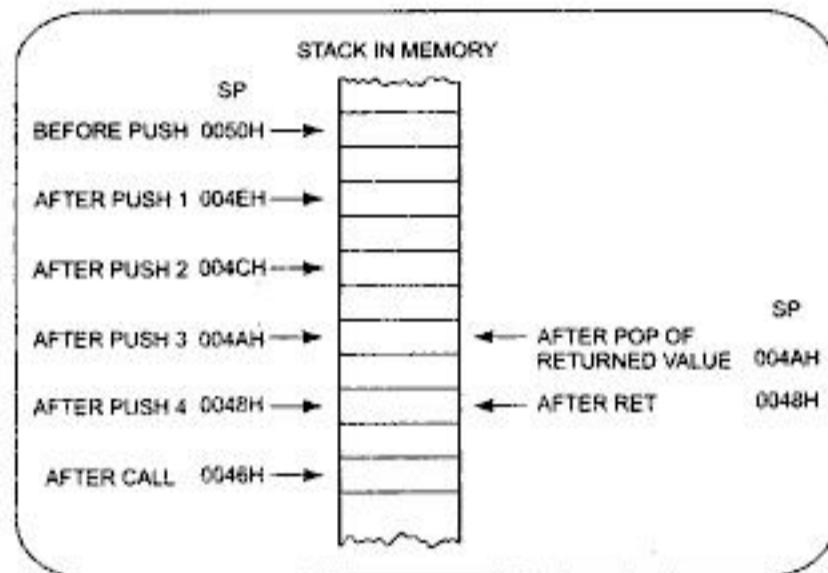


Fig 5.19 Stack diagram showing cause of stack overflow.

SUMMARY OF PASSING PARAMETERS TO AND FROM PROCEDURES

You can pass parameters between a calling program and a procedure using registers, dedicated memory locations, or the stack. The method you choose depends largely on the specific program. There are no hard rules, but here are a few guidelines. For simple programs with just a few parameters to pass, registers are usually the easiest to use. For passing arrays or other data structures to and from procedures, you can use registers to pass pointers to the start of these data structures. As we explained previously, passing pointers to the procedure is a much more versatile method than having the procedure access the data structure directly by name.

For procedures in a multiuser system program, procedures that will be called from a high-level language program, or procedures that call themselves, parameters should be passed on the stack. When writing programs which pass parameters on the stack, you should use stack diagrams such as the one in Fig. 5.18 to help you keep track of where everything is in the stack at a particular time. The following section will give you some additional guidance as to when to use the stack to pass parameters, and it will give you some additional practice following the stack and stack pointer as a program executes.

Writing and Debugging Programs Containing Procedures

The most important point in writing a program containing procedures is to approach the overall job very systematically. You carefully work out the overall structure of the program and break it down into modules which can easily be written as procedures. You then set up the data structures and write the mainline program so that you know what each procedure has to do and how parameters can be most easily passed to each procedure.

To test this mainline program, you can simulate each procedure with a few instructions which simply pass test values back to the mainline program. Some programmers refer to these "dummy" procedures as *stubs*. If the structure of the mainline program seems reasonable, you then develop each procedure and replace the dummy with it. The advantage of this approach is that you have a structure to hang the procedures on. If you write the procedures first, you have the messy problem of trying to write a mainline program to connect all the pieces together.

Now, suppose that you have approached a program as we suggested, and the program doesn't work. After you have checked the algorithm and instructions, you should check that the number of PUSH and POP instructions are equal in each procedure. If none of the checks turns up anything, you can use the system debugging tools to track down the problem. Probably the best tools to help you localize a problem to a small area are breakpoints. Run the program to a breakpoint just before a CALL instruction to see whether the correct parameters are being passed to the procedure. Put a breakpoint at the start of the procedure to see if execution ever gets to the procedure. If execution gets to the procedure, move the breakpoint to a later point in the procedure to determine whether the procedure found the parameters passed from the mainline. Use a breakpoint just before the RET instruction to see whether the procedure produced the correct results and put these results in the correct locations to pass them back to the mainline program. Inserting breakpoints at key points in your program and checking the results at those points is much more effective in locating a problem than random poking and experimenting.

Reentrant and Recursive Procedures

The terms *reentrant* and *recursive* are often used in microprocessor manufacturers' literature, but seldom illustrated with examples. Here we try to give these terms some meaning for you. You should make almost all the procedures you write reentrant, so read that section



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Figure 5.21c shows how we can represent this algorithm in slightly expanded pseudocode. Use the program flow diagram in Fig. 5.21b to help you see how execution continues after the return when $N = 1$ and $N = 3$. Can you see that if N is initially 1, the first return will return execution to the instruction following CALL FACTO in the mainline program? If the initial N was 3, for example, this return will return execution to the instruction after the call in the procedure. Likewise, the return after the multiply can send execution back to the next instruction after the call or back to the mainline program if the final result has been computed.

Fig. 5.21d shows a flowchart for this algorithm. Note that the flowchart shows the same ambiguity about where the return operations send execution to.

ASSEMBLY LANGUAGE RECURSIVE FACTORIAL PROCEDURE

Figure 5.22 shows an 8086 assembly language procedure which computes the factorial of a number in the range of 1 to 8. To save space, we have not included instructions to return an error message if the number passed to the procedure is out of this range.

The parameters in case of complex routines, may be required to be referred several times and in a random fashion, not necessarily following the last-in-first-out sequence, as in a stack. The stack thus becomes an unsuitable structure for parameter passing. It is for this purpose, the provision of BP is made, and it is for this reason, BP defaults into the stack segment. We make a separate structure (in the stack) called the stack frame, and put our parameters to be passed in this stack frame, including the output desired from the subroutine. As discussed earlier, we provide first, space for the output variable, by subtracting enough number from the stack

pointer. Then we push the input variables. Having done this in the main program, we call the subroutine. In the subroutine, the first thing we should do is to get the BP pushed onto the stack and copy the SP in BP. BP now becomes the frame pointer; the space starting from the return address, down to, and including the output space in the stack, will be the stack frame. The frame could be further expanded by providing space for the local variables of the subroutine, which may have to be referred a number of times. This space is provided by subtracting appropriate number from the stack pointer. Space above this in the stack is now available for use in the subroutine as a regular stack. This separation of stack space into frame and stack, will give a disciplined approach to the parameter passing problem of subroutines. According to this, the recursive subroutine for factorial will be as shown. Note that the stack frame and the input and output parameters are referred to in the subroutine, by indexed addressing using BP with positive displacement, while the local parameters are with negative displacement. Stack beyond the frame is available to the subroutine to be used like an ordinary stack with the LIFO operation. While returning, the process simply does move to SP from BP and then pops BP, to return to old BP, and then executes ret n, (in the program shown, return alone is used, instead of return n, which is followed by add SP, 2, which is another way of doing it), where n is the number of input bytes to be discarded from the stack. Now the output of the subroutine can be simply popped from the stack in the main program. In effect, the parameters are pushed in the calling program, and recalled using BP relative addressing in the called program. On return, the results can be popped in the main program. Based on this philosophy, the recursive factorial program can be seen to be as follows:

Details of Passing Parameters to a Subroutine Using Stack Arrays or Stack Frames.

| | | |
|-------|-----------|---|
| Main: | Mov AX, n | ; (Choose n in the range 0 to 8 only.) |
| | Sub SP, 2 | ; make space for one word output (Factorial value) |
| | Push AX | ; input parameter to the stack. |
| | Call Fact | ; call the recursive routine. |
| | Add SP, 2 | ; clear the stack of the input; to undo the Push AX above |
| | Pop AX | ; get the output result in reg. AX. |
| | Int 01 | ; pass control to the DOS. |

Fig 5.22 A recursive program to calculate the factorial of a number between 1 and 8
(Continued)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

be accessed from other modules. The statement PUBLIC DISPLAY, for example, tells the linker that a procedure or variable named DISPLAY can be legally accessed from another assembly module.

- In a module which calls a procedure or accesses a variable in another module, you must use the EXTRN directive to let the assembler know that the procedure or variable is not in this module. The EXTRN statement also gives the linker some needed information about the procedure or variable. As an example of this, the statement EXTRN DISPLAY:FAR, SECONDS:BYTE tells the linker that DISPLAY is a far procedure and SECONDS is a variable of type byte located in another assembly module.

To summarize, a procedure or variable declared PUBLIC in one module will be declared EXTRN in modules which access the procedure or variable. Now let's see how these directives are used in an actual program.

PROBLEM DEFINITION AND ALGORITHM DISCUSSION

The procedure in the following example program was written to solve a small problem we encountered when writing the program for a microprocessor-controlled medical instrument. Here's the problem.

In the program we add up a series of values read in from an A/D converter. The sum is an unsigned number of between 24 and 32 bits. We needed to scale this value by dividing it by 10. This seems easy because the 8086 DIV instruction will divide a 32-bit unsigned binary number by a 16-bit binary number. The quotient from the division, remember, is put in AX, and the remainder is put in DX. However, if the quotient is larger than 16 bits, as it will often be for our scaling, the quotient will not fit in AX. In this case the 8086 will automatically respond in the same way that it would if you tried to divide a number by zero. We will discuss the details of this response in Chapter 8. For now, it is enough to say that we don't want the 8086 to make this response. The simple solution we came up with is to do the division in two steps in such a way that we get a 32-bit quotient and a 16-bit remainder.

Our algorithm is a simple sequence of actions very similar to the way you were probably taught to do long division. We will first describe how this works with decimal numbers, and then we will show how it works with 32-bit and 16-bit binary numbers.

Figure 5.26a shows an example of long division of the decimal number 433 by the decimal number 9. The 9 won't divide into the 4, so we put a 0 or nothing into this digit

position of the quotient. We then see if 9 divides into 43. It fits 4 times, so we put a 4 in this digit position of the quotient and subtract 4×9 from the 43. The remainder of 7 now becomes the high digit of the 73, the next number we try to divide the 9 into. After we find that the 9 fits 8 times and subtract 9×8 from the 73, we are left with a final remainder of 1. Now let's see how we do this with large binary numbers.

As shown in Fig. 5.26b, we first divide the 16-bit divisor into a 32-bit number made up of a word of all 0's and the high word of the dividend. This division gives us the high word of the quotient and a remainder. The remainder becomes the high word of the dividend for the next division, just as it did for the decimal division. We move the low word of the original dividend in as the low word of this dividend and divide by the 16-bit divisor again. The 16-bit quotient from this division is the low word of the 32-bit quotient we want. The 16-bit final remainder can be used to round off the quotient or be discarded, depending on the application.

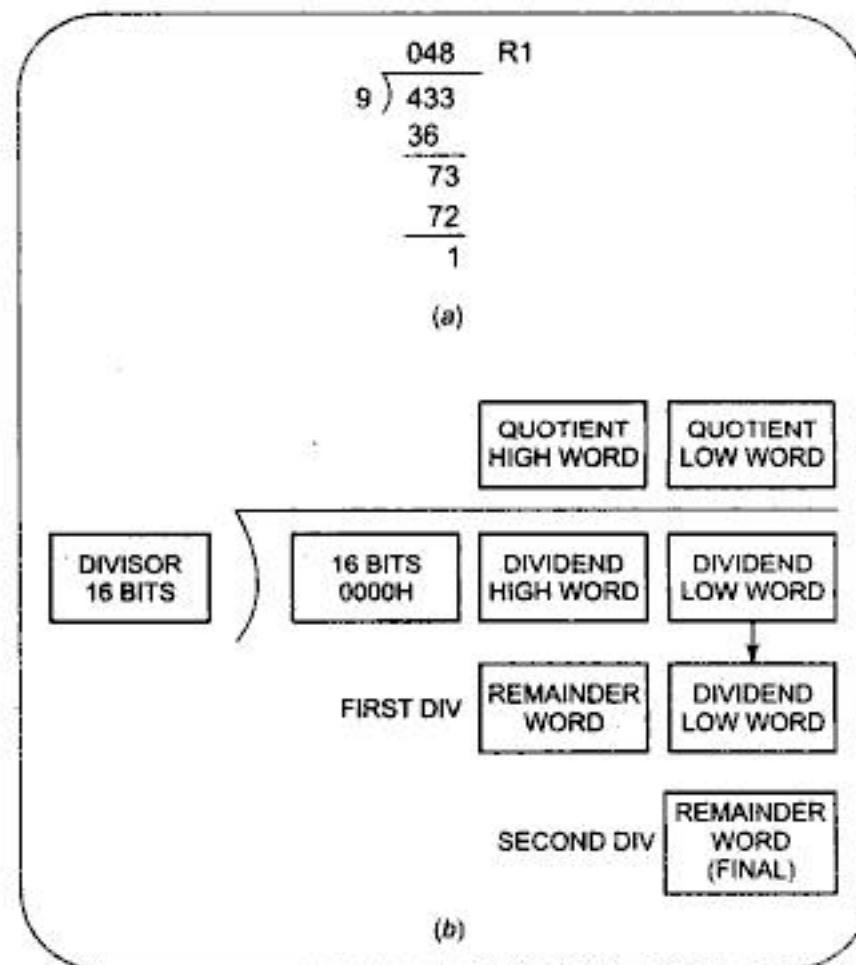


Fig. 5.26 Algorithm for smart divide procedure.
(a) Decimal analogy. (b) 8086 approach.

THE ASSEMBLY LANGUAGE PROGRAM

Figure 5.27a shows the mainline of a program which calls the procedure shown in Fig. 5.27b, which implements our division algorithm. We wrote these two as separate



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Turbo Assembler Version 1.0 05-05-89 13:09:20 Page 1

```

1 ; 8086 PROCEDURE F5-27B.ASM called by program F5-27A.ASH
2 ;ABSTRACT : PROCEDURE SMART_DIVIDE.
3 ; This procedure divides a 32-bit number by a 16-bit number
4 ; to give a 32-bit quotient and a 16-bit remainder.
5 ;INPUT : Dividend - low word in AX, high word in DX, Divisor in CX
6 ;OUTPUT : Quotient - low word in AX, high word in DX. Remainder in CX
7 ;DESTROYS : Carry - carry flag set if try to divide by zero
8 ;PORTS : None used
9
10
11 0000 DATA SEGMENT PUBLIC ; This block tells the assembler that
12 EXTRN DIVISOR:WORD ; the divisor is a word variable found
13 0000 DATA ENDS ; in the external segment named DATA
14
15 PUBLIC SMART_DIVIDE ; Make SMART_DIVIDE available to other modules
16
17 0000 PROCEDURES SEGMENT PUBLIC
18 0000 SMART_DIVIDE PROC FAR
19 ASSUME CS:PROCEDURES, DS:DATA
20 0000 83 3E 0000e 00 CMP DIVISOR, 0 ; Check for illegal divide
21 0005 74 17 JE ERROR_EXIT ; IF divisor = 0, exit procedure
22 0007 8B D8 MOV BX, AX ; Save low order of dividend
23 0009 8B C2 MOV AX, DX ; Position high word for 1st divide
24 000B BA 0000 MOV DX, 0000H ; Zero DX
25 000E F7 F1 DIV CX ; DX:AX/CX, quotient in AX, remainder in DX
26 0010 8B E8 MOV BP, AX ; Save high order of final result
27 0012 8B C3 MOV AX, BX ; Get back low order of dividend
28 0014 F7 F1 DIV CX ; DX:AX/CX, quotient in AX, remainder in DX
29 0016 8B CA MOV CX, DX ; Pass remainder back in CX
30 0018 8B D5 MOV DX, BP ; Pass high order result back in DX
31 001A F8 CLC ; Clear carry to indicate valid result
32 001B EB 02 90 JMP EXIT ; Finished
33 001E F9 ERROR_EXIT: STC ; Set carry to indicate divide by zero
34 001F C8 EXIT: RET
35 0020 SMART_DIVIDE ENDP
36 0020 PROCEDURES ENDS
37 END

```

Turbo Assembler Version 1.0 05-05-89 13:09:20 Page 2

Symbol Table

| Symbol Name | Type | Value | |
|------------------------------|----------|------------------|---------------|
| ??DATE | Text | "05-05-89" | |
| ??FILENAME | Text | "F5-27B " | |
| ??TIME | Text | "13:09:19" | |
| ??VERSION | Number | 0100 | |
| @CPU | Text | 0101H | |
| @CURSEG | Text | PROCEDURES | |
| @FILENAME | Text | F5-27B | |
| @WORDSIZE | Text | 2 | |
| DIVISOR | Word | DATA:---- Extern | |
| ERROR_EXIT | Near | PROCEDURES:001E | |
| EXIT | Near | PROCEDURES:001F | |
| SMART_DIVIDE | Far | PROCEDURES:0000 | |
| Groups & Segments | | | |
| DATA | Bit Size | Align | Combine Class |
| PROCEDURES | 16 | 0000 | Para Public |
| | 16 | 0020 | Para Public |

(b)

Fig. 5.27 (continued)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- b. What effect would it have on the execution of this program if the POPF instruction in the procedure was accidentally left out? Describe the steps you would take in tracking down this problem if you did not notice it in the program listing.
5. Show the binary codes for the following instructions.
- The instruction which will call a procedure which is 97H addresses higher in memory than the instruction after a call instruction.
 - An instruction which returns execution from a far procedure to a mainline program and increments the stack pointer by 4.
6. a. List three methods of passing parameters to a procedure and give the advantages and disadvantages of each method.
 b. Define the term *reentrant* and explain how you must pass parameters to a procedure so that it is reentrant.
7. a. Write a procedure which produces a delay of 3.33 ms when run on an 8086 with a 5-MHz clock.
 b. Write a mainline program which uses this procedure to output a square wave on bit D0 of port FFFAH.
8. Write a procedure which converts a four-digit BCD number passed in AX to its binary equivalent. Use the algorithm in Fig. 5.13.
9. The 8086 MUL instruction allows you to multiply a 16-bit number by a 16-bit binary number to give a 32-bit result. In some cases, however, you may need to multiply a 32-bit number by a 32-bit number to give a 64-bit result. With the MUL instruction and a little adding, you can easily do this. Fig. 5.28 shows in diagram form how to do it. Each letter in the diagram represents a 16-bit number. The principle is to use MUL to form partial products and add these partial products together as shown. Write an algorithm for this multiplication and then write the 8086 assembly language program for the algorithm.

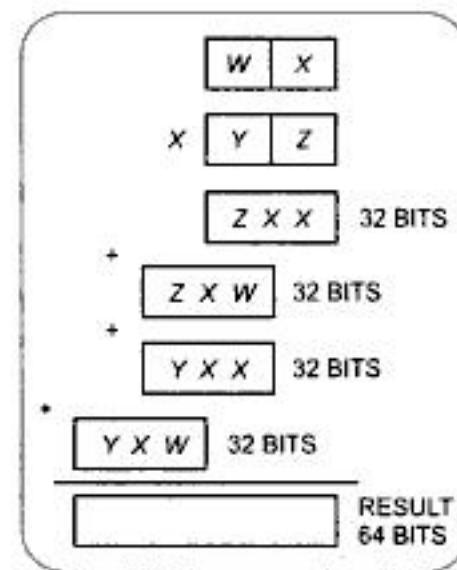


Fig. 5.28 32-bit by 32-bit multiply method for Problem 9.

10. Calculating the factorial of a number, which we did with a recursive procedure in Fig. 5-22, can easily be done with a simple REPEAT-UNTIL structure of the form

```
IF N = 1 THEN
  FACTORIAL = 1
ELSE
  FACTORIAL = 1
REPEAT
  FACTORIAL = FACTORIAL × N
  DECREMENT N
UNTIL N = 0
```

Write an 8086 procedure which implements this algorithm for an N between 1 and 8.

11. a. Show the statement you would use to tell the assembler to make the label BINADD available to other assembly modules.
 b. Show how you would tell the assembler to look for a byte type data item named CONVERSION_FACTOR in a segment named FIXUPS.
12. a. Write an assembler macro which will restore, in the correct order, the registers saved by the macro PUSH_ALL in this chapter.
 b. Write the statement you would use to call the macro you wrote in part a.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

CALL—Call a Procedure

The CALL instruction is used to transfer execution to a subprogram or procedure. There are two basic types of calls, *near* and *far*. A near call is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a near CALL instruction, it decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL onto the stack. This offset saved on the stack is referred to as the *return address*, because this is the address that execution will return to after the procedure executes. A near CALL instruction will also load the instruction pointer with the offset of the first instruction in the procedure. A RET instruction at the end of the procedure will return execution to the instruction after the call by copying the offset saved on the stack back to IP.

A far call is a call to a procedure which is in a different segment from the one that contains the CALL instruction. When the 8086 executes a far call, it decrements the stack pointer by 2 and copies the contents of the CS register to the stack. It then decrements the stack pointer by 2 again and copies the offset of the instruction after the CALL instruction to the stack. Finally, it loads CS with the segment base of the segment which contains the procedure, and loads IP with the offset of the first instruction of the procedure in that segment. A RET instruction at the end of the procedure will return execution to the next instruction after the CALL by restoring the saved values of CS and IP from the stack.

EXAMPLES:

CALL MULTO ; A direct within-segment (near or intra-segment) call. MULTO is the name of the procedure. The assembler determines the displacement of MULTO from the instruction after the CALL and codes this displacement in as part of the instruction.

CALL BX ; An indirect within-segment near or intrasegment call. BX contains the offset of the first instruction of the procedure. Replaces contents of IP with contents of register BX.

CALL WORD PTR(BX) ; An indirect within-segment near or Intrasegment call. Offset of first instruction of procedure is in two memory addresses in DS. Replaces contents of IP with contents of word memory location in DS pointed to by BX.

CALL SMART-DIVIDE ; A direct call to another segment—far or intersegment call. SMART_DIVIDE is the name of the procedure. The procedure must be declared far with SMART_DIVIDE PROC FAR at its

start (see Chapter 5). The assembler will determine the code segment base for the segment which contains the procedure and the offset of the start of the procedure. It will put these values in as part of the instruction code.

CALL DWORD PTR[BX] ; An indirect call to another segment—far or intersegment call. New values for CS and IP are fetched from four memory locations in DS. The new value for CS is fetched from [BX] and [BX + 1]; the new IP is fetched from [BX + 2] and [BX + 3].

CBW—Convert Signed Byte to Signed Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be the *sign extension* of AL. The CBW operation must be done before a signed byte in AL can be divided by another signed byte with the IDIV instruction. CBW affects no flags.

EXAMPLE:

; AX = 00000000 10011011 = -155 decimal

CBW ; Convert signed byte in AL to signed word in AX
; Result: AX = 11111111 10011011 = -155
; decimal

For further examples of the use of CBW, see the IDIV instruction description.

CLC—Clear the Carry Flag (CF)

This instruction resets the carry flag to 0. No other flags are affected.

EXAMPLE:

CLC

CLD—Clear Direction Flag

This instruction resets the direction flag to 0. No other flags are affected. If the direction flag is reset, SI and DI will automatically be incremented when one of the string instructions, such as MOVS, CMPS, or SCAS, executes. Consult the string instruction descriptions for examples of the use of the direction flag.

EXAMPLE:

CLD ; Clear direction flag so that string pointers
; autoincrement after each string operation

CLI—Clear Interrupt Flag

This instruction resets the interrupt flag to 0. No other flags are affected. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input. The CLI instruction, however, has no effect on the nonmaskable interrupt input, NMI.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

DIVBH ; AX/BH. AL = quotient = 5EH = 94 decimal
; AH = remainder = 65H = 101 decimal

Since the remainder is greater than half of the divisor, the actual quotient is closer to 5FH than to the 5EH produced. However, as indicated before, the quotient is always truncated to the next lower integer rather than rounded to the closest integer. If you want to round the quotient, you can compare the remainder with (divisor/2) and add 1 to the quotient if the remainder is greater than (divisor/2).

ESC—Escape

This instruction is used to pass instructions to a coprocessor, such as the 8087 math coprocessor which shares the address and data bus with an 8086. Instructions for the coprocessor are represented by a 6-bit code embedded in the escape instruction. As the 8086 fetches instruction bytes, the coprocessor also catches these bytes from the data bus and puts them in its queue. However, the coprocessor treats all the normal 8086 instructions as NOPs. When the 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6-bit code specified in the instruction. In most cases the 8086 treats the ESC instruction as a NOP. In some cases the 8086 will access a data item in memory for the coprocessor. A section in Chapter 11 describes the operation and use of the ESC instruction.

HLT—Halt Processing

The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The only ways to get the processor out of the halt state are with an interrupt signal on the INTR pin, an interrupt signal on the NMI pin, or a reset signal on the RESET input. See Chapter 7 for further details about the halt state.

IDIV—Divide by Signed Byte or Word—IDIV Source

This instruction is used to divide a signed word by a signed byte, or to divide a signed doubleword (32 bits) by a signed word.

When dividing a signed word by a signed byte, the word must be in the AX register. The divisor can be in an 8-bit register or a memory location. After the division, AL will contain the signed result (quotient), and AH will contain the signed remainder. The sign of the remainder will be the same as the sign of the dividend. If an attempt is made to divide by 0, the quotient is greater than 127 (7FH), or the quotient is less than -127 (81H), the 8086

will automatically do a type 0 interrupt. Interrupts are discussed in Chapter 8. For the 80186, 80286, etc., this range is -128 to +127.

When dividing a signed doubleword by a signed word, the most significant word of the dividend (numerator) must be in the DX register, and the least significant word of the dividend must be in the AX register. The divisor can be in any other 16-bit register or memory location. After the division, AX will contain a signed 16-bit quotient, and DX will contain a signed 16-bit remainder. The sign of the remainder will be the same as the sign of the dividend. Again, if an attempt is made to divide by 0, the quotient is greater than +32,767 (7FFFH), or the quotient is less than -32,767 (8001H), the 8086 will automatically do a type 0 interrupt. For the 80186, 80286, etc., this range is -32,768 to +32,767.

If the divisor does not divide evenly into the dividend, the quotient will be truncated, not rounded. An example below illustrates this. All flags are undefined after an IDIV.

If you want to divide a signed byte by a signed byte, you must first put the dividend byte in AL and fill AH with copies of the sign bit from AL. In other words, if AL is positive (sign bit = 0), then AH should be filled with 0's. If AL is negative (sign bit = 1), then AH should be filled with 1's. The 8086 Convert Byte to Word instruction, CBW, does this by copying the sign bit of AL to all the bits of AH. AH is then said to contain the "sign extension of AL." Likewise, if you want to divide a signed word by a signed word, you must put the dividend word in AX and extend the sign of AX to all the bits of DX. The 8086 Convert Word to Doubleword instruction, CWD, will copy the sign bit of AX to all the bits of DX.

EXAMPLES (CODING):

| | |
|---------------------------|--|
| IDIVBL | ; Signed word in AX/signed byte in BL |
| IDIVBP | ; Signed doubleword in DX and AX/signed word in BP |
| IDIV BYTE PTR [BX] | ; AX/byte at offset [BX] in DS |
| MOVAL, DIVIDEND | ; Position byte dividend |
| CBW | ; Extend sign of AL into AH |
| IDIVDIVISOR | ; Divide by byte divisor |

EXAMPLES (NUMERICAL):

; A signed word divided by a signed byte
; AX = 00000011 10101011 = 03ABH
; = 39 decimal
; BL = 11010011 = D3H = -2DH



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

EXAMPLES:

CMP AX,4371H ; Compare (AX – 4371H)
JAE RUN_PRESS ; Jump to label RUN_PRESS if AX
; above or equal to 4371H

CMP AX,4371H ; Compare (AX – 4371H)
JNB RUN_PRESS ; Jump to label RUN_PRESS if AX
; not below 4371H

ADD AL,BL ; Add two bytes. If result within
JNC OK ; acceptable range, continue

JB/JC/JNAE—Jump if Below/Jump if Carry/Jump if Not Above or Equal

These three mnemonics represent the same instruction. The terms *above* and *below* are used when referring to the magnitude of unsigned numbers. The number 0111 is above the number 0010. If, after a compare or some other instruction which affects flags, the carry flag is a 1, this instruction will cause execution to jump to a label given in the instruction. If CF is 0, the instruction will have no effect on program execution. The destination label for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the JB. JB/JC/JNAE affects no flags. For further explanation of Conditional Jump instructions, see Chapter 4.

EXAMPLES:

CMP AX,4371H ; Compare (AX – 4371H)
JB RUN_PRESS ; Jump to label RUN_PRESS if
; AX below 4371H

ADD BX,CX ; Add two words and jump
JCE ERROR_FIX ; to label ERROR_FIX if CF = 1

CMP AX,4371H ; Compare (AX – 4371H)
JNAE RUN_PRESS ; Jump to label RUN_PRESS if
; AX not above or equal to 4371H

JBE/JNA—Jump if Below or Equal/Jump if Not Above

These two mnemonics represent the same instruction. The terms *above* and *below are* used when referring to the magnitude of unsigned numbers. The number 0111 is above the number 0010. If, after a compare or some other instruction which affects flags, either the zero flag or the carry flag is 1, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are both 0, the instruction will have no effect on program execution. The destination label for the jump must be in the range of –128 bytes to +127 bytes from the address

of the instruction after the JBE. JBE/JNA affects no flags. For further explanation of Conditional Jump instructions, see Chapter 4.

EXAMPLES:

CMP AX,4371H ; Compare (AX – 4371H)
JBE RUN_PRESS ; Jump to label RUN_PRESS if AX
; below or equal to 4371H

CMP AX,4371H ; Compare (AX – 4371H)
JNA RUN_PRESS ; Jump to label RUN_PRESS if AX
; not above 4371H

JCXZ—Jump if the CX Register is Zero

This instruction will cause a jump to a label given in the instruction if the CX register contains all 0's. If CX does not contain all 0's, execution will simply proceed to the next instruction. Note that this instruction does not look at the zero flag when it decides whether to jump or not. The destination label for this instruction must be in the range of –128 to +127 bytes from the address of the instruction after the JCXZ instruction. JCXZ affects no flags.

EXAMPLE:

| | |
|-------------------------|-------------------------------|
| JCXZ SKIP_LOOP | ; If CX = 0, skip the process |
| NXT:SUB [BX],07H | ; Subtract 7 from data value |
| INCBX | ; Point to next value |
| LOOP NXT | ; Loop until CX = 0 |
| SKIP_LOOP: | ; Next instruction |

JE/JZ—Jump if Equal/Jump if Zero

These two mnemonics represent the same instruction. If the zero flag is set, then this instruction will cause execution to jump to a label given in the instruction. If the zero flag is not 1, then execution will simply go on to the next instruction after JE or JZ. The destination label for the JE/JZ instruction must be in the range of –128 to +127 bytes from the address of the instruction after the JE/JZ instruction. JE/JZ affects no flags.

EXAMPLES:

| | |
|-----------------------|---------------------------|
| NXT:CMP BX,DX | , Compare (BX-DX) |
| JE DONE | ; Jump to DONE if BX = DX |
| SUB BX,AX | ; Else subtract AX |
| INC CX | ; Increment counter |
| JMP NXT | ; Check again |
| DONE:MOV AX,CX | ; Copy count to AX |
| INAL,8FH | ; Read data from port 8FH |



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

instruction was included in the 8086 instruction set so that the 8085 PUSH PSW instruction could easily be simulated on an 8086. LAHF changes no flags.

LDS—Load Register and DS with Words from Memory—LDS Register, Memory Address of First Word

This instruction copies a word from two memory locations into the register specified in the instruction. It then copies a word from the next two memory locations into the DS register. LDS is useful for pointing SI and DS at the start of a string before using one of the string instructions. LDS affects no flags.

EXAMPLES:

LDS BX,[4326] ; Copy contents of memory at displacement 4326H in DS to BL, contents of 4327H to BH. Copy contents at displacement of 4328H and 4329H in DS to DS register.

LDS SI,STRING_POINTER ; Copy contents of memory at displacements STRING_POINTER and STRING_POINTER + 1 in DS to SI register. Copy contents of memory at displacements STRING_POINTER + 2 and STRING_POINTER + 3 in DS to DS register. DS:SI now points at start of desired string.

LEA—Load Effective Address—LEA Register, Source

This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register. LEA changes no flags.

EXAMPLES:

LEA BX,PRICES ; Load BX with offset of PRICES in DS

LEA BP,SS:STACK_TOP ; Load BP with offset of ; STACK_TOP in SS

LEA CX,[BX][DI] ; Load CX with EA = (BX) + (DI)

A program example will better show the context in which this instruction is used. If you look at the program in Fig. 4.21c, you will see that PRICES is an array of bytes in a segment called ARRAYS. The instruction LEA BX, PRICES will load the displacement of the first element of PRICES directly into BX. The instruction MOV AL,[BX] can then be used to bring an element from the array into AL. After one element in the array is processed, BX is incremented to point to the next element in the array.

LES—Load Register and ES with Words from Memory—LES Register, Memory Address of First Word

This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory locations is copied into the specified register, and the word from the next two memory locations is copied into the ES register. LES can be used to point DI and ES at the start of a string before a string instruction is executed. LES affects no flags.

EXAMPLES:

LES BX,[789AH] ; Contents of memory at displacements 789AH and 789BH in DS copied to BX. Contents of memory at displacements 789CH and 789DH in DS copied to ES register.

LES DI,[BX] ; Copy contents of memory at offset [BX] and offset [BX + 1] in DS to DI register. Copy contents of memory at offsets [BX + 2] and [BX + 3] to ES register.

LOCK—Assert Bus Lock Signal

Many microcomputer systems contain several microprocessors. Each microprocessor has its own local buses and memory. The individual microprocessors are connected together by a system bus so that each can access system resources such as disk drives or memory. Each microprocessor takes control of the system bus only when it needs to access some system resource. The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus while it is in the middle of a critical instruction which uses the system bus. The LOCK prefix is put in front of the critical instruction. When an instruction with a LOCK prefix executes, the 8086 will assert its bus lock signal output. This signal is connected to an external bus controller device, which then prevents any other processor from taking over the system bus. LOCK affects no flags. See Chapter 11 for further discussion of this.

EXAMPLE:

LOCK XCHG SEMAPHORE,AL ; The XCHG instruction requires two bus accesses. The LOCK prefix prevents another processor from taking control of the system bus between the two accesses.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You may want to try this with a couple of numbers to convince yourself that it gives the same result as the invert each bit and add 1 algorithm. As shown in some of the following examples, the NEG instruction is useful for changing the sign of a signed word or byte. An attempt to NEG a byte location containing -128 or a word location containing -32,768 will produce no change in the destination contents because the maximum positive signed number in 8 bits is +127 and the maximum positive signed number in 16-bits is +32,767. OF will be set to indicate that the operation could not be done. The NEG instruction updates AF, CF, SF, PF, ZF, and OF.

EXAMPLES:

| | |
|--------------------------|--|
| NEG AL | ; Replace number in AL with its ; 2's complement |
| NEG BX | ; Replace word in BX with its ; 2's complement |
| NEG BYTE PTR [BX] | ; Replace byte at offset [BX] in ; DS with its 2's complement |
| NEG WORD PTR [BP] | ; Replace word at offset [BP] in ; SS with its 2's complement |

NOTE: The BYTE PTR and WORD PTR directives are required in the last two examples to tell the assembler whether to code the instruction for a byte operation or a word operation. The [BP] reference by itself does not indicate the type of the operand.

NOP—Perform No Operation

This instruction simply uses up three clock cycles and increments the instruction pointer to point to the next instruction. NOP affects no flags. The NOP instruction can be used to increase the delay of a delay loop, as shown in Fig. 4.27a. When hand coding, a NOP can also be used to hold a place in a program for an instruction that will be added later.

NOT—Invert Each Bit of Operand—NOT Destination

The NOT instruction inverts each bit (forms the 1's complement) of the byte or word at the specified destination. The destination can be a register or a memory location specified by any one of the 24 addressing modes shown in Fig. 3.8. No flags are affected by the NOT instruction.

EXAMPLES:

| | |
|---------------|---|
| NOT BX | ; Complement contents of BX register |
|---------------|---|

NOT BYTE PTR [BX] ; Complement memory byte at
; offset [BX] in data segment

OR—Logically OR Corresponding Bits of Two Operands—OR Destination, Source

This instruction ORs each bit in a source byte or word with the corresponding bit in a destination byte or word. The result is put in the specified destination. The contents of the specified source will not be changed. The result for each bit will follow the truth table for a two-input OR gate. In other words, a bit in the destination will become a 1 if that bit is a 1 in the source operand *or* that bit is a 1 in the original destination operand. Therefore, a bit in the destination operand can be set to a 1 by simply ORing that bit with a 1 in the same bit of the source operand. A bit ORed with 0 is not changed.

The source operand can be an immediate number, the contents of a register, or the contents of a memory location specified by one of the 24 addressing modes shown in Fig. 3.8. The destination can be a register or a memory location. The source and the destination cannot both be memory locations in the same instruction. CF and OF are both 0 after OR, PF, SF, and ZF are updated by the OR instruction. AF is undefined after OR. Note that PF has meaning only for the lower 8 bits of a result.

EXAMPLES (SYNTAX):

| | |
|-----------------------------|--|
| OR AH,CL | ; CL ORed with AH, result in AH. ; CL not changed |
| OR BP,SI | ; SI ORed with BP, result in BP. ; SI not changed |
| OR SI,BP | ; BP ORed with SI, result in SI. ; BP not changed |
| OR BL,80H | ; BL ORed with immediate 80H. ; Set MSB of BL to a 1 |
| OR CX, TABLE[BX][SI] | ; CX ORed with word from ; effective address TABLE[BX][SI] ; in data segment. Word in ; memory is not changed |

EXAMPLE (NUMERICAL):

OR CX,0FF00H ; CX = 00111101 10100101
; OR CX with immediate FF00H
; Result in CX = 11111111 10100101
; Note upper byte now all 1's, lower
; byte unchanged
; CF = 0, OF = 0, PF = 1, SF = 1,
; ZF = 0



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



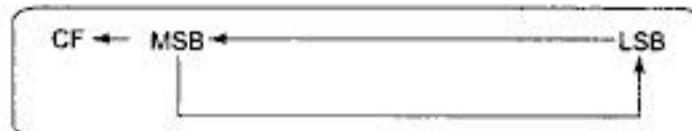
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

ROL—Rotate All Bits of Operand Left, MSB to LSB—ROL Destination, Count

This instruction rotates all the bits in a specified word or byte to the left some number of bit positions. The operation can be thought of as circular, because the data bit rotated out of the MSB is circled back into the LSB. The data bit rotated out of the MSB is also copied to CF during ROL. In the case of multiple bit rotates, CF will contain a copy of the bit most recently moved out of the MSB. See the following diagram.



The destination operand can be in a register or in a memory location specified by any one of the 24 addressing modes shown in Fig. 3.8. If you want to rotate the operand one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate more than one bit position, load the desired number in the CL register and put "CL" in the count position of the instruction.

NOTE: The 80186, 80286, 80386, etc., allow you to specify a rotate of up to 32 bit positions with either an immediate number in the instruction or a number in CL.

ROL affects only CF and OF. After ROL, CF will contain the bit most recently rotated out of the MSB. OF will be a 1 after a single bit ROL if the MSB was changed by the rotate.

The ROL instruction can be used to swap the nibbles in a byte or to swap the bytes in a word. It can also be used to rotate a bit into CF, where it can be checked and acted upon by the Conditional Jump instructions JC (Jump if Carry) and JNC (Jump if No Carry).

EXAMPLES (SYNTAX):

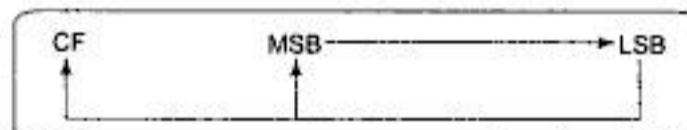
| | |
|-------------------------|--|
| ROL AX,I | ; Word in AX 1 bit position left, ; MSB to LSB and CF |
| MOV CL,04H | ; Load number of bits to rotate in CL |
| ROL BL,CL | ; Rotate BL 4 bit positions ; (swap nibbles) |
| ROL FACTOR[BX],I | ; MSB of word or byte in DS at ; EA = FACTOR[BX] ; 1 bit position left into CF |
| JC ERROR | ; Jump if CF = 1 to error routine |

EXAMPLES (NUMERICAL):

| | |
|------------------|--|
| ROL BH,I | ; CF = 0, BH = 10101110 ; Result: CF,OF = 1, BH = 01011101 |
| ROL BX,CL | ; BX = 01011100 11010011 ; CL = 8, set for 8-bit rotate ; Rotate BX 8 times left (swap bytes) ; CF = 0, BX = 11010011 01011100, ; OF undefined |

ROR—Rotate All Bits of Operand Right, LSB to MSB—ROR Destination, Count

This instruction rotates all the bits of the specified word or byte some number of bit positions to the right. The operation is described as a rotate rather than a shift because the bit moved out of the LSB is rotated around into the MSB. To help visualize the operation, think of the operand as a loop with the LSB connected around to the MSB. The data bit moved out of the LSB is also copied to CF during ROR. See the following diagram. In the case of multiple-bit rotates, CF will contain a copy of the bit most recently moved out of the LSB.



The destination operand can be in a register or in a memory location specified by any one of the 24 addressing modes shown in Fig. 3.8. If you want to rotate the operand one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate more than one bit position, load the desired number in the CL register and put "CL" in the count position of the instruction.

NOTE: The 80186, 80286, 80386, etc., allow you to specify a rotate of up to 32 bit positions with either an immediate number or a number in CL.

ROR affects only CF and OF. After ROR, CF will contain the bit most recently rotated out of the LSB. For a single-bit rotate, OF will be a 1 after ROR if the MSB is changed by the rotate.

The ROR instruction can be used to swap the nibbles in a byte or to swap the bytes in a word. It can also be used to rotate a bit into CF, where it can be checked and acted upon by the Conditional Jump instructions JC (Jump if Carry) and JNC (Jump if No Carry).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

MOV CX,80 ; CX used as element counter
CLD ; Clear DF so DI autoincrements
REPNE SCAS TEXT_STRING
; Compare byte in string with
; byte in AL

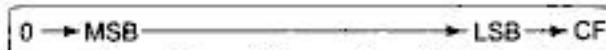
```

NOTE: Scanning is repeated as long as the bytes are not equal and the end of the string has not been reached. If a carriage return 0DH is found, ZF = 1, and DI will point at the next byte after the carriage return in the string. If a carriage return is not found, then CX = 0 and ZF = 0. The assembler uses the name of the string to determine whether the string is of type byte or type word. Instead of using the name, you can tell the assembler the type of string directly by using the mnemonic SCASB for a byte string and SCASW for a word string.

SHL—See Heading SAL

SHR—Shift Operand Bits Right, Put Zero in MSB(s)—SHR Destination, Count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted right out of the MSB position, a 0 is put in its place. The bit shifted out of the LSB position goes to CF. In the case of a multiple-bit shift, CF will contain the bit most recently shifted in from the LSB. Bits shifted into CF previously will be lost. See the following diagram.



The destination operand can be a byte or a word in a register or in a memory location specified by any one of the 24 addressing modes shown in Fig. 3.8.

If the desired number of shifts is one, this can be specified by putting a 1 in the count position of the instruction. For shifts of more than one bit position, the desired number of shifts is loaded into the CL register, and CL is put in the count position of the instruction.

NOTE: The 80186, 80286, 80386, etc., allow you to specify a shift of up to 32 bit positions with either an immediate number in the instruction or a number in CL.

The flags are affected by SHR as follows: CF contains the bit most recently shifted in from the LSB. For a count of one, OF will be a 1 if the two MSBs are not both 0's. For multiple-bit shifts, OF is meaningless. SF and ZF will be updated to show the condition of the destination. PF will have meaning only for the lower 8 bits of the destination. AF is undefined.

The SHR instruction can be used to divide an unsigned binary number by a power of 2. Shifting a binary number one bit position to the right and putting 0 in the MSB divides the number by 2. Shifting the number two bit positions to the right divides it by 4. Shifting it three bit positions to the right divides it by 8, etc. When an odd number is divided with this method, the result will be truncated. In other words, dividing 7 by 2 will give a result of 3.

EXAMPLES (SYNTAX):

SHR BP,I ; Shift word in BP one bit position right,
; 0 in MSB

MOV CL,03H ; Load desired number of shifts into CL
SHR BYTE PTR [BX] ; Shift byte in DS at offset
; [BX] 3 bits right.
; 0's in 3 MSBs

; Example of SHR used to help unpack
; two BCD digits in AL to BH and BL
MOV BL,AL
AND BL,0FH ; Copy packed BCD to BL
; Mask out upper nibble. Low BCD
; digit now in BL

MOV CL,04H ; Load count for shift in CL
SHR AL,CL ; Shift AL four bit positions right and
; put 0's in upper 4 bits

MOV BH,AL ; Copy upper BCD nibble to BH

EXAMPLES (NUMERICAL):

SHR SI,I ; SI = 10010011 10101101, CF = 0
; Result: SI = 01001001 11010110
; CF = 1, OF = 1, PF = ?, SF = 0, ZF = 0

STC—Set the Carry Flag to a 1

STC does not affect any other flags.

STD—Set the Direction Flag to a 1

STD is used to set the direction flag to a 1 so that SI and/or DI will automatically be decremented to point to the next string element when one of the string instructions executes. If the direction flag is set, SI and/or DI will be decremented by 1 for byte strings, and by 2 for word strings. STD affects no other flags. Please refer to Chapter 5 and the discussion of the REP prefix in this chapter for examples of the use of this instruction.

STI—Set Interrupt Flag (IF)

Setting the interrupt flag to a 1 enables the INTR interrupt input of the 8086. The instruction will not take effect until after the next instruction after STI. When the INTR input is enabled, an interrupt signal on this input will then cause the 8086 to interrupt program execution, push the return



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

blocks of 10 bytes each and initialize all 320 bytes to 00 when the program is loaded into memory to be run.

DW—DEFINE WORD

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement MULTIPLIER DW 437AH, for example, declares a variable of type word named MULTIPLIER. The statement also tells the assembler that the variable MULTIPLIER should be initialized with the value 437AH when the program is loaded into memory to be run. Refer to Chapter 3 for further discussion of the DW directive and how you can access variables named with a DW in your programs. Here are a few more examples of DW statements.

THREE_LITTLE_WORDS DW 1234H,3456H,5678H

; Declare array of 3 words and initialize with specified values.

STORAGE DW 100 DUP(0) ; Reserve an array of 100 words of memory and initialize all 100 words with 0000. Array is named STORAGE.

STORAGE DW 100 DUP(?) ; Reserve 100 words of storage in memory and give it the name STORAGE, but leave the words uninitialized.

END—END PROGRAM

The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statements after an END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive.

ENDP—END PROCEDURE

This directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. This directive, together with the procedure directive, PROC, is used to “bracket” a procedure. Here’s an example.

SQUARE_ROOT PROC ; Start of procedure
; Procedure instruction
; statements

SQUARE_ROOT ENDP ; End of procedure

Chapter 5 shows more examples and describes how procedures are written and called.

ENDS—END SEGMENT

This directive is used with the name of a segment to indicate the end of that logical segment. ENDS is used with the SEGMENT directive to “bracket” a logical segment containing instructions or data. Here’s an example.

| | | |
|---------------------|---|--------------------------|
| CODE SEGMENT | ; | Start of logical segment |
| | ; | containing code |
| | ; | Instruction statements |
| CODE ENDS | ; | End of segment named |
| | ; | CODE |

EQU—EQUATE

EQU is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it will replace the name with the value or symbol you equated with that name. Suppose, for example, you write the statement CORRECTION_FACTOR EQU 03H at the start of your program, and later in the program you write the instruction statement ADD AL, CORRECTION_FACTOR. When it codes this instruction statement, the assembler will code it as if you had written the instruction ADD AL,03H. The advantage of using EQU in this manner is that if CORRECTION_FACTOR is used 27 times in a program, and you want to change the value, all you have to do is change the EQU statement and reassemble the program. The assembler will automatically put in the new value each time it finds the name CORRECTION_FACTOR. If you had used 03H instead of the EQU approach, then you would have had to try to find all 27 instructions and change them yourself. Here are some more examples.

| | | |
|----------------------------------|---|--------------------|
| CONTROL_WORD EQU 11001001 | ; | Replacement |
| MOV AL,CONTROL_WORD | ; | assignment |
| DECIMAL_ADJUST EQU DAA | ; | Create clearer |
| | ; | mnemonic for DAA |
| ADD AL,BL | ; | Add BCD |
| | ; | numbers |
| DECIMAL_ADJUST | ; | Keep result in BCD |
| | ; | format |
| STRING_START EQU [BX] | ; | Give name to [BX] |

EVEN—ALIGN ON EVEN MEMORY ADDRESS

As the assembler assembles a section of data declarations or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

program, the assembler will automatically reserve 2 bytes for the displacement. Using the SHORT operator saves 1 byte of memory by telling the assembler that it needs to reserve only 1 byte for this particular jump. In order for this to work, the destination must be in the range of -128 bytes to +127 bytes from the address of the instruction after the jump. The statement JMP SHORT NEARBY_LABEL is an example of the use of SHORT.

determines the number of bytes in the type of the variable. For a byte-type variable, the assembler will give a value of 1. For a word-type variable, the assembler will give a value of 2, and for a doubleword-type variable, it will give a value of 4. The TYPE operator can be used in an instruction such as ADD BX,TYPE WORD_ARRAY, where we want to increment BX to point to the next word in an array of words.

| **TYPE**

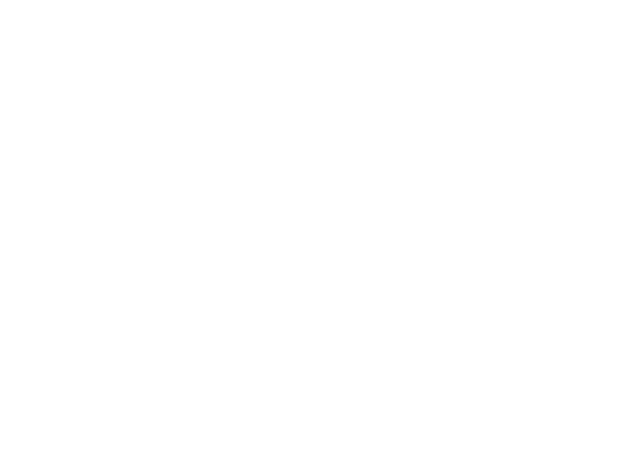
The TYPE operator tells the assembler to determine the type of a specified variable. The assembler actually



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

diagram. Their meanings should become clear to you as we work through the diagram.

The first line to look at in Fig. 7.1b is the *clock waveform*, CLK, at the top. This represents the crystal-controlled clock signal sent to the 8086 from an external clock generator device such as the 8284 shown in the top left corner of Fig. 7.1a. One cycle of this clock is called a *state*. For reference purposes, a state is measured from the falling edge of one clock pulse to the falling edge of the next clock pulse. The time interval labeled T_1 in the figure is an example of a state. Different versions of the 8086 have maximum clock frequencies of between 5 MHz and 10 MHz, so the minimum time for one state will be between 100 and 200 ns, depending on the part used and the crystal used.

A basic microprocessor operation such as reading a byte from memory or writing a byte to a port is called a *machine cycle*. The times labeled T_{CY} in Fig. 7.1b are examples of machine cycles. As you can see in the figure, a machine cycle consists of several states.

The time a microprocessor requires to fetch and execute an entire instruction is referred to as an *instruction cycle*. An instruction cycle consists of one or more machine cycles.

To summarize this, an instruction cycle is made up of machine cycles, and a machine cycle is made up of states. The time for a state is determined by the frequency of the clock signal. In this section we discuss the activities that occur on the 8086 microcomputer buses during a read machine cycle.

The best way to analyze a timing diagram such as the one in Fig. 7.1b is to think of time as a vertical line moving from left to right across the diagram. With this technique you can easily see the sequence of activities on the signal lines as you move your imaginary time line across the waveforms.

During T_1 , of a read machine cycle the 8086 first asserts the M/IO signal. It will assert this signal high if it is going to do a read from memory during this cycle, and it will assert M/IO low if it is going to do a read from a port during this cycle. The timing diagram in Fig. 7.1b shows two crossed waveforms for the M/IO signal because the signal may be going low or going high for a read cycle. The point where the two waveforms cross indicates the time at which the signal becomes valid for this machine cycle. Likewise, in the rest of the timing diagram, crossed lines are used to represent the time when information on a line or group of lines is changed.

After asserting M/IO, the 8086 sends out a high on the Address Latch Enable signal (ALE). This signal is connected to the enable input (STB) of the 74S373 octal latches, as shown in Fig. 7.1a, so these latches will be

enabled when ALE is high. As you can also see in Fig. 7.1a, the data inputs of these latches are connected to the 8086 AD0–AD15, A16–A19, and Bus High Enable (BHE) lines. After the 8086 asserts ALE high, it sends out on these lines the address of the memory location that it wants to read. Since the latches are enabled by ALE being high, this address information passes through the latches to their outputs. The 8086 then makes the ALE output low, which disables the latches. The address held on the latch outputs travels along the address bus to memory and port devices.

Note in the timing diagram in Fig. 7.1b how the activity on the ADDR/DATA lines is represented. The first point at which the two waveforms cross represents the time at which the 8086 has put a valid address on these lines. These two waveforms *do not* indicate that all 16 lines are going high or going low at this point.

After ALE goes low, the address information is held on the latches, so the 8086 no longer needs to send out the addresses. Therefore, as shown by a dashed line on the ADDR/DATA line in Fig. 7.1b, the 8086 floats the AD0–AD15 lines so that they can be used to input data from memory or from a port. At about the same time, the 8086 also removes the BHE and A16–A19 information from the upper lines and sends out some status information on those lines.

The 8086 is now ready to read data from the addressed memory location or port, so near the end of state T_2 the 8086 asserts its RD signal low. If you trace the connection of the RD signal in Fig. 7.1a, you should see that this signal is used to enable the addressed memory device or port device. When enabled, the addressed device will put a byte or word of data on the data bus. In other words, asserting the RD signal low causes the addressed device to put data on the data bus. This cause-and-effect relationship is shown on the timing diagram in Fig. 7.1b by an arrow going from the falling edge of RD to the “bus reserved for data in” section of the ADDR/DATA waveforms. The bubble on the tail of the arrow is always put on the signal transition or level that causes some action, and the point of the arrow always indicates the action caused. Arrows of this sort are only used to show the effect a signal from one device will have on another device. They are not usually used to indicate signal cause and effect within a device.

Now, referring to Fig. 7.1b again, find the section of the AD0–AD15 waveform marked off as memory access time near the bottom of the diagram. This time represents the time it takes for the memory to output valid data after it receives an address and an RD signal. If the access time for a memory device is too long, the memory will



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Personal computers can be adapted to function as logic analyzers by installing plug-in units such as the MicroCase Inc. μ Analyst 2000 or the Bitwise Designs, Inc. Logic 20.

One method of connecting signal lines to the analyzer inputs is with a *pod* and test clips such as those shown in front of the analyzer in Fig. 7.3. Another method commonly used with microcomputers is a special cable with a plug which is inserted in the microprocessor socket on the circuit board. The microprocessor is plugged into a socket on top of the plug.

Before we describe how to make measurements with a logic analyzer, we will review the basic operation of a logic analyzer.

Review of Logic Analyzer Operation

Figure 7.4 shows a functional block diagram of a simple logic analyzer. Since logic analyzers are used to detect and display only 1's and 0's, a comparator is put on each input. The reference input of the comparator is set for the logic threshold of the devices in the system you are looking at. If you are looking at TTL or CMOS signals, for example, you set the threshold to 1.4 V. The comparators then make sure that the signals to the rest of the analyzer circuitry are clear-cut 1's or 0's.

The analyzer takes a "snapshot" of the logic levels on the data inputs each time it receives a *clock* pulse. The samples are stored in an internal RAM. Different analyzers store between 256 and 1024 samples for each input channel.

As shown by the block diagram in Fig. 7.4, the analyzer can be clocked by an internally produced signal or some external signal. If you are using an analyzer to look at 8086 address and data lines, for example, you could use ALE as a clock signal. The analyzer will then take a sample each time the 8086 puts out an address and pulses ALE. The samples stored in the analyzer memory will then represent a sequence of addresses output by the 8086. As another example, you could clock the analyzer on the RD signal from an 8086. With this clock signal the analyzer will take a sample each time the 8086 does a read operation, so the samples stored in the analyzer memory will represent the sequence of data words read in from memory or from ports.

To make precise timing measurements with an analyzer, you use a clock signal from an internal, crystal-controlled oscillator. In this case the analyzer will take a sample each time a pulse from the internal clock oscillator occurs. If, for example, you choose an internal clock frequency of 50 MHz, the analyzer will take a sample every

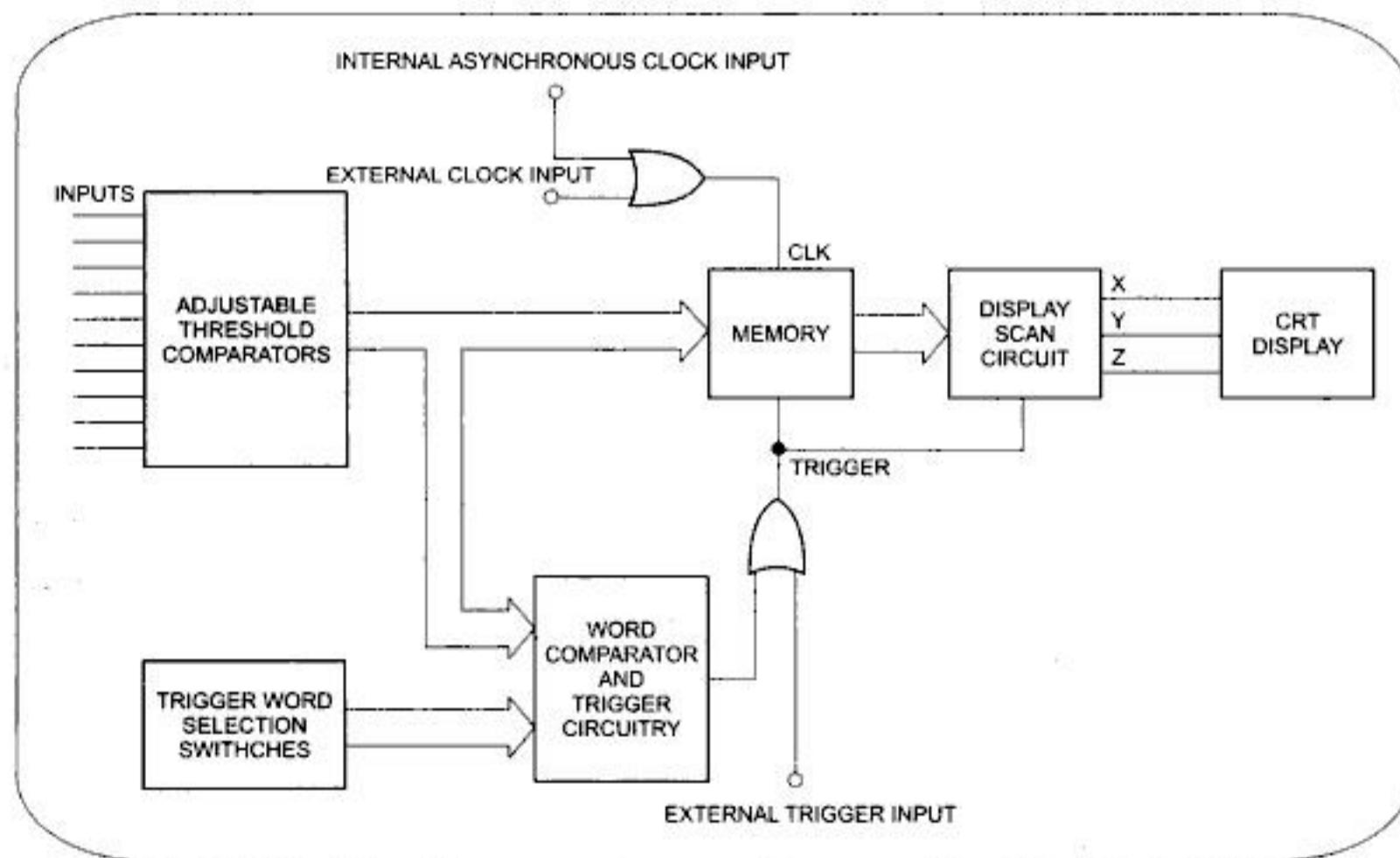


Fig. 7.4 Block diagram of simple logic analyzer.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The next step is to think about what signals to connect to the analyzer data inputs. To determine the time between a valid address from the 8086 and valid data from the memory device, you obviously need to look at the address/data lines. The number that you can trace and display depends on the particular analyzer you are using. The basic Tektronix 1230 analyzer will sample and display 16 channels in the timing mode which you use for this type of measurement. If you have an analyzer such as this, you can connect the analyzer data inputs to the AD0–AD15 pins on the 8086. The upper four address lines, A16–A19, do not change during the execution of this example program, so you don't need to look at them.

When making timing measurements with a logic analyzer, you almost always use the crystal-controlled internal clock to tell the analyzer to take samples so that you know the exact time between samples. For an SDK-86 board the memory access time for the RAM that contains the sample program will be around 100 ns. To get the best possible resolution for your timing measurement, then, you should set the analyzer clock period for the shortest time possible on your analyzer. The shortest period for the Tektronix 1230 with a 16-channel display is 40 ns per clock, so we will use this setting.

To choose the trigger word for this measurement, look again at the timing waveforms in Fig. 7.1b. The address goes out on the data bus and later the data comes back in. Since the address is the first activity, you set the word recognizer in the analyzer to trigger on the first address that is sent out.

Once you do a trace, you can determine the memory access time by counting the number of sample points between the address of 0100H appearing on the bus and the data word of FEEBH appearing on the bus. Figure 7.5b shows an example of this type of display. If your analyzer has cursors, you can position one cursor at the time when the address becomes valid, position the other cursor at the time when the data becomes valid, and read the time difference between the two from the on-screen display.

Note that the resolution of this measurement is only 40 ns, because that is the time between samples. In other words, any changes that take place between sample points will not be shown in the display until the next set of samples is taken. On many analyzers you can specify a shorter sampling period if you reduce the number of signal lines being traced. With the Tektronix 1230, for example, you can use a sample clock with a period as short as 10 ns if you can get by with sampling only four signal lines. We usually start by doing a trace of, for example, all 16 lines, and then from the 16 we choose four which show the

desired transitions. With just these four lines we can decrease the sample period to 10 ns and thereby increase the resolution of our measurement.

We obviously can't describe here all the ways to use a logic analyzer. If you have one, consult the manual for it to learn some of the finer points of its use. Also, the lab manual that is available for use with this book has some exercises to help you gain more skill with an analyzer. The point here was to show you how to use the analyzer as a "window" into what is going on in a system. By carefully choosing the signals you look at, the signal you clock on, and the word you trigger on, you can often solve difficult problems. For this reason, a logic analyzer is a valuable tool when developing a new microcomputer-based product.

Now that you know how to observe and make measurements on microcomputer bus signals, let's take a closer look at an 8086 system.

AN EXAMPLE MINIMUM-MODE SYSTEM, THE SDK-86

The previous sections showed how a clock generator, address latches, and data bus buffers are connected to an 8086 to form what we might call the minimum-mode CPU group. As shown in Fig. 7.1a, this group of ICs generates the address bus, data bus, and control bus signals needed for an 8086 minimum-mode system. In this major section of the chapter we discuss how this CPU group is connected with ROM, RAM, ports, and other devices to form a system. The system we use for this discussion is the *Intel SDK-86 system design kit*, an 8086-based unit suitable for building the prototypes of small microcomputer-based instruments.

Figure 7.6 shows a photograph of an SDK-86 board. From the photograph you can see that, in addition to the microcomputer ICs, the board has a hexadecimal keypad, some 7-segment displays, and a large open area for adding more ROM, RAM, ports, or interface circuitry. A monitor program in ROM on the board allows you to enter, execute, and debug machine code programs using the onboard hex keypad or an external CRT terminal connected to the serial port on the board. The board comes with 2 Kbytes of RAM and sockets where you can add another 2 Kbytes. The board also has six 8-bit parallel ports which you can program to be inputs or outputs. To get a better idea of the hardware functions on the board and the devices used to implement these functions, let's look at the detailed block diagram of the SDK-86 in Fig. 7.7.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

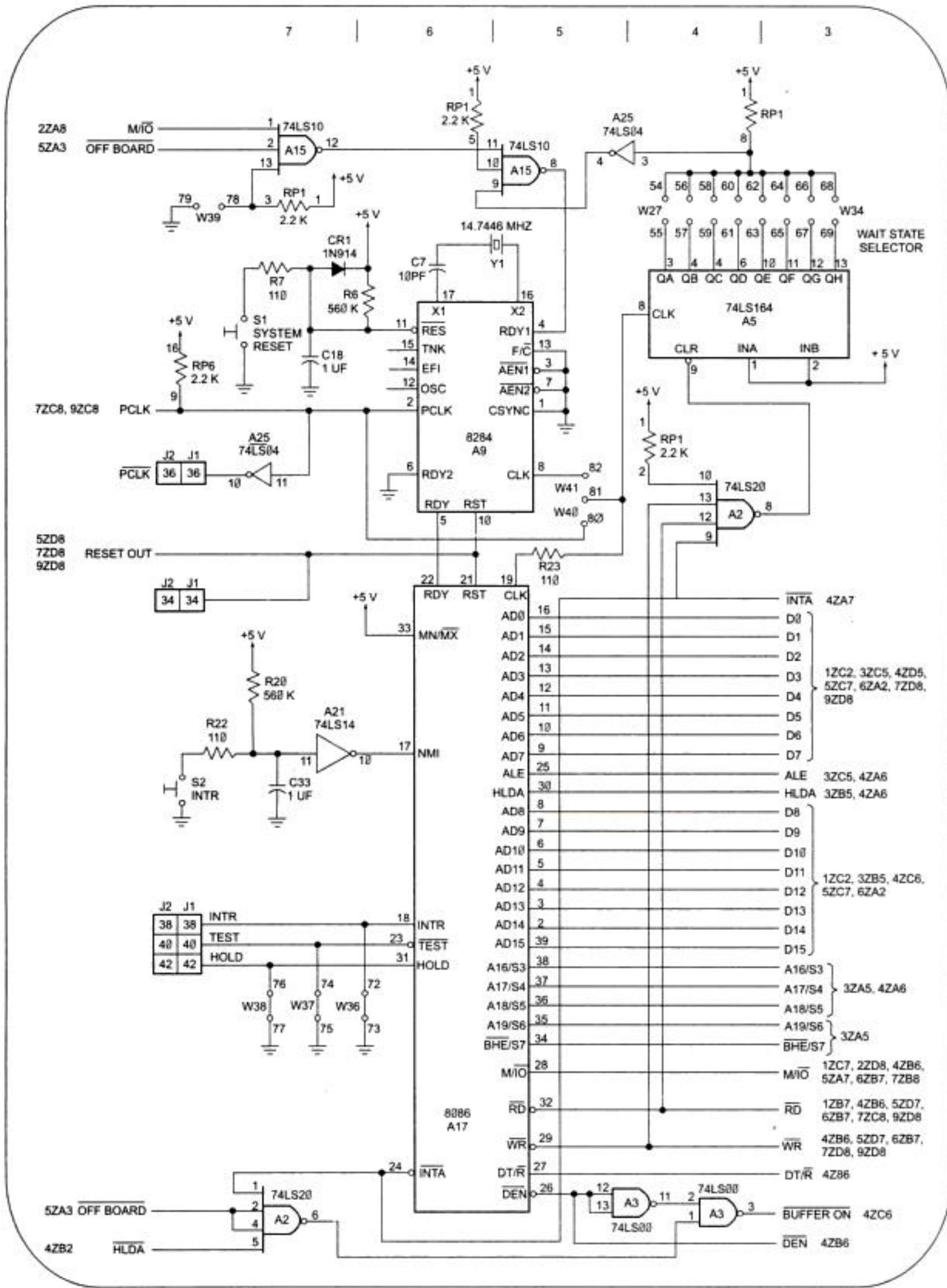


Fig. 7.8 (continued) Sheet 2 of 9.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

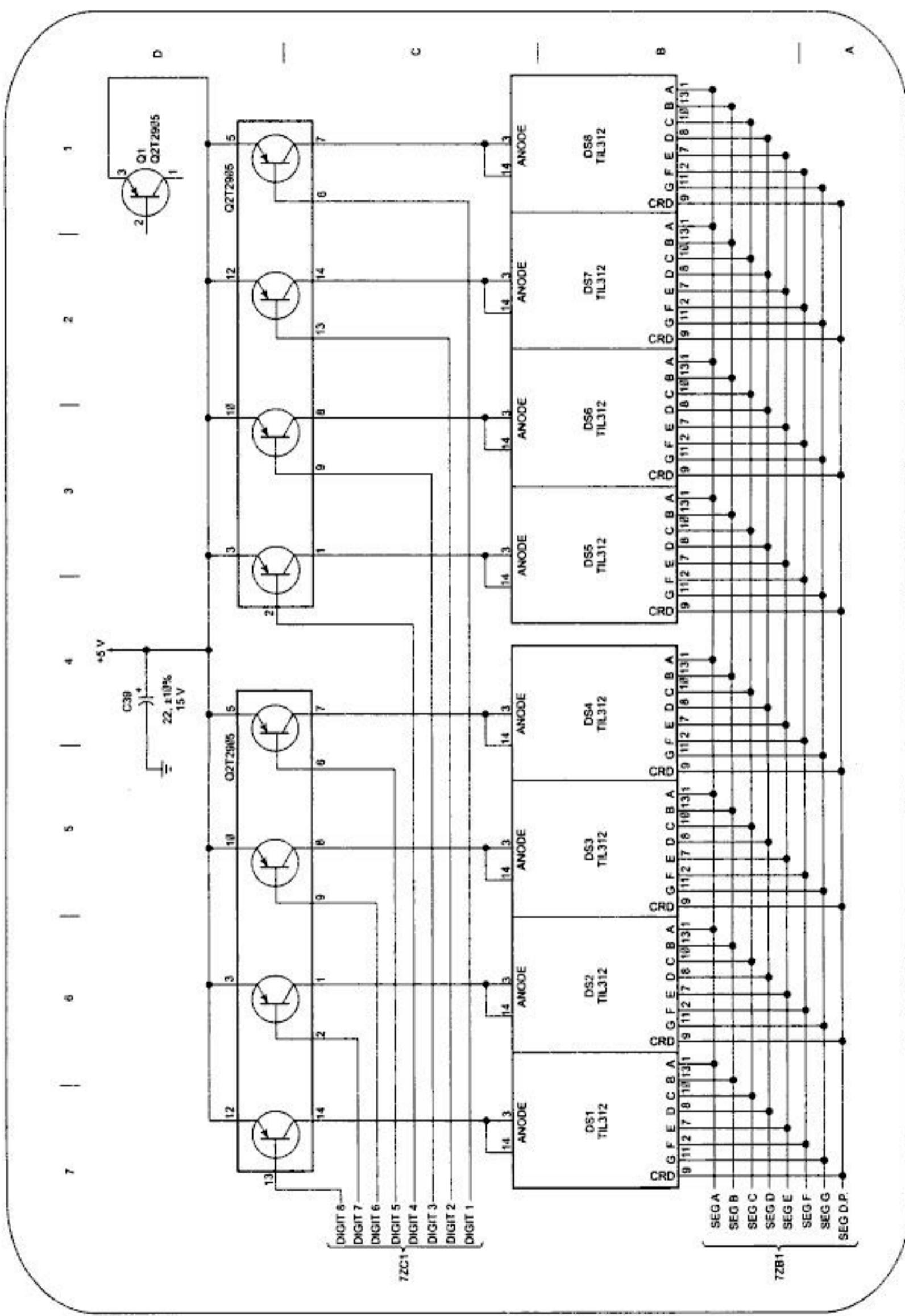


Fig. 7.8 (continued) Sheet 8 of 9.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

How the 8088 Microprocessor Accesses Memory and Ports

Now that we have shown in detail how the 8086 accesses memory and port devices, we can show you how the 8088 does it.

In Chapter 2 we mentioned that the 8088 is the CPU used in the original IBM PC and the IBM PC/XT. The instruction set of the 8088 is identical to that of the 8086, and the registers of the two are the same, but there are two major differences between the two devices. First, the 8088 instruction byte queue is only 4 bytes long instead of 6. Second, and more important, the 8088 memory is not divided into two banks as the 8086 memory is; it consists of a single bank of up to 1,048,576 bytes, as shown in Fig. 7.18.

As you can see, the 8088 has only an 8-bit data bus, D₀-D₇. All the memory devices and ports in an 8088 system are connected onto these eight lines. Address lines A₀ through A₁₉ are used with some decoders to select a desired byte in memory. The 8088 does not produce the BHE signal because it is not needed. This single bank structure means that an 8088 can read or write only a byte at a time. Therefore, an 8088 must always do two machine cycles to read or write a word. The 8088 was designed with an 8-bit data bus so that it would interface more easily with 8-bit memory devices and I/O devices.

8086 Timing Parameters

In previous sections of this chapter, we used generalized timing waveforms such as that in Fig. 7.1b. These diagrams are sufficient to show the sequence of activities on the 8086 buses. However, they are not detailed enough to determine, for example, whether a memory device is fast enough to work in a given 8086 system. To allow you to make precise timing calculations, manufacturers' data books give detailed timing waveforms and lists of timing parameters for each microprocessor. Complete timing information for the 8086 is contained in the data sheet in Appendix A. Figure 7.19, shows some timing waveforms and parameters for an 8086 minimum-mode read machine cycle.

As you look at Fig. 7.19a, remember the *5-minute freak-out rule*. Most of the time there are only a very few of these parameters that you need to worry about. In most systems, for example, you don't need to worry about the clock signal parameters, because an 8284 clock generator and a crystal will be used to produce the clock signal. The frequency of the clock signal from an 8284 is always one-third the resonant frequency of the crystal connected to

it. The 8284 is designed to guarantee the correct clock period, clock time low, clock time high, etc., as long as the correct suffix number part is used. The 8284A, for example, can be used in an 8-MHz system, but a faster part, the 8284A-1, must be used for a system where a 10-MHz clock is desired.

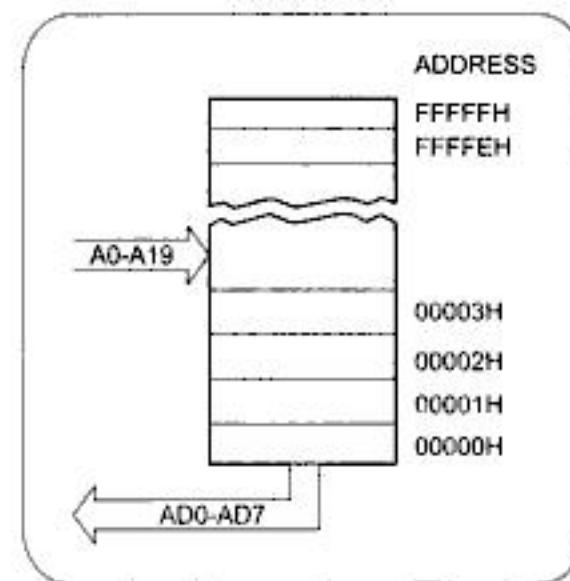


Fig. 7.18 8088 memory structure.

The edges of the clock signal cause operations in the 8086 to occur; therefore, as you can see in Fig. 7.19a, the clock waveform is used as a reference for other times. The timing values for when the 8086 puts out M/IO, addresses, ALE, and control signals, for example, are all specified with reference to an appropriate clock edge.

As we mentioned earlier, one of the main things you use these diagrams and parameters for is to find out whether a particular memory or port device is fast enough to work in a system with a given clock frequency. Here's an example of how you do this.

If you look in zone C5 of sheet 2 of the SDK-86 schematics, you will see that if jumper W41 is installed, the 8086 will receive a 4.9-MHz clock signal from the 8284. If jumper W40 is installed, the 8086 will receive the 2.45-MHz PCLK signal from the 8284. Now, suppose that you want to determine whether the 2716 EPROMs on the SDK-86 board will work correctly with no WAIT states if you install jumper W41 to run the 8086 with the 4.9-MHz clock.

First, you look up the access times for the 2716 EPROM in the appropriate data book. According to an Intel data book, the 2716 has a maximum address to output access time, t_{ACC} , of 450 ns. This means that if the 2716 is already enabled and its output buffers are turned on, it will put valid data on its outputs no more than 450 ns after an address is applied to the address inputs. The 2716 data sheet also gives a chip enable to output access time, t_{CE} ,



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

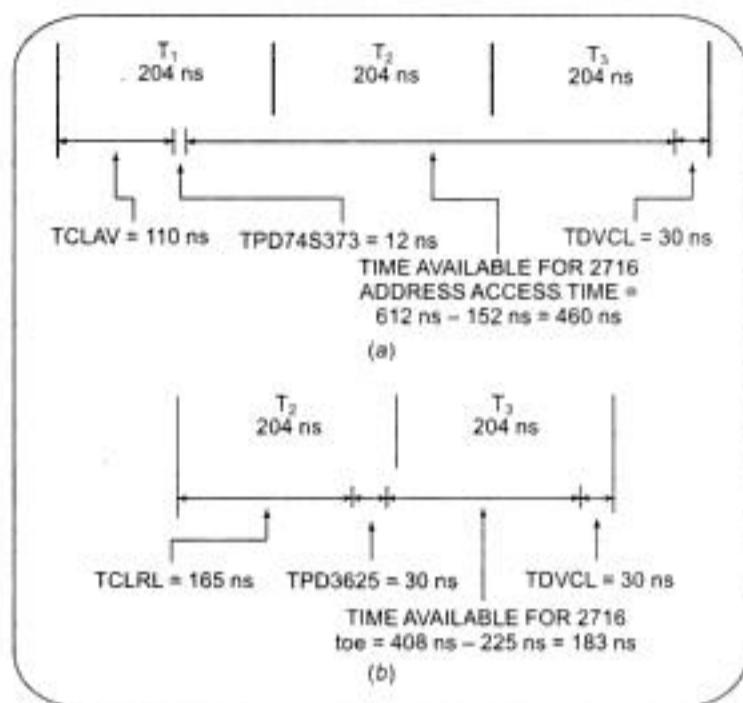


Fig. 7.20 Calculations of maximum allowable access times for 4.9-MHz 8086. (a) Time for t_{ACC} and t_{RD} . (b) Time for t_{OE} .

The following sections describe a series of steps that we have found effective in troubleshooting various microcomputer systems. The first point to impress on your mind about troubleshooting a microcomputer is that a systematic approach is almost always more effective than random poking, probing, and hoping. You don't, for example, want to spend 2 hours troubleshooting a system and finally find that the only problem is that the power supply is putting out only 3 V instead of 5 V. Use the following list of steps or a list of your own each time you have to troubleshoot a microcomputer: (1) Identify the symptoms, (2) make a careful visual and tactile inspection, (3) check the power supply, (4) do a "signal roll call," (5) systematically substitute socketed ICs, and (6) troubleshoot soldered-in ICs. The following paragraphs describe each step.

Identify the Symptoms

Make a list of the symptoms that you find or those that a customer describes to you. Find out, for example, whether the symptom is present immediately when the power is turned on or whether the system must operate for a while before the symptom shows up. If someone else describes the symptoms to you, check them yourself, or have that person demonstrate the symptoms to you. This allows you to check if the problem is with the machine or with how the person is attempting to use the machine.

Make a Careful Visual and Tactile Inspection

This step is good for preventive maintenance as well for finding a current problem. Check for components that

have been or are excessively hot. When touching components to see if any are too hot, do it gently, because a bad IC can get hot enough to give a nasty burn if you keep your finger on it too long.

Check to see that all ICs are firmly seated in their sockets and that the ICs have no bent pins. Vibration can cause ICs to work loose in their sockets. A bent pin may make contact for a while, but after heating, cooling, and vibration, it may no longer make contact. Also, inexpensive IC sockets may oxidize with age and no longer make good contact.

Check for broken wires and loose connectors. A thin film of dust, etc., may form on printed-circuit-board edge connectors and prevent them from making dependable contact. The film can be removed by gently rubbing the edge connector fingers with a cleaning pad available for this purpose. If the microcomputer has ribbon cables, check to see if they have been moved around or stressed. Ribbon cables have small wires that are easily broken. If you suspect a broken conductor in a ribbon cable, you can later make an ohmmeter check to verify your suspicions.

Check the Power Supply

From the manual for the microcomputer, determine the power supply voltages. Check the supply voltage(s) directly on the appropriate pins of some ICs to make sure the voltage is actually getting there. Check with a scope to make sure the power supplies do not have excessive noise or ripple. One microcomputer that we were called on to troubleshoot had very strange symptoms caused by 2-V peak-to-peak ripple on the 5-V supply.

Do a Signal Roll Call

The next step is to make a quick check of some key signals around the CPU of the microcomputer. If the problem is a bad IC, this can help point you toward the one that is bad. First, check if the clock signal is present and at the right frequency. If not, perhaps the clock generator IC is bad. If the microcomputer has a clock but doesn't seem to be doing anything, use an oscilloscope to check if the CPU is putting out control signals such as RD, WR, and ALE. Also, check the least significant data bus line to see if there is any activity on the buses. If there is no activity on these lines, a common cause is that the CPU is stuck in a wait, hold, halt, or reset condition by the failure of some TTL devices. To check this out, use the manual to help you predict what logic level should be on each of the CPU input control signals for normal operation. The RDY input of the 8086, for example, should be high for



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

5. It decrements the stack pointer again by 2 and pushes the current instruction pointer contents on the stack.
6. It does an indirect far jump to the start of the procedure you wrote to respond to the interrupt.

Figure 8.1 summarizes these steps in diagram form. As you can see, the 8086 pushes the flag register on the stack, disables the INTR input and the single-step function, and does essentially an indirect far call to the interrupt-service procedure. An IRET instruction at the end of the interrupt-service procedure returns execution to the main program. Now let's see how the 8086 actually gets to the interrupt procedure.

Remember from Chapter 5 that when the 8086 does a far call to a procedure, it puts a new value in the code segment register and a new value in the instruction pointer. For an indirect far call, the 8086 gets the new values for CS and IP from four memory addresses. Likewise, when the 8086 responds to an interrupt, it goes to four memory locations to get the CS and IP values for the start of the interrupt-service procedure. In an 8086 system, the first 1 Kbyte of memory, from 00000H to 003FFH, is set aside as a table for storing the starting addresses of interrupt service procedures. Since 4 bytes are required to store the CS and IP values for each interrupt-service procedure, the table can hold the starting addresses for up to 256 interrupt procedures. The starting address of an interrupt-service procedure is often called the *interrupt vector* or the *interrupt pointer*, so the table is referred to as the *interrupt-vector table* or the *interrupt-pointer table*.

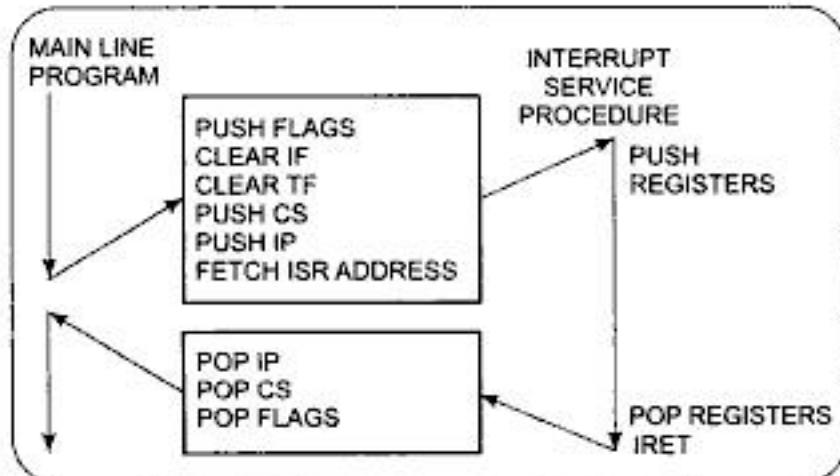


Fig. 8.1 8086 interrupt response.

Figure 8.2 shows how the 256 interrupt vectors are arranged in the table in memory. Note that the instruction pointer value is put in as the low word of the vector, and the code segment register is put in as the high word of the vector. Each doubleword interrupt vector is

identified by a number from 0 to 255. Intel calls this number the *type* of the interrupt.

The lowest five types are dedicated to specific interrupts, such as the divide-by-zero interrupt, the single-step interrupt, and the nonmaskable interrupt. Later in this chapter we explain the operation of these interrupts in detail. Interrupt types 5 to 31 are reserved by Intel

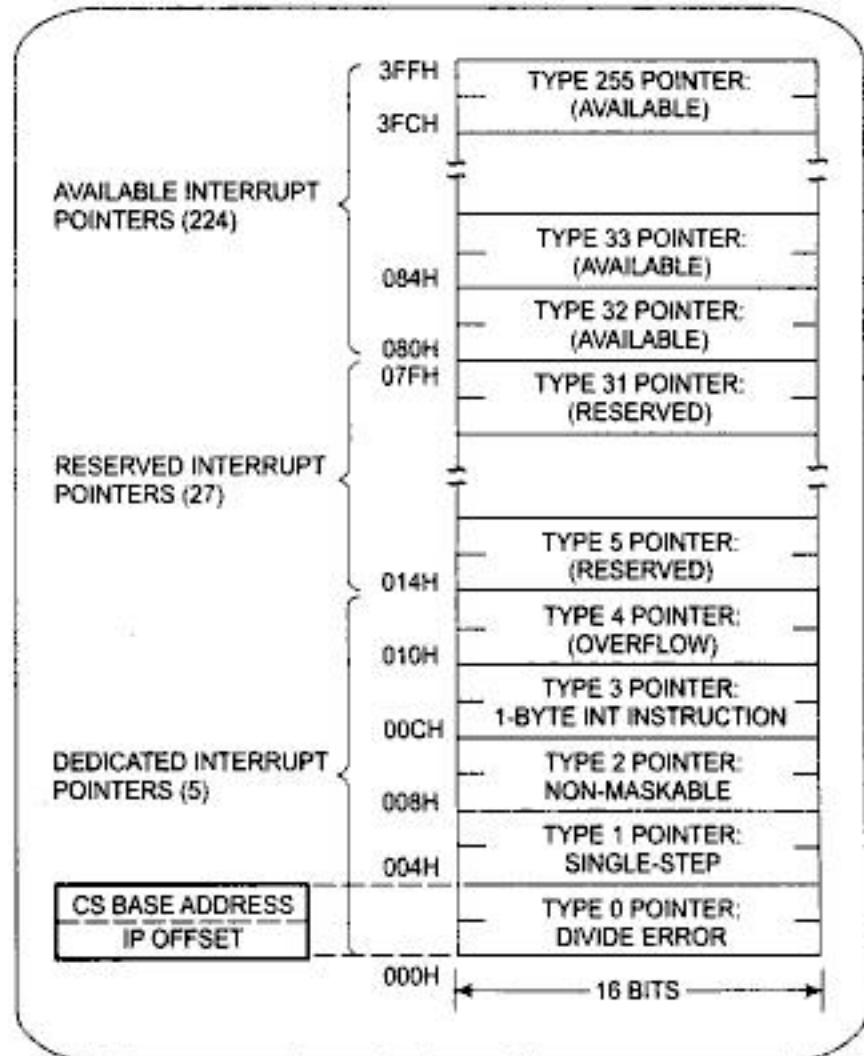


Fig. 8.2 8086 interrupt-pointer table.

for use in more complex microprocessors, such as the 80286, 80386, and 80486. In a later chapter we discuss some of these interrupt types. The upper 224 interrupt types, from 32 to 255, are available for you to use for hardware or software interrupts.

As you can see in Fig. 8.2, the vector for each interrupt type requires four memory locations. Therefore, when the 8086 responds to a particular type interrupt, it automatically multiplies the type by 4 to produce the desired address in the vector table. It then goes to that address in the table to get the starting address of the interrupt-service procedure. We will show you later how you use instructions at the start of your program to load the starting address of a procedure into the vector table.

Now that you have an overview of how the 8086 responds to interrupts, we will discuss one type of interrupt



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

1 ;8086 PROCEDURE F8-04B.ASM called by the program F8-04A.ASM
2 ;ABSTRACT: PROCEDURE BAD_DIV
3 ; Services divide-by-zero interrupt (TYPE 0).
4 ; Sets the LSB of a memory location called BAD_DIV_FLAG,
5 ; enables INTR, and returns execution to the interrupted program
6 ;DESTROYS: Nothing
7
8 0000
9
10 0000
11
12 PUBLIC BAD_DIV ; Make procedure BAD_DIV available
13 ; to other assembly modules
14 0000
15 0000
16 INT_PROC SEGMENT WORD PUBLIC ; Segment for interrupt-service procedure
17 0000 50 ; Assume CS:INT_PROC, DS:DATA
18 0001 1E ; Save AX of interrupted program
19 0002 B8 0000s ; Save DS of interrupted program
20 0005 8E D8 ; Load Data Segment register value
21 0007 C6 06 0000e 01 ; needed here
22 000C 1F ; Set LSB of BAD_DIV_FLAG byte
23 0000 58 ; Restore DS of interrupted program
24 000E CF ; Restore AX of interrupted program
25 000F ; Return to next instruction in interrupted
26 000F ; program
27 END

```

(b)

Fig. 8.4 (continued) (b) Interrupt-service procedure.

makes a *symbol table* which contains the segment and offset of each of the names and labels used in the program. The statement PUBLIC_BAD_DIV_FLAG tells the assembler to identify the name BAD_DIV_FLAG as public. This means that when the object module for this program is linked with some other object module that declares BAD_DIV_FLAG as EXTRN, the linker will be allowed to make the connection. Some programmers say that the PUBLIC directive "exports" a name or label.

The other end of this export operation is to "import" labels or names that are defined in other assembly modules. For example, the statement EXTRN BAD_DIV:FAR in our example program tells the assembler that BAD_DIV is a label of type far and that BAD_DIV is defined in some other assembly module. The INT_PROC SEGMENT WORD PUBLIC and INT_PROC ENDS statements tell the assembler that BAD_DIV is defined in a segment named INT_PROC. When the assembler reads these statements, it will make an entry in its symbol table for BAD_DIV and identify it as external. When the object module for this program is linked with the object module for the program where BAD_DIV is defined, the linker will fill in the proper values for the CS and IP of BAD_DIV.

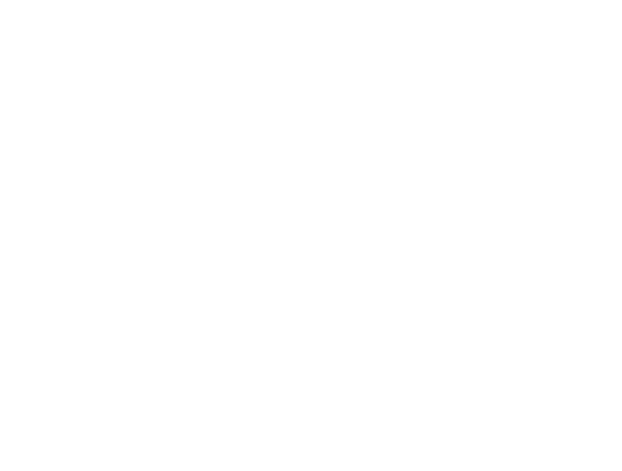
For the actual instructions of our mainline program, we declare a code segment with the statement CODE SEGMENT WORD PUBLIC.

As usual, at the start of the code segment we use an ASSUME statement to tell the assembler what logical segments to use for code, data, and stack. After this come the familiar instructions for initializing the stack segment register, the stack pointer register, and the data segment register.

The next four instructions load the address of the BAD_DIV interrupt-service procedure in the type 0 locations of the interrupt-vector table. We load ES with 0000 so that we can use it as an imaginary segment at absolute address 00000H. Then we use the statement MOV WORD PTR ES:0000 OFFSET BAD_DIV to load the offset of the interrupt-service procedure in memory at 00000H and 00001H. The statement MOV WORD PTR ES:0000 SEG BAD_DIV is used to load the segment base address of BAD_DIV into memory at 00002H and 00003H. It is necessary to load the interrupt procedure addresses in this way if you are using an SDK-86 board or using the MASM and Link programs on an IBM PC-type machine.

Next, we initialize SI as a pointer to the first input value and initialize BX as a pointer to the first of the locations we set aside for the 8-bit scaled results. CX is initialized as a counter to keep track of how many values have been scaled.

Finally, after everything is initialized, we get to the operations we set out to do. The statement MOV AX,[SI]



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

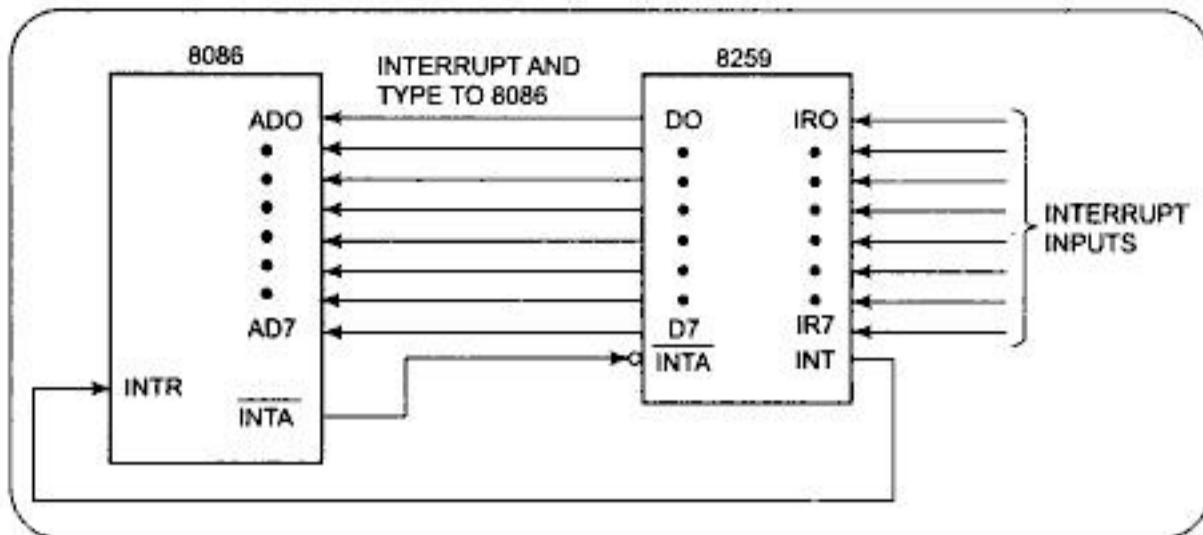


Fig. 8.5 Block diagram showing an 8259 connected to an 8086.

example, set a “flag” in a memory location as we did in the BAD_DIV procedure in Fig. 8.4b. The advantage of using the INTO and type 4 interrupt approach is that the error routine is easily accessible from any program.

SOFTWARE INTERRUPTS—TYPES 0 THROUGH 255

The 8086 INT instruction can be used to cause the 8086 to do any one of the 256 possible interrupt types. The desired interrupt type is specified as part of the instruction. The instruction INT 32, for example, will cause the 8086 to do a *type 32* interrupt response. The 8086 will push the flag register on the stack, reset TF and IF, and push the CS and IP values of the next instruction on the stack. It will then get the CS and IP values for the start of the interrupt-service procedure from the interrupt-pointer table in memory. The IP value for any interrupt type is always at an address of 4 times the interrupt type, and the CS value is at a location two addresses higher. For a type 32 interrupt, then, the IP value will be put at 4×32 or 128 decimal (80H), and the CS value will be put at address 82H in the interrupt-vector table.

Software interrupts produced by the INT instruction have many uses. In a previous section we discussed the use of the INT 3 instruction to insert breakpoints in programs for debugging. Another use of software interrupts is to test various interrupt-service procedures. You could, for example, use an INT 0 instruction to send execution to a divide-by-zero interrupt-service procedure without having to run the actual division program. As another example, you could use an INT 2 instruction to send execution to an NMI interrupt-service procedure. This allows you to test the NMI procedure without needing to apply an external signal to the NMI input of the 8086. In a later section of the chapter, we show an example of another important application of software interrupts.

INTR INTERRUPTS—TYPES 0 THROUGH 255

The 8086 INTR input allows some external signal to interrupt execution of a program. Unlike the NMI input, however, INTR can be masked (disabled) so that it cannot cause an interrupt. If the interrupt flag (IF) is cleared, then the INTR input is disabled. IF can be cleared at any time with the *Clear Interrupt* instruction, CLI. If the interrupt flag is set, the INTR input will be enabled. IF can be set at any time with the *Set Interrupt* instruction, STI.

When the 8086 is reset, the interrupt flag is automatically cleared. Before the 8086 can respond to an interrupt signal on its INTR input, you have to set IF with an STI instruction. The 8086 was designed this way so that ports, timers, registers, etc., can be initialized before the INTR input is enabled. In other words, this allows you to get the 8086 ready to handle interrupts before letting an interrupt in, just as you might want to get yourself ready in the morning with a cup of coffee before turning on the telephone and having to cope with the interruptions it produces.

Remember that the interrupt flag (IF) is also automatically cleared as part of the response of an 8086 to an interrupt. This is done for two reasons. First, it prevents a signal on the INTR input from interrupting a higher-priority interrupt-service procedure in progress. However, if you want another INTR input signal to be able to interrupt an interrupt procedure in progress, you can reenable the INTR input with an STI instruction at any time.

The second reason for automatically disabling the INTR input at the start of an INTR interrupt-service procedure is to make sure that a signal on the INTR input does not cause the 8086 to interrupt itself continuously. The INTR input is activated by a high level. In other words, whenever the INTR input is high and INTR is enabled, the 8086 will



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

1 ;8086 PROGRAM F8-09A.ASM
2 ;ABSTRACT : Mainline of program to read characters from a keyboard
3 ; The mainline of this program initializes the interrupt
4 ; table with the address of the procedure that reads
5 ; characters from a keyboard on an interrupt basis.
6 ;PORTS : Uses none in mainline. Uses FFF8H in procedure
7 ;REGISTERS : Uses CS,DS,SS,ES,SP,AX
8 ;PROCEDURES: Uses KEYBOARD
9 ; : Link mainline F8-09A.OBJ with procedure F8-09B.OBJ
10
11 0000
12 0000 64*(00)
13 0064 0000r
14 0066 64
15 0067 00
16 0068
17
18 0000
19 0000 64*(0000)
20
21 00C8
22
23 PUBLIC ASCII_POINTER, CHRCNT, KEYDONE ; Make available to other modules
24 EXTRN KEYBOARD:FAR ; Procedure in another assembly module
25
26 0000
27
28 0000 B8 0000s
29 0003 8E D0
30 0005 BC 00C8r
31 0008 38 0000s
32 0008 BE D8
33 ;Store the address for the KEYBOARD routine at address 0000:0008
34 ;Address 00008-00008 is where type 2 interrupt gets interrupt
35 ;service procedure address. CS at 0000A & 00008, IP at 00008 & 00009
36 0000 B8 0000
37 0010 8E C0
38 0012 26: C7 06 000A 0000s
39 0019 26: C7 06 0008 0000e
40 ;Simulate larger program
41 0020 EB FE
42 0022
43 CODE ENDS
END

```

(a)

Fig. 8.9 Reading characters from an ASCII keyboard on interrupt basis. (a) Initialization and mainline. (See also next page.)

other tasks between keyboard interrupts. With polling, the 8086 would not easily be able to do this.

Using Interrupts for Counting and Timing

COUNTING APPLICATIONS

As a simple example of the use of an interrupt input for counting, suppose that we are using an 8086 to control a printed-circuit-board-making machine in our computerized electronics factory. Further suppose that we want to detect each finished board as it comes out of the machine and to keep a count of finished boards so that we can compare this count with the number of boards fed in. This way we can determine if any boards were lost in the machine.

To do this count on an interrupt basis, all we have to do is detect when a board passes out of the machine and send an interrupt signal to an interrupt input on the 8086. The interrupt-service procedure for that input can simply increment the board count stored in a named memory location.

To detect a board coming out of the machine, we use an infrared LED, a phototransistor, and two conditioning gates, as shown in Fig. 8.10. The LED is positioned over the track where the boards come out, and the phototransistor is positioned below the track. When no board is between the LED and the phototransistor, the light from the LED will strike the phototransistor and turn it on. The collector of the phototransistor will then be low, as will the NMI input on the 8086. When a board passes



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

| A8-A15 | A5-A7 | A4 | A3 | A2 | A1 | A0 | MIO | Y OUTPUT SELECTED | SYSTEM BASE ADDRESS | DEVICE |
|------------------|-------|----|----|----|----|----|-----|----------------------|------------------------|----------|
| 1 | 0 | 0 | 0 | X | X | 0 | 0 | 0 | FF00 | 8259A#1 |
| 1 | 0 | 0 | 1 | X | X | 0 | 0 | 1 | FF08 | 8259A #2 |
| 1 | 0 | 1 | 0 | X | X | 0 | 0 | 2 | FF10 | |
| 1 | 0 | 1 | 1 | X | X | 0 | 0 | 3 | FF18 | |
| 1 | 0 | 0 | 0 | X | X | 1 | 0 | 4 | FF01 | 8254 |
| 1 | 0 | 0 | 1 | X | X | 1 | 0 | 5 | FF09 | |
| 1 | 0 | 1 | 0 | X | X | 1 | 0 | 6 | FF11 | |
| 1 | 0 | 1 | 1 | X | X | 1 | 0 | 7 | FF19 | |
| ALL OTHER STATES | | | | | | | | NONE | | |

Fig. 8.15 Truth table for 74LS138 address decoder in Figure 8.14.

states. Before you can use them for anything, you have to initialize them in the *mode* you need for your specific application. Initializing these devices is not usually difficult, but it is very easy to make errors if you do not do it in a very systematic way. To initialize any programmable peripheral device, you should always work your way through the following series of steps.

- Determine the system base address for the device. You do this from the address decoder circuitry or the address decoder truth table. From the truth table in Fig. 8.15, the system base address of the 8254 in our example here is FF01H.

| A1 | A0 | SELECTS |
|----|----|-----------------------|
| 0 | 0 | COUNTER0 |
| 0 | 1 | COUNTER1 |
| 1 | 0 | COUNTER2 |
| 1 | 1 | CONTROL WORD REGISTER |

(a)

| SYSTEM ADDRESS | 8254 PART |
|----------------|-------------|
| F F 0 1 | COUNTER0 |
| F F 0 3 | COUNTER1 |
| F F 0 5 | COUNTER2 |
| F F 0 7 | COUNTER REG |

(b)

Fig. 8.16 8254 addresses. (a) Internal, (b) System.

- Use the device data sheet to determine the internal addresses for each of the control registers, ports, timers, status registers, etc., in the device. Figure 8.16a shows the internal addresses for the three counters and the control word register for the

8254. A0 in this table represents the A0 input of the device, and A1 represents the A1 input of the device. Note in the schematic in Fig. 8.14 that system address line A1 is connected to the A0 input of the 8254, and system address line A2 is connected to the A1 input. We could not use system address line A0 as one of these because, as described before, we used system address line A0 as one of the inputs to the address decoder.

- Add each of the internal addresses to the system base address to determine the system address of each of the parts of the device. You need to do this so that you know the actual addresses where you have to send control words, timer values, etc. Figure 8.16b shows the system addresses for the three timers and the control register of the 8254 we added to the SDK-86 board. Note that the addresses all have to be odd because the device is connected on the upper half of the data bus.
- Look in the data sheet for the device for the format of the control word(s) that you have to send to the device to initialize it. For different devices, incidentally, the control word(s) may be referred to as command words or mode words. To initialize the 8254, you send a control word to the control register for each counter that you want to use. Figure 8.17 shows the format for the 8254 control word.
- Construct the control word required to initialize the device for your specific application. You construct this control word on a bit-by-bit basis. We have found it helpful to actually draw the eight little boxes shown at the top of Fig. 8.17 so that we don't miss any bits. (An easy way to draw the eight boxes is to draw a long rectangle, divide it in half, divide



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

loaded into the counter on the next clock pulse. Following clock pulses will decrement the new count until it reaches 0.

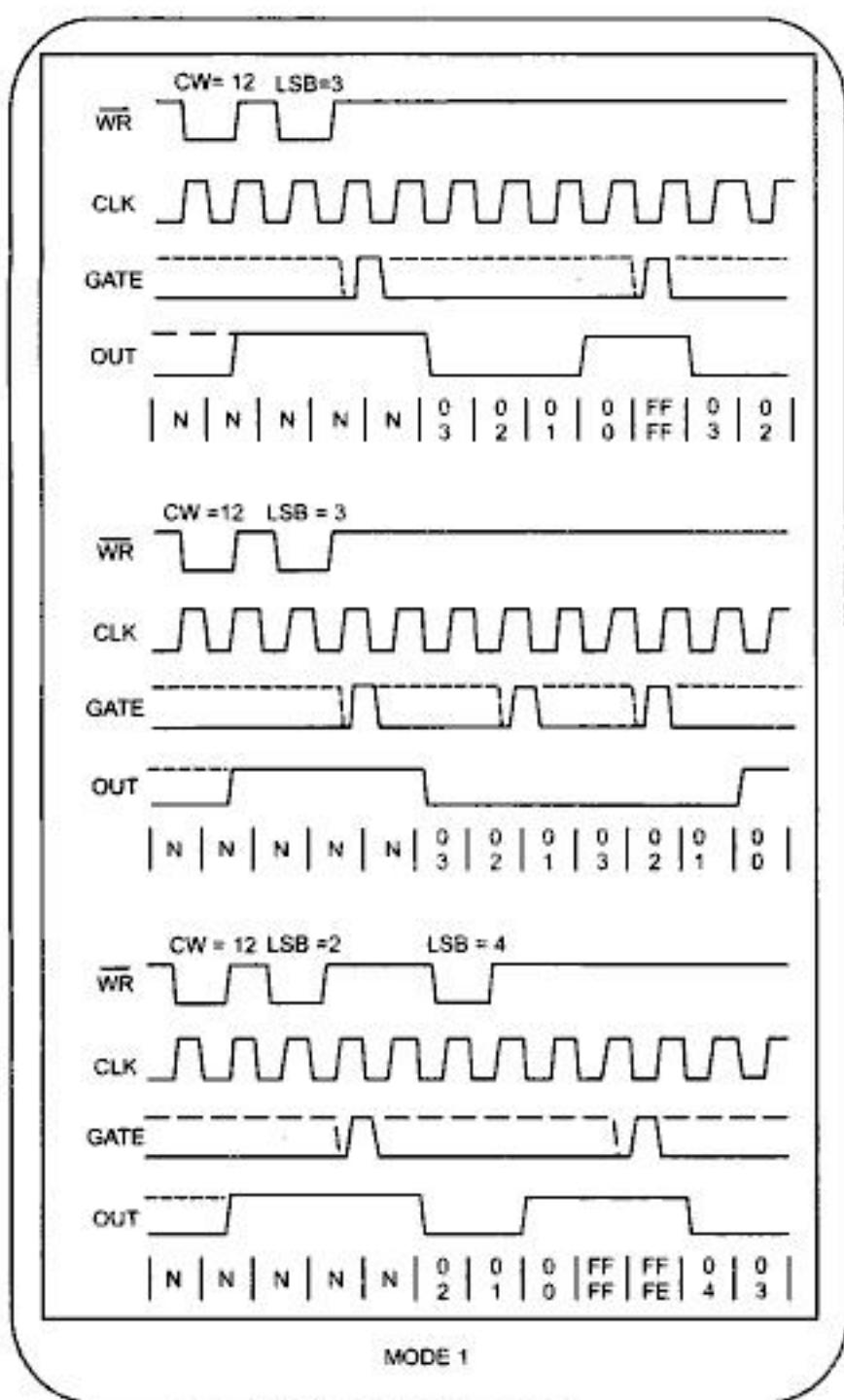


Fig. 8.19 8254 MODE 1 example timing diagrams.
(Intel Corporation)

As an example of what you can use this mode for, suppose that as one of its jobs you want to use an 8086 to control some parking lot signs around an electronics factory. The main parking lot can hold 1000 cars. When it gets full, you want to turn on a sign which directs people to another lot. To detect when a car enters the lot, you can use an optical sensor such as the one shown in Fig. 8.10. Each time a car passes through, this circuit will produce a pulse. You could connect the signal from this sensor directly to an interrupt input and have the processor count interrupts, as we did for the printed-circuit-board-making machine in a previous example. However, the less you burden the processor with trivial tasks such as this,

the more time it has available to do complex work for you. Therefore, you let a counter in an 8254 count cars and interrupt the 8086 only when it has counted 1000 cars.

You connect the output from the optical sensor circuit to the CLK input of, say, counter 1 of an 8254. You tie the GATE input of counter 1 to + 5 V so it will be enabled for counting and connect the OUT pin of counter 1 to an interrupt input on an 8259A or the NMI input on the 8086.

In the mainline program, you initialize counter 1 for mode 0, BCD counting, and read/write LSB then MSB with a control word of 01110001 binary. You want the counter to produce an interrupt after 1000 pulses from the sensor, so you will send a count of 999 decimal to the counter. The reason that you want to send 999 instead of 1000 is that, as shown in Fig. 8.18, the OUT pin will go high $N + 1$ clock pulses after the count value is written to the counter. Since you initialized the counter for read/write LSB then MSB, you send 99H and then 09H to the address of counter 1. By initializing the counter for BCD counting, you can just send the count value as a BCD number instead of having to convert it to hex.

The service procedure for this interrupt will contain instructions which turn on the parking-lot-full sign, close off the main entrance, and return to the mainline program. For this example you don't have to worry that the counter decrements from 0000 to FFFFH, because after you shut the gate, the counter will not receive any more interrupts.

MODE 1—HARDWARE-RETRIGGERABLE ONE-SHOT

The basic principle of a *one-shot* is that when a signal is applied to the trigger input of the device, its output will be asserted. After a fixed amount of time the output will automatically return to its unasserted state. For a TTL one-shot such as the 74LS122, the time that the output is asserted is determined by the time constant of a resistor and a capacitor connected to the device. For an 8254 counter in one-shot mode, the time that the output is asserted low is determined by the frequency of an applied clock and by a count loaded into the counter. The advantage of the 8254 approach is that the output pulse width can be changed under program control and, if a crystal-controlled clock is used, the output pulse width can be very accurately specified.

Figure 8.19 shows some example timing waveforms for an 8254 counter in mode 1. Let's take a look at the top set of waveforms. Again, the first dip in the WR waveform represents the control word of 12H being sent to the 8254. Use Fig. 8.17 to help you determine how this control word initializes the device. You should find that a



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

As an example of this application, suppose that you want to produce a tone that is a musical A of 440 Hz from the 2.4576-MHz PCLK signal. Dividing the PCLK signal by 5585 will give the desired 440 Hz. Therefore, you simply send a control word which programs a counter for mode 3, read/write LSB then MSB, and BCD counting. You then write the LSB of 85H and the MSB of 55H to the counter. If you want to change the frequency, all you have to do is write a new count to the count register. With a few programmable counters and some relatively simple programming, you can play your favorite songs.

MODE 4—SOFTWARE-TRIGGERED STROBE

This mode and mode 5 are often confused with mode 1, but there is an obvious difference. Mode 1 is used to produce a low-going pulse that is N clock pulses wide. If you look at the top set of waveforms for mode 4 in Fig. 8.24, you should see that mode 4 produces a low-going pulse *after* $N + 1$ clock pulses. For mode 4, the output pulse is low for the time of one input clock pulse and then returns high. In other words, in mode 4, a counter produces a low-going strobe pulse $N + 1$ clock cycles after a count is written to the count register. Mode 4 is referred to as *software-triggered* because it is the writing of the count to the count register that starts the process. Note that after the loaded count is counted down, the counter decrements to FFFFH and then continues to decrement from there.

Mode 4 can be used in a case where you want to send out some parallel data on a port and then after some delay send out a strobe signal to let the receiving system know that the data is available.

MODE 5—HARDWARE-TRIGGERED STROBE

Mode 5 is used where we want to produce a low-going strobe pulse some programmable time interval after a rising-edge trigger signal is applied to the GATE input. This mode is very useful when you want to delay a rising-edge signal by some amount of time.

Figure 8.25 shows some example waveforms for a counter operating in mode 5. For a start, let's look at the top set of waveforms. As usual, we write a control word and the desired count to a counter. As shown by the count sequence under the OUT waveform, however, the count does not get transferred to the counter until the GATE (trigger) is made high. When the trigger input is made high, the count will be transferred to the counter on the next clock pulse. Succeeding clock pulses will decrement the counter. When the counter reaches 0, the OUT pin will go low for one clock pulse time. The OUT pin will go

low $N + 1$ clock pulses after the trigger input goes high.

The second set of waveforms in Fig. 8.25 shows that if another trigger pulse occurs during the countdown time, the original count will be reloaded on the next clock pulse and the countdown will start over. The OUT pin will remain high until the count is finally countdown. If trigger pulses continue to come before the countdown is completed, the OUT pin will continue to stay high. Therefore, you can use a counter in mode 5 to produce a power fail signal, as we showed in the previous discussion of mode 1. Note that for mode 5, however, the OUT pin will be high if the power is on and go low when the power fails.

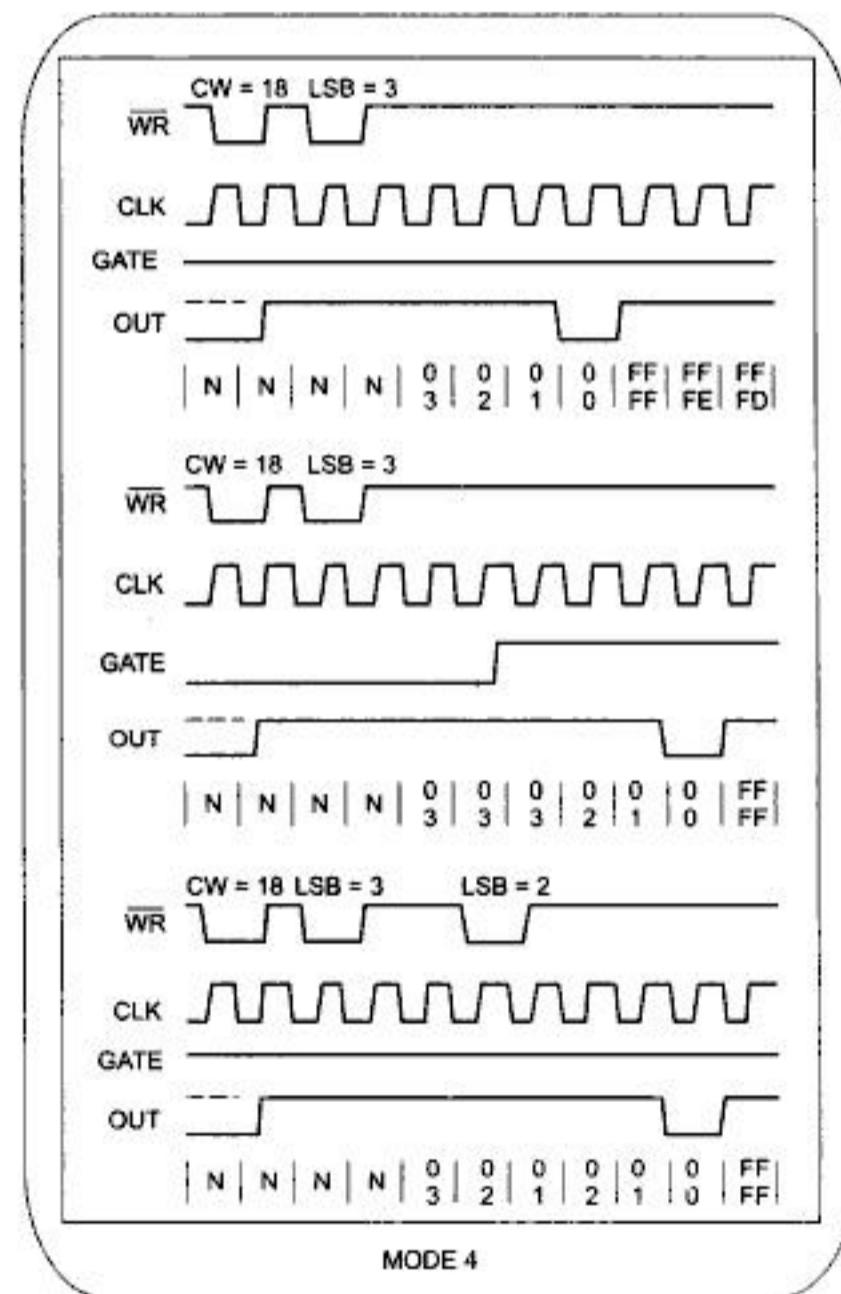


Fig. 8.24 8254 MODE 4 example timing waveforms. (Intel Corporation)

The bottom set of waveforms in Fig. 8.25 shows that if a new count is written to a counter, the new count will not be loaded into the counter until a new trigger pulse occurs.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

NOTE: An interrupt signal must remain high on an IR input until after the falling edge of the first INTA pulse.

The in-service register keeps track of which interrupt inputs are currently being serviced. For each input that is currently being serviced, the corresponding bit will be set in the in-service register.

The priority resolver acts as a "judge" that determines if and when an interrupt request on one of the IR inputs gets serviced.

As an example of how this works, suppose that IR2 and IR4 are unmasked and that an interrupt signal comes in on the IR4 input. The interrupt request on the IR4 input will set bit 4 in the interrupt request register. The priority resolver will detect that this bit is set and check the bits in the in-service register (ISR) to see if a higher-priority input is being serviced. If a higher-priority input is being serviced, as indicated by a bit being set for that input in the ISR, then the priority resolver will take no action. If no higher-priority interrupt is being serviced, then the priority resolver will activate the circuitry which sends an interrupt signal to the 8086. When the 8086 responds with INTA pulses, the 8259A will send the interrupt type that was specified for the IR4 input when the 8259A was initialized. As we said before, the 8086 will use the type number it receives from the 8259A to find and execute the interrupt-service procedure written for the IR4 interrupt.

Now, suppose that while the 8086 is executing the IR4 service procedure, an interrupt signal arrives at the IR2 input of the 8259A. This will set bit 2 of the interrupt request register. Since we assumed for this example that IR2 was unmasked, the priority resolver will detect that this bit in the IRR is set and make a decision whether to send another interrupt signal to the 8086. To make the decision, the priority resolver looks at the in-service register. If a higher-priority bit in the ISR is set, then a higher-priority interrupt is being serviced. The priority resolver will wait until the higher-priority bit in the ISR is reset before sending an interrupt signal to the 8086 for the new interrupt input. If the priority resolver finds that the new interrupt has a higher priority than the highest-priority interrupt currently being serviced, it will set the appropriate bit in the ISR and activate the circuitry which sends a new INT signal to the 8086. For our example here, IR2 has a higher priority than IR4, so the priority resolver will set bit 2 of the ISR and activate the circuitry which sends a new INT signal to the 8086. If the 8086 INTR input was reenabled with an STI instruction at the start of the IR4 service procedure, as shown in Fig. 8.28a, then this new INT signal will interrupt the 8086 again. When the 8086 sends out a second INTA pulse in response to

this interrupt, the 8259A will send it the type number for the IR2 service procedure. The 8086 will use the received type number to find and execute the IR2 service procedure.

At the end of the IR2 procedure, we send the 8259A a command word that resets bit 2 of the in-service register so that lower-priority interrupts can be serviced. After that, an IRET instruction at the end of the IR2 procedure sends execution back to the interrupted IR4 procedure. At the end of the IR4 procedure, we send the 8259A a command word which resets bit 4 of the inservice register so that lower-priority interrupts can be serviced. An IRET instruction at the end of the IR4 procedure returns execution to the mainline program. This all sounds very messy, but it is really just a special case of nested procedures. Incidentally, if the IR4 procedure did not reenable the 8086 INTR input with an STI instruction, the 8086 would not respond to the IR2-caused INT signal until it finished executing the IR4 procedure, as shown in Fig. 8.28b.

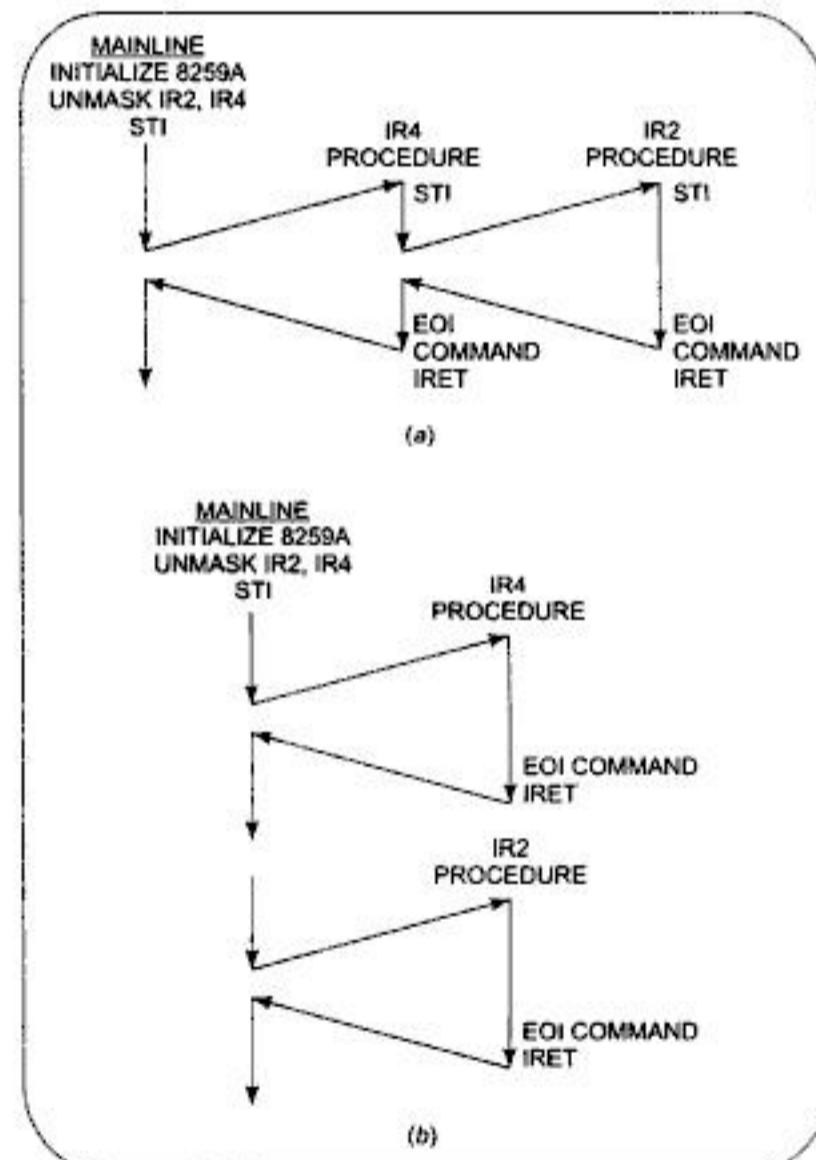


Fig. 8.28 8259A and 8086 program flow for IR4 interrupt followed by IR2 interrupt.
(a) Response with INTR enabled in IR4 procedure
(b) Response with INTR not enabled in IR4 procedure.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

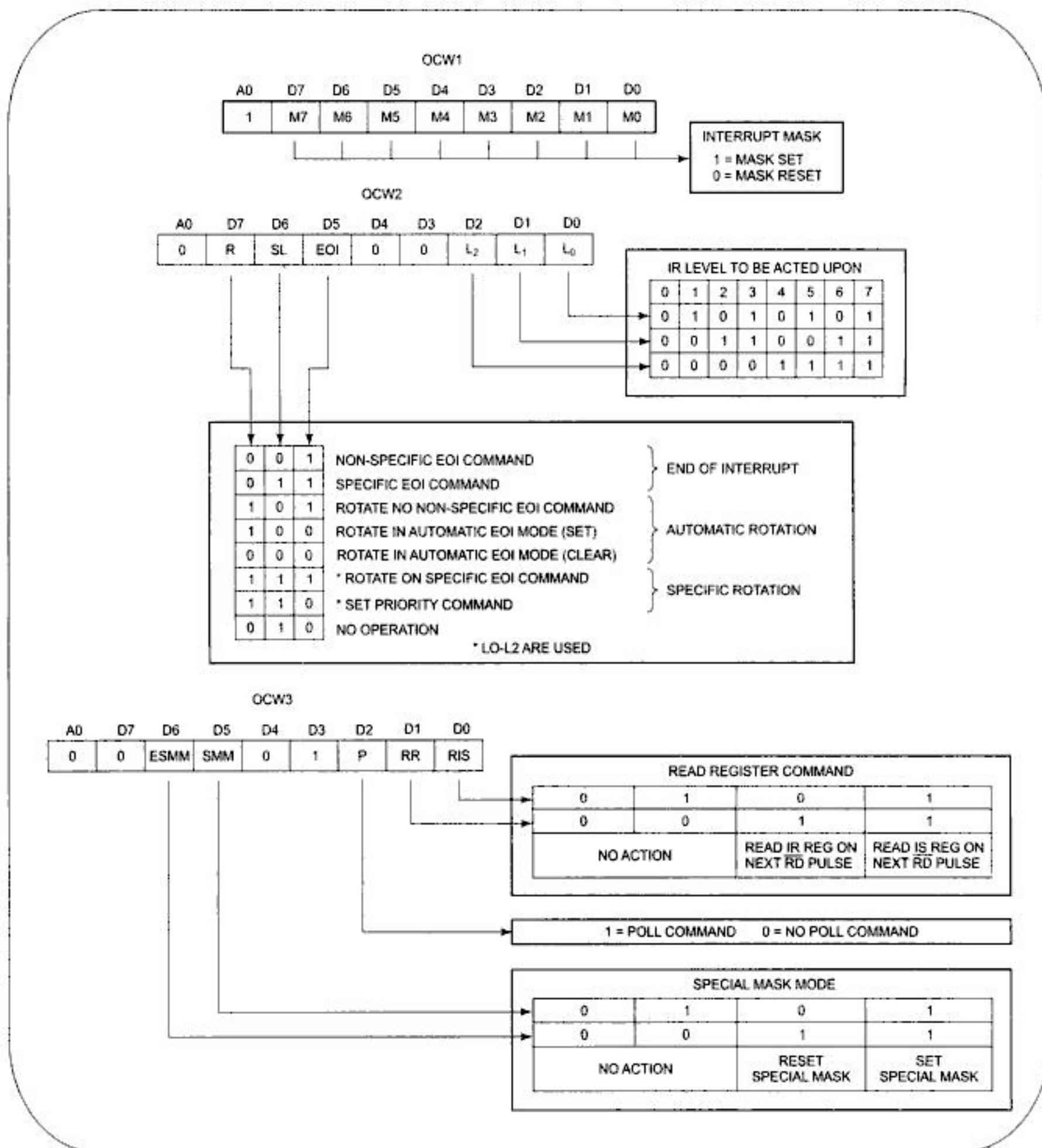


Fig. 8.30 8259A operational command words. (Intel Corporation)

words called *operation command words*, or OCWs. These are shown in Fig. 8.30. An OCW1 must be sent to an 8259A to unmask any IR inputs that you want it to respond to. For our example here, let's assume that we want to use only IR2 and IR3. Since a 0 in a bit position of OCW1

unmasks the corresponding IR input, we put 0's in these two bits and 1's in the rest of the bits. Our OCW1 is 111110011. OCW2 is mainly used to reset a bit in the in-service register. This is usually done at the end of the interrupt-service procedure, but it can be done at any time



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

you have to know is the interrupt type for the procedure and how to pass parameters to the procedure. This means that a program you write for an IBM computer will work on a compatible COMPAQ computer, even though the BIOS printer driver procedures are located at very different absolute addresses in the two machines. In later chapters we show more examples of using BIOS procedures.

This chapter has introduced you to interrupts and some interrupt applications. The following chapters will show you many more applications of interrupts because almost every microcomputer system uses a variety of interrupts.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms in the following list, use the index to help you find them in the chapter for review.

8086 interrupt response

Interrupt-service procedure

Interrupt vector, interrupt pointer

Interrupt-vector table, interrupt-pointer table

Interrupt type

Divide-by-zero interrupt—type 0

Single-step interrupt—type 1

```

1 ;8086 PROGRAM F8-33.ASM
2 ;ABSTRACT : This program sends a string of characters to a printer
3 ; from an IBM type PC. To run this program, first assemble
4 ; and use the LINK program to create the EXE file. Then
5 ; turn on your printer, & at the DOS prompt type F8-33.
6 ;REGISTERS : Uses CS, SS, DS, BX, AX, CX, DX
7 ;PORTS : Uses printer port 0
8 ;PROCEDURES: Calls BIOS printer IO procedure INT 17
9
10 0000
11 0000 C8*(0000)
12
13 0190
14
15 = 001E
16
17 0000
18 0000 48 45 4C 4C 4F 20 54 +
19 48 45 52 45 2C 20 48 +
20 4F 57 20 41 52 45 20 +
21 59 4F 55 3F
22 0019 00 0A 00 0A
23 001D
24
25 0000
26
27 0000 B8 0000s
28 0003 8E D0
29 0005 BC 0190r
30 0008 B8 0000s
31 0008 8E D8
32 0000 B4 01
33 000F BA 0000
34 0012 CD 17
35 0014 8D 1E 0000r
36 0018 B9 001E
37 001B B4 00
38 001D 8A 07
39 001F CD 17
40 0021 80 FC 01
41 0024 75 04
42 0026 F9
43 0027 EB 05 90
44 002A F8
45 002B 43
46 002C E2 ED
47 002E B8 4C00
48 0031 CD 21
49 0033
50
      ; Set aside 200 words for stack
      DW 200 DUP(0) ; Assign name to word above stack top
      STACK_TOP LABEL WORD
      STACK_SEG ENDS
      CHAR_COUNT EQU 30
      DATA SEGMENT
      MESSAGE DB 'HELLO THERE, HOW ARE YOU?'
      MESSAGE_END DB 0DH, 0AH, 0DH, 0AH ; return & line feed
      DATA ENDS
      CODE SEGMENT
      ASSUME CS:CODE, SS:STACK_SEG, DS:DATA
      START: MOV AX, STACK_SEG ; Initialize stack segment register
              MOV SS, AX
              MOV SP, OFFSET STACK_TOP ; Initialize stack pointer register
              MOV AX, DATA ; Initialize data segment register
              MOV DS, AX
              MOV AH, 01 ; Set up registers to initialize printer
              MOV DX, 0 ; port 0
              INT 17H ; Call procedure to initialize printer port
              LEA BX, MESSAGE ; Get to start of message
              MOV CX, CHAR_COUNT ; Set up a count variable
              AGAIN: MOV AH, 0 ; Load code for procedure to send character
                     MOV AL, [BX] ; Load character to be sent into AL
                     INT 17H ; Use BIOS routine to send character to printer
                     CMP AH, 01H ; If character not printed then returns AH =1
                     JNE NEXT ; IF character not printed THEN
                     NOT_RDY: STC ; Set carry to indicate message not sent
                     JMP EXIT ; and leave loop
                     NEXT: CLC ; ELSE Clear carry flag (character sent)
                            INC BX ; Move to address of next character
                            LOOP AGAIN ; Send the next character
                            EXIT: MOV AX, 4C00H ; Graceful exit to DOS
                                  INT 21H ; with function call 4CH
      CODE ENDS
      END START

```

Fig. 8.33 8086 assembly language program for outputting characters to a printer.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Digital Interfacing

The major goal of this chapter and the next is to show you the circuitry and software needed to interface a basic microcomputer with a wide variety of digital and analog devices. In each topic we try to show enough detail so that you can build and experiment with these circuits. Perhaps you can use some of them to control appliances around your house or to solve some problems at work.

In this chapter, we concentrate on the devices and techniques used to get digital data into and out of the basic microcomputer. Then, in the next chapter, we concentrate on analog interfacing.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. *Describe simple input and output, strobed input and output, and handshake input and output.*
2. *Initialize a programmable parallel-port device such as the 8255A for simple input or output and for handshake input or output.*
3. *Interpret the timing waveforms for handshake input and output operations.*
4. *Describe how parallel data is sent to a printer on a handshake basis.*
5. *Show the hardware connections and the programs that can be used to interface keyboards to a microcomputer.*
6. *Show the hardware connections and the programs that can be used to interface alphanumeric displays to a microcomputer.*
7. *Describe how an 8279 can be used to refresh a multiplexed LED display and scan a matrix keyboard.*
8. *Initialize an 8279 for a given display and keyboard format.*
9. *Show the circuitry used to interface high-power devices to microcomputer ports.*
10. *Describe the hardware and software needed to control a stepper motor.*
11. *Describe how optical encoders are used to determine the position, direction of rotation, and speed of a motor shaft.*

PROGRAMMABLE PARALLEL PORTS AND HANDSHAKE INPUT/OUTPUT

Throughout the program examples in the preceding chapters, we have used port devices to input parallel data to the microprocessor and to output parallel data from the microprocessor. Most of the available port devices, such as the 8255A on the SDK-86 board, contain two or three ports which can be programmed to operate in one of several different modes. The different modes allow you to use the devices for many common types of parallel data transfer. First we will discuss some of these common methods of transferring parallel data, and then we will show how the 8255A is initialized and used in a variety of I/O operations.

Methods of Parallel Data Transfer

SIMPLE INPUT AND OUTPUT

When you need to get digital data from a simple switch, such as a thermostat, into a microprocessor, all you have to do is connect the switch to an input port line and read the port. The thermostat data is always present and ready, so you can read it at any time.

Likewise, when you need to output data to a simple display device such as an LED, all you have to do is connect the input of the LED buffer on an output port pin and output the logic level required to turn on the light. The



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

8255A Operational Modes and Initialization

Figure 9.4, summarizes the different modes in which the ports of the 8255A can be initialized.

MODE 0

When you want to use a port for simple input or output without handshaking, you initialize that port in mode 0. If both port A and port B are initialized in mode 0, then the two halves of port C can be used together as an additional 8-bit port, or they can be used individually as two 4-bit ports. When used as outputs, the port C lines can be individually set or reset by sending a special control word to the control register address. Later we will show you how to do this. The two halves of port C are independent, so one half can be initialized as input, and the other half initialized as output.

MODE 1

When you want to use port A or port B for a handshake (strobed) input or output operation such as we discussed in previous sections, you initialize that port in mode 1.

In this mode, some of the pins of port C function as handshake lines. Pins PC0, PC1, and PC2 function as handshake lines for port B if it is initialized in mode 1. If port A is initialized as a handshake (mode 1) input port, then pins PC3, PC4, and PC5 function as handshake signals. Pins PC6 and PC7 are available for use as input lines or output lines. If port A is initialized as a handshake output port, then port C pins PC3, PC6, and PC7 function as handshake signals. Port C pins PC4 and PC5 are available for use as input or output lines. Since the 8255A is often used in mode 1, we show several examples in the following sections.

MODE 2

Only port A can be initialized in mode 2. In mode 2, port A can be used for *bidirectional handshake* data transfer. This means that data can be output or input on the same eight lines. The 8255A might be used in this mode to extend the system bus to a slave microprocessor or to transfer data bytes to and from a floppy disk controller board. If port A is initialized in mode 2, then pins PC3 through PC7 are used as handshake lines for port A. The other three pins, PC0 through PC2, can be used for I/O if port B is in mode 0. The three pins will be used for port B handshake lines if port B is initialized in mode 1. After you work your way through the mode 1 examples in the following sections, you should have little difficulty

understanding the discussion of mode 2 in the Intel data sheet if you encounter it in a system.

Constructing and Sending 8255A Control Words

Figure 9.5 shows the formats for the two 8255A control words. Note that the MSB of the control word tells the 8255A which control word you are sending it. You use the *mode definition control word* format in Fig. 9.5a to tell the device what modes you want the ports to operate in. You use the *bit set/reset control word* format in Fig. 9.5b when you want to set or reset the output on a pin of port C or when you want to enable the interrupt

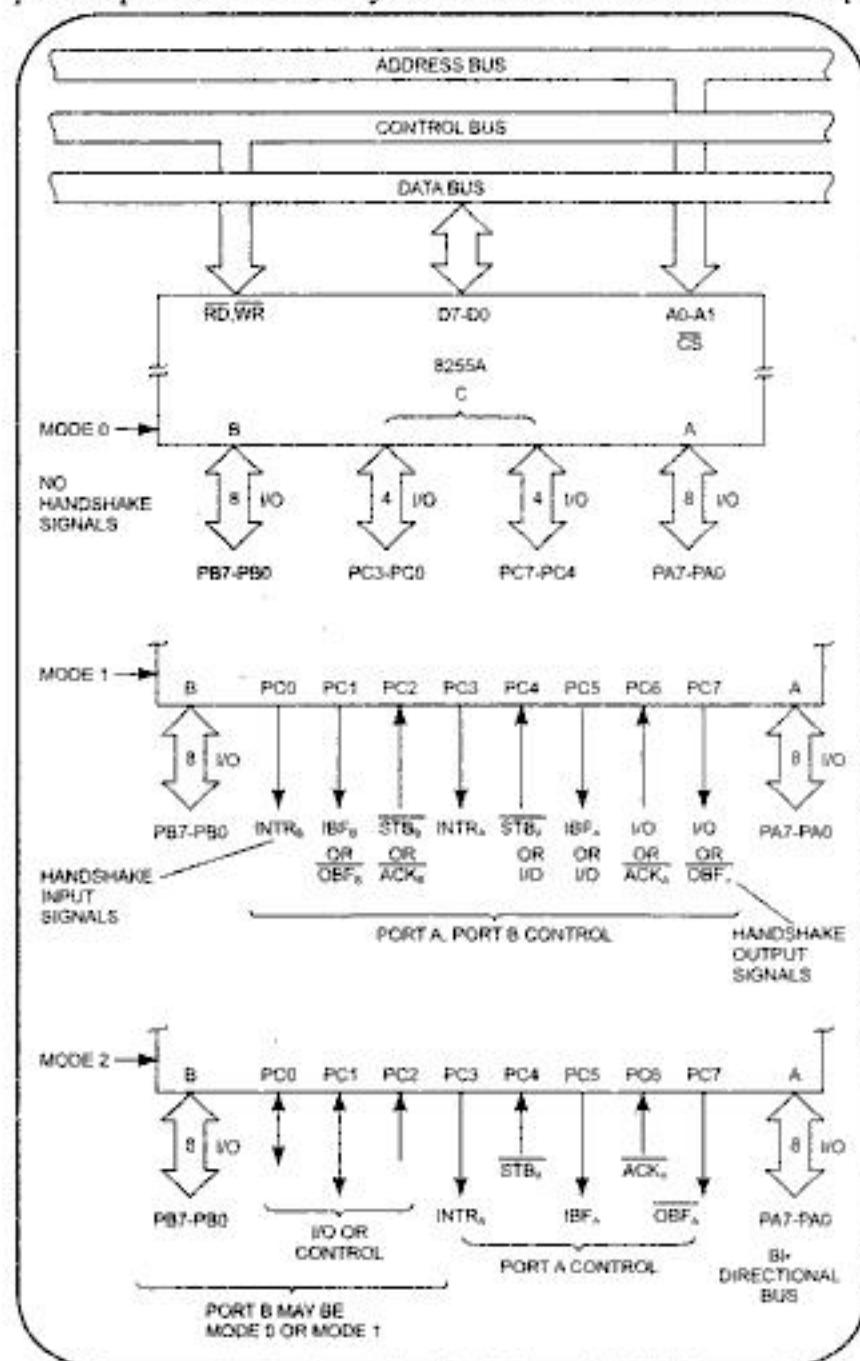


Fig. 9.4 Summary of 8255A operating modes.
(Intel Corporation)

output signals for handshake data transfers. Both control words are sent to the control register address of the 8255A.

As usual, initializing a device such as this consists of working your way through the steps we described in the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

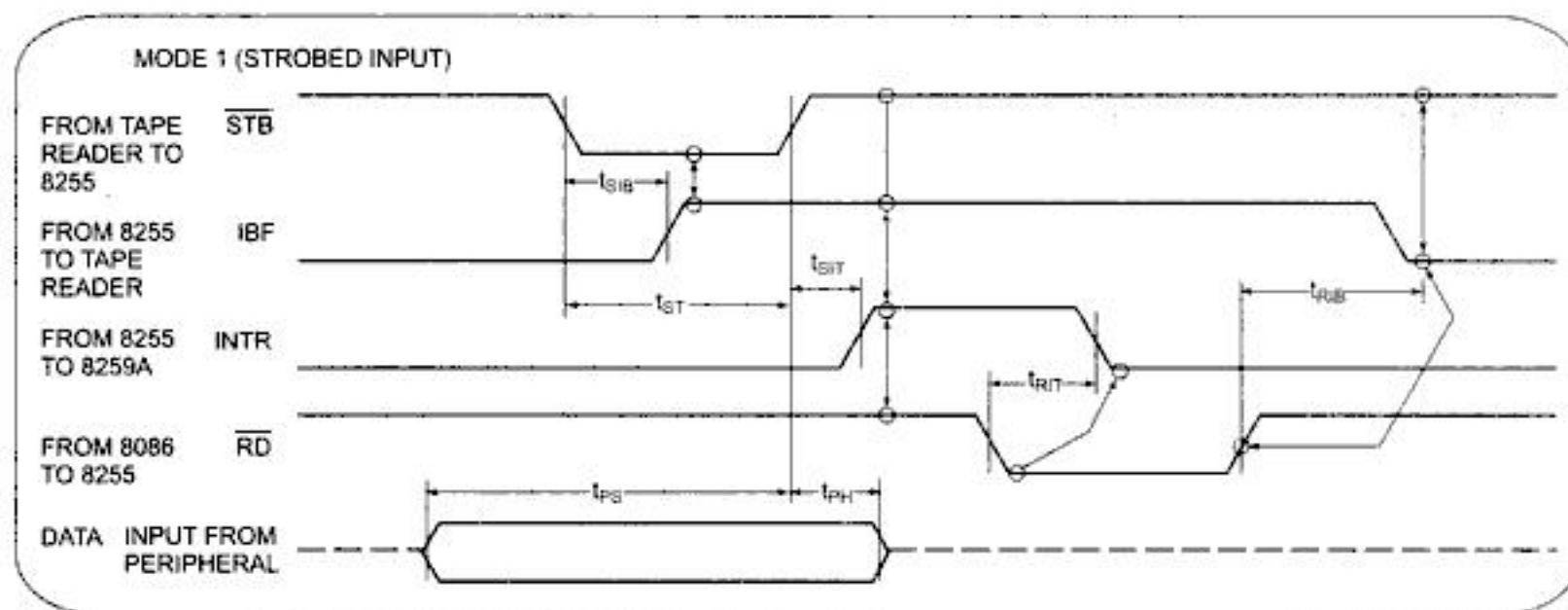


Fig. 9.10 Timing waveforms for 8255 handshake data input from a tape reader.

parts until it runs out of material. The unused bit of port C, PC7, could be connected to a mechanism which loads in more material so the lathe can continue.

The microcomputer-controlled lathe we have described here is a small example of automated manufacturing. The advantage of this approach is that it relieves humans of the drudgery of standing in front of a machine continually making the same part, day after day. We hope society can find more productive use for the human time made available.

PARALLEL PRINTER INTERFACE—HANDSHAKE OUTPUT EXAMPLE

At the end of Chapter 8, we showed you how to send a string of text characters to a printer by calling a BIOS procedure with a software interrupt. In this section we show you the hardware connections and software required to interface with a parallel printer in a system which does not have a BIOS procedure you can call to do the job.

For most common printers, such as the IBM PC printers, the Epson dot-matrix printers, and the Panasonic dot-matrix printers, data to be printed is sent to the printer as ASCII characters on eight parallel lines. The printer receives the characters to be printed and stores them in an internal RAM buffer. When the printer detects a carriage return character (ODH), it prints out the first row of characters from the print buffer. When the printer detects a second carriage return, it prints out the second row of characters, etc. The process continues until all the desired characters have been printed.

Transfer of the ASCII codes from a microcomputer to a printer must be done on a handshake basis because the microcomputer can send characters much faster than the printer can print them. The printer must in some way let the microcomputer know that its buffer is full and that

it cannot accept any more characters until it prints some out. A common standard for interfacing with parallel printers is the *Centronics Parallel Interface Standard*, named after the company that developed it. In the following sections, we show you how a Centronics parallel interface works and how to implement it with an 8255A.

Centronics Interface Pin Descriptions and Circuit Connections

Centronics-type printers usually have a 36-pin interface connector. Figure 9.11 shows the pin assignments and descriptions for this connector as it is used in the IBM PC printer and the Epson printers. Some manufacturers use one or two pins differently, so consult the manual for your specific printer before connecting it up as we show here.

Thirty-six pins may seem like a lot of pins just to send ASCII characters to a printer. The reason for the large number of lines is that each data and signal line has its own individual ground return line. For example, as shown in Fig. 9.11, pin 2 is the LSB of the data character sent to the printer, and pin 20 is the ground return for this signal. Individual ground returns reduce the chance of picking up electrical noise in the lines. If you are making an interface cable for a parallel printer, these ground return lines should only be connected together and to ground at the microcomputer end of the cable, as shown in Fig. 9.12.

While we are talking about grounds, note that pin 16 is listed as logic ground and pin 17 is listed as chassis ground. In order to prevent large noise currents from flowing in the logic ground wires, these wires should only be connected together in the microcomputer. (This precaution is necessary whenever you connect any external device or system to a microcomputer.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the upper 4 bits of port C as outputs. Fig. 9.14b shows the bit set/reset control word necessary to enable the interrupt request signal on bit PC0 for the handshake. The addresses for the 8255A, A35, on the SDK-86 board are, as shown in Fig. 7.16, port P1A—FFF9H; port P1B—FFF9BH; port PIC—FFF9DH; and control PI—FFFFH. For that system, then, both control words are output to FFFFH.

THE PRINTER DRIVER PROGRAM

Procedures which input data from or output data to peripheral devices such as disk drives, modems, and printers are often called *I/O drivers*. Here we show you one way to write the I/O driver procedure for our parallel-printer interface.

The first point to consider when writing any I/O driver is whether to do it on a polled or on an interrupt basis. For the parallel Centronics interface here, the maximum data transfer rate is about 1000 characters/second. This means that there is about 1 ms between transfers. If characters are sent on an interrupt basis, many other program instructions can be executed while waiting for the interrupt request to send the next character. Also, when the printer buffer gets full, there will be an even longer time that the processor can be working on some other job while waiting for the next interrupt. This is another illustration of how interrupts allow the computer to do several tasks "at the same time." For our example here, assume that the interrupt-request from PC0 of the 8255A is connected to the IR6 interrupt input of the 8259A shown in Fig. 8.14. The higher-priority interrupt inputs on the 8255A are left for a clock interrupt and a keyboard interrupt.

Figure 9.15a, shows the steps needed in the mainline to initialize everything and "call" the printer driver to send a string of ASCII characters to the printer.

At the start of the mainline, some named memory locations are set aside to store parameters needed for transfer of data to the printer. The memory locations set aside for passing information between the mainline and the I/O driver procedure are often called a *control block*. In the control block, a named location is set aside for a pointer to the address of the ASCII character that is currently being sent. Another memory location is set aside to store the number of characters to be sent. The number in this location will function as a counter so you know when you have sent all the characters in the buffer. Instead of using this counter approach to keep track of how many characters have been sent, a *sentinel method* can be used. With the sentinel approach you put a *sentinel* character in

memory after the last character to be sent out. MS/DOS, for example, uses a \$ (24H) as a sentinel character for some of the I/O drivers. As you read each character in from memory, you compare it with the sentinel value. If it matches, you know all the characters have been sent. The sentinel approach and the counter approach are both widely used, so you should be familiar with both.

To get the hardware ready to go, you need to initialize the 8259A and unmask the IR inputs of the 8259A that are used. The 8086 INTR input must also be enabled. Next, the 8255A must be initialized by sending it the mode control word shown in Fig. 9.14a. A bit set/ reset control word is then sent to the 8255A to make the STROBE signal to the printer high, because this is the unasserted level for the signal. When interfacing with hardware, you must always remember to put control and handshake signals such as this in known states.

Also, to make sure the printer is internally initialized, we pulse the INIT line to the printer low for a few microseconds.

When we reach a point in the mainline where we want to print a string, we first read the printer status from port A and check if the printer is selected, not out of paper, and not busy. In a more complete program, we could send a specific error message to the display indicating the type of error found. The program here just sends a general error message. If no printer error condition is found, the starting address of the string of ASCII characters is loaded into the control block location set aside for this, and the number of characters in the string is sent to a reserved location in the control block. Finally, the interrupt request pin on the 8255A is enabled so that printer interrupts can be output to the 8259A IR input. Note that this interrupt is not enabled until everything else is ready. To see how this algorithm is implemented in assembly language, work your way through Fig. 9.16a. The JMP WT at the end of this program represents continued execution of the mainline program while waiting for an interrupt from the printer.

A high on the ACKNLG line from the printer will cause the 8255A to output an Interrupt Request signal. This Interrupt Request signal goes through the 8259A to the processor and causes it to go to the interrupt service procedure.

Figure 9.15b shows the algorithm for the procedure which services this interrupt and actually sends the characters to the printer. After some registers are pushed, the 8086 INTR input is enabled so that higher-priority interrupts such as a clock can interrupt this procedure. The string address pointer is then read in from the control



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

1 ;8086 PROCEDURE F9-16B.ASM use with mainline F9-16A.ASM
2 ;ABSTRACT : Printer Driver procedure outputs a character from a buffer
3 ; to a printer. If no characters are left in the buffer then
4 ; the interrupt to the 8086 on IR6 of the 8259A is disabled.
5 ;PROCEDURES : None used
6 ;PORTS : Uses SDK-86 board Port P1B (FFFFBH) to output characters
7 ; and port P1C bits for handshake signals and printer intr
8 ;REGISTERS ; Destroys nothing
9
10 PUBLIC PRINT_IT
11 0000 DATA SEGMENT PUBLIC
12           EXTRN COUNTER :BYTE, POINTER :WORD
13           EXTRN MESSAGE_1:BYTE, PRINT_DONE:BYTE
14 0000 DATA ENDS
15
16 0000 CODE SEGMENT WORD PUBLIC
17 0000 PRINT_IT PROC FAR
18           ASSUME CS:CODE, DS:DATA
19 0000 9C     PUSHF          ; Save registers
20 0001 50     PUSH AX
21 0002 53     PUSH BX
22 0003 52     PUSH DX
23 0004 FB     STI            ; Enable higher interrupts
24 0005 BA FFFF MOV DX, OFFFBH ; Point at port B
25 0008 8B 1E 0000e MOV BX, POINTER ; Load pointer to message
26 000C 8A 07   MOV AL, [BX]    ; Get a character
27 000E EE     OUT DX, AL    ; Send the character to printer
28           ;Send printer strobe on PC4 low then high
29 000F BA FFFF MOV DX, OFFFFFH ; Point at port control addr
30 0012 B0 08   MOV AL, 00001000B ; Strobe low control word
31 0014 EE     OUT DX, AL
32 0015 B0 09   MOV AL, 00001001B ; Strobe high control word
33 0017 EE     OUT DX, AL
34           ;Increment pointer and decrement counter
35 0018 FF 06 0000e INC POINTER
36 001C FE 0E 0000e DEC COUNTER
37 0020 75 08   JNZ NEXT       ; Wait for next character?
38           ;No more characters-disable 8255A int request on PC0 by bit reset of PC2
39 0022 B0 04   MOV AL, 00000100B ; Bit reset word for PC0 interrupt
40 0024 EE     OUT DX, AL
41 0025 C6 06 0000e 01   MOV PRINT_DONE, 1
42 002A B0 20   NEXT:        MOV AL, 00100000B ; OCW2 for non-specific EOI
43 002C BA FF00   MOV DX, OFFFOOH ; Point at 8259A control addr
44 002F EE     OUT DX, AL
45 0030 5A     POP DX         ; Restore registers
46 0031 5B     POP BX
47 0032 58     POP AX
48 0033 90     POPF
49 0034 CF     IRET
50 0035           PRINT_IT ENDP
51 0035           CODE ENDS
52           END

```

(b)

Fig. 9.16 (b) Procedure.

loop through reading port C and checking bit D1 over and over until you find this bit high. The IBF pin being high means that the input data byte has been latched into the 8255A and can now be read. The timing waveforms for this case are the same as those in Fig. 9.10, except that you are not using the interrupt request output from the 8255A.

Port C bits that are not used for handshake signals and programmed as outputs can be written to by sending bit set/reset control words to the control register. Technically, bits PC0 through PC3 can also be written to directly at the port C address, but we have found it safer to just use

the bit set/reset control word approach to write to all leftover port C bits programmed as outputs.

INTERFACING A MICROPROCESSOR TO KEYBOARDS

Keyboard Types

When you press a key on your computer, you are activating a switch. There are many different ways of making these switches. Here's an overview of the construction and operation of some of the most common types.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

1 ;8086 PROGRAM F9-20.ASM
2 ;ABSTRACT : Program scans and decodes a 16-switch keypad.
3 ; It initializes the ports below and then calls a procedure
4 ; to input an 8-bit value from a 16-switch keypad and encode it.
5 ;PORTS : SDK-86 board Port P1A (FFF9H) - output, P1B (FFFFBH) - input
6 ;PROCEDURES: Calls KEYBRD to scan and decode 16-switch keypad
7 ;REGISTERS : Uses CS,DS,SS,SP,AX,DX
8
9 0000      DATA SEGMENT WORD PUBLIC
10          ;      0   1   2   3   4   5   6   7
11 0000 77 78 7D 7E B7 BB BD + TABLE DB    77H, 78H, 7DH, 7EH, 0B7H, 0BBH, 0BDH, 0BEH
12 BE
13          ;      8   9   A   B   C   D   E   F
14 0008 07 DB DD DE E7 EB ED +   DB  007H, 00BH, 00DH, 00EH, 0E7H, 0EBH, 0EDH, 0EEH
15 EE
16 0010      DATA ENDS
17 0000      STACK_SEG SEGMENT
18 0000 1E*(0000)          DW 30 DUP(0) ; Set up stack of 30 words
19          TOP_STACK LABEL WORD ; Pointer to top of stack
20 003C      STACK_SEG ENDS
21
22 0000      CODE SEGMENT WORD PUBLIC
23          ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
24 0000 B8 0000s
25 0003 BE DD
26 0005 BC 003Cr
27 0008 B8 0000s
28 0008 BE D8
29          ;Initialize ports, mode 0, Port A for output, Ports B & C for input
30 0000 BA FFFF
31 0010 B0 88
32 0012 EE
33 0013 E8 0001
34 0016 90
35          ;Program will continue here with other tasks
36
37          ;8086 PROCEDURE KEYBRD
38 ;ABSTRACT : Procedure gets a code from a 16-switch keypad and decodes it.
39 ; It returns the code for the keypress in AL and AH=00. If there
40 ; is an error in the keypress then it returns AH=01.
41 ;PORTS : Uses SDK-86 ports P1A (FFF9H) for output and P1B (FFFFBH) for input
42 ;INPUTS : Keypress from port
43 ;OUTPUTS : Keypress code or error message in AX
44 ;PROCEDURES: None used
45 ;REGISTERS : Destroys AX
46
47 0017      KEYBRD PROC NEAR
48 0017 9C          PUSHF           ; Save registers used
49 0018 53          PUSH BX
50 0019 51          PUSH CX
51 001A 52          PUSH DX
52          ;Send 0's to all rows
53 001B B0 00          MOV AL, 00
54 001D BA FFF9          MOV DX, 0FFF9H ; Load output address
55 0020 EE          OUT DX, AL ; Send 0's
56          ;Read columns to see if all keys are open
57 0021 BA FFFB          MOV DX, 0FFFFBH ; Load input port address
58 0024 EC          WAIT_OPEN:IN AL, DX
59 0025 24 0F          AND AL, 0FH ; Mask row bits
60 0027 3C 0F          CMP AL, 0FH ; Wait until no keys pressed
61 0029 75 F9          JNE WAIT_OPEN
62          ;Read columns to see if a key is pressed
63 002B EC          WAIT_PRESS:IN AL, DX ; Read columns
64 002C 24 0F          AND AL, 0FH ; Mask row bits
65 002E 3C 0F          CMP AL, 0FH ; See if keypressed
66 0030 74 F9          JE WAIT_PRESS
67          ;Debounce keypress
68 0032 B9 16EA          MOV CX, 16EAH ; Delay of 20 ms
69 0035 E2 FE          DELAY: LOOP DELAY
70          ;Read columns to see if key still pressed
71 0037 EC          IN AL, DX

```

Fig. 9.20 Assembly language instructions for keyboard detect, debounce, and encode procedure.

(Continued)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

high-quality, fully encoded keyboard at the local electronics surplus store for a few dollars. When you get the keyboard home, you find that it works perfectly, but that it outputs EBCDIC codes instead of the ASCII codes that you want. Here's how you use the 8086 XLAT instruction to easily solve this problem.

First, look at Table 1.2, which shows the ASCII and EBCDIC codes. The job you have to do here is to convert each input EBCDIC input code to the corresponding ASCII code. One way to do this is the compare technique described previously for the hex-keyboard example. For that method you would first put the EBCDIC codes in a table in memory in the order shown in Table 1.2 and set up a register as a counter and pointer to the end of the table. Then you enter a loop which compares the EBCDIC character in AL with each of the EBCDIC codes in the table until a match is found. The counter would be decremented after each compare so that when a match was found, the count register would contain the desired ASCII code.

This compare technique works well, but since EBCDIC contains 256 codes, the program will, on the average, have to do 128 compares before a match is found. The compare technique then is often too time-consuming for long tables. The XLAT method is much faster.

displacement from the start of the table equal to the value of the EBCDIC character. For example, the EBCDIC code for uppercase A is C1H, so you put the ASCII code for uppercase A, 41H, at offset C1H in the table, as shown in Fig. 9.22. Since EBCDIC code is an 8-bit code, the table will require 256 memory locations. For EBCDIC values which have no ASCII equivalent, you can just put in 00H because these locations will not be accessed. You can use the DB assembler directive to set up the table, as we did with the row-column table in Fig. 9.20.

To do the actual conversion, you simply load the BX register with the offset of the start of the table, load the EBCDIC character to be converted in the AL register, and do the XLAT instruction. When the 8086 executes the XLAT instruction, it internally adds the EBCDIC value in AL to the starting offset of the table in BX. Because of the way the table is made up, the result of this addition will be a pointer to the desired ASCII value in the table. The 8086 then automatically uses this pointer to copy the desired ASCII character from the table to AL. Later in the chapter we show you another example of the use of the XLAT instruction.

The advantage of the XLAT technique for this conversion is that, no matter where in the table the desired ASCII value is, the conversion only requires execution of two loads and one XLAT instruction. The question may occur to you at this point. If this method is so fast, why didn't we use it for the hex-keypad conversion described earlier? The answer is that since the row-column code from the hex keypad is an 8-bit code, the lookup table for the XLAT method would require 256 memory locations, but only 16 of these would actually be used. This would be a waste of memory, so the compare method is a better choice. Since code conversion is a commonly encountered problem in low-level programming, it is important for you to become familiar with both the compare and the XLAT methods so that you can use the one which best fits a particular application.

INTERFACING TO ALPHANUMERIC DISPLAYS

To give directions or data values to users, many microprocessor-controlled instruments and machines need to display letters of the alphabet and numbers. In systems where a large amount of data needs to be displayed, a CRT is usually used to display the data, so in Chapter 13 we show you how to interface a microcomputer to a CRT. In systems where only a small amount

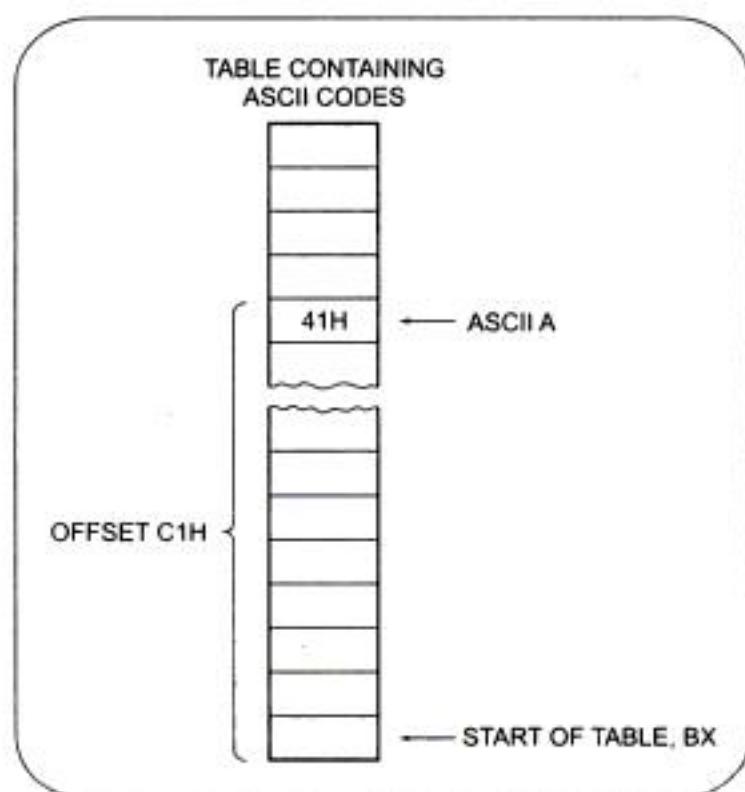


Fig. 9.22 Memory table setup for using XLAT to convert EBCDIC keycode to ASCII equivalent.

The first step in the XLAT method is to make up a memory table which contains all the ASCII codes. The trick here is to put each ASCII code in the table at a



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Second, when multiplexing displays, we pass a higher current through the displays so that they appear as bright as they would if they were not multiplexed. Here's how the 8279 keeps these displays refreshed.

The 8279 contains a 16-byte display refresh RAM. When you want to display some letters or numbers on the LEDs, you write the 7-segment codes for the letters or numbers that you want displayed to the appropriate location in this display RAM. The 8279 then automatically cycles through sending out one of the segment codes, turning on the digit for a short time and then moving on to the next digit. The top five lines in Fig. 9.26, show this multiplex operation in timing diagram form.

The 8279 first outputs the binary number for the first digit to the 7445 on the SL0 to SL3 lines (Fig. 7.8, sheet 7) to turn on the first of the digit-driver transistors. The lines SL0 and SL1 in Fig. 9.26 represent the SL0 and SL1 lines from the 8279. During this time, the 8279 outputs on the A3–A0 and B3–B0 segment lines a code which turns off all the segments. For the circuit in Fig. 7.8, sheet 7, this blanking code will be all zeros (00H). The display is blanked here to prevent "ghosting" of information from one digit to the next when the digit strobe is switched from one digit to the next.

After about 70 µs, the 8279 outputs the 7-segment code for the first digit on the A3–A0 and B3–B0 lines. This will

light the first digit with the desired pattern. After 490 µs, the 8279 outputs the blanking code again. While the displays are blanked, the 8279 sends out the BCD code for the next digit to the 7445 to enable the driver transistor for digit 2. It then sends out the 7-segment code for digit 2 on the A3–A0 and B3–B0 lines. This lights the desired pattern on digit 2. After 490 µs, the 8279 blanks the display again and goes on to digit 3. The 8279 steps through all the digits and then returns to digit 1 and repeats the cycle. Since each digit requires about 640 µs, the 8279 gets back to digit 1 after about 5.1 ms for an 8-digit display and back to digit 1 after about 10.3 ms for a 16-digit display. The time it takes to get back to a digit again is referred to as the *scan time*.

The point here is that once you load the 7-segment codes into the internal display RAM, the 8279 automatically keeps the displays refreshed without any help from the microprocessor. As we will show you later, the 8279 can be connected and initialized to refresh a wide variety of display configurations.

The 8279 can also automatically perform the three tasks for interfacing to a matrix keyboard. Remember from previous discussions that the three tasks involve putting a low on a row of the keyboard matrix and checking the columns of the matrix. If any keys are pressed in that row, a low will be present on the column which contains the

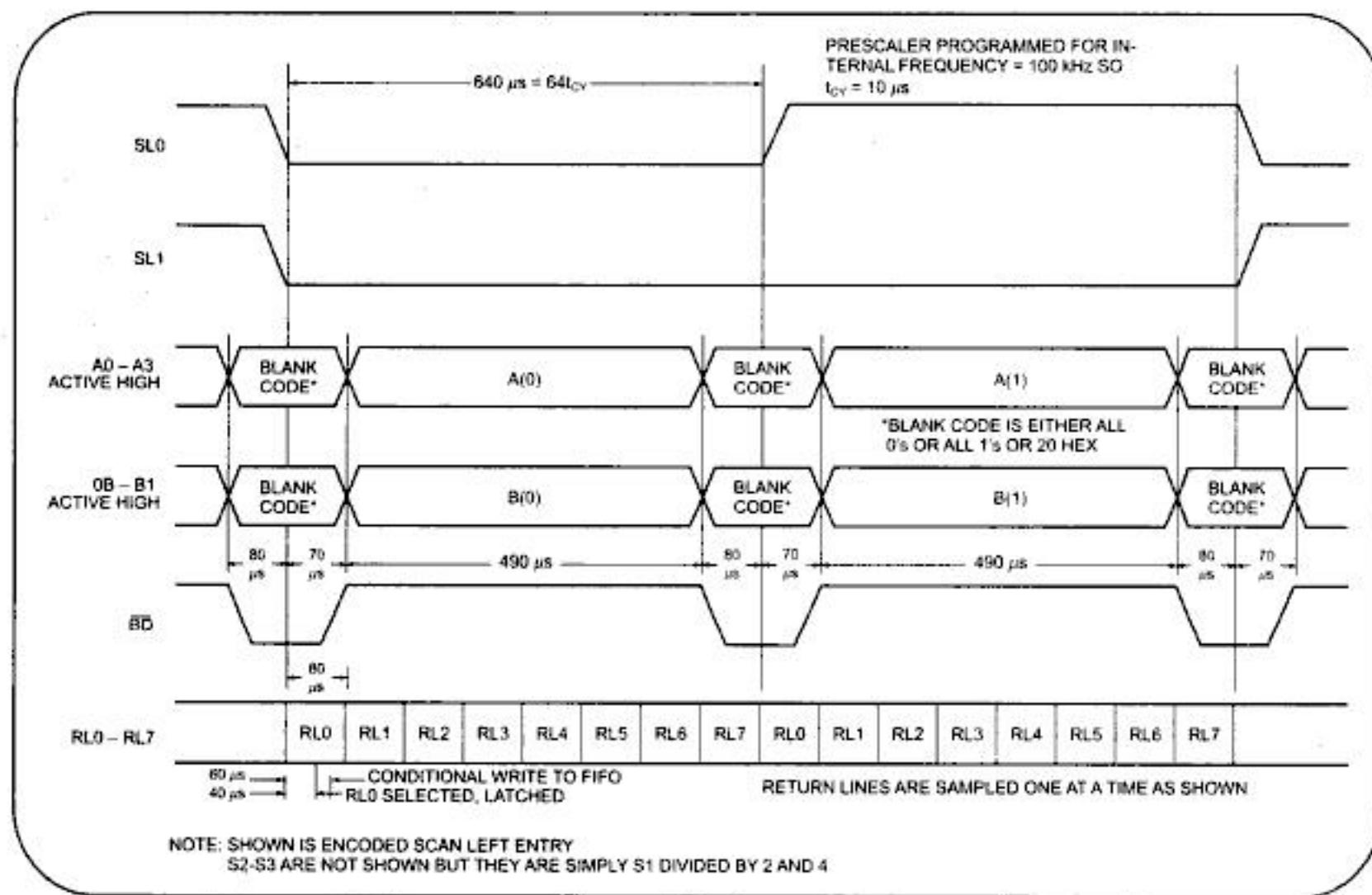


Fig. 9.26 8279 display refresh timing and keyboard scan timing. (Intel Corporation)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

control word can be used to clear the display RAM and/or the FIFO at any time. For now we are only concerned with the first function. The lower 2 bits, labeled C_D in the control word in Fig. 9.28, specify the desired blanking code. The required code will depend on the hardware connections in a particular system. For the SDK-86 a high from the 8279 turns on a segment, so the required blanking code is all 0's. Therefore you can put 0's in the 2 C_D bits. The resultant control word is 11000000.

The three control words described so far take care of the basic initialization. However, before you can send codes to the internal display RAM, you have to send the 8279 a *write-display-RAM* control word. This word tells the 8279 that data sent to the data address later should be put in the display RAM, and it tells the 8279 where to put the data in the display RAM. The 8279 has an internal 4-bit pointer to the display RAM. The lower 4 bits of the write-display-RAM control word initialize the pointer to the location where you want to write a data byte in the RAM. If you want to write a data byte to the first location in the display RAM, for example, you put 0000 in these bits. If you put a 1 in the auto increment bit, labeled AI in the figure, the internal pointer will be automatically incremented to point to the next RAM location after each data byte is written. To start loading characters in the first location in the RAM and select auto increment, then, the control word is 10010000.

```

;INITIALIZATION
MOV DX, OFFEAH ; Point at 8279 control address
MOV AL, 0000000B ; Mode set word for left entry,
; encoded scan, 2-key lockout
OUT DX, AL ; Send to 8279
MOV AL, 00111000B ; Clock word for divide by 24
OUT DX, AL
MOV AL, 11000000B ; Clear display char is alt zeros
OUT DX, AL

;SEND SEVEN SEGMENT CODE TO DISPLAY RAM
MOV AL, 10010000B ; Write display RAM, first location,
; auto increment
MOV DX, OFFEAH ; Point at 8279 control address
OUT DX, AL ; Send control word
MOV DX, OFFE8H ; Point at 8279 data address
MOV AL, 6FH ; Seven segment code for 9
OUT DX, AL ; Send to display RAM
MOV AL, SBH ; Seven segment code for 2
OUT DX, AL ; Send to display RAM
; :
; :
;READ KEYBOARD CODE FROM FIFO
MOV AL, 01000000B ; Control word for read FIFO RAM
MOV DX, OFFEAH ; Point at 8279 control address
OUT DX, AL ; Send control word
MOV DX, OFFE8H ; Point at 8279 data address
IN AL, DX ; Read FIFO RAM

```

Fig. 9.30 8086 instructions to initialize SDK-86 8279, write to display RAM, and read FIFO RAM.

Figure 9.30 shows the sequence of instructions to send the control words we have developed here to the 8279 on the SDK-86 board. Also shown are instructions to send a 7-segment code to the first location in the display RAM. Note that the control words are all sent to the control address, FFEAH, and the character going to the display RAM is sent to the data address, FFE8H. Also note from sheet 7 of Fig. 7.8 that the D0 bit of the byte sent to the display RAM corresponds to segment output B0, and D7 of the byte sent to the display corresponds to segment output A3. This is important to know when you are making up a table of 7-segment codes to send to the 8279.

You now know how to initialize an 8279 and send characters to its display RAM. Two additional points we need to show you are how to read keypressed codes from the FIFO RAM and how to read the status word. In order to read a code from the FIFO RAM, you first have to send a *read FIFO/sensor RAM* control word to the 8279 control address. Figure 9.28 shows the format for this word. For a read of the FIFO RAM, the lower 5 bits of the control word are don't cares, so you can just make them 0's. You send the resultant control word, 01000000, to the control register address and then do a read from the data address. The bottom section of Fig. 9.30 shows this.

Now, suppose that the processor receives an interrupt signal from the 8279, indicating that one or more valid keypresses have occurred. The question then comes up, How do I know how many codes I should read from the FIFO? The answer to this question is that you read the status register from the control register address before you read the FIFO. Figure 9.31 shows the format for this status word. The lowest 3 bits of the status word indicate the number of valid characters in the FIFO. You can load this number into a memory location and count it down as you read in characters. Incidentally, if more than eight characters have been entered in the FIFO, only the last eight will be kept. The error-overrun bit, labeled 0 in the status word, will be set to tell you that characters have been lost.

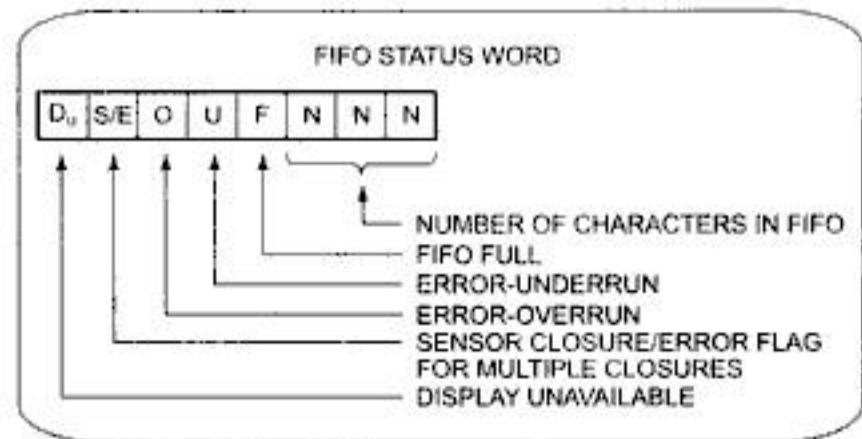


Fig. 9.31 8279 status word format.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

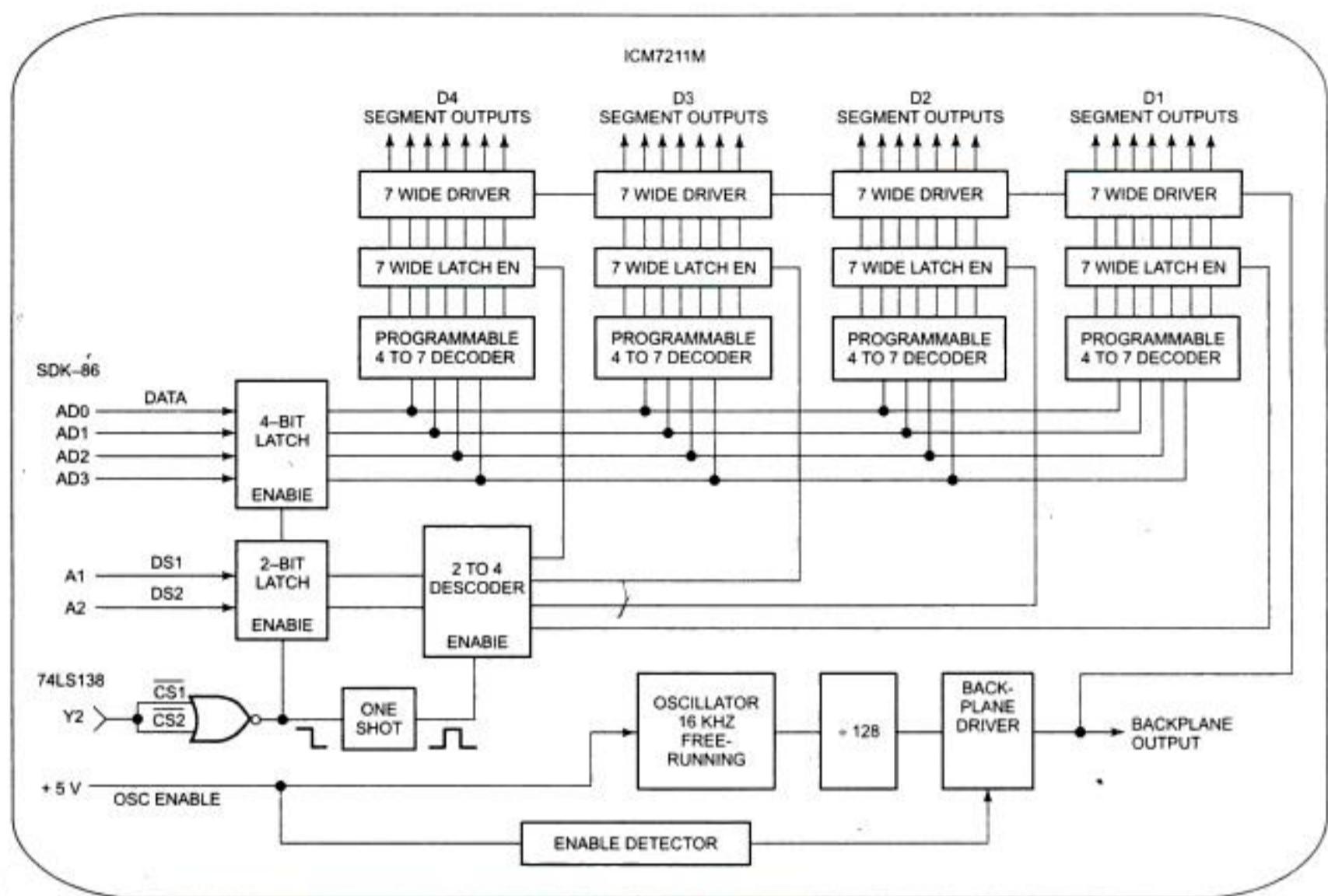


Fig. 9.34 Circuit for interfacing four LCD digits to an SDK-86 bus using Intersil ICM7211M.

INTERFACING MICROCOMPUTER PORTS TO HIGH-POWER DEVICES

The output pins on programmable port devices can typically source only a few tenths of a milliampere from the + 5 V supply and sink only 1 or 2 mA to ground. If you want to control some high-power devices such as lights, heaters, solenoids, and motors with a microcomputer, you need to use interface devices between the port pins and the high-power device. This section shows you a few of the commonly used devices and techniques.

Integrated-Circuit Buffers

One approach to buffering the outputs of port devices is with TTL buffers such as the 7406 hex inverting and 7407 hex noninverting devices. In Fig. 9.12, for example, we show 74LS07 buffers on the lines from ports to a printer. In an actual circuit the 8255A outputs to the computer-controlled lathe in Fig. 9.7 should also have buffers of this type. The 74LS06 and 74LS07 have open-collector outputs, so you have to connect a pull-up resistor from each output to + 5 V. Each of the buffers in a 74LS06 or 74LS07 can sink as much as 40 mA to ground. This is

enough current that you can easily drive an LED with each output by simply connecting the LED and a current-limiting resistor in series between the buffer output and + 5 V.

Buffers of this type have the advantage that they come six to a package, and they are easy to apply. For cases where you need a buffer on only one or two port pins or you need more current, you can use discrete transistors.

Transistor Buffers

Figure 9.35 shows some single-transistor circuits you can connect to microprocessor port lines to drive LEDs or small dc lamps. We will show you how to quickly determine the parts values to put in these circuits for your particular application. First, determine whether you want a logic high on the output port pin to turn on the device or whether you want a logic low to turn on the device. If you want a logic high to turn on the LED, then use the NPN circuit. If you want a logic low to turn on the device, use the PNP circuit. Let's use an NPN for the first example.

Next, determine how much current you need to flow through the LED, lamp, or other device. For our example



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

As the contacts become oxidized, they make a higher-resistance contact and may get hot enough to melt. Another disadvantage of mechanical relays is that they can switch on or off at any point in the ac cycle. Switching on or off at a high-voltage point in the ac cycle can cause a large amount of electrical noise, called electromagnetic interference (EMI). The solid-state relays discussed next avoid these problems to a large extent.

Figure 9.38b shows a picture of a solid-state relay which is rated for 25 A at 25°C if mounted on a suitable heat sink. Figure 9.38c shows a block diagram of the circuitry in the device and how it is connected from an output port to an ac load.

The input circuit of the solid-state relay is just an LED. A simple NPN transistor buffer and a current-limiting resistor are all that is needed to interface the relay to a microcomputer output port pin. To turn the relay on, you simply output a high on the port pin. This turns on the transistor and pulls the required 11 mA through the internal LED. The light from the LED is focused on a phototransistor connected to the actual output-control circuitry. Since the only connection between the input circuit and the output circuit is a beam of light, there are several thousand volts of isolation between the input circuitry and the output circuitry.

The actual switch in a solid-state relay is a triac. When triggered, this device conducts on either half of the ac cycle. The zero-voltage detector makes sure that the triac is only triggered when the ac line voltage is very close to one of its zero-voltage crossing points. If you output a signal to turn on the relay, the relay will not actually turn on until the next time the ac line voltage crosses zero. This prevents the triac from turning on at a high-voltage point in the ac cycle, which would produce a burst of EMI. Triacs automatically turn off when the current through them drops below a small value called the holding current, so the triac automatically turns off at the end of each half-cycle of the ac power. If the control signal is on, the trigger circuitry will automatically retrigger the triac for each half-cycle. If you send a signal to turn off the relay, it will actually turn off the next time the alternating current drops to zero. In this type of solid-state relay, the triac is always turned on or off at a zero point on the ac voltage. Zero-point switching eliminates most of the EMI that would be caused by switching the triac on at random points in the ac cycle.

Solid-state relays have the advantages that they produce less EMI, they have no mechanical contacts to arc, and they are easily driven from microcomputer ports. Their disadvantages are that they are more expensive than

equivalent mechanical relays and there is a voltage drop of a couple of volts across the triacs when they are on. Another potential problem with solid-state relays occurs when driving large inductive loads, such as motors. Remember from basic ac theory that the voltage waveform leads the current waveform in an ac circuit with inductance. A triac turns off when the current through it drops to near zero. In an inductive circuit, the voltage waveform may be at several tens of volts when the current is at zero. When the triac is conducting, it has perhaps 2 V across it. When the triac turns off, the voltage across the triac will quickly jump to several tens of volts. This large dV/dT may possibly turn on the triac at a point when you don't want it turned on. To keep the voltage across the triac from changing too rapidly, an *RC snubber* circuit is connected across the triac, as shown in Fig. 9.38c. A system example in the next chapter uses a solid-state relay to control an electric heater.

Interfacing a Microcomputer to a Stepper Motor

A unique type of motor useful for moving things in small increments is a stepper motor. Instead of rotating smoothly around and around as most motors do, stepper motors rotate, or "step," from one fixed position to the next. If you have a dot-matrix printer such as an Epson FX, look inside and you should see one small stepper motor which is used to advance the paper to the next line position and another small stepper motor which is used to move the print head to the next character position. While you are in there, you might look for a small device containing an LED and a phototransistor which detects when the print head is in the "home" position. Stepper motors are also used to position the read/write head over the desired track of a floppy disk and to move the pen around on X-Y plotters.

Common step sizes for stepper motors range from 0.9° to 30°. A stepper motor is stepped from one position to the next by changing the currents through the fields in the motor. The two common field connections are referred to as two-phase and four-phase. We will discuss *Joule-phase steppers* here because their drive circuitry is much simpler.

Figure 9.39 shows a circuit you can use to interface a small four-phase stepper such as the Superior Electric MO61-FD302, IMC Magnetics Corp. Tormax 200, or a similar, nominal 5 V unit to five microcomputer port lines. If you build up this circuit, bolt some small heat sinks on the MJE2955 transistors and mount the 10-W resistors where you aren't likely to touch them.

Since the 7406 buffers are inverting, a high on an output-port pin produces a low on a buffer output. This



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

10

Analog Interfacing and Industrial Control

In order to control the machines in our electronics factory, medical instruments, or automobiles with microcomputers, we need to determine the values of variables such as pressure, temperature, and flow. There are usually several steps in getting electrical signals which represent the values of these variables and converting the electrical signals to digital forms the microcomputer can work with.

The first step involves a *sensor*, which converts the physical pressure, temperature, or other variable to a proportional voltage or current. The electrical signals from most sensors are quite small, so they must next be amplified and perhaps filtered. This is usually done with some type of operational-amplifier (op-amp) circuit. The final step is to convert the signal to digital form with an analog-to-digital (A/D) converter.

In this chapter we review some op-amp circuits commonly used in these steps, show the interface circuitry for some common sensors, and discuss the operation and interfacing of D/A converters. We also discuss the operation and interfacing of A/D converters and show how all of these pieces are put together in a microcomputer-based scale and a microcomputer-based machine-control system. As part of these examples, we discuss the tools and techniques used to develop microcomputer-based products. Finally, we discuss how an A/D converter, a microcomputer, and a D/A converter can be used to produce a digital filter.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Recognize several common op-amp circuits, describe their operation, and predict the voltages at key points in each.
2. Describe the operation and interfacing of several common sensors used to measure temperature, pressure, flow, etc.
3. Describe the operation of a D/A converter and define D/A data-sheet parameters, such as resolution, settling time, accuracy, and linearity.
4. Draw circuits showing how to interface D/A converters with any number of bits to a microcomputer.
5. Describe briefly the operation of flash, successive-approximation, and ramp A/D converters.
6. Draw circuits showing how A/D converters of various types can be interfaced to a microcomputer.
7. Write programs to control A/D and D/A converters.
8. Describe how feedback is used to control variables such as pressure, temperature, flow, motor speed, etc.
9. Describe the operation of a "time-slice" factory-control system.
10. Describe the tools and techniques currently used to develop a microcomputer-based product.
11. Draw a block diagram of a digital filter and briefly describe its basic operation.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

this frequency the gain will drop off. The critical frequency for the circuit is determined by the equation next to the circuit. The equation assumes that R_1 and R_2 are equal and that the value of C_1 is twice the value of C_2 . R_3 is simply a damping resistor. The positive feedback supplied by C_1 is the reason the gain is only down to 0.707 at the critical frequency, rather than down to 0.5 as it would be if we cascaded two simple RC circuits.

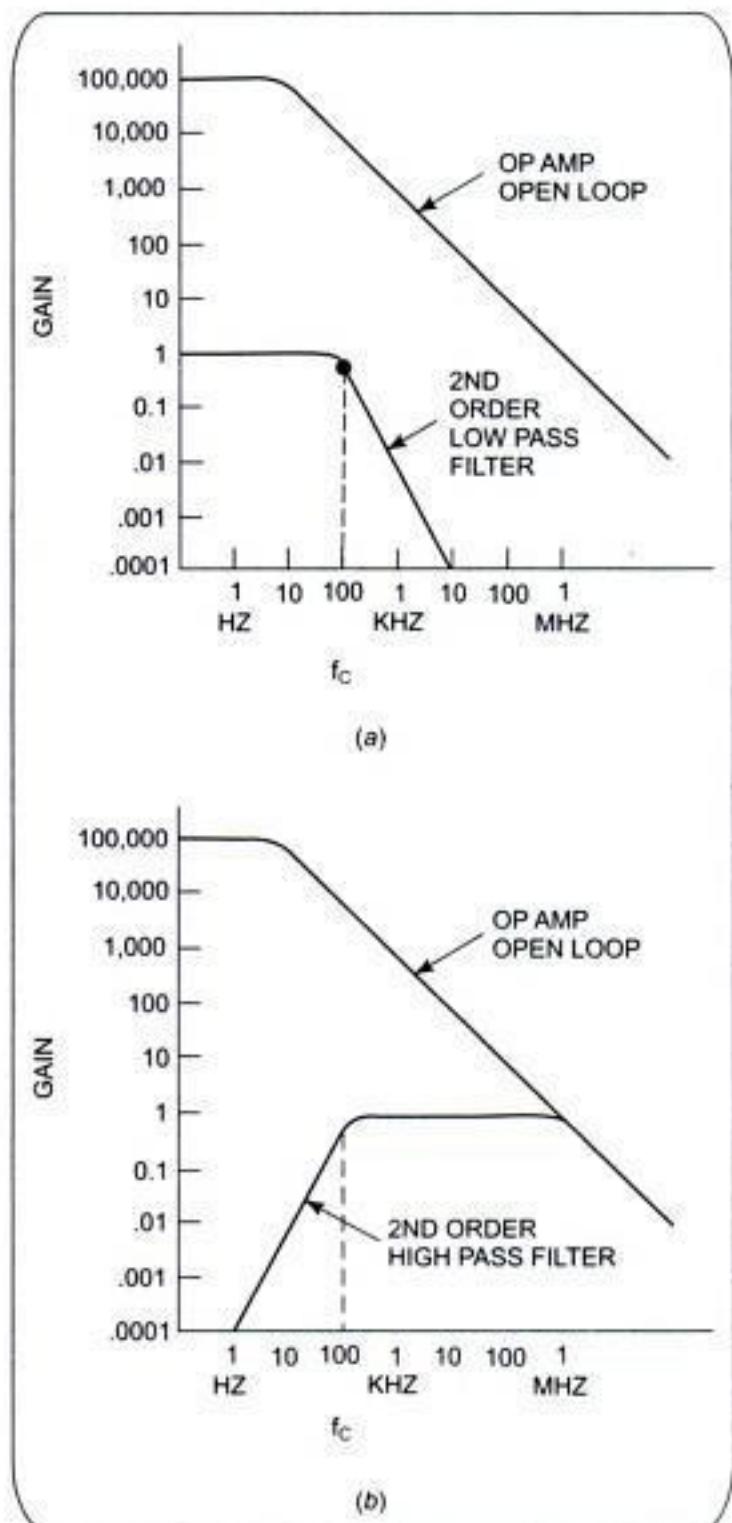


Fig. 10.3 Gain versus frequency response for second-order low-pass and high-pass filters. (a) Low-pass. (b) High-pass.

For the high-pass filter, the gain for the flat section of the response curve is also 1. Assuming that the two capacitors are equal and the value of R_2 is twice the value

of R_1 , the critical frequency is determined by the formula shown next to Fig. 10.1*f*. Again, R_3 is for damping.

A low-pass filter can be put in series with a high-pass filter to produce a bandpass filter which lets through a desired range of frequencies. There are also many different single-amplifier circuits which will pass or reject a band of frequencies.

Now that we have refreshed your memory of basic op-amp circuits, we will next discuss some of the different types of sensors you can use to produce electrical signals proportional to the values of temperatures, pressures, position, etc.

SENSORS AND TRANSDUCERS

It would take a book many times the size of this one to describe the operation and applications of all the different types of available sensors and transducers. What we want to do here is introduce you to a few of these and show how they can be used to get data for microcomputer-based machines in, for example, our electronics factory.

Light Sensors

One of the simplest light sensors is a light-dependent resistor such as the Clairex CL905 shown in Fig. 10.4*a*. A glass window allows light to fall on a zigzag pattern of

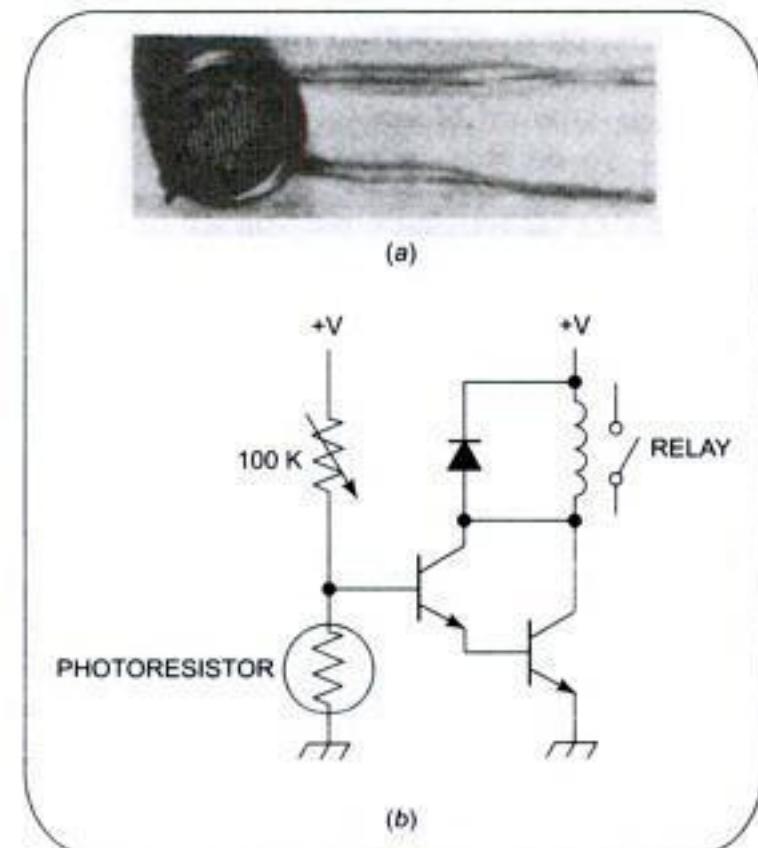


Fig. 10.4 (a) Cadmium sulfide photocell. (Clairex Electronics) (b) Light-controller relay circuit using a photocell.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

output will be $10\text{ V} \times (255/256)$ or 9.961 V. Even though the output voltage can never actually get to 10 V, this is referred to as a 10 V *output converter*. The maximum output voltage of a converter will always have a value 1 least significant bit less than the named value. As another example of this, suppose that you have a 12-bit, 10 V converter. The value of 1 LSB will be $(10\text{ V})/4096$ or 2.44 mV. The highest voltage out of this converter when it is properly adjusted will then be $(10.0000 - 0.0024)$ V or 9.9976 V.

Several different binary codes, such as *straight binary*, BCD, and *offset binary*, are commonly used as inputs to D/A converters. We will show examples of these codes in a later discussion of A/D converters.

The accuracy specification for a D/A converter is a comparison between the actual output and the expected output. It is specified as a percentage of the full-scale output voltage or current. If a converter has a full-scale output of 10 V and ± 0.2 percent accuracy, then the *maximum error* for any output will be 0.002×10.00 V or 20 mV. Ideally the maximum error for a D/A converter should be no more than $\pm \frac{1}{2}$ the value of the LSB.

Another important specification for a D/A converter is *linearity*. Linearity is a measure of how much the output ramp deviates from a straight line as the converter is stepped from no switches on to all switches on. Ideally, the deviation of the output from a straight line as the converter is stepped from no switches on to all switches on. Ideally, the deviation of the output from a straight line should be no greater than $\pm \frac{1}{2}$ the value of the LSB to maintain overall accuracy. However, many D/A converters are marketed which have linearity errors greater than that. National Semiconductor, for example, markets the DAC1020, DAC1021, DAC1022 series of 10-bit-resolution converters. The linearity specification for the DAC1020 is 0.05 percent, which is appropriate for a 10-bit converter. The DAC1021 has a linearity specification of 0.10 percent, and the DAC1022 has a specification of 0.20 percent. The question that may occur to you at this point is, What good is it to have a 10-bit converter if the linearity is only equivalent to that of an 8- or 9-bit converter? The answer to this question is that for many applications, the resolution given by a 10-bit converter is needed for small output signals, but it doesn't matter if the output value is somewhat nonlinear for large signals. The price you pay for a D/A converter is proportional not only to its resolution, but also to its linearity specification.

Still another D/A specification to look for is *settling time*. When you change the binary word applied to the input of a converter, the output will change to the appropriate new value. The output, however, may over-shoot the correct

value and "ring" for a while before finally settling down to the correct value. The time the output takes to get within $\pm \frac{1}{2}$ LSB of the final value is called settling time. As an example, the National DAC1020 10-bit converter has a typical settling time of 500 ns for a full-scale change on the output. This specification is important because if a converter is operated at too high a frequency, it may not have time to settle to one value before it is switched to the next.

D/A Applications and Interfacing to Microcomputers

D/A converters have many applications besides those where they are used with a microcomputer. In a compact-disk audio player, for example, a 14- or 16-bit D/A converter is used to convert the binary data read off the disk by a laser to an analog audio signal. Most speech-synthesizer ICs contain a D/A converter to convert stored binary data for words into analog audio signals. Here, however, we are primarily interested in the use of a D/A converter with a microcomputer.

The inputs of the D/A circuit (A1 through A8) in Fig. 10.15 can be connected directly to a microcomputer output port. As part of a program, you can produce any desired voltage on the output of the D/A. Here are some ideas as to what you might use this circuit for.

As a first example, suppose that you want to build a microcomputer-controlled tester which determines the effect of power supply voltage on the output voltage of some integrated-circuit amplifiers. If you connect the output of the D/A converter to the reference input of a programmable power supply or simply add the high-current buffer circuit shown in Fig. 10.16 to the output of the D/A, you have a power supply which you can vary under program control. To determine the output voltage of the IC under test as you vary its supply voltage, connect the input of an A/D converter to the IC output, and connect the output of the A/D converter to an input port of your microcomputer. You can then read in the value of the output voltage on the IC.

Another application you might use a D/A and a power buffer for is to vary the voltage supplied to a small resistive heater under program control. Also, the speed of small dc motors is proportional to the amount of current passed through them, so you could connect a small dc motor to the output of the power buffer and control the speed of the motor with the value you output to the D/A. Note that without feedback control, the speed of the motor will vary if the load changes. Later in the chapter we show you how to add feedback control to maintain constant motor speed under changing loads.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

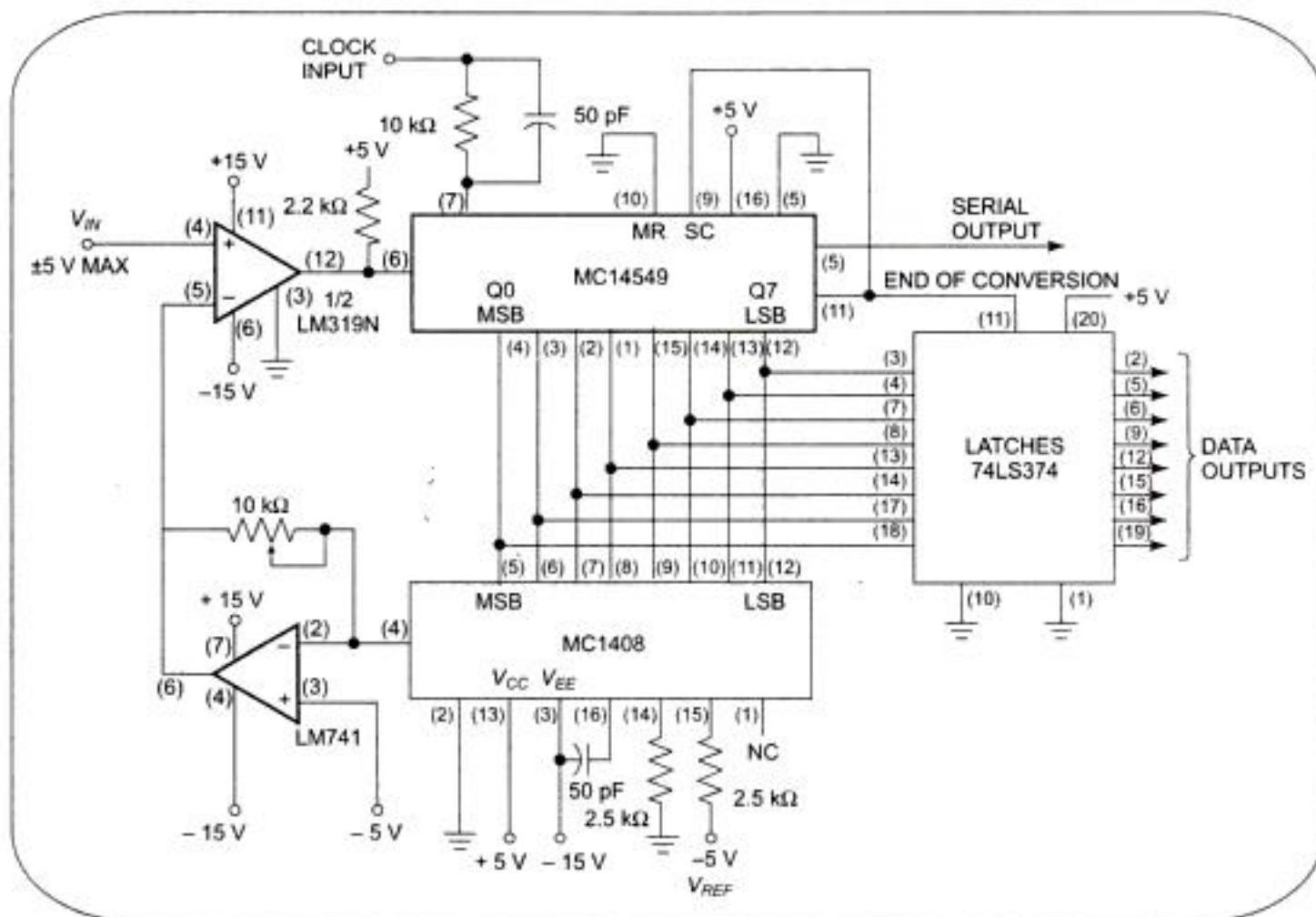


Fig. 10.20 Successive-approximation A/D converter circuit.

comparator output will be high, which tells the SAR to keep that bit on. When the next clock pulse occurs, the SAR will turn on the next most significant bit to the D/A converter. Based on the answer this produces from the comparator, the SAR will keep or reset this bit. The SAR proceeds in this way on down to the least significant bit, adding each bit to the total in turn and using the signal from the comparator to decide whether to keep that bit in the result. Only nine clock pulses are needed to do the actual conversion here. When the conversion is complete, the binary result is on the parallel outputs of the SAR, and the SAR sends out an end-of-conversion (EOC) signal to indicate this. In the circuit in Fig. 10.20, the EOC signal is used to strobe the binary result into some latches, where it can be read by a microcomputer. If the EOC signal is connected to the start-conversion (SC) input as shown, then the converter will do continuous conversions. Note in the circuit in Fig. 10.20 that the noninverting input of the op amp on the 1408 D/A converter is connected to -5 V instead of to ground. This shifts the analog input range to -5 V to $+5\text{ V}$ instead of 0 V to $+10\text{ V}$ so that sine waves and other ac signals can be input directly to the converter to be digitized.

The National ADC 1280 is a single-chip 12-bit successive-approximation converter which does a conversion in about $22\text{ }\mu\text{s}$. Datel and Analog Devices have several 12-bit converters with conversion times of about $1\text{ }\mu\text{s}$.

| UNIPOLAR BINARY CODES | | | | | |
|---------------------------------|---------------------|--------------|---------------------------|----------------------|-------------------------------------|
| VALUE | 10 VOLTS FULL SCALE | BINARY (BIN) | COMPLEMENTARY BINARY (CB) | INVERTED BINARY (IB) | INVERTED COMPLEMENTARY BINARY (ICB) |
| $+FS - 1\text{ LSB}$ | 9.9609 | 1111 1111 | 0000 0000 | | |
| $+\frac{1}{2}FS$ | 5.0000 | 1000 0000 | 0111 1111 | | |
| $+\frac{1}{2}FS - 1\text{ LSB}$ | 4.9609 | 0111 1111 | 1000 0000 | | |
| $+1\text{ LSB}$ | 0.0391 | 0000 0001 | 1111 1110 | | |
| ZERO | 0.0000 | 0000 0000 | 1111 1111 | 0000 0000 | 1111 1111 |
| -1 LSB | -0.0391 | | | 0000 0001 | 1111 1110 |
| $-\frac{1}{2}FS + 1\text{ LSB}$ | -4.9609 | | | 0111 1111 | 1000 0000 |
| $-\frac{1}{2}FS$ | -5.0000 | | | 1000 0000 | 0111 1111 |
| $-FS + 1\text{ LSB}$ | -9.9609 | | | 1111 1111 | 0000 0000 |

| UNIPOLAR BINARY CODED DECIMAL CODES | | | | | |
|-------------------------------------|---------------------|----------------------------|---|--------------------------------------|---|
| VALUE | 10 VOLTS FULL SCALE | BINARY CODED DECIMAL (BCD) | COMPLEMENTARY BINARY CODED DECIMAL (CBCD) | INVERTED BINARY CODED DECIMAL (IBCD) | INVERTED COMPLEMENTARY BINARY CODED DECIMAL (ICBCD) |
| $+FS - 1\text{ LSB}$ | 9.9 | 1001 1001 | 0110 0110 | | |
| $+\frac{1}{2}FS$ | 5.0 | 0101 0000 | 1010 1111 | | |
| $+1\text{ LSB}$ | 0.1 | 0000 0001 | 1111 1110 | | |
| ZERO | 0.0 | 0000 0000 | 1111 1111 | 0000 0000 | 1111 1111 |
| -1 LSB | -0.1 | | | 0000 0001 | 1111 1110 |
| $-\frac{1}{2}FS$ | -5.0 | | | 0101 0000 | 1010 1111 |
| $-FS + 1\text{ LSB}$ | -9.9 | | | 1001 1001 | 0110 0110 |

| BIPOLAR BINARY CODES | | | | |
|----------------------|---------------------------|--------------------|-----------------------------------|-----------------------|
| VALUE | 10 VOLTS FULL SCALE RANGE | OFFSET BINARY (OB) | COMPLEMENTARY OFFSET BINARY (COB) | TWO'S COMPLEMENT (TC) |
| $+FS$ | 5.0000 | | | |
| $+FS - 1\text{ LSB}$ | 4.9609 | 1111 1111 | 0000 0000 | 0111 1111 |
| $+1\text{ LSB}$ | 0.0391 | 1000 0001 | 0011 1110 | 0000 0001 |
| ZERO | 0.0000 | 1000 0000 | 0111 1111 | 0000 0000 |
| -1 LSB | -0.0391 | 0111 1111 | 1000 0000 | 1111 1111 |
| $-FS + 1\text{ LSB}$ | -4.9609 | 0000 0001 | 1111 1110 | 1000 0001 |
| $-FS$ | -5.0000 | 0000 0000 | 1111 1111 | 1000 0000 |

Fig. 10.21 Common A/D output codes.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

After displaying the weight, the program reads the 8279 status register to see if the operator has pressed a number key to start entering a price per pound. If no key has been pressed or if a nonnumeric key has been pressed, the program simply goes back and reads the weight again. If a number key has been pressed, the weight is removed from the address field and the letters SP (selling price) are displayed in the address field. The entered number is put in the SELL_PRICE buffer and displayed on the rightmost digit of the data field. The program then polls the 8279 status register until another keypress is detected. If the pressed key is a numeric key, then the code(s) for the previously entered number(s) will be shifted one location in the buffer to make room for the new number. The new number is then put in the first location in the buffer so that it will be displayed in the rightmost digit of the display. In other words, previously entered numbers are continuously shifted to the left as new numbers are entered. If a mistake is made, the operator can simply enter a 0 followed by the correct price per pound.

When a nonnumeric key is pressed, this is the signal that the displayed price per pound is correct and that the total price should now be computed. Before the weight and the price per pound can be multiplied, however, they must each be put in packed BCD form and converted to binary.

The PACK procedure converts four unpacked BCD digits in a memory buffer pointed to by BX to a 4-digit packed result in AX. This procedure is simply some masking and moving of nibbles. Once the weight and price per pound are packed in BCD form, the CONVERT2BIN procedure is used to convert each to its binary equivalent. The algorithm for this procedure is explained in detail in Chapter 5.

Unlike earlier processors, which required a messy procedure for multiplication, a single 8086 MUL instruction does the 16×16 binary multiply to produce the binary equivalent of the total price. The procedure BINCVT is used to convert the binary total price to the packed BCD form needed for the DISPLAY_IT procedure. Here's how the BINCVT procedure works.

In a binary number, each bit position represents a power of 2. An 8-bit binary number, for example, can be represented as:

$$\begin{aligned} b7 \times 2^7 + b6 \times 2^6 + b5 \times 2^5 + b4 \times 2^4 \\ + b3 \times 2^3 + b2 \times 2^2 + b1 \times 2^1 + b0 \end{aligned}$$

This can be shuffled around and expressed as

Binary number

$$\begin{aligned} = (((((2b7 + b6) 2 + b5) 2 + b4) 2 \\ + b3) 2 + b2) 2 + b1) 2 + b0 \end{aligned}$$

where b7 through b0 are the values of the binary bits. If we start with a binary number and do each operation in the nested parentheses in BCD with the aid of the DAA instruction, the result will be the BCD number equivalent to the original binary number.

The procedure in Fig. 10.25 produces two BCD digits of the result at a time by calling the subprocedure CNVT1. Fig. 10.26 shows a flowchart for the operation of CNVT1. The main principle here is to shift the 24-bit number left one bit position so that the MSB goes into the carry flip-flop and then add this bit to twice the previous result. We use the DAA instruction to keep the result of the addition in BCD format. If the DAA produces a carry, we add this carry back into the shifted 24-bit number in DL and BX so that it will be propagated into higher BCD digits. After each run of CNVT1 (24 runs of CNVT2), DL and BX will be left with a binary number which is equal to the original binary number minus the value of the two BCD digits produced. You can adapt this procedure to work with a different number of bits by simply calling CNVT1 more or fewer times and by adjusting the count loaded into DH to be 1 more than the number of binary bits in the number to be converted. The count has to be 1 greater because of the position of the decrement in the loop. The temperature-controller procedure in Fig. 10.35 shows another example of this conversion.

The least significant two digits of the BCD value for the total price returned by BINCVT in BL represent tenths and hundredths of a cent. If the value of these two BCD digits is greater than 49H, then the carry produced by the compare instruction and the next two higher BCD digits in BH are added to AL. This must be done in AL, because the DAA instruction, used to keep the result in BCD format, only works on an operand in AL. Any carry from these two BCD digits is propagated on to the upper two digits of the result in DL. After this rounding off, the packed BCD for the total price is left in AX.

In order for the display procedure to be able to display this price, it must be converted to unpacked BCD form and put in four successive memory locations. Another "mask and move nibbles" procedure called EXPAND does this. The DISPLAY_IT procedure is then called to display the total price on the data field. The DISPLAY_IT procedure is called again to display the letters Pr in the address field.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

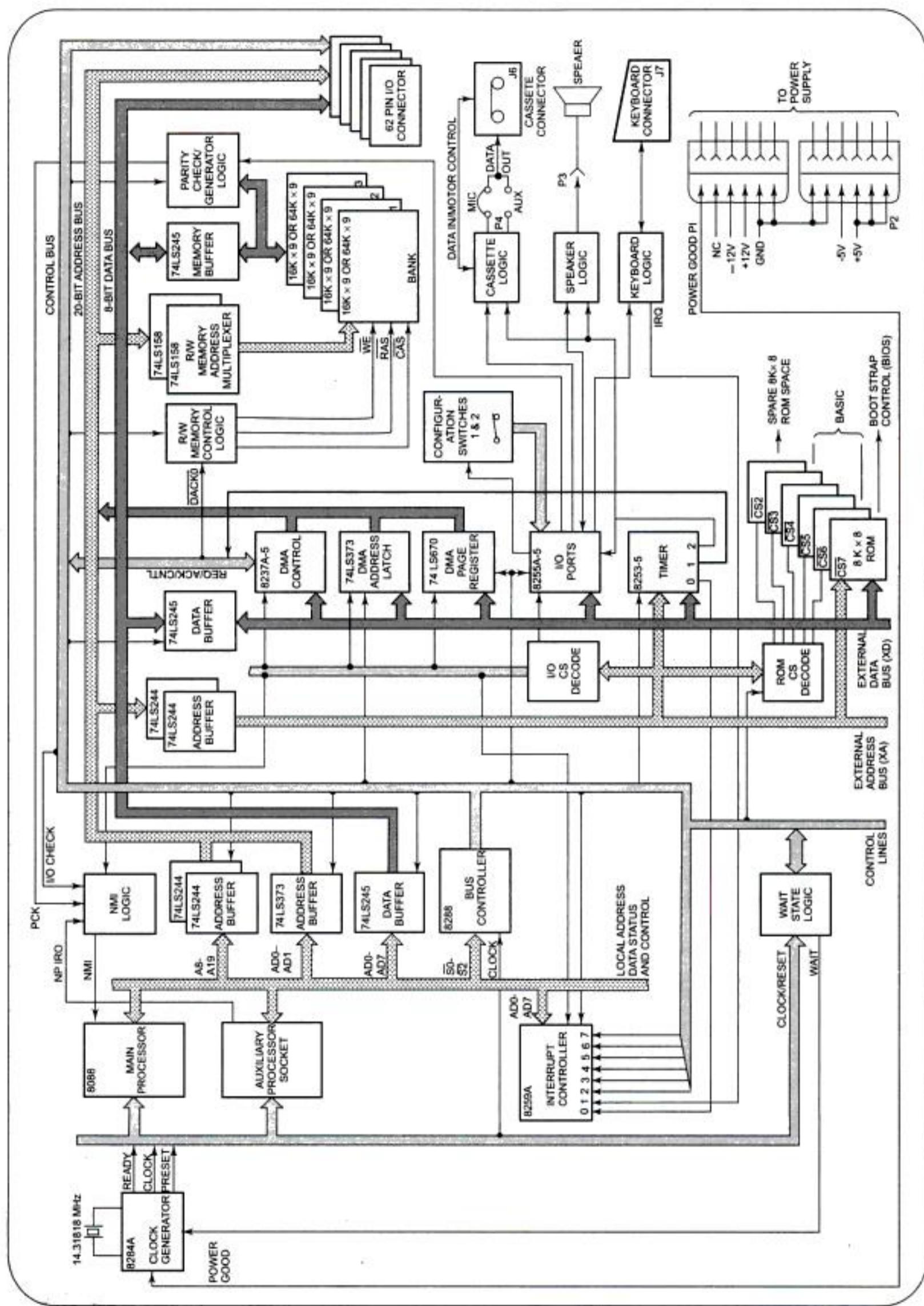


Fig. 11.2 Block diagram of circuitry on IBM PC motherboard. (IBM Corporation)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



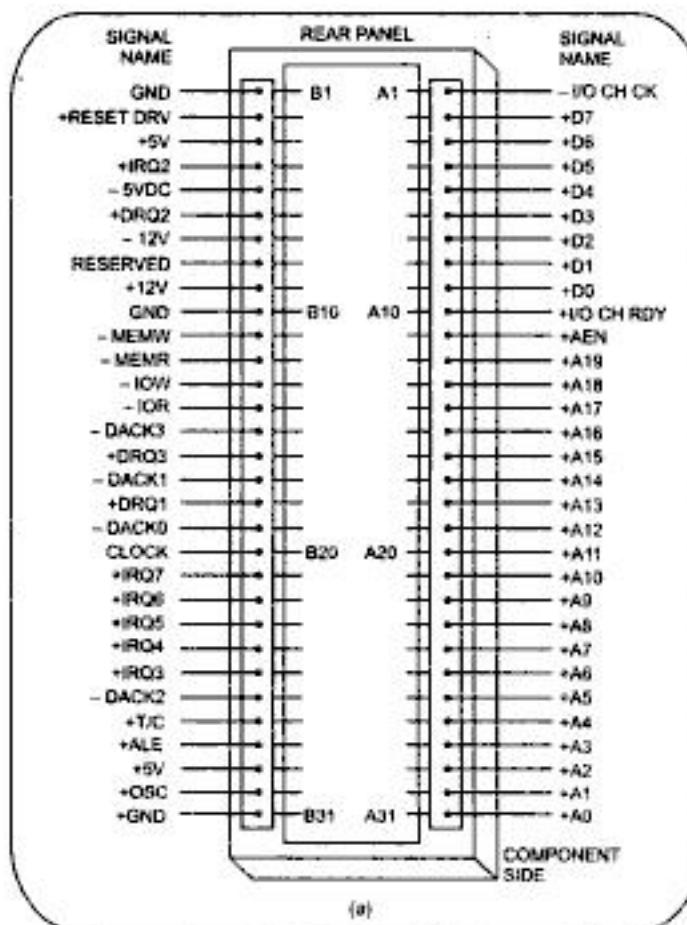
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

mode words, the starting memory address, and the number of bytes to be transferred. Each channel of the 8237 can be programmed to transfer a single byte for each request, a block of bytes for each request, or to keep transferring bytes until it receives a wait signal on the EOP input/output. Consult the data sheet in an Intel data book to get the details of each command word.

DMA and the IBM PC

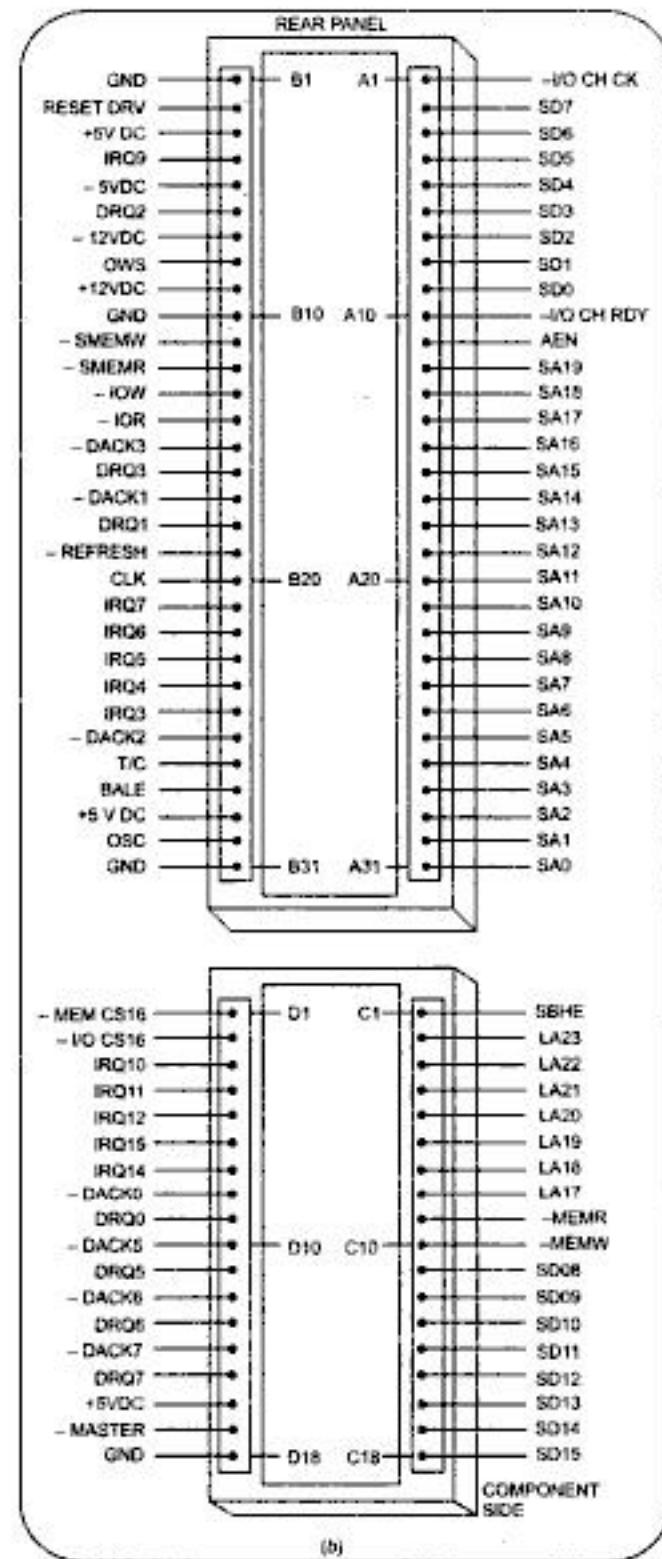
Now that you know how DMA operates, let's take a look back at the DMA section in the block diagram of the IBM PC motherboard circuitry in Fig. 11.2. The 8237A-5 is, of course, the DMA controller. The 74LS373 just under it is used to grab the upper 8 bits of the DMA address sent out on the data bus by the 8237A-5 during a transfer. This device has the same function as device U2 in Fig. 11.5. The 74LS670 just below this is used to output bits A16-A19 of the DMA transfer address, the same function performed by U3 in the circuit in Fig. 11.5.

In order that peripheral boards can interface with the motherboard circuitry on a DMA basis, the DMA signal lines are connected to the peripheral connectors shown in the upper right corner of Fig. 11.2. To see how DMA and other signals go to the peripheral boards, take a look at the pin descriptions in Fig. 11.7.



(a)

The signals shown in Fig. 11.7a are bused to all five peripheral connectors in parallel so that any board can access them. Most of the signals on these connectors should be easily recognizable to you. A + in front of a signal indicates that the signal is active high, and a - indicates that the signal is active low. A0 through A19 on the connectors are the 20 demultiplexed address lines, and D0 through D7 are the eight data lines. IRQ2 through IRQ7 are interrupt request lines which go to the 8259A priority-interrupt controller so that peripheral boards can interrupt the 8086 if necessary. Some other simple signals on the connectors are the power supply voltages; the standard ALE, -MEMW, -MEMR, -IOW, and -IOR control bus signals; and some clock signals. The I/O CH RDY pin on the connector can be asserted by a peripheral board to cause the 8086 to insert WAIT states until the peripheral board is ready.



(b)

Fig. 11.7 Pin names and numbers for peripheral slots. (a) On IBM PC motherboard. (b) On IBM PC/AT motherboard. (IBM Corporation)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Figure 11.10b shows the read timing waveforms for a Texas Instruments TMS44C257 DRAM which is designed for static column mode operation. During the first access in a row, the DRAM controller carries out a normal row address- RAS, column address- CAS sequence of signals. If the next address the controller sends out is in the same row, an external comparator will signal the DRAM controller. In response to this "same-row" signal, the DRAM controller will hold RAS and CAS low and send out just the column address to the A0-A8 inputs of the DRAMs. As long as the microprocessor continues to access memory locations in the same page (row), the controller will simply hold RAS and CAS low and send out the column part of the addresses to the DRAMs. The static column mode is more difficult to implement than the page mode, but it is faster than the page mode because it does not require CAS strobes and the associated setup and hold times.

In a high-speed microprocessor system, the static column decode technique can reduce the average number of wait states per memory access from 2 or 3 to perhaps 0.8. This is a considerable improvement, but it is not as much of an improvement as can be gained by using a cache system.

Cache Mode DRAM Systems

INTRODUCTION

Traditionally the term *cache*, which is pronounced "cash," refers to a hiding place where you put provisions for future use. As we describe how a cache memory system is implemented in a microcomputer, perhaps you can see why the term is used here.

Figure 11.11 shows in block diagram form how a simple cache memory system is implemented in an 80386 based microcomputer system. In Chapter 15 we discuss the details of the 80386 microprocessor, but for this discussion all you need to know is that the 80386 has a 32-bit data bus and a 32-bit address bus. A 32-bit address bus allows the 80386 to address up to 4 Gbytes of memory and a 32-bit data bus allows the 80386 to read or write 4 bytes in parallel.

The cache in a system such as this consists of perhaps 32 or 64 Kbytes of high-speed SRAM. The main memory consists of a few megabytes or more of slower but cheaper DRAM. The general principle of a cache system is that code and data sections currently being used are copied from the DRAM to the high-speed SRAM cache, where they can be accessed by the processor with no wait states. A cache system takes advantage of the fact that most microcomputer programs work with only small

sections of code and data at a particular time. The fancy term for this is "locality of reference." Here's how the system works.

When the microprocessor outputs an address, the cache controller checks to see if the contents of that address have previously been transferred to the cache. If the addressed code or data word is present in the cache, the cache controller enables the cache memory to output the addressed word on the data bus. Since this access is to the fast SRAM, no wait states are required.

If the addressed word is not in the cache, the cache controller enables the DRAM controller. The DRAM controller then sends the address on to the main memory to get the data word. Since the DRAM main memory is slower, this access requires one or two wait states. However, when a word is read from main memory, it not only goes to the microprocessor, it is also written to the cache. If the processor needs to access this data word again, it can then read the data directly from the cache with no wait states. The percentage of accesses where the microprocessor finds the code or data word it needs in the cache is called the *hit rate*. Current systems have average hit rates greater than 90 percent.

For write to memory operations most cache systems use a *posted-write-through* method. If the cache controller determines that the addressed word is present in the cache, the controller will write the new word to the cache with no wait states and signal the 80386 that the write is complete. The controller will then write the data word to main memory. This write to the main memory is transparent to the main processor unless the main memory is still involved in a previous write operation.

To keep track of which main memory locations are currently present in the SRAM cache, the cache controller uses a *cache directory*. For the Intel 82385 cache controller shown in Fig. 11.11, the cache directory RAM is contained in the controller. Each location in the cache is represented by an entry in the directory. The exact format for the directory entry depends on the particular cache scheme used. The three basic cache schemes are direct-mapped, two-way set associative, and fully associative. We don't have time here to do a detailed discussion of these three caching schemes, but we will give you an introduction to each so you will understand the terms if you see them in a computer magazine article or advertisement. We discuss cache systems further in Chapter 15.

A DIRECT MAPPED CACHE

Figure 11.12a shows a block diagram of how a direct-mapped 32 Kbyte cache can be implemented in an 80386



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

detects and in some cases corrects errors in the data read out from DRAMs. There are several ways to do this, depending on the amount of detection and correction needed.

The simplest method for detecting an error is with a parity bit. This is the method used in the IBM PC circuit shown in Fig. 11.2. Note in this circuit that the DRAM memory bank is 9 bits wide. Eight of these bits are the data byte being stored, and the ninth bit is a parity bit which is used to detect errors in the stored data. A 74LS280 parity generator/checker circuit generates a parity bit for each byte and stores it in the ninth location as each byte is written to memory. When the 9 bits are read out, the overall parity is checked by the parity generator checker circuit. If the parity is not correct, an error signal is sent to the NMI logic to interrupt the processor. When you first turn on the power to an IBM PC or warm boot it by pressing the Ctrl, Alt, and Del keys at the same time, one of the self-tests that it performs is to write byte patterns

to all of the RAM locations and check if the byte read back and the parity of that byte are correct. If any error is found, an error message is displayed on the screen so you don't try to load and run programs in defective RAM.

ERROR DETECTING AND CORRECTING CIRCUITS

One difficulty with a simple parity check is that two errors in a data word may cancel each other. A second problem with the simple parity method is that it does not tell you which bit in a word is wrong so that you can correct the error. More complex error detecting/ correcting codes (ECCs), often called *Hamming codes* (after the man who did some of the original work in this area), permit you to detect multiple-bit errors in a word and to correct at least one bit error.

Figure 11.15a shows in block diagram form how a T1 74AS632 error detecting and correcting (EDAC) device can be connected in the data path between a 32-bit microprocessor and 16-Mbyte DRAM main memory.

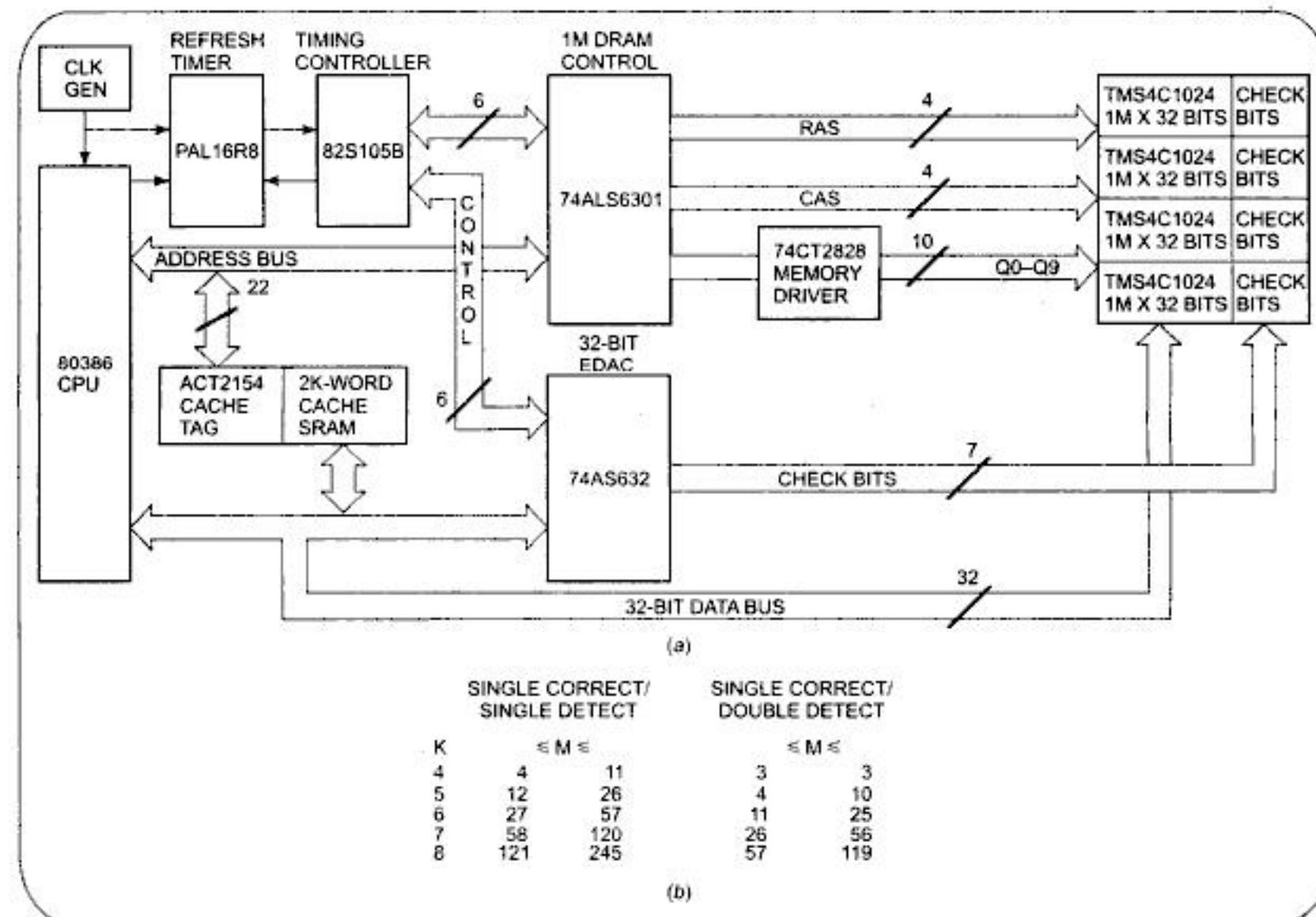


Fig. 11.15 (a) Block diagram showing how error detecting and correcting circuitry is connected in a large DRAM system. (Texas Instruments Inc.) (b) Hamming-code data bits and encoding bits and number of encoding bits required for desired degree of detection/correction.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The number of digits you can store for the exponent of a number determines the range of magnitudes of numbers you can store in your computer or calculator. The sign of the exponent indicates whether the magnitude of the number is greater than 1 or less than 1. The sign of the significand or mantissa indicates whether the number itself is positive or negative. Now let's see how you represent real numbers in binary form so the 8087 can digest them.

First let's look at the short-real format shown in Fig. 11.16. This format, which uses 32 bits to represent a number, is sometimes referred to as *single-precision* representation. In this format 23 bits are used to represent the magnitude of the number, 8 bits are used to represent the magnitude of the exponent, and 1 bit is used to indicate whether the number is negative or positive. The magnitude of the number is normalized so that there is only a single 1 to the left of the binary point. The 1 to the left of the binary point is not actually present in the representation; it is simply assumed to be there. This leaves more bits for representing the magnitude of the number. You can think of the binary point as being between the bits numbered 22 and 23. The exponent for this format is put in an *offset form*, which means that an offset of 127 (7FH) is added to the 2's complement value of the exponent. This is done so that the magnitude of two numbers can be compared without having to do arithmetic on the exponents first. The sign bit is 0 for positive numbers and 1 for negative numbers. To help make this clear to you, we will show you how to convert a decimal number to this format.

We chose the number 178.625 for this example because the fractional part converts exactly, and therefore we don't have to cope with rounding at this point. The first step is to convert the decimal number to binary, which gives 10110010.101, as shown in Fig. 11.17. Next normalize the binary number so that only a single 1 is to the left of the binary point, and represent the number of bit positions you had to move the binary point as an exponent, as shown in Fig. 11.17. The result at this point is 1.0110010101E7. If you now add the bias of 127 (7FH) to the exponent of 7, you get the biased exponent value of 86H that you need for the short-real representation. The final line in Fig. 11.17 shows the complete short-real result. For the significand you put in the binary bits to the right of the binary point. Remember, the 1 to the left of the binary point is assumed. The biased exponent value of 86H or 10000110 binary is put in as bits 23 through 30. Finally, since the number is positive, a 0 is put in bit 31 as the sign bit. The complete result is then 01000011001100101010000000000000 or 4332A000H, which is lengthy but not difficult to produce.

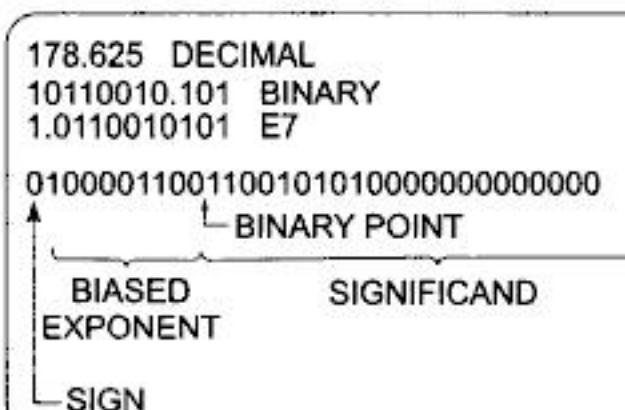


Fig. 11.17 Converting a decimal number to short-real format.

The long-real format shown in Fig. 11.16 uses 64 bits to represent each number. This format is often referred to as *double-precision* representation. This format is basically the same as that of the short-real, except that it allows greater range and accuracy because more bits are used for each number. For long-real, 52 bits are used to represent the magnitude of the number. Again, the number is normalized so that only a single 1 is to the left of the binary point. You can think of the binary point as being between the bits numbered 51 and 52. The 1 to the left of the binary point is not actually put in as one of the 64 bits. For this format, 11 bits are used for the exponent, so the offset added-to each exponent value is 1023 decimal or 3FFH. The most significant bit is the sign bit. Our example number of 178.625 is represented in this long-real or double-precision format as 4066540000000000H. Note in Fig. 11.16 the range of numbers that can be represented with this format. This range should be large enough for most of the problems you want to solve with an 8087.

The final format in Fig. 11.16 to discuss is the temporary-real format, which uses 80 bits to represent each number. This is the format that all numbers are converted to by the 8087 as it reads them in, and it is the format in which the 8087 works with numbers internally. The large number of bits used in this format reduces rounding errors in long chain calculations. To understand what this means, think of multiplying 1234×4567 in a machine that can store only the upper 4 digits of the result. The actual result of 5,635,678 will be truncated to 5,635,000. If you then divide this by 1234 to get back to the original 4567, you instead get 4566 because of the limited precision of the intermediate number.

As you can see in Fig. 11.16, the temporary-real format has a sign bit, 15 bits for a biased exponent, and 64 bits for the significand. The offset or bias added to the exponent here is 16,383 decimal or 3FFFH. A major difference in the significand for this format from that for



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

| | | |
|---|----------------|--------------|
| Type 1: Stack top and stack element | | |
| 8087 10011011 11011 d 00 11000(i) | | |
| Emulator 11001101 00011 d 00 11000(i) | | |
| Type 2: Stack top and memory operand. | | |
| 8087 10011011 11011 m 00 mod 000 r/m | | |
| Emulator 11001101 00011 m 00 mod 000 r/m | | |
| m = 0 for short real operand; 1 for long real operand | | |
| Type 3: Pop stack | | |
| 8087 10011011 11011110 11000(i) | | |
| Emulator 11001101 00011110 11000(i) | | |
| 8087 Timing (clocks) | TYPICAL | RANGE |
| slack element and stack top | 85 | 70-100 |
| stack element, stack top + pop | 90 | 75-105 |
| short real memory and stack top | 105 + EA | 90-120 + EA |
| long real memory and stack top | 110 + EA | 95-125 + EA |

Fig. 11.21 8087 FADD coding templates. (Intel Corporation)

First let's look at the coding for the FADD instruction with no specified operands. This instruction, remember, will add the contents of ST to the contents of ST(1), put the results in ST(1), and then pop the stack so that the result is at ST. The first byte of the instruction code, 10011011, is the code for the 8086 WAIT instruction. As we explain in detail later, this instruction code is put here to make the 8086 and 8087 wait until the 8087 has completed this instruction before starting the next one. The second byte shown is actually the first byte of the 8087 FADD instruction. The 5 most significant bits, 11011, identify this as an 8087 instruction. The lower 3 bits of the first code byte and the middle 3 bits of the second code byte are the opcode for the particular 8087 instruction. The bit labeled d at the start of these 6 bits is a 0 if the destination for an FADD ST(N),ST(N)-type instruction is ST. The d bit is a 1 if the destination stack element is one other than ST, as it is for the FADD instruction with no specified operands. For the FADD instruction with no specified operands, these 6 bits will be 100 000. The two most significant (MOD) bits in the second code byte are 1's because this form of the FADD instruction does not read a number from memory. The least significant 3 bits of the second instruction byte, represented by an i in the template, indicate which stack element other than ST is specified in the instruction. Since the simple FADD instruction uses ST(1) as a destination, 001 will be put in these bits. Putting all of this together for the FADD instruction with no specified source or

destination gives 10011011 11011100 11000001 binary or 9BH DCH C1H as the code bytes.

For a little more practice with this, see if you can code the 8087 instruction FADD ST,ST(2). Most of the coding for this instruction is the same as that for the previous instruction. For this one, however, the d bit is a 0 because ST is the specified destination. Also, the R/M bits are 010, because the other register involved in the addition is ST(2). The answer is 9BD8C2H. Now let's try an example which uses memory as the source of an operand for FADD.

For an FADD instruction such as FADD CORRECTION_FACTOR, which brings in one operand from memory and adds it to ST, the memory address can be specified in any of the 24 ways shown in Fig. 3.8. For the memory reference form of the FADD instruction, the MOD and R/M bits in the second code byte are used to specify the desired addressing mode. FADD CORRECTION_FACTOR represents direct addressing, so the MOD bits will be 00 and the R/M bits will be 110, as shown in Fig. 3.8. Two additional code bytes will be used to put in the direct address, low byte first. Since we are not using any of the other stack elements other than ST for this instruction, we "don't need the d bit to specify the other stack element. Instead, as shown in Fig. 11.21, this bit is labeled m. A 0 in this bit is used to specify a short-real, and a 1 in this bit is used for a long-real. Assuming CORRECTION_FACTOR is declared as a long-real, the code bytes for our FADD CORRECTION_FACTOR instruction will then be 10011011 11011100 00000110 followed by the 2 bytes of the direct address.

Now that you have an overview of how 8087 instructions are written and coded, we briefly discuss each of the 8087 instructions.

8087 Instruction Descriptions

The 8087 instruction mnemonics all begin with the letter F, which stands for floating point and distinguishes the 8087 instructions from 8086 instructions. We have found that if we mentally remove the F as we read the mnemonic, it makes it easier to connect the mnemonic and the operation performed by the instruction. Here we briefly describe the operation of each of the 8087 instructions so that you can use some of them to write simple programs. As you read through these instructions the first time, don't try to absorb them all, or you probably won't remember any of them. Concentrate first on the instructions you need to get operands from memory into the 8087, simple arithmetic instructions, and the instructions you need to get results copied back from the 8087 to memory where you can use them. Then work your way through the example program in the next section.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

PROCEDURE ANDGATE;
  CONST  TPLH = 15;
        TPHL = 10;
  VAR IN1, IN2, IN3 : INTEGER;
        DELAY, OUT : INTEGER;
  BEGIN
    IF (IN1 = 1)AND(IN2 = 1)AND(IN3 = 1)
      THEN BEGIN
        DELAY := TPLH
        OUT := 1;
      END
    ELSE BEGIN
        DELAY := TPHL
        OUT := 0
      END
    END;

```

This model is very primitive, but it should give you the idea. The constants represent the characteristics of the specific device being simulated (TPLH and TPHL). The variables represent the input logic levels (IN1–IN3), the output logic level (OUT), and the time between a change on the input and the corresponding change on the output (DELAY). Some simulators refer to these characteristics as *properties*. The schematic symbol is really part of the model for a device, so when you draw a schematic with a schematic capture program, you are actually creating a design file which contains the logical and timing characteristics of each device as well as the schematic symbols and connections.

When you set up the simulator to do a simulation run, you specify the signals you want applied to the inputs at a particular time, just as you connect signal generators to the inputs of a physical circuit. The simulator uses the model to determine the effects that the specified input signals will have on the output and schedules the output to change appropriately after the delay time for that device. As you can see, the model for the 3-input AND gate device tells the simulator program that if the input signals become all 1's, the output should be scheduled to change to a 1 after 15 ns. If the inputs change to a case where they are not all 1's, the output should be scheduled to change to a 0 after 10 ns.

The smallest increment of time used by a simulator is called its *time step*. You can think of the time step as the time resolution of the simulator. For simulating TTL and CMOS circuits, simulators usually use a time step of 1 ns or 0.1 ns because the delay times for these devices are a few ns. An important point here is that the 0.1-ns time step is simulator time, not real time. The simulator may

take 20 minutes to determine the effects that some input signal changes produce on the outputs of a complex circuit. The physical circuit would respond to the same input changes in a real time of just a few nanoseconds. The simulator essentially exercises the circuit "in slow motion" and generates an output which *represents*, or "simulates," the real-time operation of the circuit.

Now that you have an overview of how a simulator uses models, we need to talk briefly about some of the commonly used types of models. Three of these types are:

- Gate-level models
- Behavioral models
- Hardware models

As you may remember from a basic logic course, any digital circuit can be implemented with just basic gates. We didn't bother to show you, but even a complex device such as an 8086 or 80386 microprocessor can be modeled at the basic gate level for simulation. The difficulty with using gate-level models for complex devices is that simulation using these models requires a very long time. The reason for this is that the simulator must evaluate the effects of each signal change on all the intermediate circuit points (*nodes*) in the device.

If the complex device is a standard part, we usually know that all the internal circuitry works correctly, so we don't need to resimulate at the gate level of detail. To speed up the simulation of circuits containing complex devices, we often use *behavioral models*. Behavioral models simply describe the effects that input signals will have on the output signals and the signal delays between inputs and outputs. A behavioral model of a D flip-flop, for example, will indicate that 20 ns after a positive clock edge, the logic level on the D input will be transferred to the Q output if neither the Preset nor the Clear input is asserted. Behavioral models also contain properties such as setup times, hold times, and minimum pulse widths so the simulator can check for violations of these times by the signals propagating through the circuit. Sophisticated behavioral models such as the "SmartModels" from Logic Automation Inc. give detailed error messages to pinpoint a timing problem instead of making you work your way through a logic-analyzer-type display to find the problem.

For simulating microprocessors there are two types of behavioral models available. One type is called a *hardware verification model*. This type model is essentially a "black box" which will, for example, produce the correctly timed address and control bus signals for a memory-read cycle when given the proper *processor control language (PCL)*



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

21. Using the example program in Fig. 11.24 as a guide, write an 8087 program which computes the volume of a sphere. The formula is $V = \frac{4}{3}\pi R^3$.
22. a. When a coprocessor and a standard processor are connected together in a system such as that in Fig. 11.23, why are the S2-S0 status lines, the QS1-QS0 lines, the address, and the data lines of the two devices connected directly together?
b. Where does the 8087 coprocessor in Fig. 11.23 get its instructions from?
c. How does the main processor distinguish its instructions from those for the 8087 as it fetches instructions from memory?
d. Describe how the 8087 and 8088 work together to load a long-real data item from memory to the 8087 ST.
e. How does the 8087 in Fig. 11.23 signal the 8088 that it needs to use the buses?
f. How can you prevent the 8088 in Fig. 11.23 from going on with its next instruction before the 8087 has completed an instruction? What hardware connection in Fig. 11.23 is part of this mechanism?
23. a. Describe how a schematic is drawn using a schematic capture program.
b. What are the major advantages of the schematic capture approach over the traditional drafting approach?
24. a. What is meant by the term *software bread-board*?
b. Describe the major advantages of simulation over hardware prototyping.
c. What information does the simulation model for a device contain?
d. Briefly describe the steps involved in simulating a microcomputer such as the one in Fig. 11.25.
e. What information does simulation give you about a circuit such as the one in Fig. 11.25?
25. Briefly describe the sequence of steps in the electronic design automation method of designing, debugging, and producing an electronic product such as a microcomputer.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

C data types, sizes, and ranges

| type | subtype | size(bits) | range |
|-------------------------------|----------------|------------|---------------------------------|
| char | unsigned char | 8 | 0 → 255 |
| | char | 8 | -128 → +127 |
| enum | | 16 | -32,768 → +32,767 |
| int | | | |
| | unsigned short | 16 | 0 → 65,535 |
| | short | 16 | -32,768 → +32,767 |
| | unsigned int | 16 | 0 → 65,535 |
| | int | 16 | -32,768 → +32,767 |
| | unsigned long | 32 | 0 → 4,294,967,295 |
| | long | 32 | -2,147,483,648 → +2,147,483,647 |
| float | | | |
| | float | 32 | 3.4E-38 → 3.4E+38 |
| | double | 64 | 1.7E-308 → 1.07E+308 |
| | long double | 80 | 3.4E-4932 → 1.1E+4932 |
| pointer | | | |
| | near | 16 | -32,768 → +32,767 |
| | far or huge | 32 | -2,147,483,648 → +2,147,483,647 |
| (on 80386 and 80486 machines) | | | |

Fig. 12.5 C data types and sizes.

represent only the offset of a code or data word in a segment, and you use a far or huge (32-bit) pointer if you need to represent both the segment base and the offset of a code or data word. The *enumerated* data type shown as enum in Fig. 12.5 is a user-defined type which can have integer values. You probably won't use this type in your first programs.

Declaring and Initializing Simple Variables in C

CHAR VARIABLES

In your 8086 assembly language programs you declared and initialized variables with DB, DW, and DD statements. The example program in Fig. 12.1a showed you a few examples of how you declare and initialize simple variables and arrays in a C program. In this and the following sections we show how to declare and initialize variables of all the different C types. To start, Fig. 12.6 shows some

examples of how you declare and initialize char-type variables.

The first five variable declarations in Fig. 12.6 are all *extern*, which means that they are outside of any function. Variables declared outside any function are "global," so they can be accessed by any function in a program. If you declare a variable within a function, the variable is by default *automatic*, which means that it is "local" and can be accessed only within that function. For example, the declaration "char command[15];" in Fig. 12.6 is in function main, so the variable command is automatic and can be accessed only within main. Later, when we show you how to declare and use functions, we will discuss in more detail how you decide whether to make a variable *extern* or *automatic*. The general rule is to declare variables inside of main unless they *need* to be accessible to other program modules. This avoids a conflict if a module written by some other programmer has a different variable

```
/*Examples of declaring and initializing char type variables*/

char key;                                /* declare variable key but don't initialize */
char yes = 's';                            /* declare and initialize in one statement */
char bell = '\x07';                         /*initialize with ASCII bell code */
char message[20];                          /* message declared as array for 20 char - not initialized */
char err_mess[] = "Turn off the power";     /*array of char initialized with specified string */

void main()
{
    char command[15];                      /* automatic array for 15 char */
                                            /* accessible only within main */
}
```

Fig. 12.6 Declaring and initializing char variables in C.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

/* C PROGRAM F12-10A.C - COMPUTE THE SELLING PRICE OF 10 ITEMS */
#include <stdio.h>
#define PROFIT 15
#define MAX_PRICES 10

void main()
{
    int cost[] = { 20,28,15,26,19,27,16,29,39,42 };
    int prices[10];
    int index, *cpntr = cost, *ppntr = prices;

    /* array index method */
    for (index=0; index <MAX_PRICES; index++)
    {
        prices[index] = cost[index] + PROFIT;
        printf("cost = %d, price = %d, \n", cost[index],
               prices[index]);
    }

    /* pointer method */
    for (index =0; index<MAX_PRICES; index++)
    {
        *ppntr = *cpntr + PROFIT;
        printf("cost = %d, price = %d,\n", *cpntr,*ppntr);
        cpntr++; ppntr++;
    }

    /* pointer arithmetic method */
    for (index =0; index<10; index++)
    {
        *(prices +index) = *(cost +index) + PROFIT;
        printf("cost = %d, price = %d, \n",
               *(cost + index), *(prices + index));
    }
}

;8086 PROGRAM F12-10B.ASM
;ABSTRACT : Assembly language program to add profit to costs using pointers
;           ; Program adds a profit factor to each element in a
;           ; COST array and puts the result in an PRICES array.

PROFIT EQU      15H      ; profit = 15 cents
ARRAYS SEGMENT
COST    DB   20H,28H,15H,26H,19H,27H,16H,29H,39H,42H
PRICES  DB  10 DUP(0)
CPNTR   DW  OFFSET COST
PPNTR   DW  OFFSET PRICES
ARRAYS ENDS

CODE    SEGMENT
ASSUME CS:CODE, DS:ARRAYS
START: MOV  AX, ARRAYS ; Initialize data segment
       MOV  DS, AX      ; register
       MOV  CX, 0010    ; Initialize counter

DO_NEXT: MOV  BX, CPNTR ; Load cost pointer in BX
         MOV  SI, PPNTR ; Load price pointer in SI
         MOV  AL, [BX]   ; Get element pointed to by CPNTR
         ADD  AL, PROFIT ; Add the profit to value read
         DAA             ; Decimal adjust result
         MOV  [SI], AL   ; Store result at location pointed
                           ; to by PPNTR in PRICES
         INC  CPNTR     ; Increment pointers
         INC  PPNTR
         LOOP DO_NEXT    ; If not last element, do again
CODE   ENDS
END  START

```

Fig. 12.10 (a) C program which uses pointers to compute selling prices, (b) 8086 assembly language equivalent of program in 12.10a.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

/* C PROGRAM F12-15.C */
#include <stdio.h>
void main()
{
    char ch;
    printf("Game over.\n");

    prompt: printf("Enter y to play another game, n to quit.\n");
    if ((ch = getch()) == 'N' || ch == 'n')
    {
        printf("Goodbye.\n");
        exit();
    }
    else if (ch == 'Y' || ch == 'y')
    {
        printf("Here we go again.\n");
        /* goto start; */
    }
    else
    {
        ch = getchar(); /* clear buffer */
        goto prompt;
    }
}

```

Fig. 12.15 Nested if-else example.

variable used to make the decision. After each case line you write the statement(s) you want executed if the variable has that value. If, for example, the value of variable is equal to value 1, the statements after case value 1: will be executed. The break statement at the end of this block of statements will cause execution to skip over the rest of the choices in the structure. If you leave out the break statement, the actions for the next case after the selected case will be executed. The optional default directive at the end of the switch structure allows you to specify the action(s) you want taken if the value of variable does not match any of the specified values.

Figure 12.16 shows how you might use the switch statement to implement a “command recognizer” in one of your programs. This example is modeled after the commands available at the highest menu level in the Borland TC development environment we discussed earlier in the chapter. To get to the main menu in TC, you press the F10 key. To select the desired submenu you then press the key which corresponds to the first letter in the name of the submenu. The choices are F, E, R, C, P, O, D, and B. Each of these options brings up a lower-level menu or carries out a command.

In the program in Fig. 12.16 we use our new friend getch() to read a character from the keyboard. We then use a switch structure to evaluate the character and decide what action to take. To simplify the basic structure of this example, we call a function to implement each of the

desired actions. Actually, for this example we show the function calls as comments, because we did not want to declare and define all these functions. When execution returns from the called function, the break statement at the end of that line will cause execution to skip to the next statement after the switch structure. If the key pressed by the user does not match any of the choices, the default: edit_window(); statement at the end of the block sends execution back to the edit operation. You can have only one value in each case evaluation, so if you want the program to accept lower- or uppercase letters, you have to put case lines in for each. The line case ‘F’: followed by the line case ‘f’: file(); break;, for example, will call the file function if the user enters either a lower- or uppercase f. A more versatile alternative is to write a small function which converts all entered characters to lowercase before entering the switch structure. We leave this for you to do as an exercise at the end of the chapter.

THE WHILE AND DO-WHILE IMPLEMENTATIONS

In Chapter 3 we showed you how the WHILE-DO and the REPEAT-UNTIL structures are used to loop through a series of statements. In C these two structures are called the while and the do-while, respectively. The major difference between the two structures is when the exit test is done. For comparison, Fig. 12.17 shows how the two are implemented in C.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

can see, in this example the dot matrix for each character is 5 dots wide and 7 dots high. Other common dot-matrix sizes for character displays are 7 by 9, and 7 by 12, and 9 by 14.

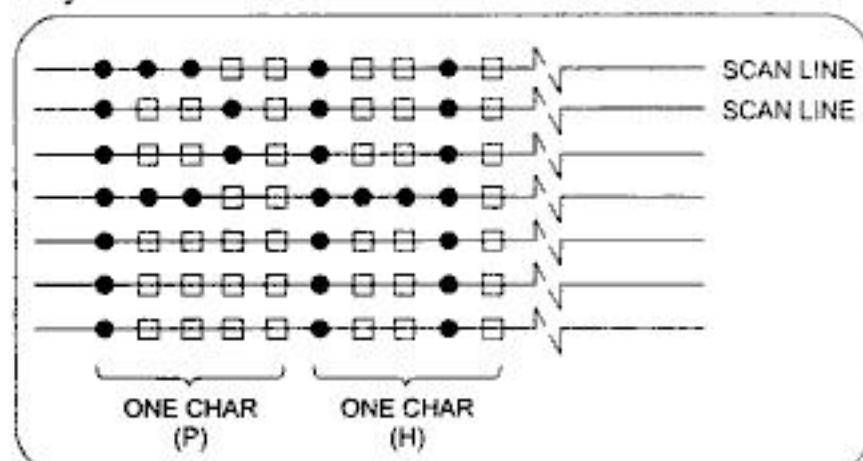


Fig. 13.5 Producing a character display on a CRT screen with dots.

Figure 13.6 shows a block diagram of the circuitry needed to keep the pattern of dots for a page of text displayed on the screen of a CRT monitor. For this example assume that the display has 25 rows of characters with 80 characters per row.

The ASCII codes for the characters to be displayed on the screen are stored in a RAM. This RAM is often referred to as the *frame buffer* or the *display refresh RAM*. The RAM must contain at least one byte location for each character to be displayed. A display size of 25 rows with 80 characters in each row then requires 25×80 or about

2 Kbytes RAM. In an actual circuit this RAM is set up so that the microprocessor can access it to change the stored characters, or the display refresh circuitry can access it to keep the display refreshed on the screen.

The dot patterns for each scan line of each character to be displayed are stored in a ROM called a *character generator ROM*. Figure 13.7 shows the matrix for a typical character-generator ROM. This ROM uses a 7 by 9 matrix for the actual character, but the total dot space for each character is a 9 by 14 dot matrix. The extra dots are included to leave space between characters and between rows of characters. Also, the extra space allows lowercase letters to be dropped in the matrix so that descenders are shown correctly. Each dot row in Fig. 13.7 represents the pattern of dots for a horizontal scan line of the character.

To start the display in the upper left corner, the character counter and the character row counter outputs are all 0's so the ASCII code for the first character in the display RAM is addressed. The ASCII code from the addressed location is output by the RAM to the address inputs of the character-generator ROM. These inputs essentially tell the character generator which character is to be displayed.

To keep track of which line in a character row is currently being swept out, we use a scan line counter. For our example here each row of characters has 14 dot rows or scan lines, so the scan line counter is a modulo- 14 counter. The outputs of this counter are connected to four additional address inputs on the character generator ROM.

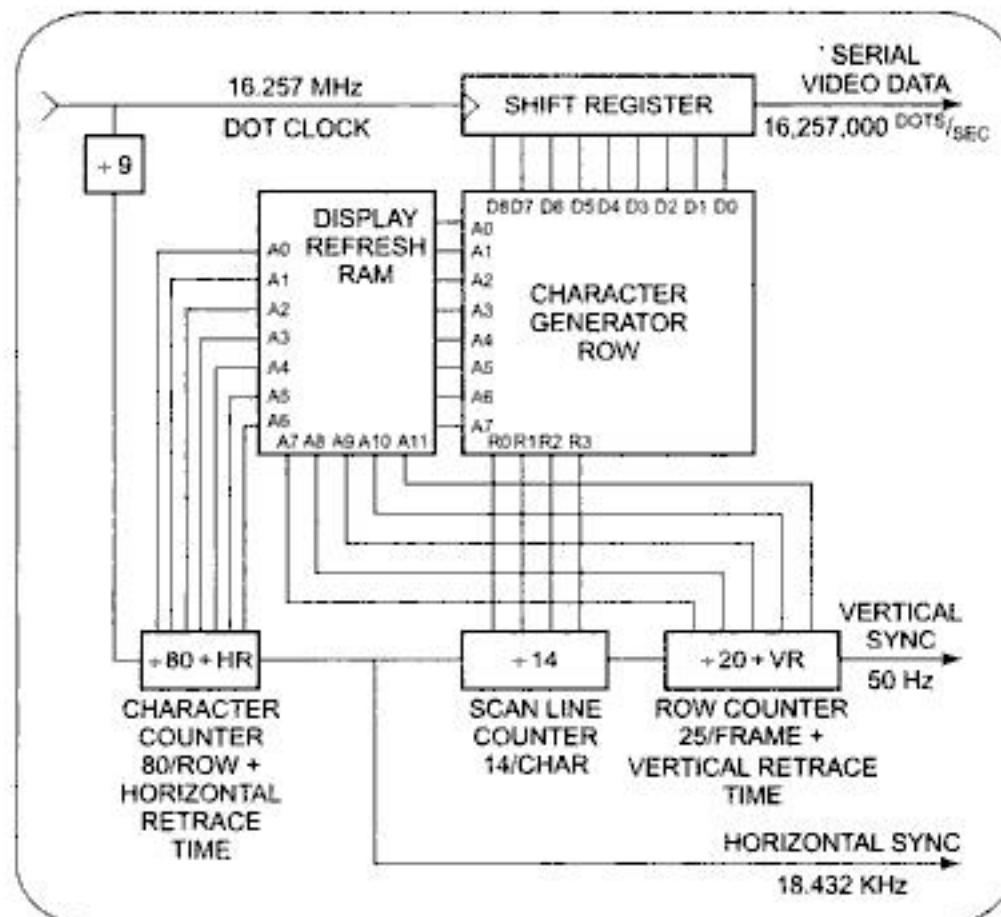


Fig. 13.6 Block diagram of circuitry to produce dot-matrix character display on CRT.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The data rate for direct formant synthesis is only about 1 Kbit/s, but the parameters must be determined with complex equipment. It is not easy to develop a custom vocabulary for a specific application. A phoneme approach solves this problem and requires a still lower data rate at the expense of lower speech quality.

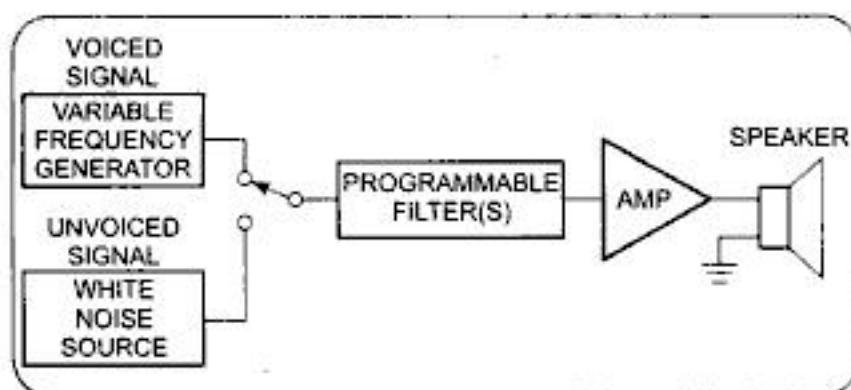


Fig. 13.45 Electronic model of human vocal tract.

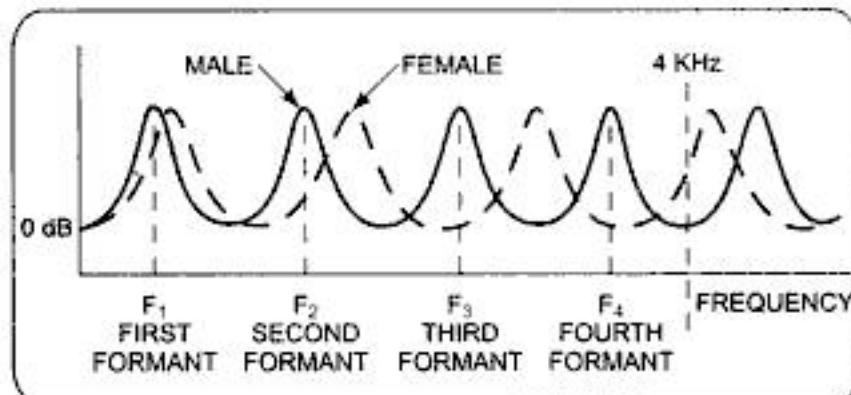


Fig. 13.46 Filter responses for formant speech synthesizer.

Phonemes are fragments of words. An example of a phoneme speech synthesizer is the Artich Technologies 263A, which can be interfaced with a microcomputer port or in some cases interfaced directly to microprocessor buses. Words are produced by sending a series of 6-bit phoneme codes to the device. Five internal 8-bit registers also allow you to control parameters such as speech rate, pitch, amplitude, articulation rate, and vocal tract filter response. Inside the 263A the 6-bit phoneme code is used to control the characteristics of some formant filters, as described in the previous paragraph. Since only one code is sent out for a relatively long period of speech, the required bit rate is only about 70 bits/s. However, the long period between codes gives less control over waveform details and, therefore, sound quality. A phoneme synthesizer has a mechanical sound. One big advantage of phoneme synthesizers is that you can make up any message you want by simply putting together a sequence of phoneme codes.

DIRECT DIGITIZATION SPEECH SYNTHESIS

Direct digitization speech synthesis produces the highest-quality speech, because it is essentially just a playback of digitally recorded speech. To start, the word you want the computer to speak is spoken clearly into a microphone. The output voltage from the microphone is amplified and applied to the input of perhaps a 12-bit A/D converter. One approach at this point might be to simply store the A/D samples for the word in a ROM and read the values out to a D/A converter when you want the computer to speak the word. The difficulty with this approach is that if the samples are taken often enough to produce good speech quality, a lot of memory is required to store the samples for a word. To reduce the amount of memory required, several speech-compression algorithms are used. These algorithms are too complex to discuss here, but the basic principles involve storing repeated waveforms only once, taking advantage of symmetry in waveforms, and not storing values for silent periods. Even with compression, however, direct digital speech requires considerable memory and a bit rate as high as 64 Kbits/s. To further reduce the memory required for direct digital speech, some systems use differential or *delta* modulation. In these systems only a 3-bit or 4-bit code, representing how much a sample has changed from the last sample, is stored in memory instead of storing the complete 12-bit value. This system works well for audio signals, since they change slowly.

The OKI Semiconductor MSM6388 device contains much of the circuitry needed to digitize and reproduce speech using adaptive-differential pulse code modulation (ADPCM). This device contains a microphone preamplifier, A/D converter, D/A converter, and some low-pass filters. The digital values produced by the A/D converter are stored in external memory. About 260 s of speech can be stored in 4 Mbits of external memory.

Another example of a direct digital synthesis system is the National Semiconductor *Digitalalker*. For further information, consult the data sheets for these devices.

Speech Recognition

Speech recognition is considerably more difficult than speech synthesis. The process is similar to trying to recognize human faces with a computer vision system. With most speech-recognition systems the first step is to train the system or, in other words, produce templates for each of the words that the system needs to recognize and store these templates in memory. To produce a template for a word, the intended user speaks the word several times into a microphone connected to the system. The system then determines several parameters or *features* for each



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

from several different topics in order for each part of the subject to really make sense. To make this approach easier, we will first give an overview of how all the pieces fit together and then describe the details of each piece later in specific sections. A problem with this subject is that it contains a great many terms and acronyms. To help you absorb all of these, you may want to make a glossary of terms as you work your way through the chapter.

Within a microcomputer data is transferred in parallel, because that is the fastest way to do it. For transferring data over long distances, however, parallel data transmission requires too many wires. Therefore, data to be sent long distances is usually converted from parallel form to serial form so that it can be sent on a single wire or pair of wires. Serial data received from a distant source is converted to parallel form so that it can easily be transferred on the microcomputer buses. Three terms often encountered in literature on serial data systems are *simplex*, *half-duplex*, and *full-duplex*. A simplex data line can transmit data only in one direction. An earthquake sensor sending data back from Mount St. Helens or a commercial radio station are examples of simplex transmission. Half-duplex transmission means that communication can take place in either direction between two systems, but can only occur in one direction at a time. An example of half-duplex transmission is a two-way radio system, where one user always listens while the other talks because the receiver circuitry is turned off during transmit. The term full-duplex means that each system can send and receive data at the same time. A normal phone conversation is an example of a full-duplex operation.

Serial data can be sent *synchronously* or *asynchronously*. For synchronous transmission, data is sent in blocks at a constant rate. The start and end of a block are identified with specific bytes or bit patterns. In a later section of the chapter we discuss synchronous data transmission in detail. For asynchronous transmission, each data character has a bit which identifies its start and 1 or 2 bits which identify its end. Since each character is individually identified, characters can be sent at any time

(asynchronously), in the same way that a person types on a keyboard.

Figure 14.1 shows the bit format often used for transmitting asynchronous serial data. When no data is being sent, the signal line is in a constant high or *marking* state. The beginning of a data character is indicated by the line going low for 1 bit time. This bit is called a *start bit*. The data bits are then sent out on the line one after the other. Note that the least significant bit is sent out first. Depending on the system, the data word may consist of 5, 6, 7, or 8 bits. Following the data bits is a parity bit, which—as we explained in Chapter 11—is used to check for errors in received data. Some systems do not insert or look for a parity bit. After the data bits and the parity bit, the signal line is returned high for at least 1 bit time to identify the end of the character. This always-high bit is referred to as a *stop bit*. Some older systems use 2 stop bits. For future reference note that the efficiency of this format is low, because 10 or 11 bit times are required to transmit a 7-bit data word such as an ASCII character.

The term *baud rate* is used to indicate the rate at which serial data is being transferred. Baud rate is defined as $1/(t)$ (the time between signal transitions). If the signal is changing every 3.33 ms, for example, the baud rate is $1/(3.33 \text{ ms})$, or 300 Bd. There is an almost unavoidable, but incorrect, tendency to refer to this as 300 bits/s. In some cases, the two do correspond, but in other cases 2 or more actual data bits are encoded in one signal transition, so data bits per second and baud do not correspond. Common baud rates are 300, 600, 1200, 2400, 4800, 9600, and 19,200.

To interface a microcomputer with serial data lines, the data must be converted to and from serial form. A parallel-in-serial-out shift register and a serial-in-parallel-out shift register can be used to do this. Also needed for some cases of serial data transfer is hand-shaking circuitry to make sure that a transmitter does not send data faster than it can be read in by the receiving system. There are available several programmable LSI devices which contain most of the circuitry needed for serial communication. A device

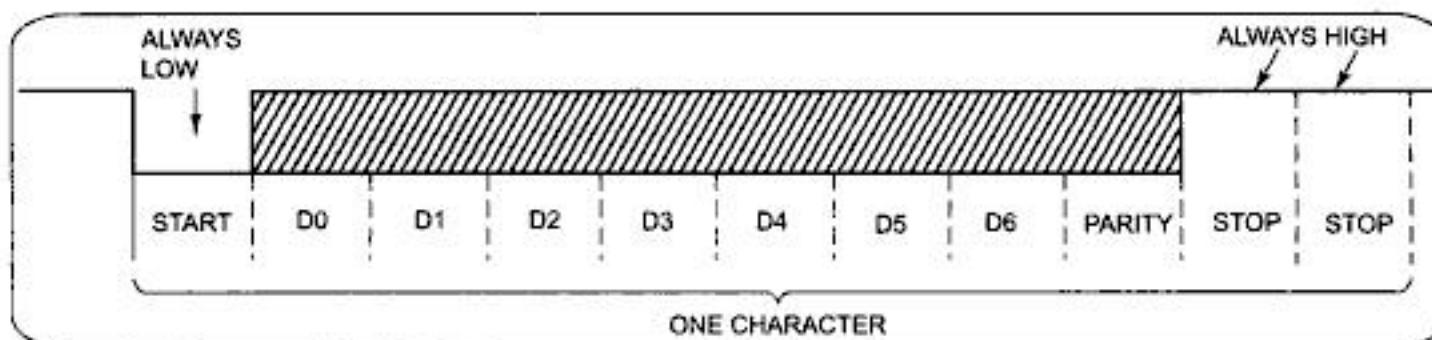


Fig. 14.1 Bit format used for sending asynchronous serial data.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

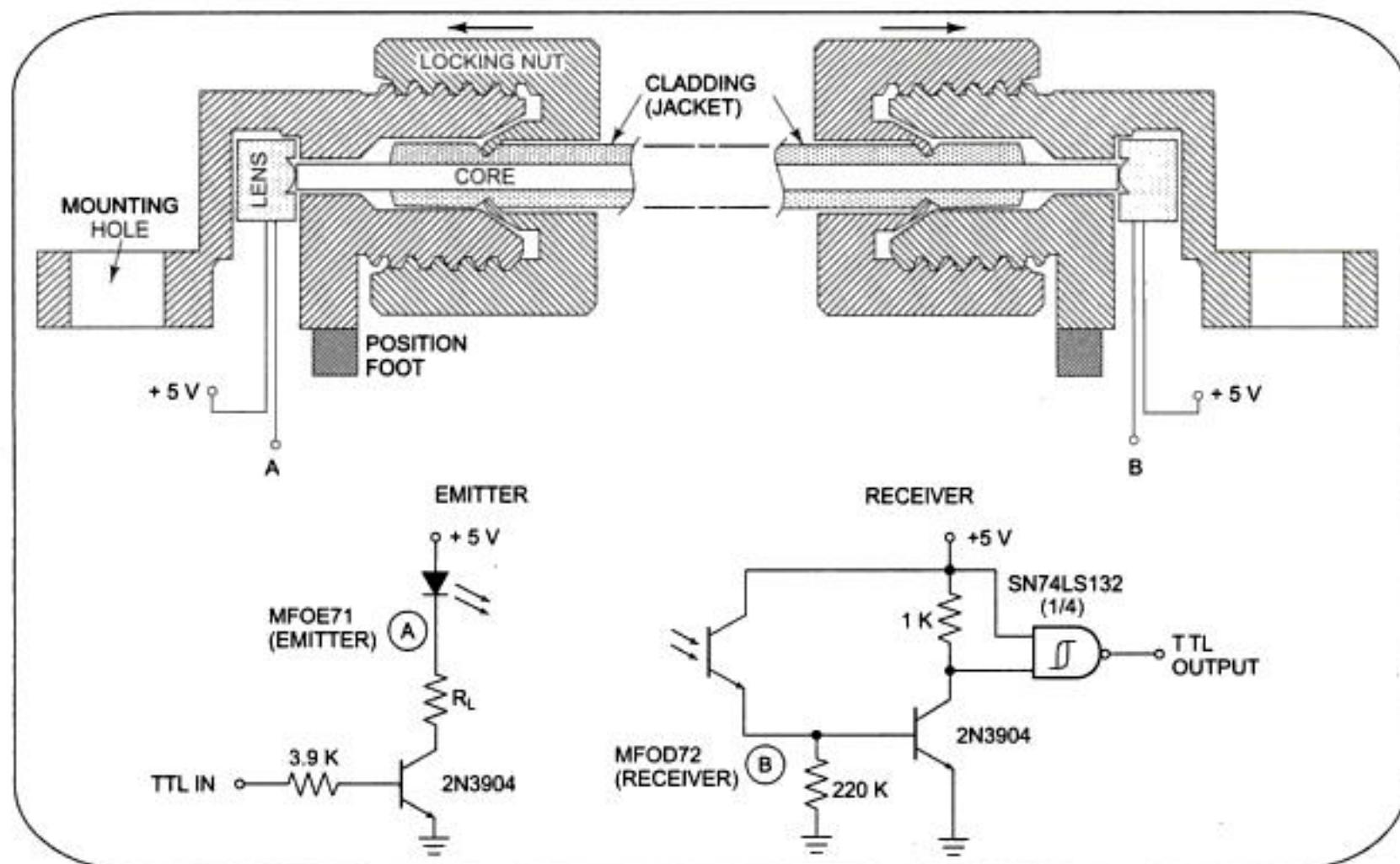


Fig. 14.21 Components of a simple fiber-optic data link.

light source here is a simple infrared LED. Higher-performance systems use an *infrared injection laser diode* (ILD) or some other laser driven by a high-speed, high-current driver. Digital data is sent over the fiber by turning the light beam on for a 1 and off for a 0.

NOTE: If you are working on a fiber-optic system you should never look directly into the end of the fiber to see if the light source is working, because the light beam from some laser diodes is powerful enough to cause permanent eye damage. Use a light meter, or point the cable at a nonreflective surface to see if the light source is working.

To convert the light signal back into an electrical signal at the receiving end, Darlington photodetectors such as the MFOD73 shown in Fig. 14.21, PIN FET devices, or avalanche photodiodes (APDs) are used. APDs are more sensitive and operate at higher frequencies, but the circuitry for them is more complex. A Schmitt trigger is usually used on the output of the detector to “square up” the output pulses.

The fiber used in a cable is made of special plastic or glass. Fiber diameters used range from 2 to 1000 μm . Larger-diameter plastic fibers are used for short-distance, low-speed transmission, and small-diameter glass fibers are used for high-speed applications such as long-distance

telephone transmission lines. As shown in Fig. 14.22e, the fiber-optic cable consists of three parts. The optical-fiber core is surrounded by a *cladding* material which is also transparent to light. An outer sheath protects the cladding and prevents external light from entering.

Now that you have an overview of an optical-fiber link, let's take a look at how the light actually propagates through the fiber and the trade-offs with different fibers.

THE OPTICS OF FIBERS

The path of a beam of light going from a material with one optical density to a material of different optical density depends on the angle at which the beam hits the boundary between the two materials. Figure 14.22 shows the path that will be taken by beams of light at various angles going from an optically dense material such as glass to a less dense material such as a vacuum or air. If the beam hits the boundary at the right angle, it will go straight through, as shown in Fig. 14.22a. When a beam hits the boundary at a small angle away from the perpendicular or *normal*, it will be bent away from the normal when it goes from the more dense to the less dense, as shown in Fig. 14.22b. A light beam going in the other direction would follow the same path. A quantity called the *index of refraction* is used to describe the amount that the light beam will be bent. Using the angle identifications shown



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

;8086 PROGRAM F14-27B.ASM
_TEXT SEGMENT BYTE PUBLIC 'CODE'
DGROUP GROUP _DATA
ASSUME CS:_TEXT,DS:DGROUP,SS:DGROUP
_TEXT ENDS

_DATA SEGMENT WORD PUBLIC 'DATA'
QUEUE DB 1000 DUP(0) ; Declare ring buffer
HEAD_POINTER DW 0 ; Pointer to read location in buffer
TAIL_POINTER DW 0 ; Pointer to write location in buffer
TIME_OUT_MESS DB 'TRANSMIT TIMEOUT - CHECK HARDWARE',ODH, OAH
EXTRB _NUMCHAR:WORD
_DATA ENDS

_TEXT SEGMENT BYTE PUBLIC 'CODE'
PUBLIC _INIT
PUBLIC _SERIAL_IN
PUBLIC _CHK_N_DISPLAY

_INIT PROC NEAR
    ; Unmask 8259A IR4
    IN AL, 21H ; Read 8259A IMR
    AND AL, 0ECH ; Unmask IR4
    OUT 21H, AL

; Initialize 8250 UART baud rate,etc.
    MOV AH, 00 ; Initialize COM1
    MOV DX, 0000 ; Point at COM1
    MOV AL, 11000111B ; 4800 Bd, No parity, 2 stop, 8-bit
    INT 14H ; via BIOS INT 14H

; Enable 8250 RxRDY interrupt
    MOV DX, 03F8H ; Point at 8250 line control port
    IN AL, DX ; Read in line control word
    AND AL, 7FH ; Set DLAB = 0
    OUT DX, AL ; Send line control word back out
    MOV AL, 01H ; Value to enable RxRDY interrupt
    MOV DX, 03F9H ; Point at interrupt enable register
    OUT DX, AL ; Enable RxRDY interrupt
    MOV AL, 08H ; Assert 8250 OUT2, RTS, DTR byte
    MOV DX, 03FCH ; Point at modem control reg in 8250
    OUT DX, AL ; Send to 8250
    RET

_INIT ENDP

_SERIAL_IN PROC FAR
    STI ; Interrupts back on for clock, etc.
    PUSH AX
    PUSH BX
    PUSH DX
    PUSH DI
    PUSH DS
    MOV AX, DGROUP ; Load current DS register value
    MOV DS, AX
    MOV DX, 03F8H ; Receiver buffer address for 8250
    IN AL, DX ; Read character
    MOV DI, TAIL_POINTER ; Get current tail pointer value
    INC DI ; Point to next storage location
    CMP DI, 1000 ; Compare with max to see if time
                  ; to wrap around
    JNE FULCHK ; No, go check if queue full
    MOV DI, 00 ; Yes, set DI for wraparound to start

```

Fig. 14.27 (Continued)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Binary Synchronous Communication Protocol—BISYNC

BISYNC is referred to as a *byte-controlled protocol* (BCP), because specified ASCII or EBCDIC characters (bytes) are used to indicate the start of a message and to handshake between the transmitter and the receiver. Incidentally, even in a full-duplex system, BISYNC protocol only allows data transfer in one direction at a time.

Figure 14.30 shows the general message format for BISYNC. For our first cycle through this we will assume that the transmitter has received a message from the receiver that it is ready to receive a transmission. If no message is being sent, the line is an “idle” condition with a continuous high on the line. To indicate the start of a message, the transmitting system sends two or more previously agreed upon sync characters. For example, a sync character might be the ASCII 16H. As we said before, the receiver uses these sync characters to synchronize its clock with that of the transmitter. A header may then be sent if desired. The header contents are usually defined for a specific system and may include information about the type, priority, and destination of the message that follows. The start of the header is indicated with a special character called *start-of-header* (SOH), which in ASCII is represented by 01H.

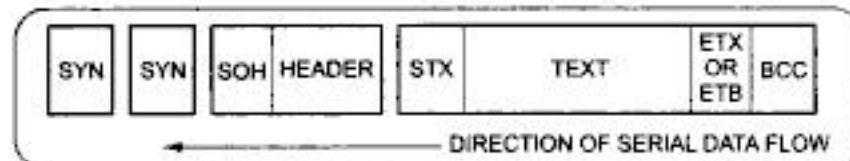


Fig. 14.30 General message format for binary synchronous communication (BISYNC).

After the header, if present, the beginning of the text portion of the message is indicated by another special character called *start-of-text* (STX), which in ASCII is represented by 02H. To indicate the end of the text portion of the message, an *end-of-text* (ETX) character or an *end-of-block* (ETB) character is sent. The text portion may contain 128 or 256 characters (different systems use different-size blocks of text). Immediately following the ETX, character 1 or 2 block check characters (BCC) are sent. For systems using ASCII, the BCC is a single byte which represents complex parity information computed for the text of the message. For systems using EBCDIC, a 16-bit *cyclic redundancy check* is performed on the text part of the message and the 16-bit result sent as 2 BCCs. The point of these BCCs is that the receiving system can recompute the value for them from the received data and compare the results with the BCCs sent from the transmitter. If the BCCs are not equal, the receiver can

send a message to the transmitter, telling it to send the message again. Now let's look at how messages are used for data transfer handshaking between the transmitter and the receiver.

To start let's assume that we have a remote “smart” terminal connected to a computer with a half-duplex connection. Further, let's assume that the computer is in the receive mode. Now, when the program in the terminal determines that it has a block of data to send to the computer, it first sends a message with the text containing only the single character ENQ (ASCII 05H), which stands for *enquiry*. The terminal then switches to receive mode to await the reply from the computer. The computer reads the ENQ message, and, if it is not ready to receive data, it sends back a text message containing the single character for *negative acknowledge*, NAK (ASCII 15H). If the receiver is ready, it sends a message containing the *affirmative acknowledge*, ACK, character (ASCII 06H). In either case, the computer then switches to receive mode to await the next message from the terminal. If the terminal received a NAK, it may give up, or it may wait a while and try again. If the terminal received an ACK, it will send a message containing a block of text and ending with a BCC character(s). After sending the message, the terminal switches to receive mode and awaits a reply from the computer as to whether the message was received correctly. The computer meanwhile computes the BCC for the received block of data and compares it with the BCC sent with the message. If the two BCCs are not equal, the computer sends a NAK message to the terminal. This tells the terminal to send the message again, because it was not received correctly. If the two BCCs are equal, then the computer sends an ACK message to the terminal, which tells it to send the next message or block of text. In a system where multiple blocks of data are being transferred, an ACK 0 message is usually sent for one block, an ACK 1 message sent for the next, and an ACK 0 again sent for the next. The alternating ACK messages are a further help in error checking. In either case, after the message is sent the computer switches to receive mode to await a response from the terminal.

A variation of BISYNC commonly used to transfer binary files in the PC environment and between Unix systems and PCs is called *XMODEM*. An XMODEM block consists of a SOH character, a block number, 128 bytes of data (padded if necessary to fill the block), and an 8-bit checksum. A transmission starts with the receiver sending a NAK character to the sender. The sender then sends a block of data. If the data is received correctly, the receiver sends back an ACK and the sender



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

example, the "interceptor" part of the resident network software determines whether the file is located on the workstation hard disk or on the server hard disk. If the file is on the workstation hard disk, the interceptor simply passes the request on to DOS and the access proceeds through DOS and BIOS as it would in stand-alone operation. If the file is on the server, the request goes to the request translator to get assembled in the proper packet format for transmission over Ethernet. The output from the request translator then goes to the network communications driver, which sends it to the server over the network. A standard set of network communication drivers written by Microsoft is called *NETBIOS*. Other companies which write network control programs either license NETBIOS from Microsoft or write their own compatible network drivers.

The server reads the requested file, converts it to packets, and sends it to the workstation. The appropriate driver reads the packets into the workstation.

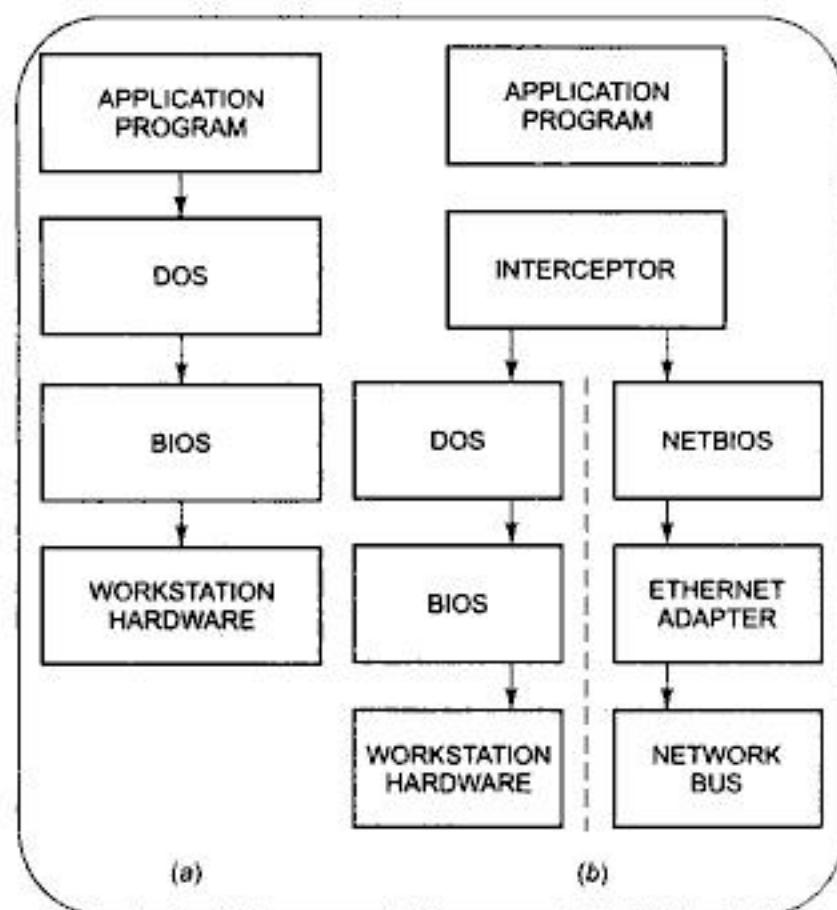


Fig. 14.39 Software hierarchy on a workstation. (a) Nonnetworked. (b) Networked.

The reply translator part of the software converts the packets to DOS file format and loads the file in memory so the application can work with it.

The network software that resides in the server is a complete operating system in itself. To install Netware 386 on the server, you first format a small partition on the server hard disk in DOS format. You then load DOS in

this partition so the system is bootable from it and load some of the basic Netware files here so you can install the rest of Netware. After you boot the system, the installation consists of working your way through a relatively simple sequence of steps outlined in the installation procedure.

Once installed, the network operating system is set up so that only the system administrator can access and change its operation. The system administrator sets up user accounts, assigns passwords, and sets the access rights for files. Application program files are usually specified as read-only so that users cannot accidentally or maliciously modify them. For files that are intended to be accessed and written to by any one of several users, Netware 386 has a default feature called *file locking*, which prevents one user from accessing the file until a previous user has finished with it. In this case the critical region is the file, and file locking provides a way to protect it.

Netware 386 uses several techniques to speed up disk access. First, it formats its partition of the hard disk differently from the way DOS formats it to make for more efficient access to the parts of a file. Second, it uses disk caches such as those we described in the last chapter to hold large blocks of data read from files. This reduces the number of read operations required to access a large file. Finally, Netware uses "elevator seeking" to reduce the amount the heads move to read a requested set of files for users. Just as an elevator moves sequentially from floor to floor instead of moving from floor to floor as requested, the head is moved to access files in the sequence they are located on disk tracks rather than strictly in the sequence they were requested.

In addition to allowing users to store and access files, the network operating system has many other useful features. It sets up a queue of files waiting to be printed or plotted so that users can just enter a print command and go on with their work. Most networks have electronic mail, which allows the system administrator to communicate with all users and users to communicate with each other. Most electronic mail systems are set up so you can define a group of users and direct mail messages to just that group.

For the reasons that we have discussed, it is likely that in the near future almost all computers will in some way be networked with other computers through telephone lines or direct connections. In the last section of the chapter we discuss a different type of computer network which is often used in a factory environment to build a "smart" test system.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The 80286, 80386 and 80486 Microprocessors

For most of the examples up to this point in the book, we have used the 8086/8088 microprocessor, because it is the simplest member of this family of Intel processors and is therefore a good starting point. Now it is time to look at the evolutionary offspring of the 8086. To give you an overview, here are a few brief notes about the members of this family.

The 80186 processor is basically an 8086 with an on-chip priority-interrupt controller, programmable timer, DMA controller, and address decoding circuitry. This processor has been used mostly in industrial control applications.

The 80286, another 16-bit enhancement of the 8086, was introduced at the same time as the 80186. Instead of the integrated peripherals of the 80186, it has virtual memory-management circuitry, protection circuitry, and a 16-Mbyte addressing capability. The 80286 was the first family member designed specifically for use as the CPU in a multiuser microcomputer.

The 80386, the next evolutionary step in the family, is a 32-bit processor with a 32-bit address bus. The 32-bit ALU allows the 80386 to process data faster, and the 32-bit address bus allows the 80386 to address up to 4 Gbytes of memory. Another enhancement of the 80386 is that segments can be as large as 4Gbytes instead of only 64-Kbytes. The memory-management circuitry and protection circuitry in the 80386 are improved over that in the 80286, so the 80386 is much more versatile as the CPU in a multiuser system.

The latest current member of this family, the 80486, has the same CPU as the 80386, so it has the same

addressing capability, memory-management, and protection features as the 80386. The main new features included in the 80486 are a built-in 8-Kbyte code/data cache and a 32-bit floating-point-unit, similar to the 8087 we discussed in Chapter 11.

As perhaps you can see from the preceding brief discussions, the 80286, 80386, and 80486 were designed for use as the CPU in a multitasking microcomputer system. To help you better understand the operation and design rationale of these processors, we start the chapter with a discussion of the problems that must be solved in writing a multitasking/multiuser operating system. We then discuss the 80286, 80386, and 80486 microprocessors in detail and explain how the features designed in these processors help solve the problems involved in implementing a multitasking operating system. After that we discuss how you develop real mode and protected mode programs for systems using these devices.

Finally in the chapter we discuss some of the directions in which microcomputer evolution seems to be heading. Included in this section are discussions of RISC processors, parallel processors, artificial intelligence, "fuzzy" logic, and neural networks.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. *Describe the difference between time-slice scheduling and preemptive priority-based scheduling.*
2. *Define the terms blocked, task queue, deadlock, deadly embrace, critical region, semaphore, kernel, memory-management unit, and virtual memory.*



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

when it is time to switch from one task to another is called the *scheduler*, *dispatcher*, or *supervisor*. The most common method of scheduling task switches is the *time-slice* method which we discussed previously. In a simple round-robin implementation of this approach, the CPU executes one task for perhaps 20 ms and then switches to the next task. After all tasks have had their turn, execution returns to the first. In the program in Fig. 10.35 we showed you how a programmable timer, priority-interrupt controller, and interrupt-service procedure can be used to implement this type of scheduling. The UNIX operating system and the OS/2 operating system use a more complex time-slice scheduling approach to implement multitasking. The advantage of the time-slice approach in a multiuser system is that all users are serviced at approximately equal time intervals. As more users are added, however, each user gets serviced less often, so each user's program takes longer to execute. This is referred to as *system degradation*. For industrial control operating systems, this variable scheduling is often not appropriate, so a different scheduling method is used.

PREEMPTIVE PRIORITY-BASED SCHEDULING

In a system which uses *preemptive priority-based scheduling*, an executing low-priority task can be interrupted by a higher-priority task. When the high-priority task finishes executing, execution returns to the low-priority task. This approach is well suited to some control applications because it allows the most important tasks to be done first. Priority-interrupt controllers such as the 8259A are often used to set up and manage the task service requests. The Intel RMX 86 operating system uses priority-based scheduling.

Preserving the Environment

The registers, data, pointers, etc., used by an executing task are referred to as its *environment*, *state*, or *context*. When a task switch occurs, the environment of the interrupted task must be saved so that the task can be restarted properly when it receives another time slice. The usual way of preserving the environment is to keep it in a special memory segment or on a stack. Some operating systems keep a separate stack for each task. In either case, when a task switch occurs the operating system saves the environment of the interrupted task and a pointer to the saved environment. When it is time to switch back to that task, the operating system uses the pointer to access the environment it saved. This process is commonly called "context switching."

A less obvious point in a multitasking system is that any global procedures have to be reentrant. This is

necessary so that if one task is executing a procedure and its time slice ends, other tasks can use the procedure, and the procedure will still complete correctly when execution returns to the first task. Refer to Fig. 5.20 if you need a refresher on reentrancy.

Accessing Resources

Another problem encountered in a multitasking system is assuring that tasks have orderly access to resources such as printers, disk drives, etc. As one example of this, suppose that a user at a terminal needs to read a file from a hard disk and print it on the system printer. Obviously the file cannot be read in from the disk and printed in one of the 20-ms time slices allotted to that user, so several provisions must be made to gain access to the resources and hang on to them long enough to get the job done properly. A flag or *semaphore* in memory is used to indicate whether the disk drive is in use by another task or not. Likewise, another semaphore is used to indicate whether the printer is in use. If a task cannot access a resource because it is busy, the task is said to be *blocked*. Now, rather than making the user type in a print command over and over until the disk drive or the printer is available, most operating systems of this type set up queues of tasks waiting for each resource. When one task finishes with a resource, it resets the semaphore for that resource. The next task in the queue can set the semaphore to indicate the resource is busy and then use the resource.

The Need for Protection

An interesting problem can occur in a multitasking operating system when two or more users attempt to read and change the contents of a memory location at the same time. As an example, suppose that an airline ticket-reservation system is operating on a time-slice basis. Now, further suppose that just before the end of his or her time slice, one user examines the memory location which represents a seat on a plane and finds the seat empty. Another user on the system can then, in his or her time slice, examine the same memory location, find it empty, mark it full, and print out a reservation confirmation on the CRT. When execution returns to the first user, his or her program has already checked the seat during its previous time slice, so it marks the seat full, and prints out a reservation confirmation on the CRT. The two people assigned to the same seat may make nasty remarks about computers unless this problem is solved.

The section of a program where the value of a variable is being examined and changed must be protected from access by other tasks until the operation is complete. The



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

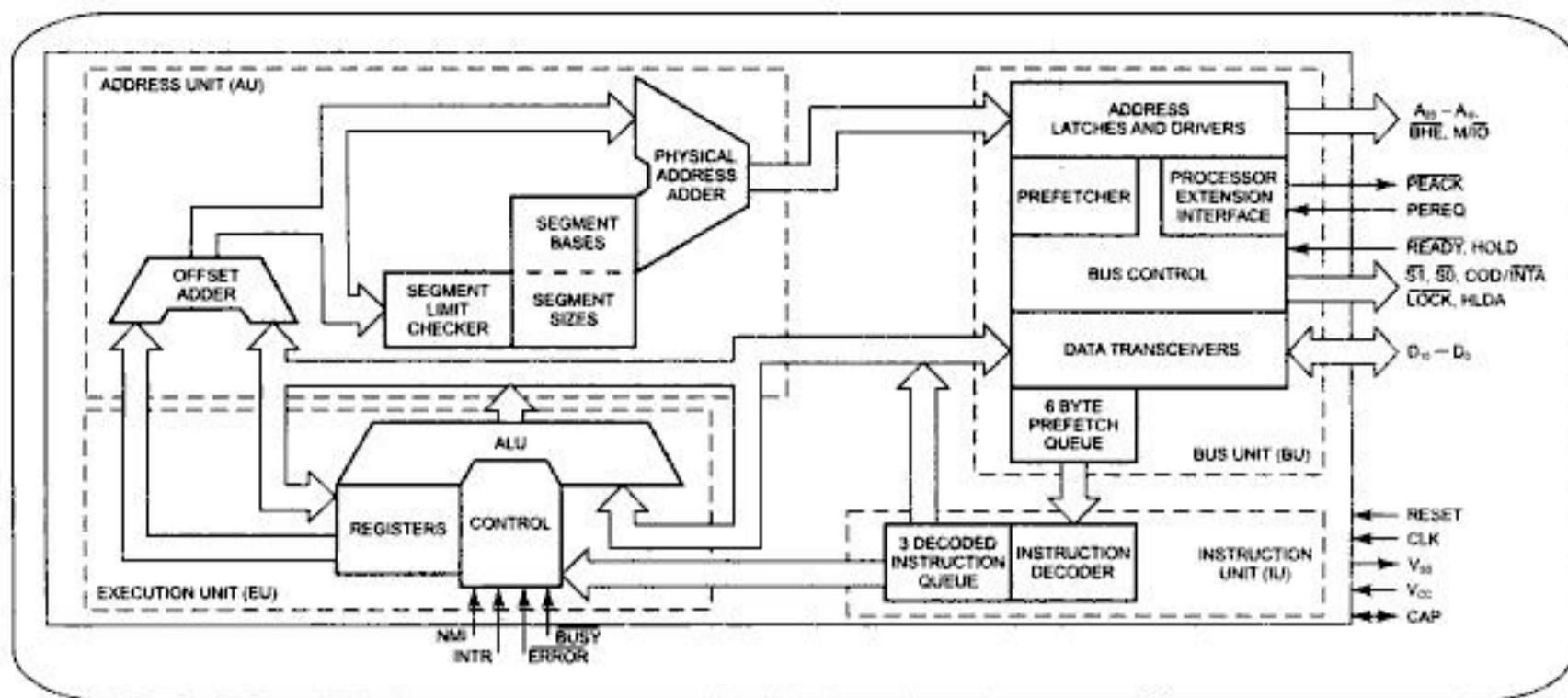


Fig. 15.9 80286 internal block diagram. (Intel Corporation)

SS, and ES registers are used to hold the base addresses for the segments currently in use. The maximum physical address space in this mode is 1 Mbyte, just as it is for the 8086.

If an 80286 is operating in its *protected virtual address mode* (protected mode), the address unit functions as a complete MMU. In this address mode the 80286 uses all 24 address lines to access up to 16 Mbytes of physical memory. In protected mode it also provides up to a gigabyte of virtual memory using the descriptor table scheme shown in Fig. 15.8.

Figure 15.10 shows the 68-pin package that is usually used for an 80286, and Fig. 15.11 shows how an 8086 is connected with some other components to form a simple system. Keep these figures handy as we work our way around the major pins of the 80286. Many of the signals of the 80286 should be familiar to you from our discussion of the 8086 signals in Chapter 7.

The 80286 has a 16-bit data bus and a 24-bit nonmultiplexed address bus. The 24-bit address bus allows the processor to access 16 Mbytes of physical memory when operating in protected mode. Memory hardware for the 80286 is set up as an odd bank and an even bank, just as it is for the 8086. The even bank will be enabled when A₀ is low, and the odd bank will be enabled when BHE is low. To access an aligned word, both A₀ and BHE will be low. External buffers are used on both the address and the data bus.

From a control standpoint, the 80286 functions similarly to an 8086 operating in maximum mode. Status signals SO, S1, and M/IO are decoded by an external

82288 bus controller to produce the control bus, read, write, and Interrupt-acknowledge signals.

The HOLD, HLDA, INTR, INTA, (NMI), READY, and LOCK and RESET pins function basically the same as they do on an 8086. An external 82284 clock generator is used to produce a clock signal for the 80286 and to synchronize RESET and READY signals.

The final four signal pins we need to discuss here are used to interface with processor extensions (coprocessors) such as the 80287 math coprocessor. The *processor extension request* (PEREQ) input pin will be asserted by a coprocessor to tell the 80286 to perform a data transfer to or from memory for it. When the 80286 gets around to do the transfer, it asserts the *processor extension acknowledge* (PEACK) signal to the coprocessor to let it know the data transfer has started. Data transfers are done through the 80286 in this way so that the coprocessor uses the protection and virtual memory capability of the MMU in the 80286. The BUSY signal input on the 80286 functions the same as the TEST 1 input does on the 8086. When the 80286 executes a WAIT instruction, it will remain in a WAIT loop until it finds the BUSY signal from the coprocessor high. If a coprocessor finds some error during processing, it will assert the ERROR input of the 80286. This will cause the 80286 to automatically do a type 16H interrupt call. An interrupt-service procedure can be written to make the desired response to the error condition.

The machine cycle waveforms for the 80286 are very similar to those of the 8086 that we showed and discussed



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

| EISA | PC BUS | EISA |
|----------------------|-----------|--------|
| GND | GND | |
| +5V | RESET DTV | |
| +5V | +5V | |
| MFG SPEC | IRQ2 | |
| MFG SPEC | -5V | |
| (KEY) | DRQ2 | |
| MFG SPEC | -12V | |
| MFG SPEC | N/C | |
| MFG SPEC | +12V | |
| M-IO | GND | |
| LOCK- | SMEMW- | |
| RESERVED | SMEMR- | |
| GND | IOW- | |
| RESERVED | IOR- | |
| BE3 | DACK3- | |
| (KEY) | DRQ3 | |
| BE2- | DACK1- | |
| BE0- | DRQ1 | |
| GND | REFRESH- | |
| +5V | CLK | |
| LA29 | IRQ7 | |
| GND | IRQ6 | |
| LA26 | IRQ5 | |
| LA24 | IRQ4 | |
| (KEY) | IRQ3 | |
| LA16 | DACK2- | |
| LA14 | TC | |
| +5V | BALE | |
| +5V | +5V | |
| GND | OSC | |
| LA10 | GND | |
| EXTENSION FOR AT BUS | | |
| LA8 | MEM CS16- | LA7 |
| LA6 | I/O CS16- | GND |
| LA5 | IRQ10 | LA23 |
| +5V | IRQ11 | LA4 |
| LA2 | IRQ12 | LA22 |
| (KEY) | IRQ15 | LA21 |
| D16 | IRQ14 | LA20 |
| D18 | DACK0- | (KEY) |
| GND | DRQ0 | LA19 |
| D21 | DACK5- | LA18 |
| D23 | DRQ5 | D19 |
| D24 | DACK6- | LA17 |
| GND | DRQ6 | D20 |
| D27 | DACK7- | MEMR- |
| (KEY) | DRQ7 | D22 |
| D29 | +5V | MEMW- |
| +5V | MASTER- | GND |
| MACKn- | GND | D25 |
| | | D26 |
| | | D28 |
| | | D29 |
| | | D30 |
| | | D31 |
| | | MREQn- |

(b)

Fig. 15.18 (Continued) (b) Pin assignments, EISA bus.

enough to also contact the lower level of contacts. These lower contacts contain the additional EISA signals. Figure 15.18b shows how these added signal lines are interspersed between the ISA signals. Note the additional address (labeled LA), data, BE#, power, ground, and control pins. The pin positions labeled KEY represent thin slots cut in the PC board so that it aligns properly when inserted. The pins labeled MFG SPEC are used to output a code which identifies the type of board to aid in system configuration during bootup.

In a multiprocessor microcomputer system a processor that can take over the bus to transfer data is called a *bus master*. A board which can take over the bus for a DMA operation is called a *DMA slave*. The EISA bus supports up to six bus masters and 8 DMA slaves. The MACKn and MREQn lines on the EISA bus are used to arbitrate bus requests by multiple masters. These signals are not bussed. An individual trace runs from each of these pins to the arbitration logic on the motherboard. Then in these signal names represents the slot number in the system. When a master wants to use the buses, it asserts its MREQ line. If the buses are free and that master is the highest-priority master requesting use of the buses, the arbitration circuitry will assert the MACK signal connected to that master. The master will use the bus for its data transfer. DMA slaves issue requests through the DREQ lines on the bus and receive control from the arbitration circuitry through the DACK lines.

Another feature of the EISA bus is that its interrupts can be individually programmed as edge-triggered for compatibility with ISA boards or level-triggered so that they are less susceptible to noise spikes and they can be shared by several sources. EISA boards use level-triggered interrupts, which can be pulled low by any one of several sources. When the CPU detects an interrupt, it polls each board or device to determine the source of the interrupt.

To help implement an EISA bus in a system, Intel makes the 82358 Bus Controller, the 82357 Integrated System Peripheral, and the 82355 Bus Master Interface Controller. Consult the data sheets for these devices to get more detailed information about the operation of an EISA bus.

THE MICROCHANNEL ARCHITECTURE BUS

IBM's MicroChannel Architecture Bus contains the same types of signals and accomplishes the same functions as EISA, but the two are completely incompatible. MCA boards are smaller and use different edge connectors. Figure 15.19, shows the MCA bus connector types used in the IBM PS/2 Model 80.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

EXAMPLE:

SETC TooBig ; Set all bits in flag TooBig if
; Carry flag set

Shifts between words

SHLD—Shift specified number of bits left from one operand into another.

EXAMPLE:

SHLD EAX, EBX, 8 ; Shift upper 8 bits from EBX
; into lower 8 bits of EAX
; EBX unchanged

SHRD—Shift specified number of bits right from one operand into another.

EXAMPLE:

SHRD EAX, EBX, 8 ; Shift lower 8 bits from EBX
; into upper 8 bits of EAX.
; EBX unchanged

INSTRUCTION ENHANCEMENTS

Several of the 386 instructions have significant improvements over the 8086/80186/80286 versions. Here are a few notes about these improvements.

1. The 386 string instructions work with double-word operands as well as with word and byte operands. A "B" at the end of an instruction mnemonic specifies byte operands, a W specifies word operands, and a D specifies double-word operands. Examples are CMPSB, CMPSW, and CMPSD.
2. The destination for a 386 conditional jump can be anywhere in the segment containing the jump instruction. Conditional far jumps must still be done by changing the jump condition and using an unconditional far jump as we showed you for the 8086 in Chapter 4.
3. The LOOP instructions can use the CX register or the ECX register as a counter. If you want CX to be used, write the instructions as LOOPW, LOOPWE, and LOOPWNE. If you want the ECX register to be used as the counter, write the instructions as LOOPD, LOOPDE, and LOOPDNE.
4. PUSHFD pushes the 32-bit EFLAGS register and POPFD restores it.
5. PUSHAD pushes the 8 general-purpose 32-bit registers on the stack, and POPAD restores these registers except for the value of ESP which is ignored.
6. IRETID pops the double word EIP, a double word for CS, and the EFLAGS register off the stack. The high word of the value popped for CS is discarded.

7. The IMUL instruction can now perform signed multiplication on any general-purpose register and a memory location or another general-purpose register.

8. In addition to the protected-mode instructions inherited from the 80286, the 386 instructions used to move data to/from the control registers (CR0-CR3), the debug registers (DR0-DR7), and the test registers (TR0-TR7) can be executed only in protected mode at privilege level 0. These instructions are simple MOV register, register instructions.

386 Programming**INTRODUCTION**

The tools and techniques used to write a program for a 386 or a 486 depend very much on whether it is a system program or an application program and whether it utilizes protected mode or not. The tools and techniques are also determined by whether the program is going to execute in a graphical user-interface environment such as Microsoft's Windows 3.0 or OS/2. In this section we give you an introduction to writing 386 programs for five different programming environments.

386 PROGRAMS FOR MS DOS-BASED SYSTEMS

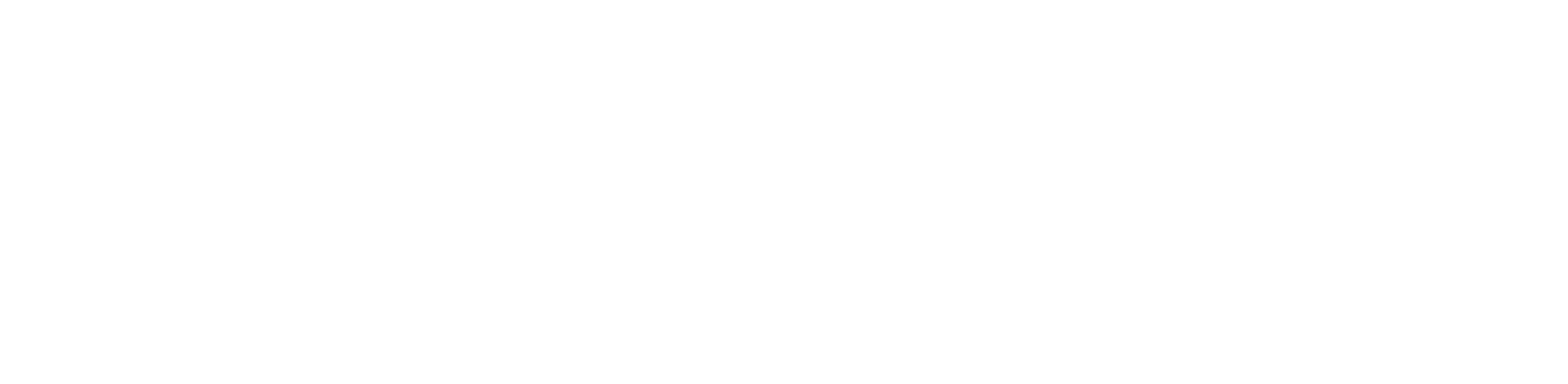
Current versions of DOS are designed to run on 8086/8088, real-mode 80286, or real-mode 386/486 systems. If you want a program to be able to run under DOS on any one of these systems, then you have to write it for the "weakest link" in the group, the 8086. The 8086 and C programming examples throughout this book were in fact compiled and run on a 386-based system. If you are writing a program that you are sure will only be run under DOS on a 386- or 486-based machine, you can write the program to take advantage of the 32-bit processing capability, addressing modes, and enhanced instructions of the 386. Assuming that you are using the MASM or TASM assembler, you tell the assembler to accept 386 instructions by putting the .386 directive at the top of your source program, as shown in Fig. 15.29a.

If you are using the simplified segment directives, make sure to put the .386 directive after the .MODEL directive, as shown in Fig. 15.29b. This order tells the assembler to create 16-bit segments which are compatible with real-mode operation. If you put the .386 directive before the .MODEL directive, the assembler will create 32-bit code and data segments which can be used only in protected mode.

Even though 386 real-mode programs are limited to 64-Kbyte segments and unless bank switching is implemented, to a 640-Kbyte address space, you can still



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



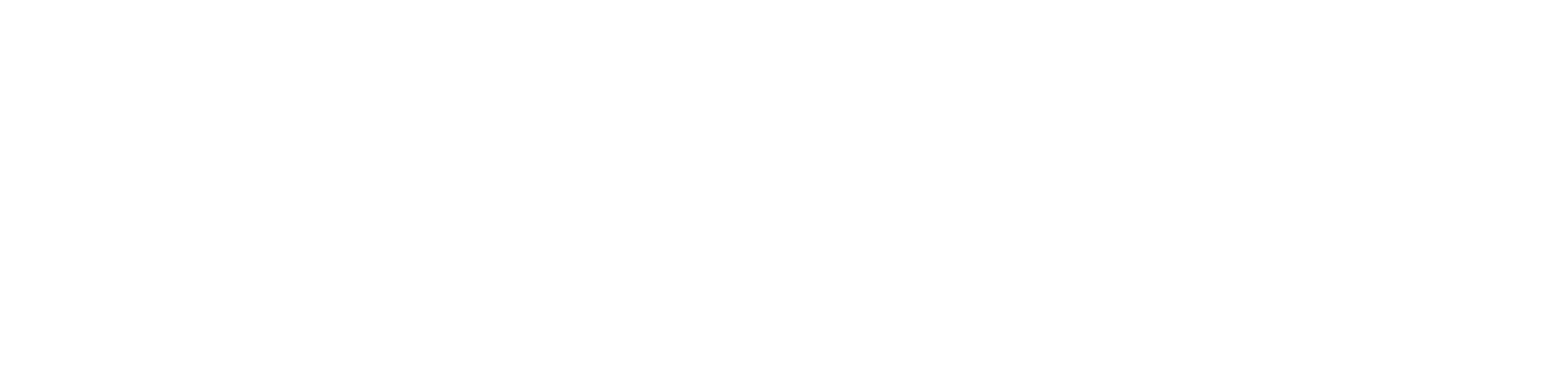
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Unlike most computer programs, which require complete information to make a decision, expert system programs are designed to make a best guess, based on the available data, just as a human expert would do. A medical diagnosis expert system, for example, will indicate the illness that most likely corresponds to a given set of symptoms and test data. To enable it to make a better guess, the system may suggest additional tests to perform.

One advantage of a system such as this is that it can make the knowledge of many experts readily available to a physician anywhere in the world via a modem connection. Another advantage is that the data base and set of rules can be easily updated as new research results and drugs become available. Other examples of expert system programs are those used to lay out PC boards and those used to lay out ICs.

NEURAL NETWORKS

Programs for some problems such as image recognition, speech recognition, weather forecasting, and three-dimensional modeling are not easily or accurately implemented on fixed-instruction-set computers such as 386/i486-based systems. For applications such as these, a new computer architecture, modeled after the human brain, shows considerable promise.

As you may remember from a general science class, the brain is composed of billions of neurons. The output of each neuron is connected to the inputs of several thousand other neurons by synapses. If the sum of the signals on the inputs of a neuron is greater than a certain threshold value, the neuron "fires" and sends a signal to other neurons. The simple op-amp circuit in Fig. 16.10a may help you see how a neuron works. Let's assume the output of the comparator is initially low. If the sum of the input signals to the adder produces an output voltage more negative than the comparator threshold voltage, the output of the comparator will go high. This is analogous to the neuron firing. The weight or relative influence of an input is determined by the value of the resistor on that input. Figure 16.10b shows a symbol commonly used to represent a neuron in neural network literature and Fig. 16.10c shows a simple mathematical model of a neuron.

As with the neurons in the human brain, the neurons in a neural network are connected to many other neurons. Fig. 16.10d shows a simple three-layer neural network. This network configuration is referred to as "feedforward," because none of the output signals are connected back to the inputs. In a "feedback" or "resonance" configured network, some intermediate or final output signals are connected back to network inputs. Researchers are currently experimenting with many

different network configurations to determine the one that works best for each type of application.

Neural network based computing can be implemented in several ways. One way is to use a dedicated processor for each neuron. The large number of neurons usually makes this impractical, and most applications don't need the speed capability. An alternative approach is to use a single processor and simulate neurons with lookup tables. The lookup table for each neuron contains the connections, input weight values, and output equation constants. Hecht-Nielsen Neurocomputers markets a PC/AT-compatible coprocessor board which uses this approach.

A neural network can also be implemented totally in software. NeuralWare, Inc. markets neural net simulation programs for both PC and Macintosh type computers. These packages can be used to learn about neural nets or develop actual applications which do not have to operate in real time. Another interesting neural network program is BrainMaker from California Scientific Software.

Neural network computers are not programmed in the way that digital computers are, they are trained. Instead of being programmed with a set of rules the way a classic expert system is, a neural network computer learns the desired behavior. The learning process may be supervised, unsupervised, or self-supervised.

In the supervised method a set of input conditions and the expected output conditions are applied to the network. The network learns by adjusting or "adapting" the weights of the interconnections until the output is correct. Another input-output set is then applied, and the network is allowed to learn this set. After a few sets the network will have learned or generalized its response so that it can give the correct response to most applied input data.

The scheme used to adapt the network is called the learning rule. As an example, one of the simplest learning rules that can be used is the Hebbian Learning Law. This law decrees that each time the input of a neuron contributes to the firing, its weight should be increased, and each time an input does not contribute, its weight should be decreased. This is somewhat analogous to a positive-negative reinforcement scheme often used in human behavior modification. In the case of the network the result is that these successive "nudges" adapt the network output to the desired result.

The major advantages of neural networks are these:

1. They do not need to be programmed; they can simply be taught the desired response. This eliminates most of the cost of programming.
2. They can improve their response by learning. A neural network designed to evaluate loan



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Bibliography

Because of the technical level of this book, the major sources of further information on the topics discussed are manufacturer's data books, application notes, and articles in current engineering periodicals. With the foundation you get from this book you should be able to comfortably read these materials. Listed below, by chapter, are some materials that will give you more details for many of the topics we discuss in the book. Following the chapter listings is a list of periodicals which we have found to be particularly helpful in keeping up with the latest advances in microcomputer evolution and applications.

Chapter 1

Hall, Douglas V., *Digital Circuits and Systems*, McGraw-Hill, Inc., New York, 1989.

Chapters 2-8

Mick, John, and Jim Brick; *Bit-Slice Microprocessor Design*, McGraw-Hill, Inc., New York, 1980.

A History of Microprocessor Development at Intel, Intel Article Reprint/AR-173, Intel Corporation, Santa Clara, Calif.

Microprocessors, Intel Corporation, Santa Clara, Calif., latest edition (Databook).

Turbo Assembler User's Guide, Borland International, Scotts Valley, Calif., 1988.

Macro Assembler User's Guide, Microsoft Corp., Bellingham, Wash., 1989.

Peripherals, Intel Corporation, Santa Clara, Calif., latest edition (Databook).

SDK-86 MCS-86 System Design Kit User's Guide, Intel Corporation, Santa Clara, Calif., 1981.

8086 System Design, Application Note, Intel Corporation Computer/AP-67, Intel Corporation, Santa Clara, Calif.

Chapters 9-10

Peripherals, Intel Corporation, Santa Clara, Calif., latest edition (Databook).

IBM PC TECHNICAL REFERENCE MANUAL, IBM Corp., Boca Raton, Fla., 1983.

Dorf, Richard C, *Robotics and Automated Manufacturing*, Reston Publishing Company, Reston, Va. 1983.

AMF Potter & Brumfield Catalog, Potter & Brumfield, Princeton, Ind., latest edition.

Optoelectronics Designer's Catalog, Hewlett-Packard, Palo Alto, Calif., latest edition.

Interfacing Liquid Crystal Displays in Digital Systems, Application Note AN-8, Beckman Instruments, Inc., Scottsdale, Ariz., latest edition.

Optoelectronics Device Data Book, DL118R1, Motorola Semiconductor Products Inc., Phoenix, Ariz., latest edition.

Sandhu, H. S., *Hands-On Introduction to ROBOTICS—The Manual for the XR-Series Robots*, Rhino Robots, Champaign, Ill., latest edition.

Slo-Syn DC Stepping Motors Catalog, DCM1078, Superior Electric Company, Bristol, Conn., latest edition.

Auslander, David M., and Paul Sagus, *Microprocessors for Measurement and Control*, Osborne/McGraw-Hill, Berkeley, Calif., 1981.

Allocca, John A., and Allen Stuart, *Transducers Theory and Applications*, Reston Publishing Company, Inc., Reston, Va., 1984.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Index

- 2, powers of, 1.2
2's complement, 1.7–1.9, 3.7
4-bit microprocessors, 2.9, 2.10
4N33 optical coupler, 8.25
5-minute rule, 4.4, 7.25, 7.39, 10.23, 14.23
7-segment display code, 1.5
7-segment light-emitting diodes (LEDs), 1.5
8-bit microprocessors, 2.10–2.11
10BaseT (thin Ethernet) networks, 14.43
16-bit microprocessors, 2.11–2.12
18-segment light-emitting diodes (LEDs), 9.32, 9.33, 9.44
32-bit microprocessors, 2.10
74LS14 inverter, 8.12, 8.13, 8.15, 8.25
74LS138 decoder, 7.27–7.30, 8.19, 8.20
74LS164 wait-state generator, 7.38, 7.13, 7.16, 7.28, 7.29
74LS181 ALU, 1.19–1.20
74LS244 drivers, 7.14, 7.37
74LS393 baud rate generator, 7.14, 7.23, 7.25
74S373 address latch, 7.13, 7.14, 7.24, 7.25, 7.37, 7.41
555 timer, 8.16, 8.17
741 operational amplifier, 10.1, 10.2
2142 SRAM, 7.14, 7.17, 7.20, 11.17
2316/2716 PROM, 13.32
2900 family of bit-slice processors, 2.10
3625 I/O decoder, 7.14, 7.21, 7.34–7.37
3625 off-board decoder, 7.37, 7.38, 7.48
3625 PROM decoder, 7.48
3625 RAM decoder, 7.24, 7.27, 7.33, 7.35
4004 microprocessor, 2.9
6502 microprocessor, 2.10
6800 microprocessor, 2.10
6801 microcontroller, 2.10
6809 microprocessor, 2.10
6845 CRT controller, 13.8–13.9
7445 decoder, 7.14, 7.21, 9.34
7447 decoder, 9.32–9.34, 9.34
8008 microprocessor, 2.9–2.10
8048 microcontroller, 2.10, 9.30–9.31
8051 microcontroller family, 2.10, 10.49
8080 microprocessor, 2.10
8085 microprocessor, 2.10
8086 assembly language
breakpoints in, 3.27
coding sheets for, 3.10
comments in, 2.16, 3.10, 3.12
converting algorithms to, 4.4
destinations in, 2.17, 2.18
fields in, 2.16
instructions for. *See* 8086 instructions
labels in, 3.10, 3.11, 3.12
mnemonics in, 2.16, 3.10, 3.11
opcodes (operation codes) for, 2.16, 3.10, 3.11, 3.13
operands in, 2.16, 3.10, 3.11
overview of, 2.16–2.20
program development tools for, 3.26–3.29
programs. *See* 8086 assembly language programs
sources in, 2.16–2.17
statements in, 2.16–2.17, 3.10, 3.12
syntax of, 3.12
8086 assembly language programs
8086 initialization, 8.35–8.39
8250 initialization, 14.27–14.28
8251A data transmission and reception, 14.8
8251A initialization, 14.7
8254 initialization, 8.35–8.39
8259A initialization, 8.35–8.39
8279 initialization, 9.39, 9.40
adding constants to arrays of data, 4.26–4.29, 12.14, 12.15–12.17
arithmetic average, 4.1–4.4
backward jump, 4.10–4.11
Basic Input Output System (BIOS)
procedure calls, 8.38–8.40
binary-coded decimal (BCD) packing, 4.5–4.8
binary-coded decimal (BCD) to binary conversion, 5.17–5.24
breakpoints in, 3.12, 3.28, 4.5, 4.6
C programs with, 12.42–12.17
circular (ring) buffer, 513–514
comparing strings, 5.4–5.6
conditional jumps, 4.15
critical region protection, 15.5–15.6
data sampling, 5.10–5.14
debugging, *See* Debugging programs
display driver, 9.40–9.41
display programming, 13.19–13.20
divide-by-zero interrupt-service procedure, 5.32–5.37
documentation of, 3.12
DOS function calls for files, 13.47
factorials, 5.26–5.31
far procedure additions to, 5.30
forward jump, 4.11
hand-coding, 2.19, 3.10–3.12, 3.18, 4.5

- heater control, 4.20–4.22
IF-THEN-ELSE structure, 4.15–4.20
IF-THEN structure, 4.15
 industrial process-control system, 10.37, 10.46
 inflation factor adjustment, 4.26–4.28
 initialization instructions for, 3.10, 4.25–4.4
 int array pointers, 12.14, 12.15–12.37
 interrupt input, 8.12–8.14
 keyboard input, 4.24–4.26
 microcomputer-based industrial process-control system, 10.37–10.40
 microcomputer-based scale, 10.23–10.32
 modules of, 5.31
 moving strings, 5.1–5.4
 multiplication, 3.19–3.26
 mutual exclusion, 15.6
 packed binary-coded decimal (BCD) code, 4.5–4.8, 5.17–5.24
 printed-circuit-board-making machine, 4.16–4.20
 printer driver, 9.13–9.16
 printer output, 8.38–8.41
 profit factor adjustment, 4.28–4.29, 12.14, 12.15–12.17
 read input port, 3.10–3.12
 real-time clock, 8.35–8.39
 relocatable, 4.13
 REPEAT-UNTIL structure, 4.22–4.29
 scale, 10.23–10.32
 sequence structure, 4.1–4.4, 4.6–4.8
 simplified segment directives in, 12.47–12.48
 stack usage additions to, 5.10, 5.11
 strobed input, 4.23–4.26
 terminal emulator, 14.23, 14.24–14.25
 unconditional jumps, 4.10–4.12
 video programming, 13.19–13.26
 WHILE-DO structure, 4.20–4.24
 8086-based microcomputers. *See also* SDK-4.25–4.26
 block diagram of, 7.1–7.2
 bus activities during read and write operations, 7.3–7.6
 overview of, 7.1–7.2
 troubleshooting, 7.42–7.46
 8086 family of microprocessors, 2.11–2.12
 8086 instructions
 arithmetic, 3.7–3.8, 4.4–4.5
 bit manipulation, 3.8
 D bit for, 3.13, 3.14
 data transfer, 3.7
 initialization, 3.10, 4.2, 4.3
 mnemonics for, 2.16, 3.10, 3.11
 MOD bit patterns for, 3.13–3.16
 opcodes (operation codes) for, 2.16, 2.19, 3.11, 3.13
 processor control, 3.9, 3.10
 program execution transfer, 3.8–3.9
 R/M bit patterns for, 3.13, 3.16
 REG bits for, 3.13–3.14
 string, 3.8, 5.1–5.6
 supersets of, 2.11
 templates for, 3.13–3.18
 timing for, 4.31–4.33
 W bit for, 3.13–3.14
 8086 microprocessor
 8087 math coprocessor cooperation with, 11.36–11.39
 8088 microprocessor versus, 7.39
 accumulator of, 2.13–2.14
 arithmetic logic unit (ALU) of, 2.11–2.13
 assembly language for *See* 8086 assembly language
 block diagram of, 2.10–2.11
 booting, 7.2, 7.7
 bus activities during read and write operations, 7.3–7.6
 bus interface unit (BIU) of, 2.11–2.15
 buses in, 2.12
 clock frequencies for, 7.39
 control circuitry in, 2.12–2.13
 decoder in, 2.12–2.13
 description of, 2.10–2.11
 direct addressing mode of, 2.19–2.20
 execution unit (EU) of, 2.11–2.14, 2.16
 flag register of, 2.13–2.14
 general-purpose registers of, 2.13–2.14
 high byte storage of, 2.19
 immediate addressing mode of, 2.18
 index registers of, 2.13, 2.16
 instruction pointer (IP) register of, 2.13, 2.14–2.15
 instructions for, *See* 8086 instructions
 interrupt response of, 8.1–8.2
 low byte storage of, 2.19
 math coprocessor for, *See* 8087 math coprocessor
 maximum mode of, 7.7, 11.4, 11.5
 memory organization of, 7.30–7.32
 minimum mode of, 7.7, 7.39–7.42, 11.4, 11.6
 pin configuration of, 7.6, 11.4
 pin description for, 7.6–7.7
 pointer registers of, 2.13, 2.14–2.15
 programmer's model of, 7.1
 programming introduction for, 2.16–2.20
 queue registers of, 2.13, 2.14
 READY input of, 7.3–7.6, 7.13, 7.38–7.39
 register addressing mode of, 2.18–2.19
 RESET response of, 7.3, 7.7
 segment registers of, 2.13, 2.14–2.16, 2.17–2.19
 stack pointer (SP) register of, 2.13–2.15
 stack segment (SS) register of, 2.13, 2.14–2.16
 system timing for, 7.3–7.6
 TEST input of, 10.34, 11.37–11.39
 timing parameters for, 7.39–7.42
 virtual 8086 mode (80386)
 WAIT instruction for, 11.34, 11.37–11.39
 WAIT states of, 7.2–7.6, 7.13, 7.38–7.39
 waveforms for, 7.3–7.6, 7.39–7.42
 word storage of, 35
 8087 math coprocessor
 8086 microprocessor cooperation with, 11.38–11.39
 architecture of, 11.27–11.28
 block diagram of, 11.27
 circuit connections for, 11.36–11.39
 control words for, 11.27–11.28
 data types for, 11.24–11.26
 double-precision numbers for, 11.24, 11.25–11.27
 fixed-point numbers for, 11.24
 floating-point numbers for, 11.25
 hypotenuse calculation with, 11.35–11.38
 instructions for, 11.30–11.35
 integers for, 11.24
 overview of, 11.23–11.24
 packed decimal numbers for, 11.24
 programming example for, 11.35–11.37
 real numbers for, 11.24–11.26
 single-precision numbers for, 11.24, 11.26
 socket for, 11.1
 stack of, 11.27–11.28
 status word for, 11.26–11.27
 8088 microprocessor, 2.12, 7.39, 11.36–11.39
 8096 microcontroller family, 2.11, 10.49–10.50
 82C08 DRAM controller, 11.12–11.15

- 8237 DMA controller
 circuit connections and operation, 11.5–11.8
 initializing, 11.8
 pin descriptions and signals for, 11.8–11.10
 timing waveforms for, 11.7–11.8
- 8250 UART
 control words for, 14.32, 14.33
 initializing, 14.32
 sending and receiving characters with, 14.34–14.35
- 8251A USART, 7.14, 7.15, 7.23
 BISYNC communication with, 14.37–14.38
 block diagram of, 14.4–14.5
 control words for, 14.5–14.8, 14.37, 14.38
 initializing, 14.5–14.8
 pin descriptions for, 14.4–14.5
 sending and receiving characters with, 14.7–14.8
 status word for, 14.6
 system connections and signals, 14.4–14.5
 write-recovery time for, 14.7
- 8253 programmable timer/counter, 8.17
- 8254 programmable timer/counter
 8253 versus, 8.17
 block diagram of, 8.18
 clock frequency for, 8.17
 control words for, 8.21–8.23, 8.30
 hardware-retriggerable one-shot, 8.24–8.25
 hardware-triggered strobe, 8.28–8.29
 initializing, 8.20–8.23
 internal addresses of, 8.20
 interrupt on terminal count, 8.23–8.24
 nonsystem clocks with, 8.29
 operation of, 8.17–8.19
 read-back feature of, 8.17, 8.30
 reading the count from, 8.29–8.30
 SDK-86 addition of, 8.18, 8.19
 software-triggered strobe, 8.28, 8.29
 square-wave mode (mode 3), 8.27–8.28
- 8254 programmable timer/counter
 system connections for, 8.18, 8.19
 timed interrupt generator, 8.25–8.27
 timing waveforms for, 8.23–8.29
- 8255A programmable parallel port
 block diagram of, 9.4
- Centronics parallel interface to, 9.9, 9.11–9.12
 control words for, 9.5–9.7, 9.12
 lathe interface using, 9.7–9.9
 operational modes of, 9.5–9.7
 paper tape reader interface using, 9.7–9.9
 printer driver program using, 9.13–9.16
 in SDK-86, 7.13, 7.14, 7.19
 status word for, 9.14–9.16
 system connections for, 9.3
- 8259A priority interrupt controller (PIC)
 8086 interrupt-acknowledge cycles for, 8.10–8.11
 cascading, 8.19, 8.33
 fixed-priority mode of, 8.31–8.32
 in-service register (ISR) of, 8.31, 8.32
 initialization command words (ICWs) of, 8.33–8.38
 initializing, 8.33–8.38
 interrupt mask register (IMR) of, 8.31, 8.32
 interrupt request register (IRR) of, 8.31, 8.32
 operation command words (OCWs) of, 8.36, 8.38
 overview of, 8.31, 8.33
 priority resolver of, 8.31, 8.32
 SDK-86 addition of, 8.18, 8.19, 8.33
 system connections for, 8.18, 8.19, 8.22
 use of, 8.29–8.31
- 8272A floppy-disk controller, 13.39–13.42
- 8279 dedicated display controller
 7-segment display interfacing with, 9.33–9.41
 connections for, 9.34–9.37
 control words for, 9.37–9.48
 decoded scan with, 9.37, 9.39
 encoded scan with, 9.37, 9.39
 initializing and communicating with, 9.36
 keypad interfacing with, 9.33–9.40
 scan time for, 9.36
 status word for, 9.40
 timing waveforms for, 9.35–9.37
- 8279 specialized I/O device, 7.14, 7.17, 7.18, 7.21
- 8284 clock generator, 7.13, 7.14, 7.16
- 8286 control and data transceivers, 7.14, 7.18
- 8421-binary-coded decimal (BCD) code, 1.4, 1.5
- 8514/A high-resolution graphics board, 13.15
- 9900 family of microprocessors, 2.10
- 32032 microprocessor, 2.10
- 68000 microprocessor, 2.10
- 68020 microprocessor, 2.10
- 80186 and 80188 microprocessors, 2.11, 10.49, 10.52, 15.1–15.2
- 80286 microprocessor, 15.11
- 80287 coprocessor for, 15.14
 architecture of, 15.11
 block diagram of, 15.11
 instructions for, 15.15–15.16
 interrupts for, 15.13, 15.14
 introduced, 2.11, 15.1–15.2
 pin configuration of, 15.12–15.13
 protected mode of, 15.14, 15.16–15.17
 real mode of, 15.11, 15.12, 15.16–15.17
 system connections for, 15.13–15.14
- 80287 math coprocessor, 15.12
- 80386 microprocessor
 80386DX version of, 15.16–15.17
 80386SX version of, 15.16–15.17
 addressing modes of, 15.34
 architecture of, 15.16
 extended industry standard architecture (EISA) bus for, 15.20–15.21
 index scaling feature of, 15.34
 industry standard architecture (ISA) bus for, 15.20
 instructions for, 15.34–15.33
 introduced, 2.11
- MicroChannel Architecture (MCA)
 bus for, 15.21–15.22
 pin configuration of, 15.15, 15.18
 programming, 15.36–15.38
 protected mode of, See 80386 protected mode
 real mode of, 15.22, 15.23
 registers of, 15.22, 15.23, 15.34–15.35
 signal groups of, 15.15–15.16
 summary of hardware and modes, 15.33
 system connections for, 15.19–15.20
 systems using, 15.19–15.20
 virtual 8086 mode of, 15.33–15.34
- 80386 protected mode
 call gates, 15.28
 flat system memory model, 15.30–15.31
 input/output (I/O) privilege levels, 15.28
 interrupt and exception handling, 15.27–15.28

- operating systems using, 15.38
 paged flat memory model, 15.31–15.32
 paging mode, 15.30–15.31
 segment privilege levels and protection, 15.27–15.28
 segmentation, 15.22–15.23
 segmented-paged memory model, 15.31–15.32
 segments-only memory model, 15.31–15.32
 task state segment, 15.28–15.29
 task switching, 15.28–15.29
 virtual memory, 15.24–15.25
 80386DX microprocessor, 15.36–15.37
 80386SX microprocessor, 15.36–15.37
 80387 math coprocessor, 15.42
 80486 built-in math coprocessor, 15.41–15.42
 80486 microprocessor, 15.41–15.43
 80586 microprocessor, 15.42–15.43
 80686 microprocessor, 15.43
 80786 built-in math coprocessor, 15.43
 80786 microprocessor, 15.43
 80960 microcontroller family, 10.52
 82385 cache controller, 11.17, 11.18–11.21
 A-bus, 2.12
 AAA instruction, 3.7, 6.2
 AAD instruction, 3.8, 6.1–6.2
 AAM instruction, 3.8, 6.1–6.2
 AAS instruction, 3.7, 6.2
 Absolute shaft encoders, 9.51–9.52
 Abstracts for programs, 3.12
 Access times for disks, 13.34–13.35
 Accessed bit, 15.10
 Accumulator (AL register), 2.13–2.14
 Accuracy (precision) of numbers, 11.25
 ACK (affirmative acknowledge) character, 14.36–14.37
 Active filters, 10.3, 10.6–10.7
 Actual arguments (parameters) of functions, 12.33–12.34
 A/D converters, *See* Analog-to-digital converters
 ADC instruction, 3.7, 6.2
 ADD instruction, 3.7, 3.18, 4.7–4.8, 6.3–6.4
 Adders, 10.3, 10.5–10.6
 Addition
 8086 instructions for, 3.7
 8087 instructions for, 11.31–11.32
 binary, 1.7–1.8
 binary-coded decimal (BCD), 1.11
 Address bus, 1.16, 1.17, 2.6–2.9
 Address counter, 2.6
 Address decoders, 7.26–7.27, 10.49–10.50
 Address inputs, 1.16, 1.17
 Address marks on disks, 13.38–13.39
 Address pipelining, 15.18–15.19
 Address transfer instructions, 3.7
 Address unit (AU) (80286), 15.11
 Addresses
 base, 2.14, 2.15, 2.16, 2.18, 3.14–3.15, 4.29–4.30
 breakpoint, 3.12, 3.27, 4.5–4.6, 12.23–12.24
 effective (EA), 2.18, 3.14–3.15, 4.29–4.30
 internal, 8.20
 labels as, 3.10, 3.11, 3.24
 logical, 15.9–15.10
 named, 3.24
 offsets (displacements) of, 2.14, 2.18, 4.27–4.28, 15.10
 physical, 2.14–2.15, 2.18, 3.14, 4.29–4.30, 15.10
 pipelined, 15.18–15.19
 return, 5.7, 6.4
 segment base, 2.14, 2.15, 2.16, 2.18, 3.14, 3.15, 4.29–4.30
 segment base: offset form of, 2.15, 2.19
 virtual, 15.9–15.10
 Addressing modes
 80386, 15.33–15.34
 defined, 2.18
 direct, 2.18–2.19
 double indexed, 3.14, 4.29–4.30
 immediate, 2.18
 MOD bit patterns for, 3.13–3.15
 R/M bit patterns for, 3.13–3.15
 real, 2.11
 register, 2.18–2.19
 single indexed, 3.14, 4.29–4.30
 summary of, 3.14–3.15
 virtual, 2.11
 Advanced Micro Devices 2900 family of bit-slice processors, 2.10
 AF (auxiliary carry flag), 2.13, 2.14, 4.12, 4.13
 Affirmative acknowledge (ACK) character, 519
 AH register, 2.13–2.14
 AL register (accumulator), 2.13–2.14
 Algorithms
 converting to assembly language, 4.4
 defined, 3.2
 downloading program, 14.26
 flowcharts for, 3.2–3.6
 microcomputer-based scale, 10.21–10.23
 program development, 3.28, 3.29
 standard structures for, 3.4–3.7
 structured programming of, 3.4–3.7
 terminal emulator, 14.23
 top-down design of, 3.3–3.7
 Align on even memory address (EVEN) directive, 6.32–6.33
 Alphanumeric codes, 1.5–1.7
 Alphanumeric displays, *See* Light-emitting diodes; Liquid-crystal displays
 ALU, *See* Arithmetic logic unit
 AM (amplitude modulation), 14.13–14.14
 American Standard Code for Information Interchange (ASCII)
 ASCII string, 13.46
 converting EBCDIC to, 9.31–9.32
 in data statements, 3.22
 described, 1.5–1.6
 extended, 13.2–13.7
 keyboard input, 4.23–4.25
 table of codes, 1.6, 13.2
 Amplifiers, *See* Operational amplifiers
 Amplitude modulation (AM), 14.13–14.14
 Analog-to-digital (A/D) converters
 8096, 10.49–10.50
 conversion time for, 10.17
 dual-slope, 10.18–10.19, 10.21
 high-speed, 10.17
 microcomputer interfacing for, 10.20–10.21
 output codes for, 10.20
 parallel comparator (flash), 10.17, 10.20–10.21
 specifications for, 10.17
 successive approximation, 10.18–10.21
 AND gate, 1.13
 AND instruction, 3.8, 4.6–4.7, 6.3
 AND matrixes, 1.14
 Answer modem, 14.17
 APIC – advanced programmable interrupt controller, 16.1, 16.7
 APIs (application program interfaces), 15.39
 Application program interfaces (APIs), 15.39
 Arithmetic average, 4.1–4.4
 Arithmetic instructions
 8086, 3.7–3.8, 4.4–4.5,

- 8087, 11.31–11.33
Arithmetic logic unit (ALU)
 74LS181, 1.18–1.19
 8086, 2.11–2.12
 defined, 1.18–1.19
 microprocessor categorization using, 2.9
 Arithmetic operators in C, 12.20
ARPL instruction (80286), 15.16
Arrays
 data, 4.25–4.29
 elements of, 4.26–4.29
 indexes of, 4.29
 type of, 4.27
ASCII, *See American Standard Code for Information Interchange*
Assembler directives
 align on even memory address (EVEN), 6.32–6.33
 ASSUME, 3.25–3.26, 6.31
 define byte (DB), 3.22, 6.31
 define doubleword (DD), 3.22, 6.31
 define quadword (DQ), 6.31
 define ten bytes (DT), 6.31–6.32
 define word (DW), 3.22, 6.32
 end procedure (ENDP), 6.32
 end program (END), 3.26, 6.32
 end segment (ENDS), 3.19–3.20, 6.32
 equate (EQU), 3.20–3.21, 6.32
 external (EXTRN), 5.32, 6.33, 6.35
GLOBAL, 6.33
GROUP, 6.33
INCLUDE, 6.33
LABEL, 6.34
LENGTH operator, 6.34
NAME, 6.34
OFFSET operator, 6.34
originate (ORG), 6.34
pointer (PTR), 6.35
procedure (PROC), 6.35
PUBLIC, 5.32, 6.33, 6.34, 6.35
SEGMENT, 3.19–3.20, 6.35
SHORT operator, 6.35
TYPE operator, 6.35
Assembler macros
 defined, 5.37
 dummy variables in, 5.38
 expanding, 5.37
 in-line code and, 5.37
 passing parameters to, 5.38
 procedures versus, 5.37, 5.38–5.39
 without parameters, 5.35–5.37
Assembler program, 2.17, 3.10, 3.19
Assembly language, *See 8086 assembly language*
Assertion level,
Assignment operator in C, 12.20
ASSUME directive, 3.24–3.25, 6.31
Asynchronous communication, 14.3
Asynchronous inputs, 1.15
Attribute code, 13.9–13.10
AU (address unit) (80286), 15.11
Audio speaker buffer, 8.27
Audio-tone generators, 8.27
Automatic storage class in C, 12.35
Auxiliary carry, 1.12
Auxiliary carry flag (AF), 2.13, 4.12–4.13
Available interrupts, 8.2
Average of numbers, 4.1–4.4
AX register, 2.13
B-bus, 2.12
Backbones for networks, 14.46–14.47
Backward jump, 4.9–4.12
Bandpass filters, 10.6
Bang-bang (on-off) control, 10.46
Bank-switched memory, 15.7–15.8
Base-2 numbers, *See Binary numbers*
Base-10 (decimal) numbers, 1.1
Base-16 (hexadecimal) numbers, 1.3, 1.11–1.12
Base address, 2.14, 2.15, 2.16, 3.14–3.15, 4.29–4.30
Base pointer (BP) register, 2.12, 2.15
Baseband transmission, 14.41–14.42
Basic Input Output System (BIOS), 8.39–8.41, 13.1–13.5, 13.19–13.21
Baud rate, 7.24, 14.2
BCC (block check characters), 14.36–14.37
BCD, *See Binary-coded decimal code*
BCP (byte-oriented protocol), 14.36–14.38
Begin flowchart symbol, 3.2–3.3
Behavioral models, 11.42
BH register, 2.12–2.15
Biased exponent of numbers, 11.26
Bidirectional bus, 2.7
Binary addition, 1.7–1.8
Binary-coded decimal (BCD) code
 8421, 1.4, 1.5
 addition with, 1.11
 in data statements, 3.22
 decimal adjust operation for, 1.11
 described, 1.4, 1.5, 1.11
 excess-1, 1.4, 1.5
packed, 4.6–4.8, 5.17–5.24, 11.24–11.30
subtraction with, 1.11
unipolar, 10.20
unpacked, 4.6–4.8
Binary codes, 10.20
Binary counters, 1.16
Binary digits, *See Bits*
Binary division, 1.10
Binary multiplication, 1.9–1.10
Binary (base-2) numbers
 2's complement of, 1.7–1.9, 3.8, 3.21, 3.22, 6.20
 in data statements, 3.22
 described, 1.1–1.4, 1.7–1.10
packed binary-coded decimal (BCD)
 conversion to, 5.19–5.24
Binary subtraction, 1.9
Binary Synchronous Communication Protocol (BISYNC), 14.36–14.38
Binary-to-decimal conversion, 1.1–1.3
Binary words
 binary digits (bits) in, *See Bits*
 byte (8-bit), 1.1
 doubleword (32-bit), 1.2
 least significant bit (LSB) of, 1.2
 most significant bit (MSB) of, 1.2, 1.3, 2.11, 2.13
 nibble (4-bit), 1.2
 word (16-bit), 1.2
Binder, 15.38
BIOS (Basic Input Output System), 8.39–8.41, 11.47, 13.1–13.5, 13.21–13.22
Bipolar binary codes, 10.20
BISYNC (Binary Synchronous Communication Protocol), 14.36–14.38
Bit-aligned block transfer (BITBLT), 13.26
Bit manipulation instructions, 3.8
Bit-mapped raster scan display, 13.11
Bit-oriented protocol (BOP), 14.38–14.39
Bit scan and test Instructions (80386), 15.35
Bit-slice processors, 2.10
BITBLT (bit-aligned block transfer), 13.26
Bits (binary digits)
 accessed, 15.10–15.11
 check (encoding), 11.23
 D, 3.13, 3.15
 defined, 1.2
 dubits, 14.13–14.14
 dirty, 15.10–15.11

- flag, 14.35–14.36
 masking, 4.6–4.7
 MOD, 2.13–3.15
 parity, 1.5, 11.21–11.22
 privilege-level, 15.10
 quadbits, 14.13–14.14
 R/M, 2.13–2.15
 REG, 2.13–2.14
 sign, 1.7–1.9
 start, 14.2–14.3
 stop, 14.2–14.3
 trits, 14.15
 W, 3.14
- Bitwise operators in C, 12.20–12.21
 BIU (bus interface unit), 2.12–2.16
 BL register, 2.13–2.14
 Block check characters (BCC), 14.36–14.37
 Blocked tasks, 15.5–15.6
 Blocks of memory, 9.13
 Boot record on disks, 13.44
 Booting the 8086, 7.3, 7.7
 BOP (bit-oriented protocol), 14.38–14.39
 Bottom-up design, 3.3
 BOUND instruction (80186/80188), 3.9, 10.52
 BOUND instruction (80286), 15.15–15.16
 BP (base pointer) register, 2.13–2.14
 Breakout box, 14.12–14.13
 Breakpoint frequency, 10.6
 Breakpoint interrupts, 8.3, 8.9
 Breakpoints, 3.12, 3.27, 4.5–4.6, 8.3, 8.9, 15.22–15.23
 Bresenham's algorithm, 13.25–13.26
 Bridges (gateways) for networks, 14.44–14.45
 Broadband bus (tree-structured) networks, 15.22–15.23
 Broadband transmission, 14.35–14.36
 BSF instruction (80386), 15.34–15.35
 BSR instruction (80386), 15.34–15.35
 BSWAP instruction (80486)
 BT instruction (80386), 15.34–15.36
 BTC instruction (80386), 15.34–15.36
 BTR instruction (80386), 15.34–15.36
 BTS instruction (80386), 15.34–15.36
 BU (bus unit) (80286), 15.11–15.12
 Buffers
 circular (ring), 14.32–14.34
 high-power, 10.15–10.16
 integrated-circuit (IC), 9.43–9.44
 inverting, 1.12
 noninverting, 1.12, 10.4
 transistor, 9.44–9.47
 Builder, 15.38
 Bus interface unit (BIU), 2.12–2.16
 Bus master, 15.20
 Bus unit (BU) (80286), 15.11
 Buses
 8086, 2.12
 A-bus, 2.12
 activities during read and write operations, 7.3–7.6
 address, 1.16, 1.17, 2.5–2.9
 B-bus, 2.12
 bidirectional, 2.7
 C-bus, 2.12
 control, 2.6–2.9
 data, 1.16, 1.17, 2.6–2.9
 extended industry standard architecture (EISA) bus, 15.20–15.21
 industry standard architecture (ISA) bus, 15.20
 MicroChannel Architecture (MCA) bus, 15.21–15.22
 BX register, 2.13
 Bypass capacitors, 7.15, 7.25
 Byte
 defined, 1.1
 high, 2.19
 low, 2.19
 Byte transfer instructions, 3.7
 Byte type, 2.18, 3.21, 6.31
 C-bus, 2.12
 C programming language
 actual arguments (parameters) of functions in, 12.33
 arithmetic operators in, 12.20
 assignment operator in, 12.20
 automatic storage class in, 12.35–12.36
 bitwise operators in, 12.20–12.21
 calling functions in, 12.32–12.35
 CASE structure in, 12.25–12.26
 char (character) pointers in, 12.18–12.20
 char (character) variables in, 12.9–12.10
 character strings in, 12.18–12.20
 combined operators in, 12.21
 curly braces in, 12.3
 data types in, 12.8–12.9
 declaring functions in, 12.32–12.34
 defining functions in, 12.32–12.34
 dereferencing pointers in, 12.13
 do-while structure in, 12.24–12.25
 enumerated data type in, 12.9
 extern (global) storage class in, 12.35–12.36
 FILE pointers in, 13.46
 float (floating-point) pointers in, 12.16–12.18
 float (floating-point) variables in, 12.11
 FOR-DO loop in, 12.27–12.32
 FOR loop in, 12.27–12.32
 formal arguments (parameters) of functions in, 12.32
 function storage classes in, 12.35–12.36
 if-else structure, 12.24–12.25
 IF-THEN-ELSE structure in, 12.24–12.25
 IF-THEN structure in, 12.24–12.25
 index method of accessing array elements, 12.29–12.31
 int array (integer array) pointers in, 12.13–12.17
 int (integer) pointers in, 12.11–12.13
 int (integer) variables in, 12.10–12.11
 Integrated Development Environment for, 12.5–12.8
 introduced, 12.7
 keyboard input library functions for, 12.39–12.40
 library functions for, 12.39–12.42
 logical operators in, 12.22–12.23
 math library functions for, 12.41–12.42
 memory models for, 12.44
 operator precedence in, 12.22–12.23
 output library functions for, 12.40
 parentheses in, 12.3–12.4
 passing array pointers to functions in, 12.36–12.37
 passing parameters by reference, 12.13
 passing parameters by value, 12.13
 pointer method of accessing array elements, 12.31–12.32
 pointers and functions with two-dimensional arrays, 12.36–12.39
 pointers to functions, 12.38
 preprocessor directives in, 12.2
 program development tools for, 12.3–12.7
 programs, See C programs
 prototypes of functions in, 12.33–12.34
 register storage class in, 12.35
 relational operators in, 12.21

- REPEAT-UNTIL structure in, 12.26–12.27
 static storage class in, 12.35
 string library functions for, 12.40–12.41
 switch structure in, 12.25–12.26
 union data structure in, 13.3–13.4
 variable declarations in, 12.9–12.11
 variable storage classes in, 12.35–12.36
 variable types in, 12.9–12.11
 WHILE-DO structure in, 12.26–12.27
 while structure in, 12.26–12.27
- C programs
 8086 assembly language programs with, 12.42–12.49
 adding constants to arrays of data, 12.1–12.5, 12.13–12.17, 12.35–12.36
 arithmetic average, 12.29–12.32, 12.36–12.38
 calling functions, 12.33–12.34
 char pointers, 12.18–12.20
 char variables, 12.9–12.10
 declaring functions, 12.33–12.34
 defining functions, 12.33–12.34
 disk file operations, 13.45–13.46
 do-while structure, 12.26–12.27
 downloading to SDK-86, 14.25–14.35
 float pointers, 12.16–12.17
 float variables, 12.11
 graphics programming, 13.26–13.28
 hypotenuse calculation, 12.41–12.42
 if-else, 12.24–12.25
 index method of accessing array elements, 12.29–12.31
 int array pointers, 12.15–12.17
 int pointers, 12.12–12.15
 int variables, 12.10–12.11
 keyboard input, 13.5–13.7
 for loops, 12.29–12.32
 passing array pointers to functions, 12.35–12.36
 pointer method of accessing array elements, 12.31–12.32
 pointers and functions with two-dimensional arrays, 12.36–12.39
 profit factor adjustment, 12.1–12.3, 12.13–12.28, 12.35–12.36
 switch structure, 12.26–12.27
 video programming, 13.26–13.28
 while structure, 12.26–12.27
- Cache memory
 80486, 15.1–15.6
- 82385 cache controller for, 11.16, 11.17–11.21
 cache directory for, 11.17
 direct-mapped, 11.17–11.18
 DRAM and, 11.17–11.18
 fully associative, 11.19–11.21
 hit rate for, 11.15, 11.17
 implementation of, 11.15, 11.17
 posted-write-through for, 11.17
 SRAM and, 11.15, 11.17, 11.21
 summary of, 11.21
 two-way set associative, 11.19–11.20
 virtual memory versus, 15.9–15.10
- Caches, disk, 11.48
- CAE (computer-aided engineering), 11.39
- CALL instruction, 3.8, 5.7–5.9, 5.14–5.15, 6.4–6.5
- Call table, 10.40, 10.45
- Called modem, 14.17–14.18
- Calling functions in C, 12.32–12.35
- Calling modem, 14.17–14.18
- Calls
 direct, 5.8–5.9, 6.4–6.5
 far (intersegment), 5.7, 5.8–5.9, 5.15–5.16
 indirect, 5.7, 5.9, 5.15, 5.16
 near (intrasegment), 5.7, 5.8, 5.14, 5.15, 6.4
- Capacitive keyswitches, 9.18
- Capacitors
 bypass, 7.15, 7.25
 filter, 7.15, 7.25
- Carrier sense, multiple access with collision detection (CSMA/CD), 14.40, 14.41–14.44
- Carry, auxiliary, 1.12
- Carry flag (CF), 2.13, 2.14, 4.12
- Cascaded devices, 8.33
- Case design, 14.46–14.47
- CASE structure, 3.4–3.6, 4.20–4.21, 12.25–12.26
- Cathode-ray tube (CRT) displays
 6845 CRT controller for, 13.8–13.9
 8514/A high-resolution graphics board for, 13.15
 attribute code and, 13.9–13.10
 bit-mapped, 13.11
 character display on, 13.6–13.7
 character generator ROM for, 13.6–13.7
 color, 13.11–13.17
- color graphics adaptor (CGA) for, 13.15–13.16
- composite video color monitor, 13.12
- display refresh RAM for, 13.7–13.8
- dot clock for, 13.7–13.8
- enhanced graphics adaptor (EGA) for, 13.17–13.8
- field of, 13.6
- frame buffer for, 13.7–13.8
- frame rate for, 13.6–13.7
- frequencies for, 13.8–13.9
- Hercules adaptors for, 13.15
- high-resolution graphics, 13.29–13.30
- horizontal sync pulse for, 13.8
- interlaced scanning for, 13.6–13.7
- monochrome, 13.5–13.6, 13.7, 13.15
- multicolor graphics array (MCGA) for, 13.15
- noninterlaced scanning for, 13.6–13.7
- overscan in, 13.9
- packed pixel storage for, 13.12–13.13
- picture element (pel or pixel) of, 13.7
- pitch of, 13.7
- planar pixel storage for, 13.12–13.14
- plasma displays, 13.31
- programming. See Video programming
- RAMDAC for, 13.7
- raster scanning for, 13.6–13.7
- terminal, 13.7
- timing for, 13.9–13.10
- vertical sync pulse for, 13.7
- video DAC for, 13.17–13.18
- video graphics array (VGA) for, 13.15, 13.18–13.20
- video monitor, 13.6
- video RAM (VRAM) for, 13.14–13.15
- CBW instruction, 3.8, 6.4
- CCD (charge-coupled device) cameras, 13.33
- CD (compact disk), 11.49
- CD-I (compact digital interactive), 13.55
- CDQ instruction (80386), 15.35
- Celeron, 16.1
- Central processing unit (CPU). See also Microprocessors
 buses connected to, 2.6–2.9
 decoding of instructions by, 2.7–2.9
 defined, 2.1, 2.7, 24
 execution of instructions by, 2.7, 2.9
 fetching of instructions by, 2.7–2.9

- general-purpose, 2.10
 microcomputer, 2.6–2.9
 purposes of, 2.7
 registers in, 2.7
- Centronics parallel interface
 connections for, 9.9–9.13
 pin descriptions for, 9.9–9.13
 printer driver program for, 9.11–9.12
 SDK-86 connections for, 9.9, 9.12, 9.13
 timing waveforms for, 9.12–9.13
- CF (carry flag), 2.13, 2.14, 4.12–4.13
- CGA (color graphics adaptor), 13.14–13.16
- CH register, 2.14–2.15
- Char (character) pointers in C, 12.18–12.20
- Char (character) variables in C, 12.9–12.10
- Character generator read-only memory (ROM), 13.7–13.9
- Character strings in C, 12.18–12.20
- Characters
 affirmative acknowledge (ACK), 14.37
 block check (BCC), 14.36–14.37
 control, 1.7
 cyclic redundancy (CRC), 13.38–13.39, 14.38, 14.39–14.41
 end-of-block (ETB), 14.36
 end-of-text (ETX), 14.36
 end-of-transmission (EOT), 14.36
 enquiry (ENQ), 14.36
 negative acknowledge (NAK), 14.37
 sentinel method for sending, 9.13
 start-of-header (SOH), 14.36
 start-of-text (STX), 14.36
 sync, 14.36
 token, 14.41
- Charge-coupled device (CCD) cameras, 13.33
- Check (encoding) bits, 11.22
- Chip enable inputs, 1.17
- CIM (computer-integrated manufacturing), 11.47
- Circular (ring) buffer, 14.32–14.34
- CISC (complex instruction set computer) processors, 13.30
- CL register, 2.13–2.14
- Cladding material, 14.21
- CLC instruction, 3.9, 6.4
- CLD instruction, 3.9, 6.4
- CLI instruction, 3.9, 6.4
 delay loops, 4.31–4.33, 5.10–5.14
- Clock waveform, 7.4
- Clocks
 nonsystem, with 8254, 8.29
 real-time, 8.16–8.17, 8.35–8.39
 states of, 7.4
- Closed-loop gain, 10.4–10.5
- Closing files, 13.46–13.48
- Clusters on disks, 13.44
- CMC instruction, 3.9, 6.4
- CMP instruction, 3.9, 6.4
- CMPS/CMPSB/CMPSW instructions, 6.5
- CNC (computer numerical control) machines, 9.7–9.9
- Code conversion
 XLAT method for, 9.31–9.32
- Code segment, 3.24
- Code segment (CS) register, 2.12, 2.14–2.16, 4.29–4.30
- Codecs, 14.18–14.19
- Coders, 14.18–14.19
- Codes
 7-segment display, 1.5
 alphanumeric, 1.5–1.6
 analog-to-digital (A/D) output, 10.19–10.20
 ASCII. *See* American Standard Code for Information Interchange
 attribute, 13.39–13.40
 BCD. *See* Binary-coded decimal code
 binary, 10.20
 bipolar binary, 10.20
 digital-to-analog (D/A) input, 10.15, 10.20
 error detecting/correcting (ECCs), 11.22–11.23
 Extended Binary-coded Decimal Interchange Code (EBCDIC), 1.6, 1.7, 9.31–9.32
 frequency modulation (FM), 13.37, 13.38
 Gray, 1.4–1.5, 9.51–9.52
 Hamming, 11.22–11.23
 linear predictive (LPC), 13.53
 Manchester, 14.43
 modified frequency modulation (MFM), 13.37–13.38
 nonreturn-to-zero (NRZ), 13.37
 run-length-limited (RLL), 13.38, 13.43
 trellis, 14.16
 unipolar binary, 10.20
- Coding sheets for programs, 3.10–3.12
- Coding templates, 3.13–3.19
- Cold-junction compensation, 10.10
- Collision of transmissions, 14.42–14.43
- Color cathode-ray tube (CRT) displays, 13.11–13.19
- Color graphics adaptor (CGA), 13.15–13.17
- Color palettes, 11.12
- Combined operators in C, 12.21
- Command words. *See* Control words
- Comment field, 2.16
- Comments in programs, 2.17, 3.10, 3.11
- Common-bus networks, 14.40–14.41
- Common-mode rejection, 10.6
- Common-mode voltage, 10.5
- Compact digital interactive (CD-I), 13.55
- Compact disk (CD), 11.49
- Companders (companding codecs), 14.18–14.19
- Comparators, 10.2–10.3
- Compare instructions, 8087, 11.33–11.34
- Compare technique for code conversion, 9.19–9.21
- Compiler program, 2.17
- Complete decoding, 7.37
- Complex instruction set computer (CISC) processors, 13.30
- Composite video color monitor, 14.16
- COMPS/COMPSB/COMPSW instructions, 3.8
- Computer-aided engineering (CAE), 11.39
- Computer-integrated manufacturing (CIM), 11.47
- Computer numerical control (CNC) machines, 9.7–9.9
- Computer vision, 13.33–13.35
- Conditional flags, 2.13–2.14, 4.12–4.13
- Conditional jumps, 4.8, 4.14–4.15, 4.23–4.24, 4.30–4.31
- Conditional transfer instructions, 3.9
- Connector flowchart symbol, 3.2
- Connector symbols, 7.25
- Constants
 8087 instructions for, 11.34
 named, 3.19–3.20
- Constellations (phase-amplitude graphs), 14.13–14.14
- Contact bounce and debouncing, 9.18, 9.19–9.21
- Contactors, 9.47
- Context (environment, or state) of tasks, 15.3–15.4
- Context switching, 15.3–15.4
- Control bus, 2.6–2.9
- Control characters, 1.7
- Control circuitry, 8086, 2.12, 2.13

- Control flags, 2.13
 Control words
 8250, 14.41–14.42
 8251A, 14.5–14.9, 14.37–14.38
 8254, 8.21–8.23, 8.29
 8255A, 9.5–9.7, 9.12
 8279, 9.36, 9.37
 Controllers
 dedicated, 2.10
 embedded. *See* Embedded controllers
 Coprocessors. *See* Math coprocessors
 Counters
 8253, 8.17
 8254. *See* 8254 programmable timer/counter
 address, 2.7
 binary, 1.5
 flip-flops as, 1.14–1.15
 location, 5.4–5.5
 registers as, 4.25–4.27
 Counting, interrupts for, 8.14–8.16
 CPU. *See* Central processing unit
 CRC (cyclic redundancy characters), 13.38, 13.39, 14.37, 14.38, 14.39
 Critical angle, 14.21, 14.22
 Critical frequency, 10.6
 Critical region, 14.34, 15.4–15.5
 CRT displays. *See* Cathode-ray tube displays
 CS (code segment) register, 2.12, 2.14–2.16, 4.29–4.30
 CSMA/CD (carrier sense, multiple access with collision detection), 14.41, 14.42–14.43
 CTS instruction (80286), 15.16
 Current loops, 10.13, 14.3, 14.9
 CWD instruction, 3.8, 6.6
 CWDE instruction (80386), 15.35
 CX register, 2.13
 Cycles (instruction and machine), 7.4
 Cyclic redundancy characters (CRC), 13.39, 13.40, 14.37, 14.38–14.40
 D bit, 3.13, 3.15
 D flip-flop, 1.14
 D latch, 1.14–1.15
 D/A converters. *See* Digital-to-analog converters
 DAA instruction, 3.7, 6.6–6.7
 Darlington transistors, 9.45–9.46
 DAS (data acquisition system), 10.20, 10.34, 10.35
 DAS instruction, 3.7, 6.6
 Data acquisition system (DAS), 10.20, 10.34, 10.35
 Data arrays, 4.26–4.27
 Data bases, defined, 2.4
 Data bus, 1.16, 1.17, 2.6–2.9
 Data communication equipment (DCE), 14.3, 14.9
 Data concentrators (multiplexers), 14.40
 Data link layer (OSI model), 14.41, 14.42
 Data outputs, 1.16–1.17
 Data-ready signal, 4.23–4.27
 Data segment, 3.24
 Data segment (DS) register, 2.12, 2.14–2.16, 2.16–2.17, 4.29–4.30
 Data statements
 binary-coded decimal (BCD) numbers in, 3.22
 binary numbers in, 3.22
 decimal numbers in, 3.22
 hexadecimal numbers in, 3.22
 numbers used in, 3.22–3.23
 Data storage registers, 1.15
 Data terminal equipment (DTE), 14.31, 14.4, 14.5
 Data transfer instructions
 8086, 3.7
 8087, 11.31
 Data-type conversion instructions (80386), 14.34–14.35
 Data types in C, 12.8–12.9
 DB (define byte) directive, 3.22, 6.31
 DCE (data communication equipment), 14.3, 14.7
 DD (define doubleword) directive, 3.22, 6.31
 Deadlock, 15.5–15.6
 Debouncing keyboards, 9.19–9.23
 Debug registers (80386), 15.22, 15.24
 Debugger program, 3.27
 Debugging programs
 5-minute rule for, 4.4
 breakpoints for, 4.4–4.6
 GO command for, 4.5
 with procedures, 5.24–5.25
 single-step command for, 4.5
 trace data for, 3.28, 7.9, 7.10–7.11
 DEC instruction, 3.7, 6.7
 Decimal adjust operation, 1.12
 Decimal (base-10) numbers
 in data statements, 3.22
 defined, 1.1
 Decimal-to-binary conversion, 1.3
 Decimal-to-hexadecimal conversion, 3
 Decision flowchart symbol, 3.2
 Decision operations, 3.3–3.4
 Declaring functions in C, 12.32–12.34
 Decoder
 8086, 2.12–2.13
 pulse code, 14.19
 Decoding instructions, 2.7–2.9
 Dedicated controllers, 2.10
 Dedicated interrupts, 8.2
 Define byte (DB) directive, 3.21, 6.31
 Define doubleword (DD) directive, 3.21, 6.31
 Define quadword (DQ) directive, 6.31
 Define ten bytes (DT) directive, 6.31, 6.32
 Define word (DW) directive, 3.22, 6.32
 Defining functions in C, 12.32–12.34
 Delay loops, 4.31–4.33, 5.10–5.14
 Delta (differential) modulation, 13.54
 Dereferencing pointers, 12.13
 Derivative feedback, 10.33–10.34
 Descramblers, 14.16
 Descriptor table, 15.10–15.11
 Design and development tools. *See* Electronic design automation
 Design rule checker (DRC) program, 11.40
 Design for test, 11.46, 11.47
 Destination index (DI) register, 2.12, 2.15
 Destination for instructions, 2.16–2.17
 DF (direction flag), 2.13
 DH register, 2.13–2.14
 DI (destination index) register, 2.12, 2.13
 Dibits, 14.15–14.16
 Differential (delta) modulation, 13.54
 Differential operational amplifiers (op amps), 10.3, 10.5
 Differential phase-shift keying (DPSK) modulation, 14.14
 Differential pressure transducer, 10.13
 Differentiators, 10.3, 10.6
 Digital feedback, 9.45–9.46
 Digital filters
 advantages of, 10.52
 block diagram of, 10.57–10.58
 development tools for, 10.57–10.58
 finite impulse response (FIR) algorithm for, 10.55
 hardware for, 10.55–10.57
 infinite impulse response (IIR) algorithm for, 10.56
 operation of, 10.54–10.55

- principle of, 10.52–10.53, 10.54
 sampling signals for, 10.54–10.55, 10.57
 software for, 10.57
 switched capacitor, 10.58
- Digital-to-analog (D/A) converters
 characteristics and specifications, 10.13–10.14
 full-scale output voltage of, 10.14
 input codes for, 10.14, 10.20
 linearity of, 10.15
 maximum error of, 10.15
 microcomputer interfacing for, 10.15–10.17
 operation of, 10.13
 palette, 13.17–13.18
 RAMDAC, 13.17
 resolution of, 10.14
 settling time for, 10.15
 video, 13.17–13.18
- Digital video interactive (DVI), 11.43–11.44
- Direct addressing mode, 2.19–2.20
- Direct calls, 5.8–5.9, 6.4–6.5
- Direct input/output (I/O), 7.29, 7.34
- Direct jumps, 4.9–4.12, 6.14
- Direct memory access (DMA)
 80186 programmable DMA unit, 10.49, 10.51
 controller for. *See* 8237 DMA controller
 defined, 10.20
 overview of, 11.5–11.6
 principle of, 10.20
 slave boards, 15.20–15.21
- Direction flag (DF), 2.13
- Directives. *See* Assembler directives
- Directory of disks, 13.44
- Dirty bit, 15.11
- Disk caches, 11.48
- Dispatcher, 15.5
- Display refresh random-access memory (RAM), 13.7–13.8
- Displays
 alphanumeric. *See* Light-emitting diodes; Liquid-crystal displays
 CRT. *See* Cathode-ray tube displays
 drivers for, 9.35–9.36
 multiplexed, 9.26, 9.27–9.28
 static, 9.26–9.27
- Distributed processing systems, 2.4–2.6
- DIV instruction, 3.8, 6.7–6.8, 8.3, 8.8, 8.11
- Divide-by-zero interrupts, 8.3–8.8, 8.11
- Division
- 8086 instructions for, 2.11, 3.8
 8087 instructions for, 11.32
 binary, 9
 programs for, 5.32–5.37
- DL register, 2.13–2.14
- DMA. *See* Direct memory access
- Do-while structure in C, 12.26–12.27
- Documentation of programs, 3.12
- DOS
 80386 programs for, 15.36–15.37
 basic input output system (BIOS)
 for, 8.39–8.40, 11.47, 13.2–13.3, 13.26–13.21, 13.22, 14.23–14.25, 15.3–15.4
 compatibility box for, 13.39
 disk cache with, 11.48
 file control block (FCB) in, 13.45
 file handle (token) in, 13.45–13.46
 function calls in, 13.45–13.46
 Microsoft Windows and, 13.39–15.40
 random-access memory (RAM) disks with, 11.49
 terminate-and-stay-resident (TSR) programs and, 15.2–15.3
- Dot-matrix light-emitting diodes (LEDs), 9.25, 9.26
- Dot-matrix printers, 11.51–11.52
- Double-density recording for disks, 13.37–13.38
- Double-handshake input/output (I/O), 9.3–9.4
- Double indexed addressing mode, 3.14, 4.29–4.30
- Double-precision numbers, 11.35–11.36
- Doubleword (32 bits), 1.2
- Doubleword type, 3.21, 6.31
- DPSK (differential phase-shift keying) modulation, 14.14
- DQ (define quadword) directive, 6.31
- DRAM. *See* Dynamic random-access memory
- DRC (design rule checker) program, 11.40
- Drivers, 9.13–9.17
- DS (data segment) register, 2.12, 2.14, 2.16, 2.19–2.20, 4.29–4.30
- DT (define ten bytes) directive, 6.31–6.32
- DTE (data terminal equipment), 14.3, 14.9
- Dual-ported random-access memory (RAM), 13.16
- Dual-slope analog-to-digital (A/D) converters, 10.18–10.19, 10.20
- Dummy procedures, 5.24
- Dummy variables, 5.38
- DVI (digital video interactive), 13.55
- DW (define word) directive, 3.21, 6.32
- DX register, 2.13
- Dynamic random-access memory (DRAM)
 82C08 controller for, 11.12–11.14
 block diagram of, 11.10, 11.11
 burst-mode refresh, 11.12
 cache memory and, 11.17, 11.18
 characteristics of, 11.10–11.12
 column-address strobe (CAS) for, 11.11–11.12
 described, 1.17–1.18
 dynamic-mode refresh, 11.12
 error detection and correction for, 11.21–11.23
 Hamming codes and, 11.22–11.23
 hard errors for, 11.21
 microcomputer interfacing for, 11.12–11.14
 page mode for, 11.15–11.16
 parity check for, 11.21–11.22
 precharging, 11.15
 refresh controllers for, 1.18, 11.12–11.13
 row-address strobe (RAS) for 11.11–11.12
 soft errors for, 11.21
 static column mode for, 11.13–11.16
 syndrome words and, 11.23–11.24
 timing in microcomputers, 11.14
 timing waveforms for, 11.10–11.11
- EA (effective address), 2.19, 3.15–3.16, 4.29–4.30
- EBCDIC (Extended Binary-Coded Decimal Interchange Code), 1.6–1.7, 9.31–9.32
- ECCs (error detecting/correcting codes) 11.22–11.23
- EDA. *See* Electronic design automation
- EDAC (error detecting and correcting) device, 11.23–11.24
- Editor program, 3.26–3.27
- EEPROM (electrically erasable programmable read-only memory), 1.17
- Effective address (EA), 2.19, 3.15–3.16, 4.29–4.30
- EGA (enhanced graphics adaptor), 13.15, 13.17
- EISA (extended industry standard architecture) bus, 15.20–15.21

- Electrical rule checker (ERC) program, 11.40
 Electrically erasable programmable read-only memory (EEPROM), 1.17
 Electromagnetic interference (EMI), 9.40–9.41
 Electronic design automation (EDA) behavioral models for, 11.42 case design, 11.46 computer-integrated manufacturing (CIM) and, 11.47 design overview and, 11.39 design review committee and, 11.39 design for test and, 11.41, 11.47 gate-level models for, 11.42 hardware models for, 11.42–11.43 initial design, 11.39–11.40 introduced, 11.39 printed-circuit-board design, 11.46 production and test, 11.47 prototyping with simulation, 11.40, 11.42–11.45 schematic capture, 11.39–11.40, 11.41 stimulus files and, 11.43–11.44 system software and, 11.47 time steps for simulation, 11.42 Elements of arrays, 4.26–4.29 Embedded controllers 6801, 2.10 8048, 2.10 8051 family, 2.11, 10.48, 10.49 8096 family, 2.11, 10.49–10.50 80186 and 80188, 2.11, 10.49, 10.50 10.51 80960 family, 10.52 microprocessors versus, 10.47 overview of, 2.11 EMI (electromagnetic interference), 9.40–9.41 Emulators, 3.26, 3.28–3.29 Enable input, 1.14–1.15 Encoding (check) bits, 11.23 END (end program) directive, 3.26, 159 End flowchart symbol, 3.2 End-of-block (ETB) character, 14.36 End-of-text (ETX) character, 14.36 End procedure (ENDP) directive, 6.32 End program (END) directive, 3.26 End segment (ENDS) directive, 3.19–3.20, 6.32 ENDP (end procedure) directive, 6.32 ENDS (end segment) directive, 3.19–3.20, 6.32 Enhanced graphics adaptor (EGA), 13.15, 13.16 Enhanced small device interface (ESDI) standard, 13.43 Enquiry (ENQ) character, 14.36 ENTER instruction 80186/80188, 3.9, 10.52 80286, 15.15 Entry point, 3.4 Enumerated data type, 12.9 Environment (context, or state) of tasks, 15.4–15.5 EO (erasable optical) disks, 11.49 EOT (end-of-transmission) character, 14.37 EPROM (erasable programmable read-only memory), 1.17 Equate (EQU) directive, 3.19–3.20, 6.32 Erasable optical (EO) disks, 11.49 Erasable programmable read-only memory (EPROM), 1.17 ERC (electrical rule checker) program, 11.40 Error detecting/correcting codes (ECCs), 11.22–11.23 Error detecting and correcting (EDAC) device, 11.23–11.24 ES (extra segment) register, 2.12, 2.14–2.16, 4.29–4.30 ESC (escape) instruction, 44, 3.9, 6.8, Escape (ESC) instruction, 3.9, 6.8 ESDI (enhanced small device interface) standard, 13.43 ETB (end-of-block) character, 14.36 Ethernet, 14.42–14.43 ETX (end-of-text) character, 14.36 EU (execution unit) 8086, 2.12–2.13, 2.16 80286, 15.11 EVEN (align on even memory address) directive, 6.33–6.34 Even parity, 1.5, 4.13 Excess-3 binary-coded decimal (BCD) code, 4 Exclusive NOR (XNOR) gate, 1.12, 1.13 Exclusive OR (XOR) gate, 1.12, 1.13 Executing instructions, 2.7–2.9 Execution unit (EU) 8086, 2.13–2.14, 2.17 Executive programs, 10.35, 10.36 Exit point, 3.4 Expanded memory, 15.9–15.10 Expansion slots, 11.2 Expert systems, 15.43 Exponent of numbers, 11.24–11.25 Exponential instructions (8087), 11.33 Extended American Standard Code for Information Interchange (ASCII), 13.2–13.6 Extended Binary-Coded Decimal Interchange Code (EBCDIC), 1.6, 1.7, 9.24–9.25 Extended industry standard architecture (EISA) bus, 15.20–15.21 Extended (XMS) memory, 15.8 Extern (global) storage class in C, 12.35–12.36 External (EXTRN) directive, 5.32, 6.33, 6.35 External hardware synchronization instructions, 3.9–3.10 Extra segment, 3.24 Extra segment (ES) register, 2.12, 2.14–2.16, 4.29–4.30 EXTRN (external) directive, 5.32, 6.32, 6.35 FABS instruction (8087), 11.33 Factorials, 5.26–5.31 FADD instruction (8087), 11.31–11.32 Far (intersegment) calls, 5.7, 5.8–5.9, 5.15, 6.4–6.5 Far (intersegment) jumps, 4.9, 6.14–6.15 Far (intersegment) procedures, 5.31–5.37 Farm connection scheme, 15.44 FAT (file allocation table), 13.44 FBBLD instruction (8087), 11.31 FBSTP instruction (8087), 11.31 FCB (file control block), 13.45 FCHS instruction (8087), 11.33 FCLEX instruction (8087), 11.34 FCOM instruction (8087), 11.33 FCOMP instruction (8087), 11.33 FCOMPP instruction (8087), 11.33 FCS (frame check sequence), 14.39 FDDI (Fiber Distributed Data Interface) standard for networks, 14.44–14.45 FDECSTP instruction (8087), 11.35 FDISI instruction (8087), 11.34 FDIV instruction (8087), 11.32

- FDIVP instruction (8087), 11.32
 FDIVR instruction (8087), 11.32
 Feedback
 derivative, 10.33–10.34
 digital, 9.46
 integral, 10.33, 10.34
 for motors, 10.15–10.16
 negative, 10.4, 10.32
 proportional, 10.33, 10.34
 FENI instruction (8087), 11.34
 Fetching instructions, 2.6–2.9, 2.14
 FFREE instruction (8087), 11.35
 FIADD instruction (8087), 11.32
 Fiber-optic data communication, 14.20–14.23
 Fiber-optic local area networks (LANs), 14.43–14.44
 FICOM instruction (8087), 11.33
 FICOMP instruction (8087), 11.33
 FIDIV instruction (8087), 11.32
 FIDIVR instruction (8087), 11.32
 Field programmable logic array (ELA), 1.13–1.14
 Fields
 of frames, 14.39
 for statements, 2.16
 FILD instruction (8087), 11.31
 File allocation table (FAT), 13.44
 File control block (FCB), 13.45
 File handle (token), 13.45–13.46
 File servers, 14.46–14.47
 Files
 closing, 13.45–13.47
 library, 3.27
 link, 3.27, 5.31
 list, 3.23, 3.26
 object, 3.26, 5.31
 opening, 13.44
 path to, 13.44
 source, 3.26
 stimulus, 11.43–11.44
 Filter capacitors, 7.15, 7.25
 Filters
 active, 10.3, 10.6–10.7
 bandpass, 10.7
 digital. *See* Digital filters
 formant, 13.53–13.54
 high-pass, 10.3, 10.6–10.7
 low-pass, 10.3, 10.6, 10.7, 10.54
 FIMUL instruction (8087), 11.31
 FINCSTP instruction (8087), 11.34
 FINIT instruction (8087), 11.34
 Finite impulse response (FIR) algorithm, 10.55
 FIR (finite impulse response) algorithm, 10.55
 FIST instruction (8087), 11.31
 FISTP instruction (8087), 11.31
 FISUB instruction (8087), 11.31
 Fixed-point numbers, 11.24
 Fixed-port instructions, 4.17, 7.34
 Flag bits, 14.35–14.36
 Flag field of frames, 14.38
 Flag register, 2.12–2.13
 Flag set/clear instructions, 3.9
 Flag transfer instructions, 3.7
 Flags
 auxiliary carry (AF), 2.13, 4.12, 4.13
 carry (CF), 2.13, 4.12–4.13
 conditional, 2.12–2.13, 4.12–4.13
 control, 2.13
 defined, 2.13
 direction (DF), 2.13
 interrupt (IF), 2.13, 8.1–8.2, 8.6–8.9
 overflow (OF), 2.13, 4.12, 4.13, 8.9–8.10
 parity (PF), 2.13, 4.12, 4.13
 sign (SF), 2.13, 4.12, 4.13
 trap (TF), 2.13, 8.1–8.2, 8.6–8.9
 zero (ZF), 2.13, 4.12, 4.13
 Flash (parallel comparator) analog-to-digital (A/D) converters, 10.17, 10.20–10.21
 Flash electrically programmable read-only memory (flash EPROM), 1.17
 FLD1 instruction (8087), 11.34
 FLD2T instruction (8087), 11.34
 FLD instruction (8087), 11.32
 FLDCW instruction (8087), 11.34
 FLDENV instruction (8087), 11.34
 FLDL2E instruction (8087), 11.34
 FLDLG2 instruction (8087), 11.34
 FLDLN2 instruction (8087), 11.34
 FLDPI instruction (8087), 11.34
 FLDZ instruction (8087), 11.34
 Flip-flops
 as counters, 1.14–1.15
 D flip-flop, 1.14
 Float (floating-point) pointers in C, 12.16–12.18
 Float (floating-point) variables in C, 12.11
 Floating-point numbers, 11.25
 Floating point (float) pointers in C, 12.16–12.18
 Floating-point processors. *See* Math coprocessors
 Floating-point (float) variables in C, 12.11
 Floppy disks
 8272A floppy-disk controller for, 13.39–13.40
 access times for, 13.36
 formats for, 13.38–13.39
 formatting, 13.43
 hardware interfacing for, 13.39–13.40
 index holes in, 13.34, 13.35
 overview of, 13.34–13.35
 packages for, 13.35
 sizes for, 13.34
 soft-sectored, 13.38–13.39
 Flow sensors, 10.12
 Flowcharts
 CASE structure, 3.5
 comparing strings, 5.4
 data sampling, 3.2, 5.12
 described, 3.2
 downloading program, 14.27
 factorials, 5.27
 IF-THEN-ELSE structure, 3.5, 4.16, 4.20
 IF-THEN structure, 3.5
 keyboard input, 9.20–9.21
 microcomputer-based industrial process-control system, 10.34, 10.35
 microcomputer-based scale, 10.21–10.22
 REPEAT-UNTIL structure, 3.5, 4.26, 4.27
 sequence structure, 3.5
 strobed input, 4.25
 symbols for, 3.2–3.3
 WHILE-DO structure, 3.5, 4.23
 FM (frequency modulation) coding for disks, 13.37–13.38
 FMUL instruction (8087), 11.32
 FMULP instruction (8087), 11.32
 FNCLX instruction (8087), 11.34
 FNDISI instruction (8087), 11.34
 FNENI instruction (8087), 11.34
 FNINT instruction (8087), 11.34
 FNOP instruction (8087), 11.35
 FNSAVE instruction (8087), 11.34
 FNSTCW instruction (8087), 11.34
 FNSTENV instruction (8087), 11.34
 FNSTSW instruction (8087), 11.34
 FOR-DO loop, 3.6, 4.30, 12.27, 12.33
 FOR loop in C, 12.27–12.31
 Force transducers, 10.10–10.13, 10.20, 10.21, 10.22

- Formal arguments (parameters) of functions, 12.32
- Formant filters, 13.52–13.53
- Forward jump, 4.11
- Four-phase stepper motors, 9.41–9.42
- Fourier series, 10.52–10.53
- FPATAN instruction (8087), 11.34
- FPLA (field programmable logic array), 1.13–1.14
- FPREM instruction (8087), 11.33
- FPTAN instruction (8087), 11.33
- Frame check sequence (FCS), 14.39
- Frames of messages, 14.38–14.39
- Frequency-domain description, 10.53, 10.54
- Frequency modulation (FM) coding for disks, 13.37–13.38
- Frequency-shift keying (FSK) modulation, 14.14–14.15
- Fricatives, 13.53
- FRNDINT instruction (8087), 11.33
- FRSTOR instruction (8087), 11.34
- FSAVE instruction (8087), 11.34
- FSCALE instruction (8087), 11.33
- FSK (frequency-shift keying) modulation, 14.13–14.14
- FSQRT instruction (8087), 11.33
- FST instruction (8087), 11.31
- FSTCW instruction (8087), 11.34
- FSTENV instruction (8087), 11.34
- FSTP instruction (8087), 11.37
- FSTS instruction (8087), 11.40
- FSUB instruction (8087), 11.38
- FSUBP instruction (8087), 11.38
- FSUBR instruction (8087), 11.38
- FSUBRP instruction (8087), 11.38
- FTST instruction (8087), 11.39
- Full-duplex communication, 14.1–14.2
- Function storage classes in C, 12.35–12.36
- Functions of programs, 3.12
- Fundamental frequency, 10.53
- Fusible matrixes, 1.14
- FWAIT instruction (8087), 11.38
- FX2M1 instruction (8087), 11.37
- FXAM instruction (8087), 11.37
- FXCH instruction (8087), 11.36
- FXTRACT instruction (8087), 11.37
- FYL2X instruction (8087), 11.37–11.38
- FYL2XP1 instruction (8087), 11.38
- Gain-bandwidth product, 10.5
- Gate-level models, 11.42–11.43
- Gates. *See Logic gates*
- Gateways (bridges) for networks, 14.45
- General-purpose central processing unit (CPU), 2.10
- General-purpose registers, 2.13–2.14
- Generators
- audio-tone, 8.27
 - ramp, 10.3, 10.6
 - square-wave, 8.27–8.28
 - timed interrupt, 8.25–8.26
- Gigabyte (unit), 2.10
- GLOBAL directive, 6.33
- Global (extern) storage class in C, 12.35–12.36
- GO command, 4.5
- Graphics
- color, 13.11–13.12
 - high-resolution, 13.29–13.30
 - monochrome, 13.11
- Graphics processors, 13.29–13.30
- Gray code, 1.4–1.5, 9.44–9.45, 14.15
- GROUP directive, 6.33
- GUIs (graphical user interfaces), 15.39–15.40
- Half-duplex communication, 14.1–14.2
- Hall effect keyswitches, 9.20–9.21
- Halt (HLT) instruction, 3.9, 6.8
- Halt state, 3.9, 6.8
- Hamming codes, 11.22–11.23
- Hand-coding programs, 2.19, 3.10–3.12, 3.18, 3.19, 4.5
- Hard disks
- access times for, 13.36
 - backup storage for, 11.49
 - cylinders of, 13.36–13.37
 - enhanced small device interface (ESDI) standard for, 13.43
 - formatting, 13.44
 - hardware interfacing for, 13.43–13.44
 - interface software for, 13.45–13.46
 - interleave factor for, 13.42–13.43
 - logical drives for, 13.43
 - overview of, 13.36–13.37
 - parking zone for, 13.37
 - partitioning, 13.45
 - small computer systems interface (SCSI) standard for, 13.42–13.43
 - ST-506 standard for, 13.43
 - Winchester, 13.37
- Hard errors, 11.21
- Hardware, defined, 2.7
- Hardware interrupts, 3.9, 6.4, 6.28, 7.7, 8.1, 8.2, 8.8–8.12
- Hardware models, 11.42–11.43
- Hardware-triggered strobes, 8.28–8.29
- Hardwired matrixes, 14
- Harmonic frequencies, 10.53
- Harvard architecture, 10.56
- HDLC (high-level data link control) protocol, 14.39–14.40
- Head pointer, 14.32–14.33
- Heap area of memory, 13.25–13.26
- Hercules display adaptor, 13.14–13.15
- Hexadecimal (base-16) numbers
- in data statements, 3.22
 - defined, 1.3, 1.10–1.11
- Hierarchical charts, 5.7
- High byte, 2.19
- High-level data link control (HDLC) protocol, 14.37–14.39
- High-level language interface instructions (80186/80188), 3.9
- High-level languages, 2.17
- High-pass filters, 10.3, 10.6–10.7
- High Performance File System (HPFS), 15.39
- High-power buffers, 10.15–10.16
- High-resolution graphics, 13.29–13.30
- HLT (halt) instruction, 3.9, 6.8
- Hysteresis, 10.2
- IA-32 family, 16.1
- IA-64 architecture, 16.2
- ICs. *See Integrated circuits*
- IDIV instruction, 3.8, 6.8–6.9, 8.3, 8.8, 8.12
- IF (interrupt flag), 2.13, 8.1–8.3, 8.8–8.12
- If-else structure in C, 12.24–12.25
- IF-THEN-ELSE structure, 3.4, 3.5, 4.15–4.20, 12.24–12.25
- IF-THEN structure, 3.4, 3.5, 4.15, 12.24–12.25
- IGBTs (isolated-gate bipolar transistors), 9.39–9.40
- IIR (infinite impulse response) algorithm, 10.55
- ILDs (infrared injection laser diodes), 14.21–14.22
- Immediate addressing mode, 2.18
- Impact printers, 11.51
- Impedance, input, 10.4
- IMUL instruction

- 80186/80188, 3.8, 6.9–6.10, 10.52
 80286, 15.15
IN instruction, 3.7, 3.18, 4.16, 6.10, 7.34
In-line code, 5.37
INC instruction, 3.7, 6.10
INCLUDE directive, 6.33
 Incremental shaft encoders, 9.44–9.45
 Index field of disks, 13.38–13.39
 Index holes in disks, 13.34–13.35
 Index of refraction, 14.21–14.22
 Index registers, 2.13, 2.18
 Index scaling, 15.35
 Indexes of arrays, 4.30
 Indirect calls, 5.8, 5.9, 6.4–6.5
 Indirect jumps, 4.9, 6.14
 Inductive kick, 9.40
 Industrial process control. *See also*
 Microcomputer-based industrial
 process-control system
 data acquisition system (DAS) for,
 10.20, 10.34, 10.35–10.37
 overview of, 10.31–10.34
 proportional integral derivative (PID)
 control loops and, 10.34, 10.35, 10.36
 residual error and, 10.34
 servo control and, 10.32–10.33
 set points and, 10.32–10.33
 Industry standard architecture (ISA)
 bus, 15.20
 Infinite impulse response (IIR) algorithm,
 10.55
 Infrared injection laser diodes (ILDs),
 14.21–14.22
 Infrared light-emitting diodes (LEDs),
 8.14–8.15
 Initialization
 instructions for, 3.10, 4.2, 4.3, 8.4
 of programmable peripheral devices,
 8.20–8.21, 8.33–8.39
 of segment registers, 3.25
 Ink-jet printers, 11.52–11.53
 Input flowchart symbol, 3.2
 Input impedance, 10.4
 Input/output (I/O)
 Basic Input Output System (BIOS)
 for, 8.39–8.41, 13.3–13.7,
 13.21–13.22, 13.23, 14.4, 14.23,
 14.25, 15.3
 direct, 7.29, 7.35
 double-handshake, 9.3–9.4
 drivers for, 9.13
 interrupt, 8.12–8.15
 memory-mapped, 7.29, 7.35
 microcomputer, 2.6–2.9
 polled, 8.11–8.12, 8.14
 ports for. *See* Ports
 read signal for, 2.7–2.9
 simple, 9.1–9.2
 simple strobe, 9.1–9.2
 single-handshake, 9.1
 write signal for, 2.7–2.9
 Input ports
 described, 2.6–2.9
 IN instruction for, 3.7, 3.18, 4.16,
 6.10, 7.34
 program for reading, 3.10–3.12
 INS/INSB/INSW instructions (80186/
 80188), 3.8
 INS instruction (80286), 15.14–15.15
 Instruction cycle, 7.4
 Instruction pointer (IP) register, 2.6, 2.13–
 2.16
 Instruction unit (IU) (80286), 15.11
 Instructions
 decoding, 2.7–2.9
 executing, 2.7–2.9
 fetching, 2.7–2.9, 2.13
 overhead, 4.32
 pipelined, 2.14
 Instrument prototyping, 10.46–10.47
 Instrumentation operational amplifiers
 (op amps), 10.2, 10.3
 Int array (integer array) pointers in C,
 12.13–12.14
 INT instruction, 3.9, 3.18, 6.10–6.11, 8.1,
 8.9, 8.11, 8.39–8.40
 Int (integer) pointers in C, 12.13
 Int (integer) variables in C, 12.9–12.10
 Integer (int) pointers in C, 12.11–12.12
 Integer (int) variables in C, 12.10–12.11
 Integral feedback, 10.33, 10.34
 Integrated circuits (ICs)
 buffers using, 9.37–9.38
 handling, 7.43–7.44
 RAMDACs, 3.17
 on schematic diagrams, 7.25
 troubleshooting, 7.44–7.45
 Integrated Development Environment
 for C, 12.3–12.7
 Integrated services digital network
 (ISDN), 14.19–14.20
 Integrators, 10.2, 10.3
 Interlaced scanning, 13.5–13.6
 Interleave factor, 13.42–13.43
 Internal addresses, 8.20
 International Standards Organization
 (ISO)
 high-level data link control (HDLC)
 protocol, 14.38–14.40
 open systems interconnection (OSI)
 model, 14.41–14.42
 Interpreter program, 2.17
 Interrupt flag (IF), 2.13, 8.1–8.3, 8.8–8.11,
 8.12
 Interrupt input/output (I/O), 8.12–8.14
 Interrupt-pointer (interrupt-vector) table,
 8.2
 Interrupt-service procedures, 5.25–5.26,
 7.7, 8.1–8.6, 8.12–8.13, 8.15
 Interrupts
 8086 response to, 8.1–8.2
 80286, 15.12–15.13
 available, 8.2
 Basic Input Output System (BIOS)
 procedure calls with, 8.39–8.41
 breakpoint, 8.2, 8.9
 CLI instruction for, 3.9, 6.4, 8.10
 for counting, 8.14–8.15, 8.16
 dedicated, 8.2
 divide-by-zero, 8.3–8.8, 8.10
 hardware, 3.9
 INT instruction for, 3.9, 3.18
 INTO instruction for, 3.9
 IRET instruction for, 3.9
 maskable (INTR), 3.9, 6.4, 6.28,
 8.1–8.10
 nonmaskable (NMI), 6.4, 7.7, 8.1–8.2,
 8.9–8.10, 8.12
 overflow, 8.2, 8.9, 8.11
 PIC for, *See* 8259A priority interrupt
 controller
 priority of, 8.11
 for real-time clocks, 8.16–8.17
 reserved, 8.3
 single-step, 8.2, 8.8, 8.12
 STI instruction for, 3.9, 6.28, 8.10
 timed interrupt generators, 8.25–8.26
 for timing, 8.15–8.18
 Intersegment (far) calls, 5.7, 5.9–5.10, 5.15
 Intersegment (far) jumps, 4.10, 6.14–6.15
 Intersegment (far) procedures, 5.30–5.37
 INTO instruction, 3.9, 6.11, 8.9, 8.12
 INTR (maskable) interrupts, 3.9, 6.4, 6.28,
 7.7, 8.1, 8.10–8.11
 Intrasegment (near) calls, 5.7, 5.8, 5.13–
 5.14, 6.4
 Intrasegment (near) jumps, 4.9–4.12,
 6.13–6.14

- Intrasegment (near) procedures, 5.9–5.14
 INVD instruction (80486), 15.42–15.43
 Inverters, 1.12
 Inverting buffers, 1.12
 Inverting operational amplifiers (op amps), 10.3, 10.4
 IP (instruction pointer) register, 2.7, 2.13, 2.14–2.15
 IRET instruction, 3.9, 6.11, 8.3, 8.12
 ISA (industry standard architecture) bus, 15.20
 ISDN (integrated services digital network), 14.20–14.21
 ISO. See International Standards Organization
 Isolated-gate bipolar transistors (IGBTs), 9.39–9.40
 Itanium, 16.2
 EPIC (Explicit Parallel Instruction Computing), 16.19
 Itanium 2, 16.2
 Iteration control instructions, 3.9
 Iteration operations, 3.2–3.6
 JA instruction, 3.9, 4.14, 6.11
 Jack (J) symbols, 7.25
 JAE instruction, 3.9, 4.14–4.15, 6.11
 JB instruction, 3.9, 4.14, 6.12
 JBE instruction, 3.9, 4.14, 6.12
 JC instruction, 3.9, 4.14, 6.12
 JCXZ instruction, 3.9, 4.31, 6.12
 JE instruction, 3.9, 4.14, 6.12
 JG instruction, 3.9, 4.14, 6.12–6.13
 JGE instruction, 3.9, 4.14, 6.13
 JL instruction, 3.9, 4.14, 6.13
 JLE instruction, 3.9, 4.14, 6.13
 JMP instruction, 3.8, 4.9–4.12, 6.13–6.14
 JNA instruction, 3.9, 4.14, 6.12
 JNAE instruction, 3.9, 4.14, 6.12
 JNB instruction, 3.9, 4.14, 6.12
 JNBE instruction, 3.9, 4.14, 6.12
 JNC instruction, 3.9, 4.14, 6.12
 JNE instruction, 3.9, 4.14, 6.12
 JNG instruction, 3.9, 4.14, 6.13
 JNGE instruction, 3.9, 4.14, 6.13
 JNL instruction, 3.9, 4.14, 6.13
 JNLE instruction, 3.9, 4.14, 6.13
 JNO instruction, 3.9, 4.14, 6.14
 JNP instruction, 3.9, 4.14, 6.14
 JNS instruction, 3.9, 4.14, 6.14
 JNZ instruction, 3.9, 4.14, 6.15
 JO instruction, 3.9, 4.14, 6.15
 JP instruction, 3.9, 4.14, 6.15
 JPE instruction, 3.9, 4.14, 6.15
 JPO instruction, 3.9, 4.14, 6.14
 JS instruction, 3.9, 4.14, 6.15
 Jukebox optical disk systems, 11.50
 Jump table, 4.20
 Jumps
 backward, 4.9–4.12
 conditional, 4.8, 4.14, 4.23–4.24, 4.30–4.31
 direct, 4.9–4.12, 6.14
 far (intersegment), 4.9, 6.13–6.15
 forward, 4.11
 indirect, 4.9, 6.14
 near (intrasegment), 4.9–4.12, 6.14–6.15
 short, 4.9, 4.12, 4.14, 4.23–4.24
 unconditional, 4.8, 4.9–4.12
 JZ instruction, 3.9, 4.14, 6.13
 Kbyte (kilobyte), 2.14
 Keyboard input, 4.23–4.26, 8.13, 8.15, 8.35–8.39, 19.39–19.40, 13.1–13.2
 Keyboards
 circuit connections, 9.19–9.20
 compare code conversion technique for, 9.22–9.23
 debouncing, 9.19–9.23
 dedicated microprocessor encoders for, 9.23–9.24
 detecting keypress on, 9.19–9.23
 EBCDIC to ASCII conversion for, 9.25–9.26
 encoding keypress on, 9.19–9.23
 hardware interfacing for, 9.23–9.24
 keyswitch types for, 9.17–9.18
 N-key rollover for, 9.32
 software interfacing for, 9.20–9.22
 two-key lockout for, 9.22–9.23
 two-key rollover for, 9.23–9.24
 XLAT code conversion technique for, 9.24–9.25
 Keypad interfacing, 9.27–9.35
 Kilobyte (Kbyte), 2.14
 LABEL directive, 6.34
 Label field, 2.16
 Labels in programs, 3.10, 3.11, 3.24
 LAHF instruction, 3.7, 6.15
 LANs. See Local area networks
 LAR instruction (80286), 15.16
 Laser printers, 11.51
 Latches
 D latch, 1.14–1.15
 defined, 1.14
 Latency time for disks, 13.36
 Lathe, 9.7–9.9
 LCDs. See Liquid-crystal displays
 LDS instruction, 3.7, 6.15
 LEA instruction, 3.7, 6.15–6.16
 Least significant bit (LSB), 1.1
 Least significant digit (LSD), 1.2, 1.3
 LEAVE instruction
 80186/80188, 3.9, 10.52–10.54
 80286, 15.15
 LEDs. See Light-emitting diodes
 LENGTH operator, 6.34
 LES instruction, 3.7, 6.16
 LGDT instruction (80286), 15.15
 Library file, 3.27
 LIIDT instruction (80286), 15.15
 Light-emitting diodes (LEDs)
 7-segment, 1.5, 9.26–9.35
 7-segment display code for, 1.5
 18-segment, 9.26, 9.35
 8279 controller for. See 8279 dedicated display controller
 described, 9.26
 directly driven (static), 9.26–9.27
 dot-matrix, 9.26, 9.35
 infrared, 8.14, 8.16
 in optical couplers, 8.25
 software-multiplexed, 9.26–9.27, 9.28
 Light sensors, 10.8–10.9
 LIM/EMS (Lotus-Intel-Microsoft Expanded Memory Standard), 15.8
 Linear predictive coding (LPC), 13.53
 Linear ramp, 10.6
 Linear variable differential transformers (LVDTs), 10.12–10.13
 Link file, 3.27, 5.31
 Link map, 3.27
 Linker program, 3.27
 Liquid-crystal displays (LCDs)
 backplane drive of, 9.36
 described, 9.25
 dynamic scattering type, 9.35
 field-effect type, 9.35
 microcomputer interfacing of, 9.36–9.37
 operation of, 9.36
 reflective-type, 13.32
 screen-type, 13.31–13.32
 transmission-type, 13.32
 List file, 3.23, 3.27
 LLDT instruction (80286), 15.15
 LMSW instruction (80286), 15.15

Load cells, 10.11, 10.12, 10.21, 10.22, 10.23
 Local area networks (LANs)
 10BaseT (thin Ethernet), 14.43
 application example of, 14.44–14.47
 backbones for, 14.44, 14.46
 bridges (gateways) for, 14.44
 distributed processing systems and, 2.4–2.6
 Ethernet, 14.42–14.43
 Fiber Distributed Data Interface (FDDI) standard for, 14.44–14.45
 fiber-optic, 14.45–14.46
 file server for, 14.46–14.47
 overview of, 14.40
 print server for, 14.45–14.46
 protocols for, 14.39, 14.40–14.41
 software overview for, 14.46–14.47
 topologies for, 14.40–14.41
 Location counters, 5.4–5.5
 Locator program, 3.27
 LOCK instruction, 3.9, 6.16
 LODS/LODSB/LODSW instructions, 3.8, 6.16
 Logarithmic instructions (8087), 11.34–11.35
 Logic analyzers
 block diagram of, 7.8
 clock qualifier for, 7.11–7.12
 display formats of, 7.9
 external clock for, 7.8–7.9
 internal clock for, 7.8–7.9
 memory access time measurement with, 7.11
 operation of, 7.8–7.9
 overview of, 7.7–7.8
 trace data with, 3.28, 7.9, 7.10–7.12
 triggering of, 7.9–7.11
 troubleshooting perspective for, 7.44–7.45
 word recognizer for, 7.9, 7.11
 Logic arrays, 1.14
 Logic gates
 AND gate, 1.13
 exclusive NOR (XNOR) gate, 1.13, 1.14
 exclusive OR (XOR) gate, 1.12, 1.13
 NAND gate, 1.13
 NOR gate, 1.12
 OR gate, 1.12
 Logical addresses, 18.8–18.20
 Logical drives, 13.45
 Logical instructions, 3.8

Logical operators in C, 12.21–12.22
 Logical segment, 3.19–3.20, 3.24
 LOOP instruction, 3.9, 4.30, 4.31, 6.17–6.18
 Loop networks, 14.40–14.41
 LOOPE instruction, 3.9, 4.31, 6.17
 LOOPNE instruction, 3.9, 4.31, 6.17
 LOOPNZ instruction, 3.9, 4.31, 6.17
 Loops
 delay, 4.31–4.33, 5.10–5.13
 FOR-DO, 3.6, 4.30
 structure for, 3.5, 3.6, 4.30–4.31
 LOOPZ instruction, 6.17
 Lotus-Intel-Microsoft Expanded Memory Standard (LIM/EMS), 15.8
 Low byte, 2.19
 Low-pass filters, 10.3, 10.6, 10.7, 10.54
 LPC (linear predictive coding), 13.53
 LSB (least significant bit), 1.1
 LSD (least significant digit), 1.2, 1.3
 LSL instruction (80286), 15.16
 LSS instruction (80386), 15.36
 LTR instruction (80286), 15.16
 LVDTs (linear variable differential transformers), 10.12–10.13
 Machine cycle, 7.4
 Machine language, 2.16
 Machines, computer numerical control (CNC), 9.7–9.9
 Macros. See Assembler macros
 Magnetic disks
 access times for, 13.34–13.35
 address marks on, 13.38–13.39
 boot record of, 13.44
 clusters on, 13.44
 cyclic redundancy characters (CRC) on, 13.38–13.39
 data bit formats for, 13.35–13.38
 date field of, 13.38–13.39
 directory of, 13.44
 DOS function calls for, 13.44–13.45
 double-density recording for, 13.37–13.38
 error detection for, 13.38–13.39
 file allocation table (FAT) for, 13.44
 floppy. See Floppy disks
 frequency modulation (FM) coding for, 13.37–13.38
 hard. See Hard disks
 ID fields of, 13.38–13.39
 index field of, 13.38–13.39
 latency time for, 13.36
 modified frequency modulation (MFM) coding for, 13.37–13.38
 nonreturn-to-zero (NRZ) coding for, 13.37
 run-length-limited (RLL) coding for, 13.38, 13.42
 seek time for, 13.36
 single-density recording for, 13.37, 13.38
 tracks of, 13.35, 13.38–13.39
 types of, 13.35
 Magnetic tapes, 13.35
 Magneto-optical (MO) recording, 11.50
 Mail, electronic, 14.41
 Mainframes, 2.1
 Mainline programs, 10.35, 10.36
 Manchester code, 14.43
 Mantissa (significand) of numbers, 11.25
 Marking state, 14.2
 Maskable (INTR) interrupts, 3.9, 6.4, 6.28, 7.7, 8.1, 8.10–8.11
 Masking bits, 4.6–4.7
 Master device, 8.33
 Math coprocessors
 8087. See 8087 math coprocessor
 80287, 15.12
 80486 built-in, 15.1
 Math library functions in C, 12.41–12.42
 MAU (multistation access unit), 14.44–14.45
 Maximum mode, 7.6
 Mbyte (megabyte), 2.14
 MCA (MicroChannel Architecture) bus
 MCGA (multicolor graphics array), 13.15
 Mechanical keyswitches, 9.18–9.20
 Mechanical relays, 9.40
 Megabyte (Mbyte), 2.14
 MEGAFLOPS (million floating-point operations per second), 13.30
 Membrane keyswitches, 9.18–9.20
 Memory
 8086 organization of, 7.30–7.33, 5.81
 access time measurements, 7.11
 bank-switched, 15.7–15.8
 blocks of, 15.7
 cache. See Cache memory
 DMA for. See Direct memory access
 expanded, 15.7–15.8
 extended (XMS), 15.8
 heap area of, 13.26, 13.28
 Lotus-Intel-Microsoft Expanded Memory Standard (LIM/EMS), 15.8
 microcomputer use of, 2.6–2.9

- multiuser/multitasking operating system management of, 15.7–15.11 named, 5.19, 5.21 nonvolatile, 1.16 purposes of, 2.6 RAM. *See* Random-access memory read signal for, 2.7–2.9 ROM. *See* Read-only memory segmentation of, 2.14–2.16, 2.18–2.19 types of, 2.6 virtual, 15.9–15.11 volatile, 1.17 write signal for, 2.7–2.9 Memory-management unit (MMU), 15.7, 15.9–15.11 Memory-mapped input/output (I/O), 7.29, 7.34 Memory models for C, 12.44 Metal-oxide-semiconductor field-effect transistors (MOSFETs), 9.39–9.40 MFLOPS (million floating-point operations per second), 13.30 MFM (modified frequency modulation) coding for disks, 13.37–13.38 MicroChannel Architecture (MCA) bus, 15.21–15.22 Microcode, 2.10 Microcomputer-based industrial process-control system 8086 assembly language program for, 10.38–10.46 block diagram of, 10.34–10.35 flowchart for, 10.35, 10.36 hardware for, 10.35, 10.36–10.37 overview of, 10.35, 10.36 Microcomputer-based instrument prototyping, 10.46–10.47 Microcomputer-based scale 8086 assembly language programs for, 10.23–10.32 algorithm for, 10.23–10.24 flowchart for, 10.23–10.24 input circuitry for, 10.22–10.23 overview of, 10.21–10.22 Microcomputer-controlled lathe, 9.7–9.8 Microcomputer development system, 2.17, 3.26 Microcomputers 8086-based. *See* 8086-based microcomputers: SDK-86 address decoders for, 7.26–7.27 block diagram of, 2.6 buses of, 2.6–2.9 CPU of, 2.6–2.9 I/O section of, 2.6–2.9 introduced, 2.2, 2.4 memory section of, 2.6 motherboards for, 11.1, 11.2–11.3 port decoders on, 7.29–7.30, 7.34–7.36 random-access memory (RAM) address decoding on, 7.27–7.28, 7.33–7.34 read-only memory (ROM) address decoding on, 7.26–7.27, 7.32–7.33 three-step program for, 2.7–7.29 troubleshooting, 7.42–7.46 Microcontrollers. *See also* Embedded controllers overview of, 2.11 Microprocessors. *See also* Central processing unit 4-bit, 2.9, 2.10 8-bit, 2.10, 2.11 16-bit, 2.10, 2.11 32-bit, 2.10 ALU categorization of, 2.9 complex instruction set computer (CISC), 13.30 defined, 2.1 embedded controllers versus, 10.49 evolution of, 2.9–2.10 reduced instruction set computer (RISC), 13.31 Scalable Processor Architecture (SPARC) Microsoft Windows, 15.39–15.40 Microstepping, 9.44–9.45 Million floating-point operations per second (MEGAFLOPS or MFLOPS), 13.30 Minicomputers, 2.1, 2.2 Minimum mode, 7.6, 7.39–7.42 Mixed-mode simulators, 11.43–11.44 Mixers, 10.3, 10.5–10.6 MMU (memory-management unit), 15.7, 15.10–15.12 MMX – multimedia extensions, 16.2 MMX technology, 16.1, 16.7 Mnemonics for instructions, 2.16, 3.10, 3.11 MO (magneto-optical) recording, 11.50 MOD bit patterns, 3.13–3.15 Mode words. *See* Control words Modems amplitude modulation (AM) for, 14.13, 14.14–14.15 answer, 14.17 called, 14.17–14.18 calling, 14.2–14.3 defined, 14.2–14.3 frequency-shift keying (FSK) modulation for, 14.14–14.15 handshake sequence for, 14.17–14.18 hardware overview of, 14.16–14.17 high-speed transmission problems with, 14.16 introduction to, 14.14 null, 14.11 originate, 14.17 phase-shift keying (PSK) modulation for, 14.14–14.15 RS-232C connections for, 14.9–14.12 XMODEM protocol for, 14.39 Modes of fibers, 14.23 Modified frequency modulation (MFM) coding for disks, 13.37–13.38 Modulator-demodulators. *See* Modems Modules of programs, 3.3, 3.27, 5.31 Monitor program, 3.12, 3.28 Monochrome cathode-ray tube (CRT) displays, 13.6, 13.7–13.10, 13.11 MOSFETs (Metal-oxide-semiconductor field-effect transistors), 9.39–9.40 Most significant bit (MSB), 1.1, 1.2, 2.11, 2.12 Most significant digit (MSD), 1.2, 1.3 Motherboards, 11.1–11.2 Motors absolute shaft encoders for, 9.44–9.45 digital-to-analog (D/A) converters and, 10.14–10.15 drivers for, 9.39–9.40 feedback for, 10.14–10.15 incremental shaft encoders for, 9.44–9.45 optical shaft encoders for, 9.43–9.45 servo control of, 10.32–10.33 stepper, 9.41–9.43 Mouse devices, 13.32–13.33 MOV instruction, 3.7, 3.14–3.15, 6.17–6.18 Move and expand instructions (80386), 15.36 MOVS/MOVSB/MOVSW instructions, 3.8, 6.18 MOVSX instruction (80386), 15.36 MOVZX instruction (80386), 15.36

- MSB (most significant bit), 1.1, 1.2, 2.11, 2.12
 MSD (most significant digit), 1.2, 1.3
 MUL instruction, 3.7, 6.18–6.19
 Multicolor graphics array (MCGA), 447
 Multilevel simulators, 11.43
 Multimode fibers, 14.22
 Multiplexed displays, 9.26–9.27, 9.28
 Multiplexers (data concentrators), 14.40
Multiplication
 8086 instructions for, 2.10, 3.7
 8087 instructions for, 11.32
 binary, 1.9–1.10
 programs for, 3.19–3.26
 Multiprocessing systems, 2.4–2.6
 Multistation access unit (MAU), 14.44–14.45
 Multitasking systems, 2.3–2.4, 2.20
Multiuser/multitasking operating systems
 accessing resources with, 15.5
 defined, 15.2
 environment preservation for, 15.5–15.6
 layers of, 15.6
 memory management for, 15.7–15.11
 protection in, 15.5–15.6
 scheduling for, 15.2–15.4
 tasks of, 15.2
 Mutual exclusion of tasks, 15.6
MVDM (Multiple Virtual DOS Machines), 15.39
N-key rollover, 9.32–9.33
NAK (negative acknowledge) character, 14.36
NAME directive, 6.34
Named addresses, 3.24
Named constants, 3.19–3.20
Named memory, 5.19, 5.21
Named variables, 3.21–3.24
NAND gate, 1.13
National PACE microprocessors, 2.10
Near (intrasegment) calls, 5.7, 5.8, 5.13–5.15, 6.4
Near (intrasegment) jumps, 4.9–4.12, 6.13–6.14
Near (intrasegment) procedures, 5.9–5.14
NEG instruction, 3.7, 6.19
Negative feedback, 10.4, 10.32
Nested procedures, 5.6–5.7
Netlist (wiring list) program, 11.40
Network layer (OSI model), 14.41–14.42
Networks
 10BaseT (thin Ethernet), 14.43
 application example of, 14.44–14.47
 broadband bus (tree-structured), 14.20–14.21
 common-bus, 14.20–14.21
 Ethernet, 14.42–14.43
 integrated services digital network (ISDN), 14.19–14.20
 LANs. *See Local area networks*
 loop, 14.40–14.41
 ring, 14.40–14.41
 star, 14.40
 token-passing ring, 14.40, 14.41, 14.43–14.45, 14.47
 topologies for, 14.40–14.43
Net Burst Architecture, 16.1, 16.12
Trace Cache, 16.12
Hyper Thread (HT) Technology, 16.14
Nibble (4 bits), 1.1
NMI (nonmaskable) interrupts, 6.4, 7.7, 8.1, 8.2, 8.8–8.9, 8.12
No operation (NOP) instruction, 3.9–3.10
Noninterlaced scanning, 13.5–13.6
Noninterruptible power supply (NPS), 11.14
Noninverting buffers, 1.12, 10.4
Noninverting operational amplifiers (op amps), 10.3–10.5
Nonmaskable (NMI) interrupts, 6.4, 7.7, 8.1, 8.2, 8.8–8.9, 8.12
Nonreentrant procedures, 5.20
Nonreturn-to-zero (NRZ) coding, 13.37
Nonvolatile memory, 1.16
NOP (no operation) instruction, 3.9, 6.20
NOR gate, 1.12
Normalizing numbers, 11.25
NOT instruction, 3.8, 6.20
NPS (noninterruptible power supply), 11.14
NRZ (nonreturn-to-zero) coding, 13.37
Null modems, 14.11
Number systems. *See Binary numbers; Decimal numbers; Hexadecimal numbers*
Object file, 3.26
Odd parity, 1.5, 4.13
OF (overflow flag), 2.13, 4.12, 4.14, 8.9
Off-page connector flowchart symbol, 3.3
OFFSET operator, 6.34
Offsets (displacements) of addresses, 2.14–2.15, 2.18, 4.27–4.30, 15.10
On-off (bang-bang) control, 10.46
One-shots, 8.24–8.25
Onionskin diagram, 15.7
Op amps. *See Operational amplifiers*
Opcode field, 2.16
Opcodes (operation codes), 2.16, 3.10, 3.11, 3.13
Open-loop gain, 10.4, 10.5
Open systems interconnection (OSI) model, 14.41–14.42
Opening files, 13.46–13.48
Operand field, 2.16
Operands, 2.16, 3.10, 3.11
Operating systems
Basic Input Output System (BIOS) of, 8.39–8.40, 13.2–13.6, 13.19–13.21, 13.22, 14.23–14.34
DOS. *See DOS*
multiuser/multitasking. *See Multiuser/multitasking operating systems*
OS/2, 15.18–15.19
Operation codes (opcodes), 2.16, 3.10, 3.11, 3.13
Operation flowchart symbol, 3.2
Operational amplifiers (op amps)
 as active filters, 10.3, 10.6–10.7
 as adders (mixers), 10.3, 10.4–10.5
 characteristics of, 10.2
 as comparators, 10.3, 10.4
 differential, 10.3, 10.5
 as differentiators, 10.3, 10.6
 instrumentation, 10.3, 10.6
 as integrators (ramp generators), 10.3, 10.6
 inverting, 10.3, 10.4
 noninverting, 10.3, 10.4–10.5
 saturation of, 10.6
 voltage gain of, 10.1, 10.2
Operator precedence in C, 12.22–12.23
Optical couplers, 8.25
Optical disks, 11.49–11.50
Optical motor shaft encoders, 9.44–9.45
Optical read-only memory (OROM), 11.49
Optical scanners, 13.34
OR gate, 1.12
OR instruction, 3.8, 4.7–4.8, 6.20
OR matrixes, 1.14
Originate (ORG) directive, 6.34

- Originate modem, 14.17
 OROM (optical read-only memory), 11.49
 OUT instruction, 3.7, 4.16, 6.21, 7.34
 Output flowchart symbol, 3.2
 Output library functions in C, 12.40
 Output ports
 described, 2.6–2.9
 OUT instruction for, 3.7, 4.16, 6.21, 7.34
 OUTS instruction (80286), 15.15
 OUTS/OUTSB/OUTSW instructions (80186/80188), 3.8, 10.52
 Overdamped response, 10.32
 Overflow
 numeric, 1.9, 4.13
 stack, 5.20–5.24
 Overflow flag (OF), 2.11, 2.12, 4.12, 4.13, 8.9
 Overhead, 4.32
 Overlays, 15.7
 Overscan, 13.10
 Overshoot, 10.32–10.33
 PACE microprocessors, 2.10
 Packed binary-coded decimal (BCD) code, 4.5–4.8, 5.17–5.23, 11.24–11.30
 Packed pixel storage, 13.12–13.13
 Packets of data, 14.41
 Paddle wheels, 10.13
 Page printers, 11.51
 PAL (programmable array logic), 1.13
 Palette digital-to-analog (D/A) converters, 13.17–13.18
 Palettes, 13.12
 Paper tape readers, 9.7–9.8
 Parallel comparator (flash) analog-to-digital (A/D) converters, 10.17, 10.20–10.21
 Parallel data transfer. *See also* 8255A
 programmable parallel port:
 Centronics parallel interface
 double-handshake input/output (I/O), 9.2–9.3
 simple input/output (I/O), 9.1–9.2
 simple strobe input/output (I/O), 9.1–9.2
 single-handshake input/output (I/O), 9.9
 Parameter passing
 by reference, 12.14
 by value, 12.13
 in named memory, 5.19, 5.21
 in registers, 5.17, 5.18
 summary of, 5.24
 to macros, 5.38
 using pointers, 5.20–5.23
 using the stack, 5.23–5.24
 Parameters
 actual arguments (parameters) of functions, 12.32
 defined, 5.17
 formal arguments (parameters) of functions, 12.33
 Parity
 defined, 1.5
 even, 1.5, 4.13
 odd, 1.5, 4.13
 Parity bit, 1.5, 11.21–11.22
 Parity flag (PF), 2.11, 2.12, 4.12, 4.13
 Parking zone for hard disks, 13.37
 Partitioning hard disks, 13.44
 Passing parameters. *See* Parameter passing
 Path to files, 13.43
 PCL (printer control language), 13.52
 PCM (pulse-code modulation), 14.18
 Pel (picture element or pixel), 13.11
 PF (parity flag), 2.11, 2.12, 4.12, 4.13
 Phase-shift keying (PSK) modulation, 14.14–14.15
 Pentium, 16.1
 Dual integer pipelines, 16.3
 Non-pipelined read cycle, 16.5
 Delayed read cycle, 16.5
 Burst mode read cycle, 16.6
 Flat memory model, 16.7
 Model specific register – MSR, 16.8
 Machine check Exception – MCE, 16.8
 System management mode – SMM, 16.9
 CPUID Instruction for identifying properties And capabilities of different processors, 16.15
 Pentium p6, 16.1, 16.10
 XMM Registers, 16.2, 16.10
 Pentium Pro, 16.1
 Pentium II, 16.1
 Pentium II, Xeon, 16.1
 Pentium III, 16.1
 Pentium III, Xeon, 16.1
 Pentium 4, 16.1
 Pentium M, 16.1, 16.14
 Phoneme synthesis, 13.54
 Photocells, 10.7–10.8
 Photodiodes, 10.7–10.8
 Photoresistors, 10.7–10.8
 Phototransistors, 8.14, 8.15, 8.25
 Physical addresses, 2.14–2.15, 2.19, 3.15, 4.29–4.30, 15.9–15.11
 Physical layer (OSI model), 14.41, 14.42
 Physical segment, 3.24
 PIC. *See* 8259A priority interrupt controller
 Picture element (pel or pixel), 13.11
 PID (proportional integral derivative) control loops, 10.34, 10.35, 10.36
 Pipelined addresses, 15.18–15.19
 Pipelined instructions, 2.14
 Pitch
 of displays, 13.11
 of sounds, 13.53
 Pixel (picture element or pel), 13.11
 PLA (programmable logic array), 1.14
 Planar pixel storage, 13.12–13.13
 Plasma displays, 13.31
 Plug (P) symbols, 7.25
 PM (Presentation Manager), 15.39
 Pointer (PTR) directive, 6.35
 Pointers
 char (character), 12.18–12.20
 dereferencing, 12.14
 float (floating-point), 12.17–12.18
 head, 14.32–14.33
 int (integer), 12.15–12.17
 int array (integer array), 12.14–12.16
 interrupt-pointer table, 8.2
 passing parameters using, 5.20–5.23
 registers as, 4.26–4.29
 tail, 14.33–14.35
 to functions, 12.39
 Polled input/output (I/O), 8.12–8.13
 POP instruction, 3.7, 5.15–5.16, 6.21
 POPA instruction
 80186/80188, 3.7, 10.52
 POPF instruction, 3.7, 6.21
 Ports
 addressing and decoding, 7.14, 7.21, 7.34–7.36
 decoders for, 7.28–7.29, 7.34–7.36
 described, 2.6–2.9
 fixed-port instructions for, 4.16, 7.34
 IN instruction for, 3.7, 3.18, 4.16, 6.10, 7.35
 OUT instruction for, 3.7, 4.16, 6.21, 7.35
 variable-port instructions for, 4.16

- Postfix operations, 12.23
 Powers of, 2, 1.2
 Precedence of operators in C, 12.23–12.24
 Precision (accuracy) of numbers, 11.25
 Preemptive priority-based scheduling, 15.5
 Prefix operations, 12.23
 Preprocessor directives, 12.2
 Presentation layer (OSI model), 14.42–14.43
 Pressure transducers, 10.11–10.12
 Primary station, 14.39
 Print servers, 15.45–15.46
 Printed-circuit-board design, 11.46
 Printer control language (PCL), 11.52
 Printer drivers, 9.13–9.17
 Printer output, 8.38–8.40
 Printers. *See also* Centronics parallel interface
 dot-matrix, 11.51–11.52
 ink-jet, 11.52–11.53
 laser, 11.51
 page, 11.51
 parallel driver program for, 9.13–9.17
 parallel interface connections for, 9.13–9.14
 Privilege-level bits, 15.10
 Procedure (PROC) directive, 6.35
 Procedures
 CALL instruction for, 3.8, 5.7–5.9, 5.14–5.15, 6.4–6.5
 debugging programs containing, 5.24–5.25
 defined, 5.6
 dummy, 5.24
 far (intersegment), 5.30–5.37
 flowchart symbol for, 3.2
 interrupt service, 5.25–5.26
 macros versus, 5.37, 5.38–5.39
 near (intrasegment), 5.11–5.14
 nested, 5.6–5.7
 nonreentrant, 5.20
 parameters of, defined, 5.17
 passing parameters to and from
 in named memory, 5.17, 5.21
 in registers, 5.17, 5.18
 summary of, 5.24
 using pointers, 5.20–5.23
 using the stack, 5.20–5.24
 program flow for, 5.6–5.7
 recursive, 5.26–5.31
 reentrant, 5.24–5.25
 RET instruction for, 3.8, 5.7, 5.8, 5.9, 5.15, 6.23
 single, 5.6, 5.7
 stacks and. *See* Stacks
 stubs, 5.24
 Process control, *See* Industrial process control
 Process flowchart symbol, 3.2
 Processes. *See* Tasks
 Processor control instructions
 8086, 3.9–3.10
 8087, 11.34
 Program development algorithm, 3.28, 3.29
 Program development tools, 3.26–3.29, 12.3–12.7
 Program execution transfer instructions, 3.8–3.9
 Programmable AND matrixes, 1.14
 Programmable array logic (PAL), 1.14
 Programmable controllers, 10.34
 Programmable logic array (PLA), 1.14
 Programmable OR matrixes, 1.14
 Programmable read-only memory (PROM), 1.14, 1.17
 Programmer's model, 7.1
 Programming languages
 8086 assembly language. *See* 8086 assembly language
 C. *See* C programming language
 high-level, 2.17
 machine language, 2.16
 Programs. *See also* Software
 8086 assembly language. *See* 8086 assembly language programs
 abstracts for, 3.12
 algorithms of. *See* Algorithms
 assembler, 2.17, 3.10, 3.19, 3.26
 bottom-up design of, 3.3
 compiler, 2.17
 debugger, 3.27, 3.28
 debugging. *See* Debugging programs
 design rule checker (DRC), 11.40
 documentation of, 3.12
 editor, 3.26–3.27
 electrical rule checker (ERC), 11.40
 error trapping for, 9.23
 executive, 10.35, 10.36
 flowcharts for. *See* Flowcharts
 functions of, 3.12
 interpreter, 2.17
 linker, 3.27
 locator, 3.27
 loops in. *See* Loops
 mainline, 10.35, 10.36
 modules of, 3.3, 3.27, 5.31
 monitor, 3.12, 3.28
 netlist (wiring list), 11.40
 pseudocode for. *See* Pseudocode
 relocatable, 3.27, 4.12
 schematic capture, 11.39–11.40, 11.41
 simulator, 11.40, 11.42–11.45
 structured, 3.3–3.7
 stubs in, 5.24
 subprograms, 2.14
 system, 3.12
 top-down design of, 3.3–3.7, 5.7
 PROM (programmable read-only memory), 1.13, 1.17
 Proportional feedback, 10.33, 10.34
 Proportional integral derivative (PID) control loops, 10.34, 10.35, 10.36
 Protected mode
 80286, 15.11, 15.14, 15.15–15.16
 80386. *See* 80386 protected mode
 Protocols
 Binary Synchronous Communication Protocol (BISYNC), 14.36–14.38
 bit-oriented (BOP), 14.38–14.40
 byte-oriented (BCP), 14.36–14.38
 defined, 14.35
 high-level data link control (HDLC) protocol, 14.38–14.40
 open systems interconnection (OSI) model for, 14.41–14.43
 synchronous data link control (SDLC), 14.38
 XMODEM, 14.36
 Prototyping
 functions in C, 12.32–12.33
 of microcomputer-based instruments, 10.46–10.47
 simulation for, 11.40, 11.42–11.45
 Pseudo operations. *See* Assembler directives
 Pseudocode
 CASE structure, 3.5, 4.20
 comparing strings, 5.4
 data sampling, 5.12
 described, 3.4
 downloading program, 14.27
 factorials, 5.27
 FOR-DO loop, 3.6, 4.30
 IF-THEN-ELSE structure, 3.5, 4.15, 4.16, 4.19, 4.20

- IF-THEN structure, 3.5, 4.15
 interrupt input, 8.12
 moving strings, 5.1–5.2
 REPEAT-UNTIL structure, 3.5, 4.22, 4.26, 4.27, 4.28
 sequence structure, 3.5
 strobed input, 4.25
 terminal emulator, 14.23
 WHILE-DO structure, 3.5, 4.20, 4.23
PSK (phase-shift keying) modulation, 14.14–14.15
PTR (pointer) directive, 6.35
PUBLIC directive, 5.31, 6.33, 6.34–6.35
 Pulse-code modulation (PCM), 14.18
PUSH instruction
 8086, 3.7, 5.15–5.16, 6.21
 80186/80188, 10.52
PUSHA instruction
 80186/80188, 3.7, 10.52
PUSHF instruction, 3.7, 6.21
 Pythagorean theorem, 11.35
QAM (quaternary amplitude modulation), 14.15–14.16
 Quadbits, 14.15–14.16
 Quadword type, 6.31
 Queue registers, 2.13, 2.14
 R (reset) inputs, 1.15
 R/M bit patterns, 3.13–3.15
RAM. See Random-access memory
RAM (random-access memory) disks, 11.48
 RAMDACs, 3.17
 Ramp generators, 10.3, 10.6
Random-access memory (RAM)
 address decoding for, 7.27–7.28, 7.33–7.34
 display refresh, 13.7–13.8
DRAM. See Dynamic random-access memory
 dual-ported, 13.16
 microcomputer use of, 2.6
 static (SRAM), 1.17–1.18, 11.11, 11.16
 video (VRAM), 13.14–13.15
 volatile nature of, 1.17
Random-access memory (RAM) disks, 11.48
 Raster scanning, 13.15–13.16
RCL instruction
 8086, 3.8, 4.7, 6.21–6.22
RCR instruction
 8086, 3.8, 6.22–6.23
- Read-only memory (ROM)
 address decoding for, 7.26–7.27, 7.32–7.33
 character generator, 13.8–13.10
 description of, 1.16–1.17
 electrically erasable programmable (EEPROM), 1.17
 erasable programmable (EPROM), 1.17
 flash EEPROM, 1.17
 mask-programmed, 1.17
 microcomputer use of, 2.6
 nonvolatile nature of, 14
 optical (OROM), 11.49
 programmable (PROM), 1.13, 1.17
- Read-write memory. *See* Random-access memory
- READY input, 7.3–7.6, 7.13, 7.14, 7.38–7.39
- Real mode
 8086, 2.11
 80286, 15.11, 15.12, 15.14–15.15
 80386, 14.26, 14.27
- Real numbers, 11.24–11.25
- Real-time clocks, 8.16–8.17, 8.35–8.39
- Real transfers (8087), 11.31
- Recursive procedures, 5.26–5.31
- Red-green-blue (RGB) monitor, 13.11–13.12
- Redirected data, 13.42
- Reduced instruction set computer (RISC) processors, 13.30
- Reentrant procedures, 5.24–5.25, 15.3–15.4
- Refresh controllers, 1.18, 11.10–11.11
- REG bits, 3.13, 3.14
- Register addressing mode, 2.18–2.19
- Register storage class in C, 12.35–12.36
- Register-to-register architecture, 10.48, 10.49
- Registers
 accumulator (AL), 2.13–2.14
 AH, 2.12–2.13
 AX, 2.13
 base pointer (BP), 2.12–2.13
 BH, 2.13–2.14
 BL, 2.13–2.14
 BX, 2.13
 CH, 2.13–2.14
 CL, 2.13–2.14
 code segment (CS), 2.13, 2.14–2.16, 4.29–4.30
 as counters, 4.25–4.29
 CX, 2.13
- data segment (DS), 2.12, 2.14–2.16, 2.18–2.20, 4.29–4.30
 data storage, 1.15
 debug (80386), 15.22–15.24
 destination index (DI), 2.12–2.15
 DH, 2.13–2.14
 DL, 2.13–2.14
 DX, 2.13
 extra segment (ES), 2.12, 2.13–2.16, 4.29–4.30
 flag, 2.13–2.14
 general-purpose, 2.13–2.14
 index, 2.13, 2.16
 instruction pointer (IP), 2.6, 2.12, 2.14–2.15
 passing parameters in, 5.18, 5.19
 as pointers, 4.26–4.29
 queue, 2.13–2.14
 segment, 2.13, 2.14–2.16, 2.18–2.20, 3.25–3.26
 shift, 1.15
 source index (SI), 2.12, 2.15
 stack pointer (SP), 2.12, 2.15
 stack segment (SS), 2.12, 2.14–2.16, 4.29–4.30
- Relational operators in C, 12.21
- Relay drivers, 9.39–9.40
- Relays
 mechanical, 9.40
 solid-state, 9.40–9.41
- Relocatable programs, 3.27, 4.12
- REP** (repeat) instruction, 3.8, 5.3–5.4, 6.23
- REPE** instruction, 3.8, 6.23
- Repeat (REP) instruction, 3.8, 5.3–5.4, 6.23
- REPEAT-UNTIL structure, 3.5, 3.6, 4.22–4.29, 4.31
- Repetition operations, 3.3–3.6
- REPNE** instruction, 3.8, 6.23
- REPNZ** instruction, 3.8, 6.23
- REPZ** instruction, 3.8, 6.23
- Reserved interrupts, 8.3
- Reset (R) inputs, 1.15
- RESET response of 8086, 7.3, 7.7
- Residual error, 10.34
- Resistance temperature detectors (RTDs), 10.10
- Resistor packs, 7.25
- RET** instruction, 3.8, 5.7, 5.8, 5.9, 5.15, 6.23
- Return address, 5.7, 6.4

Index

- Reversed division instructions, 8087, 11.32
Reversed subtraction instructions, 8087, 11.35
RGB (red-green-blue) monitor, 13.11–13.12
Ring (circular) buffer, 14.32–14.33
Ring networks, 14.40–14.41
RISC (reduced instruction set computer) processors, 13.30, 14.41
RLL (run-length-limited) coding for disks, 13.38–13.39
Robots and robotics, 10.47
ROL instruction 8086, 3.8, 4.7, 6.23–6.34
ROM. *See* Read-only Memory
ROR instruction 8086, 3.8, 6.24
ROTATE instruction 80186/80188, 10.52
Rotate instructions, 8086, 3.8
RS-232C standard, 14.9–14.12
RS-422A standard, 14.13–14.14
RS-423A standard, 14.13
RS-449 standard, 14.15
RTDs (resistance temperature detectors), 10.10
Run-length-limited (RLL) coding for disks, 13.38, 13.39
S (set) inputs, 1.15
SAHF instruction, 3.7, 6.24
SAL instruction 8086, 3.8, 6.26
Sampling theorem, 10.56
SAR instruction 8086, 3.8
Saturation of amplifiers, 10.6
SBB instruction, 3.7, 6.26, 15.6
Scalable Processor Architecture (SPARC), 571
Scale. *See* Microcomputer-based scale
SCAS/SCASB/SCASW instructions, 3.8, 6.27
Scheduling preemptive priority-based, 15.6 terminate-and-stay-resident (TSR) programs and, 15.1–15.6 time-slice, 15.6
Schematic diagrams capture programs for, 11.39–11.40, 11.41 connector symbols on, 7.25
ICs on, 7.25
input signal lines on, 11.36, 11.37
jack (J) symbols on, 7.25
output signal lines on, 11.36, 11.37
plug (P) symbols on, 7.25
resistor packs on, 7.25
SDK-86, 7.15–7.24, 7.26
zone coordinates for, 7.25, 11.36
Scientific notation, 11.25
Scramblers, 14.16
Scrubbing process, 11.23
SCSI (small computer systems interface) standard, 13.42–13.43
SDK-86. *See also* Microcomputer-based industrial process control system; Microcomputer-based scale
7-segment LCD interfacing with, 9.42–9.43
7-segment LED display interfacing with, 9.33–9.34
74LS138 address decoder added to, 8.19, 8.20
74LS164 wait-state generator, 7.3, 7.14, 7.16, 7.38–7.39
74LS244 drivers, 7.14, 7.18
74LS393 baud rate generator, 7.13, 7.14, 7.23, 7.24
74S373 address latches, 7.13, 7.14, 7.17, 7.26
2142 SRAM, 7.13, 7.14, 7.20, 7.33–7.34
2316/2716 PROM, 7.13, 7.14, 7.15, 7.32–7.33
3625 I/O decoder, 7.14, 7.21, 7.34–7.37
3625 off-board decoder, 7.14, 7.19, 7.35–7.36
3625 PROM decoder, 7.13, 7.14, 7.15, 7.32–7.33
3625 RAM decoder, 7.13, 7.14, 7.20, 7.33–7.34
8251A USART, 7.13, 7.14, 7.23
8254 programmable timer/counter added to, 8.18, 8.19
8255A programmable parallel ports, 7.13, 7.14, 7.19
8259A priority interrupt controller (PIC) added to, 8.18, 8.19, 8.33
8279 specialized I/O device, 7.14, 7.17–7.18, 7.21
8284 clock generator, 7.13, 7.14, 7.15
8286 control and data transceivers, 7.14, 7.18
block diagram of, 7.14
bypass capacitors in, 7.15, 7.25
clock frequency of, 7.13
description of, 7.12–7.17, 7.18–7.19, 7.25
display driver for, 9.40–9.41
downloading programs to, 14.8, 14.10–14.17, 14.25–14.35
filter capacitors in, 7.15, 7.25
GO command, 4.5
input/output (I/O) addressing and decoding on, 7.14, 7.21, 7.34–7.36
instrument prototyping with, 10.46–10.47
keypad interfacing with, 9.27, 9.28
off-board decoder, 7.14, 7.19, 7.37–7.38
parallel printer connection to, 9.9, 9.11–9.12
parallel printer driver program for, 9.12–9.14
port addressing and decoding on, 7.14, 7.21, 7.34–7.36
printer driver program for, 9.12–9.13
random-access memory (RAM) address decoding on, 7.13, 7.14, 7.20, 7.33–7.34
read-only memory (ROM) address decoding on, 7.13, 7.14, 7.15, 7.32–7.33
RS-232C interface for, 14.8, 14.9–14.10, 14.24–4.25
schematic diagrams of, 7.15–7.23, 7.25
single-step command, 4.5
wait-state generator, 7.13, 7.14, 7.16, 7.38–7.39
SDLC (synchronous data link control) protocol, 14.38
Second-generation microprocessors, 2.10
Secondary stations, 14.39
Seek time for disks, 13.36
Segment base address, 2.14, 2.15, 2.16, 2.18, 3.14–3.15, 4.29–4.30
Segment base:offset form of addresses, 2.15, 2.19
SEGMENT directive, 3.19–3.20, 6.35
Segment load instructions (80386)
Segment override prefix, 3.14, 3.17–3.18
Segment registers, 2.13, 2.14–2.16, 2.18–2.19, 3.25–3.26
Segment selector, 15.10
Segmentation of memory, 2.14–2.16, 2.18–2.19
Segments code, 3.24

- data, 3.24
extra, 3.24
initializing segment registers, 3.25–3.26
logical, 3.19–3.20, 3.24
physical, 3.24
stack, 3.14, 3.24
Selection flowchart symbol, 3.2
Selection operations, 3.4, 3.5
Semaphores, 15.6, 15.7
Semiconductor temperature sensors, 10.8–10.9
Sensors *See also* Transducers
defined, 10.1
flow, 10.12
light, 10.7–10.8
temperature, 10.8–10.10, 10.35–10.37
Sentinel method, 9.13
Sequence structure, 3.3, 3.5, 4.1–4.4, 4.5–4.8
Serial data communication. *See also* 8251A USART
asynchronous, 14.2–14.3
baud rate for, 14.2–14.3
full-duplex, 14.2–14.3
half-duplex, 14.2, 14.3
marking state for, 14.2–14.3
modems for, *See* Modems
RS-232C standard for, 14.9–14.12
RS-422A standard for, 14.9–14.12
RS-423A standard for, 14.9
RS-449 standard for, 14.10
simplex, 14.2–14.3
start bit for, 14.2–14.3
stop bit for, 14.2–14.3
synchronous, 14.2–14.3
Servo control, 10.31–10.32
Session layer (OSI model), 14.41–14.42
Set (S) inputs, 1.15
Set memory flag word instruction (80386), 15.36
Set points, 10.31–10.32
Settling time, 10.31–10.32
SF (sign flag), 2.12, 2.13, 4.12, 4.13
Shaft encoders
absolute, 9.44–9.45
defined, 9.44
incremental, 9.45–9.46
Shift between words instructions (80386), 15.36
SHIFT instruction
80186/80188, 10.52
80286, 15.15
Shift instructions, 8086, 3.8
Shift registers, 1.15, 13.33
SHL instruction
8086, 3.8, 6.25
SHLD instruction (80386), 15.36
Short jumps, 4.9–4.11, 4.14, 4.23–4.24
SHORT operator, 6.35
SHR instruction
8086, 3.8, 6.28–6.29
SHRD instruction (80386), 15.36
SI (source index) register, 2.12, 2.13
Sign bit, 1.7–1.9
Sign flag (SF), 2.12, 2.13, 4.12, 4.13
Signal assertion level, 1.13
Signed numbers, 1.7–1.9
Significand (mantissa) of numbers, 11.25
Simple input/output (I/O), 9.1–9.10
Simple strobe input/output (I/O), 9.1–9.10
Simplex communication, 14.1–14.2
Simulator programs, 11.40–11.45
Single-board computers, 2.2, 2.4
Single-density recording for disks, 13.36–13.37
Single-handshake input/output (I/O), 9.2
Single indexed addressing mode, 3.14, 4.29–4.30
Single-mode fibers, 14.22–14.23
Single-precision numbers, 11.26
Single-step command, 4.5
Single-step interrupts, 8.2, 8.8, 8.12
Slave device, 8.33
SLDT instruction (80286), 15.16
Slice, 2.10
Small computer systems interface (SCSI) standard, 13.42–13.43
SMSW instruction (80286), 15.16
Snubber circuits, 9.41
Soft errors, 11.21
Software. *See also* Programs
defined, 2.7
upward-compatible, 2.11, 8.1, 8.39–8.40
Software interrupts, 3.9, 3.18, 6.10–6.11, 8.1, 8.10, 8.12, 8.39–8.40
Software-triggered strobes, 8.28, 8.29
SOH (start-of-header) character, 14.36
Solar cells, 10.8
Solenoid drivers, 9.39–9.40
Solid-state relays, 9.40–9.41
Source file, 3.26
Source index (SI) register, 2.12–2.13
Source for instructions, 2.16–2.17
SP (stack pointer) register, 2.12–2.13
SPARC (Scalable Processor Architecture)
Speech recognition, 13.53, 13.54–13.55
Speech synthesis, 13.53–13.54
Square-wave generators, 8.26–8.27
Square waves, 10.53–10.54
SRAM (static random-access memory), 1.17–1.18
SS (stack segment) register, 2.12, 2.13–2.16, 4.29–4.39
SSE2 and SSE3, 16.2, 16.13, 16.14
Stack diagrams, 5.9–5.10, 5.14–5.15, 5.16, 5.20–5.24, 5.27–5.31
Stack overflow, 5.20–5.24
Stack pointer (SP) register, 2.12, 2.15
Stack segment, 3.15, 3.25
Stack segment (SS) register, 2.12, 2.14–2.16, 4.29–4.30
Stacks
8087, 11.27–11.28
data sampling program using, 5.10–5.14
defined, 2.14
operation of, 5.9–5.11, 5.14–5.15
passing parameters using, 5.20–5.24
POP instruction for, 3.7, 5.15–5.16, 6.21
PUSH instruction for, 5.15–5.16, 6.21
top of stack, 2.15
uses for, 5.9–5.10
Standard structures, 3.4–3.7
Star networks, 14.40
Start bit, 14.2
Start flowchart symbol, 3.2–3.3
Start-of-header (SOH) character, 14.36
Start-of-text (STX) character, 14.36
State (context, or environment) of tasks, 15.3–15.4
States
of clocks, 7.4
undefined, 8.20
Static displays, 9.26–9.27
Static random-access memory (SRAM), 1.17–1.18, 11.9–11.12
Static storage class in C, 12.36
STC instruction, 3.9, 6.28
STD instruction, 3.9, 6.28
Stepper motors, 9.42–9.43
STI instruction, 3.9, 6.28, 8.10
Stimulus files, 11.43–11.44
Stop bit, 14.2–14.3
Stop flowchart symbol, 3.2
STOS/STOSB/STOSW instructions, 3.8, 6.29

- Strain gages, 10.10–10.11
 Streaming tape systems, 11.49
 String instructions, 3.8, 5.1–5.6
 String library functions in C, 12.40–12.41
 Strings
 comparing, 5.4–5.6
 defined, 3.8, 5.1
 moving, 5.1–5.4
 Streaming SIMD extensions—SSE, 16.2
 Strobe input/output (I/O), 9.1–9.2
 Strobes
 described, 4.24–4.26
 hardware-triggered, 8.28–8.29
 software-triggered, 8.28, 8.29
 Structured programming, 3.4–3.7
 Stubs, 5.24
 STX (start-of-text) character, 14.36
 SUB instruction, 3.7, 6.26–6.27
 Subprograms. *See* Procedures
 Subroutine flowchart symbol, 3.2
 Subroutines. *See* Procedures
 Subtraction
 8086 instructions for, 3.7–3.8
 8087 instructions for, 11.31
 binary, 1.9
 binary-coded decimal (BCD), 1.11–1.12
 Successive approximation analog-to-digital (A/D) converters, 10.19, 10.20, 10.21
 Summing point, 10.5
 Supercomputers, 2.1, 2.2
 Supersets of instructions, 2.13
 Supervisor, 15.5
 Switch structure in C, 12.25–12.26
 Switched capacitor digital filters, 10.58
 Switched phone lines, 14.2
 Symbol table, 3.23, 3.26, 211
 Sync characters, 14.36
 Synchronization instructions, 3.9–3.10
 Synchronous communication, 14.1–14.2
 Synchronous data link control (SDLC) protocol, 14.38
 Syndrome word, 11.22–11.23, 11.24
 Syntax of assembly language, 3.12
 System commands, 3.12
 System degradation for time-sliced systems, 15.5
 System expansion slots, 11.1
 System program, 3.12
 Tachometers, 10.32–10.33
 Tail pointer, 14.33–14.34
 Tape readers, 9.6–9.8
 Tasks
 blocked, 15.6
 defined, 15.1
 environment (context, or state) of, 15.5–15.6
 mutual exclusion of, 15.6
 TDM (time-division multiplexing), 14.8–14.9
 Temperature sensors, 10.8–10.10, 10.34, 10.35–10.37
 Templates for instructions, 3.13–3.19
 Ten-byte type, 6.31–6.32
 Terabyte (unit), 2.10
 Terminal emulator program, 14.24–14.25, 14.26
 Terminals, 13.6
 Terminate-and-stay-resident (TSR) programs, 15.3–15.5
 TEST instruction, 3.8, 6.29–6.30
 Texas Instruments Graphics Architecture (TIGA) standard, 13.36
 TF (trap flag), 2.13, 8.1–8.4, 8.8–8.11
 Thermal printers, 11.51–11.52
 Thermal sensitive resistors (thermistors), 10.10
 Thermocouples, 10.9–10.10
 Thin Ethernet (10BaseT) networks, 14.43
 Thrashing process, 11.18–11.19
 Three-state outputs, 1.16–1.17, 2.7
 TIGA (Texas Instruments Graphics Architecture) standard, 13.29
 Time-division multiplexing (TDM), 14.19–14.20
 Time-domain description, 10.53, 10.54
 Time-multiplexed systems, 2.3–2.4, 2.20
 Time-slice scheduling, 15.5
 Time-sliced systems, 2.3–2.4, 2.20
 Timed interrupt generators, 8.25–8.26
 Timesharing systems, 2.3–2.4, 2.20
 Timing
 8086 instructions, 4.31–4.34
 delay loops for, 4.31–4.34, 5.10–5.14
 interrupts for, 8.14–8.17
 Timing parameters, 8086, 7.39–7.42
 Timing waveforms
 8086 maximum mode, 7.39–7.41
 8086 minimum mode, 7.39–7.42
 8086 system timing, 7.2–7.5
 8237 DMA controller, 11.7–11.8
 8254 programmable timer/counter, 8.22–8.28
 8255 handshake data input from a tape, 9.9–9.10
 8279, 9.28–9.29
 Centronics parallel interface, 9.11–9.12
 clock, 7.4
 data acquisition system (DAS), 10.35–10.36
 double-handshake input/output (I/O), 9.3–9.4
 dynamic random-access memory (DRAM), 11.10–11.12
 simple input/output (I/O), 9.1–9.2
 simple strobe input/output (I/O), 9.1–9.2
 single-handshake input/output (I/O), 9.2
 Token, defined, 14.40–14.41
 Token (file handle), 13.45–13.46
 Token-passing ring networks, 14.40, 14.41–14.42
 Top-down design, 3.3–3.7, 5.7
 Top of stack, 2.15
 Topologies
 network, 14.39–14.40
 Trace data, 3.28, 7.9, 7.10–7.11
 Trackballs, 13.31–13.32
 Tracks of disks, 13.35, 13.38–13.39
 Transceivers, 14.42–14.43
 Transcendental instructions (8087), 11.33–11.34
 Transducers. *See also* Sensors
 defined, 10.10
 differential pressure, 10.13
 force, 10.10–10.12, 10.20–10.21, 10.22
 pressure, 10.11–10.12
 Transistor buffers, 9.37–9.38
 Transistors
 Darlington, 9.38–9.39
 isolated-gate bipolar (IGBTs), 9.39–9.40
 metal-oxide-semiconductor field-effect (MOSFETs), 9.40–9.41
 Transport layer (OSI model), 14.41–14.42
 Trap flag (TF), 2.13, 8.1–8.4, 8.8–8.12
 Tree-structured (broadband bus) networks, 14.39–14.40
 Trees (structures), 5.26
 Trellis code, 14.16
 Triacs, 9.40–9.41
 Tribits, 14.15
 Trigonometric instructions (8087), 11.33
 Troubleshooting microcomputers, 7.42–7.46

- TSR (terminate-and-stay-resident) programs, 15.2–15.3
- Two-key lockout, 9.21
- Two-key rollover, 9.23
- Type byte, 2.18, 3.21, 6.31
- Type doubleword, 3.21, 6.31
- Type error, 2.18
- TYPE operator, 6.26
- Type quadword, 6.31
- Type ten bytes, 6.31–6.32
- Type word, 2.18, 3.21, 6.32
- UART (universal asynchronous receiver-transmitter), *See also* 8250 UART
- Unconditional jumps, 4.8, 4.9–4.12
- Unconditional transfer instructions, 3.8
- Undefined states, 8.20
- Underdamped response, 10.31–10.32
- Underflow, 1.9
- Union data structure in C, 13.5–13.6
- Unipolar binary-coded decimal (BCD) codes, 10.20
- Unipolar binary codes, 10.20
- Unity-gain bandwidth, 10.5
- Universal asynchronous receiver-transmitter (UART), *See also* 8250 UART
- Universal Synchronous/Asynchronous Receiver Transmitter (USART), 7.14, 7.23, 7.24, *See also* 8251A USART
- Unpacked binary-coded decimal (BCD) code, 4.6–4.8
- Unvoiced sounds, 13.53
- Upward-compatible software, 2.11
- USART (Universal Synchronous/Asynchronous Receiver Transmitter), 7.14, 7.23, 7.24, *See also* 8251A USART
- Variable-port instructions, 4.17, 7.34
- Variable storage classes in C, 12.33–12.34
- Variables
- char (character), 12.9–12.10
 - dummy, 5.38
 - float (floating-point), 12.11
 - int (integer), 12.10–12.11
 - named, 3.21–3.25
 - types in C, 12.8–12.9
- VERR instruction (80286), 15.15–15.16
- VERW instruction (80286), 15.15–15.16
- VGA (video graphics array), 13.15, 13.17–13.19
- Video cameras, 13.33–13.34
- Video digital-to-analog (D/A) converters, 13.17–13.18
- Video graphics array (VGA), 13.14, 13.15–13.39
- Video monitors, 13.6
- Video programming
- high-level, 13.27–13.29
 - introduced, 13.19
 - low-level, 13.19–13.20
- Video random-access memory (VRAM), 13.14–13.15
- Vidicons, 13.32–13.33
- Virtual 8086 mode (80386), 15.32–15.33
- Virtual address mode, 2.11
- Virtual addresses, 15.8–15.9
- Virtual ground, 10.4–10.5
- Virtual memory, 15.8–15.9
- Vocal tract model, 13.54
- Voiced sounds, 13.53
- Volatile memory, 1.17
- Voltage gain, 10.2–10.4
- Von Neumann architecture, 10.56
- VRAM (video random-access memory), 13.14–13.15
- W bit, 3.13
- WAIT instruction, 3.9, 6.29
- WAIT states, 7.2–7.6, 7.13, 7.38–7.39
- WHILE-DO structure, 3.5, 3.6, 4.20–4.22, 12.26–12.27
- While structure in C, 12.26–12.27
- Winchester hard disks, 13.37
- Windows program, 15.39–15.40
- Wiring list (netlist) program, 11.39–11.40
- Word transfer instructions, 3.7
- Word type, 2.18, 3.21, 6.32
- Words
- 8086 storage of, 2.19
 - binary. *See* Binary words
 - command. *See* Control words
 - control. *See* Control words
 - defined, 1.1
 - mode. *See* Control words
 - syndrome, 11.23–11.24
- Write once/read many (WORM) disks, 11.49
- XCHG instruction, 3.7, 6.30
- XLAT instruction, 3.7, 6.30
- XLATB instruction, 6.30
- XMODEM protocol, 14.36–14.37
- XMS (extended) memory, 15.8–15.9
- XNOR (exclusive NOR) gate, 1.12, 1.13
- XOR (exclusive OR) gate, 1.12, 1.13
- XOR instruction, 3.8, 6.30–6.31
- Z80 microprocessor, 2.10
- Zero flag (ZF), 2.13, 4.12, 4.13
- Zero-point switching, 9.41–9.42
- ZF (zero flag), 2.13, 4.12, 4.13
- Zone coordinates, 7.25, 11.36–11.37

microprocessors and Interfacing

DOUGLAS V HALL

This book has been written for a course on microprocessor 8086 by Douglas V Hall, the most trusted name in the world of microprocessors.

Features:

- ◆ Focused coverage of 8086 microprocessor.
- ◆ Examples of programs with C and assembly language modules .
- ◆ Interfacing illustrated via real life systems.
- ◆ Introduction to Neural Networks and Fuzzy Logic.
- ◆ Inclusion of a new chapter on Pentium Processors.
- ◆ Focused discussion on Software Keyboard Interfacing.
- ◆ Multimedia Technologies: MMX, SSE, SSE2 and SSE3 are also discussed.



Dedicated web site: <http://www.mhhe.com/hall/microprocessors>

Visit us at : www.tatamcgrawhill.com

ISBN-13: 978-0-07-060167-3
ISBN-10: 0-07-060167-4



9 780070 601673



Tata McGraw-Hill