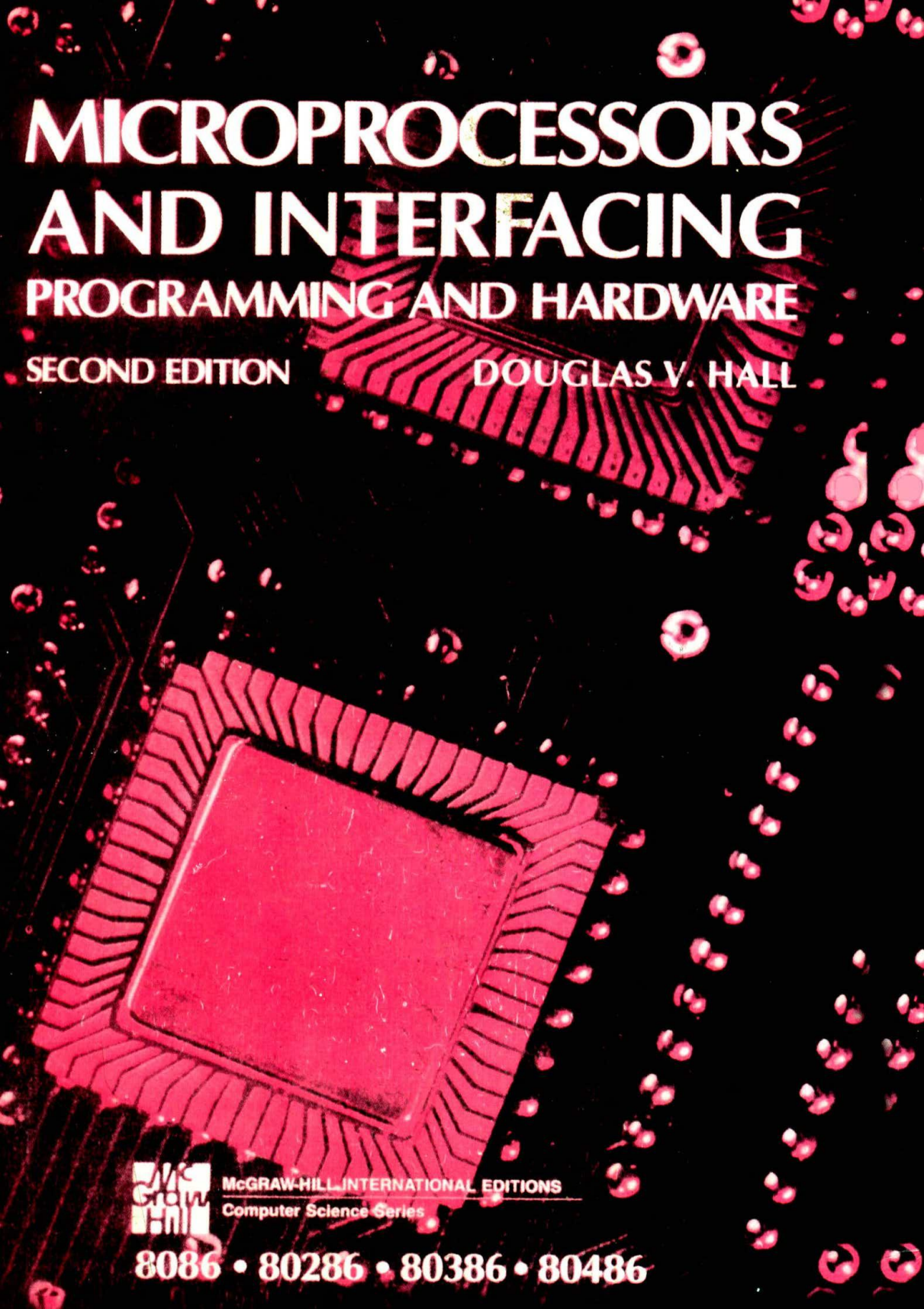
MICROPROCESSORS AND INTERFACING

PROGRAMMING AND HARDWARE

SECOND EDITION

DOUGLAS V. HALL



McGRAW-HILL INTERNATIONAL EDITIONS
Computer Science Series

8086 • 80286 • 80386 • 80486

MICROPROCESSORS AND INTERFACING

MICROPROCESSORS AND INTERFACING

PROGRAMMING AND HARDWARE

EDITION

DOUGLAS V. HALL

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices
New Delhi New York St Louis San Francisco Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan Montreal
San Juan Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

A Division of The McGraw-Hill Companies

MICROPROCESSORS AND INTERFACING: Programming & Hardware, 2/E

TATA McGraw-Hill Edition 1999

Sixth reprint 2000

Reprint, 2003

RYLQCRACRQRQL

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

For Sale in India Only

Library of Congress Cataloging-in-Publication Data

Hall, Douglas V.

Microprocessors and interfacing : Programming and hardware / Douglas V. Hall.—2nd ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-07-025742-6 (text).—ISBN 0-07-025743-4 (experiments manual).—ISBN 0-07-025744-2 (instructor's manual)

1. Micropocessors—Programming. 2. Microprocessors. 3. Computer interfaces. I. Title

QA76.6.H2994 1991

005.26—dc20 91-14526

CIP

When ordering this title use ISBN 0-07-463639-1

Published by Tata McGraw-Hill Publishing Company Limited, 7 West Patel Nagar, New Delhi 110 008, and printed at Rajkamal Electric Press, GTK Road, Delhi 110 033

TO MY STUDENTS

Let us go forward together into the future.

CONTENTS

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 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 newer processors and how these features are used in multiuser/multitasking systems.

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 Chapters 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. Chapter 10 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 11–15

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. Chapter 11 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.

Finally, 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 and the book conclude with introductions to parallel processors, neural networks, and fuzzy logic. I think you will find these newly developing areas as fascinating as I have.

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.

Chapter 15

The final chapter is on the cutting edge of the development of new microprocessors. It is my hope that all students will have the opportunity to read this chapter. At the very least students should read the section on the 386. This is a final chapter, yet it is only the beginning of their study of microprocessors.

NEW FEATURES IN THIS EDITION

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

1. The order of the topics in Chapters 4 and 5 has been improved, based on instructor feedback.
2. A greatly expanded section on digital signal processing hardware and software has been added to Chapter 10.
3. A section in Chapter 11 describes and shows an example of how electronic design automation tools such as schematic capture programs, simulator programs, and PC board layout programs are used to develop the hardware for a microcomputer system.
4. At the request of industry advisors, Chapter 12 is a completely new chapter which contains a solid introduction to the C programming language, including examples of programs with C and assembly language modules.
5. Chapters 13 and 14, the systems peripherals chapters, have been updated to reflect advances in technology such as VGA graphics, optical-disk storage, laser printers, and digital video interactive. The chapters now include both assembly language and C interface program examples.
6. The network section of Chapter 14 has been expanded to reflect the current importance of networks.

7. Chapter 15 now contains an extensive description of the features of the 386 and 486 processors and a discussion of how these features are used in multitasking environments such as Microsoft's OS/2 and Windows 3.0.
8. Introductions to neutral network computers and to fuzzy logic have been added to Chapter 15.

SPECIAL FEATURES AND SUPPLEMENTS

This book and the Experiments Manual written to accompany it contain many hardware and software exercises students can do to solidify their knowledge of microprocessors. 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 Experiments Manual contains 40 laboratory exercises that are directly coordinated to the text. Each experiment includes chapter references, required equipment, objectives, and experimental procedures.

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

The Instructor's Manual includes disk directories. There are two disks available. This set of disks contains the source code for all the programs in the text and 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.

ACKNOWLEDGMENTS

I wish to express my profound thanks to the people around me who helped make this book a reality. 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 problems 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.

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

CHAPTER 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 1-1*a*, p. 2, 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, Figure 1-1*b* 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 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: $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$.

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

(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
2^8 = 256	2^{16} = 65,536	2^{24} = 16,777,216	2^{32} = 4,294,967,296

(b)

FIGURE 1-1 (a) Digit values in binary. (b) Powers of 2.

or 4 + 0 + 1 = 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 (Figure 1-2*a*) 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 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 Figure 1-2*b*. 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 Figure 1-2*b*.

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 Figure 1-2*c*. 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

	2^5	2^4	2^3	2^2	2^1	2^0
	32	16	8	4	2	1
21_{10} =	0	1	0	1	0	1_2

(a)

227_{10} = ? Binary

Least Significant Binary Digit ↓

Division	Quotient	Remainder		Check
2)227 =	113	R1	× 1 =	1
2)113 =	56	R1	× 2 =	2
2)56 =	28	R0	× 4 =	0
2)28 =	14	R0	× 8 =	0
2)14 =	7	R0	× 16 =	0
2)7 =	3	R1	× 32 =	32
2)3 =	1	R1	× 64 =	64
2)1 =	0	R1	× 128 =	128
				227 Check

↑ Most Significant Binary Digit

$\therefore 227_{10} = 11100011_2$

(b)

		MSB	Check
2 × .625	=	1.25	1 × .5
2 × .25	=	0.50	0 × .25
2 × .50	=	1.00	1 × .125
		LSB	.625

(c)

FIGURE 1-2 Converting decimal to binary. (a) Digit value method. (b) Divide by 2 method. (c) Decimal fraction conversion.

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.

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 1-3*a* 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 Figure 1-3*b* shows the symbols for hex code.

$$16^3 \quad 16^2 \quad 16^1 \quad 16^0 . \quad 16^{-1} \quad 16^{-2} \quad 16^{-3}$$
$$4096 \quad 256 \quad 16 \quad 1 \quad \frac{1}{16} \quad \frac{1}{256} \quad \frac{1}{4096}$$

(*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*)

227_D = ? Hex LSD

16|227 = 14 *R*3 × 1 = 3

16| 14 = 0 *R*E × 16 = 224

MSD 227

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

(*c*)

FIGURE 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, Figure 1-3*c* 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-bit-wide 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, p. 4, 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

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	1 1 1 1 1 1 0	0
1	0001	1	1	0001	0011 0100	0001	0 1 1 0 0 0 0	1
2	0010	2	2	0010	0011 0101	0011	1 1 0 1 1 0 1	2
3	0011	3	3	0011	0011 0110	0010	1 1 1 1 0 0 1	3
4	0100	4	4	0100	0011 0111	0110	0 1 1 0 0 1 1	4
5	0101	5	5	0101	0011 1000	0111	1 0 1 1 0 1 1	5
6	0110	6	6	0110	0011 1001	0101	1 0 1 1 1 1 1	6
7	0111	7	7	0111	0011 1010	0100	1 1 1 0 0 0 0	7
8	1000	10	8	1000	0011 1011	1100	1 1 1 1 1 1 1	8
9	1001	11	9	1001	0011 1100	1101	1 1 1 0 0 1 1	9
10	1010	12	A	0001 0000	0100 0011	1111	1 1 1 1 1 0 1	A
11	1011	13	B	0001 0001	0100 0100	1110	0 0 1 1 1 1 1	B
12	1100	14	C	0001 0010	0100 0101	1010	0 0 0 1 1 0 1	C
13	1101	15	D	0001 0011	0100 0110	1011	0 1 1 1 1 0 1	D
14	1110	16	E	0001 0100	0100 0111	1001	1 1 0 1 1 1 1	E
15	1111	17	F	0001 0101	0100 1000	1000	1 0 0 0 1 1 1	F

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 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 figure out the Gray code for the decimal number 16. You should get 11000.

7-Segment Display Code

Figure 1-4*a* 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 Figure 1-4*b*. For a common-anode LED display such as that in Figure 1-4*c*, 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

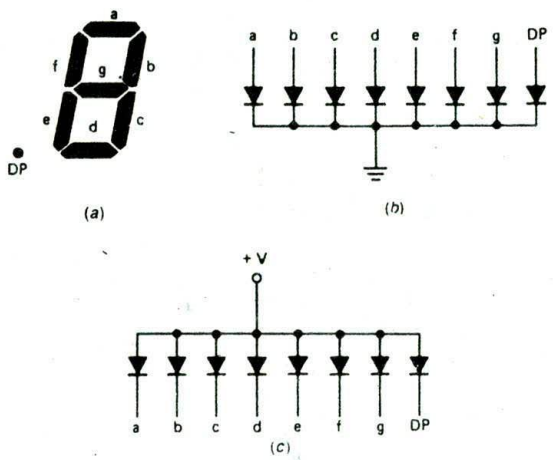


FIGURE 1-4 7-segment LED display. (a) Segment labels. (b) Schematic of common-cathode type. (c) Schematic of common-anode type.

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.

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

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
BS	08	BS	16	2	32	2	F2	\	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	0B	5	35	5	F5	—	5F	—	6D
FF	0C	FF	0C	6	36	6	F6	`	60	RES	14
CR	0D	CR	0D	7	37	7	F7	a	61	a	81
SO	0E	SO	0E	8	38	8	F8	b	62	b	82
S1	0F	S1	0F	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	1B	BYP	24	E	45	E	C5	o	6F	o	96
FS	1C	FLS	1C	F	46	F	C6	p	70	p	97
GS	1D	GS	1D	G	47	G	C7	q	71	q	98
RS	1E	RDS	1E	H	48	H	C8	r	72	r	99
US	1F	US	1F	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

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
BEL	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		

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
-----------
 [1] 01110110
  ↑
  └ Carry
```

(b)

FIGURE 1-5 Binary addition. (*a*) Truth table for 2 bits plus carry. (*b*) Addition of two 8-bit words.

alphabet, 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.

ARITHMETIC OPERATIONS ON BINARY, HEX, AND BCD NUMBERS

Binary Arithmetic

ADDITION

Figure 1-5*a* shows the truth table for addition of two binary digits and a carry in (C_{IN}) from addition of previous digits. Figure 1-5*b* shows the result of adding two 8-bit binary 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 00000111 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, 0000 0111. Invert each bit, including the MSB, to get 1111 1000. Then add 1 to get 11111001. This result is the correct representation of -7_{10}. Figure 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

	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

FIGURE 1-6 Positive and negative numbers represented with a sign bit and 2's complement.

by the 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 21_{10}, so you know that the original numbers represent -21_{10}. Again, try reconverting a few of the numbers in Figure 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. Figure 1-7*a* 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 Figure 1-7*b*, 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 1-7*c* 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 Figure 1-7*d*, 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 2^8 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:

01111111	+127
⋮	
00000001	+1
00000000	zero
11111111	−1
⋮	
10000001	−127
10000000	−128

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.

```
+13     00001101
+ 9     00001001
+22     00010110
          └─Sign bit is 0
            so result is positive
              (a)

+13     00001101
− 9     11110111 2's complement for −9 with sign bit
+ 4   1 00000100
          └─Sign bit is 0
            so result is positive
          └─Ignore carry
              (b)

+ 9     00001001
−13     11110011 2's complement for −13 with sign bit
− 4     11111100 Sign bit is 1
        00000011 So invert each bit
equals +       1 Add 1
       −00000100 Prefix with minus sign
              (c)

− 9     11110111 } 2's complement,
−13     11110011 } sign-and-magnitude form
−22     11101010 Sign bit is 1
        00010101 So invert each bit
equals +       1 Add 1
       −00010110 Prefix with minus sign
              (d)
```

FIGURE 1-7 Addition of signed binary numbers. (*a*) +9 and +13. (*b*) −9 and +13. (*c*) +9 and −13. (*d*) −9 and −13.

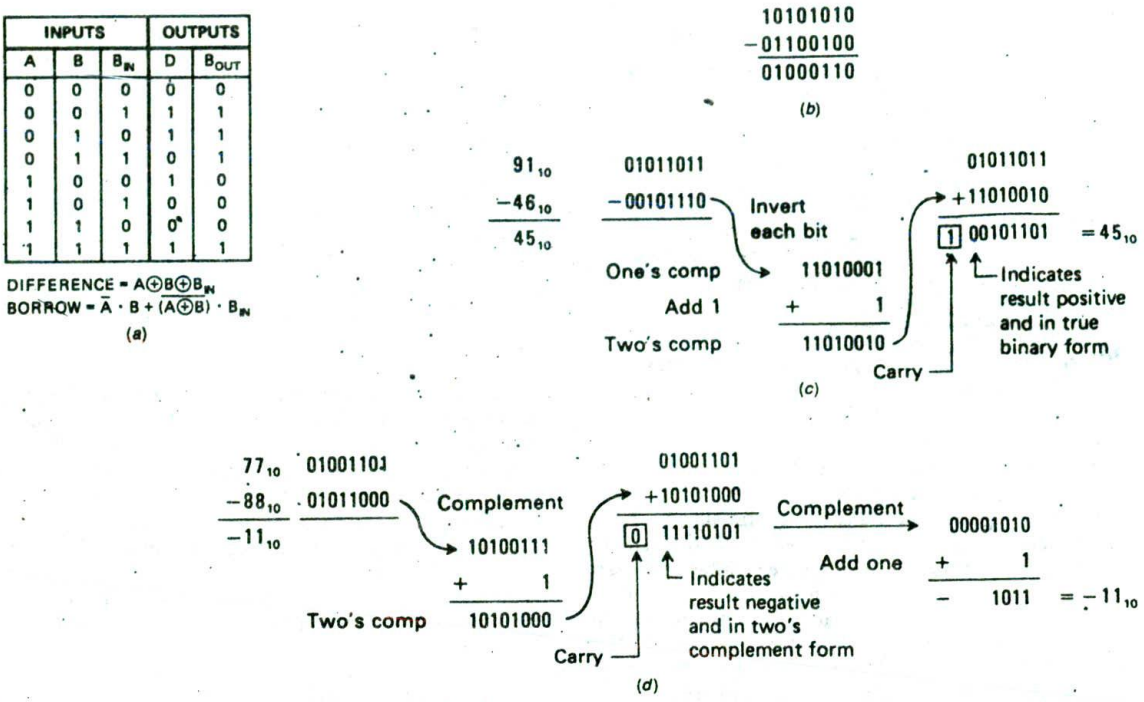


FIGURE 1-8 Binary subtraction. (a) Truth table for 2 bits and borrow. (*b*) Pencil method. (c) 2's-complement positive result. (*d*) 2's-complement negative result.

SUBTRACTION

There are two common methods for doing binary subtraction. These are the pencil method and the 2's-complement add method. Figure 1-8*a* 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. Figure 1-8*b* 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). Figure 1-8*c* 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 Figure 1-8*c*, 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. Figure 1-8*d* 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 Figure 1-8*d*.

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 −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. Figure 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

```
   11          1011     MULTIPLICAND
 x  9        x 1001     MULTIPLIER
 ----        ------
               1011 \
              0000   |
             0000    > PARTIAL PRODUCTS
            1011    /
            -------
            1100011     PRODUCT
```

FIGURE 1-9 Binary multiplication.

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.

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. Figure 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 Figure 1-10*b*, 0's can be added to the right of the binary point and division continued to convert a remainder to a binary equivalent.

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.

```
             01100   QUOTIENT
DIVISOR  110)1001000  DIVIDEND         12
            -110                     6)72
            ----
              110
             -110
             ----
                0

               (a)

          110.01                      6.25
     100)11001.00                    4)25
         -100
         ----
           100
          -100
          ----
           0100

               (b)
```

FIGURE 1-10 Binary 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 Figure 1-11*a*, 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 Figure 1-11*b*, 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 Figure 1-11*b*, 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  A16
+3F       +0011  1111          + 3     F16
---       -----------          -----------
 B9        1011  1001            1110  2510
              B     9             B16   916

                (a)                 (b)
```

FIGURE 1-11 Hexadecimal addition.

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

FIGURE 1-12 Hexadecimal subtraction.

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. Figure 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.

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. Figure 1-13

```
                BCD
 35          0011 0101
+23         +0010 0011
----        ----------
 58          0101 1000

(a)

                 BCD
  7               0111
+ 5             + 0101
----            -------
 12               1100    INCORRECT BCD
                + 0110    ADD 6
                -------
             0001 0010    CORRECT BCD 12

(b)

                 BCD
  9              1001
+ 8            + 1000
----           -------
 17          0001 0001    INCORRECT BCD
             0000 0110    ADD 6
             ---------
             0001 0111    CORRECT BCD 17

(c)
```

FIGURE 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.

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

FIGURE 1-14 BCD subtraction.

shows three examples of BCD addition. The first, in Figure 1-13*a*, 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 Figure 1-13*b*, 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 1-13*c* shows another case where a correction factor must be added. The initial addition of 9 and 8 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 Figure 1-14, subtracting 6 gives a correct BCD result of 0000 1000 or 8.

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.

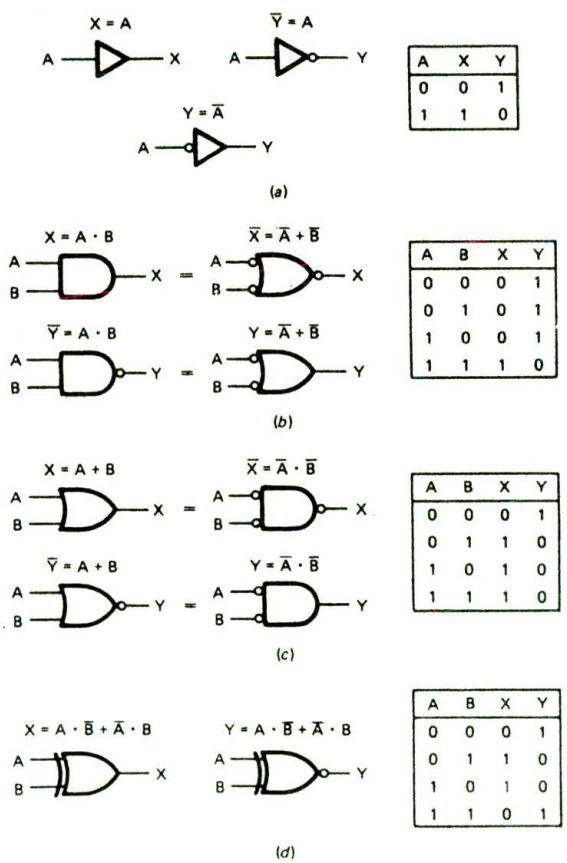
Inverting and Noninverting Buffers

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 Figure 1-15*a* 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 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 Figure 1-15*a* 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 $\overline{CS}$ 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.



A	X	Y
0	0	1
1	1	0

A	B	X	Y
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

A	B	X	Y
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

A	B	X	Y
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

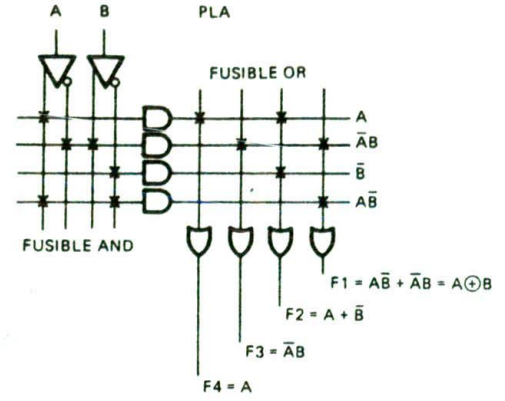
FIGURE 1-15 Buffers and logic gates. (*a*) Buffers. (*b*) AND-NAND. (*c*) OR-NOR. (*d*) Exclusive OR.

Logic Gates

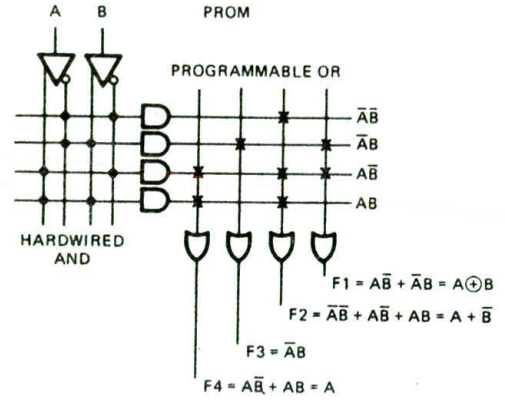
Figure 1-15*b* shows the symbols and truth tables for simple logic gates. A symbol with a flat back and a round front indicates that the device performs the logical *AND* function. This means that the output will be asserted if the A input is asserted *and* the B input is asserted. Again, bubbles or no bubbles are used to indicate the assertion level of each input and output. The first AND symbol in Figure 1-15*b* 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 Figure 1-15*b* 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 Figure 1-15*b* to see if you agree with this.

Figure 1-15*c* 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 Figure 1-15*c* has no bubble on the output, so the output will be asserted high if the A *and* B inputs are both asserted low.

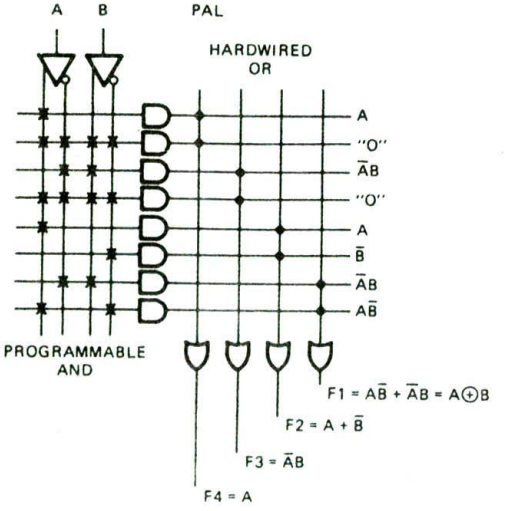
A logic symbol with a curved back indicates that the output of the device will be asserted if the A input is asserted *or* the B input of the device is asserted. Again, bubbles or no bubbles are used to indicate the assertion level for inputs and outputs. Note in Figure 1-15*b* and *c* that each of the 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 Figure 1-15*b* 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.



(a)



(b)



(c)

FIGURE 1-16 FPLA, PROM, and PAL programmed to implement some simple logic functions. (*a*) FPLA. (*b*) PROM. (*c*) PAL.

Figure 1-15*d* 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.

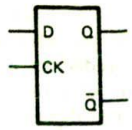
In a *programmable array logic* device or *PAL*, the connections in the OR matrix are fixed and the AND matrix connections are programmable. PALs are often used to implement combinational logic and address decoders in microcomputer systems.

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

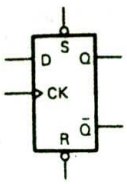
THE D LATCH

A *latch* is a digital device that stores a 1 or a 0 on its output. Figure 1-17*a* 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.



D	CK	Q	$\bar{Q}$
X	0	Q_N	$\bar{Q}_N$
0	1	0	1
1	1	1	0

(*a*)



S	R	D	CK	Q	$\bar{Q}$
1	1	1	↑	1	0
1	1	0	↑	0	1
1	1	X	0	Q_N	$\bar{Q}_N$
1	1	X	1	Q_N	$\bar{Q}_N$
0	1	X	X	1	0
1	0	X	X	0	1
0	0	X	X	•	•

(*b*)

FIGURE 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 1-17*b* 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 Figure 1-17*b* 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 Figure 1-17*b* 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 Figure 1-18*a*. 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 Figure 1-18*b*, 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

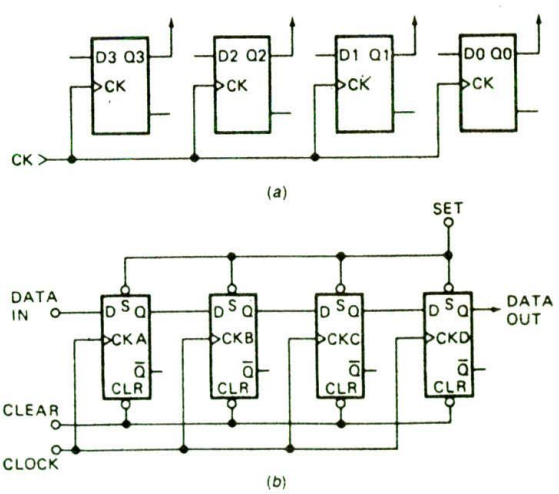
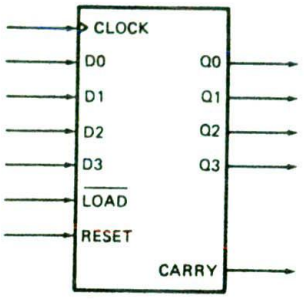


FIGURE 1-18 Registers. (a) Simple data storage. (*b*) Shift register.



(a)

Q3	Q2	Q1	Q0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

(b)

FIGURE 1-19 Four-bit, presettable binary counter. (*a*) Schematic symbol. (*b*) Count sequence.

when they are clocked. Figure 1-19*a* 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 binary count on the outputs to be incremented by 1. As shown in Figure 1-19*b*, 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.

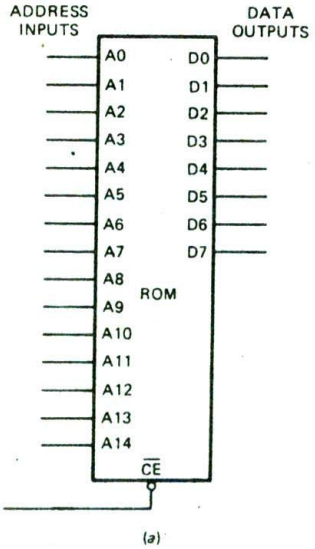
ROMs, RAMs, and Buses

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.

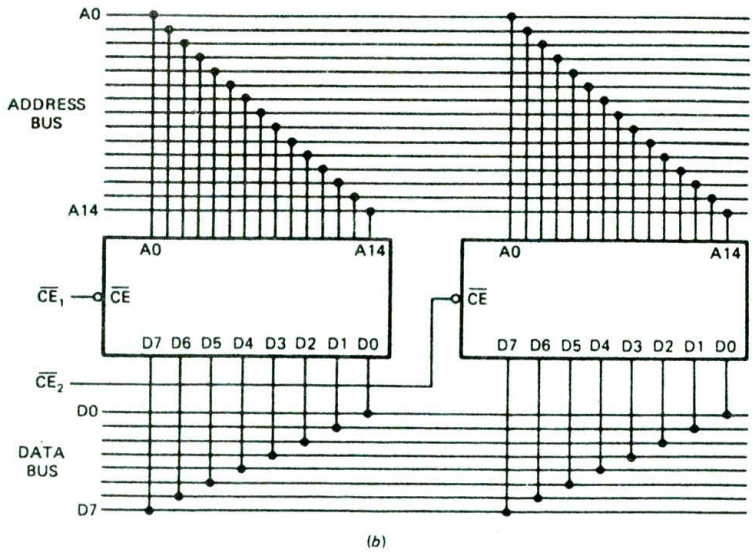
ROMS

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 1-20*a* 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 $\overline{CE}$ input of the ROM is not asserted, then all the outputs will be in the high-



(a)



(b)

FIGURE 1-20 ROMs. (*a*) Schematic symbol. (*b*) Connection in parallel.

impedance state. Most ROMs also switch to a lower-power-consumption standby mode if $\overline{CE}$ is not asserted. If the $\overline{CE}$ 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 Figure 1-20*a* 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 × 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 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 Figure 1-20*b*. 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/$\overline{W}$ 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/$\overline{W}$ 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/$\overline{W}$ 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

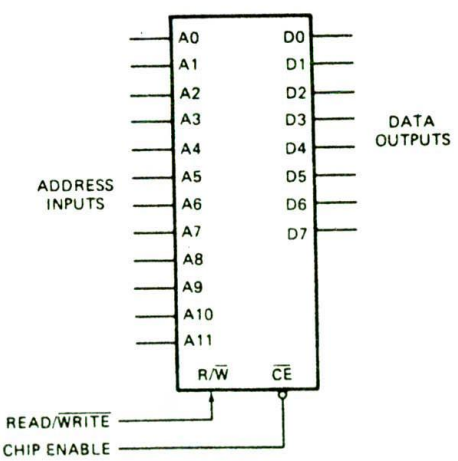


FIGURE 1-21 RAM schematic symbol.

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.

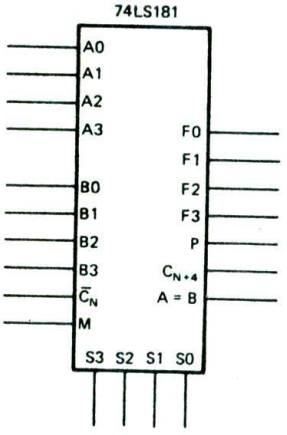
Arithmetic Logic Units

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. Figure 1-22*a* 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 1-22*b* 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. Figure 1-22*c* 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. Figure 1-22*c* shows the result that will be produced by ORing two 4-bit words. Figure 1-22*c* also shows for your reference the result that would be produced by exclusive ORing these two 4-bit words together.

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, $\overline{C}_N$, allows a carry from an operation on previous words to be included in the current operation. If the $\overline{C}_N$



(*a*)

SELECTION				ACTIVE–HIGH DATA		
				M = H LOGIC FUNCTIONS	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0		$\overline{C}_N$ = H (NO CARRY)	$\overline{C}_N$ = L (WITH CARRY)
L	L	L	L	F = $\overline{A}$	F = A	F = A PLUS 1
L	L	L	H	F = $\overline{A+B}$	F = A + B	F = (A + B) PLUS 1
L	L	H	L	F = $\overline{A}$B	F = A + $\overline{B}$	F = (A + $\overline{B}$) PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's COMPL)	F = 0
L	H	L	L	F = $\overline{AB}$	F = A PLUS A$\overline{B}$	F = A PLUS A$\overline{B}$ PLUS 1
L	H	L	H	F = $\overline{B}$	F = (A + B) PLUS A$\overline{B}$	F = (A + B) PLUS A$\overline{B}$ PLUS 1
L	H	H	L	F = A⊕B	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	F = A$\overline{B}$	F = A$\overline{B}$ MINUS 1	F = A$\overline{B}$
H	L	L	L	F = $\overline{A}$ + B	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	F = A⊕B	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	F = (A + $\overline{B}$) PLUS AB	F = (A + $\overline{B}$) PLUS AB PLUS 1
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS A*	F = A PLUS A PLUS 1
H	H	L	H	F = A + $\overline{B}$	F = (A + B) PLUS A	F = (A + B) PLUS A PLUS 1
H	H	H	L	F = A + B	F = (A + $\overline{B}$) PLUS A	F = (A + $\overline{B}$) PLUS A PLUS 1
H	H	H	H	F = A	F = A MINUS 1	F = A

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

(*b*)

A = A3 A2 A1 A0
B = B3 B2 B1 B0
F = F3 F2. F1 F0

A = 1 0 1 0
B = 0 1 1 0
F = A + B = 1 1 1 0

A = 1 0 1 0
B = 0 1 1 0
F = A · B = 0 0 1 0

A = 1 0 1 0
B = 0 1 1 0
F = A⊕B = 1 1 0 0

(*c*)

FIGURE 1-22 Arithmetic logic unit (ALU). (*a*) Schematic symbol. (*b*) Truth table. (*c*) Sample AND, OR, and XOR operations.

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 $\overline{C}_N$ 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

FPLA, 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

1. Write the decimal equivalent for each integral power of 2 from 2^0 to 2^{20}.
2. Convert the following decimal numbers to binary:
   a. 22
   b. 76
   c. 500
3. Convert the following binary numbers to decimal:
   a. 1011
   b. 11010001
   c. 1110111001011001
4. Convert to hexadecimal:
   a. 53 decimal
   b. 756 decimal
   c. 01101100010 binary
   d. 11000010111 binary
5. Convert to decimal:
   a. D3H
   b. 3FEH
   c. 44H
6. Convert the following decimal numbers to BCD:
   a. 86
   b. 62
   c. 33
7. 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?
8. Define *parity* and describe how it is used to detect an error in transmitted data.
9. Show addition of:
   a. $1001 1_2$ and 1011_2 in binary
   b. 37_{10} and 25_{10} in BCD
   c. 4AH and 77H
10. Express the following decimal numbers in 8-bit sign-and-magnitude form:
    a. +26
    b. −7
    c. −26
    d. −125
11. Show the subtraction, in binary, of the following decimal numbers using both the pencil method and the 2's-complement addition method:
    a. 7 − 4
    b. 37 − 26
    c. 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 Figure 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?

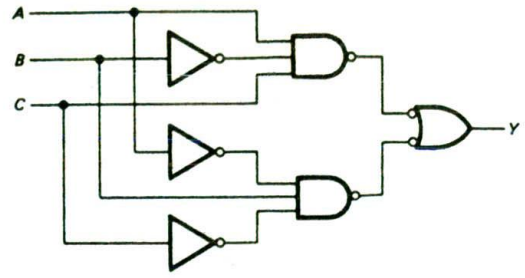


FIGURE 1-23 Circuit for problem 15.

19. Using Figure 1-22*b*, 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?

CHAPTER 2

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

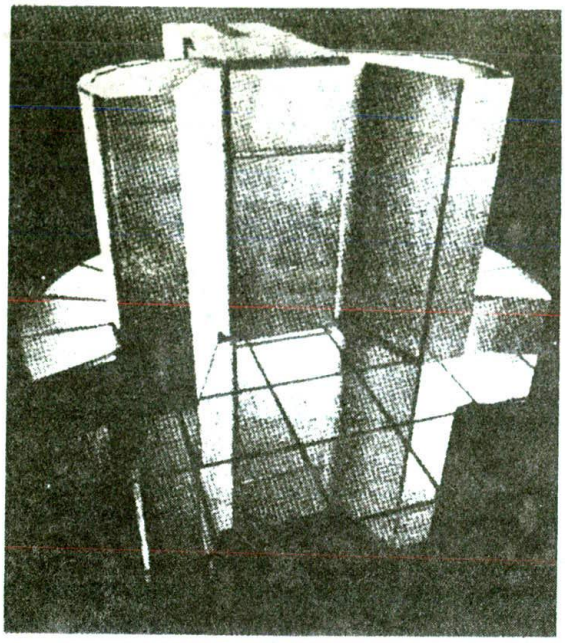
Computers come in a wide variety of sizes and capabilities. The largest and most powerful are often called *mainframes*. Mainframe computers may fill an entire room. They are designed to work at very high speeds with large data words, typically 64 bits or greater, and they have massive amounts of memory. Computers of this type are used for military defense 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 most powerful mainframes are called *supercomputers*. Figure 2-1*a*, p. 20, shows a photograph of a Cray Y-MP/832 supercomputer, which contains eight central processors and 32 million 64-bit words of memory.

Minicomputers

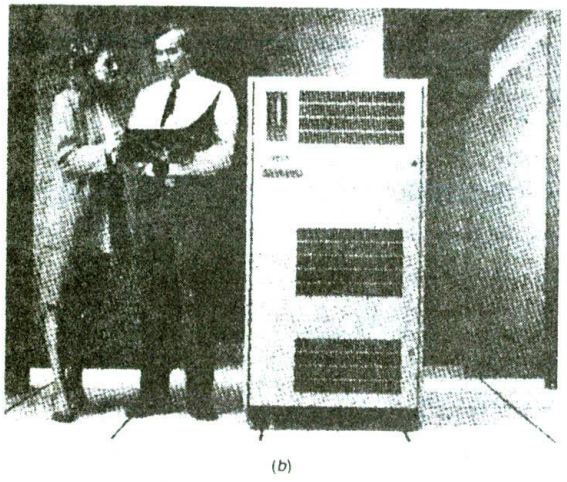
Scaled-down versions of mainframe computers are often called *minicomputers*. The main unit of a minicomputer usually fits in a single rack or box. A minicomputer runs more slowly, works directly with smaller data words (often 32-bit words), and does not have as much memory as a mainframe. Computers 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. Figure 2-1*b* 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. Figure 2-2*a*, p. 21, shows the SDK-86 board, and Figure 2-2*b* shows the Compaq 386/25 system.



(a)



(b)

FIGURE 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.*)

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.

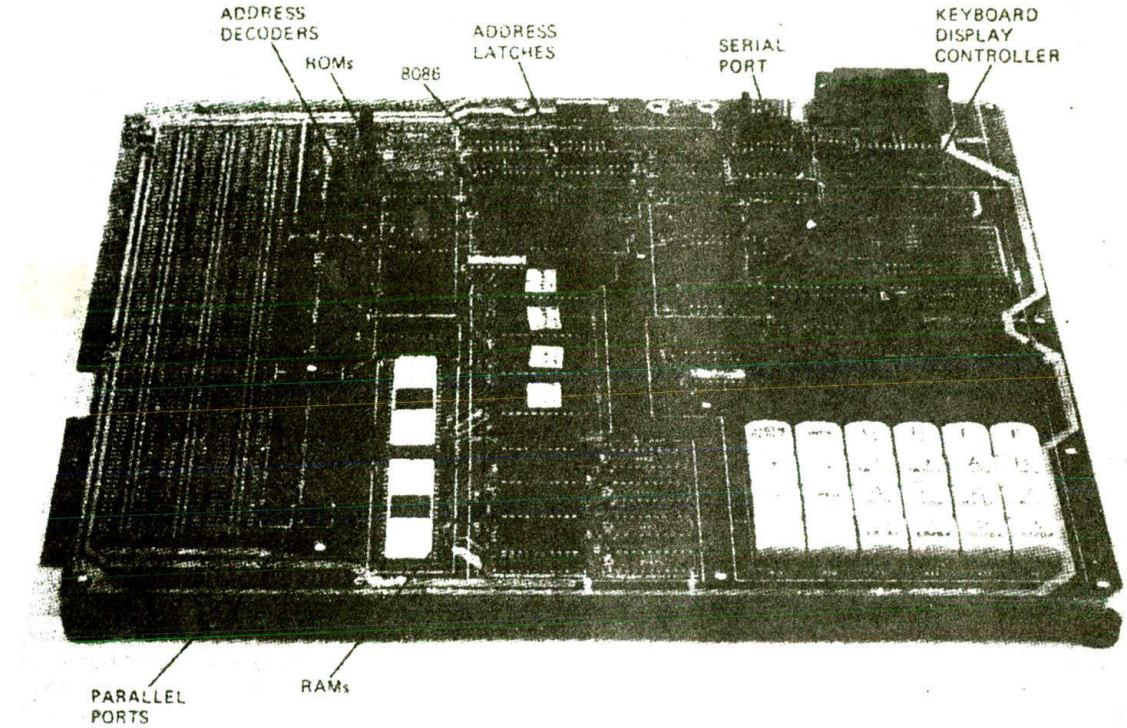
Computerizing an Electronics Factory—Problem

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 as possible 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 bookkeeping. 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 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.

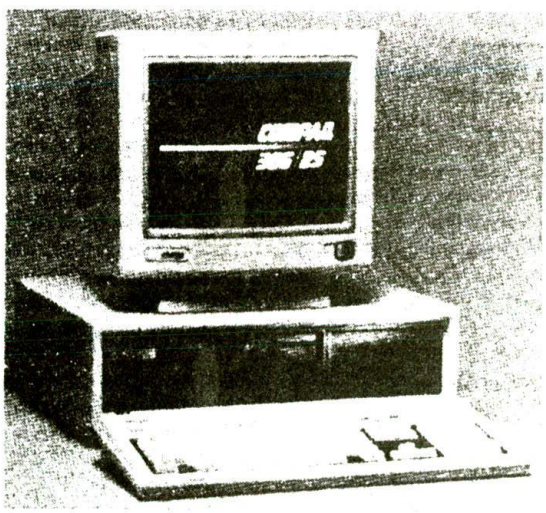
TIMESHARING AND MULTITASKING SYSTEMS

One common method of providing computer access is a *timesharing* system such as shown in Figure 2-3, p. 22. 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.

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



(*a*)



(*b*)

FIGURE 2-2 (*a*) Photograph of Intel SDK-86 board. (*Intel Corp.*) (*b*) Photograph of Compaq 386/25. (*Compaq Corp.*)

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 *data base*. 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*

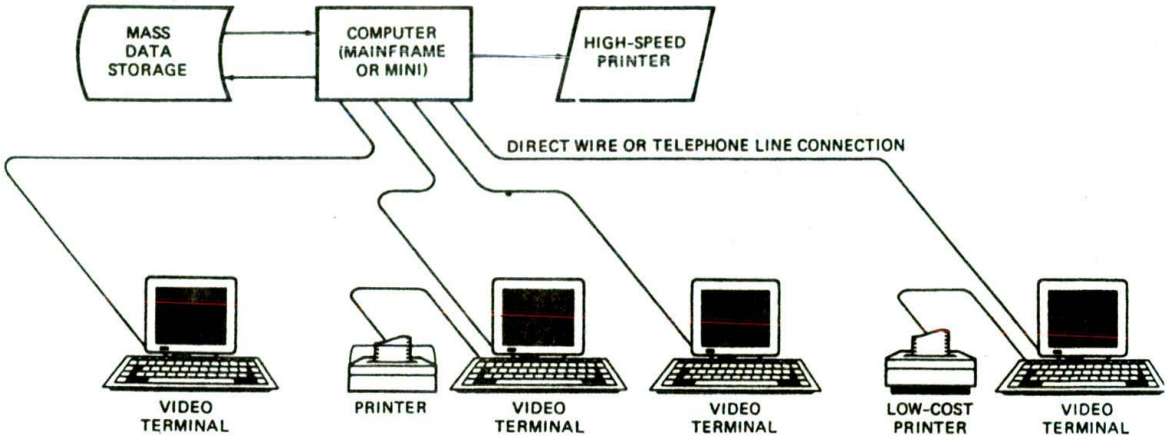


FIGURE 2-3 Block diagram of a computer timesharing system.

processing system. Figure 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, however, 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 can continue working until they need to access the large computer for something. Second, the burden on the

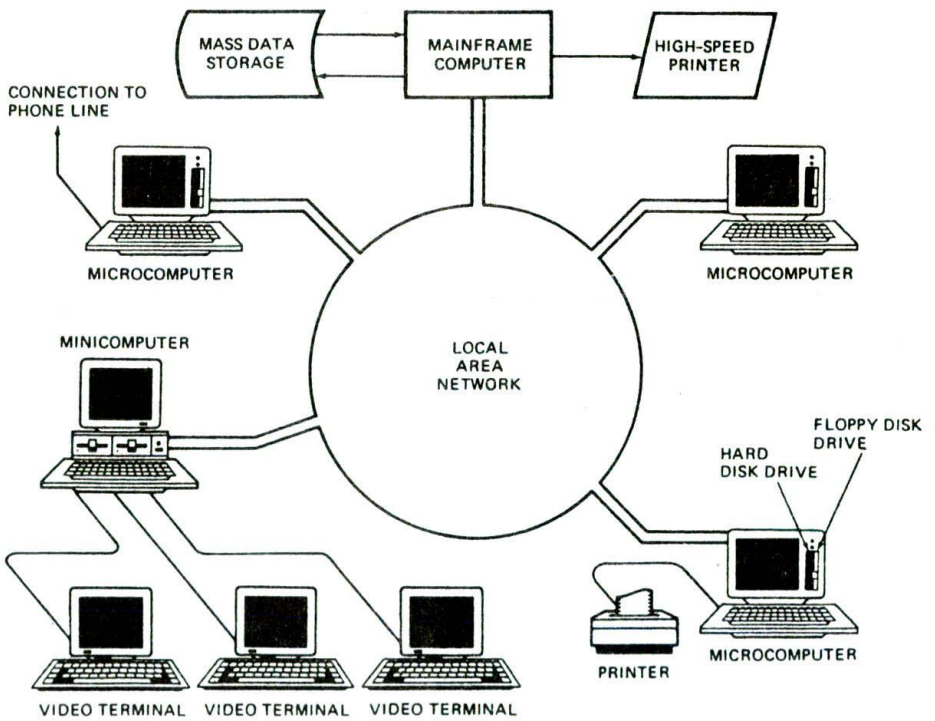


FIGURE 2-4 Block diagram of a distributed processing computer system.

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 workstations 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 those in 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 the inventory records of a supermarket, for example.

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,

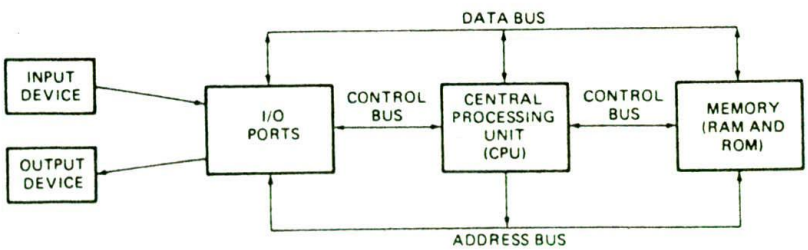


FIGURE 2-5 Block diagram of a simple microcomputer.

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.

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 Figure 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

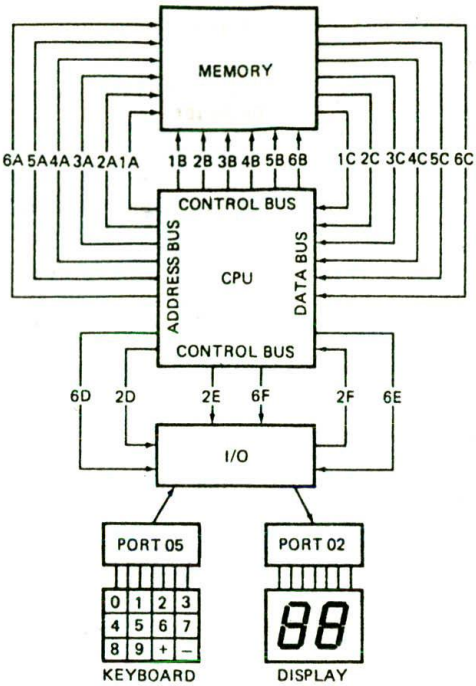
- A computer or microcomputer consists of memory, a CPU, and some input/output circuitry.
- These three parts are connected by the address bus, the data bus, and the control bus.
- The sequence of instructions or program for a computer is stored as binary numbers in successive memory locations.
- The 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 read in.
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.



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
00101H	00000101	05	PORT 05H
00102H	00000100	04	ADD
00103H	00000111	07	07H
00104H	11100110	E6	OUTPUT TO
00105H	00000010	02	PORT 02

(*b*)

FIGURE 2-6 (*a*) Execution of a three-step computer program. (*b*) Memory addresses and memory contents for a three-step program.

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. Figure 2-6*b* 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 Figure 2-6*a*. 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 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 has been *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 toward 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 arithmetic logic unit, 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.

The Intel 80386 is a 32-bit microprocessor which can directly address up to 4 gigabytes of memory. The 80386 contains more sophisticated features than the 80286 for use in multiuser and multitasking microcomputer systems. In Chapter 15 we discuss the features of the 80386 and the 80486, which is an evolutionary step up from the 80386.

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 Figure 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 Figure 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 which 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. Figure 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 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.

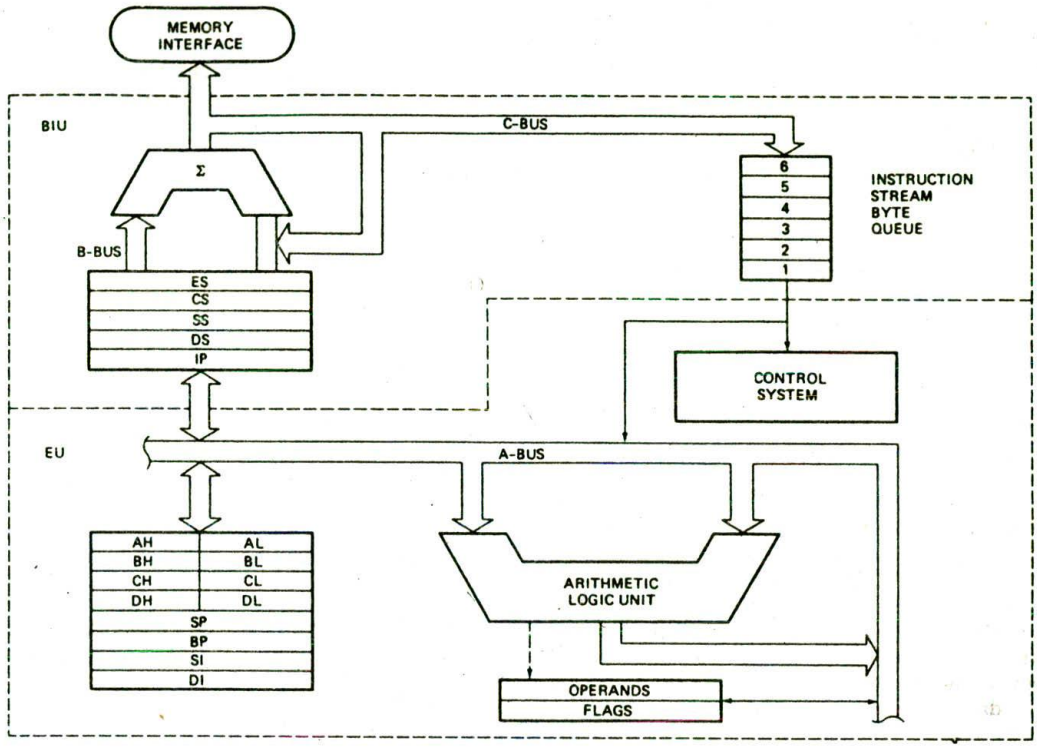


FIGURE 2-7 8086 internal block diagram. (*Intel Corp.*)

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 Figure 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

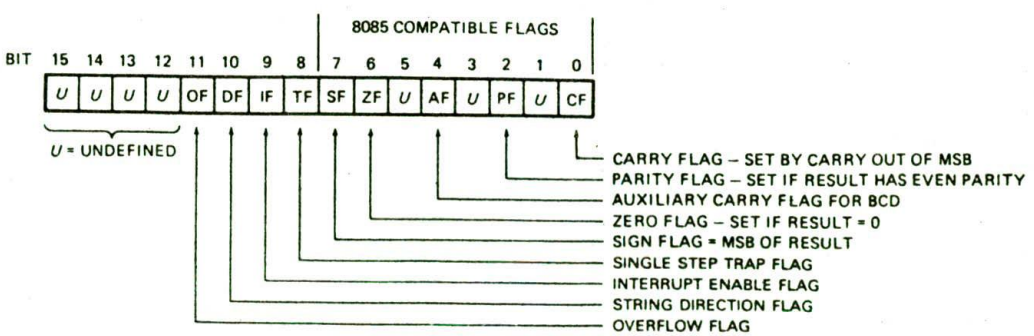


FIGURE 2-8 8086 flag register format. (*Intel Corp.*)

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

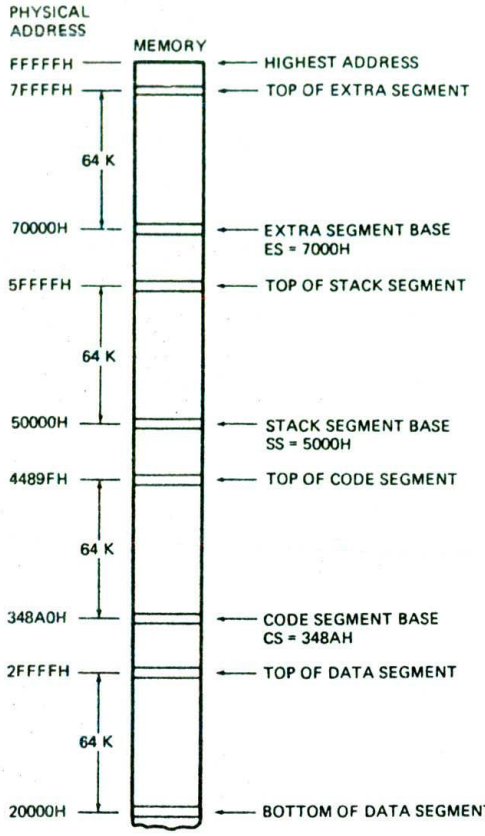


FIGURE 2-9 One way four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086.

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

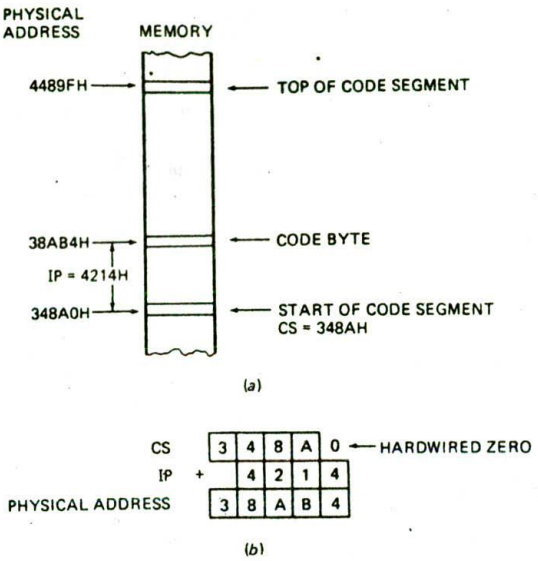


FIGURE 2-10 Addition of IP to CS to produce the physical address of the code byte. (a) Diagram. (b) Computation.

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. Figure 2-10*a* 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. Figure 2-10*b* 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 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*. Figure 2-11*a* 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. Figure 2-11*b* shows 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.

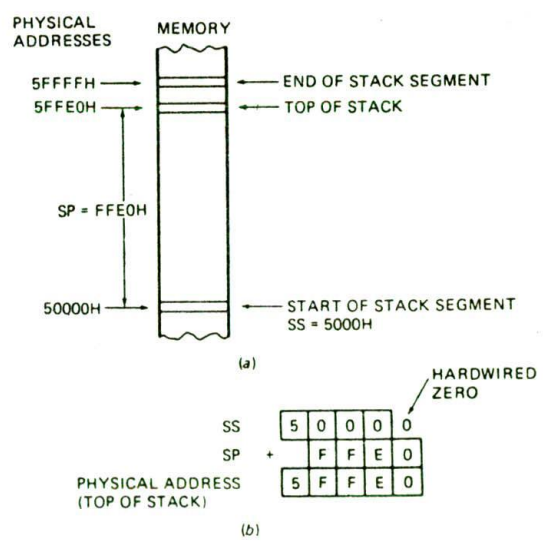


FIGURE 2-11 Addition of SS and SP to produce the physical address of the top of the stack. (a) Diagram. (b) Computation.

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 Figure 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 Figure 2-6*b* 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.

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.

LABEL FIELD	OP CODE FIELD	OPERAND FIELD	COMMENT FIELD
NEXT:	ADD	AL, 07H	; ADD CORRECTION FACTOR

FIGURE 2-12 Assembly language program statement format.

Assembly language statements are usually written in a standard form that has four *fields*, as shown in Figure 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 Figure 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 Figure 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 Figure 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 Figure 2-12 is the *comment field*, 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 the 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 Figure 2-6*b*.

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 is the one most often used. Figure 2-13*a* shows in graphic form how the EA is added to the data segment base to point to an operand in memory. Figure 2-13*b* shows how the 20-bit physical address is generated by the BIU. The starting address for the data segment in Figure 2-13*b* 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

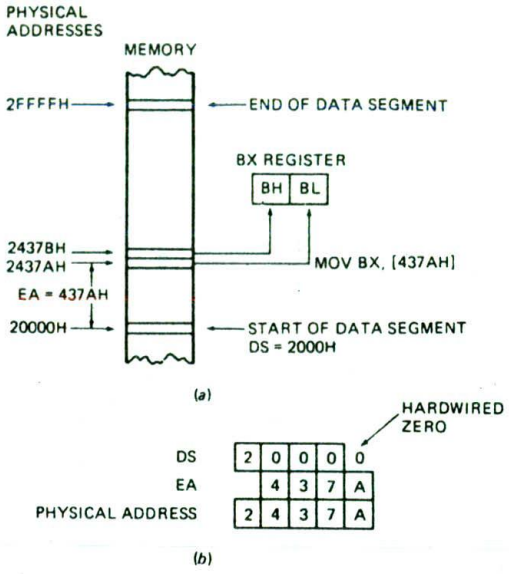


FIGURE 2-13 Addition of data segment register and effective address to produce the physical address of the data byte. (*a*) Diagram. (*b*) Computation.

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 Figure 2-13*a*. The BIU calculates the 20-bit physical memory address by adding the effective address 437AH to the data segment base, as shown in Figure 2-13*b*. 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 Figure 2-13*a*. The contents of the next higher address, displacement 437BH, will be copied into the BH register, as shown by the left arrow in Figure 2-13*a*. 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 Figure 2-13*a*.

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 time-sharing 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. MOV AL, 0DBH
    c. MOV DH, CL
    d. MOV BX, 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].

CHAPTER 3

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. Figure 3-1, p. 38, 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, p. 38, 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*

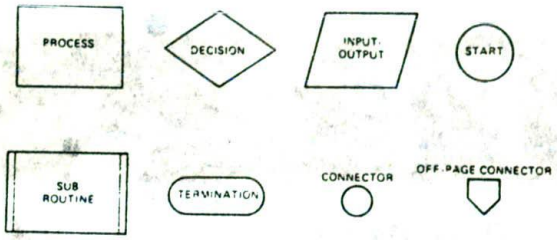


FIGURE 3-1 Flowchart symbols.

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 Figure 3-2 is an example.

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 subprogram, then just "call" this subprogram 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 Figure 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.

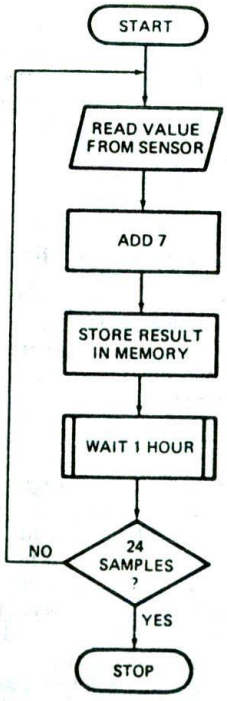


FIGURE 3-2 Flowchart for program to read in 24 data samples from a port, correct each value, and store each in a memory location.

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 Figure 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 Figure 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 Figure 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. Figure 3-3, p. 40, 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. Figure 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 Figure 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 Figure 3-3*a* 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 3-3*b* 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 Figure 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 Figure 3-3*d*. 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 Figure 3-3*e* 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

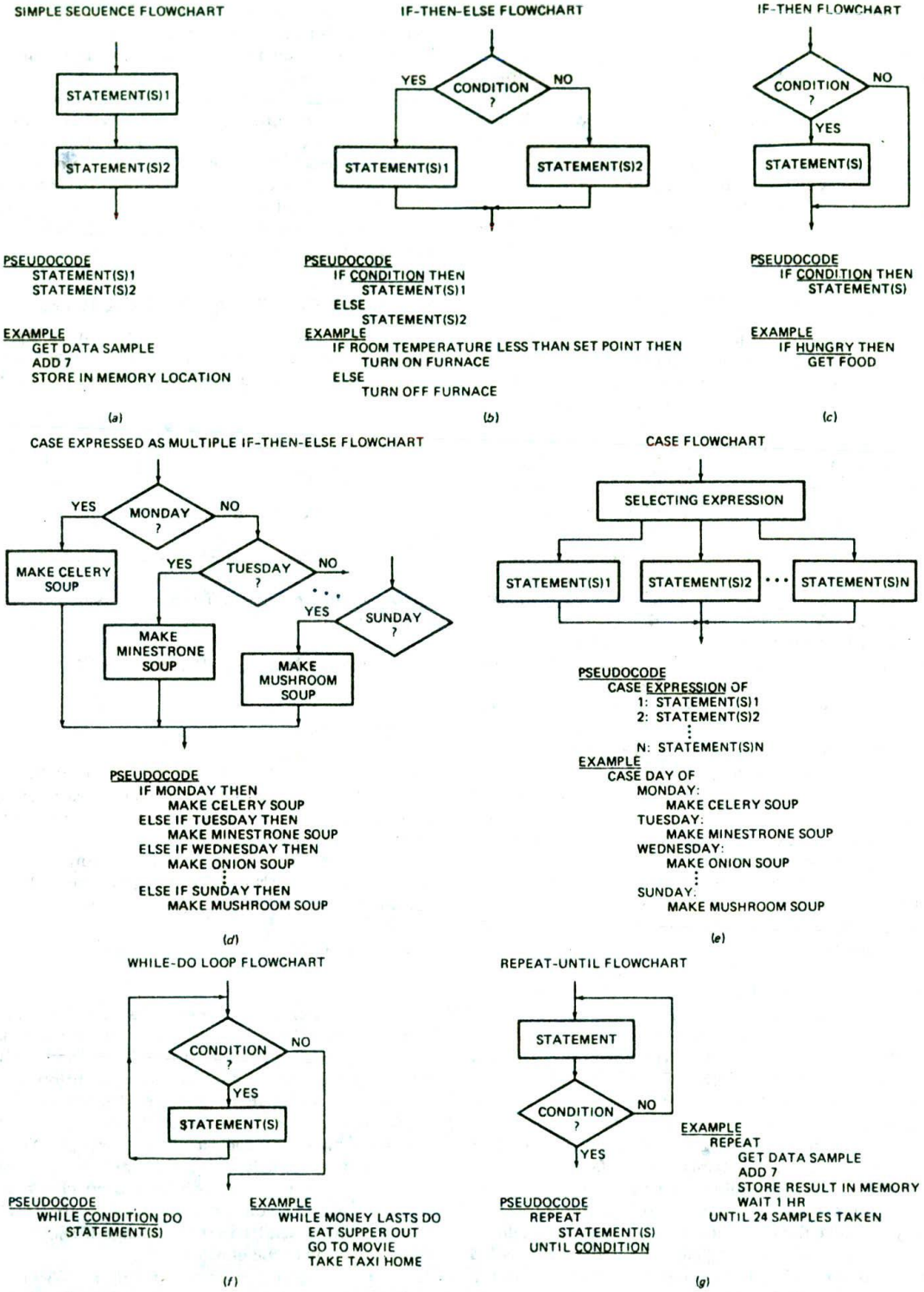


FIGURE 3-3 Standard program structures. (*a*) Sequence. (*b*) IF-THEN-ELSE. (*c*) IF-THEN. (*d*) CASE expressed as nested IF-THEN-ELSE. (*e*) CASE. (*f*) WHILE-DO. (*g*) REPEAT-UNTIL.

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 Figure 3-3*f* 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 Figure 3-3*f* 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 Figure 3-3*g*. 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

```
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.

Division instructions:

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

Logical instructions:

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.

Shift instructions:

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.

Rotate instructions:

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.

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 is used to specifically indicate that a string of bytes is to be acted upon. A "W" in the mnemonic is used to indicate that a string of words is to be acted upon.

REP	An instruction prefix. Repeat following instruction until CX = 0.
REPE/REPZ	An instruction prefix. Repeat instruction until CX = 0 or zero flag ZF ≠ 1.
REPNE/REPNZ	An instruction prefix. Repeat until CX = 0 or ZF = 1.
MOVS/MOVSB/MOVSW	Move byte or word from one string to another.
COMPS/COMPSB/COMPSW	Compare two string bytes or two string words.
INS/INSB/INSW	(80186/80188) Input string byte or word from port.
OUTS/OUTSB/OUTSW	(80186/80188) Output string byte or word to port.
SCAS/SCASB/SCASW	Scan a string. Compare a string byte with a byte in AL or a string word with a word in AX.
LODS/LODSB/LODSW	Load string byte into AL or string word into AX.
STOS/STOSB/STOSW	Store byte from AL or word from AX into string.

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.

High-level language interface instructions:

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 addtion 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 Figure 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, p. 46, 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 here the operation of these instructions

PROGRAMMER D. V. HALL SHEET 1 OF 1

PROGRAM TITLE READ TEMPERATURE & 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
00101					result. This location will be loaded
00102					with a data byte as read in
00103					& corrected by the program.
00104					XX means "don't care" about
00105					contents of location.
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
01	10				memory set aside for storing data
02	00				
03	8E		MOV	DS, AX	
04	D8				
05	E4		IN	AL, 05H	Read temperature from
06	05				port 05H
07	04		ADD	AL, 07H	Add correction factor
08	07				of +07
09	A2		MOV	[0000], AL	Store result in reserved
0A	00				memory
0B	00				
0C	CC		INT	3	Stop, wait for command
0D					from user
0E					
0F					

FIGURE 3-4 Assembly language program on standard coding form.

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 Figure 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 Figure 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

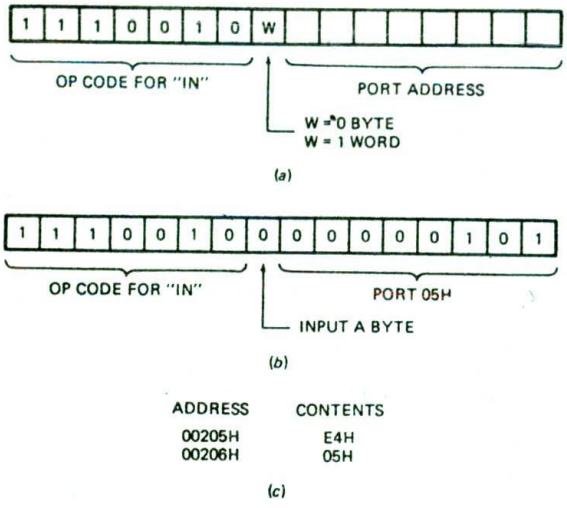


FIGURE 3-5 Coding template for 8086 IN (fixed port) instruction. (*a*) Template. (*b*) Example. (*c*) Hex codes in sequential memory locations.

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.

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 Figure 3-5*a*. 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 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. Figure 3-5*c* 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 Figure 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 Figure 3-7. Look in one of these places to find the code for the CL register. You should get 001.

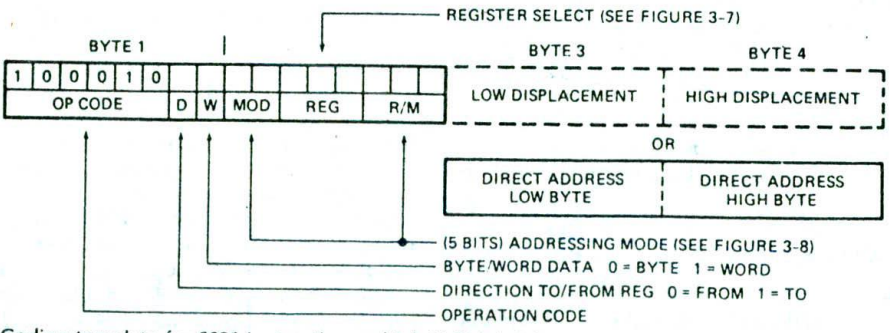


FIGURE 3-6 Coding template for 8086 instructions which MOV data between registers or between a register and a memory location.

REGISTER	W=1	CODE W=0
AL	AX	000
BL	BX	011
CL	CX	001
DL	DX	010
AH	SP	100
BH	DI	111
CH	BP	101
DH	SI	110

SEGREG	CODE
CS	01
DS	11
ES	00
SS	10

FIGURE 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. Figure 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

R/M \ MOD	00	01	10	11	
				W = 0	W = 1
000	[BX] + [SI]	[BX] + [SI] + d8	[BX] + [SI] + d16	AL	AX
001	[BX] + [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

FIGURE 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.

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 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.

4. 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 Figure 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. Figure 3-9*a* 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. Figure 3-9*b* shows the meaning of the bits in this form. This second form, incidentally, is the form that the Intel 8086 Macroassembler produces.

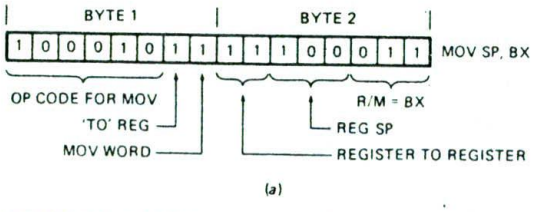
CODING MOV CL, [BX]

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 which represents register CL in the REG field of the second byte of the instruction code. The codes for each register are shown in Figure 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 Figure 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]. Figure 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.



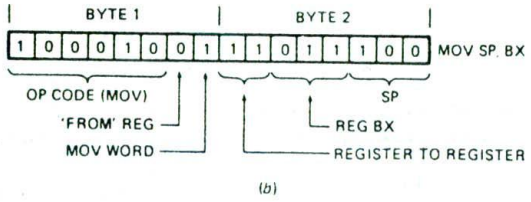


FIGURE 3-9 MOV instruction coding examples. (*a*) MOV SP, BX. (*b*) MOV SP, BX alternative.

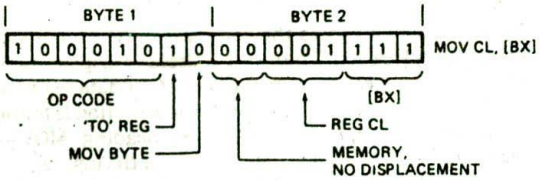


FIGURE 3-10 MOV CL, [BX].

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. Figure 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.

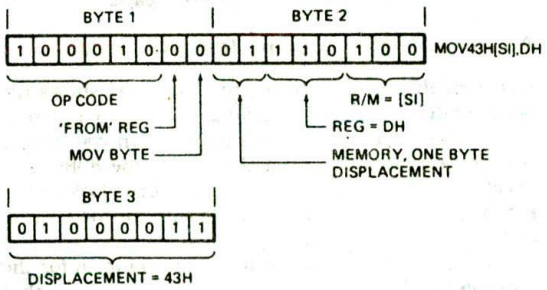


FIGURE 3-11 MOV 43H[SI], DH.

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 Figure 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

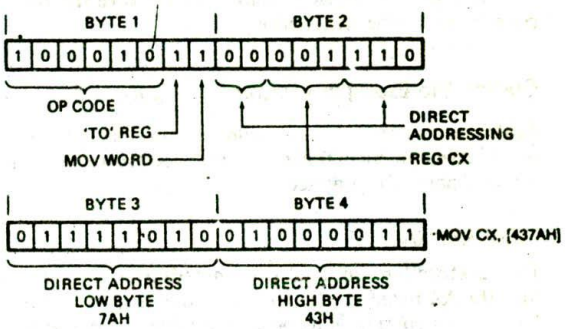


FIGURE 3-12 MOV CX, [437AH].

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. Figure 3-12 spells this out in detail.

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 Figure 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. Figure 3-13 shows the result you should get and how the code

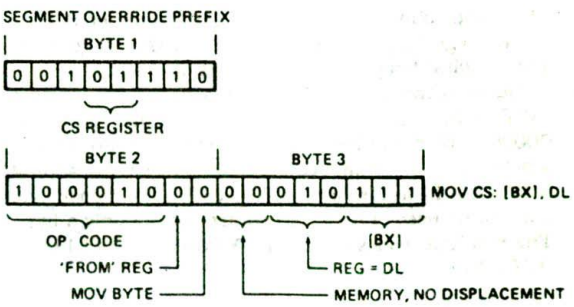


FIGURE 3-13 MOV CS:[BX], DL.

A-2807

for the segment override prefix is put before the other code bytes for the instruction.

Coding the Example Program in Figure 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 Figure 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 Figure 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 10001001 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, p. 54, 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 Figure 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

```
                ; 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.
```

FIGURE 3-14 Assembly language source program to multiply two 16-bit binary numbers to give a 32-bit result.

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 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 Figure 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 Figure 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 Figure 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.

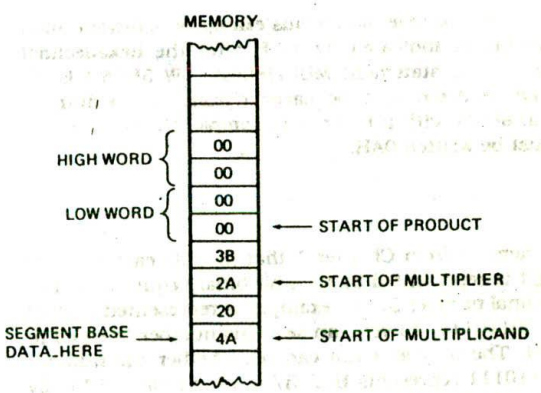


FIGURE 3-15 Data arrangement in memory for multiply program.

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 Figure 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 Figure 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 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 Figure 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 Figure 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 Figure 3-16. When the MUL MULTIPLIER instruction executes, the BIU will add the displacement contained in the instruction to the data

```
Turbo Assembler  Version 1.0                                    Page 1

                                ; 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

 0000                         DATA_HERE    SEGMENT
 0000  204A                                MULTIPLICAND  DW 204AH      ; First word here
 0002  382A                                MULTIPLIER    DW 382AH      ; Second word here
 0004  02*(0000)                           PRODUCT       DW 2 DUP(0)   ; Result of multiplication here
 0008                         DATA_HERE    ENDS

                              CODE_HERE    SEGMENT
                                           ASSUME    CS:CODE_HERE, DS:DATA_HERE
 0000  B8 0000s               START:       MOV  AX, DATA_HERE          ; Initialize DS register
 0003  8E D8                               MOV  DS, AX
 0005  A1 0000r                            MOV  AX, MULTIPLICAND       ; Get one word
 0008  F7 26 0002r                         MUL  MULTIPLIER             ; Multiply by second word
 000C  A3 0004r                            MOV  PRODUCT, AX            ; Store low word of result
 000F  89 16 0006r                         MOV  PRODUCT+2, DX          ; Store high word of result
 0013  CC                                  INT  3                      ; Wait for command from user
 0014                         CODE_HERE    ENDS
                                           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
@WORDSIZE                     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
```

FIGURE 3-16 Assembler listing for example program in Figure 3-14.

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 Figure 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 Figure 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 Figure 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. Figure 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 SEGMENT, 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 Figure 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 Figure 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 Figure 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 Figure 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 Figure 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 Figure 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. Figure 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.

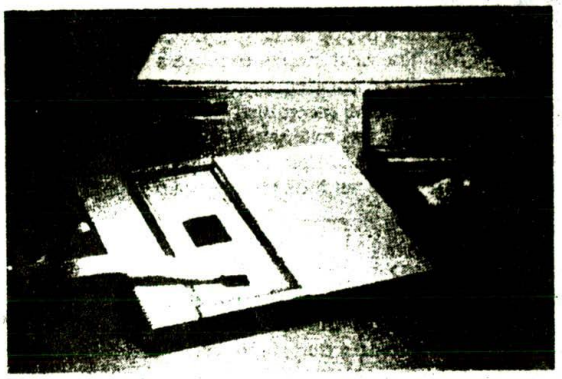


FIGURE 3-17 Applied Microsystems ES 1800 16-bit emulator. *(Applied Microsystems Corp.)*

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 Figure 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 Figure 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 Figure 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 Figure 3-16 gives some additional information about the segments and names used in the program. The statement CODE_HERE 16 0014 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 SDKCOM1 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 Figure 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

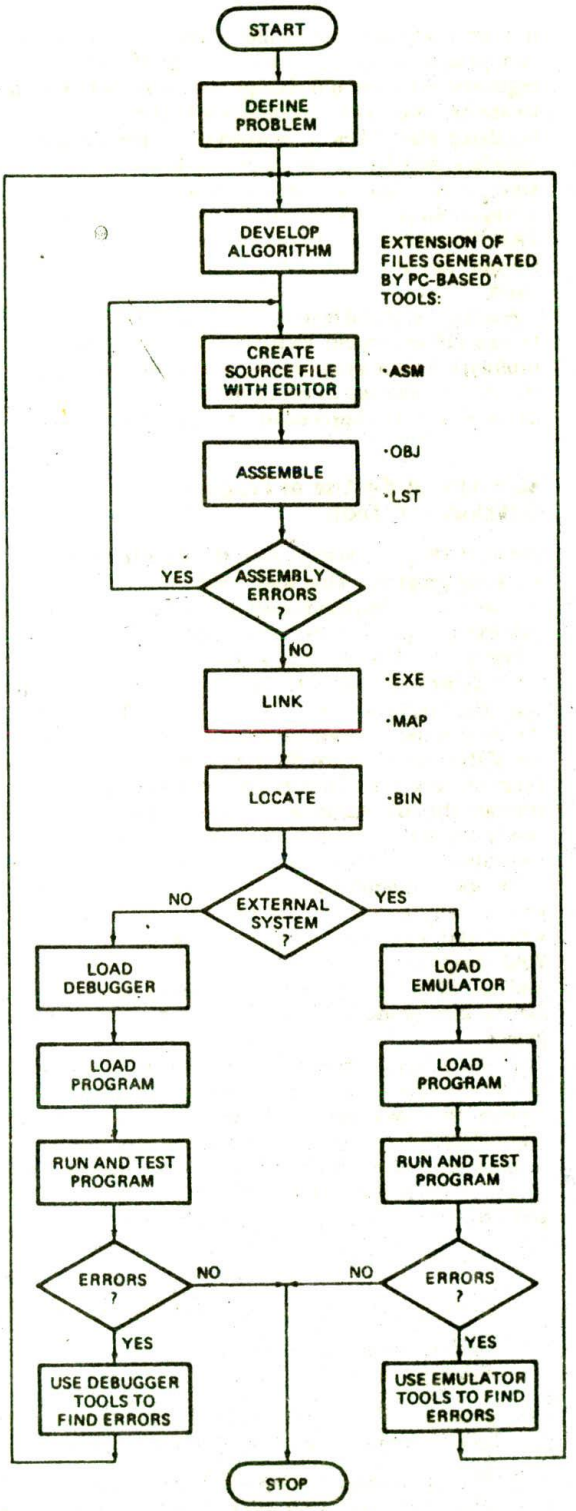


FIGURE 3-18 Program development algorithm (see p. 62).

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.

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 (p. 61) 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 Figure 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

1. List the major steps in developing an assembly language program.
2. What is the main advantage of a top-down design approach to solving a programming problem?
3. Why should you develop a detailed algorithm for a program before writing down any assembly language instructions?
4. *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?
5. 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 Figure 3-19, answer the following questions:
   a. What physical address will the next instruction be fetched from?
   b. What is the physical address for the top of the stack?

ES	6000
CS	4000
SS	7000
DS	5000
IP	43E8
SP	0000
BP	2468
SI	4C00
DI	7D00

DATA SEGMENT

5000CH	D7
5000BH	9A
5000AH	7C
50009H	DB
50008H	C3
50007H	B2
50006H	49
50005H	21
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

FIGURE 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 Figure 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 Figure 3-19. (Use Chapter 6.)
   a. ADD BL, AL
   MOV [0004], BL
   b. MOV CL, 04
   ROR DI, CL
   c. ADD AL, BH
   DAA
   d. MOV BX, 000AH
   MOV AL, [BX]
   SUB AL, CL
   INC BX
   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.
   a. MOV BL, AL
   b. MOV [BX], CX
   c. ADD BX, 591H[DI]
   d. SUB [2048], DH
   e. XCHG CH, ES:[BX]
   f. ROR AX, 1
   g. OUT DX, AL
   h. AND AL, OFH
   i. NOP
   j. IN AL, DX

13. Describe the function of each assembler directive and instruction statement in the short program shown in Figure 3-20.

```
;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
```

FIGURE 3-20 Program for Problem 13.

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 Figure 3-18, p. 61.

CHAPTER 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 Figure 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 Figure 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

```
 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                                      ASSUME CS:CODE, DS:DATA
16 0000  B8 0000s               START:  MOV  AX, DATA       ; Initialize data segment
17 0003  8E D8                          MOV  DS, AX
18 0005  A0 0000r                       MOV  AL, HI_TEMP    ; Get first temperature
19 0008  02 06 0001r                    ADD  AL, LO_TEMP    ; Add second to it
20 000C  B4 00                          MOV  AH, 00H        ; Clear all of AH register
21 000E  80 D4 00                       ADC  AH, 00H        ; Put carry in LSB of AH
22 0011  B3 02                          MOV  BL, 02H        ; Load divisor in BL register
23 0013  F6 F3                          DIV  BL             ; Divide AX by BL. Quotient in AL,
24                                                          ; and remainder in AH
25 0015  A2 0002r                       MOV  AV_TEMP, AL    ; Copy result to memory
26 0018                         CODE    ENDS
27                                      END  START
```

FIGURE 4-1 8086 program to average two temperatures.

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 Figure 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 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 Figure 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 Figure 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*. Figure 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. Figure 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
```

FIGURE 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. Figure 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

FIGURE 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,0FH 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 Figure 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

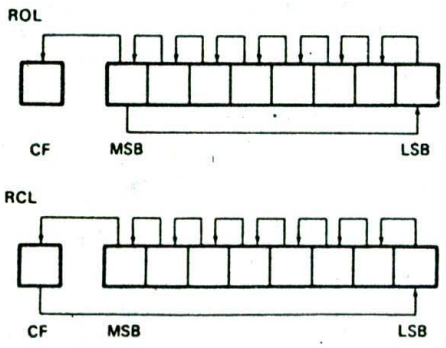


FIGURE 4-4 ROL instruction and RCL instruction operations for byte operands.

should help you remember that the carry flag is included in the rotated loop when the RCL instruction executes.

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 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 Figure 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.

```
 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, 0FH   ; Mask upper 4 bits of first digit
14 0007  24 0F                         AND  AL, 0FH   ; 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
```

FIGURE 4-5 List file of 8086 assembly language program to produce packed BCD from two ASCII characters.

For the example program here, we use the instruction OR AL,BL to pack the two BCD nibbles. Bits ORed with 0's will not be changed. Bits ORed with 1's will become or stay 1's. Again look at Figure 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 Figure 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. Figure 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 Figure 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.

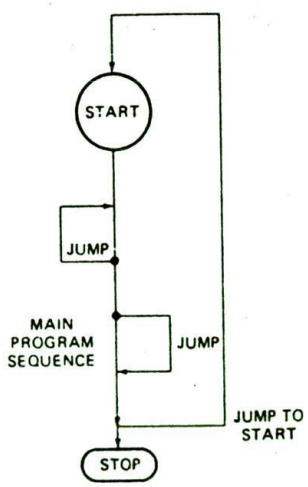


FIGURE 4-6 Change in program flow that can be caused by jump instructions.

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

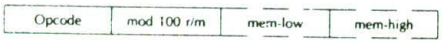
The 8086 Unconditional Jump instruction, JMP, has five different types. Figure 4-7 shows the names and instruction coding templates for these five types. We will first summarize how these five types work to give you an overview; then we will describe in detail the two types you need for your programs at this point. The JMP instruction description in Chapter 6 shows examples of each of the five types.

JMP = Jump

Within segment or group, IP relative—near and short

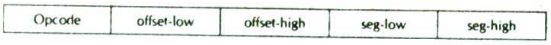
Opcode	Clocks	Operation
E9	15	IP ← IP + Disp16
EB	15	IP ← IP + Disp8 (Disp8 sign-extended)

Within segment or group, Indirect



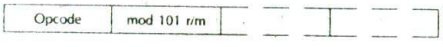
Opcode	Clocks	Operation
FF	11	IP ← Reg16
FF	18+EA	IP ← Mem16

Inter-segment or group, Direct



Opcode	Clocks	Operation
EA	15	CS ← segbase IP ← offset

Inter-segment or group, Indirect



Opcode	Clocks	Operation
FF	24+EA	CS ← segbase IP ← offset

FIGURE 4-7 8086 Unconditional Jump instructions. (*Intel Corporation*)

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.

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 Figure 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. Figure 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 Figure 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 Figure 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 Figure 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 Figure 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 Figure 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 Figure 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 Figure 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 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 Figure 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

```
 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            CODE   ENDS
13                         END
```

FIGURE 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  B8 0000                 THERE: MOV AX, 0000H ; Zero accumulator before addition instructions
12 0008  90                              NOP            ; Dummy instruction to represent continuation of execution
13 0009                          CODE   ENDS
14                                      END
```

FIGURE 4-9 List file of program demonstrating "forward" JMP.

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 Figure 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 the JMP instruction to the label. If the assembler finds the displacement to be outside the range of −128 bytes to +127 bytes, then it will code the instruction as a direct within-segment near JMP with 2 bytes of displacement. If the assembler finds the displacement to be within the −128- to +127-byte range, then it will code the instruction as a direct within-segment short-type JMP with a 1-byte displacement. In the latter case, the assembler will put the code for a NOP instruction, 90H, in the third byte it had reserved for the JMP instruction. The instruction codes for the JMP THERE instruction on line 8 of Figure 4-9 demonstrate this. As shown in the instruction template in Figure 4-7, EBH is the basic opcode for the direct within-segment short JMP. The 03H represents the displacement to the JMP destination. Since we are jumping forward in this case, the displacement is a positive number. The 90H in the next memory byte is the code for a NOP instruction. The displacement is calculated from the offset of this NOP instruction, 0002H, to the offset of the destination label, 0005H. The difference of 03H between these two is the displacement you see coded in the instruction.

If you are hand coding a program such as this, you will probably know how far it is to the label, and you can leave just 1 byte for the displacement if that is enough. If you are using an assembler and you don't want to waste the byte of memory or the time it takes to fetch the extra NOP instruction, you can write the instruction as JMP SHORT label. The SHORT operator is a promise to the assembler that the destination will not be outside the range of −128 to +127 bytes. Trusting your promise, the assembler then reserves only 1 byte for the displacement.

Note that if you are making a JMP from an address near the start of a 64-Kbyte segment to an address near the end of the segment, you may not be able to get there with a jump of +32,767. The way you get there is to JMP backward around to the desired destination address. An assembler will automatically do this for you.

One advantage of the direct near- and short-type JMPs is that the destination address is specified *relative* to the address of the instruction after the JMP instruction. Since the JMP instruction in this case does not contain an absolute address or offset, the program can be loaded anywhere in memory and still run correctly. A program which can be loaded anywhere in memory to be run is said to be *relocatable*. You should try to write your programs so that they are relocatable.

Now that you know about unconditional JMP instructions, we will discuss the 8086 flags, so that we can show how the 8086 Conditional Jump instructions are used to implement the rest of the standard programming structures.

The 8086 Conditional Flags

The 8086 has six *conditional flags*. They 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). Chapter 1 shows numerical examples of some of the conditions indicated by these flags. Here we review these conditions and show how some of the important 8086 instructions affect these flags.

THE CARRY FLAG WITH ADD, SUBTRACT, AND COMPARE INSTRUCTIONS

If the addition of two 8-bit numbers produces a sum greater than 8 bits, the carry flag will be set to a 1 to indicate a carry into the next bit position. Likewise, if

the addition of two 16-bit numbers produces a sum greater than 16 bits, then the carry flag will be set to a 1 to indicate that a final carry was produced by the addition.

During subtraction, the carry flag functions as a borrow flag. If the bottom number in a subtraction is larger than the top number, then the carry/borrow flag will be set to indicate that a borrow was needed to perform the subtraction.

The 8086 compare instruction has the format CMP destination,source. The source can be an immediate number, a register, or a memory location. The destination can be a register or a memory location. The comparison is done by subtracting the contents of the specified source from the contents of the specified destination. Flags are updated to reflect the result of the comparison, but neither the source nor the destination is changed. If the source operand is greater than the specified destination operand, then the carry/borrow flag will be set to indicate that a borrow was needed to do the comparison (subtraction). If the source operand is the same size as or smaller than the specified destination operand, then the carry/borrow flag will not be set after the compare. If the two operands are equal, the zero flag will be set to a 1 to indicate that the result of the compare (subtraction) was all 0's. Here's an example and summary of this for your reference.

CMP BX, CX

condition	CF	ZF
CX > BX	1	0
CX < BX	0	0
CX = BX	0	1

The compare instruction is very important because it allows you to easily determine whether one operand is greater than, less than, or the same size as another operand.

THE PARITY FLAG

Parity is a term used to indicate whether a binary word has an even number of 1's or an odd number of 1's. A binary number with an even number of 1's is said to have *even parity*. The 8086 parity flag will be set to a 1 after an instruction if the lower 8 bits of the destination operand has an even number of 1's. Probably the most common use of the parity flag is to determine whether ASCII data sent to a computer over phone lines or some other communications link contains any errors. In Chapter 14 we describe this use of parity.

THE AUXILIARY CARRY FLAG

This flag has significance in BCD addition or BCD subtraction. If a carry is produced when the least significant nibbles of 2 bytes are added, the auxiliary carry flag will be set. In other words, a carry out of bit 3 sets the auxiliary carry flag. Likewise, if the subtraction of the least significant nibbles requires a borrow, the auxiliary carry/borrow flag will be set. The auxiliary carry/borrow flag is used only by the DAA and DAS instructions. Consult the DAA and DAS instruction descriptions in Chapter 6 and the BCD operation examples section of Chapter 1 for further discussion of addition and subtraction of BCD numbers.

THE ZERO FLAG WITH INCREMENT, DECREMENT, AND COMPARE INSTRUCTIONS

As the name implies, this flag will be set to a 1 if the result of an arithmetic or logic operation is zero. For example, if you subtract two numbers which are equal, the zero flag will be set to indicate that the result of the subtraction is zero. If you AND two words together and the result contains no 1's, the zero flag will be set to indicate that the result is all 0's.

Besides the more obvious arithmetic and logic instructions, there are a few other very useful instructions which also affect the zero flag. One of these is the compare instruction CMP, which we discussed previously with the carry flag. As shown there, the zero flag will be set to a 1 if the two operands compared are equal.

Another important instruction which affects the zero flag is the decrement instruction, DEC. This instruction will decrement (or, in other words, subtract 1 from) a number in a specified register or memory location. If, after decrementing, the contents of the register or memory location are zero, the zero flag will be set. Here's a preview of how this is used. Suppose that we want to repeat a sequence of actions nine times. To do this, we first load a register with the number 09H and execute the sequence of actions. We then decrement the register and look at the zero flag to see if the register is down to zero yet. If the zero flag is not set, then we know that the register is not yet down to zero, so we tell the 8086, with a Jump instruction, to go back and execute the sequence of instructions again. The following sections will show many specific examples of how this is done.

The increment instruction, INC destination, also affects the zero flag. If an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result in the destination will be all 0's. The zero flag will be set to indicate this.

THE SIGN FLAG—POSITIVE AND NEGATIVE NUMBERS

When you need to represent both positive and negative numbers for an 8086, you use 2's complement sign-and-magnitude form as described in Chapter 1. In this form, the most significant bit of the byte or word is used as a sign bit. A 0 in this bit indicates that the number is positive. A 1 in this bit indicates that the number is negative. The remaining 7 bits of a byte or the remaining 15 bits of a word are used to represent the magnitude of the number. For a positive number, the magnitude will be in standard binary form. For a negative number, the magnitude will be in 2's complement form. After an arithmetic or logic instruction executes, the sign flag will be a copy of the most significant bit of the destination byte or the destination word. In addition to its use with signed arithmetic operations, the sign flag can be used to determine whether an operand has been decremented beyond zero. Decrementing 00H, for example, will give FFH. Since the MSB of FFH is a 1, the sign flag will be set.

THE OVERFLOW FLAG

This flag will be set if the result of a signed operation is too large to fit in the number of bits available to represent it. To remind you of what *overflow* means, here is an example. Suppose you add the 8-bit signed number 01110101 (+117 decimal) and the 8-bit signed number 00110111 (+55 decimal). The result will be 10101100 (+172 decimal), which is the correct binary result in this case, but is too large to fit in the 7 bits allowed for the magnitude in an 8-bit signed number. For an 8-bit signed number, a 1 in the most significant bit indicates a negative number. The overflow flag will be set after this operation to indicate that the result of the addition has overflowed into the sign bit.

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 Figure 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

MNEMONIC	CONDITION TESTED	"JUMP IF . . ."
JA/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	(CF or 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
JP/JPE	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.

FIGURE 4-10 8086 Conditional Jump instructions.

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 Figure 4-10, the JAE instruction says, "Jump if above or equal" to the label 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 Figure 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. Figure 4-11*a* 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 ≠ BX, then the ADD AX,0002H instruction after the JE instruction will be executed before the MOV CL,07H instruction.

```
        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*)

FIGURE 4-11 Programming conditional jumps. (*a*) Destinations closer than ±128 bytes. (*b*) Destinations further than ±128 bytes.

The implementation in Figure 4-11*a* 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 −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 Figure 4-11*a* to implement an IF-THEN structure. If you are not sure whether the destination will be in range, the instruction sequence shown in Figure 4-11*b* 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. Figure 3-3*b* 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 of actions is done before the program goes on with the next mainline instruction. An example will show how we implement this structure.

Suppose that in the computerized factory we discussed in Chapter 2, we have an 8086 microcomputer which controls a printed-circuit-board-making machine. Part of the job of this 8086 is to check a temperature sensor and turn on a green lamp or a yellow lamp depending on the value of the temperature it reads in. If the temperature is below 30°C, we want to turn on a yellow lamp to tell the operator that the solution is not up to temperature. If the temperature is greater than or equal

to 30°C, we want to light a green lamp. With a system such as this, the operator can visually scan all the lamps on the control panel until all the green lamps are lit. When all the lamps are green, the operator can push the GO button to start making boards. The reason that we have the yellow lamp is to let the operator know that this part of the machine is working, but that the temperature is not yet up to 30°C.

Figure 4-12 shows with flowcharts and with pseudocode two ways we can represent the algorithm for this problem. The difference between the two is simply a matter of whether we make the decision based on the temperature being below 30°C or based on the temperature being above or equal to 30°C. The two approaches are equally valid, but your choice determines which Conditional Jump instruction you use to implement the algorithm. Since this program involves reading data in from a port and writing data out to a port, we need to talk briefly about the 8086 IN and OUT instructions before we discuss the details of how these two algorithms can be implemented in assembly language.

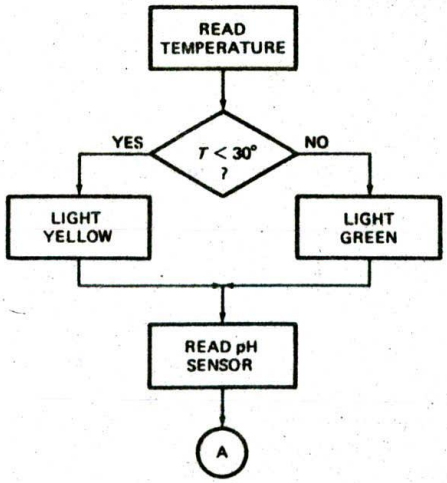
THE 8086 IN AND OUT INSTRUCTIONS

The 8086 has two types of input instruction, *fixed*-port and *variable*-port. The fixed-port instruction has the format IN AL,port or IN AX,port. The term *port* in these instructions represents an 8-bit port address to be put directly in the instruction. The instruction IN AX,04H, for example, will copy a word from port 04H to the AX register. The 8-bit port address in this type of IN instruction allows you to address any one of 256 possible input ports, but the port address is fixed. The program cannot change the port address as it executes. Keep this in mind as we discuss the variable-port IN instruction.

The variable-port input instruction has the format IN AL,DX or IN AX,DX. When using the variable-port input instruction, you must first put the address of the desired 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.

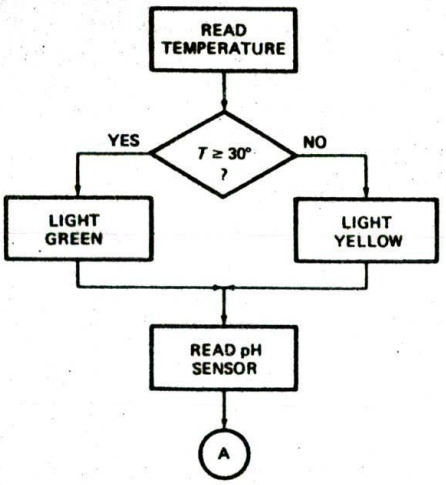
The 8086 also has a fixed-port output instruction and a variable-port output instruction. The fixed-port output instruction has the form OUT port,AL or OUT port,AX. Here again the term *port* represents an 8-bit port address written in the instruction. OUT 0AH,AL, for example, will copy the contents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




```
READ TEMPERATURE
IF TEMPERATURE < 30° THEN
    LIGHT YELLOW LAMP
ELSE
    LIGHT GREEN LAMP
READ pH SENSOR
```

(a)




```
READ TEMPERATURE
IF TEMPERATURE ≥ 30° THEN
    LIGHT GREEN LAMP
ELSE
    LIGHT YELLOW LAMP
READ pH SENSOR
```

(b)

FIGURE 4-12 Flowcharts and pseudocode for two ways of expressing algorithm for printed-circuit-board-making machine. (*a*) Temperature below 30° test. (*b*) Temperature above 30° test.

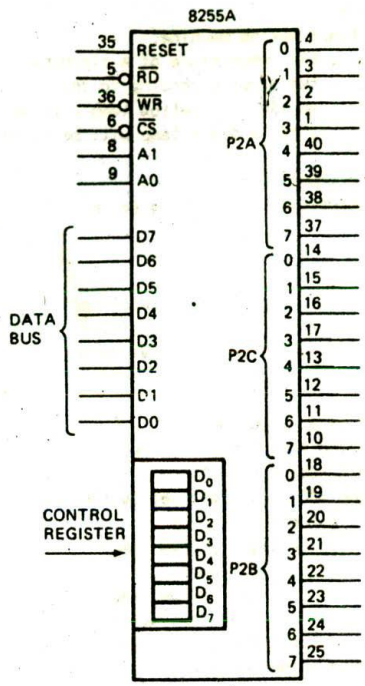


FIGURE 4-13 Block diagram of SDK-86 board's 8255A port.

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 Figure 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	FFFAH	PORT 1B	FFFBH
PORT 2C	FFFCH	PORT 1C	FFFDH
CONTROL2	FFFEH	CONTROL1	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,0FFFEH.

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 4-14*a*, p. 80, shows the list file of the 8086 assembly language implementation of the algorithm in Figure 4-12*a*. 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 FFFAH. A 1 sent to a bit position of port FFFAH turns on the lamp connected to that line.

The next two instructions in the example program read the temperature in from an analog-to-digital converter connected to input port FFF8H.

After we read the data in from the port, we compare it with our set-point value of 30°C. If the input value is below 30°C, then we jump to the instructions which turn on the yellow lamp. If the temperature is above or equal to 30°C, we jump to the instructions which turn on the green lamp. Note that we have implemented this algorithm in such a way that the JB instruction will always be able to reach the label YELLOW.

To actually turn on a lamp, we load a 1 in the appropriate bit of the AL register with a MOV instruction and send the byte to the lamp control port, FFFAH. The instruction sequence MOV AL,01H—OUT DX,AL, for example, will light the yellow lamp by sending a 1 to bit 0 of port FFFAH.

The instruction sequence MOV AL,02H—OUT DX,AL will light the green lamp by sending a 1 to bit 1 of port FFFAH. Note that control words are sent to the control register address in an 8255 and data words are read from or written to the individual port addresses. Here's another way to implement this program in assembly language.

Figure 4-14*b* shows another equally valid assembly language program segment to solve our problem. This one uses a Jump if Above or Equal instruction, JAE, at the decision point and switches the order of the actions. This program more closely follows the second algorithm statement in Figure 4-12*b*. Perhaps you can see from these examples why two programmers may write very different programs to solve even very simple programming problems.

Multiple IF-THEN-ELSE Assembly Language Programs

In the preceding section we showed how to implement and use the IF-THEN-ELSE structure, which chooses between two alternative courses of action. In

```
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, 0FFFEH      ; 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 FFF8                         MOV DX, 0FFF8H      ; 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         ; Load code to light yellow lamp
24 0013  BA FFFA                         MOV DX, 0FFFAH      ; 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         ; Load code to light green lamp
28 001C  BA FFFA                         MOV DX, 0FFFAH      ; Point DX at output port
29 001F  EE                              OUT DX, AL          ; Send code to light green lamp
30 0020  BA FFFC                 EXIT:   MOV DX, 0FFFCH      ; Next mainline instruction
31 0023  EC                              IN  AL, DX          ; Read ph sensor
32 0024                          CODE    ENDS
33                                       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         ; Load code to light green lamp
24 0013  BA FFFA                         MOV DX, 0FFFAH      ; 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         ; Load code to light yellow lamp
28 001C  BA FFFA                         MOV DX, 0FFFAH      ; Point DX at output port
29 001F  EE                              OUT DX, AL          ; Send code to light yellow lamp
30 0020  BA FFFC                 EXIT:   MOV DX, 0FFFCH      ; Next mainline instruction
31 0023  EC                              IN  AL, DX          ; Read ph sensor
32 0024                          CODE    ENDS
33                                       END
```

(*b*)

FIGURE 4-14 List file for printed-circuit-board-making machine program.
(*a*) Below 30° version. (*b*) Program section for above 30° version.

many situations we want a computer to choose one of several alternative actions based on the value of some variable read in or on a command code entered by a user. To choose one alternative from several, we can *nest* IF-THEN-ELSE structures. The result has the form

```
IF condition THEN
    action
ELSE IF condition THEN
        action
    ELSE
        action
```

It is important to note that in this structure the last ELSE is part of the IF-THEN just before it. Figure 3-3*d* showed a flowchart and pseudocode for a "soup cook" example using this structure, but the soup cook example is too messy to implement here. Therefore, while the printed-circuit-board-making machine from the last section is still fresh in your mind, we will expand that example to show you how a multiple IF-THEN-ELSE is implemented.

Suppose that we want to have three lamps on our printed-circuit-board-making machine. We want a yellow lamp to indicate that the temperature is below 30°C, a green lamp to indicate that the temperature is above or equal to 30°C but below 40°C, and a red lamp to indicate that the temperature is at or above 40°C. Figure 4-15 shows three ways to indicate what we want to do here. The first way, in Figure 4-15*a*, simply indicates the desired action next to each temperature range. You may find this form very useful in visualizing problems where the alternatives are based on the range of a variable. Don't miss the ASCII-to-hexadecimal problem at the end of the chapter for some practice with this.

Once you get a problem such as this defined in list form, you can easily convert it to a flowchart or pseudocode. When writing the flowchart or the pseudocode, it is best to start at one end of the overall range and work your way to the other. For example, in the flowchart in Figure 4-15*c*, start by checking whether the temperature is below 30°. If the temperature is not below 30°, then it must be above or equal to 30°, and you do not have to do another test to determine this. You then check whether the temperature is below 40°. If the temperature is above or equal to 30°, but below 40°, then you know that the temperature is in the green lamp range. If the temperature is not below 40°, then you know that the temperature must be above or equal to 40°. In other words, two carefully chosen tests will direct execution to one of the three alternatives.

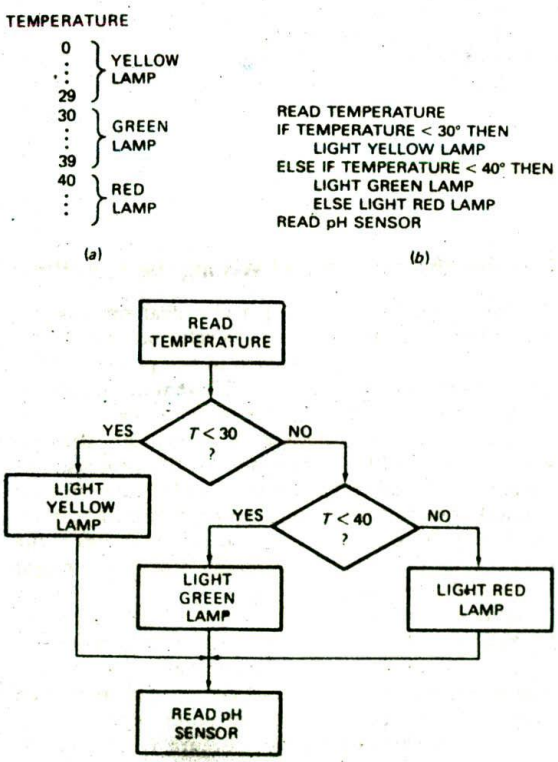


FIGURE 4-15 Algorithm for three-lamp printed-circuit-board-making machine. (*a*) Condition list. (*b*) Pseudocode. (*c*) Flowchart.

Figure 4-16, p. 82, shows how we can write a program for this algorithm in 8086 assembly language. In the program, we first initialize port FFFAH as an output port. We then read in the temperature from an A/D converter connected to port FFF8H. We compare the temperature read in with the first set-point value, 30°. If the temperature is below 30°, the Jump if Below instruction, JB, will cause a jump to the label YELLOW. If the jump is not taken, we know the temperature is above or equal to 30°, so we go on to 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.

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 Figure 3-3*d*, 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 Figure 4-16. However, expressing

```
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, 0FFFEH      ; 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, 0FFF8H      ; Point DX at input port
18 0009  EC                             IN  AL, DX          ; Read temp from sensor on input port
19 000A  BA FFFA                        MOV DX, 0FFFAH      ; Point DX at output port
20 000D  3C 1E                          CMP AL, 30          ; Compare temp with 30°C
21 000F  72 0A                          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, 0FFFCH      ; Next mainline instruction
33 0027  EC                             IN  AL, DX          ; Read ph sensor
34 0028                         CODE    ENDS
35                                      END
```

FIGURE 4-16 List file for three-lamp printed-circuit-board-making machine program.

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 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°. Inciden-

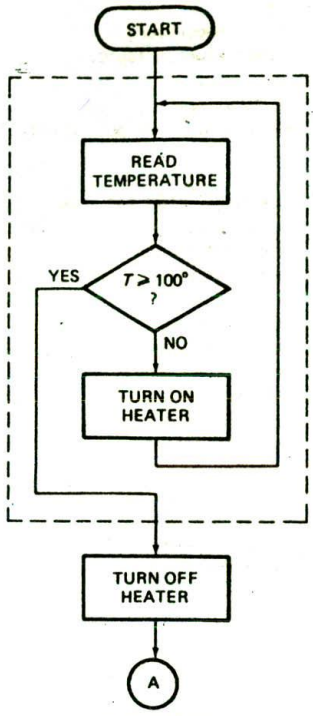


FIGURE 4-17 Flowchart and pseudocode for heater control program.

tally, 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 4-18*a*, p. 84, 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 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 FFFAH 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 Figure 4-17*a* and by the indentation in the pseudocode in Figure 4-17*b*.

Solving a Potential Problem of Conditional Jump Instructions

In the example program in Figure 4-18*a*, 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 Figure 4-18*a* 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 Figure 4-11. To refresh your memory, Figure 4-18*b* 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.

```
1                                       ; 8086 PROGRAM   F4-18A.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, 0FFFEH    ; Point DX to port control register
11 0003  B0 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, 0FFF8H    ; 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, 0FFFAH    ; 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, 0FFFAH    ; Point DX to output port
24 001B  EE                              OUT  DX, AL        ; Turn heater off
25 001C                          CODE    ENDS
26                                       END
```

(*a*)

```
14 0006  BA FFF8                 TEMP_IN:   MOV  DX, 0FFF8H    ; 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, 0FFFAH    ; 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, 0FFFAH    ; Point DX at output port
25 001E  EE                              OUT  DX, AL        ; Turn heater off
26 001F                          CODE    ENDS
27                                       END
```

(*b*)

FIGURE 4-18 List file for heater control program. (*a*) First approach. (*b*) Improved version of WHILE-DO section of program.

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.

Defining the Problem and Writing the Algorithm

Many systems that interface with a microcomputer output data on parallel-signal lines and then output a separate signal to indicate that valid data is on the parallel lines. The data-ready signal is often called a

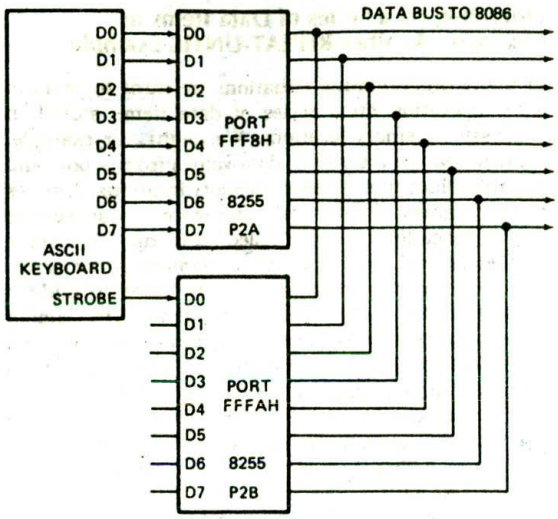


FIGURE 4-19 ASCII-encoded keyboard with strobe connected to microcomputer port.

strobe. An example of a strobed data system such as this is an ASCII-encoded computer-type keyboard. Figure 4-19 shows how the parallel data lines and the strobe line from such a keyboard are connected to ports of a microcomputer. When a key is pressed on the keyboard, circuitry in the keyboard detects which key is pressed and sends the ASCII code for that key out on the eight data lines connected to port FFF8H. After the data has had time to settle on these lines, the circuitry in the keyboard sends out a key-pressed strobe, which lets you know that the data on the eight lines is valid. A strobe can be an active high signal or an active low signal. For the example here, assume that the strobe signal goes high when a valid ASCII code is on the parallel data lines. As you can see in Figure 4-19, we have connected this strobe line to the least significant bit of port FFFAH so that we can input the strobe signal.

If we want to read the data from this keyboard, we can't do it at just any time. We must wait for the strobe to go high so that we know that the data we read will be valid. Basically, what we have to do is look at the strobe signal and test it over and over until it goes high. Figure 4-20*a*, p. 86, shows how we can represent this operation with a flowchart, and Figure 4-20*b* shows the pseudocode. We want to repeat the read-strobe-and-test loop until the strobe is found to be high. Then we want to exit the loop and read in the ASCII code byte. The basic REPEAT-UNTIL structure is shown by the indentation in the pseudocode. Note that the read ASCII data action is not part of this structure and is therefore not indented.

Implementing the Algorithm with Assembly Language

Figure 4-20c shows the 8086 assembly language to implement this algorithm. To read in the key-pressed strobe signal, we first load the address of the port to which it is connected into the DX register. Then we use the variable-port input instruction, IN AL,DX, to read the strobe data to AL. This input instruction copies a byte of data from port FFFAH to the AL register. We care about only the least significant bit of the byte read in from the port, however, because that is where the strobe is connected. To determine whether the strobe is present, we need to check just this bit and determine whether it is a 1. Here are three different ways you can do this.

The first way, shown in Figure 4-20c, is to AND the byte in AL with the immediate number 01H. Remember that a bit ANDed with a 0 becomes a 0 (is masked). A bit ANDed with a 1 is not changed. If the least significant bit is a 0, then the result of the ANDing will be all 0's. The zero flag ZF will be set to a 1 to indicate this. If the least significant bit is a 1, the zero flag will not be set to a 1 because the result of the ANDing will still have a 1 in the least significant bit. The Jump if Zero instruction, JZ, will check the state of the zero flag; if it finds the zero flag set, it will jump to the label LOOK_AGAIN. If the JZ instruction finds the zero flag not set (indicating that the LSB was a 1), it passes execution on to the instructions which read in the ASCII data.

Another way to check the least significant bit of the strobe word is with the TEST instruction instead of the AND instruction. The 8086 TEST instruction has the format TEST destination,source. The TEST instruction ANDs the contents of the specified source with the contents of the specified destination and sets flags according to the result. However, the TEST instruction does not change the contents of either the source or the destination. The AND instruction, remember, puts the result of the ANDing in the specified destination. The TEST instruction is useful if you want to set flags without changing the operands. In the example program in Figure 4-20c, the AND AL,01H instruction could be replaced with the TEST AL,01H instruction.

Still another way to check the least significant bit of the strobe byte is with a Rotate instruction. If you rotate the least significant bit into the carry flag, you can use a Jump if Carry or Jump if Not Carry instruction to control the loop. For this example program, you could use either the ROR instruction or the RCR instruction. To verify this, take a look at the discussions of these instructions in Chapter 6. Assuming that you use the ROR instruction, the check and jump instruction sequence would look like this:

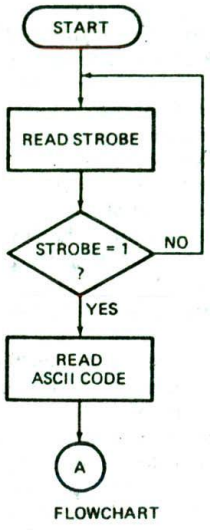
```
LOOK_AGAIN:IN AL, DX
           ROR AL, 1          ; Rotate LSB into carry
           JNC LOOK_AGAIN; If LSB = 0, keep looking
```

For your programs you can use the way of checking a bit that seems easiest in a particular situation.

To read the ASCII data, we first have to load the port address, FFF8H, into the DX register. We then use the variable-port input instruction IN AL,DX to copy the ASCII data byte from the port to the AL register.

The main purpose of the preceding section was to show you how you can use a Conditional Jump instruction to make the 8086 REPEAT a series of actions UNTIL

the flags indicate that some condition is present. The following section shows another example of implementing the REPEAT-UNTIL structure. This example also shows you how a register-based addressing mode is used to access data in memory.



FLOWCHART

(*a*)

```
REPEAT
    READ KEYPRESSED STROBE
UNTIL STROBE = 1
READ ASCII CODE FOR KEY PRESSED
```

PSEUDOCODE

(*b*)

Operating on a Series of Data Items in Memory—Another REPEAT-UNTIL Example

In many programming situations we want to perform some operation on a series of data items stored in successive memory locations. We might, for example, want to read in a series of data values from a port and put the values in successive memory locations. A series of data values of the *same type* stored in successive memory locations is often called an *array*. Each value in the array is referred to as an *element* of the array. For our example program here, we want to add an inflation factor of 03H to each price in an eight-element array of prices. Each price is stored in a byte location as packed BCD (two BCD digits per byte). The prices then are in the range of 1 cent to 99 cents. Figure 4-21*a* shows a flowchart and Figure 4-21*b* shows a pseudocode algorithm for the operations that we want to perform. Follow through whichever form you feel more comfortable with.

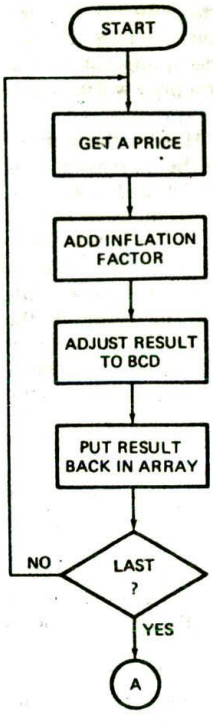
We read one of the BCD prices from memory, add the inflation factor to it, and adjust the result to keep it in BCD format. The new value is then copied back to the array, replacing the old value. After that, a check is made to see whether all the prices have been operated on. If they haven't, then we loop back and operate on the next price. The two questions that may occur to you at this point are, "How are we going to indicate in the program which price we want to operate on, and how are we going to know when we have operated on all of the prices?" To indicate which price we are operating on at a particular time, we use a register as a *pointer*. To keep track of how many prices we have operated on, we use another register as a *counter*. The example program in Figure 4-21c shows one way in which the algorithm for this problem can be implemented in assembly language.

The example program in Figure 4-21c uses several assembler directives. Let's review the function of these

```
 1                                ;  8086 PROGRAM F4-20C.ASM
 2                          ;ABSTRACT  : Program to read ASCII code after a strobe signal
 3                                ;  is sent from a keyboard
 4                          ;REGISTERS : Uses CS, DX, AL
 5                          ;PORTS     : Uses FFFAH - strobe signal input on LSB
 6                                ;       FFF8H - ASCII data input port
 7
 8 0000                     CODE       SEGMENT
 9                                     ASSUME CS:CODE
10 0000  BA FFFA                       MOV DX, 0FFFAH     ; Point DX at strobe port
11 0003  EC                LOOK_AGAIN: IN  AL, DX         ; Read keyboard strobe
12 0004  24 01                         AND AL, 01         ; Mask extra bits and set flags
13 0006  74 FB                         JZ  LOOK_AGAIN     ; If strobe is low then keep looking
14 0008  BA FFF8                       MOV DX, 0FFF8H     ; else point DX at data port
15 000B  EC                            IN  AL, DX         ; Read in ASCII code
16 000C                    CODE       ENDS
17                                     END
```

(*c*)

FIGURE 4-20 Flowchart, pseudocode, and assembly language for reading ASCII code when a strobe is present. (*a*) Flowchart. (*b*) Pseudocode. (*c*) List file of program.



FLOWCHART

(a)

```
REPEAT
    GET A PRICE FROM ARRAY
    ADD INFLATION FACTOR
    ADJUST RESULT TO CORRECT BCD
    PUT RESULT BACK IN ARRAY
UNTIL ALL PRICES ARE INFLATED
```

PSEUDOCODE

(b)

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.

The statement COST DB 20H,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, p. 88, 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

```
1                                       ; 8086 PROGRAM  : F4-21C.ASM
2                                ;ABSTRACT  : Program adds an inflation factor to a series of prices
3                                       ; in memory. It copies the new price over the old price.
4                                ;REGISTERS : Uses DS, CS, AX, BX, CX
5                                ;PORTS     : None used
6
7 0000                           ARRAYS   SEGMENT
8 0000  20 28 15 26 19 27 16 +           COST     DB     20H, 28H, 15H, 26H, 19H, 27H, 16H, 29H
9       29
10 0008 36 55 27 42 38 41 29 +           PRICES   DB     36H, 55H, 27H, 42H, 38H, 41H, 29H, 39H
11      39
12 0010                          ARRAYS   ENDS
13
14 0000                          CODE     SEGMENT
15                                       ASSUME CS:CODE, DS:ARRAYS
16 0000  B8 0000s                START:   MOV  AX, ARRAYS   ; Initialize data segment
17 0003  8E D8                           MOV  DS, AX        ; register
18 0005  8D 1E 0008r                     LEA  BX, PRICES    ; Initialize pointer
19 0009  B9 0008                         MOV  CX, 0008H     ; Initialize counter
20 000C  8A 07                   DO_NEXT: MOV  AL, [BX]      ; Copy a price to AL
21 000E  04 03                           ADD  AL, 03H       ; Add inflation factor
22 0010  27                              DAA                ; Make sure result is BCD
23 0011  88 07                           MOV  [BX], AL      ; Copy result back to memory
24 0013  43                              INC  BX            ; Point to next price
25 0014  49                              DEC  CX            ; Decrement counter
26 0015  75 F5                           JNZ  DO_NEXT       ; If not last, go get next
27 0017                          CODE     ENDS
28                                       END  START
```

(c)

FIGURE 4-21 Adding a constant to a series of values in memory. (a) Flowchart. (b) Pseudocode. (c) List file of program.

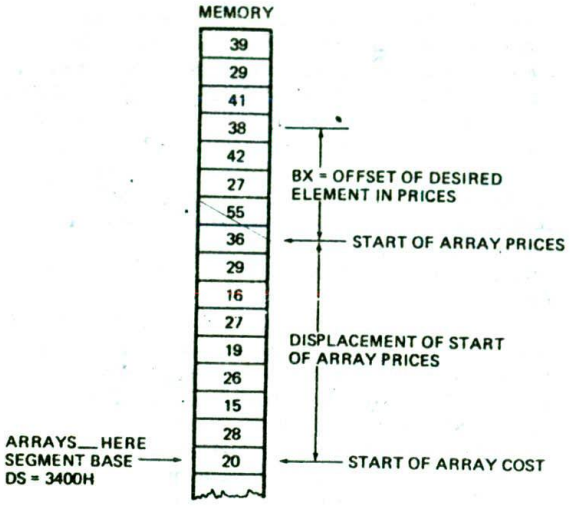


FIGURE 4-22 Data arrangement in memory for "inflate prices" program.

described for the example program in Figure 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 Figure 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,MULTIPLICAND 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 result of the addition will be left in AL. We want the prices in the array to be in BCD form, so we have to make sure the result is adjusted to be a legal BCD number. For example, if we add 03 to 29, the result in AL will be 2C. Most people would not understand this as a price, so we have to adjust the result to the desired BCD number. The Decimal Adjust after Addition instruction DAA will automatically make this adjustment for us. DAA will adjust the 2CH by adding 6 to the lower nibble and the carry produced to the upper nibble. The result of this in AL will be 32H, which is the result we want from adding 03 to 29. Note that the DAA instruction works only on the AL register. For further examples of DAA operation, consult the DAA instruction description in Chapter 6.

The INC BX instruction adds 1 to the number in BX. BX now contains the effective address or offset of the next price in the array. We like to say that BX now points to the next element in the array.

The DEC CX instruction decrements the count we set up in the CX register by 1. If CX contains 0 after this decrement, the zero flag will be set to a 1. The JNZ DO_NEXT checks the zero flag. If it finds the zero flag set, it just passes execution out of the structure to the next mainline instruction. If it finds the zero flag not set, the JNZ instruction will cause a jump to the label DO_NEXT. In other words, the 8086 will repeat the sequence of instructions between the label and the JNZ instruction until CX is counted down to zero. Each time through the loop, BX will be incremented to point to the next price in the array.

Still Another REPEAT-UNTIL Example

Using a pointer to access data items in memory is a powerful technique that you will want to use in many of your programs, so Figure 4-23 shows still another example. In this example, we want to add a profit of 15 cents to each element of an array called COST and put the result in the corresponding element of an array called PRICES. The algorithm for this example is

```
REPEAT
        Get an item from cost array
        Add profit factor
        Adjust result to correct BCD
        Put result into price array
UNTIL all prices are calculated
```

The assembly language implementation of this algorithm is very similar to that for the last example, except for the way we use the pointers. In this example we need to point to the same element in two different arrays. To do this, we use the BX register to keep track of which element we are currently accessing in the arrays. At the

```
1                                       ; 8086 PROGRAM F4-23.ASM
2                                ;ABSTRACT  : Program adds a profit factor to each element in a
3                                       ; COST array and puts the result in an PRICES array.
4                                ;REGISTERS : Uses DS, CS, AX, BX, CX
5                                ;PORTS     : None used
6
7       = 0015                   PROFIT     EQU      15H          ; profit = 15 cents
8 0000                           ARRAYS     SEGMENT
9 0000  20 28 15 26 19 27 16 +             COST     DB  20H, 28H, 15H, 26H, 19H, 27H, 16H, 29H
10      29
11 0008 08*(00)                            PRICES   DB  8 DUP(0)
12 0010                          ARRAYS     ENDS
13
14 0000                          CODE       SEGMENT .
15                                          ASSUME  CS:CODE, DS:ARRAYS
16 0000 B8 0000s                 START:     MOV  AX, ARRAYS      ; Initialize data segment
17 0003 8E D8                               MOV  DS, AX          ; register
18 0005 B9 0008                             MOV  CX, 0008H       ; Initialize counter
19 0008 BB 0000                             MOV  BX, 0000H       ; Initialize pointer
20 000B 8A 87 0000r              DO_NEXT:   MOV  AL, COST[BX]    ; Get element [BX] from COST
21 000F 04 15                               ADD  AL, PROFIT      ; Add the profit to value
22 0011 27                                  DAA                  ; Decimal adjust result
23 0012 88 87 0008r                         MOV  PRICES[BX], AL  ; Store result in PRICES at [BX]
24 0016 43                                  INC  BX              ; Point to next element in arrays
25 0017 49                                  DEC  CX              ; Decrement the counter
26 0018 75 F1                               JNZ  DO_NEXT         ; If not last element, do again
27 001A                          CODE       ENDS
28                                          END  START
```

FIGURE 4-23 List file of "price-calculating" program.

start of the program, then, we initialize BX as a pointer to the first element of each array with MOV BX,0000H. The instruction MOV AL,COST[BX] then will copy the first value from the array COST into AL. The effective address for this instruction will be produced by adding the displacement represented by the name COST to the contents of BX.

After the Addition and Decimal Adjust instructions, the instruction MOV PRICES[BX],AL copies the result of the addition to the first element of PRICES. The 8086 computes the effective address for this instruction by adding the contents of BX to the displacement represented by the name PRICES.

The BX register is incremented, so that if CX has not been decremented to zero, COST[BX] and PRICES[BX] will each access the next element in the array when execution goes through the DO_NEXT loop again. A programmer familiar with higher-level languages would probably say that BX is being used as an array index in this example.

Another Look at 8086 Addressing Modes

The preceding examples showed you how a register can be used as a pointer or index to access a sequence of data items in memory. While these examples are fresh in your mind, we want to show you more about the 8086 addressing modes we introduced you to in Chapter 3.

Figure 4-24, p. 90, summarizes all the ways you can tell the 8086 to calculate an effective address and a physical address for accessing data in memory. In all cases, the physical address is generated by adding an effective address to one of the segment bases, CS, SS, DS, or ES. The effective address can be a direct displacement specified directly in the instruction, as, for example, MOV AX,MULTIPLIER. The effective address or offset can be specified to be in a register, as in the instruction MOV AL,[BX]. Also, the effective address can be specified to be the contents of a register plus a displacement included in the instruction. The instruction MOV AX,PRICES[BX] is an example of this addressing mode. For this example, PRICES represents the displacement of the start of the array from the segment base, and BX represents the number of the element in the array that we want to access. The effective address of the desired element, then, is the sum of these two.

For working with more complex data structures such as the array of records shown in Figure 4-25, p. 90, you can tell the 8086 to compute an effective address by adding the contents of BX or BP plus the contents of SI or DI plus an 8-bit or a 16-bit displacement contained in the instruction. You can, for example, use an instruction such as MOV AL, PATIENTS[BX][SI] to access the balance due field in the array of medical records shown in Figure 4-25. The name PATIENTS in this instruction represents the displacement of the array PATIENTS from the start of the data segment. The BX register holds the offset of the start of the desired record in the array. The SI register holds the offset of the start of the desired field in the record. To access the next record in the array, you simply add a number equal to the length of the record to the BX register. To access another field in a record, you just change the value in the SI register.

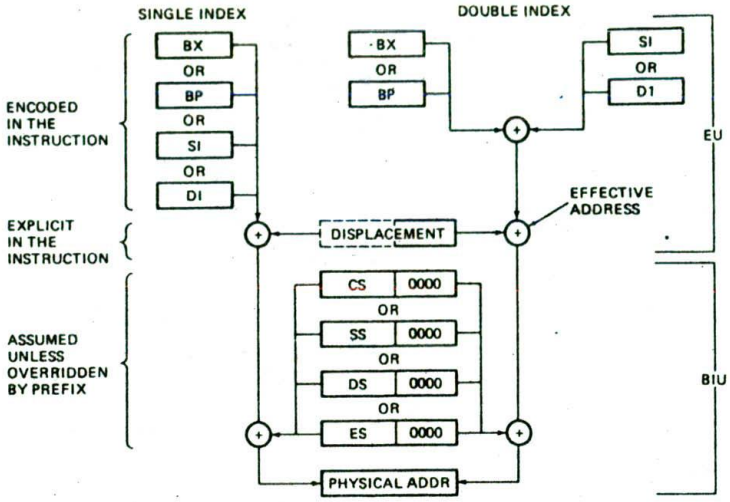


FIGURE 4-24 Summary of 8086 addressing modes.

When BX, SI, or DI is used to contain all or part of the effective address, the physical address will be produced by adding the effective address to the data segment base in DS. When BP is used to contain all or part of the effective address, the physical address will be produced by adding the effective address to the stack segment base in SS. For any of these four, you can use a segment override prefix to tell the 8086 to add the effective address to some other segment base. The instruction MOV AL,CS:[BX] tells the 8086 to produce a physical memory address by adding the offset in BX to the code segment base instead of adding it to the data segment base. An exception to this is that with a special group of instructions called *string instructions*, an offset in DI will always be added to the extra segment base in ES to produce the physical address.

```
SEGMENT BASE
    .       Name PATIENTS represents displacement of
    .       start of array of records from segment base
    .
PATIENTS    ; array of patient records start here

                          RECORD 1
                          TV N. BEER
                          1324 Down Street
                          PORTLAND, OR 97219
                          2/15/45
                          247 lb
                          $327.56

BX holds offset of -------> RECORD 2
desired record  in array    IM A. RUNNER
                            17197 Hatton Road
                            Oregon City, OR 97045
                            6/30/41
SI holds offset of -------> 145 lb
desired field in record     $0.00

                            RECORD 3
```

FIGURE 4-25 Use of double indexed addressing mode.

The 8086 LOOP Instructions

In the second REPEAT-UNTIL example, we showed you how to make a program repeat a sequence of instructions a specific number of times. To do this, you load the desired number of repeats in a register or memory location. Each time the sequence of instructions executes, the count value in the register or memory location is decremented by 1. When the count is decremented to zero, the zero flag will be set. You use a Conditional Jump instruction to check this flag and to decide whether to repeat the instruction sequence in the loop again.

The need to perform a sequence of actions a specified number of times in a program is so common that some programming languages use a specific structure to express it. This structure, derived from the basic WHILE-DO, is called the FOR-DO loop. It has the form

```
FOR count = 1 to count = n DO
    action
    action
```

where *n* is the number of times we want to do the sequence of actions.

The common need to repeat a sequence of actions a specified number of times led the designers of the 8086 to give it a group of instructions which make this easier for you. These instructions are the LOOP instructions.

INSTRUCTION OPERATION

The LOOP instructions are basically Conditional Jump instructions which have the format LOOP label. LOOP instructions, however, combine two operations in each instruction. The first operation is to decrement the CX

LOOP	Loop until CX = 0
LOOPE/LOOPZ	Loop if zero flag set and CX ≠ 0
LOOPNE/LOOPNZ	Loop if zero flag not set and CX ≠ 0
JCXZ	Jump if CX = 0

FIGURE 4-26 8086 LOOP instructions.

register by 1. The second operation is to check the CX register and, in some cases, also the zero flag to decide whether to do a jump to the specified label. The simple LOOP label instruction then can be used in place of the DEC CX—JNZ label instruction sequence we used in Figure 4-21c.

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 Figure 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 Figure 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 Figure 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 ≠ 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 ≠ 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 MHz) or 0.2 μs. An instruction which takes 4 clock cycles, then, will take 4 clock cycles × 0.2 μs/clock cycle or 0.8 μs 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 4-27*a* 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 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*)

FIGURE 4-27 Delay loop program and calculations. (*a*) Program. (*b*) Calculations.

The LOOP KILL_TIME instruction will decrement CX and, if CX is not down to zero yet, do a jump to the label KILL_TIME. The program then will cause the 8086 to execute the two NOP instructions and the LOOP instruction over and over until CX is counted down to zero. The number in CX will determine how long this takes. Here's how you determine the value to put in CX for a given amount of delay.

First you calculate the number of clock cycles needed to produce the desired delay. If you are running your 8086 with a 5-MHz clock, then the time for each clock cycle is 1/(5 MHz) or 0.2 μs. Now, suppose that you want to create a delay of 1 ms or 1000 μs with a delay loop. If you divide the 1000 μs desired by the 0.2 μs per clock cycle, you get the number of clock cycles required to produce the desired delay. For this example you need a total of 1000/0.2 or 5000 processor clock cycles to produce the desired delay. We will call this number C_T for future reference.

The next step is to write the number of clock cycles required for each instruction next to that instruction, as shown in Figure 4-27*a*. Then you look at the program to determine which instructions get executed only once. The number of clock cycles for the instructions which execute only once will only contribute to the total once. Instructions which only enter the calculation once are often called *overhead*. We will represent the number of cycles of overhead with the symbol C_o. In Figure 4-27*a*, the only instruction which executes just once is MOV CX,N, which takes 4 clock cycles. For this example, then, $C_o = 4$.

Next you determine how many clock cycles are required for the loop. The two NOPs in the loop require a total of 6 clock cycles. The LOOP instruction requires 17 clock cycles if it does the jump back to KILL_TIME, but it requires only 5 clock cycles when it exits the loop. The jump takes longer because the instruction byte queue has to be reloaded starting from the new address. For all but the very last time through the loop, it will require 17 clock cycles for the LOOP instruction. Therefore, you can use 17 as the number of cycles for the LOOP instruction and compensate later for the fact that the last time it takes 12 cycles less. For the example program, the number of cycles per loop $C_L = 6 + 17$ or 23.

The total number of clock cycles delayed by the loop is equal to the number of times the loop executes multiplied by the time per loop. To be somewhat more accurate, you can subtract the 12 cycles that were not used when the last LOOP instruction executed. The total number of clock cycles required for the example program to execute is

$$C_T = C_o + N(C_L) - 12$$

To find the value for N for a desired amount of delay, put in the required C_T, 5000 for this example, and solve the result for N. Figure 4-27*b* shows how this is done. The resultant value for N is 218 decimal or 0DAH. This is the number of times you want the loop to repeat, so this is the value of N that you will load into CX before entering the loop.

With the simple relationship shown in Figure 4-27*b*, you can determine the value of N to put in a delay loop you write, or you can determine the time a delay loop written by someone else will take to execute.

If you can't get a long enough delay by counting down a single register or memory location, you can nest delay loops. An example of this nesting is

```
                          ; number of states
        MOV BX, COUNT1    ; 4
CNTDN1: MOV CX, COUNT2    ; 4(COUNT1)
CNTDN2: LOOP CNTDN2       ; ((17 × COUNT2) − 12)COUNT1
        DEX BX            ; 2(COUNT1)
        JNZ CNTDN1        ; 16(COUNT1) − 12
```

The principle here is to load CX with COUNT2 and count CX down COUNT1 times. To determine the number of states that this program section will take to execute, observe that the LOOP instruction will execute COUNT2 times for each time CX is loaded with COUNT1. The total number of states, then, is COUNT1 times the number of states for the last four instructions plus 4, for the MOV BX,COUNT1 instruction. The best way to approach getting values for the two unknowns, COUNT1 and COUNT2, is to choose a value such as FFFFH for COUNT2 and then solve for the value of COUNT1. A couple of tries should get reasonable values for both COUNT1 and COUNT2.

Notes about Using Delay Loops for Timing

There are several additional factors you have to take into account when determining the time that a sequence of instructions will require to execute.

1. The BIU and the EU are asynchronous, so for some instruction sequences an extra clock cycle may be required. For a given sequence of instructions the added cycles are always the same, but obviously these cycles are not included in the numbers given in Appendix B.

2. The number of clock cycles required to read a word from memory or write a word to memory depends on whether the first byte of the word is at an even address or at an odd address. The 8086 will require 4 additional clock cycles to read or write a word located on an odd address.

3. The number of clock cycles required to read a byte from memory or write a byte to memory depends on the addressing mode used to access that byte. A table at the start of Appendix B shows the number of clock cycles that must be added for each addressing mode. According to Appendix B, the basic mem 8 to reg 8 instruction requires 8 + EA clock cycles. The [BX] addressing mode requires 5 clock cycles, so the instruction MOV AL,[BX] requires 8 + 5 or 13 clock cycles to execute.

4. If a given microcomputer system is designed to insert WAIT states during each memory access, this will increase the number of clock cycles required for each memory access. In Chapter 7 we discuss the use of WAIT states.

In summary, the calculations we showed you how to do in the preceding section give you the approximate time it will take a sequence of instructions to execute. If you really need to know the precise time a sequence of instructions requires to execute, the only way to determine it is to use a logic analyzer or emulator to measure the actual number of clock cycles.

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.

Defining a problem

Setting up a data structure

Making an initialization checklist

Masking using the AND instruction

Packed and unpacked BCD numbers

Debugging—breakpoints, trace, single step

Conditional flags: CF, PF, AF, ZF, SF, OF

Unconditional JMP instructions
- Direct and indirect near (intrasegment) jumps
- Direct and indirect far (intersegment) jumps
- Short jumps

Conditional jumps

Fixed- and variable-port input/output instructions

Based and indexed addressing modes

Loop instruction

Processor clock cycles

Delay loops

REVIEW QUESTIONS AND PROBLEMS

1. Describe the operation and results of each of the following instructions, given the register contents shown in Figure 4-28 (below question 3). Include in your answer the physical address or register that each instruction will get its operands from and the physical address or register that each instruction will put the result in. Use the instruction descriptions in Chapter 6 to help you. Assume that the instructions below are independent, not sequential, unless listed together under a letter.
   a. ROL AX,CL
   b. IN AL,DXP
   c. MOV CX,[BX]
   d. ADD AX,[BX][SI]
   e. JMP 023AH
   f. JMP BX

2. Construct the binary codes for the instructions of Questions 1*a* through 1*f*.

3. Predict the state of the six 8086 conditional flags after each of the following instructions or group of instructions executes. Use the register contents shown in Figure 4-28. Assume that all flags are reset before the instructions execute. Use the detailed instruction descriptions in Chapter 6 to help you.
   a. MOV AL,AH
   b. ADD BL,CL
   c. ADD CL,DH
   d. OR CX,BX

```
CS = 2000        AX = A407
DS = 3000        BX = 2483
SS = 4000        CX = 0002
ES = 3000        DX = FFFA
SP = FFFF
BP = 0009
SI = 4200
DI = 4300
```

FIGURE 4-28 Figure for Chapter 4 problems.

4. See if you can find any errors in the following instructions or groups of instructions.
   a.
```
CNTDOWN: MOV BL, 72H
         DEC BL
         JNZ CNTDOWN
```
   b. ADD CX,AL
   c. JMP BL
   d. JNZ [BX]

5. a. Write an algorithm for a program which adds a byte number from one memory location to a byte from the next memory location, puts the sum in a third memory location, and saves the state of the carry flag in the least significant bit of a fourth memory location. Mask the upper 7 bits of the memory location where the carry is stored.
   b. Write an 8086 assembly language program for this algorithm. *Hints:* Set up data declarations similar to those in Figure 3-14. Use a Rotate instruction to get the carry flag state into the LSB of a register or memory location.
   c. What additional instructions would you have to add to this program so that it correctly adds 2 BCD bytes?

For each of the following programming problems, draw a flowchart or write the pseudocode for an algorithm to solve the problem. Then write an 8086 assembly language program to implement the algorithm. If you have an 8086 system available, enter and assemble your source program, then load the object code for the program into memory so that you can run and test it. If the program does not work correctly, use the single-step or breakpoint approaches described earlier in this chapter to help you debug it.

6. Convert a packed BCD byte to two ASCII characters for the two BCD digits in the byte. For example, given a BCD byte containing 57H (01010111 binary), produce the two ASCII codes 35H and 37H.

7. In order to avoid hand keying programs into an SDK-86 board, we wrote a program to send machine code programs from an IBM PC to an SDK-86 board through a serial link. As part of this program, we had to convert each byte of the machine code program to ASCII codes for the two nibbles in the byte. In other words, a byte of 7AH has to be sent as 37H, the ASCII code for 7, and 41H, the ASCII code for A. Once you separate the nibbles of the byte, this conversion is a simple IF-THEN-ELSE situation. Write an algorithm and assembly language program section which does the needed conversion.

8. A common problem when reading a series of ASCII characters from a keyboard is the need to filter out those codes which represent the hex digits 0 to 9 and A to F, and convert these ASCII codes to the hex digits they represent. For example, if we read in 34H, the ASCII code for 4, we want to mask the upper 4 bits to leave 04, the 8-bit hex code for 4. If we read in 42H, the ASCII code for B, we want to add 09 and mask the upper 4 bits to leave 0B, the 8-bit code for hex B. If we read in an ASCII code that is not in the range of 30H to 39H or 41H to 46H, then we want to load an error code of FFH instead of the hex value of the entered character. Figure 4-29 shows the desired action next to each range of ASCII values. Write an algorithm and an assembly language program which implements these actions. *Hint:* A nested IF-THEN-ELSE structure might be useful.

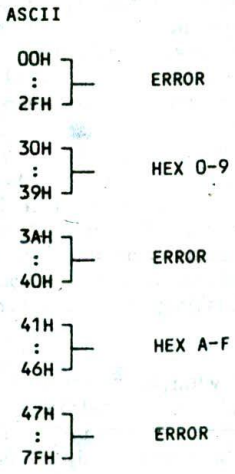


FIGURE 4-29 ASCII chart for Problem 8.

9. Compute the average of 4 bytes stored in an array in memory.

10. Compute the average of any number of bytes in an array in memory. The number of bytes to be added is in the first byte of the array.

11. Add a 5-byte number in one array to a 5-byte number in another array. Put the sum in another array. Put the state of the carry flag in byte 6 of the array that contains the sum. The first value in each array is the least significant byte of that number. *Hint:* See Figure 4-23.

12. An 8086-based process control system outputs a measured Fahrenheit temperature to a display on its front panel. You need to write a short program which converts the Fahrenheit temperature to Celsius so that the system can be sold in Europe. The relationship between Fahrenheit and Celsius is $C = (F - 32)5/9$. The Fahrenheit temperature will always be in the range of 50° to 250°. Round the Celsius value to the nearest degree.

13. An ASCII keyboard outputs parallel ASCII + parity to port FFF8H of an SDK-86 board. The keyboard also outputs a strobe to the least significant bit (D0) of port FFFAH. (See Figure 4-19.) When you press a key, the keyboard outputs the ASCII code for the pressed key on the eight parallel lines and outputs a strobe pulse high for 1 ms. You want to poll the strobe over and over until you find it high. Then you want to read in the ASCII code, mask the parity bit (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 μs 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 μs (0.5 ms), output a low, wait 500 μs, output a high, etc. Remember that, before you can output to a port device, you must first initialize it as in Figure 4-18*a*. If you connect a buffer such as that shown in Figure 8-23 and a speaker to D0 of the port, you will be able to hear the tone produced.

CHAPTER 5

Strings, Procedures, and Macros

The last chapter showed you how quite a few of the 8086 instructions work and how jump instructions are used to implement IF-THEN-ELSE, WHILE-DO, and REPEAT-UNTIL program structures. The first section of this chapter introduces you to the 8086 string instructions, which can be used to repeat some operations on a sequence of data words in memory. The major point of this chapter, however, is to show you how to write and use subprograms called *procedures*. A final section of the chapter shows you how to write and use assembler *macros*.

OBJECTIVES

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

1. Use the 8086 string instructions to perform a variety of operations on a sequence of data words in memory.
2. Describe how a stack is initialized and used in 8086 assembly language programs which call procedures.
3. Write an 8086 assembly language program which calls a near procedure.
4. Write an 8086 assembly language program which calls a far procedure.
5. Write, assemble, link, and run a program which consists of more than one assembly module.
6. Write and use an assembler macro.

THE 8086 STRING INSTRUCTIONS

Introduction and Operation

A *string* is a series of bytes or words stored in successive memory locations. Often a string consists of a series of ASCII character codes. When you use a word processor or text editor program, you are actually creating a string of this sort as you type in a series of characters. One important feature of a word processor is the ability to move a sentence or group of sentences from one place in the text to another. Doing this involves moving a string of ASCII characters from one place in memory to another. The 8086 Move String instruction, MOVS, allows you to do operations such as this very easily.

Another important feature of most word processors is the ability to search through the text looking for a given word or phrase. The 8086 Compare String instruction, CMPS, can be used to do operations of this type. In a similar manner, the 8086 SCAS instruction can be used to search a string to see whether it contains a specified character. A couple of examples should help you see how these instructions work.

MOVING A STRING

Suppose that you have a string of ASCII characters in successive memory locations in the data segment, and you want to move the string to some new sequence of locations in the data segment. To help you visualize this, take a look at the strings we set up in the data segment in Figure 5-1*b*, p. 96, to test our program.

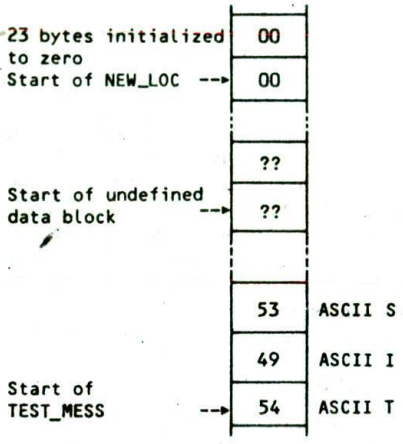
The statement TEST_MESS DB 'TIS TIME FOR A NEW HOME' sets aside 23 bytes of memory and gives the first memory location the name TEST_MESS. This statement will also cause the ASCII codes for the letters enclosed in the single quotes to be written in the reserved memory locations when the program is loaded in memory to be run. This array or string then will contain 54H, 49H, 53H, 20H, etc. The statement DB 100 DUP(?) will set aside 100 memory locations, but the DUP(?) in the statement tells the assembler not to initialize these 100 locations. We put these bytes in to represent the block of text that we are going to move our string over. The statement NEW_LOC DB 23 DUP(0) sets aside 23 memory locations and gives the first byte the name NEW_LOC. When this program is loaded in memory to be run, the 23 locations will be loaded with 00 as specified by the DUP(0) in the statement. To help you visualize this, Figure 5-1*a* shows a memory map for this data segment. Now that you understand the data structure for the problem, the next step is to write an algorithm for the program.

The basic pseudocode algorithm shown here for the operations you want to perform doesn't really help you see how you might implement the algorithm in assembly language.

```
REPEAT
MOVE BYTE FROM SOURCE STRING
        TO DESTINATION STRING
UNTIL ALL BYTES MOVED
```

In Chapter 3 we introduced you to the use of pointers to access data in sequential memory locations, so your next thought might be to expand the algorithm as shown next:

```
INITIALIZE SOURCE POINTER, SI
INITIALIZE DESTINATION POINTER, DI
INITIALIZE COUNTER, CX
REPEAT
    COPY BYTE FROM SOURCE TO DESTINATION
    INCREMENT SOURCE POINTER
    INCREMENT DESTINATION POINTER
    DECREMENT COUNTER
UNTIL COUNTER = 0
```



(a)

We often describe an algorithm in general terms at first and then expand sections as needed to help us see how the algorithm is implemented in a specific language. In the expanded algorithm you can see that as part of the initialization list you need to initialize the two pointers and a counter. The REPEAT-UNTIL loop then consists of moving a byte, incrementing the pointers to point to the source and destination for the next byte, and decrementing the counter to determine whether all the bytes have been moved.

As it turns out, the single 8086 string instruction, MOVSB, will perform all the actions in the REPEAT-UNTIL loop. The MOVSB instruction will copy a byte from the location pointed to by the SI register to a location pointed to by the DI register. It will then automatically increment SI to point to the next source location, and increment DI to point to the next destination location. Actually, as we will show you soon, we can specify whether we want SI and DI to increment or decrement. If you add a special prefix called the *repeat*

```
 1                                      ; 8086 PROGRAM F5-01.ASM
 2                                ;ABSTRACT  : Program moves a string from the location TEST_MESS
 3                                           ; to the location NEW_LOC.
 4                                ;REGISTERS ; Uses CS, DS, ES, SI, DI, AX, CX
 5                                ;PORTS     ; None used
 6
 7 0000                           DATA  SEGMENT
 8 0000  54 49 53 20 54 49 4D +        TEST_MESS DB 'TIS TIME FOR A NEW HOME' ; String to move
 9       45 20 46 4F 52 20 41 +
10       20 4E 45 57 20 48 4F +
11       4D 45
12 0017  64*(??)                              DB  100 DUP(?)              ; Stationary block of text
13 007B  17*(00)                       NEW_LOC   DB  23. DUP(0)            ; String destination
14 0092                          DATA  ENDS
15
16 0000                          CODE  SEGMENT
17                                     ASSUME CS:CODE, DS:DATA, ES:DATA
18
19 0000  B8 0000s                START:MOV AX, DATA        ; Initialize data segment register
20 0003  8E D8                         MOV DS, AX
21 0005  8E C0                         MOV ES, AX         ; Initialize extra segment register
22 0007  8D 36 0000r                   LEA SI, TEST_MESS  ; Point SI at source string
23 000B  8D 3E 007Br                   LEA DI, NEW_LOC    ; Point DI at destination location
24 000F  B9 0017                       MOV CX, 23         ; Use CX register as counter
25 0012  FC                            CLD                ; Clear direction flag so pointers autoincrement
26                                                        ; after each string element is moved
27 0013  F3> A4                  REP   MOVSB              ; Move string bytes until all moved
28
29 0015                          CODE ENDS
30                                    END START
```

(b)

FIGURE 5-1 Program for moving a string from one location to another in memory. (a) Memory map. (b) Assembly language program.

prefix in front of the MOVSB instruction, the MOVSB instruction will be repeated and CX decremented until CX is counted down to zero. In other words, the REP MOVSB instruction will move the entire string from the source location to the destination location if the pointers are properly initialized.

In order for the MOVSB instruction to work correctly, the source index register, SI, must contain the offset of the start of the source string, and the destination index register, DI, must contain the offset of the start of the destination location. Also, the number of string elements to be moved must be loaded into the CX register.

As we said previously, the string instructions will automatically increment or decrement the pointers after each operation, depending on the state of the direction flag DF. If the direction flag is cleared with a CLD instruction, then the pointers in SI and DI will automatically be incremented after each string operation. If the direction flag is set with an STD instruction, then the pointers in SI and DI will be automatically decremented after each string operation. For this example, it is easier to initialize the pointers to the starting offsets of each string and increment the pointers after each operation, so you will include the CLD instruction as part of the initialization.

Figure 5-1*b* shows how this algorithm can be implemented in assembly language. The first two MOV instructions in the program initialize the data segment register. The next instruction initializes the extra segment register. This is necessary because for string instructions, an offset in DI is added to the segment base represented by the number in the ES register to produce a physical address. If DS and ES are initialized with the same value, as we did with the first three instructions in this program, then SI and DI will point to locations in the same segment.

The next step in the program is to load SI with the effective address or offset of the first element in the source string. In the example we used the LEA instruction, but an alternative way to do this is with the instruction MOV SI,OFFSET TEST_MESS. The DI register is then initialized to contain the effective address or offset of the first destination location.

Next we load the CX register with the number of bytes in the string. Remember, CX functions as a counter to keep track of how many string bytes have been moved at any given time. Finally, we make the direction flag a zero with the Clear Direction Flag instruction, CLD. This will cause both SI and DI to be automatically incremented after a string byte is moved.

When the Move String Byte instruction, MOVSB, executes, a byte pointed to by SI will be copied to the location pointed to by DI. SI and DI will be automatically incremented to point to the next source and the next destination locations. The count register will be automatically decremented. The MOVSB instruction by itself will just copy one byte and update SI and DI to point to the next locations. However, as we said before, the repeat prefix, REP, will cause the MOVSB to be executed and the CX to be decremented over and over again until the CX register is counted down to zero. Incidentally, when the program is coded, the 8-bit code for the REP prefix, 11110011, is put in the memory location before the code for the MOVSB instruction.

After the MOVSB instruction is finished, SI will be pointing to the location after the last source string byte, DI will be pointing to the location after the last destination address, and CX will be zero.

The MOVSW instruction can be used to move a string of words. Depending on the state of the direction flag, SI and DI will automatically be incremented or decremented 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 MOVSB 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 MOVSB 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.

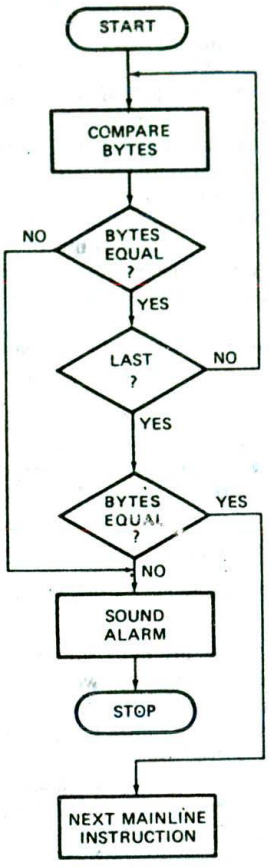
USING THE COMPARE STRING BYTE TO CHECK A PASSWORD

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, p. 98, 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 Figure 5-2*c*. 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, p. 99, 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 offset of each item in a segment. A $ is used to symbolically represent the current value of the locatio



(a)

```
REPEAT
    COMPARE SOURCE BYTE WITH DESTINATION BYTE
UNTIL (BYTES NOT EQUAL) OR (END OF STRING)
IF BYTES NOT EQUAL THEN
    SOUND ALARM
    STOP
ELSE DO NEXT MAINLINE INSTRUCTION
```

(b)

```
INITIALIZE PORT DEVICE FOR OUTPUT
INITIALIZE SOURCE POINTER – SI
INITIALIZE DESTINATION POINTER – DI
INITIALIZE COUNTER – CX
REPEAT
    COMPARE SOURCE BYTE WITH DESTINATION BYTE
    INCREMENT SOURCE POINTER
    INCREMENT DESTINATION POINTER
    DECREMENT COUNTER
UNTIL (STRING BYTES NOT EQUAL) OR (CX = 0)
IF STRING BYTES NOT EQUAL THEN
    SOUND ALARM
    STOP
ELSE DO NEXT MAINLINE INSTRUCTION
```

(c)

FIGURE 5-2 Flowchart and pseudocode for comparing strings program. (a) Flowchart. (b) Initial pseudocode. (c) Expanded pseudocode.

counter at any point. The statement STR_LENGTH EQU ($-PASSWORD) in the data segment then tells the assembler to compute the value for a constant called STR_LENGTH by subtracting the offset of PASSWORD from the current value in the location counter. The value of STR_LENGTH will be the length of the string PASSWORD. Note that the EQU statement must be in the data segment immediately after the password array so that the location counter contains the desired value. As you will see later, this trick with the $ sign allows you to load the number of string elements in CX symbolically, rather than having to manually count the number. This trick has the further advantage that if the password is changed and the program reassembled, the instruction that loads CX with the string length will automatically use the new value.

The statement INPUT_WORD DB 8 DUP(0) will set aside eight memory locations and assign the name INPUT_WORD to the first location. The DUP(0) in the statement tells the assembler to put 00H in each of these locations. We assume that a keyboard interface program section will load these locations with ASCII codes read from the keyboard as a user enters a password. We like to initialize locations such as this with zeros, so that during debugging we can more easily tell if the keyboard section correctly loaded the ASCII codes for the pressed keys in these locations.

Now let's look at the code segment section of the program. The ASSUME statement tells the assembler that the instructions will be in the segment CODE. It also tells the assembler that any references to the data segment or to the extra segment will mean the segment DATA. Remember that when you are using string instructions, you have to tell the assembler what to assume about the extra segment, because with string instructions an offset in DI is added to the extra segment base to produce the physical address.

The first three MOV statements in the program initialize the data and extra segment registers. Since we initialize DS and ES with the same values, both SI and DI will point to locations in the segment DATA. The next three instructions initialize port P2B of an SDK-86 board as an output port.

LEA SI,PASSWORD loads the effective address or offset of the start of the FAILSAFE string into the SI register. Since PASSWORD is the first data item in the segment DATA, SI will be loaded with 0000H. LEA DI,INPUT_WORD loads the effective address or offset of the start of the INPUT_WORD string into the DI register. Since the offset of INPUT_WORD is 0008H, DI will be loaded with this value. The MOV CX,STR_LENGTH statement uses the EQU we defined previously to initialize CX with the number of bytes in the string. The Clear Direction flag instruction tells the 8086 to automatically increment SI and DI after two string bytes are compared.

The CMPSB instruction will compare the byte pointed to by SI with the byte pointed to by DI and set the flags according to the result. It will also increment the pointers, SI and DI, to point to the next string elements. The REPE prefix in front of this instruction tells the 8086 to decrement the CX register after each compare, and repeat the CMPSB instruction if the compared bytes

```
1                                       ; 8086 PROGRAM F5-03.ASM
2                                ;ABSTRACT  : This program inputs a password and sounds an alarm
3                                           ; if the password is incorrect
4                                ;REGISTERS : Uses CS, DS, ES, AX, DX, CX, SI, DI
5                                ;PORTS     : Uses FFFAH - Port 2B on SDK-86 for alarm output
6
7 0000                           DATA  SEGMENT
8 0000  46 41 49 4C 53 41 46 +         PASSWORD    DB   'FAILSAFE'      ; Password
9       45
10      = 0008                          STR_LENGTH  EQU  ($ - PASSWORD) ; Compute length of string
11 0008 08*(00)                         INPUT_WORD  DB   8 DUP(0)       ; Space for user password input
12 0010                          DATA  ENDS
13
14 0000                          CODE  SEGMENT
15                                      ASSUME CS:CODE, DS:DATA, ES:DATA
16 0000 B8 0000s                        MOV  AX, DATA
17 0003 8E D8                           MOV  DS, AX             ; Initialize data segment register
18 0005 8E C0                           MOV  ES, AX             ; Initialize extra segment register
19 0007 BA FFFE                         MOV  DX, 0FFFEH         ; These next three instructions
20 000A B0 99                           MOV  AL, 99H            ; set up an output port on
21 000C EE                              OUT  DX, AL             ; the SDK-86 board
22 000D 8D 36 0000r                     LEA  SI, PASSWORD       ; Load source pointer
23 0011 8D 3E 0008r                     LEA  DI, INPUT_WORD     ; Load destination pointer
24 0015 B9 0008                         MOV  CX, STR_LENGTH     ; Load counter with password length
25 0018 FC                              CLD                     ; Increment DI & SI
26 0019 F3> A6                   REPE   CMPSB                   ; Compare the two string bytes
27 001B 75 03                           JNE  SOUND_ALARM        ; If not equal, sound alarm
28 001D EB 08 90                        JMP  OK                 ;  else continue
29 0020 B0 01                    SOUND_ALARM:MOV  AL, 01        ; To sound alarm, send a 1
30 0022 BA FFFA                         MOV  DX, 0FFFAH         ; to the output port whose
31 0025 EE                              OUT  DX, AL             ; address is in DX
32 0026 F4                              HLT                     ; and HALT.
33 0027 90                       OK:    NOP                     ; Program continues if password is OK
34 0028                          CODE   ENDS
35                                      END
```

FIGURE 5-3 Assembly language program for comparing strings.

were equal *and* CX is not yet decremented down to zero. As we mentioned before, when this instruction is coded, the code for the prefix will be put in memory before the code for the CMPSB instruction.

If the zero flag is not set when execution leaves the repeat loop, then we know that the two strings are not equal. This means that the password entered was not valid, so we want to sound an alarm. The JNE SOUND_ALARM will check the zero flag and, if it is not set, do a jump to the specified label. If the zero flag is set, indicating a valid password, then execution falls through to the JMP OK instruction. This JMP instruction simply jumps over the instructions which sound the alarm and stop the computer.

For this example, we assume that the alarm control is connected to the least significant bit of port FFFAH and that a 1 output to this bit turns on the alarm. The MOV AL,01 instruction loads a 1 in the LSB of AL. The MOV DX,0FFFAH instruction points DX at the port that the alarm is connected to, and the OUT DX,AL instruction copies this byte to port FFFAH. Finally, the HLT instruction stops the computer. An interrupt or reset will be required to get it started again.

As the preceding examples show, the string instructions make it very easy to implement some commonly needed REPEAT-UNTIL algorithms. Some of the programming problems at the end of the chapter will give you practice with MOVS, CMPS, and SCAS instructions.

WRITING AND USING PROCEDURES

Introduction

Often when writing programs you will find that you need to use a particular sequence of instructions at several different points in a program. To avoid writing the sequence of instructions in the program each time you need them, you can write the sequence as a separate "subprogram" called a *procedure*. Each time you need to execute the sequence of instructions contained in the procedure, you use the CALL instruction to send the 8086 to the starting address of the procedure in memory. Figure 5-4*a*, p. 100, shows in diagram form how a CALL instruction causes execution to go from the mainline program to a procedure. A RET instruction at the end of the procedure returns execution to the next instruction in the mainline. As shown in Figure 5-4*b*, procedures can even be "nested." This means that one procedure calls another procedure as part of its instruction sequence. Follow the arrows in Figure 5-4*b*

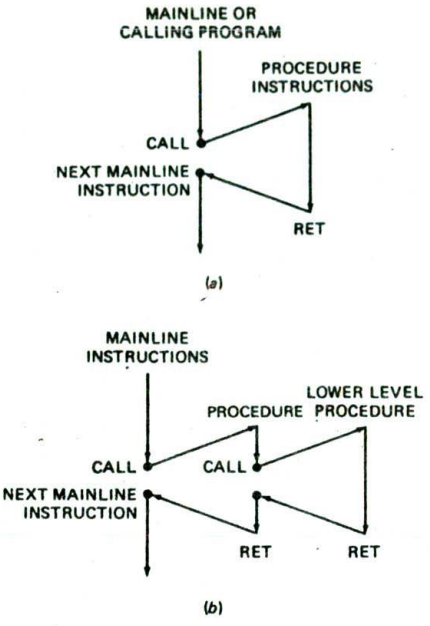


FIGURE 5-4 Program flow to and from procedures. (*a*) Single procedures. (*b*) Nested procedures.

to see how this works. Now, before we get into the details of how to write and use procedures, we need to discuss another reason we use procedures in programs.

Recall from Chapter 2 the *top-down design* approach to solving a programming problem. In this approach, the problem is carefully defined, and then the overall job is broken down into modules. Each of these modules is broken down into smaller modules. The division process is continued until the algorithm for each module is clearly obvious. Figure 5-5 shows an example of how this modular structure can be represented in diagram form. A diagram such as this is often called a *hierarchical chart*. The point of all this is to break a large problem down into manageable-size pieces which can be individually written, tested, and debugged. The individual modules are usually written as procedures and called from a mainline program which implements the highest level of the hierarchy. This approach has the added advantage that a person can read the mainline program to get an overview of what the program does and then work down into the procedures to see the amount of detail needed at a particular point. Also, tested and debugged procedures can be used in writing new programs. Now that you know what procedures are used for, we will discuss the 8086 CALL and RET.

The 8086 CALL and RET Instructions

As shown in Figure 5-4, a CALL instruction in the mainline program loads the instruction pointer and in some cases also the code segment register with the starting address of the procedure. The next instruction fetched will be the first instruction of the procedure. At the end of the procedure, a RET instruction sends execution back to the next instruction after the CALL in the mainline program. The RET instruction does this by loading the instruction pointer and, if necessary, the code segment register with the address of the next instruction after the CALL instruction.

The question that may occur to you at this point is, "If a procedure can be called from anywhere in a program, how does the RET instruction know where to return execution to?" The answer to this question is that when a CALL instruction executes, it automatically stores the return address in a special section of memory called the *stack*. A later section will introduce you to how the 8086 stack works. For now, let's take a closer look at the 8086 CALL and RET instructions.

THE CALL INSTRUCTION OVERVIEW

As we said previously, the 8086 CALL instruction performs two operations when it executes. First, it stores the address of the instruction after the CALL instruction on the stack. This address is called the *return* address because it is the address that execution will return to after the procedure executes. If the CALL is to a procedure in the same code segment, then the call is *near*, and only the instruction pointer contents will be saved on the stack. If the CALL is to a procedure in another code segment, the call is *far*. In this case, both the instruction pointer and the code segment register contents will be saved on the stack.

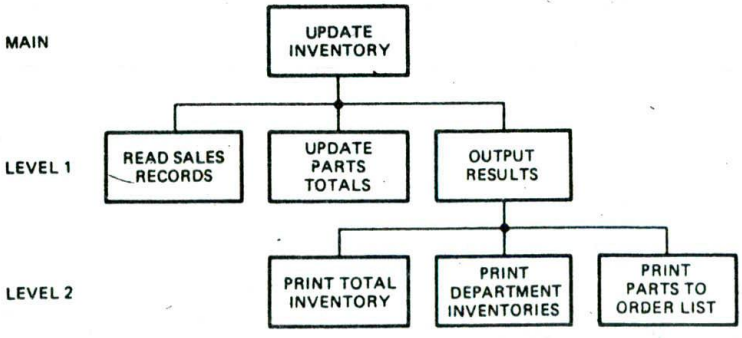


FIGURE 5-5 Hierarchical chart for inventory update program.

The second operation of the CALL instruction is to change the contents of the instruction pointer and, in some cases, the contents of the code segment register to contain the starting address of the procedure. This second function of the CALL instruction is very similar to the operation of the JMP instructions we discussed in Chapter 4.

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, Figure 5-6*a* 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

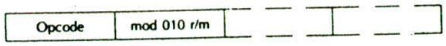
The indirect within-segment CALL instruction is also a near call. When this form of CALL executes, the instruction pointer is replaced with a 16-bit value from a specified register or memory location. As indicated by the MOD-R/M byte in the coding template, the source of the value can be any of the eight 16-bit registers or a memory location specified by any one of the 24 addressing modes shown in Figure 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

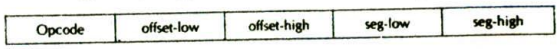
Opcode	Clocks	Operation
E8	19	IP ← IP+Disp16—(SP) ← return link

Within segment or group, Indirect



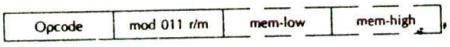
Opcode	Clocks	Operation
FF	16	IP ← Reg16—(SP) ← return link
FF	21+EA	IP ← Mem16—(SP) ← return link

Inter-segment or group, Direct



Opcode	Clocks	Operation
9A	28	CS ← segbase IP ← offset

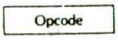
Inter-segment or group, Indirect



Opcode	Clocks	Operation
FF	37+EA	CS ← segbase IP ← offset

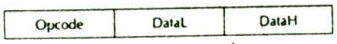
(*a*)

RET = Return from Subroutine



Opcode	Clocks	Operation
C3	8	intra-segment return
CB	18	inter-segment return

Return and add constant to SP



Opcode	Clocks	Operation
C2	12	intra-segment ret and add
CA	17	inter-segment ret and add

(*b*)

FIGURE 5-6 8086 CALL and RET instruction formats. (*a*) CALL. (*b*) RET. (*Intel Corporation*)

THE DIRECT INTERSEGMENT FAR CALL

The direct intersegment far call is used when the procedure is in a segment with a different name from that where the CALL is located. If the procedure is in another segment, you have to change both the instruction

pointer and the code segment register to get to it. For this form of the CALL instruction, the new value for the instruction pointer is written in as bytes 2 and 3 of the instruction code. Note that the low byte of the new IP value is written before the high byte. The new value for the code segment register is written in as bytes 4 and 5 of the instruction code. Again the low byte is written before the high byte. A program example later in this chapter shows you how to write your programs so that an assembler can find a procedure label in another segment.

THE INDIRECT INTERSEGMENT FAR CALL

This form of the CALL instruction replaces the instruction pointer and the code segment register contents with two 16-bit values from memory. Since two 16-bit values are needed, the values cannot come from a register. The MOD-R/M byte in the instruction is used to specify the addressing mode for the memory location where the 8086 goes to get the new values. The first word from memory is put in the instruction pointer, and the second word from memory is put in the code segment register. The instruction CALL DWORD PTR [BX], for example, will get a new value for IP from [BX] and [BX + 1] in the data segment and a new value for CS from offsets [BX + 2] and [BX + 3] in the data segment.

THE 8086 RET INSTRUCTION

When the 8086 does a near call, it saves the instruction pointer value for the instruction after the CALL on the stack. A RET at the end of the procedure copies this value from the stack back to the instruction pointer to return execution to the calling program. This then returns execution to the mainline program. When the 8086 does a far call, it saves the contents of both the instruction pointer and the code segment register on the stack. A RET instruction at the end of the procedure copies these values from the stack back into the IP and CS registers to return execution to the mainline program. Obviously we need one form of the RET instruction to handle returns from near procedures and another form of the instruction to handle returns from far procedures. Actually, the 8086 has four forms of the RET instruction Figure 5-6*b* shows the coding templates for these four.

The simple within-segment form of RET copies a word from the top of the stack to the instruction pointer register. This is the instruction form you will usually use to return from a near procedure. The within-segment adding immediate to SP form is also used to return from a near procedure. When this form executes, however, it will copy the word at the top of the stack to the instruction pointer and also add an immediate number contained in the instruction to the contents of SP. Later, we show you what this form is used for.

The intersegment form of the RET instruction is used to return from far procedures. When this form of the RET instruction executes, it will copy the word from the top of the stack to the instruction pointer. It will then increment the stack pointer by 2 and copy the next word from the stack to the code segment register. The intersegment adding immediate to SP form of the instruction also copies a new value for IP and a new value for CS from the stack. However, it also adds a 16-bit immediate number contained in the instruction code to SP.

NOTE: If you are using an assembler, the assembler will automatically code a near RET for a near procedure and a far RET for a far procedure.

The 8086 Stack

Throughout the preceding discussions of the CALL and RET instructions, we have talked about writing words to the stack and copying these words back to the instruction pointer and/or code segment register. Now we will show you how to set up a stack in your programs and how the stack is used.

The *stack* is a section of memory you set aside for storing return addresses. The stack is also used to save the contents of registers for the calling program while a procedure executes. A third use of the stack is to hold data or addresses that will be acted upon by a procedure.

The 8086 lets you set aside up to an entire 64-Kbyte segment of memory as a stack. Remember from the block diagram in Figure 2-7 that the 8086 contains a stack segment register and a stack pointer register. The stack segment register is used to hold the upper 16 bits of the starting address you give to the stack segment. If you decide to start the stack segment at 70000H, for example, the stack segment register will contain 7000H. The stack pointer register is used to hold the offset of the last word written on the stack. The 8086 produces the physical address for a stack location by adding the offset contained in the SP register to the stack segment base address represented by the 16-bit number in the SS register.

An important point about the operation of the stack is that the SP register is automatically decremented by 2 before a word is written to the stack. This means that at the start of your program you must initialize the SP register to point to the top of the memory you set aside as a stack, rather than initializing it to point to the bottom location. To help you visualize this, Figure 5-7 shows how we set up the stack in memory for this example program.

Before a CALL instruction, assume that the SS register contains 7000H and the SP register contains 0050H. The physical address of the current top of the stack, then, will be 70050H. If the 8086 executes a near CALL instruction, the SP register will automatically be decremented by 2 and the contents of the IP register will be written to the stack as shown.

When a near RET instruction executes, the IP value stored in the stack will be copied back to the IP register, and the SP register will be automatically incremented by 2. After a CALL—RET sequence, then, the SP register is again pointing to the initial top-of-stack location.

From the preceding discussion you should see that if you are going to call procedures or use the stack in some other way in your program, you need to declare a stack

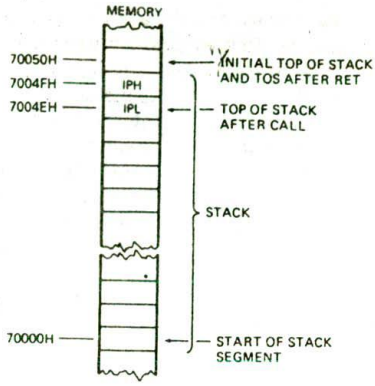


FIGURE 5-7 Stack diagram showing how the return address is pushed onto the stack by CALL.

segment at the start of your program. You also need to initialize the SS register with the base address of the stack segment and initialize the SP register with the offset of the top of the stack. Figure 5-8 shows the pieces you need to add to your programs to declare a stack segment and to initialize SS and SP.

The STACK_SEG SEGMENT STACK and STACK_SEG ENDS statements in Figure 5-8 are used to declare a logical segment that will be used for the stack. The STACK directive tells the assembler that this segment will be used as a last-in–first-out stack.

> NOTE: If you are going to use the IBM program EXE2BIN on your programs so that you can download them to an SDK-86, omit the STACK directive here. The linker will then give you an error message, WARNING—NO STACK SEGMENT, but you can ignore this warning.

You don't need all 64,000 bytes of the logical segment in your programs, so you tell the assembler to set aside 40 decimal or 28H words of storage in this logical

```
; 8086 PROGRAM fragment showing the initialization
; of stack segment register and stack pointer register

STACK_SEG SEGMENT STACK
          DW      40 DUP(0)
STACK_TOP LABEL   WORD
STACK_SEG ENDS

CODE      SEGMENT
          ASSUME CS:CODE, SS:STACK_SEG
          MOV AX, STACK_SEG ; Initialize stack
          MOV SS, AX        ; segment register
          LEA SP, STACK_TOP ; Initialize stack pointer
;             :                 Continue with program
;             :
CODE      ENDS
          END
```

FIGURE 5-8 Required program additions when using a stack.

segment with the DW 40 DUP(0) statement. If the actual stack is limited to approximately the size actually needed, this segment can be overlapped with other logical segments to save on the amount of physical memory required for a program.

Since words are written to the stack starting from the highest location, it is convenient to have a name attached to this location so that you can initialize the SP register with a name instead of a number. The statement STACK_TOP LABEL WORD in Figure 5-8 gives the name STACK_TOP to the next even address after the 40 words you set aside for the stack.

We arbitrarily choose to start the stack segment at address 70000H for this example, and we set a stack length of 40 words with the DW 40 DUP(0) statement. Since each memory address represents a byte, these 40 words will occupy the 80 addresses 70000H to 7004FH, as shown in Figure 5-7. The label STACK_TOP is associated with address 70050H, the next address after the stack. We will explain later why you want the name at the address after the actual stack.

The next program addition you need to look at is in the ASSUME statement. Note that we have added the term SS:STACK_SEG to tell the assembler that any reference in the program to the stack means the segment STACK_SEG. This term tells the assembler that SS will contain the starting address of STACK_SEG, but it does not load this value into the SS register. Loading the SS register must be done with program instructions, just as you initialize the data segment register and the extra segment register with program instructions. Remember, you can't load an immediate number directly into a segment register, so you load the starting address of the segment into a register and then copy it into the stack segment register. The MOV AX,STACK_SEG and MOV SS,AX instructions do this. Now all you have to do is initialize the stack pointer.

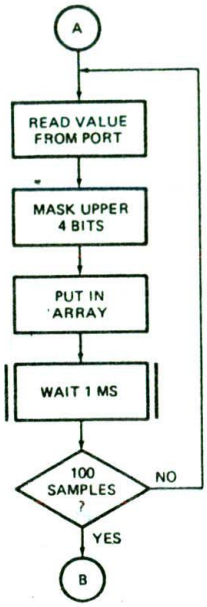
You want to initialize SP so that the first word written to the stack goes to the highest location in the memory you set aside for the stack. All the instructions which write a word to the stack decrement the stack pointer by 2 before writing the word. Therefore, you want the stack pointer to be initially loaded with the next even address above the actual stack. We gave this location the name STACK_TOP, so you can use the LEA SP,STACK_TOP instruction to initialize the stack pointer with the desired offset. You could also have used the instruction MOV SP,OFFSET STACK_TOP to initialize the stack pointer.

Now that you know the initialization steps required in a program that uses procedures, we will show you how to write and call a procedure. We will also take another look at how the stack functions during a CALL—RET sequence.

A Near Procedure Call and Return Example

DEFINING THE PROBLEM AND WRITING THE ALGORITHM

Delay loops such as that shown in Figure 4-20 are often written as procedures so that they can be called from anywhere in a program. Suppose that you want to write



(*a*)

```
Initialize
REPEAT
   Get data sample from port
   Mask upper 4 bits
   Put in array
   Wait 1 ms
UNTIL 100 samples taken
```

(*b*)

```
DATA SAMPLES PROGRAM
Initialize pointer to array, SI
Initialize counter, BX
REPEAT
   Read port
   Mask upper 4 bits
   Put in array
   Wait 1 ms procedure
   Increment pointer, SI
   Decrement counter, BX
UNTIL counter = 0

WAIT-1MS  PROCEDURE
   Load count value
   REPEAT
      Decrement count value
UNTIL count = 0
```

(*c*)

FIGURE 5-9 Algorithm for data samples at 1-ms intervals program. (*a*) Flowchart. (*b*) Pseudocode. (*c*) Pseudocode expanded.

a program which reads 100 data words from a port at 1-ms intervals, masks the upper 4 bits of each word, and puts each result in an array in memory. Before you read on, see if you can write a flowchart or pseudocode for this problem. Then compare your results with those in Figure 5-9*a* or *b*. We hope you recognized this problem as a REPEAT-UNTIL situation.

The next step is to expand the algorithm to take into account the specific architectural features of the 8086 that you can use to implement the algorithm. Figure 5-9*c* shows one way to do this expansion.

At the start you initialize a pointer to the array and a counter to keep track of how many values have been put 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.

Note that even the expanded algorithm in Figure 5-9*c* is general enough that it could be implemented on almost any microprocessor. Let's see how it can be translated to run on an 8086.

THE 8086 ASSEMBLY LANGUAGE PROGRAM

Figure 5-10 shows the assembly language program for the expanded algorithm in Figure 5-9*c*. Read through this program and see how much of it you can remember and/or figure out before you read our explanations in the following paragraphs. Deciphering a program written by someone else is an important skill to develop.

At the start of the program, we declare a logical segment for data with the DATA SEGMENT—DATA ENDS statements. The statement PRESSURES DW 100 DUP(0) in this segment sets aside 100 words of memory to store the values read in from a pressure sensor. This statement also initializes these 100 words to all 0's. It really doesn't matter what values are initially in these locations because the program is going to write values in them. However, as we mentioned in an earlier example, we like to initialize arrays such as this to all 0's so that during debugging we can tell whether the program wrote any values to these locations.

Next, we declare a logical segment to be used for the stack with the STACK_SEG SEGMENT and STACK_SEG ENDS statements. As described previously, the statement DW 40 DUP(0) sets up a stack length of 40 words and initializes these words to all 0's. Again we really don't care what value these words have initially because the 8086 will be writing values there as we call procedures. The statement STACK_TOP LABEL WORD gives a name to the next even address after the highest address in the stack we set up.

Now let's work our way through the main program and the procedure in the code segment. We have to tell the assembler which logical segments are being used for code, data, and stack in the program. The ASSUME CS:CODE, DS:DATA, SS:STACK_SEG statement does this. The ASSUME statement, however, does not actually initialize the segment registers. We have to do this with

```
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       = FFF8                   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  B8 0000s                START:     MOV  AX, DATA            ; Initialize data segment register
24 0003  8E D8                          MOV  DS, AX
25 0005  B8 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 ; Intialize 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 FFF8                        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, 0FFFH           ; 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                             INC  SI
38 0022  4B                             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 23F2                        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
49                                      END
```

FIGURE 5-10 Assembly language program to read in 100 samples of data at 1-ms intervals.

program instructions. The MOV AX,DATA and MOV DS,AX instructions initialize the data segment register. The MOV AX,STACK_SEG and MOV SS,AX instructions initialize the stack segment register. The MOV SP,OFFSET STACK_TOP statement initializes the stack pointer register. The program to this point is essentially just housekeeping chores. After a few more initialization instructions, you will finally see some action.

The statement LEA SI,PRESSURES initializes the SI register as a pointer to the first location in the array PRESSURES. It loads the effective address or offset of the first word in PRESSURES into SI. For our example here, PRESSURES is the first data item in the segment, so the value loaded into SI will be 0000H. We chose to use the BX register as a sample counter, so we use the statement MOV BX,NBR_SAMPLES to initialize BX with the number of samples we want to take and store. We could have just used the instruction MOV BX,100 to initialize BX with the number of words in the array. However, representing the number of samples symboli-

cally ensures that this number will get updated if we increase the length of PRESSURES. To represent the length of PRESSURES symbolically, we used the NBR_SAMPLES EQU ((S-PRESSURES)/2) statement in the data segment. The (S-PRESSURES) in this statement tells the assembler to subtract the offset of PRESSURES from the value in the location counter. This value then represents the number of bytes in the array. The /2 in the expression tells the assembler to divide the number of bytes by 2 to give the number of words, which is the number we want to load into BX. Finally, we are going to get to some action.

The final initialization instruction is to point DX at the port that we will read to get the data value from the pressure sensor. As indicated by the PRESSURE_PORT EQU OFFF8H statement at the top of the program, the pressure sensor is connected to port FFF8H. Since this port address is larger than FFH, we have to use the variable-port input instruction. For this input instruction, we first load the port address in the DX register with the MOV DX,PRESSURE_PORT instruction, then read the data word in with the IN AX,DX instruction. Notice how much more understandable it makes a program when we use a name such as PRESSURE_PORT in an instruction rather than 0FFF8H, the numerical port address. If you are working with an assembler, EQU statements are a handy way to give names to constants in your program.

When we get the pressure value into AX, we mask out the upper 4 bits with the AND AX,0FFFH instruction. The reason why we want to do this is that the analog-to-digital converter that the pressure sensor is connected to is a 12-bit unit. The upper four bits of the 16-bit port are not connected to anything and may pick up random noise signals. To prevent noise signals on the upper 4 bits from getting put in memory with our data, we mask these bits out by ANDing them with 0's. The instruction MOV [SI],AX then copies the data word from the AX register to the memory location pointed to by SI in the data segment.

To produce the desired delay between samples, we CALL the WAIT_1MS procedure. This is a direct within-segment CALL because the procedure is contained in the same code segment as the CALL instruction.

We use the PROC and ENDP directives to "bracket" the assembly language statements of the procedure. Putting a name in front of these directives allows us to call the procedure by name. For the example in Figure 5-10, we gave the procedure the name WAIT_1MS to remind us of the function of the procedure. To produce the desired delay, we load a number into the CX register with the MOV CX,23F2H instruction and count the number down to 0 with the LOOP HERE instruction. The LOOP instruction, remember, decrements CX by 1 and jumps to the specified label if CX is not yet down to 0. Since we put the label HERE directly on the LOOP instruction, the LOOP instruction will simply execute over and over until CX reaches 0. When CX gets counted down to zero, the RET instruction at the end of the procedure will return execution to the next instruction after the CALL in the mainline program.

Since this procedure is in the same code segment as the mainline program, only the instruction pointer has to be changed to get back to the mainline. This is an example of a near procedure return. If you are hand coding a program such as this, make sure to use the correct form of the RET instruction.

Now, back in the mainline program, we need to get ready to read the next data value. First, we want to get SI pointed to the location where we want to put the next data word. Since each address represents a byte, and we are storing words, we have to increment the pointer by 2 to point to the next storage location. We used two INC SI instructions to do this, but you could use the single instruction ADD SI,02H to do the same job. After updating the pointer, we decrement the sample counter in BX with the DEC BX instruction. If BX is not yet counted down to 0, the JNZ NEXT_VALUE instruction will cause the 8086 to read in and process another value from the port. If BX is 0, indicating that all 100 samples have been taken, execution goes on to the next mainline instruction after JNZ. Now let's take another look at what happens to the stack and the stack pointer as this example program executes.

Another Look at Stack Operation During a CALL and RET

For the example program in the last section, we started the stack at address 70000H, so the stack segment register was initialized with 7000H. We set a stack length of 40 decimal or 28H words with the DW 40 DUP(0) statement. These 40 words will occupy the 80 (50H) memory locations from 70000H to 7004FH, as shown in Figure 5-11*a*. Initially we want the stack pointer to point at the next address above the stack. Therefore, we initialized the stack pointer to offset 0050H, the next even address above our actual stack, with the MOV SP,OFFSET STACK_TOP instruction.

After the 8086 fetches the CALL instruction from the instruction-byte queue in the BIU, it automatically increments the instruction pointer to 0020H, the offset of the next instruction after the CALL. You can see this if you look at line 36 in the program listing in Figure 5-10. The instruction pointer then contains the address we want execution to return to after the procedure is completed. When the near CALL instruction in our example program executes, the 8086 first decrements the stack pointer by 2. Then it copies the return address in the instruction pointer to the memory location now pointed to by the stack pointer. If the stack pointer contained 0050H before being decremented, then after being decremented by 2 it contains 004EH. The physical address pointed to by the stack pointer and the stack segment register will be 7004EH. The low byte of the instruction pointer will be copied to address 7004EH, and the high byte of the instruction pointer will be copied to address 7004FH, as shown in Figure 5-11*a*. This follows the Intel convention of putting the lower byte of a word at the lower address in memory. After the CALL instruction executes, the stack pointer is left

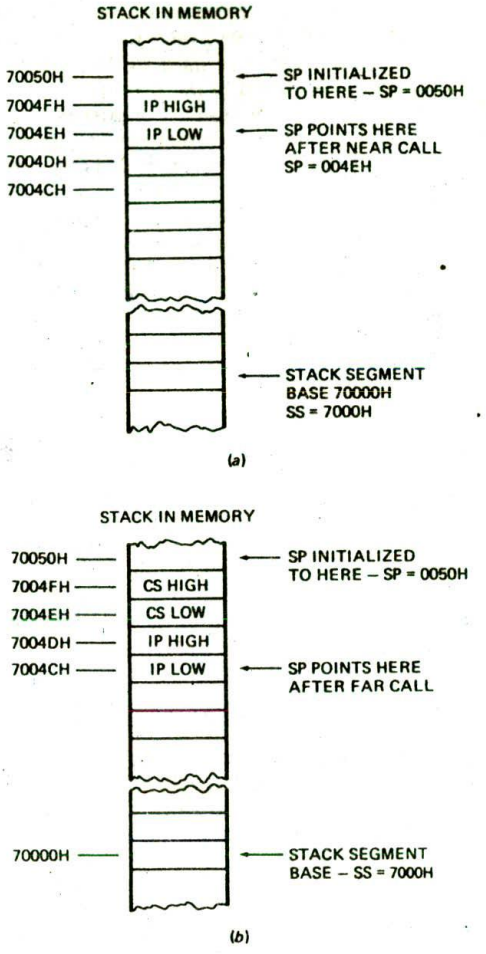


FIGURE 5-11 Stack diagram for program in Figure 5-10. (*a*) For near CALL. (*b*) For far CALL.

pointing to offset 004EH. This location is now the top of the stack, or TOS.

When the RET instruction at the end of the procedure in the example program executes, the 8086 copies the return address from the top of the stack to the instruction pointer. Since the top of the stack was at offset 004EH, the word from addresses 7004EH and 7004FH will be copied to the instruction pointer. After it copies the word from the top of the stack to the instruction pointer, the 8086 increments the stack pointer by 2. For our example here, it will increment the stack pointer from 004EH to 0050H. The stack pointer is now back where it was before the CALL instruction executed. Note that the return address is still present in memory because the RET instruction simply copied it to the instruction pointer and incremented the stack pointer over it.

When the 8086 executes a far CALL instruction, it decrements the stack pointer by 2 and copies the contents of the code segment register to the stack. It then decrements the stack pointer by 2 again and copies the offset of the next mainline instruction from the instruction pointer to the stack. To help you visualize this, Figure 5-11*b* shows how these would be written to the stack, assuming the same stack starting addresses that we used for the previous example. As you can see from this figure, after a far CALL the top of the stack will be four addresses lower than it was before the CALL.

A far RET used at the end of a far procedure will copy the word from the top of the stack to the instruction pointer and increment the stack pointer by 2. It will then copy the word from the new top of the stack to the code segment register. The next instruction will then be fetched from the physical address after the far CALL instruction. The top of the stack will be back to where it was before the CALL and RET.

As we mentioned previously, the stack is also used to save the contents of registers while a procedure executes and to hold data that the procedure is to act on. The next section shows you how we do this.

Using PUSH and POP to Save Register Contents

In the example program in Figure 5-10, we used the BX register to keep track of how many data samples we had taken in. After each data sample was taken in, we decremented the BX register and used the JNZ instruction to determine whether to take another sample or to exit. We would like to have used the CX register to keep track of the number of samples taken so that we could have used a single LOOP instruction instead of the DEC BX and JNZ label instructions. The reason that we couldn't use CX for this in the program is because CX is used in the procedure. Any value we put in CX in the mainline program would be written over by the MOV CX,23F2H instruction in the procedure. It is very common to want to use registers both in the mainline program and in a procedure without the two uses interfering with each other. The PUSH and POP instructions make this very easy to do.

The PUSH register/memory instruction decrements the stack pointer by 2 and copies the contents of the specified 16-bit register or memory location to memory at the new top-of-stack location. The PUSH CX instruction, for example, will decrement the stack pointer by 2 and copy the contents of the CX register to the stack where the stack pointer now points. This instruction, then, can be used to save the contents of CX while a procedure executes. The next question is, how do we get the saved value back when we want it?

The POP register/memory instruction copies a word from the top of the stack to the specified 16-bit register or memory location and increments the stack pointer by 2. The POP CX instruction, for example, will copy a word from the top of the stack to the CX register and increment the stack pointer by 2. After a POP, the stack pointer will point to the next word on the stack.

You can PUSH any of the 16-bit general-purpose registers AX, BX, CX, and DX; any of the base or pointer registers BP, SP, SI, and DI; any of the segment registers

CS, DS, SS, and ES; or even a word from a memory location specified by one of those 24 memory addressing modes in Figure 3-8. A separate instruction, PUSHF, 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. Figure 5-12*a* 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 5-12*b* 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 Figure 5-12*a*. 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. Likewise, we often want a procedure to make some processed data values or addresses available to the main program. These addresses or data values passed back and forth between the mainline and the procedure are commonly called *parameters*. The four major ways of passing parameters to and from a procedure are:

1. In registers
2. In dedicated memory locations accessed by name

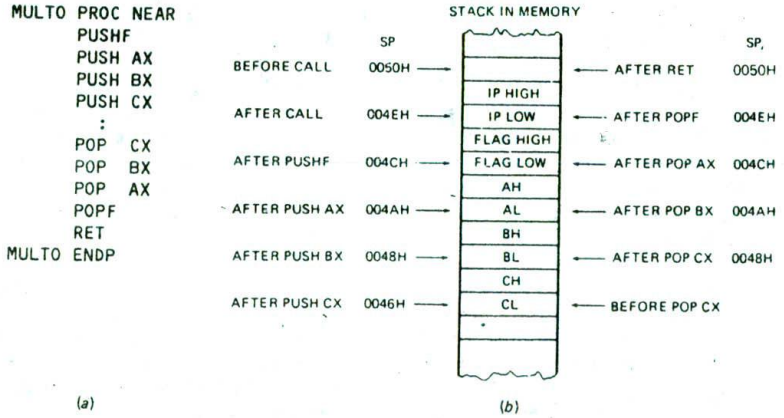


FIGURE 5-12 . Using PUSH and POP instructions. (*a*) Instruction sequence. (*b*) Effect on stack and stack pointer.

```
4596 = (4 x 1000) + (5 x 100) + (9 x 10) + (6 x 1)

   1 = 0001H   therefore     6 = 6 x 0001H = 0006H
  10 = 000AH   therefore    90 = 9 x 000AH = 005AH
 100 = 0064H   therefore   500 = 5 x 0064H = 01F4H
1000 = 03E8H   therefore  4000 = 4 x 03E8H = 0FA0H
                          ----               ------
                          4596               = 11F4H
```

FIGURE 5-13 BCD-to-binary algorithm.

3. With pointers passed in registers
4. With the stack

In the following sections we use a simple program to show you how each of these methods works.

DEFINING THE PROGRAMMING PROBLEM

A common programming need is to convert a packed BCD number to its binary equivalent. You might, for example, want to convert a packed BCD such as 0100 0101 1001 0110, which represents 4596 decimal, to 0001000111110100 binary, or 11F4H. There are several ways to do this conversion, but to us the easiest is based on using the value of each placeholder in the BCD number.

Figure 5-13 shows the names and values for each digit in a four-digit BCD number such as 4596. When you write a number such as this, it means that you have a total of 4 thousands + 5 hundreds + 9 tens + 6 units. To determine the value of this number in binary, you just multiply the number in each digit position by the value of that digit position in binary and add up the results. The right-hand side of Figure 5-13 shows how this works. A microprocessor, of course, uses the binary equivalents, but to make it easier for you to see what is going on here, we have represented the binary values with their hexadecimal equivalents.

The units position has a value of 1 in hex, so multiplying this by 6 units gives 0006H. The tens position has a value of 1010 binary, or 0AH. Multiplying this value by 9, the number of tens, gives 005AH. The value of the hundreds position in the BCD number is 01100100 binary, or 64H. When you multiply this value by 5, the number of hundreds, you get 01F4H. When you multiply the hex value of the thousands position, 03E8H, by 4 (the number of thousands), you get 0FA0H. Adding up the results for the four digits gives 11F4H or 0001000111110100, which is the binary equivalent of 4596 BCD. You can use this method to convert a BCD number with any number of digits to its binary equivalent, but to save space here we will show the program for just a two-digit BCD number.

From the example in Figure 5-13, perhaps you can see that the algorithm for this program is the simple sequence of operations

Separate nibbles

Save lower nibble (don't need to multiply by 1)

Multiply upper nibble by 0AH

Add lower nibble to result of multiplication

We want to implement this program as a procedure which can be called from anywhere in a mainline program. For our first version, we pass the BCD number to the procedure in a register.

PASSING PARAMETERS IN REGISTERS

Figure 5-14, p. 110, shows our first version of a procedure to convert a two-digit packed BCD number to its binary equivalent. The BCD number is copied from memory to the AL register and then passed to the procedure in the AL register. We start the procedure by pushing the flag register and the other registers we use in the procedure. Notice that we don't push the AX register because we are using it to pass a value to the procedure and expecting the procedure to pass a different value back to the calling program in it.

The function of the rest of the instructions in the procedure should be reasonably clear from the comments with them. We first make a copy of the BCD number in AL to BL so that we have two copies to work on. We then mask the upper nibble of the copy in BL. Since multiplying this nibble by 1 would not change its value, we are done with it for now. Next, we mask the lower nibble of the other copy of the BCD number and rotate this nibble into the lower nibble position of the byte so that we can multiply it correctly. When we multiply this nibble by the digit weight of 0AH, the result is left in the AX register. However, since the result can never be greater than 8 bits, we can disregard the contents of AH. Finally, we add the lower nibble we saved in BL to the result in AL to get the binary total. The desired result is left in AL. Before returning to the main program, we pop the registers we pushed at the start of the procedure. Since we did not push AX, the binary value in AX at the end of the procedure will be there when execution returns to the calling program.

The disadvantage of using registers to pass parameters is that the number of registers limits the number of parameters you can pass. You can't, for example, pass an array of 100 elements to a procedure in registers.

PASSING PARAMETERS IN GENERAL MEMORY

As you read through the preceding example, the question that may have occurred to you is, "Why didn't you simply access the BCD_INPUT value and the BIN_VALUE by name from the procedure?" The answer to the question is that we can directly access the parameters by name from the procedure, but in some cases there are problems with doing it this way. Figure 5-15, p. 111, shows a procedure that accesses the parameters directly by name.

In this procedure we first push the flags and all the registers used in the procedure. We then copy the BCD number into AL with the MOV AL,BCD_INPUT 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.

```
                        ; 8086 PROGRAM F5-14.ASM
                ;ABSTRACT  : BCD to BINARY conversion program that uses a
                        ; procedure to convert BCD numbers to binary.
                        ; Program uses the AL register to pass parameters
                        ; to the procedure
                ;REGISTERS : Uses CS, DS, SS, SP, AX
                ;PORTS     : None Used
                ;PROCEDURES: BCD_BIN

0000            DATA        SEGMENT
0000  17            BCD_INPUT   DB 17H      ; storage for BCD value
0001  ??            BIN_VALUE   DB ?        ; storage for binary value
0002            DATA        ENDS

0000            STACK_SEG   SEGMENT     STACK
0000  64*(0000)             DW 100 DUP(0)  ; stack of 100 words
                        TOP_STACK   LABEL   WORD
00C8            STACK_SEG   ENDS

0000            CODE        SEGMENT
                        ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
0000  B8 0000s  START:      MOV  AX, DATA           ; Initialize data segment
0003  8E D8                 MOV  DS, AX             ; register
0005  B8 0000s              MOV  AX, STACK_SEG      ; Initialize stack segment
0008  8E D0                 MOV  SS, AX             ; register
000A  BC 00C8r              MOV  SP, OFFSET TOP_STACK ; Initialize stack pointer

000D  A0 0000r              MOV  AL, BCD_INPUT
0010  E8 0005               CALL BCD_BIN            ; Do the conversion
0013  A2 0001r              MOV  BIN_VALUE, AL      ; Store the result
0016  90                    NOP                     ; Continue with program here
0017  90                    NOP                     ;

                ;PROCEDURE:  BCD_BIN - Converts BCD numbers to binary.
                ;INPUT     :  AL with BCD value
                ;OUTPUT    :  AL with binary value
                ;DESTROYS  :  AX

0018            BCD_BIN     PROC  NEAR
0018  9C                   PUSHF               ; Save flags
0019  53                   PUSH BX             ; and registers used in procedure
001A  51                   PUSH CX             ; before starting the conversion
                ;Do the conversion
001B  8A D8                 MOV  BL, AL         ; Save copy of BCD in BL
001D  80 E3 0F              AND  BL, 0FH        ; and mask
0020  24 F0                 AND  AL, 0F0H       ; Separate upper nibble
0022  B1 04                 MOV  CL, 04         ; Move upper BCD digit to low
0024  D2 C8                 ROR  AL, CL         ; nibble position for multiply
0026  B7 0A                 MOV  BH, 0AH        ; Load conversion factor in BH
0028  F6 E7                 MUL  BH             ; Multiply upper BCD digit in AL
                                                ; by 0AH in BH, leave result in AL
002A  02 C3                 ADD  AL, BL         ; Add lower BCD digit to MUL result
                ;End of conversion, binary result in AL
002C  59                   POP  CX             ; Restore registers
002D  5B                   POP  BX
002E  9D                   POPF
002F  C3                   RET                 ; and return to mainline
0030            BCD_BIN     ENDP

0030            CODE        ENDS
                            END     START
```

FIGURE 5-14 Example program passing parameters in registers.

```
1                                       ; 8086 PROGRAM F5-15.ASM
2                                ;ABSTRACT  : BCD to BINARY conversion program that uses a
3                                           ; procedure to convert BCD numbers to binary.
4                                           ; Program uses dedicated memory locations to
5                                           ; pass parameters to the procedure.
6                                ;REGISTERS : Uses CS, DS, SS, SP, AX
7                                ;PORTS     : None used
8                                ;PROCEDURES: Uses BCD_BIN

SAME DATA STRUCTURE AND INITIALIZATION AS FIGURE 5-14 LINES 9 THROUGH 27

28 000D  E8 0002                        CALL BCD_BIN                ; Do the conversion
29 0010  90                             NOP                         ; Continue with program here
30 0011  90                             NOP                         ;
31
32
33
34                               ;PROCEDURE:  BCD_BIN - Converts BCD numbers to binary.
35                               ;INPUT     :  Data from dedicated memory location BCD_INPUT
36                               ;OUTPUT    :  Data to dedicated memory location BIN_VALUE
37                               ;DESTROYS  :  Nothing
38
39 0012                          BCD_BIN    PROC    NEAR
40 0012  9C                             PUSHF               ; Save flags
41 0013  50                             PUSH AX             ; and registers
42 0014  53                             PUSH BX
43 0015  51                             PUSH CX
44 0016  A0 0000r                       MOV  AL, BCD_INPUT ; Get BCD value from memory
45                               ;Do the conversion
46 0019  8A D8                          MOV  BL, AL         ; Save copy of BCD in BL
47 001B  80 E3 0F                       AND  BL, 0FH        ; and mask
48 001E  24 F0                          AND  AL, 0F0H       ; Separate upper nibble
49 0020  B1 04                          MOV  CL, 04         ; Move upper BCD digit to low
50 0022  D2 C8                          ROR  AL, CL         ; nibble position for multiply
51 0024  B7 0A                          MOV  BH, 0AH        ; Load conversion factor in BH
52 0026  F6 E7                          MUL  BH             ; Multiply upper BCD digit in AL
53                                                          ; by 0AH in BH, leave result in AL
54 0028  02 C3                          ADD  AL, BL         ; Add lower BCD digit to MUL result
55                               ;End of conversion, binary value in AL
56 002A  A2 0001r                       MOV  BIN_VALUE, AL ; Store binary value in memory
57 002D  59                             POP  CX             ; Restore flags and
58 002E  5B                             POP  BX             ; registers
59 002F  58                             POP  AX
60 0030  9D                             POPF
61 0031  C3                             RET
62 0032                          BCD_BIN    ENDP
63
64 0032                          CODE       ENDS
65                                      END     START
```

FIGURE 5-15 Example program passing parameters in named memory locations.

The approach used in Figure 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. Figure 5-16, p. 112, 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

```
1                                       ; 8086 PROGRAM F5-16.ASM
2                                ;ABSTRACT  : BCD to BINARY conversion program that uses a
3                                       ; procedure to convert BCD numbers to binary.
4                                       ; Program shows how to use pointers to pass
5                                       ; parameters to a procedure.
6                                ;REGISTERS : Uses CS, DS, SS, SP, AX, SI, DI
7                                ;PORTS     : Uses none
8                                ;PROCEDURES: Uses BCD_BIN

SAME DATA STRUCTURE AND INITIALIZATION AS FIGURE 5-14 LINES 9 THROUGH 27

28                               ;Put pointer to BCD storage in SI and pointer to binary storage in DI
29 0000  BE 0000r                       MOV  SI, OFFSET BCD_INPUT   ; Create pointers to BCD and
30 0010  BF 0001r                       MOV  DI, OFFSET BIN_VALUE   ; binary storage
31 0013  E8 0001                        CALL BCD_BIN                ; Do the conversion
32 0016  90                             NOP                         ; Continue with program here
33
34                               ;PROCEDURE:  BCD_BIN - Converts BCD numbers to binary.
35                               ;INPUT     :  SI, points to location in memory of data
36                               ;OUTPUT    :  DI, points to location in memory for result
37                               ;DESTROYS  :  Nothing
38
39 0017                          BCD_BIN    PROC     NEAR
40 0017  9C                             PUSHF               ; Save flags
41 0018  50                             PUSH AX             ; and registers
42 0019  53                             PUSH BX
43 001A  51                             PUSH CX
44 001B  8A 04                          MOV  AL, [SI]       ; Get BCD value from memory
45                               ;Do the conversion
46 001D  8A D8                          MOV  BL, AL         ; Save copy of BCD in BL
47 001F  80 E3 0F                       AND  BL, 0FH        ; and mask
48 0022  24 F0                          AND  AL, 0F0H       ; Separate upper nibble
49 0024  B1 04                          MOV  CL, 04         ; Move upper BCD digit to low
50 0026  D2 C8                          ROR  AL, CL         ; nibble position for multiply
51 0028  B7 0A                          MOV  BH, 0AH        ; Load conversion factor in BH
52 002A  F6 E7                          MUL  BH             ; Multiply upper BCD digit in AL
53                                                          ; by 0AH in BH, leave result in AL
54 002C  02 C3                          ADD  AL, BL         ; Add lower BCD digit to MUL result
55                               ;End of conversion, binary value in AL
56 002E  88 05                          MOV  [DI], AL       ; Store binary value in memory
57 0030  59                             POP  CX             ; Restore flags and
58 0031  5B                             POP  BX             ; registers
59 0032  58                             POP  AX
60 0033  9D                             POPF
61 0034  C3                             RET
62 0035                          BCD_BIN    ENDP
63
64 0035                          CODE       ENDS
65                                      END      START
```

FIGURE 5-16 Example program passing parameters using pointers to named memory locations.

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, p. 114, 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 Figure 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, p. 115, 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.

An alternative to using the SP register is to use the BP register to access the parameters in the stack. Remember from Chapter 2 that an offset in the BP register will be added to the stack segment register to produce a physical memory address. This means that the BP register can easily be used as a second pointer to a location in the stack. Here's how we use it this way in our example program.

After pushing all the registers at the start of the procedure, we copy the contents of the stack pointer register to the BP register with the MOV BP,SP instruction. BP then points to the same location as the stack pointer. Then we use the MOV AX,[BP + 12] instruction to copy the desired word from the stack to AX. The 8086 will produce the effective address for this instruction by adding the displacement of 12, specified in the instruction, to the contents of the BP register. As you can see in Figure 5-18, the effective address produced by adding 12 to the contents of BP will be that of the desired parameter. Note that the MOV AX,[BP + 12] instruction does not change the contents of BP. BP can then be used to access other parameters on the stack by simply specifying a different displacement in the instruction used to access the parameter.

Once we have the BCD number copied from the stack into AL, the instructions which convert it to binary are the same as those in the previous versions. When we want to put the binary value back in the stack to return it to the calling program, we again use BP as a pointer to the stack. The instruction MOV [BP + 12],AX will copy AX to a stack location 12 addresses higher than that to which BP is pointing. This, of course, is the same location we used to pass the BCD number to the procedure. After we pop the registers and return to the calling program, the registers will all have the values they had before the CALL instruction executed. AX will contain the original BCD number, and the stack pointer will be pointing to the binary value, now at the top of the stack. In the mainline program we can now pop this hex value into a register with an instruction such as POP CX.

Whenever you are using the stack to pass parameters, it is very important to keep track of what you have pushed on the stack and where the stack pointer is at each point in a program. We have found that diagrams such as the one in Figure 5-18 are very helpful in doing this. One potential problem to watch for when using the stack to pass parameters is *stack overflow*. Stack overflow means that the stack fills up and overflows the memory space you set aside for it. To see how this can easily happen if you don't watch for it, consider the following. Suppose that we use the stack to pass four word parameters to a procedure, but that we pass only one word parameter back to the calling program on the stack. Figure 5-19, p. 115, shows a stack diagram for this situation. Before a CALL instruction, the four parameters to be passed to the procedure are pushed on the stack. During the procedure, the parameter to be returned is put in the stack location previously occupied by the fourth input parameter. After the RET instruction at the end of the procedure executes, the stack pointer will be left pointing at this value. Now assume that we pop this value into a register. The POP instruction will copy the value to a register and increment the stack pointer by 2. The stack pointer now points to the third word we pushed to pass to the procedure. In other words the stack pointer is six addresses lower than it was when we started this process. Now suppose that we call this procedure many times in the course of the mainline program. Each time we push four words on the stack but only pop one word off, the stack pointer will be left six addresses lower than it was before the process. The top of the stack will keep moving downward. When the

```
1                                       ; 8086 PROGRAM F5-17.ASM
2                               ;ABSTRACT  : BCD to BINARY conversion program that uses a
3                                       ; procedure to convert BCD numbers to binary.
4.                                      ; Program shows how to use the stack to pass
5                                       ; parameters to a procedure.
6                               ;REGISTERS : Uses CS, DS, SS, SP, AX
7                               ;PORTS     : Uses none
8                               ;PROCEDURES: Uses BCD_BIN

SAME DATA STRUCTURE AND INITIALIZATION AS FIGURE 5-14 LINES 9 THROUGH 27

28 000D  A0 0000r                       MOV  AL, BCD_INPUT          ; Move BCD value into AL
29 0010  50                             PUSH AX                     ; and push it onto onto stack
30 0011  E8 0005                        CALL BCD_BIN                ; Do the conversion
31 0014  58                             POP  AX                     ; Get the binary value
32 0015  A2 0001r                       MOV  BIN_VALUE, AL          ; and save it
33 0018  90                             NOP                         ; Continue with program here
34
35                              ;PROCEDURE:  BCD_BIN - Converts BCD numbers to binary.
36                              ;INPUT     :  None - BCD value assumed to be on stack before call
37                              ;OUTPUT    :  None - Binary value on top of stack after return
38                              ;DESTROYS  :  Nothing
39
40 0019                         BCD_BIN     PROC    NEAR
41 0019  9C                             PUSHF               ; Save flags
42 001A  50                             PUSH AX             ; and registers
43 001B  53                             PUSH BX
44 001C  51                             PUSH CX
45 001D  55                             PUSH BP
46 001E  8B EC                          MOV  BP, SP         ; Make a copy of the stack pointer
47 0020  8B 46 0C                       MOV  AX, [BP+12]    ; Get BCD number from stack
48                              ;Do the conversion
49 0023  8A D8                          MOV  BL, AL         ; Save copy of BCD in BL
50 0025  80 E3 0F                       AND  BL, 0FH        ; and mask
51 0028  24 F0                          AND  AL, 0F0H       ; Separate upper nibble
52 002A  B1 04                          MOV  CL, 04         ; Move upper BCD digit to low
53 002C  D2 C8                          ROR  AL, CL         ; nibble position for multiply
54 002E  B7 0A                          MOV  BH, 0AH        ; Load conversion factor in BH
55 0030  F6 E7                          MUL  BH             ; Multiply upper BCD digit in AL
56                                                          ; by 0AH in BH, leave result in AL
57 0032  02 C3                          ADD  AL, BL         ; Add lower BCD digit to MUL result
58                              ;End of conversion, binary value in AL
59 0034  89 46 0C                       MOV  [BP+12], AX    ; Put binary value on stack
60 0037  5D                             POP  BP             ; Restore flags and
61 0038  59                             POP  CX             ; registers
62 0039  5B                             POP  BX
63 003A  58                             POP  AX
64 003B  9D                             POPF
65 003C  C3                             RET
66 003D                         BCD_BIN     ENDP
67
68 003D                         CODE        ENDS
69                                          END     START
```

FIGURE 5-17 Example program passing parameters on the stack.

stack pointer gets down to 0000H, the next push will roll it around to FFFEH and write a word at the very top of the 64-Kbyte stack segment. If you overlapped segments as you usually do in a small system, the word may get written in a memory location that you are using for data or your program code, and your data or code will be lost! This is what we mean by the term *stack overflow*.

The cure for this potential problem is to use stack diagrams to help you keep the stack balanced. You need to keep the number of pops equal to the number of pushes or in some other way make sure the stack pointer gets back to its initial location.

For this example, we could use an ADD SP,06H instruction after the POP instruction to get the stack pointer back up the additional six addresses to where it

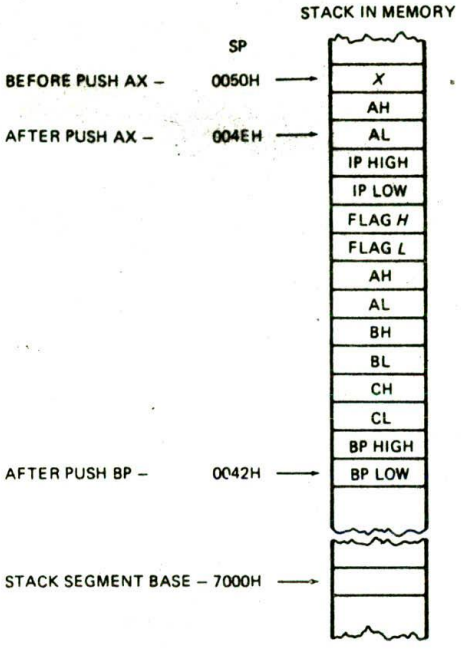


FIGURE 5-18 Stack diagram for program in Figure 5-17.

was before we pushed the four parameters onto the stack.

For other cases such as this, the 8086 RET instruction has two forms which help you to keep the stack balanced. Remember from a previous section of this chapter that the 8086 has four forms of the RET instruction. The regular near RET instruction copies the return address from the stack to the instruction pointer and increments the stack pointer by 2. The regular far RET instruction copies the return IP and CS values from the stack to IP 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.

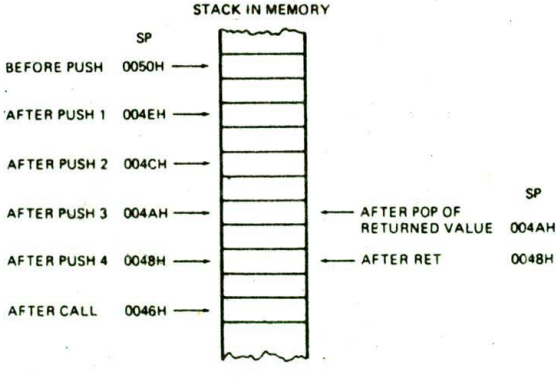


FIGURE 5-19 Stack diagram showing cause of stack overflow.

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 Figure 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.

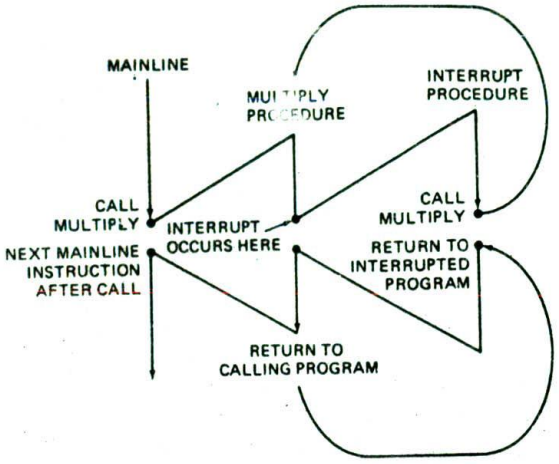


FIGURE 5-20 Program execution flow for reentrant procedure.

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 carefully. You will seldom have to write a recursive procedure, so the main points to look for in that section are the definition of the term and the operation of the stack as a recursive procedure operates.

REENTRANT PROCEDURES

The 8086 has a signal input which allows a signal from some external device to interrupt the normal program execution sequence and call a specified procedure. In our electronics factory, for example, a temperature sensor in a flow-solder machine could be connected to the interrupt input. If the temperature gets too high, the sensor sends an interrupting signal to the 8086. The 8086 will then stop whatever it is doing and go to a procedure which takes whatever steps are necessary to cool down the solder bath. This procedure is called an *interrupt service procedure*. Chapter 8 discusses 8086 interrupts and interrupt service procedures in great detail, but it is appropriate to introduce the concept here.

Now, suppose that the 8086 was in the middle of executing a multiply procedure when the interrupt signal occurred, and that we also need to use the multiply procedure in the interrupt service subroutine. Figure 5-20 shows the program execution flow we want for this situation. When the interrupt occurs, execution goes to the interrupt service procedure. The interrupt service procedure then calls the multiply procedure when it needs it. The RET instruction at the end of the multiply procedure returns execution to the interrupt service procedure. A special return instruction at the end of the interrupt service procedure returns execution to the multiply procedure where it was executing when the interrupt occurred.

In order for the program flow in Figure 5-20 to work correctly, the multiply procedure must be written in such a way that it can be interrupted, used, and "reentered" without losing or writing over anything. A procedure which can function in this way is said to be *reentrant*.

To be reentrant, a procedure must first of all push the flags and all registers used in the procedure. Also, to be reentrant, a program should use only registers or the stack to pass parameters. To see why this second point is necessary, let's take another look at the program in Figure 5-15. This program uses the named variables BCD_INPUT and BIN_VALUE. The procedure BCD_BIN accesses these two directly by name.

Now, suppose that the 8086 is in the middle of executing the BCD_BIN procedure and an interrupt occurs. Further suppose that the interrupt service procedure loads some new value in the memory location named BCD_INPUT, and calls the BCD_BIN procedure again. The initial value in BCD_INPUT has now been written over. If the interrupt occurred before the first execution of the procedure had a chance to read this value in, the value will be lost forever. When execution returns to BCD_BIN after the interrupt service procedure, the value used for BCD_INPUT will be that put there by the interrupt service routine instead of the desired initial value. There are several ways we can handle the parameters so that the procedure BCD_BIN is reentrant.

The first is to simply pass the parameters in registers, as we did in the program in Figure 5-14. If the interrupt procedure and the BCD_BIN procedure each push and pop all the registers they use, all the parameters from the interrupted execution will be saved and restored. When execution returns to BCD_BIN again, the registers will contain the same data they did when the interrupt occurred. The interrupted execution will then complete correctly.

A second method of making the BCD_BIN procedure reentrant is to pass pointers to the data items in registers, as we did in the program in Figure 5-16.

Again, if the interrupt procedure and the BCD_BIN procedure each push and pop the registers they use, execution will return to the interrupted procedure with data intact.

The third way to make the BCD_BIN procedure reentrant is by passing parameters or pointers on the stack, as we did in the version in Figure 5-17. In this version, the mainline program pushes the BCD number onto the stack and then calls the procedure. The procedure pushes registers on the stack and uses BP to access the BCD number relative to where the stack pointer ended up. If an interrupt occurs, the interrupt service procedure will push on the stack the BCD number it wishes to convert and call BCD_BIN. This second BCD number will be pushed on the stack at a different location from the first BCD number that was pushed.

The BCD_BIN procedure will use BP to access the new BCD value and pass the binary value back on the stack. If the BCD_BIN and interrupt procedure each save and restore the registers they use, the first execution of the procedure will produce correct results when it is reentered.

If you are writing a procedure that you may want to call from a program written in a high-level language such as Pascal or C, then you should definitely use the stack for passing parameters because that is how these languages do it. In a later chapter we show you how to pass parameters between C programs and assembly language programs.

RECURSIVE PROCEDURES

A *recursive procedure* is a procedure which calls itself. This seems simple enough, but the question you may be thinking is, "Why would we want a procedure to call itself?" The answer is that certain types of problems, such as choosing the next move in a computer chess program, can best be solved with a recursive procedure. Recursive procedures are often used to work with complex data structures called *trees*.

We usually write recursive procedures in a high-level language such as C or Pascal, except in those cases where we need the speed gained by writing in assembly language. However, the assembly language example in the following sections should help you understand how recursion works and how the stack is used by recursive and other nested procedures.

Recursive Procedure Example

ALGORITHM

Most of the examples of recursive procedures that we could think of are too complex to show here. Therefore, we have chosen a simple problem which could be solved without recursion.

The problem we have chosen to solve is to compute the factorial of a given number in the range of 1 to 8. The factorial of a number is the product of the number and all the positive integers less than the number. For example, 5 factorial is equal to 5 × 4 × 3 × 2 × 1. The word "factorial" is often represented with "!." For example, 5! is another way to represent 5 factorial.

What we want here is a recursive procedure which will compute the factorial of a number N which we pass to it on the stack, then pass the factorial back to the calling program on the stack. The basic algorithm can be expressed very simply as

IF N = 1 THEN factorial = 1,
  ELSE factorial = N × (factorial of N − 1)

This says that if the number we pass to the procedure is 1, the procedure should return the factorial of 1, which is 1. If the number we pass is not 1, then the procedure should multiply this number by the factorial of the number minus 1.

Now here's where the recursion comes in. Suppose we pass a 3 to the procedure. When the procedure is first called, it has the value of 3 for N, but it does not have the value for the factorial of N − 1 that it needs to do the multiplication indicated in the algorithm. The procedure solves this problem by calling itself to compute the needed factorial of N − 1. It calls itself over and over until the factorial of N − 1 that it has to compute is the factorial of 1.

Figure 5-21, p. 118, shows several ways in which we can represent this process. In the program flow diagram in Figure 5-21*a*, you can see that if the value of N passed to the procedure is 1, then the procedure simply loads 1 into the stack location reserved for N! and returns to the calling program. Figure 5-21*b* shows the program flow that will occur when the number passed to the procedure is some number other than 1. If we call the procedure with N = 3, the procedure will call itself to compute (N − 1)! or 2!. It will then call itself again to compute the value of the next (N − 1)! or 1!. Since 1! = 1, the procedure will return this value to the program that called it. In this case the program that called it was a previous execution of the same procedure that needed this value to compute 2!. Given this value, it will compute 2! and return the value to the program that called it. Here again, the program that called it was a previous execution of the same procedure that needed 2! to compute the factorial of 3. Given the factorial of 2, this call of the procedure can now compute 3! and return to the program that called it. For the example here, the return now will be to the mainline program.

Figure 5-21*c* shows how we can represent this algorithm in slightly expanded pseudocode. Use the program flow diagram in Figure 5-21*b* 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.

Figure 5-21*d* shows a flowchart for this algorithm. Note that the flowchart shows the same ambiguity about where the return operations send execution to.

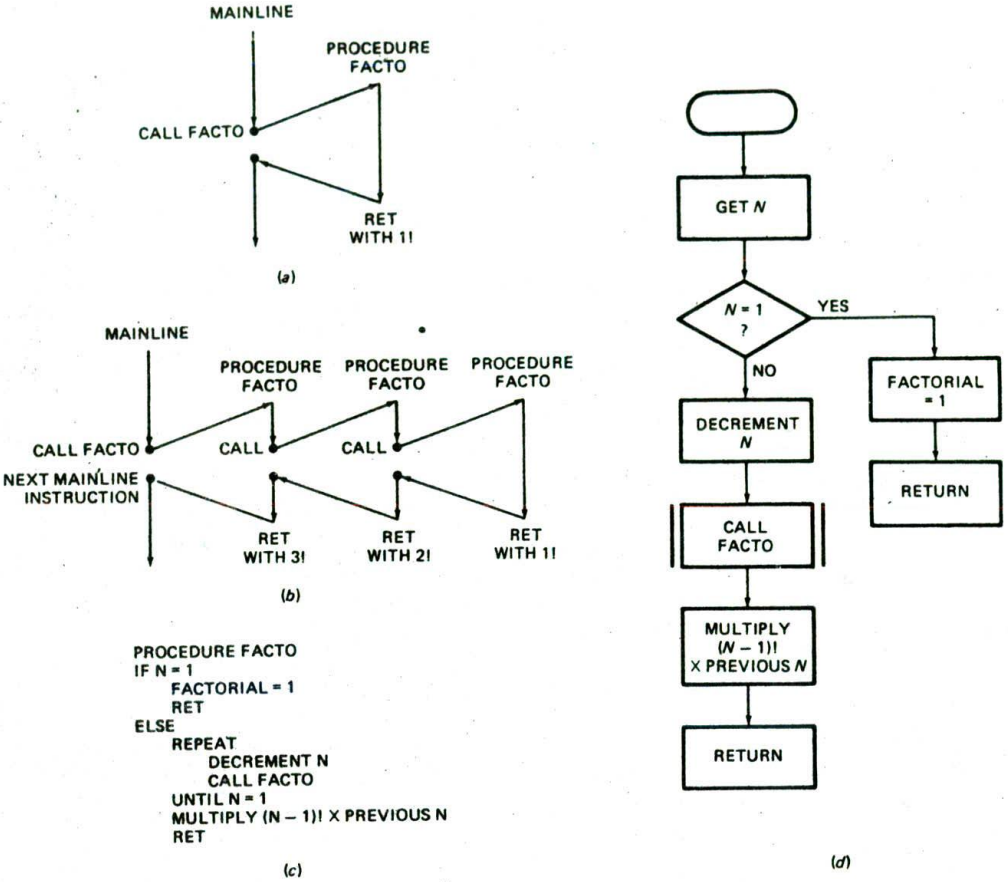


FIGURE 5-21 Algorithm for program to compute factorial for a number N between 1 and 8. (*a*) Flow diagram for N = 1. (*b*) Flow diagram for N = 3. (*c*) Pseudocode. (*d*) Flowchart.

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. Figure 5-23, p. 120, shows, with a stack diagram, how the stack will be affected if this procedure is called with N = 3. When working your way through a recursive procedure or any procedure which uses the stack extensively, a stack diagram such as this is absolutely necessary to keep track of everything.

The first parts of the program are housekeeping chores. We start the mainline program by declaring a stack segment and setting aside a stack of 200 words with a label at the top of the stack. The first three instructions in the code segment of the mainline program initialize the stack segment register and the stack pointer register. The SUB SP,04 instruction after this will decrement the stack pointer register by 4. In other words, we skip the stack pointer down over 2 words in the stack. These two word locations will be used to pass the computed factorial from the procedure back to the mainline program. Next we load the number whose factorial we want into AX and push the value on the stack where the procedure will access it. Now we are ready to call the procedure. The procedure is near because it is in the same code segment as the instruction which calls it.

At the start of the procedure, we save the flags and all the registers used in the procedure on the stack. Let's take a look at Figure 5-23 to see what is on the stack at this point. As you can see, the stack now has the space for the result, the passed value, the return address, and the pushed registers. Unfortunately, the value of N is buried 10 addresses up the stack from where the stack pointer was left after BP was pushed. To access this buried value, we first copy SP to BP with the MOV BP,SP instruction so that BP points to the top of the stack. Then we use the MOV AX,[BP + 10] instruction to copy

```
                                ; 8086 PROGRAM F5-22.ASM
                              ;ABSTRACT  : Program computes the factorial of a number between 1 and 8
                              ;REGISTERS : Uses CS, SS, SP, AX, DX
                              ;PORTS     : None used
                              ;PROCEDURES: Uses FACTO

0000                          STACK_SEG   SEGMENT     STACK
0000  C8*(0000)                           DW      200 DUP(0)      ; Set aside 200 words for stack
                              STACK_TOP   LABEL   WORD            ; Assign name to word above stack top
0190                          STACK_SEG   ENDS

                                NUMBER      EQU     08            ; 8! = 40320 = 9D80H

0000                          CODE      SEGMENT
                                        ASSUME CS:CODE, SS:STACK_SEG
0000  B8 0000s                START:    MOV  AX, STACK_SEG          ; Initialize stack segment register
0003  8E D0                             MOV  SS, AX
0005  BC 0190r                          MOV  SP, OFFSET STACK_TOP   ; Initialize stack pointer
0008  83 EC 04                          SUB  SP, 0004H              ; Make space in stack for factorial
000B  B8 0008                           MOV  AX, NUMBER             ; to be returned and put number
000E  50                                PUSH AX                     ; to be passed on stack
000F  E8 0009                           CALL FACTO                  ; Compute factorial of number
0012  83 C4 02                          ADD  SP, 2                  ; Get over original number in stack
0015  58                                POP  AX                     ; Get low word of the result
0016  5A                                POP  DX                     ; Get high word of the result
0017  90                                NOP                         ; Simulate next mainline instruction
0018  EB 3A 90                         JMP  FIN                    ; Or EXIT program

                              ;PROCEDURE: FACTO: Recursive procedure that computes the factorial of a number
                              ;INPUT     : Takes data (number = N) from the stack
                              ;OUTPUT    : Returns with result on stack above original data
                              ;DESTROYS  : Nothing

001B                          FACTO     PROC      NEAR
001B  9C                                PUSHF                       ; Save flags and registers
001C  50                                PUSH AX                     ; on the stack
001D  52                                PUSH DX
001E  55                                PUSH BP
001F  8B EC                             MOV  BP, SP                 ; Point BP at top of stack
0021  8B 46 0A                          MOV  AX,[BP+10]             ; Copy number from stack to AX
0024  3D 0001                           CMP  AX, 0001H              ; If N not = 1 THEN
0027  75 0D                             JNE  GO_ON                  ;  compute factorial
0029  C7 46 0C 0001                     MOV  WORD PTR [BP+12], 0001H ; ELSE load 1! on stack
002E  C7 46 0E 0000                     MOV  WORD PTR [BP+14], 0000H ; and return to calling program
0033  EB 1A 90                         JMP  EXIT
0036  83 EC 04                GO_ON:   SUB  SP, 0004H              ; Make space in stack for
                                                                    ;  preliminary factorial
0039  48                                DEC  AX                     ; Decrement number now in AX
003A  50                                PUSH AX                     ; Save N-1 on stack
003B  E8 FFDD                           CALL FACTO                  ; Compute factorial of N-1
003E  8B EC                             MOV  BP, SP                 ; Point BP at top of stack
0040  8B 46 02                          MOV  AX, [BP+2]             ; Last (N-1)! from stack to AX
0043  F7 66 10                          MUL  WORD PTR [BP+16]       ; Multiply by previous N
0046  89 46 12                          MOV  [BP+18], AX            ; Copy new factorial to stack
0049  89 56 14                          MOV  [BP+20], DX
004C  83 C4 06                          ADD  SP, 0006H              ; Point SP at pushed register
004F  5D                       EXIT:    POP  BP                     ; Restore registers
0050  5A                                POP  DX
0051  58                                POP  AX
0052  9D                                POPF
0053  C3                                RET
0054                           FACTO     ENDP
0054  90                       FIN:     NOP
0055                           CODE      ENDS
                                        END     START
```

FIGURE 5-22 Program which uses a recursive procedure to calculate the factorial of a number between 1 and 8.

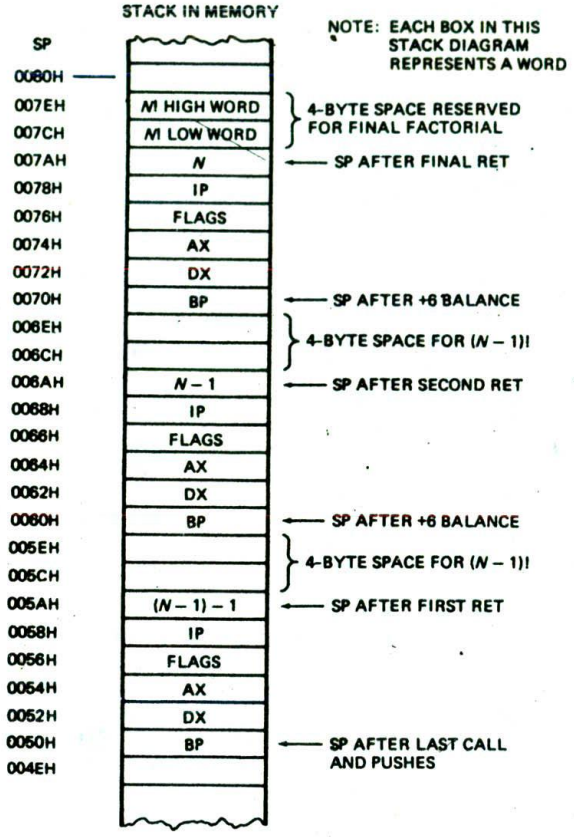


FIGURE 5-23 Stack diagram for program in Figure 5-22 showing contents of stack for N = 3.

N from the stack to AX. Now that the procedure has the value of N, let's work through how it gets processed.

If the value of N read in is 1, then the factorial is 1. We want to put 00000001H in the stack locations we reserved for the result, restore the registers, and return to the mainline program. Follow this path through the program in Figure 5-22. Note how the MOV WORD PTR [BP + 12],0001H instruction is used to load a value to a location buried in the stack. The WORD PTR directives tell the assembler that you want to move a word to the specified memory location. Without these directives, the assembler will not know whether to code the instruction for moving a byte or for moving a word. The MOV WORD PTR [BP + 14],0000H instruction is likewise used to move a word value to the stack location reserved for the high word of the factorial.

Now let's see what happens if the number passed to FACTO is a 3. The CMP AX,0001H instruction and the JNE GO_ON instructions determine that N is not 1 and send execution to the SUB SP,04H instruction. According to the algorithm, we are going to find the value of N! by multiplying N times the value of (N – 1)!. We will be calling FACTO again to find the value of (N – 1)!. The SUB SP,04H instruction skips the stack pointer down over four addresses in the stack to offset 006CH for our example. The value of (3 – 1)! will be returned in these locations.

The next step in the program is to decrement N by 1 and push the value of N – 1 on the stack at offset 006AH, where it can be accessed during the next call of FACTO.

Next we call FACTO again to compute the value of (N – 1)!. The IP flags and registers will again be pushed on the stack. As shown in Figure 5-23, the stack pointer is now pointing at offset 0060H, and the value of N – 1 that we need is again buried 10 addresses up in the stack. This is no problem, because the MOV BP,SP and MOV AX,[BP + 10] instructions will allow us to access the value. We started with N = 3 for this example, so the value of N – 1 that we read in at this point is equal to 2. Since this value is not 1, execution will again go to the label GO_ON. The SUB SP,04 instruction will again skip the stack pointer down over four addresses to offset 005CH. This leaves space for (2 – 1)!, which will be returned by the next call of FACTO. We decrement N – 1 by 1 to give a result of 1 and then push this value on the stack at offset 005AH. We then call FACTO to compute the factorial of 1.

After calling FACTO again and pushing all the registers on the stack, the stack pointer now points to offset 0050H. FACTO then reads N = 1 from the stack with the MOV AX,[BP + 10] instruction. When the CMP AX,0001H instruction in FACTO finds that the number passed to it is 1, FACTO loads a factorial value of 1 into the four memory locations we most recently set aside for a returned factorial at offsets 005CH to 005FH. The MOV WORD PTR [BP + 12],0001 and MOV WORD PTR [BP + 14],0000 instructions do this. Since N was a 1, execution will go to the EXIT label. The registers will then be popped and execution returned to the next instruction after the CALL instruction that last called FACTO.

Now in this case FACTO was called from a previous execution of FACTO, so the return will be to the MOV BP,SP instruction after CALL FACTO. The MOV BP,SP instruction points BP at the top of the stack at 005AH, so that we can access data on the stack without affecting the stack pointer. The MOV AX,[BP + 2] instruction after this copies the low word of (N – 1 – 1)! or 1 from the stack to AX so that we can multiply it by N – 1. We need only the lower word of the two we set aside for the factorial, because for an N of 8 or less, only the lower word will contain data. Restricting the allowed range of N for this example means that we only have to do a 16-bit by 16-bit multiplication. We could increase the allowed range of N by simply setting aside larger spaces in the stack for factorials and including instructions to multiply larger numbers.

In this example, the MUL WORD PTR [BP + 16] instruction multiplies the (N – 1 – 1)! in AX by the previous N from the stack. The low word of the product is left in AX, and the high word of the product is left in DX. The MOV [BP + 18],AX and the MOV [BP + 20],DX instructions copy these two words to the stack locations we reserved for the next factorial result at offsets 006CH to 006FH.

The next operation we would like to do in the program is pop the registers and return. As you can see from Figure 5-23, however, the stack pointer is now pointing at some old data on the stack at offset 005AH, not at the first register we want to pop. To get the stack pointer pointing where we want it, we add 6 to it with the ADD SP,06H instruction. Then we pop the registers and return.

After the pops and return, the stack pointer will be pointing at N − 1 at offset 006AH, and the value for 2! will be in the stack at offsets 006AH to 006FH in the stack. We still have one more computation to produce the desired 3!. Therefore, the return is again to the MOV BP,SP instruction after CALL in FACTO. The instructions after this will multiply 2! times 3 to produce the desired 3!, and copy 3! to the stack as described in the preceding paragraph. The ADD SP,06H instruction will again adjust the stack pointer so that we can pop the registers and return. Since we have done all the required computations, this time the return will be to the mainline program. The desired result, 3!, will be in the memory locations we reserved for it in the stack at offsets 007AH to 007FH.

After the final return, the stack pointer will be pointing at offset 007AH in the stack. We add 2 to the stack pointer so that it points to the factorial result and pop the result into the DX and AX registers. This brings the stack pointer back to its initial value.

If you work your way through the flow of the stack and the stack pointer in this example program, you should have a good understanding of how the stack functions during nested procedures.

Writing and Calling Far Procedures

INTRODUCTION AND OVERVIEW

A *far procedure* is one that is located in a segment which has a different name from the segment containing the CALL instruction. To get to the starting address of a far procedure, the 8086 must change the contents of both the code segment register and the instruction pointer.

```
CODE  SEGMENT
      ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
      :
      :
      CALL MULTIPLY_32
      :
CODE  ENDS

PROCEDURES SEGMENT
  MULTIPLY_32 PROC FAR
      ASSUME CS:PROCEDURES
      :
      :
  MULTIPLY_32 ENDP
PROCEDURES    ENDS
```

FIGURE 5-24 Program additions needed for a far procedure.

Therefore, if you are hand coding a program which calls a far procedure, make sure to use one of the intersegment forms of the CALL instruction shown in Figure 5-6. Likewise, at the end of a far procedure, both the contents of the code segment register and the contents of the instruction pointer must be popped off the stack to return to the calling program, so make sure to use one of the intersegment forms of the RET instruction to do this.

If you are using an assembler to assemble a program containing a far procedure, there are a few additional directives you have to give the assembler. The following sections show you how to put these needed additions into your programs. The first case we will describe is one in which the procedure is in the same assembly module, but it is in a segment with a different name from the segment that contains the CALL instruction.

ACCESSING A PROCEDURE IN ANOTHER SEGMENT

Suppose that in a program you want to put all of the mainline program in one logical segment and you want to put several procedures in another logical segment to keep them separate from the mainline program. Figure 5-24 shows some program fragments which illustrate this situation. For this example, our mainline instructions are in a segment named CODE. A procedure called MULTIPLY_32 is in a segment named PROCEDURES. Since the procedure is in a different segment from the CALL instruction, the 8086 must change the contents of the code segment register to access it. Therefore, the procedure is far.

You let the assembler know that the procedure is far by using the word FAR in the MULTIPLY_32 PROC FAR statement. When the assembler finds that the procedure is declared as far, it will automatically code the CALL instruction as an intersegment call and the RET instruction as an intersegment return.

Now the remaining thing you have to do, so that the program gets assembled correctly, is to make sure that the assembler uses the right code segment for each part of the program. You use the ASSUME directive to do this. At the start of the mainline program, you use the statement ASSUME CS:CODE to tell the assembler to compute the offsets of the following instructions from the segment base named CODE. At the start of the procedure, you use the ASSUME CS:PROCEDURES statement to tell the assembler to compute the offsets for the instructions in the procedure starting from the segment base named PROCEDURES.

When the assembler finally codes the CALL instruction, it will put the value of PROCEDURES in for CS in the instruction. It will put the offset of the first instruction of the procedure in PROCEDURES as the IP value in the instruction.

To summarize, then, if a procedure is in a different segment from the CALL instruction, you must declare it far with the FAR directive. Also, you must put an ASSUME statement in the procedure to tell the assembler what segment base to use when calculating the offsets of the instructions in the procedure.

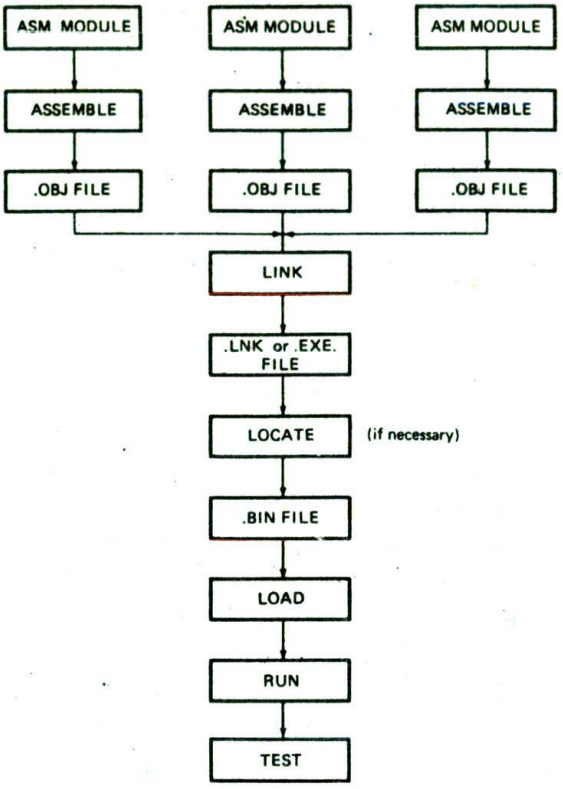


FIGURE 5-25 Chart showing the steps needed to run a program that has been written in modular form.

ACCESSING A PROCEDURE AND DATA IN A SEPARATE ASSEMBLY MODULE

As we have discussed previously, the best way to write a large program is to divide it into a series of modules. Each module can be individually written, assembled, tested, and debugged as shown in Figure 5-25. The object code files for the modules can then be linked together. Finally, the resulting link file can be located, run, and tested.

As we said earlier in this chapter, the individual modules of a large program are often written as procedures and called from a mainline or executive program. In the preceding section we showed you how to access a procedure in a different segment from the CALL instruction. Here we show you how to access a procedure or data in a different assembly module.

In order for a linker to be able to access data or a procedure in another assembly module correctly, there are two directives that you must use in your modules. We will give you an overview of these two and then show with an example how they are used in a program.

1. In the module where a variable or procedure is declared, you must use the PUBLIC directive to let the linker know that the variable or procedure can 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.
2. 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 5-26*a* 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 Figure 5-26*b*, 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

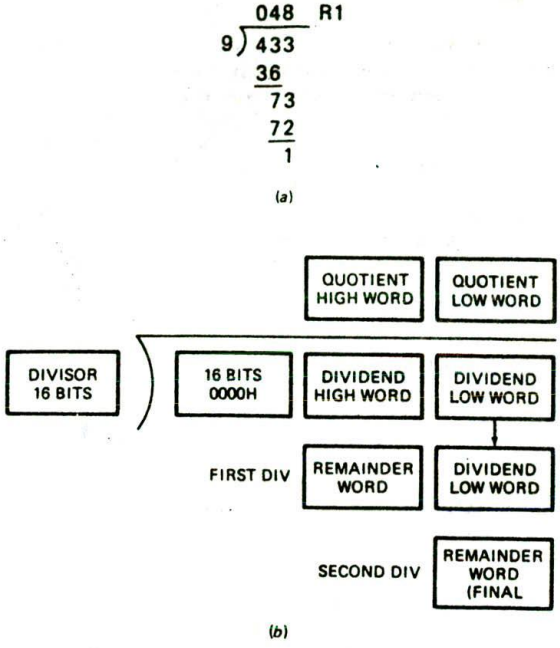


FIGURE 5-26 Algorithm for smart divide procedure. (*a*) Decimal analogy. (*b*) 8086 approach.

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.

THE ASSEMBLY LANGUAGE PROGRAM

Figure 5-27*a*, pp. 124-5, shows the mainline of a program which calls the procedure shown in Figure 5-27*b*, p. 126, which implements our division algorithm. We wrote these two as separate assembly modules to show you how to add PUBLIC and EXTRN statements so that the modules are linkable. Let's look closely at these added parts before we discuss the actual division procedure.

The first added part of the program to look at is in the statement DATA SEGMENT WORD PUBLIC. The word PUBLIC in this statement tells the linker that this segment can be combined (concatenated) with segment(s) that have the same name but are located in other modules. In other words, if two or more assembly modules have PUBLIC segments named DATA, their contents will be pulled together in successive memory locations when the program modules are linked. You should then declare a segment PUBLIC anytime you want it to be linked with other segments of the same name in other modules.

The next addition to look at is the statement PUBLIC DIVISOR in the mainline module in Figure 5-27*a*. This statement is necessary to tell the assembler and the linker that it is legal for the data item named DIVISOR to be accessed from other assembly modules. Essentially what we are doing here is telling the assembler to put the offset of DIVISOR in a special table where it can be accessed when the program modules are linked. Whenever you want a named data item or a label to be accessible from another assembly module, you must declare it as PUBLIC

The other side of this coin is that, when you need to access a label, procedure, or variable in another module, you must use the EXTRN directive to tell the assembler that the label or data item is not in the present module. If you don't do this, the assembler will give you an error message because it can't find the label or variable in the current module. In the example program, the statement EXTRN SMART_DIVIDE:FAR tells the assembler that we will be accessing a label or procedure of type FAR in some other assembly module. For this example, we will be accessing our procedure, SMART_DIVIDE. We enclose the EXTRN statement with the PROCEDURES SEGMENT PUBLIC and the PROCEDURES ENDS statements to tell the assembler and linker that the procedure SMART_DIVIDE is located in the segment PROCEDURES. There are some cases in which these statements are not needed, but we have found that bracketing the EXTRN statement with SEGMENT-ENDS directives in this way is the best way to make sure that the linker can find everything when it links modules. As you can see in the table at the end of the assembler listing in Figure 5-27*a*, SMART_DIVIDE is identified as an external label of type FAR, found in a segment named PROCEDURES.

Now let's see how we handle EXTRN and PUBLIC in the procedure module in Figure 5-27*b*. The procedure accesses the data item named DIVISOR, which is defined in the mainline module. Therefore, we must use the statement EXTRN DIVISOR:WORD to tell the assembler that DIVISOR, a data item of type word, will be found in some other module. Furthermore, we enclose the EXTRN statement with the DATA SEGMENT PUBLIC and DATA ENDS statements to tell the assembler that DIVISOR will be found in a segment named DATA.

The procedure SMART_DIVIDE must be accessible from other modules, so we declare it public with the PUBLIC SMART_DIVIDE statement in the procedure module. If we needed to make other labels or data items public, we could have listed them separated by commas after PUBLIC SMART_DIVIDE. An example is PUBLIC SMART_DIVIDE, EXIT.

NOTES:

1. If we had needed to access DIVIDEND also, we could have written the EXTRN statement as EXTRN DIVISOR:WORD,DIVIDEND:WORD. To add more terms, just separate them with a comma.

2. Constants defined with an EQU directive in one module can be imported to another module by identifying them as EXTRN of type ABS. For example, if you declare CORRECTION_FAC-

```
Turbo Assembler  Version 1.0       05-05-89 13:09:06        Page 1

                                    ; 8086 PROGRAM F5-27A.ASM
                                ;ABSTRACT  : Program divides a 32-bit number by a 16-bit number
                                           ; to give a 32-bit quotient and a 16-bit remainder.
                                ;REGISTERS : Uses CS, DS, SS, AX, SP, BX, CX
                                ;PORTS     : None used
                                ;PROCEDURES: Far procedure SMART_DIVIDE

0000                            DATA        SEGMENT     WORD  PUBLIC
0000  403B 8C72                             DIVIDEND    DW    403BH, 8C72H   ; Dividend = 8C72403BH
0004  5692                                  DIVISOR     DW    5692H          ; 16-bit divisor
0006                            DATA        ENDS

0000                            MORE_DATA   SEGMENT     WORD
0000  02*(0000)                             QUOTIENT    DW     2 DUP(0)
0004  0000                                  REMAINDER   DW     0
0006                            MORE_DATA   ENDS

0000                            STACK_SEG   SEGMENT     STACK
0000  64*(0000)                             DW          100 DUP(0)     ; Stack of 100 words
                                            TOP_STACK   LABEL  WORD    ; Name pointer to top of stack
00C8                            STACK_SEG   ENDS

                                PUBLIC      DIVISOR

0000                            PROCEDURES SEGMENT      PUBLIC     ; Let assembler know that SMART_DIVIDE
                                     EXTRN SMART_DIVIDE : FAR      ; is a label of type FAR and is located
0000                            PROCEDURES ENDS                    ; in the segment PROCEDURES

0000                            CODE        SEGMENT    WORD     PUBLIC
                                            ASSUME     CS:CODE, DS:DATA, SS:STACK_SEG
0000  B8 0000s                  START:      MOV  AX, DATA             ; Initialize data segment
0003  8E D8                                 MOV  DS, AX               ; register
0005  B8 0000s                              MOV  AX, STACK_SEG        ; Initialize stack segment
0008  8E D0                                 MOV  SS, AX               ; register
000A  BC 00C8r                              MOV  SP, OFFSET TOP_STACK ; Initialize stack pointer
000D  A1 0000r                              MOV  AX, DIVIDEND         ; Load low  word of dividend
0010  8B 16 0002r                          MOV  DX, DIVIDEND + 2     ; Load high word of dividend
0014  8B 0E 0004r                          MOV  CX, DIVISOR          ; Load divisor
0018  9A 00000000se                        CALL SMART_DIVIDE         ; Quotient returned in DX:AX
                                ; Remainder returned in CX, carry set if result invalid
001D  73 03                                 JNC  SAVE_ALL             ; IF carry = 0, result valid
001F  EB 13 90                              JMP  STOP                 ; ELSE carry set, don't save result
                                  ASSUME DS:MORE_DATA                 ; Change data segment
0022  1E                        SAVE_ALL:   PUSH DS                   ; Save old DS
0023  B8 0000s                             MOV  BX, MORE_DATA        ; Load new data segment
0026  8E D8                                 MOV  DS, BX               ; register
0028  A3 0000r                              MOV  QUOTIENT, AX         ; Store low  word of quotient
002B  89 16 0002r                           MOV  QUOTIENT + 2, DX     ; Store high word of quotient
002F  89 0E 0004r                           MOV  REMAINDER, CX        ; Store remainder
                                  ASSUME DS:DATA
0033  1F                                    POP  DS                   ; Restore initial DS
0034  90                        STOP:       NOP
0035                            CODE        ENDS
                                            END     START
```

FIGURE 5-27 Assembly language program to divide a 32-bit number by a 16-bit number and return a 32-bit quotient. (*a*) Mainline program module (continued on p. 125). (*b*) Procedure module (p. 126).

```
Turbo Assembler  Version 1.0        05-05-89 13:09:06   Page 2
Symbol Table

Symbol Name                        Type   Value

??DATE                             Text   "05-05-89"
??FILENAME                         Text   "F5-27A  "
??TIME                             Text   "13:09:05"
??VERSION                          Number 0100
@CPU                               Text   0101H
@CURSEG                            Text   CODE
@FILENAME                          Text   F5-27A
@WORDSIZE                          Text   2
DIVIDEND                           Word   DATA:0000
DIVISOR                            Word   DATA:0004
QUOTIENT                           Word   MORE_DATA:0000
REMAINDER                          Word   MORE_DATA:0004
SAVE_ALL                           Near   CODE:0022
SMART_DIVIDE                       Far    PROCEDURES:---- Extern
START                              Near   CODE:0000
STOP                               Near   CODE:0034
TOP_STACK                          Word   STACK_SEG:00C8

Groups & Segments                  Bit Size Align  Combine Class

CODE                               16  0035 Word   Public
DATA                               16  0006 Word   Public
MORE_DATA                          16  0006 Word   none
PROCEDURES                         16  0000 Para   Public
STACK_SEG                          16  00C8 Para   Stack
```

(a)

FIGURE 5-27 (*continued*)

TOR EQU 07 in one module, you can import CORRECTION_FACTOR to another module with the statement
EXTRN CORRECTION_FACTOR:ABS.

Now that we have explained the use of PUBLIC and EXTRN, let's work our way through the rest of the program. At the start of the mainline, the ASSUME statement tells the assembler which logical segments to use as code, data, and stack. We then initialize the data segment, stack segment, and stack pointer registers as described in previous example programs. Now, before calling the SMART_DIVIDE procedure, we copy the dividend and divisor from memory to some registers. The dividend and the divisor are passed to the procedure in these registers. As we explained in a previous section, if we pass parameters to a procedure in registers, the procedure does not have to refer to specific named memory locations. The procedure is then more general and can more easily be called from any place in the mainline program. However, in this example we referenced the named memory location, DIVISOR, from the procedure just to show you how it can be done using the EXTRN and PUBLIC directives. The procedure is of type FAR, so when we call it, both the code segment register and the instruction pointer contents will be changed.

In the procedure shown in Figure 5-27*b*, we first check to see if the divisor is zero with a CMP DIVISOR,0 instruction. If the divisor is zero, the JE instruction will send execution to the label ERROR_EXIT. There we set the carry flag with STC as an error indicator and return to the mainline program. If the divisor is not zero, then we go on with the division. To understand how we do the division, remember that the 8086 DIV instruction divides the 32-bit number in DX and AX by the 16-bit number in a specified register or memory location. It puts a 16-bit quotient in AX and a 16-bit remainder in DX. Now, according to our algorithm in Figure 5-26*b*, we want to put 0000H in DX and the high word of the dividend in AX for our first DIV operation. MOV BX,AX saves a copy of the low word of the dividend for future reference. MOV AX,DX copies the high word of the dividend into AX where we want it, and MOV DX,0000H puts all 0's in DX. After the first DIV instruction executes, AX will contain the high word of the 32-bit quotient we want as our final answer. We save this in BP with the MOV BP,AX instruction so that we can use AX for the second DIV operation.

The remainder from the first DIV operation was left in the DX register. As shown by the diagram in Figure 5-26*b*, this is right where we want it for the second DIV operation. All we have to do now, before we do the second DIV operation, is to get the low word of the original dividend back into AX with the MOV AX,BX instruction. After the second DIV instruction executes, the 16-bit quotient will be in AX. This word is the low word of our

```
Turbo Assembler  Version 1.0        05-05-89 13:09:20         Page 1

                                    ; 8086 PROCEDURE F5-27B.ASM called by program F5-27A.ASM
                                ;ABSTRACT  : PROCEDURE SMART_DIVIDE.
                                    ; This procedure divides a 32-bit number by a 16-bit number
                                    ; to give a 32-bit quotient and a 16-bit remainder.
                                ;INPUT     : Dividend - low word in AX, high word in DX, Divisor in CX
                                ;OUTPUT    : Quotient - low word in AX, high word in DX. Remainder in CX
                                    ; Carry    - carry flag set if try to divide by zero
                                ;DESTROYS  : AX, BX, CX, DX, BP, FLAGS
                                ;PORTS     : None used

1,1 0000                        DATA      SEGMENT  PUBLIC ; This block tells the assembler that
                                   EXTRN  DIVISOR:WORD    ; the divisor is a word variable found
                                DATA      ENDS            ; in the external segment named DATA

                                PUBLIC  SMART_DIVIDE      ; Make SMART_DIVIDE available to other modules

                                PROCEDURES   SEGMENT   PUBLIC
                                SMART_DIVIDE PROC      FAR
                                  ASSUME CS:PROCEDURES, DS:DATA
0000  83 3E 0000e 00                   CMP  DIVISOR, 0     ; Check for illegal divide
0005  74 17                            JE   ERROR_EXIT     ; IF divisor = 0, exit procedure
0007  8B D8                            MOV  BX, AX         ; Save low order of dividend
0009  8B C2                            MOV  AX, DX         ; Position high word for 1st divide
000B  BA 0000                          MOV  DX, 0000H      ; Zero DX
000E  F7 F1                            DIV  CX             ; DX:AX/CX, quotient in AX, remainder in DX
0010  8B E8                            MOV  BP, AX         ; Save high order of final result
0012  8B C3                            MOV  AX, BX         ; Get back low order of dividend
0014  F7 F1                            DIV  CX             ; DX:AX/CX, quotient in AX, remainder in DX
0016  8B CA                            MOV  CX, DX         ; Pass remainder back in CX
0018  8B D5                            MOV  DX, BP         ; Pass high order result back in DX
001A  F8                               CLC                 ; Clear carry to indicate valid result
001B  EB 02 90                         JMP  EXIT           ; Finished
001E  F9                        ERROR_EXIT:  STC           ; Set carry to indicate divide by zero
001F  CB                        EXIT:        RET
0020                            SMART_DIVIDE ENDP
0020                            PROCEDURES   ENDS
                                             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               Bit Size Align  Combine Class

DATA                            16  0000 Para   Public
PROCEDURES                      16  0020 Para   Public
```

(b)

FIGURE 5-27 *(continued)*

desired 32-bit quotient. We just leave this word in AX to be passed back to the mainline program. The DX register was left with the final remainder. We copy this remainder to CX with the MOV CX,DX instruction to be passed back to the mainline program. After the first DIV operation, we saved the high word of our 32-bit quotient in BP. We now use the MOV DX,BP instruction to copy this word back to DX, where we want it to be when we return to the mainline program. You really don't have to shuffle the results around the way we did with these last three instructions, but we like to pass parameters to and from procedures in as systematic a way as possible so that we can more easily keep track of everything. After the shuffling, we clear the carry flag with CLC before returning to indicate that the result in DX and AX is valid.

Back in the mainline program, we check the carry flag with the JNC instruction. If the carry flag is set, we know that the divisor was 0, no division was done, and there is no result to put in memory. If the carry flag is not set, then we know that a valid 32-bit quotient was returned in DX and AX and a 16-bit remainder was returned in CX. We now want to copy this quotient and this remainder to some named memory locations we set aside for them.

If you look at some earlier lines in the program, you will see that the memory locations called QUOTIENT and REMAINDER are in a segment called MORE_DATA. At the start of the mainline program, we tell the assembler to ASSUME that we will be using DATA as the data segment. Now, however, we want to access some data items in MORE_DATA using DS. To do this, we have to do two things. First, we have to tell the assembler to ASSUME DS:MORE_DATA. Second, we have to load the segment base of MORE_DATA into DS. In our program we save the old value of DS by pushing it on the stack. We do this so that we can easily reload DS with the base address of DATA later in the program. The MOV BX,MORE_DATA and MOV DS,BX instructions load the base address of MORE_DATA into DS. The three MOV instructions after this copy the quotient and the remainder into the named memory locations.

Finally, in the program we point DS back at DATA so that later instructions can access data items in the DATA segment. To do this, we first tell the assembler to ASSUME DS:DATA. Then we pop the base address of DATA off the stack into DS. As you write more complex programs, you will often want to access different segments at different times in the program, so we wrote this example to show you how to do it. Remember, when you change segments, you have to do a new ASSUME statement and include instructions which initialize the segment register to the base address of the new segment.

WRITING AND USING ASSEMBLER MACROS

Macros and Procedures Compared

Whenever we need to use a group of instructions several times throughout a program, there are two ways we can avoid having to write the group of instructions each time we want to use it. One way is to write the group of instructions as a separate procedure. We can then just call the procedure whenever we need to execute that group of instructions. A big advantage of using a procedure is that the machine codes for the group of instructions in the procedure only have to be put in memory once. Disadvantages of using a procedure are the need for a stack, and the overhead time required to call the procedure and return to the calling program.

When the repeated group of instructions is too short or not appropriate to be written as a procedure, we use a macro. A *macro* is a group of instructions we bracket and give a name to at the start of our program. Each time we "call" the macro in our program, the assembler will insert the defined group of instructions in place of the "call." In other words, the macro call is like a shorthand expression which tells the assembler, "Every time you see a macro name in the program, replace it with the group of instructions defined as that macro at the start of the program." An important point here is that the assembler generates machine codes for the group of instructions each time the macro is called. Replacing the macro with the instructions it represents is commonly called "expanding" the macro. Since the generated machine codes are right *in-line* with the rest of the program, the processor does not have to go off to a procedure and return. Therefore, using a macro avoids the overhead time involved in calling and returning from a procedure. A disadvantage of generating in-line code each time a macro is called is that this will make the program take up more memory than using a procedure.

The examples which follow should help you see how to define and call macros. For these examples we use the syntax of MASM and TASM. If you are developing your programs on some other machine, consult the assembly language programming manual for your machine to find the macro definition and calling formats for it.

Defining and Calling a Macro Without Parameters

For our first example, suppose that we are writing an 8086 program which has many complex procedures. At the start of each procedure, we want to save the flags and all the registers by pushing them on the stack. At the end of each procedure, we want to restore the flags and all the registers by popping them off the stack. Each procedure would normally contain a long series of PUSH instructions at the start and a long series of POP instructions at the end. Typing in these lists of PUSH and POP instructions is tedious and prone to errors. We could write a procedure to do the pushing and another procedure to do the popping. However, this adds more complexity to the program and is therefore not appropriate. Two simple macros will solve the problem for us.

Here's how we write a macro to save all the registers.

```
PUSH_ALL MACRO
         PUSHF
         PUSH AX
         PUSH BX
```

```
        PUSH CX
        PUSH DX
        PUSH BP
        PUSH SI
        PUSH DI
        PUSH DS
        PUSH ES
        PUSH SS
ENDM
```

The PUSH_ALL MACRO statement identifies the start of the macro and gives the macro a name. The ENDM identifies the end of the macro.

Now, to call the macro in one of our procedures, we simply put in the name of the macro just as we would an instruction mnemonic. The start of the procedure which does this might look like this:

```
BREATH_RATE     PROC FAR
ASSUME CS:PROCEDURES, DS:PATIENT_PARAMETERS
   PUSH_ALL                          : Macro call
   MOV AX, PATIENT_PARAMETERS : Initialize data
   MOVE DS, AX                       : segment reg
```

When the assembler assembles this program section, it will replace PUSH_ALL with the instructions that it represents and insert the machine codes for these instructions in the object code version of the program. The assembler listing tells you which lines were inserted by a macro call by putting a + in each program line inserted by a macro call. As you can see from the example here, using a macro makes the source program much more readable because the source program does not have the long series of push instructions cluttering it up.

The preceding example showed how a macro can be used as simple shorthand for a series of instructions. The real power of macros, however, comes from being able to pass parameters to them when you call them. The next section shows you how and why this is done.

Passing Parameters to Macros

Most of us have received computer printed letters of the form:

> Dear MR. HALL,
>
> We are pleased to inform you that you may have won up to $1,000,000 in the *Reader's Weekly* sweepstakes. To find out if you are a winner, MR. HALL, return the gold card to *Reader's Weekly* in the enclosed envelope before OCTOBER 22, 1991. You can take advantage of our special offer of three years of *Reader's Weekly* for only $24.95 by putting an X in the YES box on the gold card. If you do not wish to take advantage of this offer, which is one third off the newsstand price, mark the no box on the gold card.
>
> Thank you.

A letter such as this is an everyday example of the macro with parameters concept. The basic letter "macro" is written with dummy words in place of the addressee's name, the reply date, and the cost of a three-year subscription. Each time the macro which prints the letter is called, new values for these parameters are passed to the macro. The result is a "personal"-looking letter.

In assembly language programs, we likewise can write a generalized macro with dummy parameters. Then, when we call the macro, we can pass it the actual parameters needed for the specific application. Suppose, for example, we are writing a word processing program. A frequent need in a word processing program is to move strings of ASCII characters from one place in memory to another. The 8086 MOVS instruction is intended to do this. Remember from the discussion of the string instructions at the beginning of this chapter, however, that in order for the MOVS instruction to work correctly, you first have to load SI with the offset of the source start, DI with the offset of the destination start, and CX with the number of bytes or words to be moved. We can define a macro to do all of this as follows:

```
MOVE_ASCII MACRO NUMBER, SOURCE, DESTINATION
     MOV CX, NUMBER          ; Number of characters to be moved in CX
     LEA SI, SOURCE          ; Point SI at ASCII source
     LEA DI, DESTINATION     ; Point DI at ASCII destination
     CLD                     ; Autoincrement pointers after move
REP MOVSB                    ; Copy ASCII string to new location
  ENDM                       ;
```

The words NUMBER, SOURCE, and DESTINATION in this macro are called *dummy variables*. When we call the macro, values from the calling statement will be put in the instructions in place of the dummies. If, for example, we call this macro with the statement MOVE_ASCII 03DH,BLOCK_START,BLOCK_DEST, the assembler will expand the macro as follows.

```
     MOV CX, 03DH            ; Number of characters to be moved in CX
     LEA SI, BLOCK_START     ; Point SI at ASCII destination
     LEA DI, BLOCK_DEST      ; Point DI at ASCII destination
     CLD                     ; Autoincrement pointers after move
REP MOVSB                    ; Copy ASCII string to new location
```

We do not have space here to show you very much of what you can do with macros. Read through the assembly language programming manual for your system to find more details about working with macros. To help stick in your mind the differences between procedures and macros, here is a comparison between the two.

Summary of Procedures Versus Macros

PROCEDURE

Accessed by CALL and RET mechanism during program execution. Machine code for instructions only put in memory once. Parameters passed in registers, memory locations, or stack.

MACRO

Accessed during assembly with name given to macro when defined. Machine code generated for instructions each time called. Parameters passed as part of statement which calls macro.

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.

Strings and 8086 string instructions

Procedures and nested procedures

CALL and RET instructions

Near and far procedures

Direct intersegment far call

Indirect intersegment far call

Direct intrasegment near call

Indirect intrasegment near call

Stack: top of stack, stack pointer

PUSH and POP instructions

Parameter, parameter passing methods

Stack overflow

Reentrant and recursive procedures

Interrupt

Interrupt service procedure

Separate assembly modules

PUBLIC and EXTRN directives

Macro

REVIEW QUESTIONS AND PROBLEMS

1. *a.* Given the following data structure, use the 8086 string instructions to help you write a program which moves the string "Charlie T. Tuna" from OLD_HOME to NEW_HOME, which is just above the initial location.

```
NAMES_HERE    SEGMENT
  OLD_HOME    DB 'CHARLIE T. TUNA'
  NEW_HOME    DB 15 DUP(0)
NAMES_HERE    ENDS
```

   b. Use the string instructions to write a simple program to move the string "Charlie T. Tuna" up four addresses in memory. Consider whether the pointers should be incremented or decremented after each byte is moved in order to keep any needed byte from being written over. *Hint:* Initialize DI with the value of SI + 4.

2. Use the 8086 string instructions to write a program which scans a string of 80 characters looking for a carriage return (0DH). If a carriage return is found, put the length of the string up to the carriage return in AL. If no carriage return is found, put 50H (80 decimal) in AL.

3. Show the 8086 instruction or group of instructions which will:
   a. Initialize the stack segment register to 4000H and the stack pointer register to 8000H.
   b. Call a near procedure named FIXIT.
   c. Save BX and BP at the start of a procedure and restore them at the end of the procedure.
   d. Return from a procedure and automatically increment the stack pointer by 8.

4. *a.* Use a stack map to show the effect of each of the following instructions on the stack pointer and on the contents of the stack.

```
      MOV SP,4000H
      PUSH AX
      CALL MULTO
      POP AX
MULTO PROC NEAR
      PUSHF
      PUSH BX
      .
      .
      .
      POP BX
      POPF
      RET
MULTO ENDP
```

   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.
   a. The instruction which will call a procedure which is 97H addresses higher in memory than the instruction after a call instruction.
   b. 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 Figure 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. Figure 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.

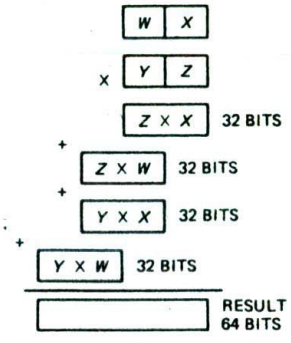


FIGURE 5-28 32-bit by 32-bit multiply method for Problem 9.

10. Calculating the factorial of a number, which we did with a recursive procedure in Figure 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*.

CHAPTER 6

8086 Instruction Descriptions and Assembler Directives

This chapter consists of two major sections. The first section is a dictionary of all the 8086/8088 instructions. For each instruction, we give a detailed description of its operation, the correct syntax for the instruction, and the flags affected by the instruction. Numerical examples are shown for those instructions for which they are appropriate. Instead of putting the binary codes for the instructions here, we have listed them alphabetically in Appendixes A and B. Putting the codes together in a table makes them easier to find if you are hand coding a program.

The second major section of this chapter is a dictionary of commonly used 8086 assembler directives. The directives described here are those defined for the Intel 8086 macro assembler, the Microsoft macro assembler (MASM), and the Borland Turbo Assembler (TASM). If you are using some other assembler, it probably has similar capabilities, but the names may be different.

You will probably use this chapter mostly as a reference to get the details of an instruction or directive as you write programs of your own or decipher someone else's programs. However, you should skim through the chapter at least once to give yourself an overview of the material it contains. You should not try to absorb all of this chapter at once. Many of the instructions described are used and discussed in various example programs throughout the book. For these instructions, we have included references to the appropriate sections in the text.

INSTRUCTION DESCRIPTIONS

AAA—ASCII Adjust for Addition

Numerical data coming into a computer from a terminal is usually in ASCII code. In this code, the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each. After the addition, the AAA instruction is used to make sure the result is the correct unpacked BCD. A simple numerical example will show how this works.

EXAMPLE

```
                ; Assume AL = 0011 0101, ASCII 5
                ; BL = 0011 1001, ASCII 9
ADD AL, BL      ; Result: AL = 0110 1110 = 6EH, which
                ; is incorrect BCD
AAA             ; Now AL = 00000100, unpacked BCD 4.
                ; CF = 1 indicates answer is 14 decimal
```

NOTE: OR AL with 30H to get 34H, the ASCII code for 4, if you want to send the result back to a CRT terminal. The 1 in the carry flag can be rotated into the low nibble of a register, ORed with 30H to give the ASCII code for 1, and then sent to the terminal.

The AAA instruction works only on the AL register.

The AAA instruction updates AF and CF, but OF, PF, SF, and ZF are left undefined.

AAD—BCD-to-Binary Convert before Division

AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. After the division, AL will contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder. PF, SF, and ZF are updated. AF, CF, and OF are undefined after AAD.

EXAMPLE:

```
                ; AX = 0607H unpacked BCD for 67 decimal
                ; CH = 09H, now adjust to binary
AAD             ; Result: AX = 0043 = 43H = 67 decimal
DIV CH          ; Divide AX by unpacked BCD in CH
                ; Quotient: AL = 07 unpacked BCD
                ; Remainder: AH = 04 unpacked BCD
                ; Flags undefined after DIV
```

NOTE: If an attempt is made to divide by 0, the 8086 will do a type 0 interrupt. The type 0 interrupt response is described in Chapter 8.

AAM—BCD Adjust after Multiply

Before you can multiply two ASCII digits, you must first mask the upper 4 bits of each. This leaves unpacked BCD (one BCD digit per byte) in each byte. After the two unpacked BCD digits are multiplied, the AAM instruc-

tion is used to adjust the product to two unpacked BCD digits in AX.

AAM works only after the multiplication of two unpacked BCD bytes, and it works only on an operand in AL. AAM updates PF, SF, and ZF, but AF, CF, and OF are left undefined.

EXAMPLE:

```
                ; AL = 00000101 = unpacked BCD 5
                ; BH = 00001001 = unpacked BCD 9
MUL BH          ; AL × BH; result in AX
                ; AX = 00000000 00101101 = 002DH
AAM             ; AX = 00000100 00000101 = 0405H,
                ; which is unpacked BCD for 45.
                ; If ASCII codes for the result are
                ; desired, use next instruction
OR AX,3030H     ; Put 3 in upper nibble of each byte.
                ; AX = 00110100 00110101 = 3435H,
                ; which is ASCII code for 45
```

AAS—ASCII Adjust for Subtraction

Numerical data coming into a computer from a terminal is usually in ASCII code. In this code the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to subtract the ASCII codes for two decimal digits without masking the "3" in the upper nibble of each. The AAS instruction is then used to make sure the result is the correct unpacked BCD. Some simple numerical examples will show how this works:

EXAMPLE:

```
                ; ASCII 9–ASCII 5 (9–5)
                ; AL = 00111001 = 39H = ASCII 9
                ; BL = 00110101 = 35H = ASCII 5
SUB AL, BL      ; Result: AL = 00000100 = BCD 04
                ; and CF = 0
AAS             ; Result: AL = 00000100 = BCD 04
                ; and CF = 0; no borrow required

                ; ASCII 5–ASCII 9 (5–9)
                ; Assume AL = 00110101 = 35H =
                  ASCII 5
                ; and BL = 00111001 = 39H = ASCII 9
SUB AL, BL      ; Result: AL = 11111100 = – 4
                ; in 2's complement and CF = 1
AAS             ; Result: AL = 00000100 = BCD 04
                ; and CF = 1; borrow needed
```

The AAS instruction leaves the correct unpacked BCD result in the low nibble of AL and resets the upper nibble of AL to all 0's. If you want to send the result back to a CRT terminal, you can OR AL with 30H to produce the correct ASCII code for the result. If multiple-digit numbers are being subtracted, the CF can be taken into account by using the SBB instruction when subtracting the next digits.

The AAS instruction works only on the AL register. It updates AF and CF, but OF, PF, SF, and ZF are left undefined.

ADC—Add with Carry—ADC Destination,Source
ADD—Add—ADD Destination,Source

These instructions add a number from some source to a number from some destination and put the result in the specified destination. The Add with Carry instruction, ADC, also adds the status of the carry flag into the result. The source may be an immediate number, a register, or a memory location specified by any one of the 24 addressing modes shown in Figure 3-8. The destination may be a register or a memory location specified by any one of the 24 addressing modes in Figure 3-8. The source and the destination in an instruction cannot both be memory locations. The source and the destination must be of the same type. In other words, they must both be byte locations, or they must both be word locations. If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with 0's before adding. Flags affected: AF, CF, OF, PF, SF, ZF.

EXAMPLES (CODING):

```
                ; Add immediate number 74H to
ADD AL,74H      ; contents of AL. Result in AL

                ; Add contents of BL plus carry
                  status
ADC CL,BL       ; to contents of CL.

                ; Add contents of BX
ADD DX,BX       ; to contents of DX

                ; Add word from memory at offset
                  [SI]
ADD DX,[SI]     ; in DS to contents of DX

                ; Add byte from effective
ADC AL,PRICES[BX]  ; address PRICES[BX] plus carry
                ; status to contents of AL

                ; Add contents of AL to
ADD PRICES[BX],AL  ; contents of memory location at
                ; effective address PRICES[BX]
```

EXAMPLES (NUMERICAL):

```
                ; Addition of unsigned numbers
                ; CL = 01110011 = 115 decimal
                ; + BL = 01001111 = 79 decimal
ADD CL,BL       ; Result in CL
                ; CL = 11000010 = 194 decimal


                ; Addition of signed numbers
                ; CL = 01110011 = + 115 decimal
                ; + BL = 01001111 = + 79 decimal
ADD CL,BL       ; Result in CL
                ; CL = 11000010 = – 62 decimal—
                ; incorrect because result too large to fit
                ; in 7 bits
```

FLAG RESULTS FOR SIGNED ADDITION EXAMPLE

CF = 0 No carry out of bit 7.
PF = 0 Result has odd parity.
AF = 1 Carry was produced out of bit 3.
ZF = 0 Result in destination was not 0.
SF = 1 Copies most significant bit of result; indicates negative result if you are adding signed numbers.
OF = 1 Set to indicate that the result of the addition was too large to fit in the lower 7 bits of the destination used to represent the magnitude of a signed number. In other words, the result was greater than +127 decimal, so the result overflowed into the sign bit position and incorrectly indicated that the result was negative. If you are adding two signed 16-bit values, the OF will be set if the magnitude of the result is too large to fit in the lower 15 bits of the destination.

NOTE: PF is meaningful only for an 8-bit result. AF is set only by a carry out of bit 3. Therefore, the DAA instruction cannot be used after word additions to convert the result to correct BCD.

AND—AND Corresponding Bits of Two Operands—AND Destination,Source

This instruction ANDs each bit in a source byte or word with the same number 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 position will follow the truth table for a two-input AND gate. In other words, a bit in the specified destination will be a 1 only if that bit is a 1 in both the source and the destination operands. Therefore, a bit can be masked (reset) by ANDing it with 0.

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 Figure 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 AND. PF, SF, and ZF are updated by AND. AF is undefined. Note that PF has meaning only for an 8-bit operand.

EXAMPLES (CODING):

```
                ; AND word in DS at offset [SI]
                  with word in CX register
AND CX,[SI]     ; Result in CX register

AND BH,CL       ; AND byte in CL with byte in BH
                ; Result in BH

                ; AND word in BX with immediate
AND BX,00FFH    ; 00FFH. Masks upper byte, leaves
                ; lower byte unchanged
```

EXAMPLE (NUMERICAL):

```
                ; BX = 10110011 01011110
AND BX,00FFH    ; Mask out upper 8 bits of BX
                ; Result: BX = 00000000 01011110
                ; CF, OF, PF, SF, ZF = 0
```

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 intrasegment) 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.

CMC—Complement the Carry Flag

If the carry flag (CF) is a 0 before this instruction, it will be set to a 1 after the instruction. If the carry flag is 1 before this instruction, it will be reset to a 0 after the instruction executes. CMC affects no other flags.

EXAMPLE:

```
CMC   ; Invert the carry flag
```

CMP—Compare Byte or Word—CMP Destination,Source

This instruction compares a byte from the specified source with a byte from the specified destination, or a word from the specified source with a word from the specified destination. The source can be an immediate number, a register, or a memory location specified by one of the 24 addressing modes shown in Figure 3-8. The destination can be a register or a memory location. However, the source and the destination cannot both be memory locations in the same instruction. The comparison is actually done by subtracting the source byte or word from the destination byte or word. The source and the destination are not changed, but the flags are set to indicate the results of the comparison. AF, OF, SF, ZF, PF, and CF are updated by the CMP instruction. For the instruction CMP CX,BX, CF, ZF, and SF will be left as follows:

```
          CF  ZF  SF
CX = BX   0   1   0   ; Result of subtraction is 0
CX > BX   0   0   0   ; No borrow required, so CF = 0
CX < BX   1   0   1   ; Subtraction required
                      ; borrow, so CF = 1
```

EXAMPLES:

```
                      ; Compare immediate number
CMP AL,01H            ; 01H with byte in AL

                      ; Compare byte in CL with
CMP BH,CL             ; byte in BH

                      ; Compare word in DS at
                      ; displacement TEMP_MIN
CMP CX,TEMP_MIN       ; with word in CX

                      ; Compare CX with word in DS
CMP TEMP_MAX,CX       ; at displacement TEMP_MAX

                      ; Compare immediate 49H
CMP PRICES[BX],49H    ; with byte at offset
                      ; [BX] in array PRICES
```

NOTE: The Compare instructions are often used with the Conditional Jump instructions, described in a later section. Having the Compare instructions formatted the way they are makes this use very easy to understand. For example, given the instruction sequence

```
CMP BX,CX
JAE TARGET
```

you can mentally read it as "jump to target if BX is above or equal to CX." In other words, just mentally insert the first operand after the J for jump and the second operand after the condition.

CMPS/CMPSB/CMPSW—Compare String Bytes or String Words

A string is a series of the same type of data items in sequential memory locations. The CMPS instruction can be used to compare a byte in one string with a byte in another string or to compare a word in one string with a word in another string. SI is used to hold the offset of a byte or word in the source string, and DI is used to hold the offset of a byte or a word in the other string. The comparison is done by subtracting the byte or word pointed to by DI from the byte or word pointed to by SI. The AF, CF, OF, PF, SF, and ZF flags are affected by the comparison, but neither operand is affected. After the comparison, SI and DI will automatically be incremented or decremented to point to the next elements in the two strings. If the direction flag has previously been set to a 1 with an STD instruction, then SI and DI will automatically be decremented by 1 for a byte string or by 2 for a word string. If the direction flag has previously been reset to a 0 with a CLD instruction, then SI and DI will automatically be incremented after the compare. They will be incremented by 1 for byte strings and by 2 for word strings.

The string pointed to by DI must be in the extra segment. The string pointed to by SI must be in the data segment.

The CMPS instruction can be used with a REPE or REPNE prefix to compare all the elements of a string. For further discussion of strings, see the discussion at the start of Chapter 5.

EXAMPLE:

```
MOV SI,OFFSET FIRST_STRING
                    ; Point SI at source string

MOV DI,OFFSET SECOND_STRING
                    ; Point DI at destination string
CLD                 ; DF cleared, so SI and DI will
                    ; autoincrement after compare
MOV CX,100          ; Put number of string elements
                      in CX
REPE CMPSB          ; Repeat the comparison of
                      string bytes
                    ; until end of string or until
                    ; compared bytes are not equal
```

NOTE: CX functions as a counter which the REPE prefix will cause to be decremented after each compare. The B attached to CMPS tells the assembler that the strings are of type byte. If you want to tell the assembler that the strings are of type word, write the instruction as CMPSW. The REPE CMPSW instruction will cause the pointers in SI and DI to be incremented by 2 after each compare if the direction flag is cleared or decremented by 2 if the direction flag is set.

CWD—Convert Signed Word to Signed Doubleword

CWD copies the sign bit of a word in AX to all the bits of the DX register. In other words, it extends the sign of AX into all of DX. The CWD operation must be done before a signed word in AX can be divided by another signed word with the IDIV instruction. CWD affects no flags.

EXAMPLE:

```
        ; DX = 00000000 00000000
        ; AX = 11110000 11000111 = - 3897 decimal
CWD     ; Convert signed word in AX to signed
        ; doubleword in DX:AX
        ; Result: DX = 11111111 11111111
        ; AX = 11110000 11000111 = - 3897 decimal
```

For a further example of the use of CWD, see the IDIV instruction description.

DAA—Decimal Adjust AL after BCD Addition

This instruction is used to make sure the result of adding two packed BCD numbers is adjusted to be a legal BCD number. The result of the addition must be in AL for DAA to work correctly. If the lower nibble in AL after an addition is greater than 9 or AF was set by the addition, then the DAA instruction will add 6 to the lower nibble in AL. If the result in the upper nibble of AL is now greater than 9 or if the carry flag was set by the addition or correction, then the DAA instruction will add 60H to AL. A couple of simple examples should clarify how this works.

EXAMPLES:

```
                    ; AL = 0101 1001 = 59 BCD
                    ; BL = 0011 0101 = 35 BCD
ADD AL,BL           ; AL = 1000 1110 = 8EH
DAA                 ; Add 0110 because 1110 > 9
                    ; AL = 1001 0100 = 94 BCD

                    ; AL = 1000 1000 = 88 BCD
                    ; BL = 0100 1001 = 49 BCD
ADD AL,BL           ; AL = 1101 0001, AF = 1
DAA                 ; Add 0110 because AF = 1
                    ; AL = 1101 0111 = D7H
                    ; 1101 > 9 so add 0110 0000
                    ; AL = 0011 0111 = 37 BCD, CF = 1
```

The DAA instruction updates AF, CF, PF, and ZF. OF is undefined after a DAA instruction.

A decimal up counter can be implemented using the DAA instruction as follows:

```
MOV COUNT,00H   ; Initialize count in memory
                ; location to 0
                ; Other instructions here
MOV AL, COUNT   ; Bring count into AL to work on
ADD AL,01H      ; Can also count up by 2, by 3, or
                ; by some other number using the
                ; ADD instruction
DAA             ; Decimal adjust the result
MOV COUNT,AL    ; Put decimal result back
                ; in memory
```

DAS—Decimal Adjust after BCD Subtraction

This instruction is used after subtracting two packed BCD numbers to make sure the result is correct packed BCD. The result of the subtraction must be in AL for DAS to work correctly. If the lower nibble in AL after a subtraction is greater than 9 or the AF was set by the subtraction, then the DAS instruction will subtract 6 from the lower nibble of AL. If the result in the upper nibble is now greater than 9 or if the carry flag was set, the DAS instruction will subtract 60 from AL. A couple of simple examples should clarify how this works.

EXAMPLES:

```
                ; AL = 1000 0110 = 86 BCD
                ; BH = 0101 0111 = 57 BCD
SUB AL,BH       ; AL = 0010 1111 = 2FH, CF = 0
DAS             ; Lower nibble of result is 1111,
                ; so DAS automatically subtracts
                ; 0000 0110 to give AL = 00101001
                ; = 29 BCD

                ; AL = 0100 1001 = 49 BCD
                ; BH = 0111 0010 = 72 BCD
SUB AL,BH       ; AL = 1101 0111 = D7H, CF = 1
DAS             ; Subtracts 0110 0000 (- 60H)
                ; because 1101 in upper nibble > 9
                ; AL = 01110111 = 77 BCD, CF = 1
                ; CF = 1 means borrow was needed
```

The DAS instruction updates AF, CF, SF, PF, and ZF, but OF is undefined.

A decimal down counter can be implemented using the DAS instruction as follows:

```
MOV AL,COUNT    ; Bring count into AL to work on
SUB AL,01H      ; Decrement. Can also count down
                ; by 2, 3, etc., using SUB instruction
DAS             ; Keep results in BCD format
MOV COUNT,AL    ; Put new count back in memory
```

DEC—Decrement Destination Register or Memory—DEC Destination

This instruction subtracts 1 from the destination word or byte. The destination can be a register or a memory location specified by any one of the 24 addressing modes shown in Figure 3-8. AF, OF, PF, SF, and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing 00H or a 16-bit destination containing 0000H is decremented, the result will be FFH or FFFFH with no carry (borrow).

EXAMPLES

```
DEC CL          ; Subtract 1 from contents of CL register

DEC BP          ; Subtract 1 from contents of BP register
```

DEC BYTE PTR [BX]; Subtract 1 from byte at offset [BX] in DS. The BYTE PTR directive is necessary to tell the assembler to put in the correct code for decrementing a byte in memory, rather than decrementing a word. The instruction essentially says, "Decrement the byte in memory pointed to by the offset in BX."

DEC WORD PTR [BP] ; Subtract 1 from a word at offset [BP] in SS. The WORD PTR directive tells the assembler to put in the code for decrementing a word pointed to by the contents of BP. An offset in BP will be added to the SS register contents to produce the physical address.

DEC TOMATO_CAN_COUNT ; Subtract 1 from byte or word named TOMATO_CAN_COUNT in DS. If TOMATO_CAN_COUNT was declared with a DB, then the assembler will code this instruction to decrement a byte. If TOMATO_CAN_COUNT was declared with a DW, then the assembler will code this instruction to decrement a word.

DIV—Unsigned Divide—DIV Source

This instruction is used to divide an unsigned word by a byte or to divide an unsigned doubleword (32 bits) by a word.

When a word is divided by a byte, the word must be in the AX register. The divisor can be in a register or a memory location. After the division, AL will contain an 8-bit result (quotient), and AH will contain an 8-bit remainder. If an attempt is made to divide by 0 or if the quotient is too large to fit in AL (greater than FFH), the 8086 will automatically do a type 0 interrupt. Interrupts are explained in Chapter 8.

When a doubleword is divided by a word, the most significant word of the doubleword must be in DX, and the least significant word of the doubleword must be in AX. After the division, AX will contain the 16-bit result (quotient), and DX will contain a 16-bit remainder. Again, if an attempt is made to divide by 0 or if the quotient is too large to fit in AX (greater than FFFFH), the 8086 will do a type 0 interrupt.

For a DIV, the dividend (numerator) must always be in AX or DX and AX, but the source of the divisor (denominator) can be a register or a memory location specified by any one of the 24 addressing modes shown in Figure 3-8. If the divisor does not divide an integral number of times into the dividend, the quotient is

truncated, not rounded. The example below will illustrate this. All flags are undefined after a DIV instruction.

If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0's. The SUB AH,AH instruction is a quick way to do this. Likewise, if you want to divide a word by a word, put the dividend word in AX and fill DX with all 0's. The SUB DX,DX instruction does this quickly.

EXAMPLES (SYNTAX):

```
DIV BL          ; Divide word in AX by byte in BL.
                  Quotient in AL, remainder in AH

DIV CX          ; Divide doubleword in DX and AX by
                  word in CX. Quotient in AX,
                  remainder in DX.

DIV SCALE[BX]   ; AX/(byte at effective address
                  SCALE[BX]) if SCALE[BX] is of type
                  byte or (DX and AX)/(word at effective
                  address SCALE [BX]) if SCALE[BX]
                  is of type word
```

EXAMPLE (NUMERICAL):

```
        ; AX = 37D7H = 14,295 decimal
        ; BH = 97H = 151 decimal
DIV BH  ; 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):

```
IDIV BL         ; Signed word in AX/signed byte
                  in BL
```

```
IDIV BP              ; Signed doubleword in DX and
                       AX/signed word
                     ; in BP

IDIV BYTE PTR [BX]   ; AX/byte at offset [BX] in DS

MOV AL,DIVIDEND      ; Position byte dividend
CBW                  ; Extend sign of AL into AH
IDIV DIVISOR         ; Divide by byte divisor
```

EXAMPLES (NUMERICAL):

```
                     ; A signed word divided by a signed byte
                     ; AX = 00000011 10101011 = 03ABH
                     ; = 39 decimal
                     ; BL = 11010011 = D3H = - 2DH
                     ; = - 45 decimal
IDIV BL              ; Quotient: AL = ECH = - 14H = - 20
                       decimal
                     ; Remainder: AH = 27H = + 39 decimal
```

NOTE: The quotient is negative because positive was divided by negative. The remainder has same sign as dividend (positive).

```
                     ; A signed byte divided by a signed byte
                     ; AL = 11011010 = - 26 H = - 38 decimal
                     ; CH = 00000011 = + 3H = + 3 decimal
CBW                  ; Extend sign of AL through AH,
                     ; AX = 11111111 11011010
IDIV CH              ; Divide AX by CH
                     ; AL = 11110100 = - 0CH = - 12 decimal
                     ; AH = 11111110 = - 2H = - 2 decimal
```

Although the quotient is actually closer to 13 (12.666667) than to 12, the 8086 truncates it to 12 rather than rounding it to 13. If you want to round the quotient, you can compare the magnitude of the remainder with (divisor/2) and add 1 to the quotient if the remainder is greater than (divisor/2). Note that the sign of the remainder is the same as the sign of the dividend (negative). All flags are undefined after IDIV.

IMUL—Multiply Signed Numbers—IMUL Source

This instruction multiplies a signed byte from some source times a signed byte in AL or a signed word from some source times a signed word in AX. The source can be another register or a memory location specified by any one of the 24 addressing modes shown in Figure 3-8. When a byte from some source is multiplied by AL, the signed result (product) will be put in AX. A 16-bit destination is required because the result of multiplying two 8-bit numbers can be as large as 16 bits. When a word from some source is multiplied by AX, the result can be as large as 32 bits. The high-order (most significant) word of the signed result is put in DX, and the low-order (least significant) word of the signed result is put in AX. If the magnitude of the product does not require all the bits of the destination, the unused bits will be filled with copies of the sign bit. If the upper byte of a 16-bit result or the upper word of a 32-bit result contains only copies of the sign bit (all 0's or all 1's), then CF and the OF will both be 0. If the upper byte of a 16-bit result or the upper word of a 32-bit result contains part of the product, CF and OF will both be 1. You can use the status of these flags to determine whether the upper byte or word of the product needs to be kept. AF, PF, SF, and ZF are undefined after IMUL.

If you want to multiply a signed byte by a signed word, you must first move the byte into a word location and fill the upper byte of the word with copies of the sign bit. If you move the byte into AL, you can use the 8086 Convert Byte to Word instruction, CBW, to do this. CBW extends the sign bit from AL into all the bits of AH. Once you have converted the byte to a word, you can do word times word IMUL. The result of this multiplication will be in DX and AX.

EXAMPLES (CODING):

```
IMUL BH ; Signed byte in AL times signed byte in BH,
          result in AX

IMUL AX              ; AX times AX, result in DX
                     ; and AX

                     ; Multiplying a signed byte
                     ; by a signed word
MOV CX,MULTIPLIER    ; Load signed word in CX
MOV AL,MULTIPLICAND  ; Load signed byte in AL
CBW                  ; Extend sign of AL into AH
IMUL CX              ; Result in DX and AX
```

EXAMPLES (NUMERICAL):

```
                     ; 69 × 14
                     ; AL = 01000101 = 69 decimal
                     ; BL = 00001110 = 14 decimal
IMUL BL              ; AX = 03C6H = + 966 decimal
                     ; MSB = 0, positive result magnitude
                     ; in true form. SF = 0, CF,OF = 1

                     ; -28 × 59
                     ; AL = 11100100 = - 28 decimal
                     ; BL = 00111011 = + 59 decimal
IMUL BL              ; AX = F98CH = - 1652 decimal
                     ; MSB = 1, negative result magnitude
                     ; in 2's complement. SF,CF,OF = 1
```

IMUL—80186/80188 Only—Integer (Signed) Multiply Immediate—IMUL Destination Register,Source,Immediate Byte or Word

This version of the IMUL instruction functions in the same way as the IMUL instruction described in the preceding section, except that this version allows you to multiply an immediate byte or word by a byte or word in a specified register and put the result in a specified general-purpose register. If the immediate number is a byte, it will be automatically sign-extended to 16 bits. The source of the other operand for the multiplication

an be a register or a memory location specified by any ne of the 24 addressing modes shown in Figure 3-8. since the result is put in a 16-bit general-purpose egister, only the lower 16 bits of the product are saved!

XAMPLE:

```
MUL CX,BX,07H   ; Multiply contents of BX by 07H
                ; CX = lower 16 bits of result
```

IN—Copy Data from a Port—IN Accumulator,Port

The IN instruction will copy data from a port to the AL or AX register. If an 8-bit port is read, the data will go to AL. If a 16-bit port is read, the data will go to AX. The IN instruction has two possible formats, fixed port and variable port.

For the fixed-port type, the 8-bit address of a port is specified directly in the instruction.

EXAMPLES:

```
IN AL,0C8H      ; Input a byte from port 0C8H to AL

IN AX,34H       ; Input a word from port 34H to AX

A_TO_D EQU 4AH
IN AX,A_TO_D    ; Input a word from port 4AH to AX
```

For the variable-port-type IN instruction, the port address is loaded into the DX register before the IN instruction. Since DX is a 16-bit register, the port address can be any number between 0000H and FFFFH. Therefore, up to 65,536 ports are addressable in this mode.

EXAMPLES:

```
MOV DX,0FF78H   ; Initialize DX to point to port
IN AL,DX        ; Input a byte from 8-bit port
                ; 0FF78H to AL

IN AX,DX        ; Input a word from 16-bit port
                ; 0FF78H to AX
```

The variable-port IN instruction has the advantage that the port address can be computed or dynamically determined in the program. Suppose, for example, that an 8086-based computer needs to input data from 10 terminals, each having its own port address. Instead of having a separate procedure to input data from each port, we can write one generalized input procedure and simply pass the address of the desired port to the procedure in DX. The IN instructions do not change any flags.

INC—Increment—INC Destination

The INC instruction adds 1 to a specified register or to a memory location specified in any one of the 24 ways shown in Figure 3-8. AF, OF, PF, SF, and ZF are affected (updated) by this instruction. Note that the carry flag (CF) is not affected. This means that if an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result will be all 0's with no carry.

EXAMPLES:

```
INC BL    ; Add 1 to contents of BL register
INC CX    ; Add 1 to contents of CX register
```

INC BYTE PTR [BX] ; Increment byte in data segment at offset contained in BX. The BYTE PTR directive is necessary to tell the assembler to put in the right code to indicate that a byte in memory, rather than a word, is to be incremented. The instruction essentially says, "Increment the byte pointed to by the contents of BX."

INC WORD PTR [BX] ; Increment the word at offset of [BX] and [BX + 1] in the data segment. In other words, increment the word in memory pointed to by BX.

INC MAX_TEMPERATURE ; Increment byte or word named MAX_TEMPERATURE in data segment. Increment byte if MAX_TEMPERATURE declared with DB. Increment word if MAX_TEMPERATURE declared with DW.

INC PRICES [BX] ; Increment element pointed to by [BX] in array PRICES. Increment a word if PRICES was defined as an array of words with a DW directive. Increment a byte if PRICES was defined as an array of bytes with a DB directive.

> NOTE: The PTR operator is not needed in the last two examples because the assembler knows the type of the operand from the DB or DW used to declare the named data initially.

INT—Interrupt Program Execution—INT Type

The term *type* in the instruction format refers to a number between 0 and 255 which identifies the interrupt. When an 8086 executes an INT instruction, it will:

1. Decrement the stack pointer by 2 and push the flags onto the stack.
2. Decrement the stack pointer by 2 and push the contents of CS onto the stack.
3. Decrement the stack pointer by 2 and push the offset of the next instruction after the INT number instruction on the stack.
4. Get a new value for IP from an absolute memory address of 4 times the type specified in the instruction. For an INT 8 instruction, for example, the new IP will be read from address 00020H.
5. Get a new value for CS from an absolute memory address of 4 times the type specified in the instruc-

tion plus 2. For an INT 8 instruction, for example, the new value of CS will be read from address 00022H.

6. Reset both IF and TF. Other flags are not affected.

Chapter 8 further describes the use of this instruction.

EXAMPLES:

INT 35 ; New IP from 0008CH, new CS from 0008EH

INT 3 ; This is a special form which has the single-byte code of CCH. Many systems use this as a breakpoint instruction. New IP from 0000CH, new CS from 0000EH.

INTO—Interrupt on Overflow

If the overflow flag (OF) is set, this instruction will cause the 8086 to do an indirect far call to a procedure you write to handle the overflow condition. Before doing the call, the 8086 will:

1. Decrement the stack pointer by 2 and push the flags onto the stack.
2. Decrement the stack pointer by 2 and push CS onto the stack.
3. Decrement the stack pointer by 2 and push the offset of the next instruction after the INTO instruction onto the stack
4. Reset TF and IF. Other flags are not affected. To do the call, the 8086 will read a new value for IP from address 00010H and a new value of CS from address 00012H.

Chapter 8 further describes the 8086 interrupt system.

EXAMPLE:

INTO ; Call interrupt procedure if OF = 1

IRET—Interrupt Return

When the 8086 responds to an interrupt signal or to an interrupt instruction, it pushes the flags, the current value of CS, and the current value of IP onto the stack. It then loads CS and IP with the starting address of the procedure which you write for the response to that interrupt. The IRET instruction is used at the end of the interrupt service procedure to return execution to the interrupted program. To do this return, the 8086 copies the saved value of IP from the stack to IP, the stored value of CS from the stack to CS, and the stored value of the flags back to the flag register. Flags will have the values they had before the interrupt, so any flag settings from the procedure will be lost unless they are specifically saved in some way.

NOTE: The RET instruction should not normally be used to return from interrupt procedures because it does not copy the flags from the stack back to the flag register. See Chapter 8 for further discussion of interrupts and the use of IRET.

JA/JNBE—Jump if Above/Jump if Not Below or Equal

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, the zero flag and the carry flag are both 0, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are not 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 JA. JA/JNBE affects no flags. For further explanation of Conditional Jump instructions, see Chapter 4.

EXAMPLES:

```
CMP AX,4371H      ; Compare by subtracting 4371H
                  ; from AX
JA RUN_PRESS      ; Jump to label RUN_PRESS if AX
                  ; above 4371H

CMP AX,4371H      ; Compare (AX − 4371H)
JNBE RUN_PRESS    ; Jump to label RUN_PRESS if AX
                  ; not below or equal to 4371H
```

JAE/JNB/JNC—Jump if Above or Equal/Jump if Not Below/Jump if No Carry

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 0, this instruction will cause execution to jump to a label given in the instruction. If CF is 1, 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 JAE. JAE/JNB/JNC affects no flags. For further explanation of Conditional Jump instructions, see Chapter 4.

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
JC 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
    INC BX        ; 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

IN AL,8FH         ; Read data from port 8FH
SUB AL,30H        ; Subtract minimum value
JZ START_MACHINE  ; Jump to label if result of
                  ; subtraction was 0
```

JG/JNLE—Jump if Greater/Jump if Not Less Than or Equal

These two mnemonics represent the same instruction. The terms *greater* and *less* are used to refer to the relationship of two signed numbers. Greater means more positive. The number 00000111 is greater than the number 11101010, because in signed notation the second number is negative. This instruction is usually used after a Compare instruction. The instruction will cause a jump to a label given in the instruction if the zero flag is 0 and the carry flag is the same as the overflow flag. The destination label must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JG/JNLE instruction. If the jump is not taken, execution simply goes on to the next instruction after the JG or JNLE instruction. JG/JNLE affects no flags.

EXAMPLES:

```
CMP BL,39H      ; Compare by subtracting 39H from BL
JG NEXT_1       ; Jump to label if BL more positive
                ; than 39H
CMP BL,39H      ; Compare by subtracting
                ; 39H from BL
JNLE NEXT_1     ; Jump to label if BL not less than
                ; or equal to 39H
```

JGE/JNL—Jump if Greater Than or Equal/Jump if Not Less Than

These two mnemonics represent the same instruction. The terms *greater* and *less* are used to refer to the relationship of two signed numbers. Greater means more positive. The number 00000111 is greater than the number 11101010, because in signed notation the second number is negative. This instruction is usually used after a Compare instruction. The instruction will cause a jump to a label given in the instruction if the sign flag is equal to the overflow flag. The destination label must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JGE/JNL instruction. If the jump is not taken, execution simply goes on to the next instruction after the JGE or JNL instruction. JGE/JNL affects no flags.

EXAMPLES:

```
CMP BL,39H      ; Compare by subtracting 39H from BL
JGE NEXT_1      ; Jump to label if BL more positive
                ; than 39H.or equal to 39H

CMP BL,39H      ; Compare by subtracting 39H from BL
JNL NEXT_1      ; Jump to label if BL not less than 39H
```

JL/JNGE—Jump if Less Than/Jump if Not Greater Than or Equal

These two mnemonics represent the same instruction. The terms *greater* and *less* are used to refer to the relationship of two signed numbers. Greater means more positive. The number 00000111 is greater than the number 11101010, because in signed notation the second number is negative. This instruction is usually used after a Compare instruction. The instruction will cause a jump to a label given in the instruction if the sign flag is not equal to the overflow flag. The destination label must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JL/JNGE instruction. If the jump is not taken, execution simply goes on to the next instruction after the JL or JNGE instruction. JL/JNGE affects no flags.

EXAMPLES:

```
CMP BL,39H      ; Compare by subtracting 39H from BL
JL AGAIN        ; Jump to label if BL more negative
                ; than 39H

CMP BL,39H      ; Compare by subtracting 39H from B
JNGE AGAIN      ; Jump to label if BL not more positiv
                ; than 39H or BL not equal to 39H
```

JLE/JNG—Jump if Less Than or Equal/Jump if Not Greater

These two mnemonics represent the same instruction The terms *greater* and *less* are used to refer to th relationship of two signed numbers. Greater mean more positive. The number 00000111 is greater tha the number 11101010, because in signed notation th second number is negative. This instruction is usuall used after a Compare instruction. The instruction wi cause a jump to a label given in the instruction if th zero flag is set, or if the sign flag is not equal to th overflow flag. The destination label must be in the rang of −128 bytes to +127 bytes from the address of th instruction after the JLE/JNG instruction. If the jum is not taken, execution simply goes on to the nex instruction after the JLE/JNG instruction. JLE/JNG affects no flags.

EXAMPLES:

```
CMP BL,39H      ; Compare by subtracting 39H from BL
JLE NXT_1       ; Jump to label if BL more negative
                ; than 39H or equal to 39H

CMP BL,39H      ; Compare by subtracting 39H from BL
JNG PRINTER     ; Jump to label if BL not more
                ; positive than 39H
```

JMP—Unconditional Jump to Specified Destination

This instruction will always cause the 8086 to fetch its next instruction from the location specified in the instruction rather than from the next location after the JMP instruction. If the destination is in the same code segment as the JMP instruction, then only the instruction pointer will be changed to get to the destination location. This is referred to as a *near jump*. If the destination for the jump instruction is in a segment with a name different from that of the segment containing the JMP instruction, then both the instruction pointer and the code segment register contents will be changed to get to the destination location. This is referred to as a *far jump*. The JMP instruction affects no flags. Refer to Chapter 4 for a detailed discussion of the different forms of the unconditional JMP instruction.

EXAMPLES:

```
JMP CONTINUE ; Fetch next instruction from address
             at label CONTINUE. If the label is in the same segment,
             an offset coded as part of the instruction will be added
             to the instruction pointer to produce the new fetch
             address. If the label is in another segment, then IP and
             CS will be replaced with values coded in as part of the
```

instruction. This type of jump is referred to as direct because the displacement of the destination or the destination itself is specified directly in the instruction.

JMP BX ; Replace the contents of IP with the contents of BX. BX must first be loaded with the offset of the destination instruction in CS. This is a near jump. It is also referred to as an indirect jump because the new value for IP comes from a register rather than from the instruction itself, as in a direct jump.

JMP WORD PTR [BX] ; Replace IP with a word from a memory location pointed to by BX in DS. This is an indirect near jump.

JMP DWORD PTR [SI] ; Replace IP with a word pointed to by SI in DS. Replace CS with a word pointed to by SI + 2 in DS. This is an indirect far jump.

JNA—See Heading JBE

JNAE—See Heading JB

JNB—See Heading JAE

JNBE—See Heading JA

JNC—See Heading JAE

JNE/JNZ—Jump if Not Equal/Jump if Not Zero

These two mnemonics represent the same instruction. If the zero flag is 0, then this instruction will cause execution to jump to a label given in the instruction. If the zero flag is 1, then execution will simply go on to the next instruction after JNE or JNZ. The destination label for the JNE/JNZ instruction must be in the range of −128 to +127 bytes from the address of the instruction after the JNE/JNZ instruction. JNE/JNZ affects no flags.

EXAMPLES:

```
NXT: IN AL,0F8H      ; Read data value from port
     CMP AL,72       ; Compare (AL-72)
     JNE NXT         ; Jump to NXT if AL ≠ 72
     IN AL,0F9H      ; Read next port when
                     ;  AL = 72

     MOV BX,2734H    ; Load BX as counter
NXT_1: ADD AX,0002H  ; Add count factor to AX
     DEC BX          ; Decrement BX
     JNZ NXT_1:      ; Repeat until BX = 0
```

JNG—See Heading JLE

JNGE—See Heading JL

JNL—See Heading JGE

JNLE—See Heading JG

JNO—Jump if No Overflow

The overflow flag will be set if the result of some signed arithmetic operation is too large to fit in the destination register or memory location. The JNO instruction will cause the 8086 to jump to a destination given in the instruction if the overflow flag is not set. The destination must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JNO instruction. If the overflow flag is set, execution will simply continue with the next instruction after JNO. JNO affects no flags.

EXAMPLE:

```
     ADD AL,BL     ; Add signed bytes in AL and BL
     JNO DONE      ; Process done if no overflow
     MOV AL,00H    ; Else load error code in AL
DONE: OUT 24H,AL   ; Send result to display
```

JNP/JPO—Jump if No Parity/Jump if Parity Odd

If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is odd, then the parity flag will be 0. The JNP/JPO instruction will cause execution to jump to a specified destination address if the parity flag is 0. The destination address must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JNP/JPO instruction. If the parity flag is set, execution will simply continue on to the instruction after the JNP/JPO instruction. The JNP/JPO instruction affects no flags.

EXAMPLE:

```
IN AL,0F8H      ; Read ASCII character from UART
OR AL,AL        ; Set flags
JPO ERROR1      ; Even parity expected, send error
                ; message if parity found odd
```

JNS—Jump if Not Signed (Jump if Positive)

This instruction will cause execution to jump to a specified destination if the sign flag is 0. Since a 0 in the sign flag indicates a positive signed number, you can think of this instruction as saying "jump if positive." If the sign flag is set, indicating a negative signed result, execution will simply go on to the next instruction after JNS. The destination for the jump must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JNS. JNS affects no flags.

EXAMPLE:

```
DEC AL          ; Decrement counter
JNS REDO        ; Jump to label REDO if counter has not
                ; decremented to FFH
```

JNZ—See Heading JNE

JO—Jump if Overflow

The JO instruction will cause the 8086 to jump to a destination given in the instruction if the overflow flag

is set. The overflow flag will be set if the magnitude of the result produced by some signed arithmetic operation is too large to fit in the destination register or memory location. The destination for the JO instruction must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JO instruction. If the overflow flag is not set, execution will simply continue with the next instruction after JO. JO affects no flags.

EXAMPLE:

```
ADD AL,BL     ; Add signed bytes in AL and BL
JO ERROR      ; Jump to label ERROR if overflow
              ; from add
MOV SUM,AL    ; Else put result in memory location
              ; named SUM
```

JP/JPE—Jump if Parity/Jump if Parity Even

If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is even, then the parity flag will be set. If the parity flag is set, the JP/JPE instruction will cause execution to jump to a specified destination address. If the parity flag is 0, execution will simply continue on to the instruction after the JP/JPE instruction. The destination address must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JP/JPE instruction. The JP/JPE instruction affects no flags.

EXAMPLE:

```
IN AL,F8H     ; Read ASCII character from UART
OR AL,AL      ; Set flags
JPE ERROR2    ; Odd parity expected, send error
              ; message if parity found even
```

JPE—See Heading JP

JPO—See Heading JNP

JS—Jump if Signed (Jump if Negative)

This instruction will cause execution to jump to a specified destination if the sign flag is set. Since a 1 in the sign flag indicates a negative signed number, you can think of this instruction as saying "jump if negative" or "jump if minus." If the sign flag is 0, indicating a positive signed result, execution will simply go on to the next instruction after JS. The destination for the jump must be in the range of −128 bytes to +127 bytes from the address of the instruction after the JS. JS affects no flags.

EXAMPLE:

```
ADD BL,DH     ; Add signed byte in DH to signed
              ; byte in BL
JS TOO_COLD   ; Jump to label TOO_COLD if result
              ; of addition is negative number
```

JZ—See Heading JE

LAHF—Copy Low Byte of Flag Register to AH

The lower byte of the 8086 flag register is the same as the flag byte for the 8085. LAHF copies these 8085 equivalent flags to the AH register. They can then be pushed onto the stack along with AL by a PUSH AX instruction. An LAHF instruction followed by a PUSH AX instruction has the same effect as the 8085 PUSH PSW instruction. The LAHF 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 Figure 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.
```

LODS/LODSB/LODSW—Load String Byte into AL or Load String Word into AX

This instruction copies a byte from a string location pointed to by SI to AL, or a word from a string location pointed to by SI to AX. If the direction flag is cleared (0), SI will automatically be incremented to point to the next element of the string. For a string of bytes, SI will be incremented by 1. For a string of words, SI will be incremented by 2. If the direction flag (DF) is set (1), SI will be automatically decremented to point to the next string element. For a byte string, SI will be decremented by 1, and for a word string, SI will be decremented by 2. LODS affects no flags.

EXAMPLE:

```
CLD                          ; Clear direction flag so SI
                             ; is autoincremented

MOV SI, OFFSET SOURCE_STRING
                             ; Point SI at start
                             ; of string
LODS SOURCE_STRING           ; Copy byte or word from
                             ; string to AL or AX
```

NOTE: The assembler uses the name of the string to determine whether the string is of type byte or type word. Instead of using the string name to do this, you can use the mnemonic LODSB to tell the assembler that the string is of type byte or the mnemonic LODSW to tell the assembler that the string is of type word.

LOOP—Jump to Specified Label if CX ≠ 0 after Autodecrement—LOOP Label

This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX is not 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the autodecrement, execution will simply go on to the next instruction after LOOP. The destination address for the jump must be in the range of −128 bytes to +127 bytes from the address of the instruction after the LOOP instruction. LOOP affects no flags. See Chapter 4 for further discussion and examples of the LOOP instruction.

EXAMPLE:

```
        MOV BX, OFFSET PRICES
                             ; Point BX at
                             ; first element in array
        MOV CX,40            ; Load CX with number of
                             ; elements in array
NEXT:   MOV AL,[BX]          ; Get element from array
        ADD AL,07H           ; Add correction factor
        DAA                  ; Decimal adjust result
```

```
MOV [BX],AL      ; Put result back in array
INC BX
LOOP NEXT        ; Repeat until all elements
                 ; adjusted
```

LOOPE/LOOPZ—Loop While CX ≠ 0 and ZF = 1

LOOPE and LOOPZ are two mnemonics for the same instruction. This instruction is used to repeat a group of instructions some number of times or until the zero flag becomes 0. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX ≠ 0 and ZF = 1, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the autodecrement or if ZF = 0, execution will simply go on to the next instruction after LOOPE/LOOPZ. In other words, the two ways to exit the loop are CX = 0 or ZF = 0. The destination address for the jump must be in the range of −128 bytes to +127 bytes from the address of the instruction after the LOOPE/LOOPZ instruction. LOOPE/LOOPZ affects no flags. See Chapter 4 for further discussion and examples of the LOOPE/LOOPZ instruction.

EXAMPLE:

```
        MOV BX,OFFSET ARRAY ; Point BX to just
        DEC BX              ; before start of array
        MOV CX,100          ; Put number of array
                            ; elements in CX
NEXT:   INC BX              ; Point to next
                            ; element in array
        CMP [BX],0FFH       ; Compare array
                            ; element with FFH
        LOOPE NEXT
```

NOTE: The next element is checked if the element equals FFH and the element was not the last one in the array. If CX = 0 and ZF = 1 on exit, all elements were equal to FFH. If CX ≠ 0 on exit from the loop, then BX points to the first element that was not FFH. If CX = 0 and ZF = 0 on exit, then the last element was not FFH.

LOOPNE/LOOPNZ—Loop While CX ≠ 0 and ZF = 0

LOOPNE and LOOPNZ are two mnemonics for the same instruction. This instruction is used to repeat a group of instructions some number of times or until the zero flag becomes a 1. The number of times the instruction sequence is to be repeated is loaded into the count register CX. Each time the LOOPNE/LOOPNZ instruction executes, CX is automatically decremented by 1. If CX ≠ 0 and ZF = 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the autodecrement or if ZF = 1, execution will simply go on to the next instruction after LOOPNE/LOOPNZ. In other words, the two ways to exit the loop are CX = 0 and ZF = 1. The destination address for the jump must be in the range of −128 bytes to +127 bytes from the address of the instruction after the LOOPNE/LOOPNZ instruction. LOOPNE/LOOPNZ affects no flags. See Chapter 4 for further discussion and examples of the LOOPNE/LOOPNZ instruction.

EXAMPLE:

```
        MOV BX,OFFSET ARRAY ; Point BX to just
        DEC BX              ; before start of array
        MOV CX,100          ; Put number of array
                            ; elements in CX
NEXT:   INC BX              ; Point to next
                            ; element in array
        CMP [BX],0DH        ; Compare array
                            ; element with 0DH
        LOOPNE NEXT
```

NOTE: When the LOOPNE instruction executes, CX will be decremented by 1. If CX ≠ 0 and ZF = 0, execution will go to the label NEXT. If CX = 0 or ZF = 1, execution will go on to the next instruction after LOOPNE. If CX = 0 and ZF = 0 on exit, 0DH was not found in the array. If CX ≠ 0 on exit from the loop, then BX points to the first element which contains 0DH. If CX = 0 and ZF = 1 on exit from the loop, the last array element was 0DH.

LOOPNZ—See Heading LOOPNE

LOOPZ—See Heading LOOPE

MOV—Copy a Word or Byte—MOV Destination,Source

The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number. The source and destination in an instruction cannot both be memory locations. The source and destination in a MOV instruction must both be of type byte, or they must both be of type word. MOV instructions do not affect any flags.

EXAMPLES:

```
MOV CX,037AH     ; Put the immediate number
                 ; 037AH in CX

MOV BL,[437AH]   ; Copy byte in DS at offset
                 ; 437AH to BL

MOV AX,BX        ; Copy contents of register BX to AX

MOV DL,[BX]      ; Copy byte from memory at [BX]
                   to DL
                 ; BX contains offset of byte in DS

MOV DS,BX        ; Copy word from BX to DS register
```

MOV RESULTS[BP],AX; Copy AX to two memory locations—AL to the first location, AH to the second. EA of the first memory location is the sum of the displacement represented by RESULTS and contents of BP. Physical address = EA + SS.

MOV CS:RESULTS[BP],AX ; Same as the above instruction, but physical address = EA + CS because of the segment override prefix CS.

MOVS/MOVSB/MOVSW—Move String Byte or String Word—MOVS Destination String Name,Source String Name

This instruction copies a byte or a word from a location in the data segment to a location in the extra segment. The offset of the source byte or word in the data segment must be in the SI register. The offset of the destination in the extra segment must be contained in the DI register. For multiple-byte or multiple-word moves, the number of elements to be moved is put in the CX register so that it can function as a counter. After the byte or word is moved, SI and DI are automatically adjusted to point to the next source and the next destination. If the direction flag is 0, then SI and DI will be incremented by 1 after a byte move and incremented by 2 after a word move. If the DF is a 1, then SI and DI will be decremented by 1 after a byte move and decremented by 2 after a word move. MOVS affects no flags.

When using the MOVS instruction, you must in some way tell the assembler whether you want to move a string as bytes or as words. There are two ways to do this. The first way is to indicate the names of the source and destination strings in the instruction, as, for example, MOVS STRING_DUMP,STRING_CREATE. The assembler will code the instruction for a byte move if STRING_DUMP and STRING_CREATE were declared with a DB. It will code the instruction for a word move if they were declared with a DW. Note that this reference to the source and destination strings does not load SI and DI. This must be done with separate instructions. The second way to tell the assembler whether to code the instruction for a byte or word move is to add a "B" or a "W" to the MOVS mnemonic. MOVSB, for example, says move a string as bytes. MOVSW says move a string as words.

EXAMPLE:

```
MOV SI,OFFSET SOURCE_STRING
                        ; Load offset of start of source
                        ; string in DS into SI
MOV DI,OFFSET DESTINATION_STRING
                        ; Load offset of start of
                          destination
                        ; string in ES into DI
CLD                     ; Clear direction flag to auto-
                        ; increment SI & DI after move
MOV CX,04H              ; Load length of string into CX
                        ; as counter
REP MOVSB               ; Decrement CX and copy
                        ; string bytes until CX = 0
```

After the move, SI will be 1 greater than the offset of the last byte in the source string. DI will be 1 greater than the offset of the last byte in the destination string. CX will be 0.

MUL—Multiply Unsigned Bytes or Words—MUL Source

This instruction multiplies an unsigned byte from some source times an unsigned byte in the AL register or an unsigned word from some source times an unsigned word in the AX register. The source can be a register or a memory location specified by any one of the 24 addressing modes shown in Figure 3-8. When a byte is multiplied by the contents of AL, the result (product) is put in AX. A 16-bit destination is required because the result of multiplying an 8-bit number by an 8-bit number can be as large as 16 bits. The most significant byte of the result is put in AH, and the least significant byte of the result is put in AL. When a word is multiplied by the contents of AX, the product can be as large as 32 bits. The most significant word of the result is put in the DX register, and the least significant word of the result is put in the AX register. If the most significant byte of a 16-bit result or the most significant word of a 32-bit result is 0, CF and OF will both be 0's. Checking these flags, then, allows you to detect and perhaps discard unnecessary leading 0's in a result. AF, PF, SF, and ZF are undefined after a MUL instruction.

If you want to multiply a byte by a word, you must first move the byte to a word location such as an extended register and fill the upper byte of the word with all 0's.

> NOTE: You cannot use the 8086 Convert Byte to Word instruction, CBW, to do this. The CBW instruction fills the upper byte of AX with copies of the MSB of AL. If the number in AL is 80H or greater, CBW will fill the upper half of AX with 1's instead of with 0's. Once you get the byte converted correctly to a word with 0's in the upper byte, you can then do a word times word multiply. The 32-bit result will be in DX and AX.

EXAMPLES:

```
MUL BH                  ; AL times BH, result in AX

MUL CX                  ; AX times CX, result high word
                          in DX,
                        ; low word in AX

MUL BYTE PTR [BX]       ; AL times byte in DS pointed
                        ; to by [BX]
```

MUL CONVERSION_FACTOR[BX] ; Multiply AL times byte at effective address CONVERSION_FACTOR[BX] if it was declared as type byte with DB. Multiply AX times word at effective address CONVERSION_FACTOR[BX] if it was declared as type word with DW.

; Example showing a byte multiplied by a word

```
MOV AX,MULTIPLICAND_16  ; Load 16-bit
                        ; multiplicand into AX
MOV CL,MULTIPLIER_8     ; Load 8-bit multiplier
                        ; into CL
MOV CH,00H              ; Set upper byte of CX
                        ; to all 0's
MUL CX                  ; AX times CX, 32-bit
                        ; result in DX and AX
```

NEG—Form 2's Complement—NEG Destination

This instruction replaces the number in a destination with the 2's complement of that number. The destination can be a register or a memory location specified by any one of the 24 addressing modes shown in Figure 3-8. This instruction forms the 2's complement by subtracting the original word or byte in the indicated destination from zero. 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 Figure 4-27*a*. 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 Figure 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 Figure 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):

```
                    ; CX = 00111101 10100101
ORCX,0FF00H         ; 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
```

OUT—Output a Byte or Word to a Port—OUT Port,Accumulator AL or AX

The OUT instruction copies a byte from AL or a word from AX to the specified port. The OUT instruction has two possible forms, fixed port and variable port.

For the fixed-port form, the 8-bit port address is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

EXAMPLES:

```
OUT 3BH,AL    ; Copy the contents of AL to port 3BH

OUT 2CH,AX    ; Copy the contents of AX to port 2CH
```

For the variable-port form of the OUT instruction, the contents of AL or AX will be copied to the port at an address contained in DX. Therefore, the DX register must always be loaded with the desired port address before this form of the OUT instruction is used. The advantage of the variable-port form of addressing is described in the discussion of the IN instruction. The OUT instruction does not affect any flags.

EXAMPLES:

```
MOV DX,0FFF8H   ; Load desired port address in DX
OUT DX,AL       ; Copy contents of AL to port FFF8H
OUT DX,AX       ; Copy contents of AX to port FFF8H
```

POP—POP Destination

The POP instruction copies a word from the stack location pointed to by the stack pointer to a destination specified in the instruction. The destination can be a general-purpose register, a segment register, or a memory location. The data in the stack is not changed. After the word is copied to the specified destination, the stack pointer is automatically incremented by 2 to point to the next word on the stack. No flags are affected by the POP instruction.

NOTE: POP CS is illegal.

EXAMPLES:

```
POP DX              ; Copy a word from top of stack to DX
                    ; Increment SP by 2

POP DS              ; Copy a word from top of stack to DS
                    ; Increment SP by 2

POP TABLE [BX]      ; Copy a word from top of stack to
                    ; memory in DS with EA =
                    ; TABLE + [BX]
```

POPF—Pop Word from Top of Stack to Flag Register

This instruction copies a word from the two memory locations at the top of the stack to the flag register and increments the stack pointer by 2. The stack segment register and the word on the stack are not affected. All flags are affected.

PUSH—PUSH Source

The PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment where the stack pointer then points. The source of the word can be a general-purpose register, a segment register, or memory. The stack segment register and the stack pointer must be initialized before this instruction can be used. PUSH can be used to save data on the stack so that it will not be destroyed by a procedure. It can also be used to put data on the stack so that a procedure can access it there as needed. No flags are affected by this instruction. Refer to Chapter 5 for further discussion of the stack and the PUSH instruction.

EXAMPLES:

```
PUSH BX             ; Decrement SP by 2, copy BX
                      to stack

PUSH DS             ; Decrement SP by 2, copy DS
                      to stack

PUSH AL             ; Illegal, must push a word

PUSH TABLE [BX]     ; Decrement SP by 2, copy word
                    ; from memory in DS at
                    ; EA = TABLE + [BX] to stack
```

PUSHF—Push Flag Register on the Stack

This instruction decrements the stack pointer by 2 and copies the word in the flag register to the memory location(s) pointed to by the stack pointer. The stack segment register is not affected. No flags are changed.

RCL—Rotate Operand Around to the Left through CF—RCL Destination,Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to the left. The

operation is circular because the MSB of the operand is rotated into the carry flag and the bit in the carry flag is rotated around into the LSB of the operand. See the following diagram.

The "C" in the middle of the mnemonic should help you remember that CF is in the rotated loop and help distinguish this instruction from the ROL instruction. For multibit rotates, CF will contain the bit most recently rotated out of the MSB.

The destination operand can be in a register or in a memory location specified by any one of the 24 addressing modes shown in Figure 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 into 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.

RCL affects only CF and OF. After RCL, CF will contain the bit most recently rotated out of the MSB. OF will be a 1 after a single-bit RCL if the MSB was changed by the rotate. OF is undefined after a multibit rotate.

The RCL instruction is a handy way to move CF into the LSB of a register or memory location to save it after addition or subtraction.

EXAMPLES (SYNTAX):

```
RCL DX,1            ; Word in DX 1 bit left, MSB to
                    ; CF, CF to LSB

MOV CL,4            ; Load number of bit positions to
                    ; rotate into CL
RCL SUM[BX],CL      ; Rotate byte or word at effective
                    ; address SUM[BX] 4 bits left
                    ; Original bit 4 now in CF, original
                    ; CF now in bit 3
```

EXAMPLES (NUMERICAL):

```
                    ; CF = 0, BH = 10110011
RCL BH,1            ; Result: BH = 01100110
                    ; CF = 1, OF = 1 because MSB changed

                    ; CF = 1, AX = 00011111 10101001

MOV CL,2            ; Load CL for rotating 2 bit positions
RCL AX,CL           ; Result: CF = 0, OF undefined
                    ; AX = 01111110 10100110
```

RCR—Rotate Operand Around to the Right through CF—RCR Destination,Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to the right. The operation is circular because the LSB of the operand is rotated into the carry flag and the bit in the carry flag is rotated around into the MSB of the operand. See the following diagram.

The "C" in the middle of the mnemonic should help you remember that CF is in the rotated loop and should help distinguish this instruction from the ROR instruction. For multibit rotates, CF will contain the bit most recently rotated 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 Figure 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 into 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.

RCR affects only CF and OF. After RCR, CF will contain the bit most recently rotated out of the MSB. OF will be a 1 after a single-bit RCR if the MSB was changed by the rotate. OF will be undefined after multibit rotates.

EXAMPLES (CODING):

```
RCR BX,1            ; Word in BX right 1 bit
                    ; CF to MSB, LSB to CF

MOV CL,04H          ; Load CL for rotating
                    ; 4 bit positions
RCR BYTE PTR [BX]   ; Rotate byte at offset [BX] in
                    ; DS 4 bit positions right
                    ; CF = original bit 3. Bit 4
                    ; = original CF
```

EXAMPLES (NUMERICAL):

```
                    ; CF = 1, BL = 00111000
RCR BL,1            ; Result: BL = 10011100, CF = 0
                    ; OF = 1 because MSB
                    ; changed to 1

                    ; CF = 0, WORD PTR [BX]
                    ; = 01011110 00001111
MOV CL,02H          ; Load CL for rotate 2 bit
                    ; positions
```

```
RCR WORD PTR [BX], CL   ; Rotate word in DS at
                        ; offset [BX] 2 bits right
                        ; CF = original bit 1.
                        ; Bit 14 = original CF
                        ; WORD PTR [BX] =
                        ; 10010111 10000011
```

REP/REPE/REPZ/REPNE/REPNZ—(Prefix) Repeat String Instruction until Specified Conditions Exist

REP is a prefix which is written before one of the string instructions. It will cause the CX register to be decremented and the string instruction to be repeated until CX = 0. The instruction REP MOVSB, for example, will continue to copy string bytes until the number of bytes loaded into CX has been copied.

REPE and REPZ are two mnemonics for the same prefix. They stand for Repeat if Equal and Repeat if Zero, respectively. You can use whichever prefix makes the operation clearer to you in a given program. REPE or REPZ is often used with the Compare String instruction or with the Scan String instruction. REPE or REPZ will cause the string instruction to be repeated as long as the compared bytes or words are equal (ZF = 1) *and* CX is not yet counted down to zero. In other words, there are two conditions that will stop the repetition: CX = 0 or string bytes or words *not* equal.

EXAMPLE:

```
REPE CMPSB ; Compare string bytes until end of string
           ; or until string bytes not equal. See the discussion of the
           ; CMPS instruction for a more detailed example of the use
           ; of REPE.
```

REPNE and REPNZ are also two mnemonics for the same prefix. They stand for Repeat if Not Equal and Repeat if Not Zero, respectively. REPNE or REPNZ is often used with the Scan String instruction. REPNE or REPNZ will cause the string instruction to be repeated until the compared bytes or words are equal (ZF = 1) *or* until CX = 0 (end of string).

EXAMPLE:

```
REPNE SCASW ; Scan a string of words until a word in
            ; the string matches the word in AX or until all of the
            ; string has been scanned. See the discussion of SCAS
            ; for a more detailed example of the use of this prefix
```

The string instruction used with the prefix determines which flags are affected. See the individual instructions for this information. Also see Chapter 5 for further examples of the REP instruction with string instructions.

NOTE: Interrupts should be disabled when multiple prefixes are used, such as LOCK, segment override, and REP with string instructions on the 8086/8088. This is because, during an interrupt response, the 8086 can remember only the prefix just before the string instruction. The 80186, 80286, etc., will remember all the prefixes and start up correctly after an interrupt during a string instruction.

RET—Return Execution from Procedure to Calling Program

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction which was used to call the procedure. If the procedure is a near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the instruction pointer with a word from the top of the stack. The word from the top of the stack is the offset of the next instruction after the CALL. This offset was pushed onto the stack as part of the operation of the CALL instruction. The stack pointer will be incremented by 2 after the return address is popped off the stack.

If the procedure is a far procedure (in a different code segment from the CALL instruction which calls it), then the instruction pointer will be replaced by the word at the top of the stack. This word is the offset part of the return address put there by the CALL instruction. The stack pointer will then be incremented by 2. The code segment register is then replaced with a word from the new top of the stack. This word is the segment base part of the return address that was pushed onto the stack by a far call operation. After the code segment word is popped off the stack, the stack pointer is again incremented by 2.

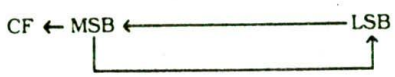
A RET instruction can be followed by a number, for example, RET 6. In this case the stack pointer will be incremented by an additional six addresses after the IP or the IP and CS are popped off the stack. This form is used to increment the stack pointer over parameters passed to the procedure on the stack.

The RET instruction affects no flags.

Please refer to Chapter 5 for further discussion of the CALL and RET instructions.

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 Figure 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,1              ; 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],1      ; 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
```

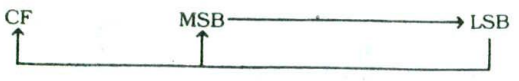
EXAMPLES (NUMERICAL):

```
                      ; CF = 0, BH = 10101110
ROL BH,1              ; Result: CF,OF = 1, BH = 01011101

                      ; BX = 01011100 11010011
                      ; CL = 8, set for 8-bit rotate
ROL BX,CL             ; 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 Figure 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).

EXAMPLES (SYNTAX):

```
ROR BL,1              ; Rotate all bits in BL right 1 bit position
                      ; LSB to MSB and to CF

MOV CL,08H            ; Load CL with number of bit
                      ; positions to be rotated
ROR WORD PTR [BX],CL  ; Rotate word in DS at offset
                      ; [BX] 8 bit positions right
                      ; (swap bytes in word)
```

EXAMPLES (NUMERICAL):

```
                      ; CF = 0, BX = 00111011 01110101
ROR BX,1              ; Rotate all bits of BX 1 bit position right
                      ; CF = 1, BX = 10011101 10111010

                      ; CF = 0, AL = 10110011, OF = 1
MOV CL,04H            ; Load CL for rotate 4 bit positions
ROR AL,CL             ; Rotate all bits of AL 4 bits right
                      ; CF = 0, AL = 00111011, OF = ?
```

SAHF—Copy AH Register to Low Byte of Flag Register

The lower byte of the 8086 flag regsiter corresponds exactly to the 8085 flag byte. SAHF replaces this 8085 equivalent flag byte with a byte from the AH register. SAHF is used with the POP AX instruction to simulate the 8085 POP PSW instruction. As described under the heading LAHF, an 8085 PUSH PSW instruction will be translated to an LAHF—PUSH AX sequence to run on an 8086. An 8085 POP PSW .nstruction will be translated to a POP AX—SAHF sequence to run on an 8086. SAHF changes the flags in the lower byte of the flag register.

SAL/SHL—Shift Operand Bits Left, Put Zero in LSB(s)—SAL/SHL Destination,Count

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination some number of bit positions to the left. As a bit is shifted out of the LSB position, a 0 is put in the LSB position. The MSB will be shifted into CF. In the case of multiple-bit shifts, CF will contain the bit most recently shifted in from the MSB. Bits shifted into CF previously will be lost. See the following diagram.

The destination operand can be a byte or a word. It can be in a register or in a memory location specified by any one of the 24 addressing modes shown in Figure 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 1 bit position, the desired number of shifts is loaded into the CL register, and CL is put in the count position of the instruction. The advantage of using the CL register is that the number of shifts can be dynamically calculated as the program executes.

> 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 as follows: CF contains the bit most recently shifted in from MSB. For a count of one, OF will be 1 if CF and the current MSB are not the same. For multiple-bit shifts, OF is undefined. SF and ZF will be updated to reflect the condition of the destination. PF will have meaning only for an operand in AL. AF is undefined.

The SAL or SHL instruction can also be used to multiply an unsigned binary number by a power of 2. Shifting a binary number one bit position to the left and putting a 0 in the LSB multiplies the number by 2. Shifting the number two bit positions multiplies it by 4. Shifting the number three bit positions multiplies it by 8, etc. For this specific type of multiply, the SAL method is faster than using MUL, but you must make sure that the result does not become too large for the destination.

EXAMPLES (SYNTAX):

```
SAL BX,1                ; Shift word in BX 1 bit position left,
                        ; 0 in LSB

MOV CL,02H              ; Load desired number of shifts in CL
SAL BP,CL               ; Shift word in BP left (CL) bit
                        ; positions, 0's in 2 LSBs

SAL BYTE PTR [BX],1     ; Shift byte in DS at offset [BX]
                        ; 1 bit position left, 0 in LSB

                        ; Example of SAL instruction's
                        ; use to help pack BCD
IN AL,COUNTER_DIGIT     ; Unpacked BCD from
                        ; counter to AL
MOV CL,04H              ; Set count for 4 bit positions
SAL AL,CL               ; Shift BCD to upper nibble,
                        ; 0's in lower nibble. Ready to OR
                        ; another BCD digit into
                        ; lower nibble of AL
```

EXAMPLE (NUMERICAL):

```
                ; CF = 0, BX = 11100101 11010011
SAL BX,1        ; Shift BX register contents 1 bit position left
                ; CF = 1, BX = 11001011 10100110
                ; OF = 0, PF = ?, SF = 1, ZF = 0
```

SAR—Shift Operand Bits Right, New MSB = Old MSB—SAR Destination,Count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. In other words, the sign bit is copied into the MSB. The LSB will be shifted into CF. In the case of multiple bit shifts, 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. It can be in a register or in a memory location specified by any one of the 24 addressing modes shown in Figure 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 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 the same. After a multibit SAR, OF will be 0. SF and ZF will be updated to show the condition of the destination. PF

will have meaning only for an 8-bit destination. AF will be undefined after SAR.

The SAR instruction can be used to divide a signed byte or word by a power of 2. Shifting a binary number right one bit position divides it by 2. Shifting a binary number right two bit positions divides it by 4. Shifting it right three positions divides it by 8, etc. For unsigned numbers, a 0 is put in the MSB after the old MSB is shifted right. (See discussion of SHR instruction.) For signed binary numbers, the sign bit must be copied into the new MSB as the old sign bit is shifted right. This is necessary to retain the correct sign in the result. SAR shifts the operand right and copies the sign bit into the MSB as required for this operation. Using SAR to do a divide by 2, however, gives slightly different results than using the IDIV instruction to do the same job. IDIV always truncates a signed result toward 0. For example, an IDIV of 7 by 2 gives 3, and an IDIV of −7 by 2 gives −3. SAR always truncates a result in a downward direction. Using SAR to divide 7 by 2 gives 3, but using SAR to divide −7 by 2 gives −4.

EXAMPLES (SYNTAX):

```
SAR DI,1 ; Shift word in DI one bit position right,
         ; new MSB = old MSB

MOV CL,02H              ; Load desired number of
                        ; shifts in CL
SAR WORD PTR [BP],CL    ; Shift word at offset [BP]
                        ; in stack segment right
                        ; two bit positions. Two MSBs
                        ; are now copies of
                        ; original MSB
```

EXAMPLES (NUMERICAL):

```
         ; AL = 00011101 = + 29 decimal CF = 0
SAR AL,1 ; Shift signed byte in AL right
         ; to divide by 2
         ; AL = 00001110 = + 14 decimal. CF = 1,
         ; OF = 0, PF = 0, SF = 0, ZF = 0

         ; BH = 11110011 = − 13 decimal
SAR BH, 1; Shift signed byte in BH right to
         ; divide by 2
         ; BH = 11111001 = − 7 decimal. CF = 1,
         ; OF = 0, PF = 1, SF = 1, ZF = 0
```

SBB—Subtract with Borrow—SBB Destination,Source

SUB—Subtract—SUB Destination,Source

These instructions subtract the number in the indicated source from the number in the indicated destination and put the result in the indicated destination. For subtraction, the carry flag (CF) functions as a borrow flag. The carry flag will be set after a subtraction if the number in the specified source is larger than the number in the specified destination. In other words, the carry/borrow flag will be set if a borrow was required to do the subtraction. The Subtract instruction, SUB, subtracts just the contents of the specified source from the contents of the specified destination. The Subtract with Borrow instruction, SBB, subtracts the contents of the source and the contents of CF from the contents of the indicated destination. The source may be an immediate number, a register, or a memory location specified by any of the 24 addressing modes shown in Figure 3-8. The destination can also be a register or a memory location. However, the source and the destination cannot both be memory locations in an instruction. The source and the destination must both be of type byte or both be of type word. If you want to subtract a byte from a word, you must first move the byte to a word location such as a 16-bit register and fill the upper byte of the word with 0's. AF, CF, OF, PF, SF, and ZF are updated by the SUB instruction.

EXAMPLES (SYNTAX):

```
SUB CX,BX       ; CX − BX. Result in CX

SBB CH,AL       ; Subtract contents of AL and
                ; contents of CF from
                ; contents of CH. Result in CH

SUB AX,3427H    ; Subtract immediate number
                ; 3427H from AX

SBB BX,[3427H]  ; Subtract word at displacement
                ; 3427H in DS and contents
                ; of CF from BX
```

SUB PRICES[BX],04H ; Subtract 04 from byte at effective address PRICES[BX] if PRICES declared with DB. Subtract 04 from word at effective address PRICES[BX] if PRICES declared with DW.

SBB CX,TABLE[BX] ; Subtract word from effective address TABLE[BX] and status of CF from CX.

SBB TABLE[BX],CX ; Subtract CX and status of CF from word in memory at effective address TABLE[BX].

EXAMPLES (NUMERICAL):

```
         ; Example subtracting unsigned numbers
         ; CL = 10011100 = 156 decimal
         ; BH = 00110111 = 55 decimal
SUB CL, BH ; Result: CF,AF,SF,ZF = 0, OF,PF = 1
         ; CL = 01100101 = 101 decimal

         ; First example subtracting signed numbers
         ; CL = 00101110 = + 46 decimal
         ; BH = 01001010 = + 74 decimal
SUB CL, BH ; Results: AF,ZF = 0, PF = 1
         ; CL = 11100100 = − 28 decimal
         ; CF = 1, borrow required
         ; SF = 1, result negative
         ; OF = 0, magnitude of result fits in 7 bits
```

```
                ; Second example subtracting
                ; signed numbers
                ; CL = 10100001 = - 95 decimal
                ; BH = 01001100 = + 76 decimal
SUB CL, BH      ; Results: CF,ZF = 0, AF,PF = 1
                ; CL = 01010101 = + 85 decimal
                ; SF = 0, result positive !
                ; OF = 1, invalid result
```

The overflow flag being set indicates that the magnitude of the expected result, −171 decimal, is too large to fit in the 7 bits used for the magnitude in an 8-bit signed number. If the Interrupt on Overflow instruction, INTO, has been executed previously, this error will cause the 8086 to perform a software interrupt procedure. Part of this procedure is a user-written subroutine to handle the error.

NOTE: The SBB instruction allows you to subtract two multibyte numbers because any borrow produced by subtracting less significant bytes is included in the result when the SBB instruction executes. Although the preceding examples were for 8-bit numbers to save space, the principles are the same for 16-bit numbers. For 16-bit signed numbers, however, SF is a copy of bit 15, and the least significant 15 bits of the number are used to represent the magnitude. Also, PF and AF function only for the lower 8 bits.

SCAS/SCASB/SCASW—Scan a String Byte or a String Word

SCAS compares a byte in AL or a word in AX with a byte or word pointed to by DI in ES. Therefore, the string to be scanned must be in the extra segment, and DI must contain the offset of the byte or the word to be compared. If the direction flag is cleared (0), then DI will be incremented after SCAS. If the direction flag is set (1), then DI will be decremented after SCAS. For byte strings, DI will be incremented or decremented by 1, and for word strings, DI will be incremented or decremented by 2. SCAS affects AF, CF, OF, PF, SF, and ZF, but it does not change either the operand in AL (AX) or the operand in the string. This instruction is often used with a repeat prefix to find the first occurrence of a specified byte or word in a string.

EXAMPLE:

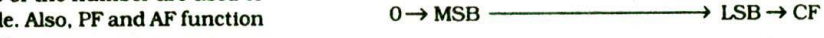
```
                ; Scan a text string of 80 characters
                ; for a carriage return, 0DH.
                ; Put offset of string into DI
MOV DI,OFFSET TEXT_STRING
MOV AL,0DH      ; Byte to be scanned for into AL
MOV CX,80       ; CX used as element counter
CLD             ; Clear DF so 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 Figure 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,1  ; 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           ; Copy packed BCD to BL
AND BL,0FH          ; 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):

```
          ; SI = 10010011 10101101, CF = 0
SHR SI,1  ; 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 address and flags on the stack, and execute an interrupt service procedure. An IRET instruction at the end of the interrupt service procedure will restore the flags which were pushed onto the stack, and return execution to the interrupted program. STI does not affect any other flags.

Please refer to Chapter 8 for a thorough discussion of interrupts.

STOS/STOSB/STOSW—Store Byte or Word in String

The STOS instruction copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI. In effect, it replaces a string element with a byte from AL or a word from AX. After the copy, DI is automatically incremented or decremented to point to the next string element in memory. If the direction flag (DF) is cleared, then DI will automatically be incremented by 1 for a byte string or incremented by 2 for a word string. If the direction flag is set, DI will be automatically decremented by 1 for a byte string or decremented by 2 for a word string. STOS does not affect any flags.

EXAMPLES:

```
   ; Point DI at start of destination string
MOV DI,OFFSET TARGET_STRING
STOS TARGET_STRING
   ; Assembler uses string name to determine
     whether string is of type byte or type word. If
     byte string, then string byte replaced with
     contents of AL. If word string, then string word
     replaced with contents of AX
   ; Point DI at start of destination string.

MOV DI,OFFSET TARGET_STRING
STOSB
   ; "B" added to STOS mnemonic directly tells
     assembler to replace byte in string with byte
     from AL. STOSW would tell assembler directly to
     replace a word in the string with a word from AX.
```

SUB—See Heading SBB

TEST—AND Operands to Update Flags—TEST Destination,Source

This instruction ANDs the contents of a source byte or word with the contents of the specified destination word. Flags are updated, but neither operand is changed. The TEST instruction is often used to set flags before a Conditional Jump instruction.

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 Figure 3-8. The destination operand can be in a register or in a memory location. The source and the destination cannot both be memory locations in an instruction. CF and OF are both 0's after TEST. PF, SF, and ZF will be updated to show the results of the ANDing. PF has meaning only for the lower 8 bits of the destination. AF will be undefined.

EXAMPLES (SYNTAX):

```
TEST AL,BH          ; AND BH with AL, no result stored.
                    ; Update PF, SF, ZF

TEST CX,0001H       ; AND CX with immediate number
                    ; 0001H, no result stored.
                    ; Update PF, SF, ZF

TEST BP,[BX][DI]    ; AND word at offset [BX][DI] in
                    ; DS with word in BP, no result
                    ; stored. Update PF, SF, and ZF
```

```
                    ; Example of a polling sequence
                    ; using TEST
AGAIN; IN AL,2AH    ; Read port with strobe
                    ; connected to LSB
TEST AL,01H         ; AND immediate 01H with AL
                    ; to test if LSB of AL is 1 or 0
                    ; ZF = 1 if LSB of result is 0
                    ; No result stored
JZ AGAIN            ; Read port again if LSB = 0
```

EXAMPLES (NUMERICAL):

```
                ; AL = 01010001
TEST AL,80H     ; AND immediate 80H with AL to test
                ; if MSB of AL is 1 or 0
                ; ZF = 1 if MSB of AL = 0.
                ; AL = 01010001 (unchanged)
                ; PF = 0, SF = 0,
                ; ZF = 1 because ANDing produced 00
```

WAIT—Wait for Test Signal or Interrupt Signal

When this instruction executes, the 8086 enters an idle condition in which it is doing no processing. The 8086 will stay in this idle state until the 8086 TEST input pin is made low or until an interrupt signal is received on the INTR or the NMI interrupt input pins. If a valid interrupt occurs while the 8086 is in this idle state, the 8086 will return to the idle state after the interrupt service procedure executes. It returns to the idle state because the address of the WAIT instruction is the address pushed on the stack when the 8086 responds to the interrupt request. WAIT affects no flags. The WAIT instruction is used to synchronize the 8086 with external hardware such as the 8087 math coprocessor. In Chapter 11 we describe how this works.

XCHG—XCHG Destination,Source

The XCHG instruction exchanges the contents of a register with the contents of another register or the contents of a register with the contents of a memory location(s). The XCHG cannot directly exchange the contents of two memory locations. A memory location can be specified as the source or as the destination by any of the 24 addressing modes summarized in Figure 3-8. The source and destination must both be words, or they must both be bytes. The segment registers cannot be used in this instruction. No flags are affected by this instruction.

EXAMPLES:

```
XCHG AX,DX  ; Exchange word in AX with word in DX

XCHG BL,CH  ; Exchange byte in BL with byte in CH

XCHG AL,PRICES [BX]  ; Exchange byte in AL with
                     ; byte in memory at
                     ; EA = PRICES [BX] in DS
```

XLAT/XLATB—Translate a Byte in AL

The XLATB instruction is used to translate a byte from one code to another code. The instruction replaces a byte in the AL register with a byte pointed to by BX in a lookup table in memory. Before the XLATB instruction can be executed, the lookup table containing the values for the new code must be put in memory, and the offset of the starting address of the lookup table must be loaded in BX. The code byte to be translated is put in AL. To point to the desired byte in the lookup table, the XLATB instruction adds the byte in AL to the offset of the start of the table in BX. It then copies the byte from the address pointed to by (BX + AL) back into AL. XLATB changes no flags. The section "Converting One Keyboard Code to Another" in Chapter 9 should clarify the use of the XLATB instruction.

EXAMPLE:

```
                ; 8086 routine to convert ASCII code
                ; byte to EBCDIC equivalent.
                ; ASCII code byte is in AL at start.
                ; EBCDIC code in AL at end
MOV BX,OFFSET EBCDIC_TABLE
                ; Point BX at start of EBCDIC
                ; table in DS
XLATB           ; Replace ASCII in AL with
                ; EBCDIC from table
```

The XLATB instruction can be used to convert any code of 8 bits or less to any other code of 8 bits or less.

XOR—Exclusive OR Corresponding Bits of Two Operands—XOR Destination,Source

This instruction Exclusive-ORs each bit in a source byte or word with the same number bit in a destination byte or word. The result replaces the contents of the specified destination. The contents of the specified source will not be changed. The result for each bit position will follow the truth table for a two-input Exclusive OR gate. In other words, a bit in the destination will be set to a 1 if that bit in the source and that bit in the original destination were not the same. A bit Exclusive-ORed with a 1 will be inverted. A bit Exclusive-ORed with a 0 will not be changed. Because of this, you can use the XOR instruction to selectively invert or not invert bits in an operand.

The source operand can be an immediate number, the contents of a register, or the contents of a memory location specified by any one of the addressing modes shown in Figure 3-8. The destination can be a register or a memory location. The source and destination cannot both be memory locations in the same instruction. CF and OF are both 0 after XOR. PF, SF, and ZF are updated. PF has meaning only for an 8-bit operand. AF is undefined after XOR.

EXAMPLES (SYNTAX):

```
XOR CL,BH   ; Byte in BH Exclusive-ORed with byte
            ; in CL. Result in CL. BH not changed
```

```
XOR BP,DI   ; Word in DI Exclusive-ORed with word
            ; in BP. Result in BP. DI not changed

XOR WORD PTR [BX],00FFH
   ; Exclusive-OR immediate number 00FFH with
   ; word at offset [BX] in data segment. Result in
   ; memory location [BX]

EXAMPLE (NUMERICAL):

            ; BX = 00111101 01101001
            ; CX = 00000000 11111111
XOR BX,CX   ; Result: BX = 00111101 10010110
            ; Note bits in lower byte are inverted
            ; CF,OF,SF,ZF = 0, PF = 1, AF = ?
```

ASSEMBLER DIRECTIVES

The words defined in this section are directions to the assembler, not instructions for the 8086. The assembler directives described here are those for the Intel 8086 macro assembler (ASM86), the Borland Turbo Assembler (TASM), and the IBM macro assembler (MASM). If you are using some other assembler, consult the manual for it to find the corresponding directives.

ASSUME

The ASSUME directive is used to tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS:CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE. The statement ASSUME DS:DATA tells the assembler that for any program instruction which refers to the data segment, it should use the logical segment called DATA. If, for example, the assembler reads the statement MOV AX,[BX] after it reads this ASSUME, it will know that the memory location referred to by [BX] is in the logical segment DATA. You must tell the assembler what to assume for any segment you use in a program. If you use a stack in your program, you must tell the assembler the name of the logical segment you have set up as a stack with a statement such as ASSUME SS:STACK_HERE. For a program with string instructions which use DI, the assembler must be told what to assume for the extra segment with a statement such as ASSUME ES:STRING_DESTINATION. For further discussion of the ASSUME directive, refer to the appropriate section of Chapter 3.

DB—Define Byte

The DB directive is used to declare a byte-type variable, or to set aside one or more storage locations of type byte in memory. The statement CURRENT_TEMPERATURE DB 42H, for example, tells the assembler to reserve 1 byte of memory for a variable named CURRENT_TEMPERATURE and to put the value 42H in that memory location when the program is loaded into RAM to be run. Refer to Chapter 3 for further discussion of the DB directive and to Chapter 4 for a discussion of how you can access variables named with a DB in your programs. Here are a few more examples of DB statements.

```
PRICES DB 49H,98H,29H ; Declare array of 3 bytes
named PRICES and initialize 3 bytes as shown.

NAME_HERE DB 'THOMAS' ; Declare array of 6 bytes
and initialize with ASCII codes for letters in THOMAS.

TEMPERATURE_STORAGE DB 100 DUP(?) ; Set aside
100 bytes of storage in memory and give it the name
TEMPERATURE_STORAGE, but leave the 100 bytes
uninitialized. Program instructions will load values into
these locations.

PRESSURE_STORAGE DB 20H DUP(0) ; Set aside 20H
bytes of storage in memory, give it the name PRES-
SURE_STORAGE, and put 0 in all 20H locations.
```

DD—Define Doubleword

The DD directive is used to declare a variable of type doubleword or to reserve memory locations which can be accessed as type doubleword. The statement ARRAY_POINTER DD 25629261H, for example, will define a doubleword named ARRAY_POINTER and initialize the doubleword with the specified value when the program is loaded into memory to be run. The low word, 9261H, will be put in memory at a lower address than the high word. A declaration of this type is often used with the LES or LDS instruction. The instruction LES DI,ARRAY_POINTER, for example, will copy the low word of this doubleword, 9261H, into the DI register and the high word of the doubleword, 2562H, into the extra segment register.

DQ—Define Quadword

This directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory. The statement BIG_NUMBER DQ 243598740192A92BH, for example, will declare a variable named BIG_NUMBER and initialize the 4 words set aside with the specified number when the program is loaded into memory to be run. The statement STORAGE DQ 100 DUP(0) reserves 100 quadwords of storage and initializes them all to 0 when the program is loaded into memory to be run.

DT—Define Ten Bytes

DT is used to tell the assembler to define a variable which is 10 bytes in length or to reserve 10 bytes of storage in memory. The statement PACKED_BCD DT 11223344556677889900 will declare an array named PACKED_BCD which is 10 bytes in length. It will initialize the 10 bytes with the values 11223344556677889900 when the program is loaded into memory to be run. This directive is often used when

declaring data arrays for the 8087 math coprocessor, discussed in Chapter 11. The statement RESULTS DT 20H DUP(0) will declare an array of 20H 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 at any time. The EVEN directive tells the assembler to increment the location counter to the next even address if it is not already at an even address. The 8086 can read a word from memory in one bus cycle if the word is at an even address. If the word starts at an odd address, the 8086 must do two bus cycles to get the 2 bytes of the word. Therefore, a series of words can be read much more quickly if they are at even addresses. When EVEN is used in a data segment, the location counter will simply be incremented to the next even address if necessary. When EVEN is used in a code segment, the location counter will be incremented to the next even address if necessary. A NOP instruction will be inserted in the location incremented over. Here's an example which shows why you might want to use EVEN in a data segment.

```
DATA_HERE SEGMENT
                    ; Location counter will point
                    ; to 0009 after assembler
                    ; reads next statement
SALES_AVERAGES DB 9 DUP(?)
                    ; declare array of 9 bytes
EVEN                ; Increment location
                    ; counter to 000AH
INVENTORY_RECORDS DW 100 DUP(0)
                    ; Array of 100 words starting
                    ; on even address for
                    ; quicker read
DATA_HERE ENDS
```

EXTRN

The EXTRN directive is used to tell the assembler that the names or labels following the directive are in some other assembly module. For example, if you want to call a procedure which is in a program module assembled at a different time from that which contains the CALL instruction, you must tell the assembler that the procedure is external. The assembler will then put information in the object code file so that the linker can connect the two modules together. For a reference to an external named variable, you must specify the type of the variable, as in the statement EXTRN DIVISOR:WORD. Constants defined with an EQU in another module are identified as type ABS in an EXTRN statement. For a reference to a label, you must specify whether the label is near (in a code segment with the same name) or far (in a code segment with a different name). The statement EXTRN SMART_DIVIDE:FAR tells the assembler that SMART_DIVIDE is a label of type far in another assembly module. Names or labels referred to as external in one module must be declared public with the PUBLIC directive in the module in which they are defined.

EXTRN statements should usually be bracketed with SEGMENT_ENDS directives which identify the segment in which the external name or label will be found. Here's an example of how to do this.

```
PROCEDURES_HERE SEGMENT
  EXTRN SMART_DIVIDE:FAR ; Found in segment
                         ; PROCEDURES_HERE
PROCEDURES_HERE ENDS
```

Refer to Chapter 5 for a thorough discussion of the use of the EXTRN and PUBLIC directives.

GLOBAL—Declare Symbols as PUBLIC or EXTRN

The GLOBAL directive can be used in place of a PUBLIC directive or in place of an EXTRN directive. For a name or symbol defined in the current assembly module, the GLOBAL directive is used to make the symbol available to other modules. The statement GLOBAL DIVISOR, for example, makes the variable DIVISOR public so that it can be accessed from other assembly modules.

The statement GLOBAL DIVISOR:WORD tells the assembler that DIVISOR is a variable of type word which is in another assembly module or EXTRN.

GROUP—Group-Related Segments

The GROUP directive is used to tell the assembler to group the logical segments named after the directive into one logical group segment. This allows the contents of all the segments to be accessed from the same group segment base. The assembler sends a message to the linker and/or locator telling it to link the segments so that the segments are physically in the same 64-Kbyte segment. An example of the GROUP directive would be SMALL_SYSTEM GROUP CODE,DATA,STACK_SEG. An appropriate ASSUME statement to follow this would be ASSUME CS:SMALL_SYSTEM, DS:SMALL_SYSTEM, SS:SMALL_SYSTEM.

INCLUDE—Include Source Code from File

This directive is used to tell the assembler to insert a block of source code from the named file into the current source module. This shortens the source code. An alternative is to use the editor block commands to copy the file into the current source module.

LABEL

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 at any time. The LABEL directive is used to give a name to the current value in the location counter. The LABEL directive must be followed by a term which specifies the type you want associated with that name. If the label is going to be used as the destination for a jump or a call, then the label must be specified as type near or type far. If the label is going to be used to reference a data item, then the label must be specified as type byte, type word, or type doubleword. Here's how we use the LABEL directive for a jump address.

```
ENTRY_POINT LABEL FAR   ; Can jump to here from
                        ; another segment

NEXT: MOV AL,BL         ; Cannot do a far jump
                        ; directly to a label
                        ; with a colon
```

Here's how we use the LABEL directive for a data reference.

```
STACK_SEG SEGMENT STACK
DW 100 DUP(0)           ; Set aside 100 words
                        ; for stack
STACK_TOP LABEL WORD    ; Give name to next
                        ; location after last
STACK_SEG ENDS          ; word in stack
```

To initialize stack pointer, then, MOV SP,OFFSET STACK_TOP.

LENGTH—Not Implemented in IBM MASM

LENGTH is an operator which tells the assembler to determine the number of elements in some named data item, such as a string or an array. When the assembler reads the statement MOV CX,LENGTH STRING1, for example, it will determine the number of elements in STRING1 and code this number in as part of the instruction. When the instruction executes, then, the length of the string will be loaded into CX. If the string was declared as a string of bytes, LENGTH will produce the number of bytes in the string. If the string was declared as a word string, LENGTH will produce the number of words in the string.

NAME

The NAME directive is used to give a specific name to each assembly module when programs consisting of several modules are written. The statement NAME PC_BOARD, for example, might be used to name an assembly module which contains the instructions for controlling a printed-circuit-board-making machine.

OFFSET

OFFSET is an operator which tells the assembler to determine the offset or displacement of a named data item (variable) or procedure from the start of the segment which contains it. This operator is usually used to load the offset of a variable into a register so that the variable can be accessed with one of the indexed addressing modes. When the assembler reads the statement MOV BX,OFFSET PRICES, for example, it will determine the offset of the variable PRICES from the start of the segment in which PRICES is defined and code this displacement in as part of the instruction. When the instruction executes, this computed displacement will be loaded into BX. An instruction such as ADD AL,[BX] can then be used to add a value from PRICES to AL.

ORG—Originate

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 at any time. The location counter is automatically set to 0000 when the assembler starts reading a segment. The ORG directive allows you to set the location counter to a desired value at any point in the program. The statement ORG 2000H tells the assembler to set the location counter to 2000H, for example.

A "$" is often used to symbolically represent the current value of the location counter. The $ actually represents the next available byte location where the assembler can put a data or code byte. The $ is often used in ORG statements to tell the assembler to make some change in the location counter relative to its current value. The statement ORG $ + 100 tells the assembler to increment the value of the location counter by 100 from its current value. A statement such as this might be used in a data segment to leave 100 bytes of space for future use.

PROC—Procedure

The PROC directive is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive, the term *near* or the term *far* is used to specify the type of the procedure. The statement SMART_DIVIDE PROC FAR, for example, identifies the start of a procedure named SMART_DIVIDE and tells the assembler that the procedure is far (in a segment with a different name from the one that contains the instruction which calls the procedure). The PROC directive is used with the ENDP directive to "bracket" a procedure. Refer to the ENDP discussion for an example of this. Also refer to Chapter 5 for a thorough discussion of how procedures are written and called.

PTR—Pointer

The PTR operator is used to assign a specific type to a variable or to a label. It is necessary to do this in any instruction where the type of the operand is not clear. When the assembler reads the instruction INC [BX], for example, it will not know whether to increment the byte pointed to by BX or to increment the word pointed to by BX. We use the PTR operator to clarify how we want the assembler to code the instruction. The statement INC BYTE PTR [BX] tells the assembler that we want to increment the byte pointed to by BX. The statement INC WORD PTR [BX] tells the assembler that we want to increment the word pointed to by BX. The PTR operator assigns the type specified before PTR to the variable specified after PTR.

The PTR operator can be used to override the declared type of a variable. Suppose, for example, that we have declared an array of words with the statements WORDS DW 437AH, 0B972H, 7C41H. Normally we would access the elements in this array as words. However, if we want to access a byte in the array, we can do it with an instruction such as MOV AL, BYTE PTR WORDS.

We also use the PTR operator to clarify our intentions when we use indirect Jump instructions. The statement JMP [BX], for example, does not tell the assembler whether to code the instruction for a near jump or for a far jump. If we want to do a near jump, we write the instruction as JMP WORD PTR [BX]. If we want to do a far jump, we write the instruction as JMP DWORD PTR [BX]. Please refer to Chapter 3 for further discussion of the 8086 jump instructions.

Public

Large programs are usually written as several separate modules. Each module is individually assembled, tested, and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared public in the module

in which it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules. An example is the statement PUBLIC DIVISOR,DIVIDEND, which makes the two variables DIVISOR and DIVIDEND available to other assembly modules.

If an instruction in a module refers to a variable or label in another assembly module, the assembler must be told that it is external with the EXTRN directive. Refer to the discussion of the EXTRN directive to see how this is done.

SEGMENT

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment. The statement CODE SEGMENT, for example, indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directives are used to "bracket" a logical segment containing code or data. Refer to the ENDS directive for an example of how this is done.

Additional terms are often added to a SEGMENT directive statement to indicate some special way in which we want the assembler to treat the segment. The statement CODE SEGMENT WORD tells the assembler that we want the contents of this segment located on the next available word (even) address when segments are combined and given absolute addresses. Without this WORD addition, the segment will be located on the next available paragraph (16-byte) address, which might waste as much as 15 bytes of memory. The statement CODE SEGMENT PUBLIC tells the assembler that this segment may be put together with other segments named CODE from other assembly modules when the modules are linked together.

SHORT

The SHORT operator is used to tell the assembler that only a 1-byte displacement is needed to code a Jump instruction. If the jump destination is after the Jump instruction in the 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.

TYPE

The TYPE operator tells the assembler to determine the type of a specified variable. The assembler actually 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.

CHAPTER 7

8086 System Connections, Timing, and Troubleshooting

As we showed you in Chapter 2, a microcomputer consists of a CPU, memory, and ports. These parts are connected together by three major buses: the address bus, the control bus, and the data bus. In Chapters 3 through 6, however, we made little mention of the hardware of a microcomputer because we were mostly concerned in these chapters with how a microcomputer is programmed. In this chapter we come back to take a closer look at microcomputer hardware.

OBJECTIVES

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

1. Draw a diagram showing how RAMs, ROMs, and ports are added to an 8086 CPU to make a simple microcomputer.
2. Describe how addresses sent out on the 8086 data bus are demultiplexed.
3. Describe the signal sequence on the buses as a simple 8086-based microcomputer fetches and executes an instruction.
4. Describe how a logic analyzer is connected to microcomputer signal lines and how it is used to make state and timing measurements.
5. Describe how address decoding circuitry gives a specific address to each device in a system and makes sure that only one device is enabled at a time.
6. Calculate the access time required for a memory device or port to work correctly in an 8086 microcomputer system.
7. List a series of steps you might take to troubleshoot a malfunctioning microcomputer system that once worked.

A BASIC 8086 MICROCOMPUTER SYSTEM

Introduction

In previous chapters we worked with what is often called the *programmer's model* of the 8086. This model shows features such as internal registers, number of address lines, number of data lines, and port addresses, which you need to write programs. Now we will look at the bus signals, timing, and circuit connections of an 8086 and an 8088. In a later chapter we will show the hardware connections for the 80286 and 80386 microprocessors.

System Overview

Figure 7-1*a* shows a block diagram of a simple 8086-based microcomputer. This diagram is a closer look at the generalized microcomputer in Figure 2-5. First, find the 8086 CPU, the ROM, and the RAM in Figure 7-1*a*. Next, look for the ports, represented by the block labeled MCS-80 PERIPHERAL. As we discuss in detail later, there is a wide variety of port devices available. Some examples are parallel port devices such as the 8255A, serial port devices, special port devices which interface with CRTs, port devices which interface with keyboards, and port devices which interface with floppy disks.

Next, find the control bus, address bus, and data bus in Figure 7-1*a*. The basic control bus consists of the signals labeled M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ at the top of the figure. If the 8086 is doing a read from memory or from a port, the $\overline{RD}$ signal will be asserted. If the 8086 is doing a write to memory or to a port, the $\overline{WR}$ signal will be asserted. During a read from memory or a write to memory, the M/$\overline{IO}$ signal will be high, and during port operations the M/$\overline{IO}$ signal will be low. As we show you in detail later, the $\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$ signals are used to enable addressed devices.

The address bus and the data bus are shown separately on the right side of Figure 7-1*a*, but where they leave the 8086, the two buses are shown as a single bus labeled ADDR/DATA. The reason for this is that, in order to save pins, the lower 16 bits of addresses are multiplexed on the data bus. Here's an overview of how this works.

As a first step in any operation where it accesses memory or a port, the 8086 sends out the lower 16 bits of the address on the data bus. External latches such as the 74LS373 octal devices shown in Figure 7-1*a* are used to "grab" this address and hold it during the rest of the operation. To strobe these latches at the proper time, the 8086 outputs a signal called Address Latch Enable or ALE. Once the address is stored on the outputs of the latches, the 8086 removes the address from the address/data bus and uses the bus for reading or writing data.

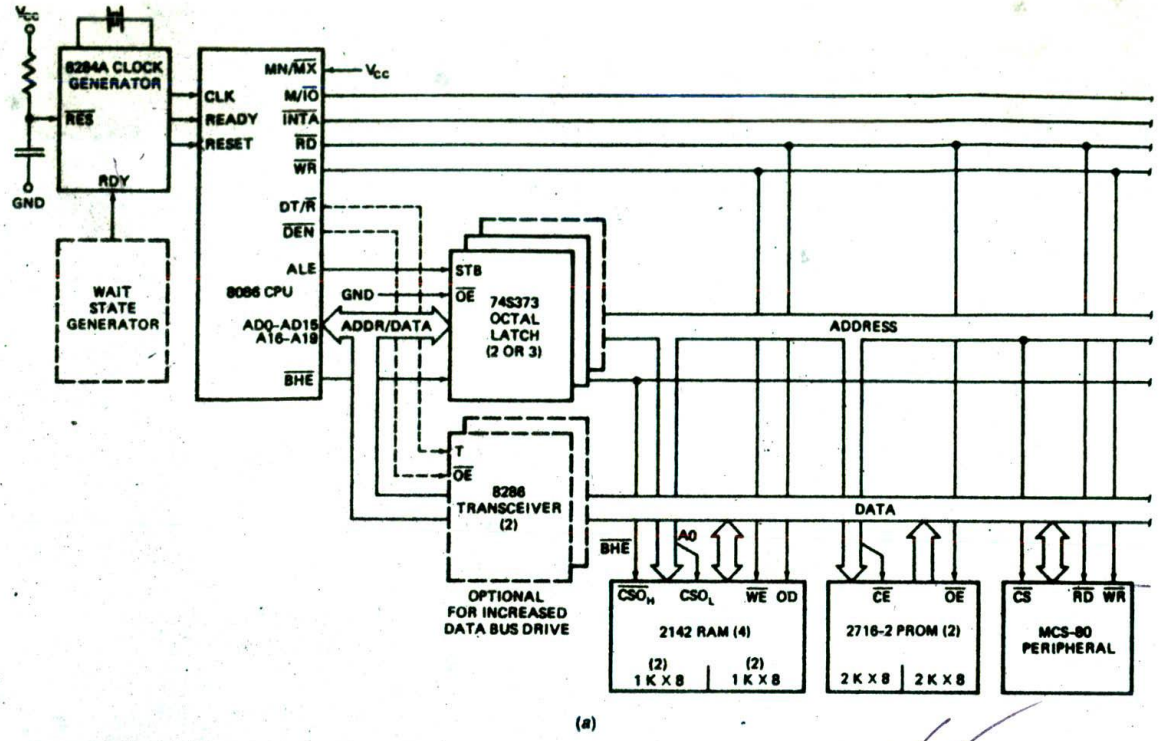


FIGURE 7-1 (a) Block diagram of a simple 8086-based microcomputer. (*See also next page.*)

Another section of Figure 7-1*a* to look at briefly is the block labeled 8286 Transceiver. This block represents bidirectional three-state buffers. For a very small system these buffers are not needed, but as more devices are added to a system, they become necessary. Here's why. Most of the devices—such as ROMs, RAMs, and ports—connected on microprocessor buses have MOS inputs, so on a dc basis they don't require much current. However, each input or output added to the system data bus, for example, acts like a capacitor of a few picofarads connected to ground. In order to change the logic state on these signal lines from low to high, all this added capacitance must be charged. To change the logic state to a low, the capacitance must be discharged. If we connect more than a few devices on the data bus lines, the 8086 outputs cannot supply enough current drive to charge and discharge the circuit capacitance fast enough. Therefore, we add external high-current drive buffers to do the job.

Buffers used on the data bus must be bidirectional because the 8086 sends data out on the data bus and also reads data in on the data bus. The *Data Transmit/Receive signal*, DT/$\overline{R}$, from the 8086 sets the direction in which data will pass through the buffers. When DT/$\overline{R}$ is asserted high, the buffers will be set up to transmit data from the 8086 to ROM, RAM, or ports. When DT/$\overline{R}$ is asserted low, the buffers will be set up to allow data to come into the 8086 from ROM, RAM, or ports.

The buffers used on the data bus must have three-state outputs so the outputs can be floated when the bus is being used for other operations. For example, you certainly don't want data bus buffer outputs enabled onto the data bus while the 8086 is putting out the lower 16 bits of an address on these lines. The 8086 asserts the $\overline{DEN}$ signal to enable the three-state outputs on data bus buffers at the appropriate time in an operation.

The final section of Figure 7-1*a* to look at is the 8284A clock generator in the upper left corner. This device uses a crystal to produce the stable-frequency clock signal which steps the 8086 through execution of its instructions in an orderly manner. The 8284A also synchronizes the RESET signal and the READY signal with the clock so that these signals are applied to the 8086 at the proper times. When the RESET input is asserted, the 8086 goes to address FFFF0H to get its next instruction. The first instruction of the system start-up program is usually located at this address, so asserting this signal is a way to *boot*, or start, the system. We will discuss the use of the READY input in the next section.

Now that you have an overview of the basic system connections for an 8086 microcomputer, let's take a look at the signal present on the buses as an 8086 reads data from memory or from a port.

8086 Bus Activities During a Read Machine Cycle

Figure 7-1*b* shows the signal activities on the 8086 microcomputer buses during simple read and write

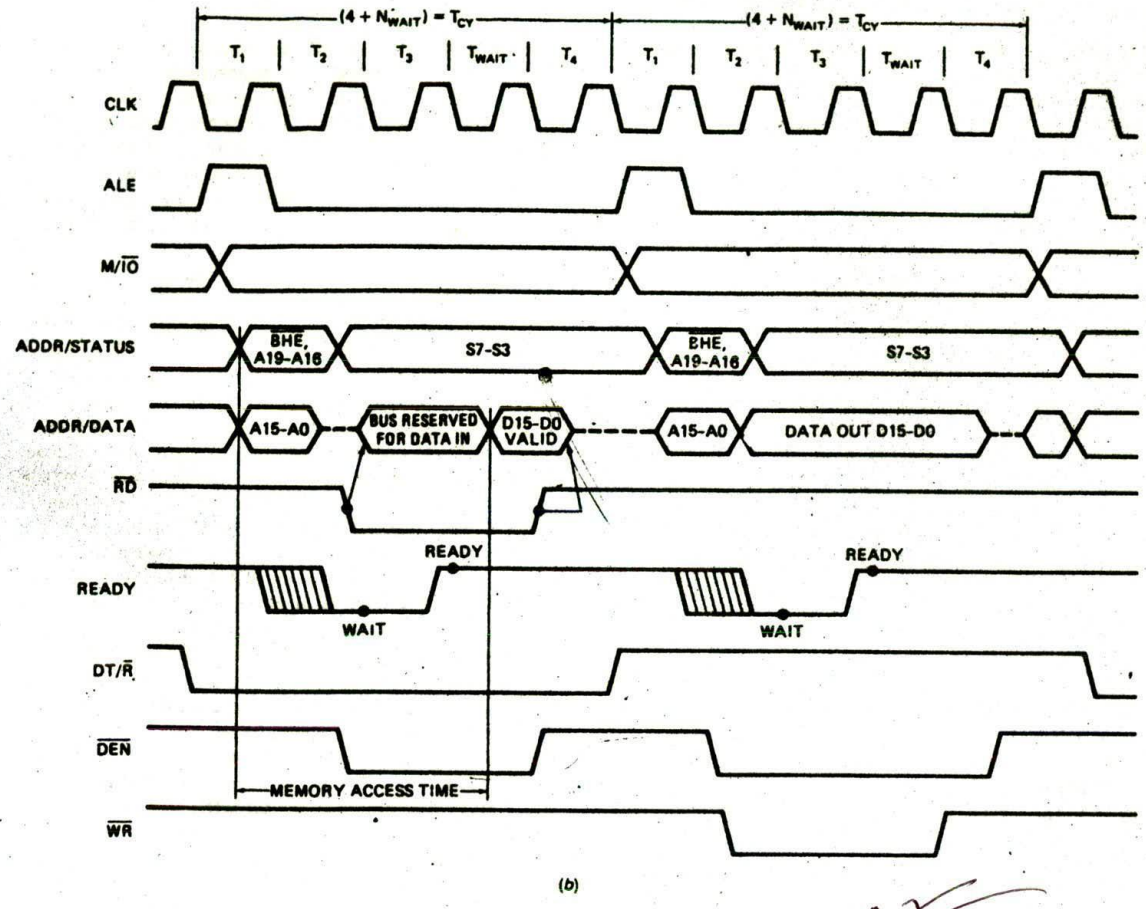


FIGURE 7-1 (*continued*) (*b*) Basic 8086 system timing. (*Intel Corporation*)

operations. Don't be overwhelmed by all the lines on this diagram. Their meanings should become clear to you as we work through the diagram.

The first line to look at in Figure 7-1*b* 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 Figure 7-1*a*. 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 Figure 7-1*b* 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 Figure 7-1*b* 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/$\overline{IO}$ 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/$\overline{IO}$ low if it is going to do a read from a port during this cycle. The timing diagram in Figure 7-1*b* shows two crossed waveforms for the M/$\overline{IO}$ 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/$\overline{IO}$, 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 Figure 7-1*a*, so these latches will be enabled when ALE is high. As you can also see in Figure 7-1*a*, the data inputs of these latches are connected to the 8086 AD0–AD15, A16–A19, and *Bus High Enable* ($\overline{BHE}$) 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 Figure 7-1*b* how the activitiy 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 Figure 7-1*b*, 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 $\overline{BHE}$ 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 $\overline{RD}$ signal low. If you trace the connection of the $\overline{RD}$ signal in Figure 7-1*a*, 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 $\overline{RD}$ 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 Figure 7-1*b* by an arrow going from the falling edge of $\overline{RD}$ 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 Figure 7-1*b* 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 $\overline{RD}$ signal. If the access time for a memory device is too long, the memory will not have valid data on its outputs soon enough in the machine cycle for the 8086 to receive it correctly. The 8086 will then treat whatever garbage happens to be on the data bus as valid data and go on with the next machine cycle. As long as Murphy's law is still in force, the garbage read in will probably cause the entire program to crash. A section later in the chapter shows you how to calculate whether a particular ROM, RAM, or port device has a short-enough access time to work properly in a given 8086 system. For now, however, we just need you to understand the concept so we can show you one way that an 8086 can accommodate a slow device.

To refresh your memory, look again at the block diagram in Figure 7-1*a* to find an input on the 8086 CPU labeled READY. When this pin is high, the 8086 is "ready" and operates normally. If the READY input is made low at the right time in a machine cycle, the 8086 will insert one or more *WAIT* states between T_3 and T_4 in that machine cycle. The read timing diagram in Figure 7-1*b* shows an example of this. An external hardware device is set up to pulse READY low before the rising edge of the clock in T_2. After the 8086 finishes T_3 of the machine cycle, it enters a WAIT state. During a WAIT state, the signals on the buses remain the same as they were at the start of the WAIT state. The address of the addressed memory location is held on the output of the latches, so it does not change, and as you can see from the timing diagram in Figure 7-1*b*, the control bus signals, M/$\overline{IO}$ and $\overline{RD}$, also do not change during the WAIT state, T_{WAIT}. The memory or port device then has at least one more clock cycle to get its data output. If the READY input is made high again during T_3 or during the WAIT state, as shown in Figure 7-1*b*, then after one WAIT state the 8086 will go on with the regular T_4 of the machine cycle.

If the 8086 READY input is still low at the end of a WAIT state, then the 8086 will insert another WAIT state. The 8086 will continue inserting WAIT states until the READY input is made high again.

To summarize, inserting the WAIT state(s) freezes the action on the buses. This gives the addressed device one or more extra clock cycles to put out valid data. As an example of how this is used, we can use slower (cheaper) ROM in a system by adding a simple circuit which pulses the READY input low each time the ROM is addressed. No WAIT states will be inserted in the read machine cycle for reading data from faster devices such as the RAM in the system.

Note in Figure 7-1*a* that a READY input signal is usually passed through the 8284A clock generator IC so that the READY signal actually applied to the 8086 is synchronized with the system clock.

Now let's look back at Figure 7-1*b* to see how $\overline{DEN}$ and DT/$\overline{R}$ function during a read machine cycle. During T_1 of the machine cycle the 8086 asserts DT/$\overline{R}$ low to put the data buffers in the receive mode. Then, after the 8086 finishes using the data bus to send out the lower 16 address bits, it asserts $\overline{DEN}$ low to enable the data bus buffers. The data put on the data bus by an addressed port or memory will then be able to come in through the buffers to the 8086 on the data bus.

The activities on the 8086 buses during a read machine cycle can be summarized as follows. The 8086 asserts M/$\overline{IO}$ high if the read is to be from memory and asserts M/$\overline{IO}$ low if the read is going to be from a port.

At about the same time, the 8086 asserts ALE high to enable the external address latches. It then sends out $\overline{BHE}$ and the desired address on the AD0–A19 lines. When the 8086 pulls the ALE line low, the address information is latched on the outputs of the external latches. After the 8086 is through using the AD0–AD15 lines for an address, it removes the address from these lines and puts the lines in the input mode (floats them). The 8086 then asserts its $\overline{RD}$ signal low. The $\overline{RD}$ signal going low turns on the addressed memory or port, which then outputs the desired data on the data bus. To complete the cycle the 8086 brings the $\overline{RD}$ line high again. This causes the addressed memory or port to float its outputs on the data bus. If the 8086 READY input is made low before or during T_2 of a machine cycle, the 8086 will insert WAIT states as long as the READY input is low. When READY is made high, the 8086 will continue with T_4 of the machine cycle. WAIT states can be used to give slow devices additional time to put out valid data. If a system is large enough to need data bus buffers, then the 8086 DT/$\overline{R}$ signal connected to these buffers will set them for input during a read operation or set them for output during a write operation. The 8086 $\overline{DEN}$ signal will enable the buffers at the appropriate time in the machine cycle.

8086 Bus Activities During a Write Machine Cycle

Now that we have analyzed the 8086 bus activities for a read machine cycle, let's take a look at the timing diagram for a write machine cycle in the right-hand side of Figure 7-1*b*. Most of this diagram should look very familiar to you because it is very similar to that for a read cycle.

During T_1 of a write machine cycle the 8086 asserts M/$\overline{IO}$ low if the write is going to be to a port, and it asserts M/$\overline{IO}$ high if the write is going to be to memory. At about the same time, the 8086 raises ALE high to enable the address latches. The 8086 then outputs $\overline{BHE}$ and the address that it will be writing to on AD0–A19. Incidentally, when writing to a port, lines A16–A19 will always be low, because the 8086 only sends out 16-bit port addresses. After the address has had time to pass through the latches, the 8086 brings ALE low again to latch the address on the outputs of the latches. Besides holding the address, these latches also function as buffers for the address lines. After the address information is latched, the 8086 removes the address information from AD0–AD15 and outputs the desired data on the data bus. It then asserts its $\overline{WR}$ signal low. The $\overline{WR}$ signal is used to turn on the memory or port that the data is to be written to. After the addressed memory or port has had time to accept the data from the data bus, the 8086 raises the $\overline{WR}$ signal line high again and floats the data bus.

If the memory or port device cannot accept the data word within a normal machine cycle, external hardware can be set up to pulse the READY input low each time that memory or a port device is addressed. If the READY input is pulsed low before or during T_2 of the machine cycle, the 8086 will insert a WAIT state after state T_3. Remember that during WAIT states the signals on the data bus, address bus, and control bus are held constant, so the addressed device has one or more extra clock cycles to accept the data from the data bus. If the READY input is made high before the end of the WAIT state, the 8086 will go on with state T_4 as soon as it finishes the WAIT state. If the READY input is still low just before the end of the WAIT state, the 8086 will insert another WAIT state. It will continue to insert WAIT states until READY is made high. The point here is that the 8086 can be forced to insert as many WAIT states as are necessary for the addressed device to accept the data.

If the system is large enough to need buffers on the data bus, then DT/$\overline{R}$ will be connected to the direction input on the buffers. During a write cycle, the 8086 asserts DT/$\overline{R}$ high to put the buffers in the transmit mode. When the 8086 asserts $\overline{DEN}$ low to enable the buffers, data output from the 8086 will pass through the buffers to the addressed port or memory location.

Work your way across the timing diagrams for the read and write machine cycles in Figure 7-1*b* until you feel that you understand the sequence of activities that occurs.

A Closer Look at the 8086

Figure 7-2, p. 168, shows a pin diagram for the 8086. You don't need to learn the detailed functions of all these pins. The main reason for showing you this is so that if you want to look at some of the 8086 signals with a scope or logic analyzer, you know which pins to connect to. We also want to make a few comments about some of the pins to give you a clearer idea of how an 8086-based microcomputer functions.

> NOTE: For reference, part of an 8086-data sheet showing all the pin descriptions is shown in Appendix A.

First, in Figure 7-2, find V_{CC} on pin 40 and ground on pins 1 and 20. Next, find the clock input, labeled CLK, on pin 19. As we showed you in the preceding sections, an 8086 requires a clock signal from some external clock generator to synchronize internal operations in the processor. Different versions of the 8086 have maximum clock frequencies ranging from 5 MHz to 10 MHz.

Now look for the address/data bus lines, AD0–AD15. Remember from the previous section that the 8086 has a 20-bit address bus and a 16-bit data bus and that the lower 16 address lines are multiplexed out on the data bus to minimize the number of pins needed. The 8086 sends out a signal called Address Latch Enable, or ALE, on pin 25 to strobe the external address latches. The upper 4 bits of the 20-bit address are sent out on the lines labeled A16/S3 through A19/S6. The double mnemonic on these pins shows that address bits A16 through A19 are sent out on these lines during the first part of a machine cycle, and status information, which identifies the type of operation being done in that cycle, is sent out on these lines during a later part of the cycle.

Having found the address bus and the data bus, now

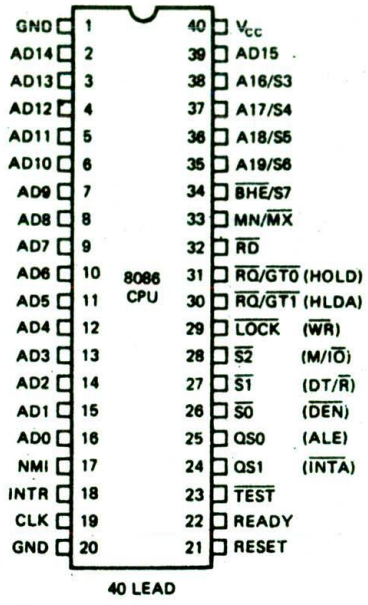


FIGURE 7-2 8086 pin diagram. (*Intel Corporation*)

look for the control bus signal pins. Pin 32 of the 8086 in Figure 7-2 is labeled $\overline{RD}$. This signal will be asserted low when the 8086 is reading data from memory or from a port. Pin 29 has the labels $\overline{WR}$ and $\overline{LOCK}$ next to it because it has two functions. The function of this pin and the functions of the other pins between 24 and 31 depend on the *mode* in which the 8086 is operating.

The operating mode of the 8086 is determined by the logic level applied to the MN/$\overline{MX}$ input, pin 33. If pin 33 is asserted high, then the 8086 will function in *minimum mode*, and pins 24 through 31 will have the functions shown in parentheses next to the pins in Figure 7-2. In minimum mode, for example, pin 29 will function as $\overline{WR}$, which will go low any time the 8086 writes to a port or to a memory location. The $\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$ signals form the heart of the control bus for a minimum-mode 8086 system. The 8086 is operated in minimum mode in systems such as the SDK-86 where it is the only microprocessor on the system buses.

If the MN/$\overline{MX}$ pin is asserted low, then the 8086 is in *maximum mode*. In this mode, pins 24 through 31 will have the functions described by the mnemonics next to the pins in Figure 7-2. In this mode, the control bus signals ($\overline{S0}$, $\overline{S1}$, and $\overline{S2}$) are sent out in encoded form on pins 26, 27, and 28. An external bus controller device decodes these signals to produce the control bus signals required for a system which has two or more microprocessors sharing the same buses. In Chapter 11 we discuss 8086 maximum-mode operation and show its use in multiple-microprocessor systems.

Another important pin on the 8086 is pin 21, the RESET input. If this input is asserted and then released, the 8086 will, no matter what it was doing, fetch its next instruction from physical address FFFF0H. At this address, then, you put the first instruction you want the microcomputer to execute after a reset or when the power is first turned on.

Finally, notice that the 8086 has two interrupt inputs, the *nonmaskable interrupt* (NMI) input on pin 17 and the *Interrupt* (INTR) input on pin 18. A signal can be applied to one of these inputs to cause the 8086 to stop executing its current program and go execute an interrupt procedure which takes care of the condition that caused the interrupt. You might, for example, connect a temperature sensor from a steam boiler to an interrupt input on an 8086. If the boiler gets too hot, then the temperature sensor will assert the interrupt input. This will cause the 8086 to stop executing its current program and go execute an interrupt-service procedure, which turns off the fuel supply to the boiler. A return instruction at the end of the interrupt-service procedure sends execution back to the interrupted program. In the next chapter we discuss interrupts further and show you how to write interrupt-service procedures.

Now we show you how to use a logic analyzer to observe and make measurements on microprocessor bus signals such as those we discussed in the preceding section.

USING A LOGIC ANALYZER TO OBSERVE MICROPROCESSOR BUS SIGNALS

Introduction

It is difficult to observe microprocessor bus signals with a standard scope because you can only look at two signal lines at a time. A logic analyzer such as the Tektronix 1230 shown in Figure 7-3 allows you to observe and make measurements on 16 to 64 signal lines at once. The least expensive version of the 1230 allows you to look at up to 16 signals at once, but expansion boards can be added to increase the number of input signal lines to 32, 48, or 64. Hewlett-Packard, Gould, and several other companies make comparable stand-alone logic analyzers.

Personal computers can be adapted to function as logic analyzers by installing plug-in units such as the

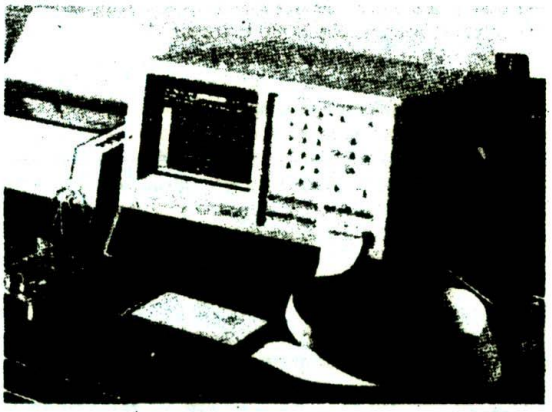


FIGURE 7-3 Tektronix 1230 logic analyzer. (*Courtesy of Tektronix Inc.*)

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 Figure 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 Figure 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 $\overline{RD}$ 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 20 ns. You can then determine the time between two events by counting the number of samples and multiplying the number by 20 ns.

If the analyzer is receiving either an internal or an external clock, it will be continuously taking samples of the input data and storing these samples in the internal RAM. A *trigger* signal tells the analyzer when to stop taking samples and display the samples stored in the RAM. As shown by the block diagram in Figure 7-4, you can use some external signal to trigger the analyzer, or you can use a *word recognizer* in the analyzer to produce a trigger signal. A word recognizer compares the binary word on the input signal lines with a word you set with switches or a keyboard. When the two words match, the word recognizer sends out a trigger signal.

Since the analyzer is continuously taking samples, you can set the analyzer for a *pretrigger* display, a *center trigger* display, or a *posttrigger* display. For an analyzer that displays 256 samples, pretrigger means that the display will show the 256 samples that were taken just before the trigger occurred. For center trigger mode, 128 samples taken before the trigger and 128 samples taken after the trigger will be displayed. Posttrigger mode

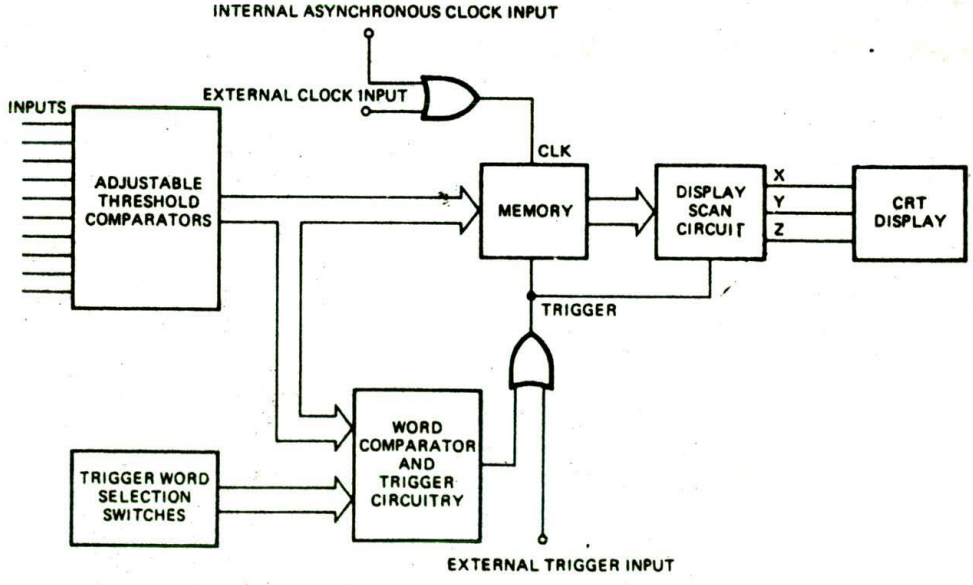
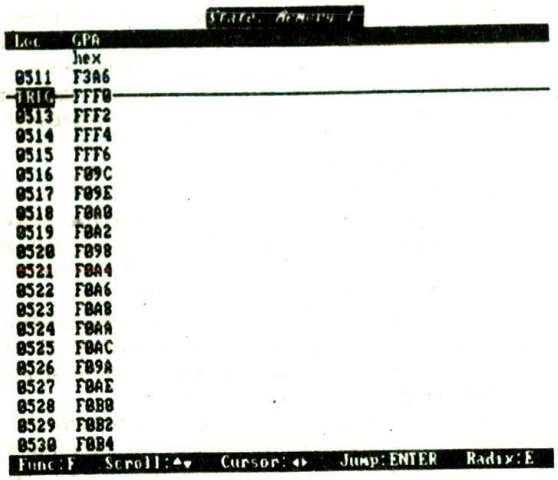
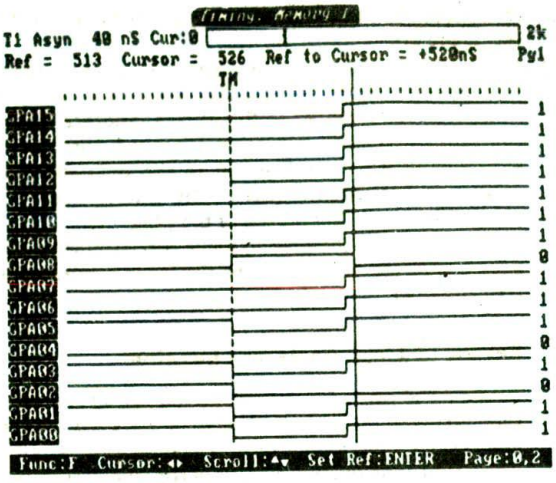


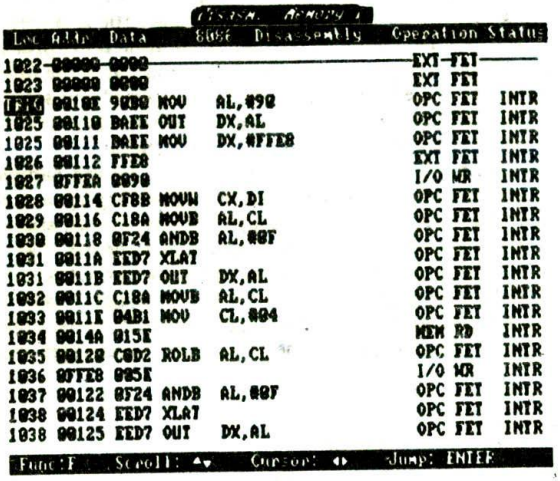
FIGURE 7-4 Block diagram of simple logic analyzer.



(a)



(b)



(c)

FIGURE 7-5 Logic analyzer display formats. (*a*) State listing showing sequences of addresses output by SDK-86 after reset. (*b*) Timing diagram display showing time between address output by 8086 and data output from RAM. (*c*) Disassembly listing showing execution part of SDK-86 display program.

means that the analyzer will take 256 more samples after the trigger and display them.

Figure 7-5 shows some of the formats in which a logic analyzer can display the samples stored in its RAM. The series of displayed data samples is often called a *trace*.

The *state table list* shown in Figure 7-5*a* is useful for observing, for example, a sequence of addresses sent out or a sequence of data words read in by a microprocessor. To determine whether a particular address line is shorted, you might tell the analyzer to display the table in binary so you can see the individual 1's and 0's. However, a hexadecimal listing such as that in Figure 7-5*a* makes it easier to recognize if a microcomputer is putting out addresses in the right sequence. Some analyzers, such as the Tektronix 1230, allow you to take a series of samples from a functioning system, store these samples in a second memory in the analyzer, and then compare them with a series of samples taken from a nonfunctioning system. We have found this feature quite helpful in troubleshooting malfunctioning instruments which have poor documentation.

The *timing diagram format* shown in Figure 7-5*b* is most useful when making time measurements with an internal clock. As we mentioned before, you can measure times by simply counting the number of clock pulses between two events and multiplying by the time per clock pulse. Some analyzers, such as the Tektronix 1230, allow you to determine the time between two events by placing a cursor on each event and reading the time between cursors directly on the screen.

The *disassembly format* shown in Figure 7-5*c* allows you to determine if a microprocessor is fetching and executing a sequence of instructions correctly. To produce this type of display, the logic analyzer must have additional hardware and software for the specific microprocessor that you are working with.

The following are the three major points you have to think about when you connect a logic analyzer up to do a trace:

1. The data inputs of the analyzer are connected to the system signals you want displayed in the trace.
2. The clock signal specified for the analyzer tells it when to take data samples and store them in its memory. To produce a trace which shows the sequence of states that a system steps through, you usually use an external clock. When you are using an external clock, you specify the clock edge which occurs when valid data is on the data inputs. For making timing measurements you usually use the crystal-controlled internal clock.

3. The trigger specified for the analyzer tells it when to stop taking samples and display the set of samples stored in its memory. Usually you will use the internal word recognizer to trigger the analyzer when a specified word is present on the data inputs.

Now that you have an overview of logic analyzer operation, here are some specific examples of how you observe 8086 bus signals and timing. Exercises in the lab manual which goes with this book give still more detailed examples.

MAKING A TRACE OF A SEQUENCE OF ADDRESSES

The first step in using a logic analyzer to look at microcomputer signals is to decide what specific signals you want to look at and connect the analyzer data inputs to those signals. If you want to do a trace which shows the sequence of addresses that the 8086 outputs as it executes a test program, you connect the data inputs of the analyzer pod to the 8086 AD0–AD15 pins.

The next step is to decide what signal to clock the analyzer on. To make this decision, you look carefully at the 8086 timing waveforms in Figure 7-1*b* to find a signal edge which occurs when valid addresses are on the AD0–AD15 lines. One possible signal to use for clocking the analyzer is the 8086 CLK signal shown at the top of the waveforms in Figure 7-1*b*. This signal has a falling edge when the address is valid on AD0–AD15, but it also has falling edges when the lines are floating and when the data from or to memory is on the lines. In other words, if the analyzer is clocked on this signal, the trace will show a mixture of data, addresses, and garbage, which you have to sort out.

A better choice for an analyzer clock signal is the 8086 ALE signal, because this signal is present only when addresses are on the AD0–AD15 lines. To use ALE as a clock signal, connect the External Clock input of the analyzer to the 8086 ALE pin. To determine which edge of the ALE signal to clock the analyzer on, look closely at the 8086 timing waveforms in Figure 7-1*b*. At the time when the positive edge of the ALE signal occurs, the 8086 has not yet output the address, so clocking the analyzer on this edge will not grab the addresses. The falling edge of the ALE signal occurs when the address is solidly settled on the AD0–AD15 lines, so you should set the analyzer to clock on the falling edge of ALE.

The final step is to determine what to trigger the analyzer on. Since you want to make a trace of a sequence of addresses, the logical choice here is to choose an internal trigger and set the internal word recognizer to produce a trigger when the first program address is present on the data inputs. For example, if the first program instruction is in memory at 00100H, you would set the analyzer to trigger when this address is present. When you specify the trigger position, set the analyzer for "begin" so that the trace listing starts with the specified address. The example logic analyzer trace in Figure 7-5*a* shows this type of display.

MAKING A TRACE OF A SEQUENCE OF DATA WORDS

As a second example of using an analyzer to look at microcomputer signals, suppose that you want to do a trace which shows the sequence of data words read in from memory as the 8086 executes a test program. For this trace you connect the analyzer data inputs to the 8086 AD0–AD15 pins, because the data comes in on these lines.

To determine what signal to clock the analyzer on and which edge of that signal to specify, you again look closely at the 8086 timing waveforms in Figure 7-1*b*. From these waveforms you should see that the 8086 $\overline{RD}$ signal is asserted during a Memory Read operation, so this is an appropriate signal to connect to the analyzer's External Clock input. The rising edge of the $\overline{RD}$ signal occurs when valid data is on the data bus, so set the analyzer to clock on a rising edge.

Since you want a trace of the data words read in from memory by the 8086, you need to look at the test program to determine what to trigger the analyzer on. For this example, assume the simple test program shown here is entered in memory and run.

```
00100 EB HERE:JMP HERE  ; Endless loop which does nothing
00101 FE
00102 90        NOP     ; Just more words to fetch
00103 90        NOP
00104 04        ADD AL,55H
```

Since the 8086 has a 16-bit data bus, it can read in a word (2 bytes) at a time if the word starts on an even address. When reading in the code bytes for this program then, the 8086 will send out address 00100H and assert both A0 and $\overline{BHE}$. The byte containing EBH will come into the 8086 on AD0–AD7, and the byte containing FEH will come into the 8086 on AD8–AD15. The first data word read in from memory then is FEEBH, so this is the word you set the analyzer to trigger on.

When the trace is completed, it will show the sequence of words FEEBH, 9090H, and 0455H over and over. The only part of this program that the 8086 executes is the HERE:JMP HERE instruction represented by the codes EBH and FEH. While the 8086 is decoding the JMP instruction, however, it fetches the codes for the following instructions and stores them in its queue, ready to be used. This is analogous to the way a helper sets up a stack of bricks for a bricklayer, so the bricklayer does not have to wait for the helper to go to the truck and get each brick as needed. In this program, however, the JMP instruction tells the 8086 to go back and fetch the JMP instruction again. The words 9090H and 0455H are fetched from memory and stored in the 8086 queue, but they are never used.

USING A CLOCK QUALIFIER IN LOGIC ANALYZER MEASUREMENTS

In the preceding example we showed you how to produce a trace of the data words read in from memory by an 8086. Now suppose that you are executing a program which reads data words from memory and data words

from ports. If you simply clock the analyzer on the rising edge of the $\overline{RD}$ signal as you did for the preceding example, the trace will contain both the data words read from memory and the data words read from ports. You can use the M/$\overline{IO}$ signal to produce a trace which contains only the words read from memory or only the words read from ports.

Remember from our previous discussions that when the 8086 writes a word to a memory location or reads a word from a memory location, it will assert its M/$\overline{IO}$ signal high. When the 8086 writes a word to a port or reads a word from a port, it will assert the M/$\overline{IO}$ signal low.

To produce a trace of only the data words read from ports, you connect the $\overline{RD}$ signal to the External Clock input of the analyzer and connect the M/$\overline{IO}$ signal to an input on the analyzer labeled *Clock Qualifier*. The principle here is that if this input is used, the analyzer will respond to a clock signal only if a specified level is present on that input. For this example, you want the analyzer to take a sample on the rising edge of the $\overline{RD}$ if M/$\overline{IO}$ is low. Therefore, you will specify a low as the active level for the clock qualifier input. Depending on the analyzer, the active level for the clock qualifier input may be set in a menu, by a switch on the pod, or by connecting the qualifier signal to a specific input on the data pod.

To produce a trace of only the data words read from memory, you can clock on the rising edge of $\overline{RD}$, connect the M/$\overline{IO}$ signal to the Clock Qualifier input, and specify a high for the clock qualifier. The point here is that by carefully choosing the clock signal and the qualifier signal, you can usually produce a trace of just the data you want.

MEASURING MEMORY ACCESS TIME WITH A LOGIC ANALYZER

As shown in the 8086 timing waveforms in Figure 7-1*b*, one type of memory access time is the time it takes for a memory device to produce valid data on its outputs after an address is applied to its address inputs. With a little thought you can use a logic analyzer to measure the actual memory access time in a system.

The first step in this measurement is to enter and run a test program which reads from the desired memory device over and over. For this example, we will use the same program we used in the preceding example. To make it easy to refer to, we repeat it here.

```
00100 EB HERE:JMP HERE  ; Endless loop which does nothing
00101 FE
00102 90        NOP     ; Just more words to fetch
00103 90        NOP
00104 04        ADD AL,55H
```

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 Figure 7-1*b*. 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 7-5*b* 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 Figure 7-1*a*, 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 Figure 7-7, p. 174.

Whenever you are approaching a system that is new to you, it is a good idea to study the detailed block diagram of the system carefully before you start digging into the actual schematics. The schematics for even a small system such as this are often spread over many pages. Without the overview that the block diagram gives, it is very difficult for you to see how all the schematic pieces fit together.

The first parts to look at in Figure 7-7 are the 8086 CPU and the 8284 *clock generator*. Note that the 8284 has a 14.7456-MHz crystal connected to it. According to the data sheet for the 8284, the frequency of the crystal connected to the 8284 will be divided by 3 to produce the clock signal sent to the 8086. Therefore, the actual 8086 clock frequency for this board will be 4.915 MHz. Another clock signal called PCLK, which is also produced by the 8284, has a frequency of half the clock frequency, or in this case 2.45 MHz. This signal is used as a general-purpose clock signal throughout the system. The hardware RST signal and the RDY signal are also passed through the 8284 to synchronize them with the clock signal before they are sent to the

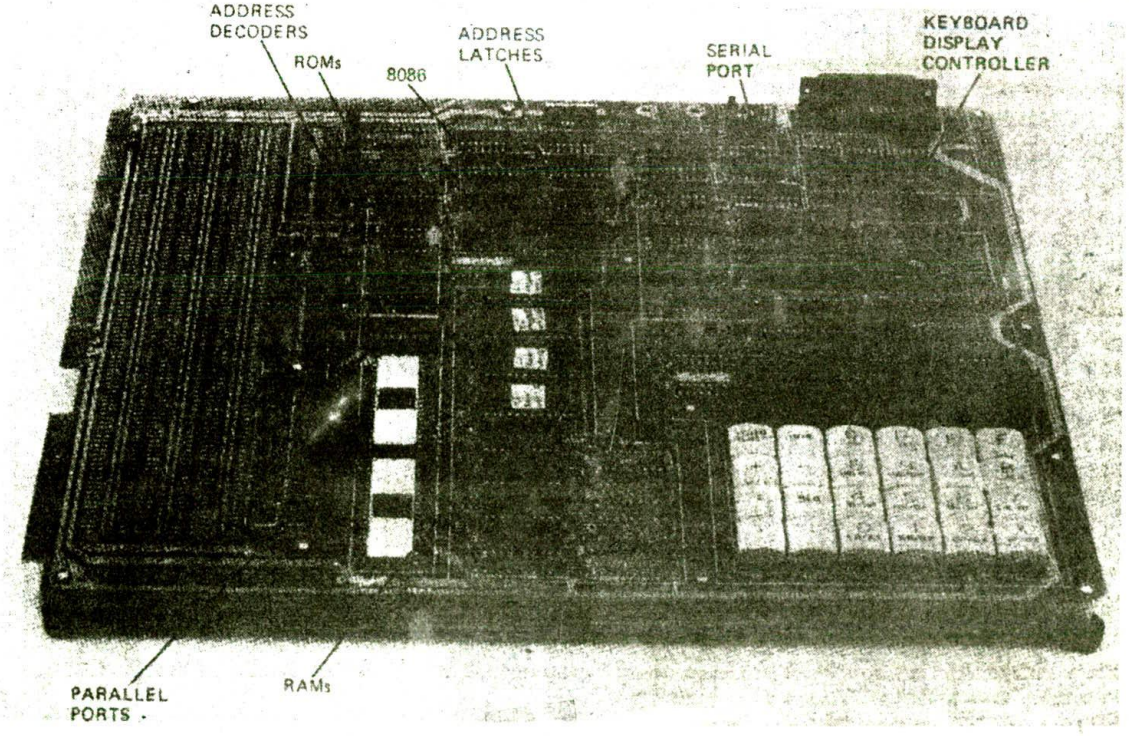


FIGURE 7-6 Intel SDK-86 microprocessor development board. (*Intel Corporation*)

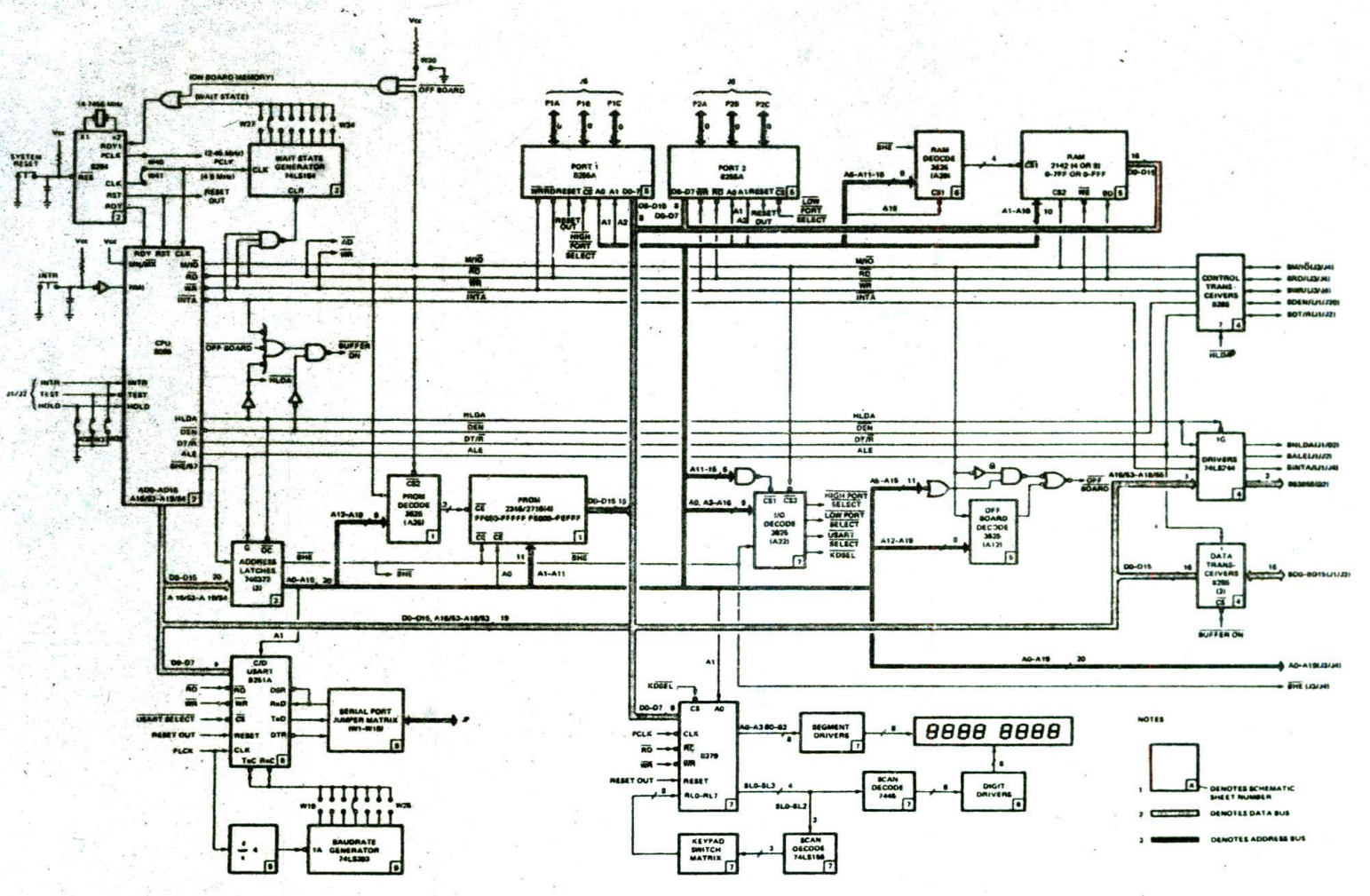


FIGURE 7-7 Detailed block diagram of SDK-86 board. The block diagram shown here and the schematics in Figure 7-8 are for the original Intel SDK-86 board. The SDK-86 board is now manufactured by University Research and Development Associates, Inc. (URDA), 4516 Henry Street, Suite 407, Pittsburgh, PA 15213. The URDA SDK-86 board is essentially the same as the original Intel board, except that it uses two 2732A EPROMs instead of 4 2716s, it uses 6264 static RAMs instead of 2142 static RAMs, and it uses discrete logic gates as address decoders instead of using 3625 PROMs. If you have an URDA board, consult the schematic for it to see the details of these sections. All functions and addresses on the URDA board are the same as those on the Intel board. (*Intel Corporation*)

8086. As you can see in Figure 7-7, considerable circuitry is connected to the RDY1 input so that several conditions can cause a WAIT state to be inserted in a machine cycle. The structure labeled W27 through W34 above the WAIT-state generator in Figure 7-7 represents wire-wrap pins which can be jumpered to specify the number of WAIT states desired in a machine cycle. We will discuss this in detail later.

By this time you may have noticed that the symbols for the 8284, 8086, and WAIT-state generator each have a small box containing a 2 in their lower right corner. This number tells you that the detailed schematic for these parts will be found on sheet number two of the set of schematics. Figure 7-8 on pages 176–184 shows the complete schematic set for the SDK-86 board, so you can check this out if you wish.

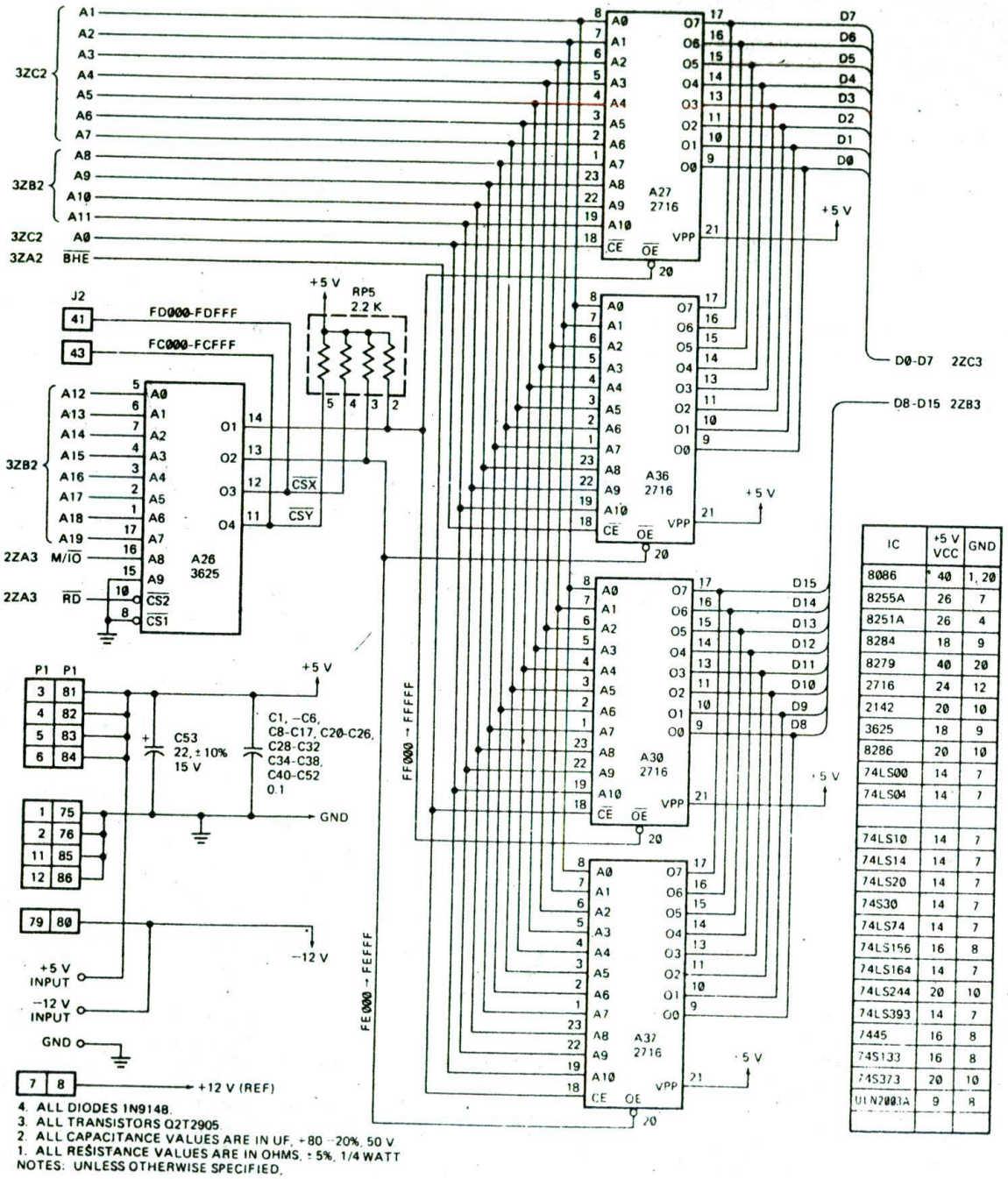
The next parts to look for in the block diagram of the SDK-86 are the *address latches*, which you know are needed to grab address information during T1 of a machine cycle. The box just below the 8086 in the diagram indicates that three 74S373s are used for address latches. AD0–AD15, A16–A19, and $\overline{BHE}$ are connected to the inputs of these latches. As expected, ALE is used to enable the latches. The information held on the output of the latches after ALE goes low is A0–A19 and $\overline{BHE}$. The /20 after A0–A19 on the output of the latches indicates that there are 20 lines in this group. A heavy black line is used to distinguish the demultiplexed address bus from the data bus.

Next, follow the address lines to the right on the diagram to find the ROM in the system. The box labeled PROM indicates that four 2316 or 2716 devices are used for ROM in the system. Each of these devices holds 2 Kbytes of memory. Also indicated in the PROM box in the diagram are the absolute addresses where these devices are located. Two of the EPROMs occupy the address space from FE000H to FEFFFH, and the other two occupy the address space from FF000H to FFFFFH. The 3625 PROM decoder connected to these EPROMs has two related purposes. The first is to produce a signal which turns on the desired ROM when you send out an address in the range assigned to that device. The second purpose is to make sure that only one device is outputting signals onto the data bus at a time. We discuss in detail later how address decoders are connected to give a desired address to a particular device in a system. Note that the enable input, $\overline{CS2}$, of the decoder PROM is connected to the $\overline{RD}$ signal from the 8086. This is done so that the PROM decoder will be enabled only if the 8086 is doing a read operation. Can you see why you would not want a ROM to be turned on if you accidentally sent out an address in its range during a write operation? The answer is that attempting to write to the outputs of a ROM can burn out both the ROM and the buffer outputs. The (A26) in the PROM decoder box of the block diagram, incidentally, indicates that the 3625 IC will be numbered A26 on the schematic sheet where it is found.

Follow the address bus to the upper right corner of the block diagram in Figure 7-7 to find how RAM is implemented in this system. The board comes with 2 Kbytes of static RAM contained in four 2142s, but there are sockets for another four 2142s. The initial four devices occupy the address space from 00000H to 007FFH. If four more 2142s are added, they will be in the address space 00800H–00FFFH. Another 3625 PROM is used here as a RAM decoder. As with the PROM decoder, the purposes of this device are to turn on a memory device which corresponds to a particular address sent out on the address bus and to make sure that only one device at a time is outputting data on a data bus line. The 8086 can read or write a byte or it can read or write a word. Therefore, 16 data lines are connected to the RAM block.

Now let's find the *system ports* in the block diagram in Figure 7-7. Two 8255As at the top of the page give the system *programmable* parallel ports. The term *programmable* in this case means that, as part of your program, you send the 8255A a *control byte*. The control byte tells the 8255A whether you want a particular group of lines on the device to function as outputs or as inputs. In Chapter 9 we show you how to make up and send these control words. The two 8255As in this system can be used individually to input or output parallel bytes. They can also be used together to input or output words. For byte input or output operations, only one of the devices will be turned on by asserting its $\overline{CS}$ input low. For word input or output operations, both 8255As will be turned on by asserting their $\overline{CS}$ inputs low. The high byte of a word to be output, for example, will then be sent to one of the ports in the PORT 1 device. The low byte of the word to be output will go to the corresponding port in the PORT 2 device. To be more specific, if the high byte of an output word goes to port P1A, then the low byte of that word will go to port P2A. In a later section of the chapter, we show how the addresses work out for these ports.

Most systems need a serial port so they can communicate with CRT terminals, modems, and other devices which require data to be sent and received in serial form. As shown in the lower left corner of Figure 7-7, the SDK-86 uses an 8251A as a *serial port*. The letters USART on this device stand for *universal synchronous/asynchronous receiver transmitter*, which is quite a mouthful. Chapter 13 discusses the initialization and use of the 8251A. For now, just think of this device as two back-to-back shift registers. One shift register accepts a parallel byte from the system data bus and shifts it out the TxD output in serial form. The other shift register shifts in serial data from the RxD input and converts it to parallel bytes which can be read by the 8086 on the system data bus. The 8251A has only eight data inputs, so data can only be written to or read from the 8251A a byte at a time. Therefore, only the lower 8 bits of the data bus are connected to it. Each of the shift registers in the 8251A requires a clock signal with a frequency of 16 or 64 times the rate at which you want to shift data bits in or out. The clock for the transmit shift register is called TxC, and the clock for the receive shift register is called RxC on the block diagram. These are tied together because you usually want to send and receive data at the same rates. The clock for these inputs is produced by dividing the 2.45-MHz PCLK signal from the 8284 clock generator. Wire-



IC	+5 V VCC	GND
8086	40	1, 20
8255A	26	7
8251A	26	4
8284	18	9
8279	40	20
2716	24	12
2142	20	10
3625	18	9
8286	20	10
74LS00	14	7
74LS04	14	7
74LS10	14	7
74LS14	14	7
74LS20	14	7
74S30	14	7
74LS74	14	7
74LS156	16	8
74LS164	14	7
74LS244	20	10
74LS393	14	7
7445	16	8
74S133	16	8
74S373	20	10
ULN2003A	9	8

FIGURE 7-8 SDK-86 complete schematics; see also pages 177–184. Sheet 1 of 9. (*Intel Corporation*)

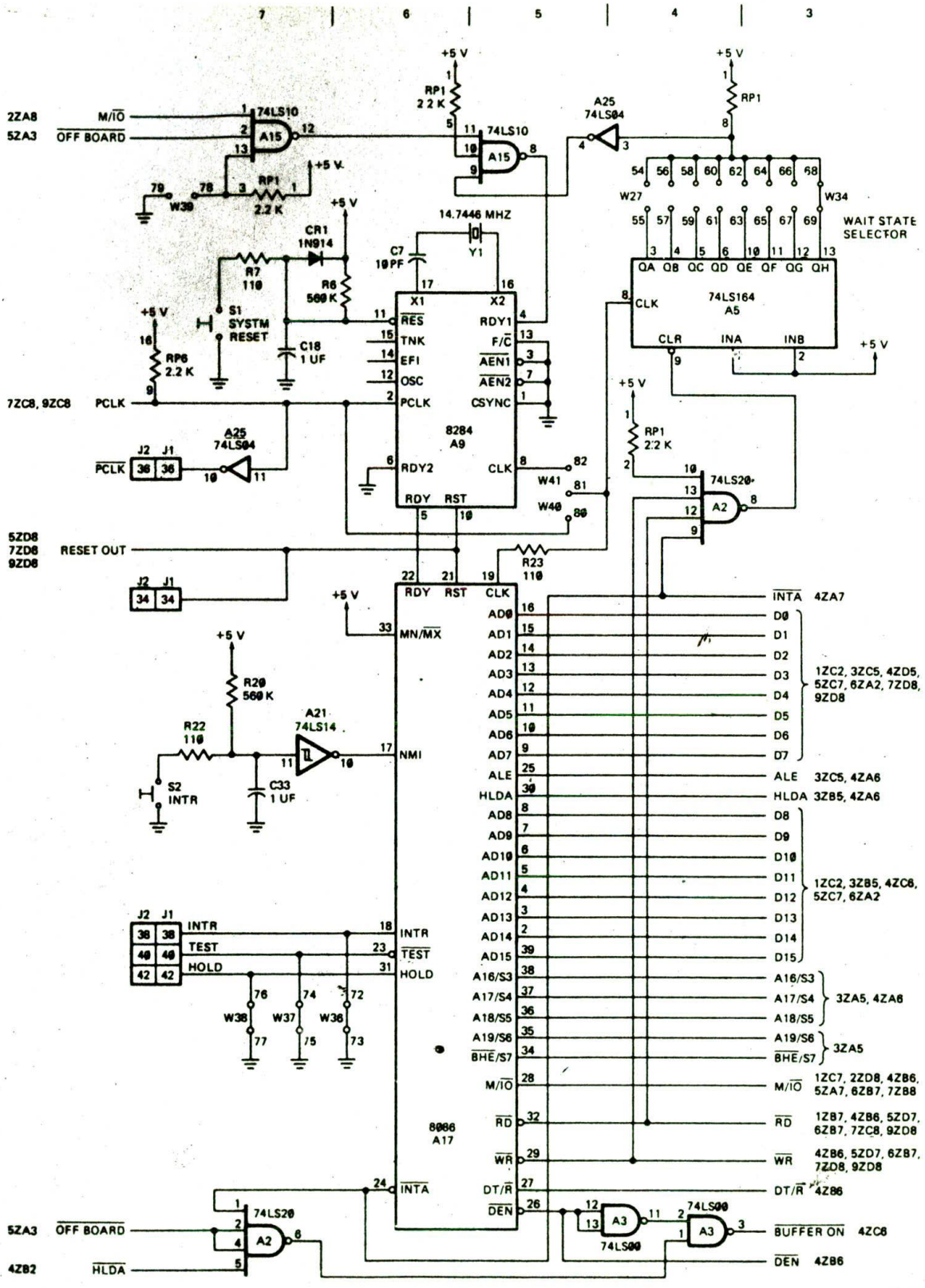


FIGURE 7-8 (*continued*) Sheet 2 of 9.

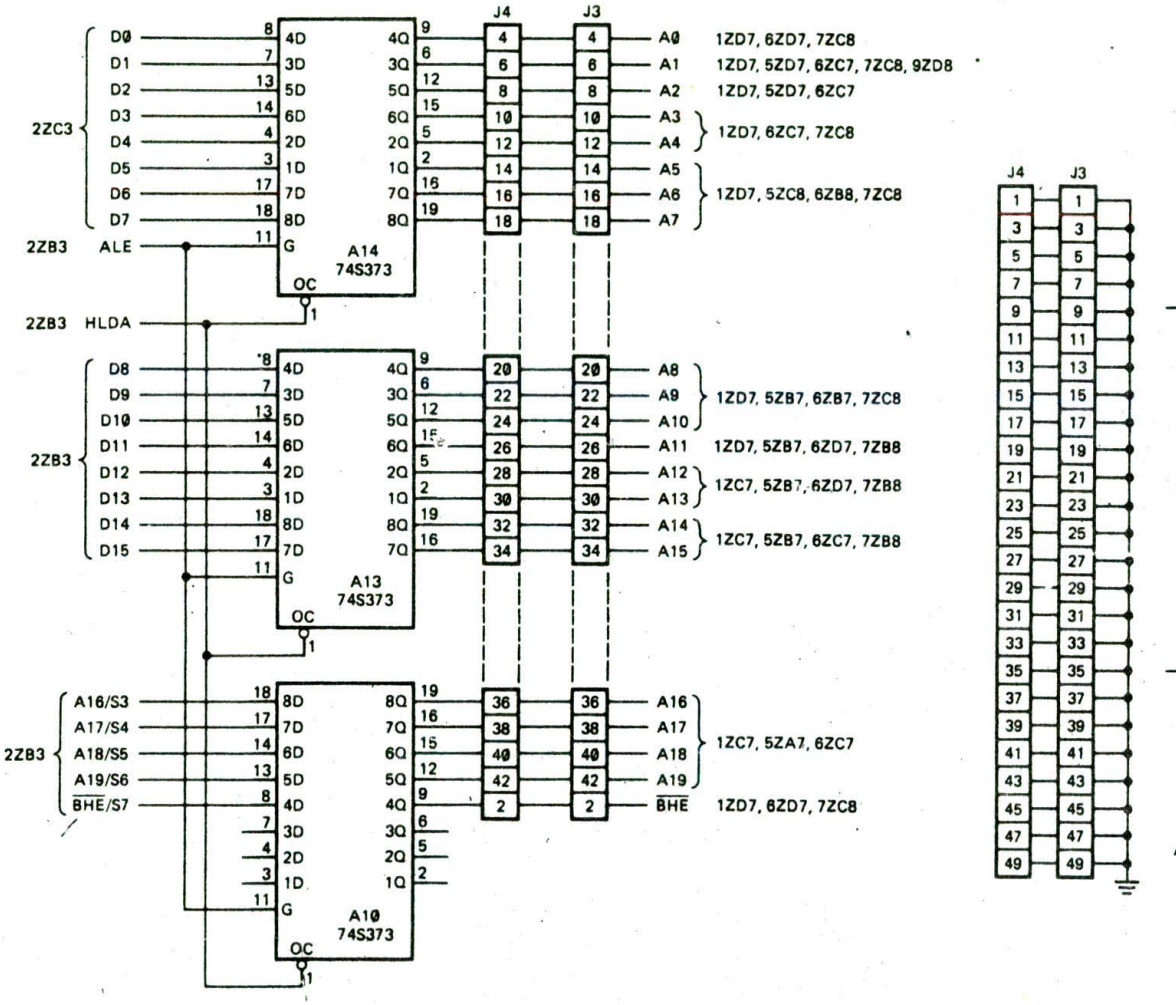


FIGURE 7-8 (*continued*) Sheet 3 of 9.

wrap jumper pins, W19–W25, allow you to select the desired TxC and RxC frequency from a divider chain in the 74LS393 *baud rate generator*. *Baud rate* is a way of specifying the rate at which data bits are shifted in or out of a serial device. Baud rate for a device such as the 8251A is defined as 1 over the time per bit. If the time per bit is 416 μs, for example, then the baud rate is 2400 baud. Common baud rates for serial data transmission are 300, 600, 1200, 2400, 9600, and 19,200.

The final port device to discuss here is the 8279 in the bottom center of the SDK-86 block diagram (Figure

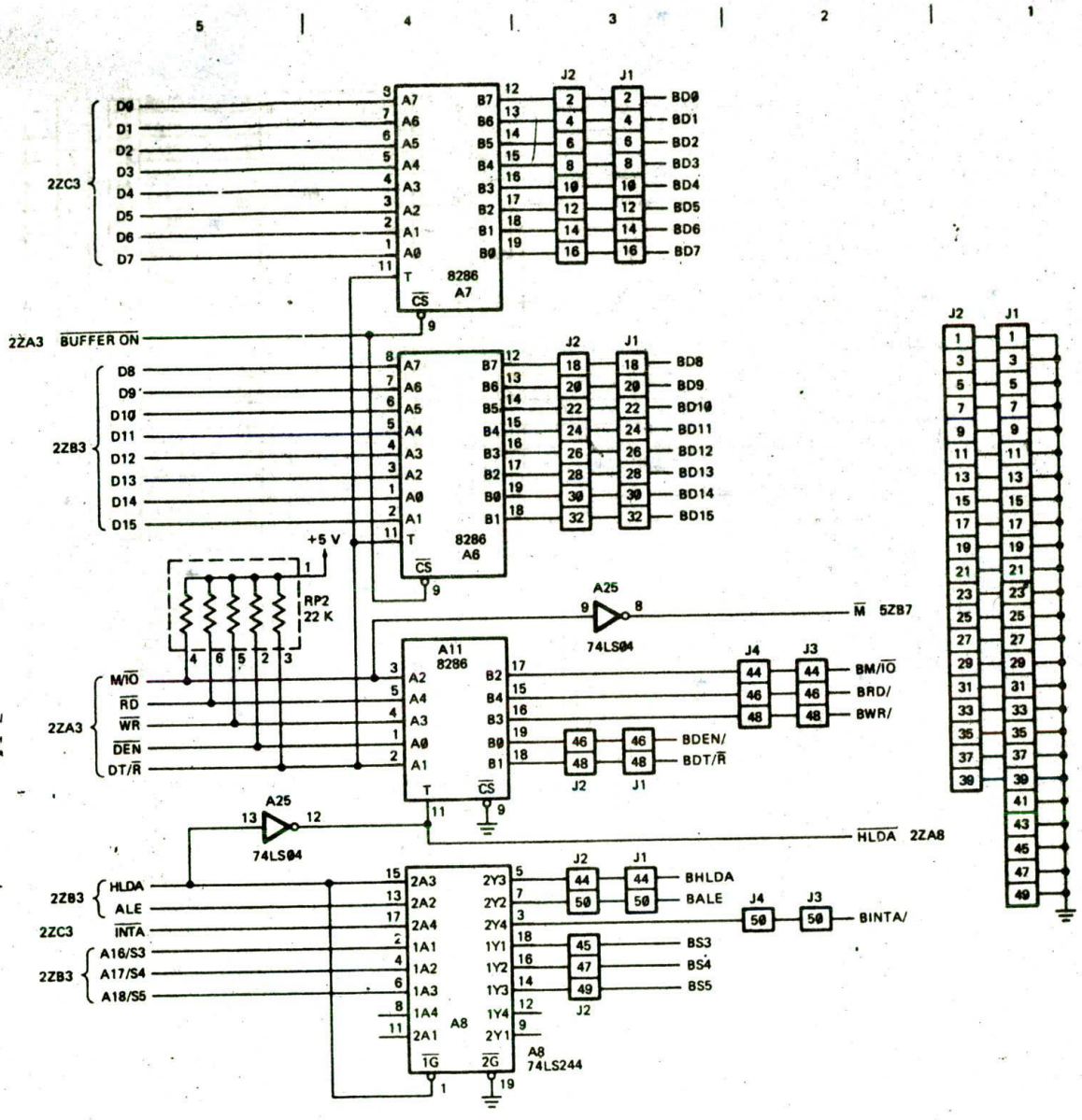


FIGURE 7-8 (*continued*) Sheet 4 of 9.

7-7). The 8279 is a *specialized input/output device* which has two major functions. The first function is to scan the hex keypad, detect when a key is pressed, debounce the signal from a pressed key, and store the code for the pressed key in an internal RAM, where it can be read by the 8086. The second major function of the 8279 is to refresh the multiplexed display on the eight 7-segment LED displays. Seven-segment codes for the digits to be displayed are sent to a RAM in the 8279. The 8279 then automatically sends out the code for one digit and turns on that digit. After a millisecond or so, the 8279 sends out the 7-segment code for the next digit and turns on that digit. The process is continued until all digits have been lit, and then the 8279 cycles back to the first digit again. In Chapter 9 we discuss in detail how you use an 8279. The main point for now is that this device takes care of scanning a keyboard and refreshing a display so that you don't have to do these operations as part of your program.

Now that you have an overview of the ports in this system, see if you can find in the block diagram the decoder which selects an addressed port. You should find the 3625 PROM labeled (A22) about in the center of the block diagram. Later we discuss how this device

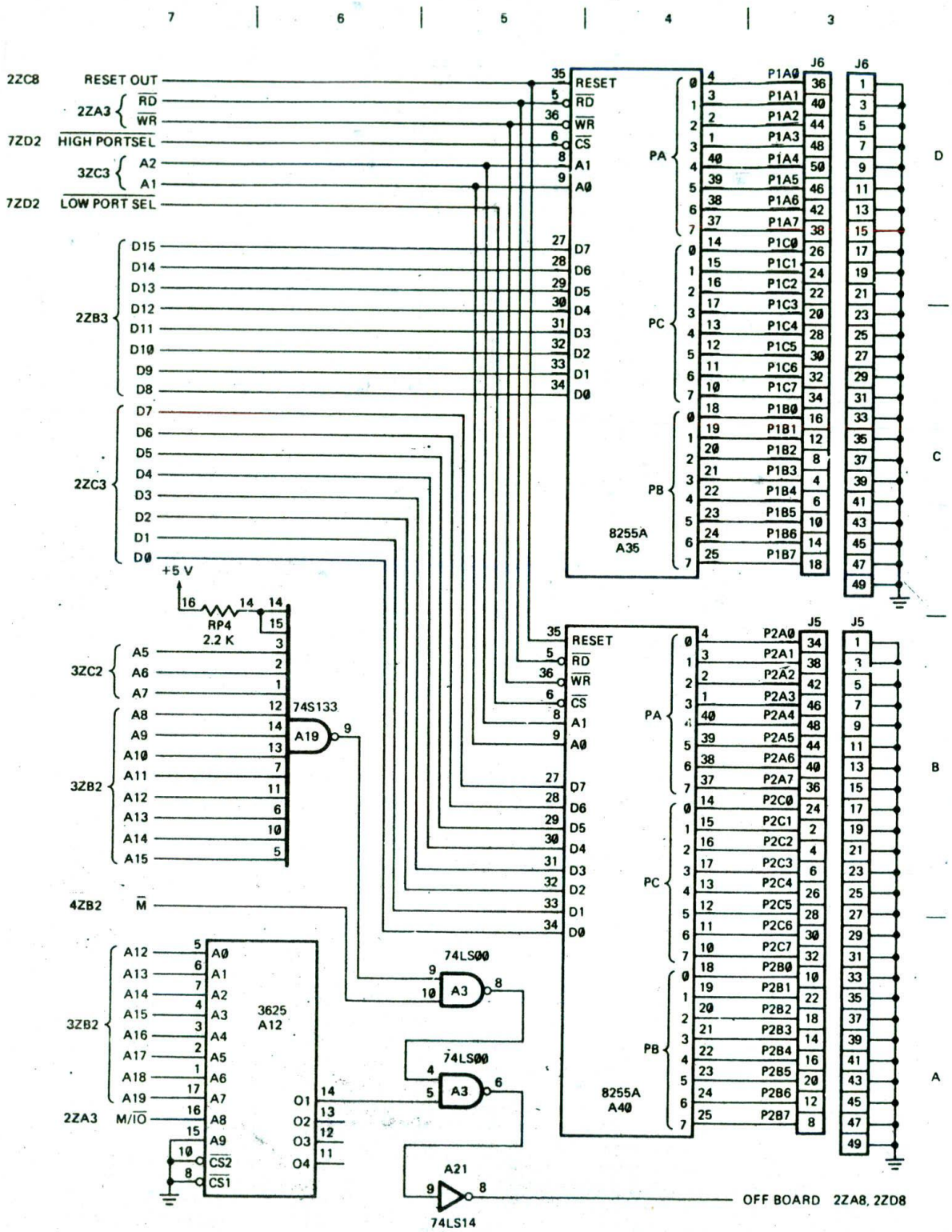


FIGURE 7-8 (*continued*) Sheet 5 of 9.

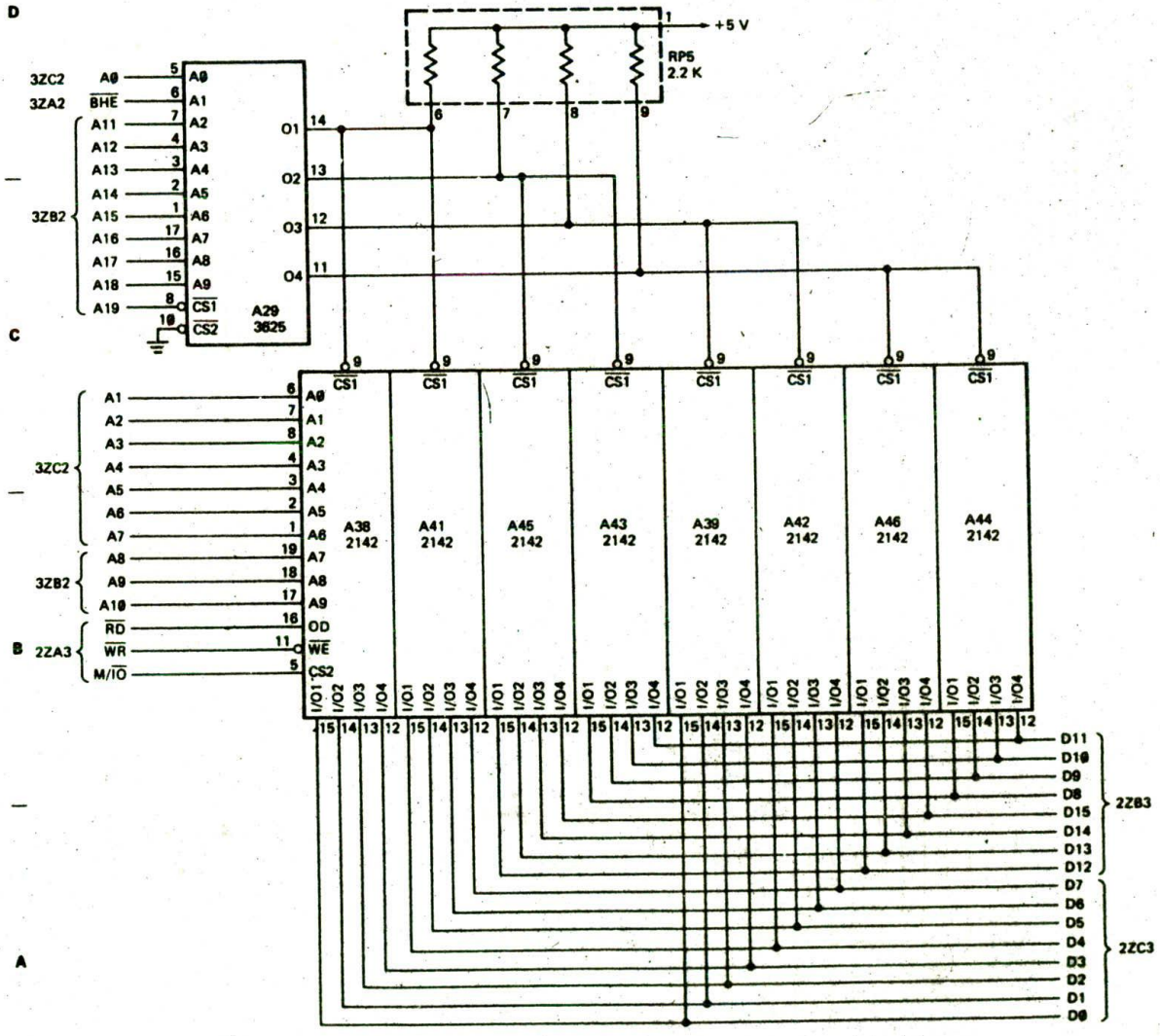


FIGURE 7-8 (*continued*) Sheet 6 of 9.

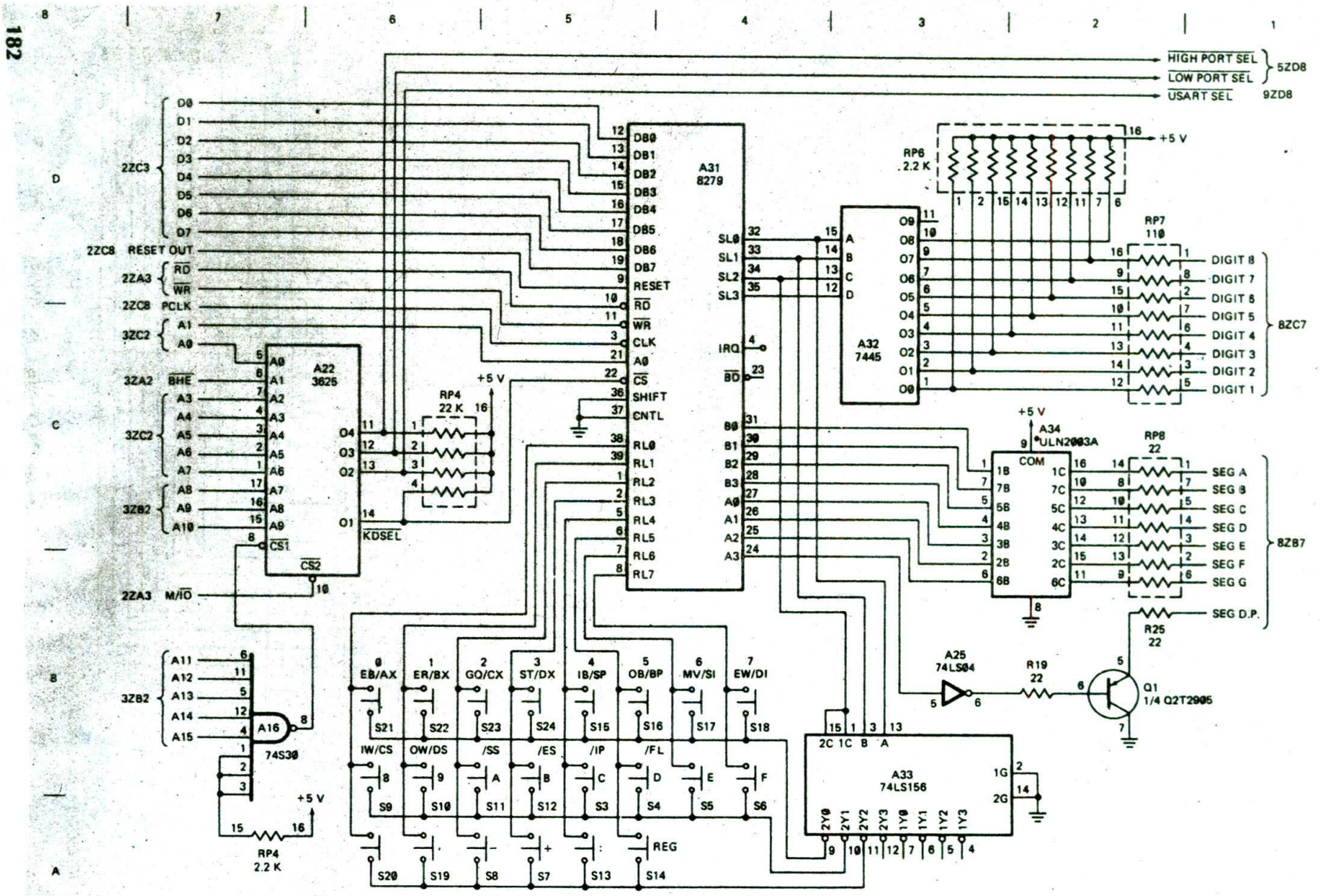


FIGURE 7-8 (*continued*) Sheet 7 of 9.

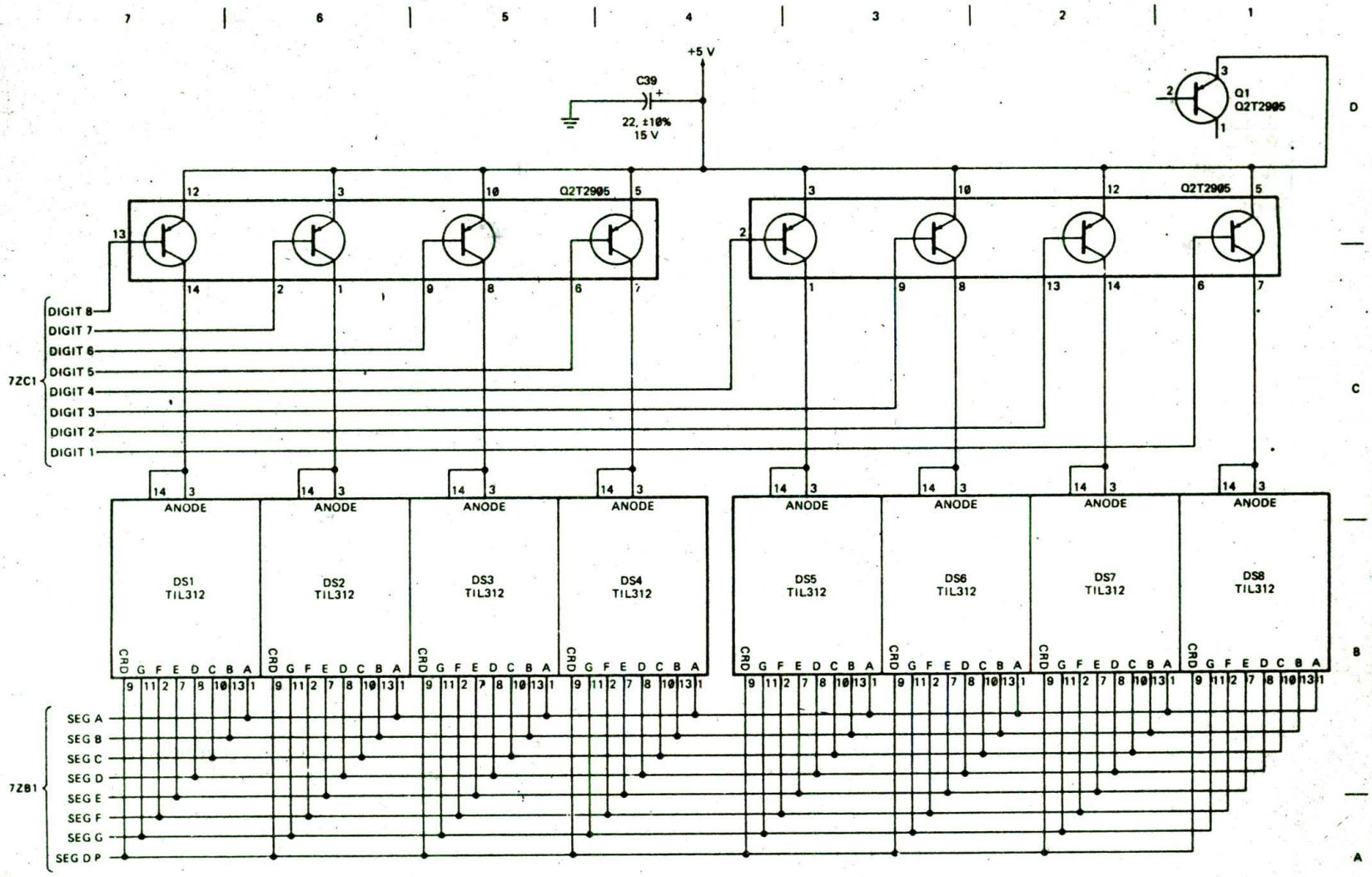
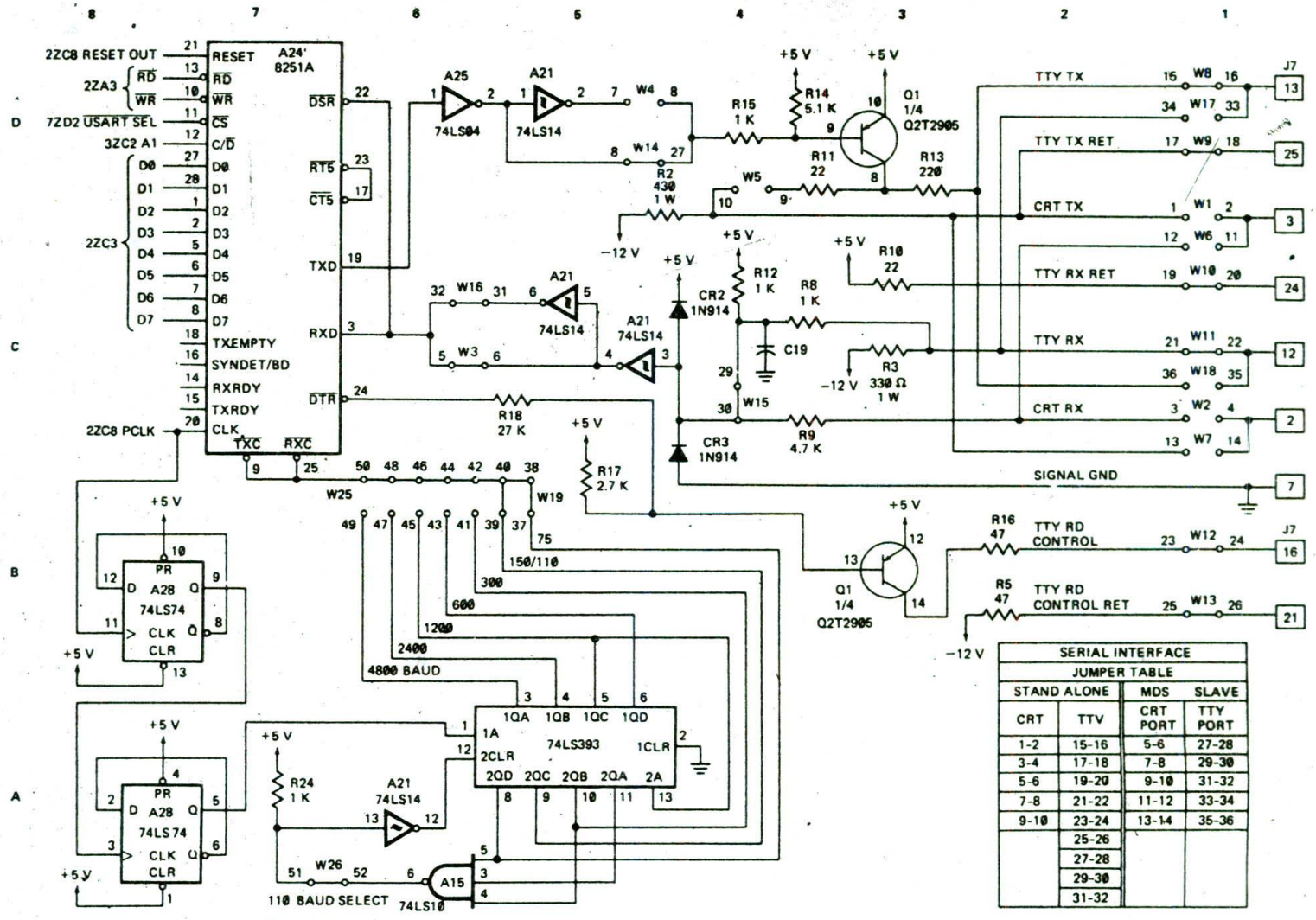


FIGURE 7-8 (*continued*) Sheet 8 of 9.



SERIAL INTERFACE JUMPER TABLE

STAND ALONE		MDS	SLAVE
CRT	TTY	CRT PORT	TTY PORT
1-2	15-16	5-6	27-28
3-4	17-18	7-8	29-30
5-6	19-20	9-10	31-32
7-8	21-22	11-12	33-34
9-10	23-24	13-14	35-36
	25-26		
	27-28		
	29-30		
	31-32		

FIGURE 7-8 (*continued*) Sheet 9 of 9.

produces the port select signals from a port address sent out by the 8086.

The final parts of the SDK-86 block diagram to take a look at are the buffers along the right-hand edge. The purpose of these devices is to buffer the data and control bus lines so that they can drive additional ROM, RAM, or ports that you might add to the expansion area of the board. Note that the address lines are already buffered by the 74S373 address latches.

A First Look at the SDK-86 Schematics

Now that you have seen an overview of the SDK-86, the next step is to take a first look at Figure 7-8, which shows the actual schematics for the board. At first the many pages of schematics may seem overwhelming to you, but if you use the *5-minute freak-out rule* and then approach the schematics one part at a time, you should have no trouble understanding them. The schematics simply show greater detail for each of the parts of the block diagram that we discussed in the preceding sections of the chapter.

At this point we want to make clear that it is not the purpose of this chapter to make you an expert on the circuit connections of an SDK-86 board. We use parts of these schematics to demonstrate some major concepts, such as address decoding, and to show how the parts are connected together to form a small but real system. Even if you do not have an SDK-86 board, you can learn a great deal from these schematics about how an 8086 system functions. Multipage schematics such as these are typical for any microprocessor-based board or product, so you need to get used to working with them.

Before getting started on the next major concept, we will discuss some of the symbols commonly used on microprocessor system schematics. First, take a look at the numbers across the top and bottom of each schematic and the letters along the sides of each. These are called *zone coordinates*. You use these coordinates to identify the location of a part or connection on the schematic, just as you might use similar coordinates on a road map to help you locate Bowers Avenue. For example, on sheet 1 of the schematics, find the lines labeled A1 through A7 in the upper left corner. Next to these lines you should see 3ZC2. This indicates that these address lines come from zone C2 on sheet 3. To see what the lines actually connect to, first find schematic sheet 3. Then move across the row of the schematic labeled C until you come to the column labeled 2. This zone is small enough that you should easily be able to find where these lines come from. The zone coordinates next to these lines on sheet 3 indicate the other schematic sheets and zones that these lines go to. For practice, try finding where a few more lines connect from and to.

The next points to look at on the schematics are the numbers on the ICs. In addition to a part number such as 2716, each IC has a number of the form A36. This second number is used to help locate the IC on the printed circuit board. The number is commonly silk-screened on the board next to the corresponding IC. Usually IC numbers are sequential and start from the upper left corner of the component side of a board. There may be several 2716s on the board, but only one will be labeled A36.

In addition to ICs, another type of device often found on microprocessor boards is a *resistor pack*. You can find an example in zone C5 of schematic sheet 1. As you can see from the schematic, this device contains four 2.2-kΩ resistors. Resistor packs may physically be thin, vertical, rectangular wafers, or they may be in packages similar to small ICs. The advantages of resistor packs are that they take up less printed-circuit-board space and that they are easier to install than individual resistors.

Some other symbols to look at in the schematics are the structures with labels such as J2 and P1. You can find examples of these in zones C7 and B7 of schematic sheet 1. These symbols are used to indicate *connectors*. The number in the rectangular box specifies the pin number on the connector that a signal goes to. The letter P stands for *plug*. A connector is considered a plug if it plugs into something else. In the case of the SDK-86, the connector labeled P1 is the printed-circuit-board edge connector. The letter J next to a connector stands for *jack*. A connector is considered a jack if something else plugs into it. On the SDK-86 board the jacks J1 through J6 are 50-pin connectors that you can plug ribbon cable connectors into. These jacks allow the address bus, data bus, control bus, and parallel ports to be connected to additional circuitry.

One more point to notice on the SDK-86 schematics is the capacitors on the power supply inputs shown in zone B6 of sheet 1. As you can see there, the schematic shows a large number of 0.1-μF capacitors in parallel with a 22-μF capacitor. Most systems have *filtering* such as this on their power lines. You may wonder what is the use of putting all these small capacitors in parallel with one which is obviously many times larger. The point of this is that the large capacitor filters out, or *bypasses*, low-frequency noise on the power lines, and the small capacitors, spread around the board, bypass high-frequency noise on the power supply lines. Noise is produced on the power supply lines by devices switching from one logic state to another. If this noise is not filtered out with bypass capacitors, it may become great enough to disturb system operation.

Glance through the SDK-86 schematics to get an idea of where various parts are located and to see what additional information you can pick up from the notes on them. In the next section of this chapter, we discuss how microcomputer systems address memory and ports. As part of the discussion, we cycle back to these schematics to see how the SDK-86 does it.

Addressing Memory and Ports in Microcomputer Systems

ADDRESS DECODER CONCEPT

While discussing the block diagram of the SDK-86 board earlier in this chapter, we mentioned that the 3625 devices on the board serve as *address decoders*. One function of an address decoder is to produce a signal

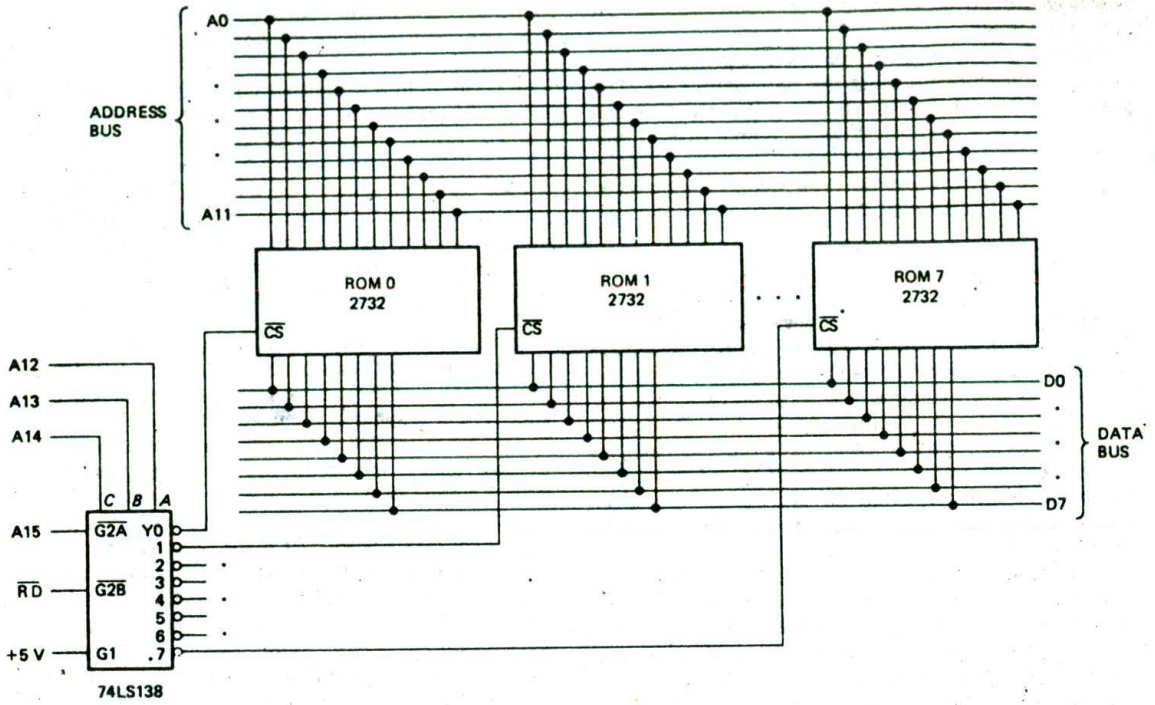


FIGURE 7-9 Parallel ROMs with decoder.

which enables the ROM, RAM, or port device that you want enabled for a particular address. A second, related function of an address decoder is to make sure that only one device at a time is enabled to put data on the data bus lines.

It seems that every microcomputer system does address decoding in a different way from every other system. Therefore, instead of memorizing the method used in one particular system, it is important that you understand the concept of address decoding. You can then figure out any system you have to work on.

AN EXAMPLE ROM DECODER

To start, look at Figure 7-9. This figure shows how eight EPROMs can be connected in parallel on a common address bus and common data bus. Just by looking at the schematic you can see that these EPROMs output bytes of data because each has eight outputs connected to the system data bus. The number of address lines connected to each device gives you an indication of how many bytes are stored in it. Each EPROM has 12 address lines (A0–A11) connected to it. Therefore, the number of bytes stored in the device is 2^{12} or 4096. If you have trouble with this, think of how many bits a counter has to have to count the 4096 states from 0 to 4095 decimal, or 0000H to 0FFFH.

Note that each 2732 in Figure 7-9 has a Chip Select, $\overline{CS}$, input. When this input is asserted low, the addressed byte in a device will be output on the data bus. The 74LS138 in Figure 7-9 makes sure that the $\overline{CS}$ input of only one ROM device at a time is low.

If the 74LS138 is enabled by making its $\overline{G2A}$ and $\overline{G2B}$ inputs low and its G1 input high, then only one output of the device will be low at a time. The output that will be low is determined by the 3-bit address applied to the C, B, and A select inputs. For example, if CBA is 000, then the Y0 output will be low, and all the other outputs will be high. This will assert the $\overline{CS}$ input of ROM 0. If CBA is 001, the Y1 output will be low and the ROM 1 will be selected. If CBA is 111, then Y7 will be low, and only ROM 7 will be enabled. Now let's see what address range these connections on the 74LS138 will give each of these ROMs in the system.

To determine the addresses of ROMs, RAMs, and ports in a system, a good approach in many cases is to use a worksheet such as that in Figure 7-10. To make one of these worksheets, you start by writing the address bits and the binary weight of each address bit across the top of the paper, as shown in the figure. To make it easier to convert binary addresses to hex, it helps if you mark off the address lines in groups of four, as shown. Next, draw vertical lines which mark off the three address lines that connect to the decoder select inputs (C, B, and A). For the decoder in Figure 7-9, address lines A14, A13, and A12 are connected to the C, B, and A inputs of the decoder, respectively. Then write under each address bit the logic level that must be on that line to address the first location in the first EPROM.

To address the first location in any of the EPROMs, the A0 through A11 address lines must all be low, so put a 0 under each of these address bits on the worksheet. To enable EPROM 0, the select inputs of the decoder must

		HEX DIGIT				HEX DIGIT				HEX DIGIT				HEX DIGIT				HEX EQUIVALENT ADDRESS
		2^{15} A15	2^{14} A14	2^{13} A13	2^{12} A12	2^{11} A11	2^{10} A10	2^{9} A9	2^{8} A8	2^{7} A7	2^{6} A6	2^{5} A5	2^{4} A4	2^{3} A3	2^{2} A2	2^{1} A1	2^{0} A0	
BLOCK 1	START	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0000
	END	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	= 0FFF
BLOCK 2	START	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	= 1000
	END	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	= 1FFF
BLOCK 3	START	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	= 2000
	END	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	= 2FFF
BLOCK 4	START	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	= 3000
	END	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	= 3FFF
BLOCK 5	START	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 4000
	END	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	= 4FFF
BLOCK 6	START	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	= 5000
	END	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	= 5FFF
BLOCK 7	START	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	= 6000
	END	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	= 6FFF
BLOCK 8	START	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	= 7000
	END	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	= 7FFF
			DECODER ADDRESS INPUTS															

FIGURE 7-10 Address decoder worksheet showing address decoding for eight 2732s in Figure 7-9.

be all 0's. Since address lines A14, A13, and A12 are connected to these select inputs, they must then all be 0's to enable EPROM 0. Write a 0 under each of these address bits on the worksheet. Since address line A15 is connected to the $\overline{G2A}$ enable input of the decoder, it must be asserted low in order for the decoder to work at all. Write a 0 under the A15 bit on your worksheet. Note that the $\overline{RD}$ signal from the microprocessor control bus is connected to the $\overline{G2B}$ enable input of the decoder. The decoder then will only be enabled during a read operation. This is done to make sure that data cannot accidentally be written to ROM. The G1 enable input of the decoder is permanently asserted by tying it to +5 V because we don't need it for anything else in this circuit.

You can now read the starting address of EPROM 0 directly from the worksheet as 0000H. The highest address in EPROM 0 is that address where A0–A11 are all 1's. If you put a 1 under each of these bits as shown on the worksheet, you can see that the ending address for EPROM 0 is 0FFFH. Remember that A12–A14 have to be low to select EPROM 0. A15 has to be low to enable the decoder. The address range of EPROM 0 is said to be 0000H to 0FFFH, a 4-Kbyte block.

Now let's use the worksheet to determine the address range for EPROM 1. EPROM 1 is enabled when A15 is 0, A14 is 0, A13 is 0, and A12 is 1. For the first address in EPROM 1, address lines A0 through A11 must all be low. Therefore, the starting address of EPROM 1 is 1000H. Its ending address, when A0 through A11 are all 1's, is 1FFFH. If you look at the worksheet in Figure 7-10, you should see that the address ranges for the other six EPROMs in the system are 2000H to 2FFFH, 3000H to 3FFFH, 4000H to 4FFFH, 5000H to 5FFFH, 6000H to 6FFFH, and 7000H to 7FFFH. In this system, then, we use address lines A14, A13, and A12 to select one of eight EPROMs in the overall address range of 0000H to 7FFFH. Some people like to think of address lines A14, A13, and A12 as "counting off" 4096-byte blocks of memory. If you think of the address lines as the outputs of a 16-bit counter, you can see how this works. The end address for each EPROM has all 1's in address bits A0–A11. When you increment the address to access the next byte in memory, these bits all go to 0, and a 1 rolls over into bits A14, A13, and A12. This increments the count in these 3 bits by 1 and enables the next highest 4096-byte EPROM. The count in these bits goes from binary 000 to 111.

AN EXAMPLE RAM DECODER

The system in Figure 7-9 contains only ROM. In most systems, you want to have ROM, RAM, and ports. To give you more practice with basic address decoding, we will show you now how you can add a decoder for RAM to the system.

Suppose that you want to add eight 2K × 8 RAMs to the system, and you want the first RAM to start at address 8000H, just above the EPROMs, which end at address 7FFFH.

To start, make a worksheet similar to the one in Figure 7-10. Addressing one of the 2048 bytes (2^{11}) in each RAM requires 11 address lines, A0 through A10. These lines will be connected directly to the address inputs on each RAM, so draw a vertical line on the worksheet to indicate this.

The three address lines A11, A12, and A13 will be used to select one of the eight RAMS, so write a 3-bit binary count sequence under these three columns in your worksheet.

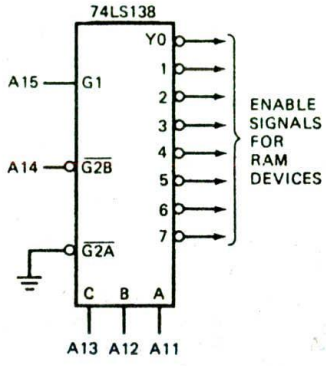
We want the RAM to start at address 8000H. For this address, A15 is a 1 and A14 is a 0, so mark these values in the appropriate columns in your worksheet. Your completed worksheet should look like the one in Figure 7-11*a*, p. 188. Now, let's see how you can implement this truth table with hardware.

Since you want to select one of eight RAM devices, you can use another 74LS138 such as the one we used for

HEX DIGIT				HEX DIGIT				HEX DIGIT				HEX DIGIT				HEX EQUIVALENT	START OF
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	ADDRESS	BLOCK
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 8000H	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	= 8800H	2
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	= 9000H	3
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	= 9800H	4
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	= A000H	5
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	= A800H	6
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	= B000H	7
1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	= B800H	8

DECODER ADDRESS INPUTS

(*a*)



(*b*)

FIGURE 7-11 Address decoder. (*a*) Worksheet for eight 2-Kbyte RAMs starting at address 8000H. (*b*) Schematic for 74LS138 connections.

the EPROMs. You want to select 2048-byte blocks of memory, so address line A11 will be connected to the A input of the decoder, A12 will be connected to the B input of the decoder, and A13 will be connected to the C input of the decoder.

You want the block of RAM selected by the outputs of this decoder to start at address 8000H. For this, address A15 is high and A14 is low. The G1 enable input of the decoder is active high, so you connect it to the A15 address line. This input will then be asserted when A15 is high. You connect the A14 address line to the $\overline{G2A}$ input of the decoder so that this input will be asserted when A14 is low. Because you don't need to use it in this circuit, you can simply tie the $\overline{G2B}$ input of the decoder to ground so that it will be asserted all the time. Figure 7-11*b* shows the connections for this decoder. Note that you don't connect the 8086 $\overline{RD}$ signal to an enable input on a RAM decoder, because you want to enable the RAMs for both read and write operations.

From the worksheet or truth table in Figure 7-11*a*, you can quickly determine the address range for each of the RAMs. The first RAM will start at address 8000H. The ending address for this RAM will be at the address where bits A0–A10 are all 1's. If you put 1's under these bits on your worksheet, you should see that the ending address for the first RAM is 87FFH. For practice, work out the hexadecimal addresses for each of the other seven RAMs. When you finish, compare your results with those in Figure 7-11*a*. The eight RAMs occupy the address space from 8000H to BFFFH.

AN EXAMPLE PORT DECODER

Figure 7-12*a* shows how another 74LS138 can be connected in a system to produce chip select signals for some port devices. The truth table or address decoder worksheet in Figure 7-12*b* shows the system address which corresponds to each of the decoder outputs.

First, note that A15 and A14 must be high to enable the decoder, so these bits are 1's in the worksheet. Then notice that A13 and A12 must be low to enable the decoder, so these columns on the worksheet contain 0's. Finally, address lines A3, A4, and A5 are connected to the decoder select inputs, so we wrote a 3-bit binary count sequence in these columns in the worksheet.

Address lines A0, A1, and A2 will be connected directly to the port devices to address individual ports and control registers in the devices. This is the same idea as connecting the lower address lines directly to a ROM so that we can address one of the bytes stored there.

Address lines A6 through A11 are not connected to the port devices or to the decoder, so they have no effect on selecting a port. We don't care then whether these bits are 1's or 0's. As you will see, these "don't care" bits mean that there are many addresses which will turn on one of the port devices. To give the simplest address for each device, however, we assume that each of these don't care bits is 0. Write 0's under each of these bits on your worksheet. You should now see that the address C000H will cause the Y0 output of the decoder to be asserted. The address C008H will cause the Y1 output of the decoder to be asserted. Using address lines A3, A4, and A5 on the decoder select inputs, then, leaves eight address spaces for each port device.

To see that any one of several different addresses can select one of these port devices, replace the 0 you put under A6 on the first line of your worksheet with a 1. This represents a system address of C040H. A15 and A14 are 1's and A13, A12, A5, A4, and A3 are 0's for this address. Therefore, this address will also cause the Y0 output of the decoder to be asserted. You can try other combinations of 1's and 0's on A6 through A11 if you need to further convince yourself that these bits

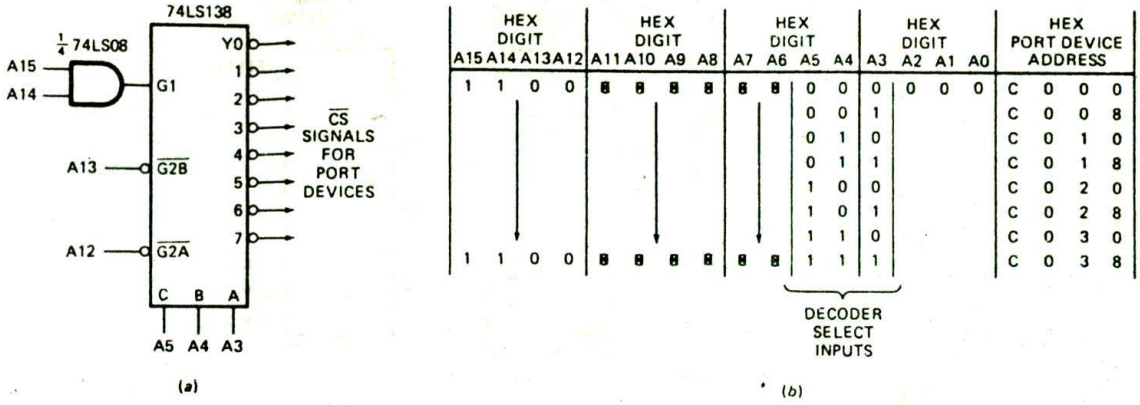


FIGURE 7-12 Adding a port device decoder. (*a*) Schematic for 74LS138 connections. (*b*) Address decoder worksheet.

don't matter when addressing ports. Again, we usually use 0's for these bits to give the simplest address.

Using a decoder which translates memory addresses to chip select signals for port devices is called *memory-mapped I/O*. In this system a port will be written to or read from in the same way as any other memory location. In other words, if this were an 8088 system, you would use an instruction such as MOV AL,DS:BYTE PTR 0C000H to read a byte of data from the first port to the AL register instead of using the MOV DX,0C000H and IN AL,DX instructions. The advantage of memory-mapped I/O is that any instruction which references memory can be used to input data from or output data to ports. In a system such as this, for example, the single instruction ADD AL,DS:BYTE PTR[0C000H] could be used to input a byte of data from the port at address C000H and add the byte to the AL register. The disadvantage of memory-mapped I/O is that some of the system memory address space is used up for ports and is therefore not available for memory.

You can use memory-mapped I/O with any microprocessor, but some microprocessors, such as those of the 8086 family, allow you to set up separate address spaces for input ports and for output ports. You access ports in these separate address spaces directly with the IN and OUT instructions. Having separate address spaces for input and output ports is called *direct I/O*. The advantage of direct I/O is that none of the system memory space is used for ports. The disadvantage is that only the specialized IN and OUT instructions can be used to input or output data.

In a later section of this chapter, we show how direct I/O is done with the 8086, but first we will discuss how the 8086 addresses memory.

8086 and 8088 Addressing and Address Decoding

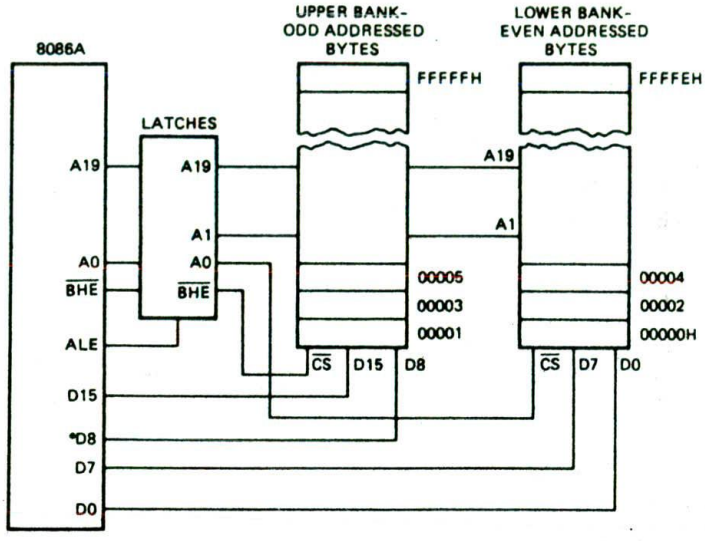
8086 MEMORY BANKS

The 8086 has a 20-bit address bus, so it can address 2^{20} or 1,048,576 addresses. Each address represents a stored byte. As you know from previous chapters, when you write a word to memory with an instruction such as MOV DS:WORD PTR[437AH],BX, the word is actually written into two consecutive memory addresses. Assuming that DS contains 0000, the low byte of the word is written into the specified memory address, 0437AH, and the high byte of the word is written into the next-higher address, 0437BH. To make it possible to read or write a word with one machine cycle, the memory for an 8086 is set up as two "banks" of up to 524,288 bytes each. Figure 7-13*a*, p. 190, shows this in diagram form.

One memory bank contains all the bytes which have even addresses such as 00000, 00002, and 00004. The data lines of this bank are connected to the lower eight data lines, D0 through D7, of the 8086. The other memory bank contains all the bytes which have odd addresses such as 00001, 00003, and 00005. The data lines of this bank are connected to the upper eight data lines, D8 through D15, of the 8086. Address line A0 is used as part of the enabling for memory devices in the lower bank. An addressed memory device in this bank will be enabled when address line A0 is low, as it will be for any even address. Address lines A1 through A19 are used to select the desired memory device in the bank and to address the desired byte in that device.

Address lines A1 through A19 are also used to select a desired memory device in the upper bank and to address the desired byte in that bank. An additional part of the enabling for memory devices in the upper bank is a separate signal called *bus high enable*, $\overline{\text{BHE}}$. $\overline{\text{BHE}}$ is multiplexed out from the 8086 on a signal line at the same time as an address is sent out. An external latch, strobed by ALE, grabs the $\overline{\text{BHE}}$ signal and holds it stable for the rest of the machine cycle, just as is done with addresses. Figure 7-13*b* shows you the logic level that will be on the $\overline{\text{BHE}}$ and A0 lines for different types of memory accesses.

If you read a byte from or write a byte to an even address such as 00000H, A0 will be low and $\overline{\text{BHE}}$ will be high. The lower bank will be enabled, and the upper bank will be disabled. A byte will be transferred to or from the addressed location in the low bank on D0–



(a)

ADDRESS	DATA TYPE	$\overline{BHE}$	A0	BUS CYCLES	DATA LINES USED
0000	BYTE	1	0	ONE	D0-D7
0000	WORD	0	0	ONE	D0-D15
0001	BYTE	0	1	ONE	D7-D15
0001	WORD	0	1	FIRST	D0-D7
		1	0	SECOND	D7-D15

(b)

FIGURE 7-13 8086 memory banks. (a) Block diagram. (b) Signals for byte and word operations.

D7. For an instruction such as MOV AH,DS:BYTE PTR[0000], the 8086 will automatically transfer the byte of data from the lower data bus lines to AH, the upper byte of the AX register. You just write the instruction and the 8086 takes care of getting the data in the right place.

Now, if the DS register contains 0000H and you use an instruction such as MOV AX,DS:WORD PTR[0000] to read a word from memory into AX, both A0 and $\overline{BHE}$ will be asserted low. Therefore, both banks will be enabled. The low byte of the word will be transferred from address 00000H to the 8086 on D0–D7. The high byte of the word will be transferred from address 00001H to the 8086 on D8–D15. The 8086 memory, remember, is set up in banks so that words, which have their low byte at an even address, can be transferred to or from the 8086 in one bus cycle. When programming an 8086, then, it is important to start an array of words on an even address for most efficient operation. If you are using an assembler, the EVEN directive is used to do this.

When you use an instruction such as MOV AL,DS:BYTE PTR[0001] to access just a byte at an odd address, A0 will be high and $\overline{BHE}$ will be asserted low. Therefore, the low bank will be disabled, and the high bank will be enabled. The byte will be transferred from memory address 00001H in the high bank to the 8086 on lines D8–D15. The 8086 will automatically transfer the byte of data from the higher eight data lines to AL, the low byte of the AX register. Note that address 00001H is actually the first location in the upper bank.

The final case in Figure 7-13*b* is the one where you want to read a word from or write a word to an odd address. The instruction MOV AX,DS:WORD PTR[0001H] copies the low byte of a word from address 00001 to AL and the high byte from address 00002H to AH. In this case, the 8086 requires two machine cycles to copy the two bytes from memory. During the first machine cycle the 8086 will output address 00001H, assert $\overline{BHE}$ low, and assert A0 high. The byte from address 00001H will be read into the 8086 on lines D8–D15 and put in AL. During the second machine cycle the 8086 will send out address 00002H. Since this is an even address, A0 will be low. However, since we are accessing only a byte, $\overline{BHE}$ will be high. The second byte will be read into the 8086 on lines D0–D7 and put in AH. Note that the 8086 automatically takes care of getting a byte to the correct register regardless of which data lines the byte comes in on.

The main reason that the A0 and $\overline{BHE}$ signals function

the way they do is to prevent the writing of an unwanted byte into an adjacent memory location when the 8086 writes a byte. To understand this, think what would happen if both memory banks were turned on for all write operations and you wrote a byte to address 00002 with the instruction MOV DS:BYTE PTR[0002],AL. The data from AL would be written to address 00002 as desired. However, if the upper bank were also enabled, the random data on D8–D15 would be written into address 00003. Since the 8086 is designed so that $\overline{BHE}$ is high during this byte write, the upper bank of memory is not enabled. This prevents the random data on D8–D15 from being written to address 00003.

Now that you have an overview of address decoding and of the 8086 memory banks, let's look at some examples of how all this is put together in a small system.

ROM ADDRESS DECODING ON THE SDK-86

Sheet 1 of the SDK-86 schematics in Figure 7-8 shows the circuit connections for the EPROMs and EPROM decoder. The 2716 EPROMs there are 2K × 8 devices. Two of the EPROMs have their eight data outputs connected in parallel to system data lines D0–D7. These two EPROMs then give 4 Kbytes of storage in the lower memory bank. The other two EPROMs have their data outputs connected in parallel to system data lines D8–D15 to give 4 Kbytes of storage in the upper bank of ROM.

Eleven address lines are needed to address the 2 Kbytes in each device. Therefore, system address lines A1–A11 are connected to all the EPROMs in parallel. Remember that A0 cannot be used to select a byte in the EPROMs because, as we described in the last section, it is used to enable or disable the lower bank.

A 2716 has two enable inputs, $\overline{CE}$ and $\overline{OE}$. In order for the 2716 to output an addressed byte, both of these enable inputs must be asserted low. The $\overline{CE}$ inputs of the two devices in the lower bank are connected to system address line A0, so the $\overline{CE}$ inputs of these devices will be asserted if A0 is low. The $\overline{CE}$ inputs of the two 2716s in the upper bank are connected to the $\overline{BHE}$ line. The $\overline{CE}$ inputs of these devices then will be asserted whenever $\overline{BHE}$ is asserted low. To summarize, then, the two devices labeled A27 and A36 form the lower bank of EPROMs and the two devices labeled A30 and A37 form the upper bank of EPROMs in this system. To see how the $\overline{OE}$ enable input of each of these devices gets asserted and to determine the address that each device will have in the system, you need to look next at the 3625 address decoder labeled A26 on sheet 1 of Figure 7-8.

A 3625 is a 1K × 4 bipolar PROM which functions as an address decoder, just as the 74LS138 performs in Figures 7-9 and 7-11. Since a 3625 has open collector outputs, a pull-up resistor to +5 V is required on each output. The dotted box around the four resistors on the schematic indicates the four are all contained in one package, resistor pack 5 (RP5). The 3625 translates an address to a signal which is used as part of the enabling of the desired device. Using a PROM as an address decoder, however, is for several reasons much more powerful than using a simple decoder such as the 74LS138. In the first place, the 3625 is programmable, which means that you can move the memory devices to new addresses in memory by simply programming a new PROM. Second, the large number of inputs on the PROM allows you to select a specific area of memory without using external gates. If, for example, you wanted the $\overline{G2A}$ input of a 74LS138 to be asserted if A11–A15 were all high, you would have to use an external NAND gate to detect this condition. With a PROM, you can just make this condition part of the truth table you use to burn the PROM.

Now, to analyze any address decoder circuit, first determine what signals are required to enable the decoder. The $\overline{CS1}$ enable input of the 3625 EPROM decoder is tied to ground, so it is permanently enabled. The $\overline{CS2}$ enable input is tied to the $\overline{RD}$ signal from the 8086, so that the decoder will only be enabled if the 8086 is doing a read operation. As explained previously, you don't want to accidentally enable a ROM if you send out a wrong address during a write operation.

The next step in analyzing a decoder circuit using a PROM is to consult the manufacturer's manual for the system. You have to do this because, for a PROM, the relationship between the inputs and the outputs cannot be determined directly from the schematic.

Figure 7-14 shows the truth table for the PROM from the SDK-86 manual. This truth table is just a compressed form of writing an address decoder worksheet such as those we used in the previous discussion of address decoding. From the truth table you can see that in order for the O1 output of the 3625 to be asserted low, $M/\overline{IO}$ has to be high. This is reasonable, since this decoder is enabling memory devices, not port devices. Also, address lines A12 through A19 have to be high in order for the O1 output of the PROM to be asserted low. Since the upper eight address bits must all be 1's for the O1 output to be asserted, the lowest address which

PROM INPUTS				PROM OUTPUTS				PROM ADDRESS BLOCK SELECTED
$M/\overline{IO}$	A14-A19	A13	A12	O4	O3	O2	O1	
1	1	1	1	1	1	1	0	FF000H-FFFFFH
1	1	1	0	1	1	0	1	FE000H-FEFFFH
1	1	0	1	1	0	1	1	FD000H-FDFFFH ($\overline{CSX}$)
1	1	0	0	0	1	1	1	FC000H-FCFFFH ($\overline{CSY}$)
ALL OTHER STATES				1	1	1	1	NONE

FIGURE 7-14 Truth table for an SDK-86 ROM decoder PROM (A26).

will cause this is FF000H. If you refer to sheet 1 of the SDK-86 schematics in Figure 7-8, you will see that the O1 output of the decoder PROM connects to the $\overline{OE}$ enable inputs of two of the 2716 EPROMs, A27 and A30. These $\overline{OE}$ outputs then will be enabled whenever the 8086 sends out an address in the range of FF000H to FFFFFH. To fully enable these devices, however, their $\overline{CE}$ inputs must also be asserted.

The $\overline{CE}$ input of the A27 EPROM is connected to system address line A0, so this device will be enabled whenever the 8086 does a memory read from an even address (A0 = 0) in the range FF000H to FFFFEH.

The $\overline{CE}$ enable input of A30 is connected to the system $\overline{BHE}$ line. As shown in Figure 7-13, $\overline{BHE}$ will be asserted low whenever the 8086 accesses a byte at an odd address or a word at an even address. Therefore, the A30 EPROM will be enabled when the 8086 reads a byte from an odd address in the range FF000H to FFFFFH. A30 will also be enabled when the 8086 asserts both A0 and $\overline{BHE}$ low to read a word that starts on an even address in the range FF000H to FFFFFH.

Note in Figure 7-14 and the first sheet of Figure 7-8 that the O2 output of the 3625 decoder PROM will be asserted for addresses in the range of FE000H to FEFFFH. This signal is used as part of the enabling for the A36 and A37 EPROMS. A0 and $\overline{BHE}$ provide the rest of the enabling for these devices, just as we described previously for A27 and A30 devices. For practice, trace these signals on the first sheet of Figure 7-8.

Also note in the first sheet of Figure 7-8 that the 3625 ROM decoder has two unused outputs which can be used as part of the enabling for EPROMs you add to the prototyping section of the board. As shown in Figure 7-14, the address ranges for these two outputs are FD000H to FDFFFH and FC000H to FCFFFH.

The four SDK-86 EPROMs actually contain two monitor programs. One monitor, in devices A27 and A30, allows you to use the hex keypad for entering and running programs. The other monitor, in devices A36 and A37, allows you to use an external CRT terminal to enter and run programs. The EPROMs are put at this high address in memory on the SDK-86 board because, after a RESET, the 8086 goes to address FFFF0H to get its first instruction. Since we want the SDK-86 to execute its monitor program after we press the RESET button, we locate the EPROM containing the monitor program such that this address is in it. You can interchange the actual EPROM devices so that either the keypad monitor or the serial monitor executes when you press the RESET button.

RAM ADDRESS DECODING ON THE SDK-86

To give you another example of memory address decoding in a real system, we now discuss the RAM decoding of the SDK-86 board. Sheet 6 of the SDK-86 schematics in Figure 7-8 shows the circuit for the system RAM and RAM decoder. Let's look at this schematic to see what we can learn from it.

First, take a look at the input and output lines on the 2142 static RAM devices. From the fact that each device has four data I/O lines, you can conclude that the devices store 4-bit words. The fact that each device has 10 address inputs, A0–A9, indicates that each one stores 2^{10} or 1024 of these 4-bit words. To store bytes, two 2142s are enabled in parallel. Devices A38 and A41, for example, are enabled together to store bytes from the lower eight data lines, and devices A43 and A45 are enabled together to store bytes from the upper eight data lines. Note next that the control bus signals $\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$ are connected to all the 2142s. $\overline{RD}$ is connected to the *output disable*, OD, pin on the 2142s. When the $\overline{RD}$ signal is high or when the device is not enabled, the output buffers will be disabled. During a read operation the $\overline{RD}$ signal is asserted low. If a 2142 is enabled and its OD input is low, the output buffers will be turned on so that an addressed word is output onto the data bus.

$\overline{WR}$ from the 8086 is connected to the *write enable*, $\overline{WE}$, input of the 2142s. If a 2142 is enabled, data on the data bus will be written into the addressed location in the RAM when the 8086 asserts $\overline{WR}$ low.

The 2142s have two *enable* inputs, $\overline{CS1}$ and CS2. The M/$\overline{IO}$ signal from the 8086 is connected to the CS2 input of all the 2142s. Since the CS2 input is active high, it will be asserted whenever the 8086 is doing a memory operation. The $\overline{CS1}$ inputs of the 2142s are connected in pairs to the outputs of a 3625 PROM which functions as an address decoder.

In order to assert any of its outputs and enable some RAM, the 3625 must itself be enabled. Since the $\overline{CS2}$ enable input of the 3625 PROM is tied to ground, it is permanently enabled. The $\overline{CS1}$ enable input will be asserted when system address line A19 is low. To determine any more information about this PROM, you need to look at the truth table for the device. Before we go on to that, however, note that A0 and $\overline{BHE}$ are connected to two of the address inputs on the 3625 PROM. Knowing what you do about 8086 memory banks, why do you think we want A0 and $\overline{BHE}$ to be part of what determines the outputs for this decoder? If you don't have the answer to this question, a look at the truth table for the device in Figure 7-15 should help you.

According to the third line of the truth table/address decoder worksheet in Figure 7-15, the O1 output of the PROM will be asserted low if A12 through A18 are low, A11 is low, $\overline{BHE}$ is high, and A0 is low. The O1 output then will be asserted for even system addresses starting with 00000H. A low on the O1 output will enable the A38 and A41 RAMs, which are connected to the lower half of the data bus. These two devices are part of the lower bank of RAM.

Next, look at the second line of the PROM truth table in Figure 7-15. From this line you should see that the O2 output of the PROM will be asserted low if A12 through A18 are low, A11 is low, $\overline{BHE}$ is low, and A0 is high. The O2 output will then be asserted for odd system addresses starting with 0001H. A low on the O2 output will enable the A43 and A45 RAMs, which are connected on the upper half of the data bus. These two devices are part of the upper bank of RAM.

Now, suppose we want to write a 16-bit word to RAM at an even address. To do this, we want both O1 and O2 to be asserted low so that both the lower-bank RAMs

PROM INPUTS				PROM OUTPUTS				BYTE(S) SELECTED (ADDRESS BLOCK)
A12-A18	A11	$\overline{BHE}$	A0	O4	O3	O2	O1	
0	0	0	0	1	1	0	0	BOTH BYTES (0H-07FFH)
0	0	0	1	1	1	0	1	HIGH BYTE (0H-07FFH)
0	0	1	0	1	1	1	0	LOW BYTE (0H-07FFH)
0	1	0	0	0	0	1	1	BOTH BYTES (0800H-0FFFH)
0	1	0	1	0	1	1	1	HIGH BYTE (0800H-0FFFH)
0	1	1	0	1	0	1	1	LOW BYTE (0800H-0FFFH)
ALL OTHER STATES				1	1	1	1	NONE

FIGURE 7-15 Truth table for an SDK-86 RAM decoder PROM (A29).

and the upper-bank RAMs are enabled. According to the first line of the PROM truth table in Figure 7-15, O1 and O2 will both be asserted low if $\overline{BHE}$ and A0 are both low. Remember from Figure 7-13 that $\overline{BHE}$ and A0 will both be low whenever you write a word to an even address or read a word from an even address. This last case gives the answer to the question we asked earlier about why A0 and $\overline{BHE}$ are connected to the address decoder PROM inputs. The two inputs are required to tell the PROM decoder to assert both O1 and O2 for a word read or write operation.

The address range for the A38, A41, A43, and A45 RAMs is 00000H to 007FFH. Another look at the PROM truth table in Figure 7-15 should show you that RAMS A39, A42, A44, and A46 contain 2 Kbytes more in the range 00800H to 00FFFH. Again, both banks of this additional RAM will be enabled if A0 and $\overline{BHE}$ are both low, as they are for reading or writing a word to an even address.

SDK-86 PORT ADDRESSING AND PORT DECODING

In a previous section of this chapter we described *memory-mapped input/output*. In a system with memory-mapped I/O, port devices are addressed and selected by decoders as if they were memory devices. The main advantage of memory-mapped I/O is that any instruction which refers to memory can theoretically be used to read from or write to a port. The single instruction ADD BH,DS:BYTE PTR[437AH] could be used to read a byte from a memory-mapped port and add the byte read in to the BH register. The disadvantage of memory-mapped I/O is that the ports occupy part of the system memory space. This space is then not available for storing data or instructions.

To avoid having to use part of the system memory space for ports, 8086 family microprocessors have a separate address space for ports. Having a separate address space for ports is called *direct I/O* because this separate address space is accessed directly with the IN and the OUT instructions.

Remember from previous chapters that the 8086 IN and OUT instructions each have two forms, *fixed* port and *variable* port. For fixed-port instructions, an 8-bit port address is written as part of the instruction. The instruction IN AL,38H, for example, copies a byte from port 38H to the AL register. For variable-port input or output operations, the 16-bit port address is first loaded into the DX register with an instruction such as MOV DX,0FFF8H. The instruction IN AL,DX is then used to copy a byte from port FFF8H to the AL register. MOV DX,0038H followed by IN AL,DX has the same effect as IN AL,38H.

Whenever the 8086 executes an IN or OUT instruction to access a port, none of the segment registers are involved in producing the physical address sent out by the 8086. The port address is sent out directly from the 8086 on lines AD0–AD15, and 0's are output on lines A16–A19.

In an 8086 system which uses direct I/O, the M/$\overline{IO}$ signal is used to enable a memory decoder or a port decoder. Remember that the M/$\overline{IO}$ signal being high was one of the enabling conditions for the SDK-86 ROM and RAM decoders we discussed in previous sections. As you will see, a low on M/$\overline{IO}$ is used to enable a port decoder.

During the execution of an IN instruction, the $\overline{RD}$ signal from the 8086 will be low. This signal can be used to enable an addressed input port device. During execution of an OUT instruction the $\overline{WR}$ signal from the 8086 will be low. This signal can be used to enable an addressed output port device. Since the 8086 outputs up to a 16-bit address for direct I/O operations, it can address any one of 2^{16} or 65,536 input ports and any one of 65,536 output ports.

For an example of how direct I/O ports are addressed and selected in a real system, we will again look at the SDK-86 schematics in Figure 7-8, sheet 7. Here another 3625 PROM, A22, is used to produce the chip select signals for four I/O devices. The O1 output of the PROM is used to enable the 8279 keyboard/display interface device, which we discuss in a section of Chapter 9. The O2 output of the PROM is used to enable the 8251A USART shown on sheet 9 of the schematics. The 8251A allows communication with other systems in serial form. A section in Chapter 14 discusses the operation of this device. The O3 and O4 outputs are connected to two 8255A parallel port devices, shown on sheet 5 of the schematics. These devices can be enabled individually to input or output bytes. They can also be enabled together to input or output words. A section in Chapter 9 shows you how to tell each port in these devices whether you want it to be an input or an output.

Take a look now at the 3625 decoder PROM to determine what conditions enable it. You should find that

PROM INPUTS						PROM OUTPUTS*			
A11-A15	A5-A10	A4	A3	$\overline{BHE}$	A0	O4 HIGH PORT SELECT	O3 LOW PORT SELECT	O2 USART SELECT	O1 KDSEL
1	1	0	1	0	0	1	1	1	0
1	1	0	1	1	0	1	1	1	0
1	1	1	0	0	0	1	1	0	1
1	1	1	0	1	0	1	1	0	1
1	1	1	1	0	0	0	0	1	1
1	1	1	1	0	1	0	1	1	1
1	1	1	1	1	0	1	0	1	1
ALL OTHER STATES						1	1	1	1

(*a*)

PORT ADDRESS	PORT FUNCTION
0000 to FFDF	OPEN
FFE8	READ/WRITE 8279 DISPLAY RAM OR READ 8279 FIFO
E9	
EA	READ 8279 STATUS OR WRITE 8279 COMMAND
EB	
EC	RESERVED.
ED	
EE	RESERVED
FFEF	
FFF0	READ/WRITE 8251A DATA
F1	
F2	READ 8251A STATUS OR WRITE 8251A CONTROL
F3	
F4	RESERVED
F5	
F6	RESERVED
FFF7	
FFF8	READ/WRITE 8255A PORT P2A
F9	READ/WRITE 8255A PORT P1A
FA	READ/WRITE 8255A PORT P2B
FB	READ/WRITE 8255A PORT P1B
FC	READ/WRITE 8255A PORT P2C
FD	READ/WRITE 8255A PORT P1C
FE	WRITE 8255A P2 CONTROL
FFFF	WRITE 8255A P1 CONTROL

(*b*)

FIGURE 7-16 Truth table and map for SDK-86 port decoder. (*a*) Truth table. (*b*) Map.

the $\overline{CS2}$ enable input of the PROM will be asserted when M/$\overline{IO}$ is low, as it is during an input or output operation. Furthermore, you should see that the $\overline{CS1}$ input will be asserted when A11 to A15 are all high. Now, to see what addresses cause each of the PROM outputs to be asserted, refer to the truth table for the PROM in Figure 7-16*a*. From this figure you can see that to assert the O1 output low, A5 through A15 have to be high, A4 has to be low, A3 has to be high, and A0 has to be low. $\overline{BHE}$ can be either high or low. Note, however, that only the lower eight data lines, D0–D7, are connected to the 8279. Therefore, data must be sent to or read from the 8279 at an even byte address. In other words, data must be sent as a byte to an even address or as the lower byte of a word to an even address.

The system base address for this device then is FFE8H. System address line A1 is connected to the 8279 to select one of two internal addresses in the device. A1 low selects one internal address, and A1 high selects the other internal address. A1 low gives system address FFE8H, and A1 high gives system address FFEAH. These are then the two addresses for the 8279 in this system.

According to the truth table in Figure 7-16*a*, the O2 output of the decoder PROM will be asserted low when A4 through A15 are high and A3 and A0 are low. $\overline{BHE}$ can be either low or high, but, since only the lower eight data lines are connected to the 8251A USART, data must be sent to or read from the device as bytes at an even address. Again, system address line A1 is used to select one of two internal addresses in the 8251A (Figure 7-8, sheet 9). A1 low selects one internal address and A1 high selects the other internal address. Therefore, the two system addresses for this device are FFF0H and FFF2H.

Now, before discussing the O3 and O4 outputs of the decoder PROM, we will take a brief look at the two 8255

parallel port devices they enable. These devices are shown on sheet 5 of the schematics in Figure 7-8. Each of these devices contains three 8-bit parallel ports and a control register. System address lines A1 and A2 are used to address the desired port or register in the device, just as lower address lines are used to address the desired internal location in a memory device. Note that the lower eight data lines, D0–D7, are connected to the A40 device, and the upper eight data lines are connected to the A35 device. This is done so that you have several input or output possibilities. You can read a byte from or write a byte to an even-addressed port in device A40. You can read a byte from or write a byte to an odd-addressed port in device A35. You can read a word from or write a word to a 16-bit port made up from an 8-bit port from device A40 and an 8-bit port from device A35. To input or output a word, both devices have to be enabled. Now let's look at the decoder truth table to determine what addresses enable the various ports in these devices.

The A40 device will be enabled by the O3 output of the 3625 decoder PROM if address lines A3 through A15 are high and A0 is low. A1 and A2 are used to select internal ports of the 8255A. Let's assume that these two bits are 0 for the first address in the device. To select the A port in the A40 8255A, address lines A1 and A2 have to be low. The system address that will enable this device and select the A port within it is FFF8H. Other values of A2 and A1 will select one of the other ports or the control register in this device. Figure 7-16*b* shows the system addresses for the ports and control register in this 8255. Note that the ports in this device (A40) are identified as port 2A, port 2B, and port 2C. These all have even addresses because A0 must be low for this device to be selected.

The A35 8255A, which contains port 1A, port 1B, and port 1C, will be enabled by the O4 output of the decoder PROM if A3 through A15 are high and the $\overline{BHE}$ line is low. If this 8255A is being enabled for a byte read or write, then the A0 line will also be high. A2 and A1 are again used to address one of the ports or the control register within the 8255A. A2 = 0 and A1 = 1 will select port 1A in this 8255A. As shown in Figure 7-16*b*, then, the system address for port 1A is FFF9H. Port 1B will be accessed with a system address of FFFBH, port 1C will be accessed with a system address of FFFDH, and the internal control register will be accessed with a system address of FFFFH.

As we said before, the 8086 can input a 16-bit value in one operation by enabling a port device on the lower half of the data bus and a port device on the upper half of the data bus at the same time. When the 8086 on an SDK-86 board executes the instruction sequence MOV DX,FFF8H–IN AX,DX, both A0 and $\overline{BHE}$ will be low during the IN instruction. As shown by the fifth line in the truth table, this will cause both the O3 and the O4 outputs of the port decoder to be low. These signals will enable both the A40 and A35 port devices. The byte of data on port 2A will be input to the 8086 on the lower half of the data bus, and the byte of data on port 1A will be input to the 8086 on the upper half of the data bus.

Note in the truth table in Figure 7-16*a* that the 3625 PROM decoder will enable a port device only when the specific address assigned to that device is sent out by the 8086. This is sometimes called *complete decoding* because all the address lines play a part in selecting a device and one of its internal ports or registers. As we show in Chapter 8, adding another decoder to produce enable signals for more port devices is very easy in a system which uses this complete decoding.

THE SDK-86 "OFF-BOARD" DECODER

Take a look at the *off-board circuitry* in zone A5 on sheet 5 of the SDK-86 schematics. The purpose of this circuitry is to produce the signal $\overline{OFF\ BOARD}$ whenever the 8086 sends out a memory or port address which does not correspond to a device decoded on the board. The $\overline{OFF\ BOARD}$ signal will be asserted low if pin 4 of the A3 NAND gate is low or if pin 5 of the A3 NAND gate is low. According to the truth table for the A12 PROM in Figure 7-17, the O1 output will be low if the 8086 is doing a memory operation and the address sent out is not in one of the ranges decoded for the onboard RAM or ROM.

In order for pin 4 of the A3 NAND gate to be low, both pin 9 and pin 10 of the A3 NAND gate must be high. Pin 10 will be high if the 8086 is doing an input or output operation (IO/$\overline{M}$ from the 8286 inverting buffer equals 1). Pin 9 of the A3 NAND gate will be high if any one of the A19 NAND gate inputs is low. Since system address lines A5 through A15 are connected to the inputs of the 74LS133 NAND gate, the signal to pin 9 of A3 will be high for any address less than FFE0H. In other words, pin 4 of the A3 NAND gate will be asserted low for any I/O operation in an address range not selected by the A22 port decoder.

The $\overline{OFF\ BOARD}$ signal produced by the previously discussed PROM and logic gates is connected to an input of a NAND gate labeled A2 on sheet 2 of the schematics. If $\overline{OFF\ BOARD}$ is asserted low, or $\overline{INTA}$ is asserted low, or HLDA is asserted low, the output of this gate will be high. For now, all we are interested in is the fact that if $\overline{OFF\ BOARD}$ is asserted low, a high will be applied to

PROM INPUTS									PROM OUTPUT (O1)	CORRESPONDING ADDRESS BLOCK
M/$\overline{IO}$	A19	A18	A17	A16	A15	A14	A13	A12		
1	0	0	0	0	0	0	0	0	1 (INACTIVE)	0H-0FFFH (ON-BOARD RAM)
1	1	1	1	1	1	1	1	0	1 (INACTIVE)	FE000H-FEFFFH (ON-BOARD PROM)
1	1	1	1	1	1	1	1	1	1 (INACTIVE)	FF000H-FFFFFH (ON-BOARD PROM)
ALL OTHER STATES									0 (ACTIVE)	01000H-FDFFFH (OFF-BOARD)

FIGURE 7-17 SDK-86 off-board decoder PROM truth table.

pin 1 of the A3 NAND gate in zone A4 of the schematic. If the $\overline{\text{DEN}}$ signal from the 8086 is also asserted low, the signal labeled $\overline{\text{BUFFER ON}}$ will be asserted low. The $\overline{\text{DEN}}$ signal from the 8086 will be asserted whenever the 8086 reads in data from a memory location or a port or when it writes data to a memory location or a port. The $\overline{\text{BUFFER ON}}$ signal produced here is used to enable the 8286 data bus buffers (A6 and A7) shown on sheet 4 of the schematics. Now here's the point of all this.

In the next chapter we show you how to add another I/O decoder and some other devices to the prototyping area of an SDK-86 board. To drive these additional devices, the address, data, and control buses must all be buffered. The address bus on the SDK-86 board is buffered by the 74S373 address latches shown on sheet 3 of the schematics. Data bus and control bus buffers are not needed to drive the ROM, RAM, and port devices that come with the SDK-86 board. To read data from or write data to external devices, however, the data bus is buffered by two 8286s, shown as A7 and A6 on sheet 4 of the SDK-86 schematics. These two buffers are turned on when the $\overline{\text{BUFFER ON}}$ signal, described in the preceding paragraph, is asserted low. The 8286 buffers are bidirectional. When these buffers are enabled, the *Data Transmit/Receive* signal, DT/$\overline{\text{R}}$, from the 8086 will determine in which direction the buffers are pointed. If DT/$\overline{\text{R}}$ is high, the buffers will be enabled to write data to some external device. If DT/$\overline{\text{R}}$ is low, the buffers will be enabled to read data in from some external device.

The control bus signals are buffered by an 8286 labeled A11 and a 74LS244 labeled A8 on sheet 4 of the SDK-86 schematics. These buffers are permanently enabled to send out the control bus signals except during a HOLD state, which we will explain later.

THE SDK-86 WAIT-STATE GENERATOR CIRCUITRY

Now that you know how the $\overline{\text{OFF BOARD}}$ signal is produced on the SDK-86 board, we can explain the operation of the *WAIT-state generator circuitry* shown on sheet 2 of the schematics.

In a previous section of the chapter we showed you that if the RDY input of the 8086 is asserted low, the 8086 will insert one or more WAIT states in the machine cycle it is currently executing. Figure 7-1*b* shows how a WAIT state is inserted in an 8086 machine cycle. During a WAIT state, the information on the buses is held constant. The signal levels on the buses at the start of the WAIT state remain there throughout the WAIT state. The main purpose of inserting one or more WAIT states in a machine cycle is to give an addressed memory device or I/O device more time to accept or output data. In the next major section of the chapter, we show you how to determine whether a WAIT state is needed for a given device with a given 8086 clock frequency. For now, however, let's just see how the circuitry on the SDK-86 board causes the 8086 to insert a selected number of WAIT states.

WAIT states are inserted by pulling the RDY1 input of the 8284 clock generator IC low (Figure 7-8, sheet 2, zone D5). The 8284 internally synchronizes the RDY1 input signal with the clock signal and sends the resultant signal to the RDY input of the 8086. For the SDK-86, the RDY1 input will be asserted low if all three inputs of the A15 NAND gate shown in zone D5 of the schematic are high. Pin 10 of this device is tied to +5 V, so it is permanently high. Pin 11 of A15 will be high if any of the inputs of the NAND gate in zone D7 are asserted low. Pin 1 of gate A15 will be low whenever the 8086 does an input or output operation. Pin 2 of gate A15 will be low whenever the 8086 accesses a port or memory location which is not decoded on the board. In other words, with these connections, the selected number of WAIT states will be inserted in each machine cycle when the 8086 does a read from or a write to an on-board I/O device or when the 8086 does a read from or a write to any device not decoded on the board. If jumper W39 is installed on pin 13 of A15, pin 11 of A15 will always be high. The number of WAIT states selected by the W27–W34 jumpers will be inserted for all read and write operations.

The desired number of WAIT states to be inserted is selected by putting a jumper between two pins in the W27–W34 matrix shown in zone D3 (sheet 2) of the schematic. If a jumper is installed in the W27 position, for example, no WAIT states will be inserted. If a jumper is installed in the W28 position, one WAIT state will be inserted. The pattern continues to jumper W34, which will cause seven WAIT states to be inserted in each machine cycle. Here's how the WAIT-state generator itself works.

The 74LS164 WAIT-state generator is an 8-bit shift register. At the start of a machine cycle, the $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{INTA}}$ signals from the 8086 are all high. These three signals being high will cause the A2 NAND gate in zone C4 to assert the clear input, CLR, of the shift register. The outputs of the shift register will then all be low. One of these lows will be coupled through a jumper and an inverter to pin 9 of the A15 NAND gate we discussed previously. This high on pin 9, together with a high on pin 11, will cause the RDY1 input of the 8284 to be pulled low. However, WAIT states will not be inserted unless RDY1 remains low long enough. Now, when $\overline{\text{RD}}$, $\overline{\text{WR}}$, or $\overline{\text{INTA}}$ goes low in the machine cycle, the $\overline{\text{CLR}}$ input of the 74LS164 shift register will go high, and the shift register will function normally. The highs on the INA and INB inputs will be loaded onto the QA output on the next positive edge of the clock. If the WAIT-state jumper is in the W27 position, then this high on the QA output will, through the inverter and NAND gate, cause the RDY1 input of the 8284 to go high again. For this case, the RDY1 input goes high soon enough that no WAIT states are inserted.

The high loaded into the 74LS164 shift register is shifted one stage to the right by each successive clock pulse. When the high reaches the jumper connected to the A25 inverter, it will cause the RDY1 input of the 8284 to go high. The 8086 will then exit from a WAIT state on the next clock pulse. The number of WAIT states inserted in a machine cycle is determined by how many stages the high has to be shifted before it reaches the installed jumper.

To summarize all this, the 8086 will insert the selected number of WAIT states in any machine cycle which accesses any device not addressed on the board or any

I/O device on the board. If jumper W39 is inserted, the selected number of WAIT states will be inserted for any onboard or off-board access. The purpose of inserting WAIT states is to give the addressed device more time to accept or output data.

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 Figure 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 $\overline{BHE}$ 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 Figure 7-1*b*. 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, pp. 198-9, shows some timing waveforms and parameters for an 8086 minimum-mode read machine cycle.

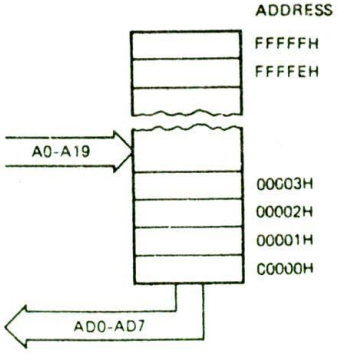


FIGURE 7-18 8088 memory structure.

As you look at Figure 7-19*a*, 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.

The edges of the clock signal cause operations in the 8086 to occur; therefore, as you can see in Figure 7-19*a*, the clock waveform is used as a reference for other times. The timing values for when the 8086 puts out M/$\overline{IO}$, 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}, of 450 ns. This means that if an address is already present on the address inputs of the 2716 and the output buffers are already enabled, the 2716 will put valid data on its outputs no later than 450 ns after the $\overline{CE}$ input is asserted low. A third parameter given for the 2716 in the data book is an output enable to output time, t_{OE}, of 120 ns maximum. This means that if the device already has an address on its address inputs, and its $\overline{CE}$ input is already asserted, valid data will appear on the output pins at most 120 ns after the $\overline{OE}$ pin is asserted low.

Now that you have these three parameters for the 2716, the next step is to check whether each one of

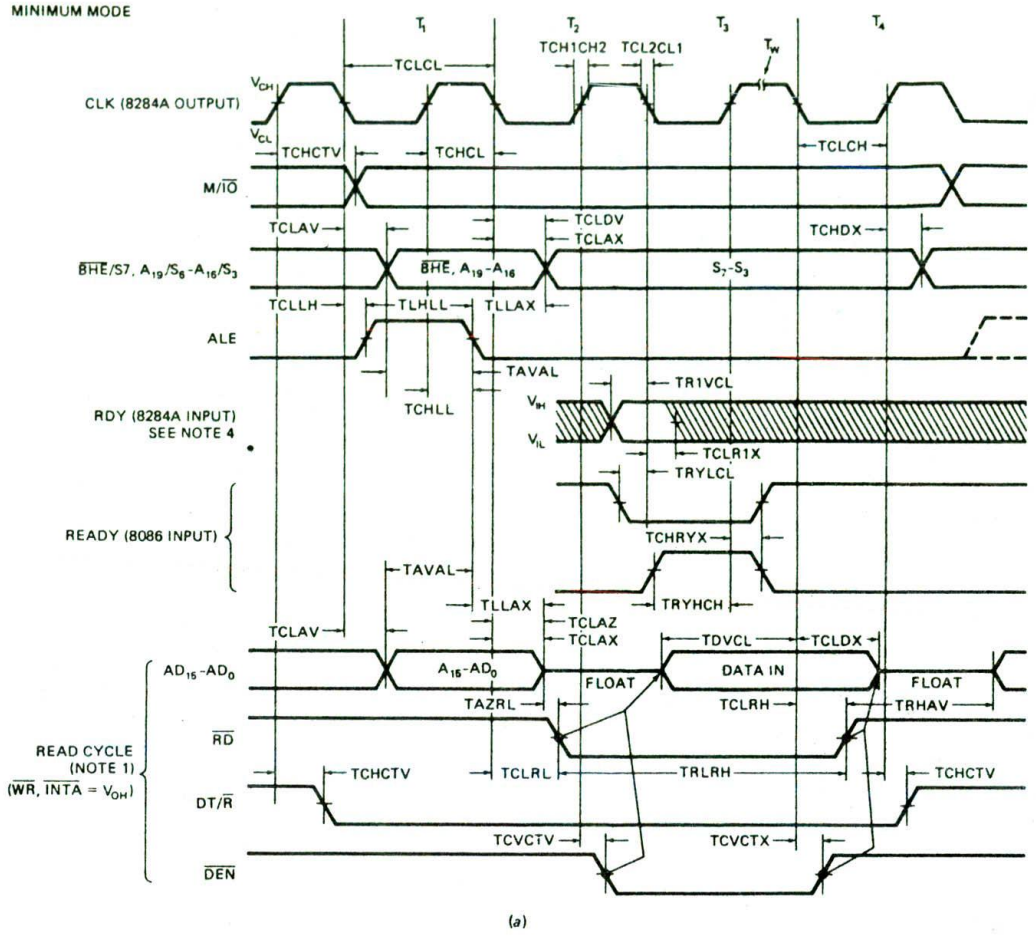


FIGURE 7-19 8086 minimum-mode timing waveforms and parameters. (a) Read waveforms. (*See also next page.*)

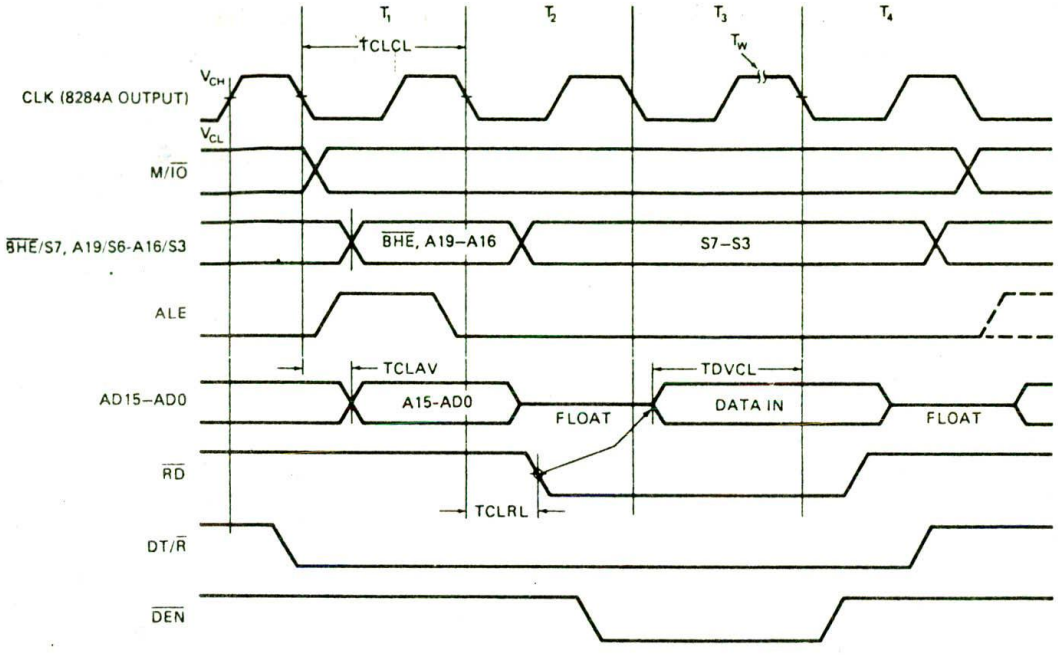
these times is short enough for the device to work with a 4.9-MHz 8086. In other words, does the 2716 put out valid data soon enough after it is addressed and enabled to satisfy the requirements of the 8086? To determine this, you need to look at both the 8086 timing parameters and how the 2716 is addressed and enabled on the SDK-86 board.

To make it easier for you to find the important parameters for these calculations, we show in Figure 7-19*b* a simplified version of the timing diagram in Figure 7-19*a*. You should try to do this simplification mentally whenever you are faced with a timing diagram. As shown by the timing diagram in Figure 7-19*b*, the 8086 sends out M/$\overline{IO}$, $\overline{BHE}$, and an address during T_1 of the machine cycle. Note on the AD15–AD0 lines of the timing diagram that the 8086 outputs this information within a time labeled TCLAV after the falling edge of the clock at the start of T_1. TCLAV stands for *time from clock low to address valid*. According to the 8086 column of the data sheet shown in Figure 7-19*c*, the maximum value of this time is 110 ns.

Now look further to the right on the AD15–AD0 lines. You should see that valid data must arrive at the 8086 from memory a time TDVCL before the falling edge of the clock at the end of T_3. TDVCL stands for *time data must be valid before clock goes low*. The data sheet gives a value of 30 ns for this parameter.

The time between the end of the TCLAV interval and the start of the TDVCL interval is the time available for getting the address to the memory and for the t_{ACC} of the memory device. You can determine this time by subtracting TCLAV and TDVCL from the time for three clock cycles. With a 4.9-MHz clock, each clock cycle will be 204 ns. Three clock cycles then total 612 ns. Subtracting a TCLAV of 110 ns and a TDVCL of 30 ns leaves 472 ns available for getting the address to the 2716 and for its t_{ACC}. To help you visualize these times, Figure 7-20*a*, p. 200, shows this operation in simplified diagram form.

If you look at sheets 1 and 3 of the SDK-86 schematics, you should see that the $\overline{BHE}$ signal and the A0–A11 address information go from the 8086 through the



(*b*)

MINIMUM COMPLEXITY SYSTEM TIMING REQUIREMENTS

SYMBOL	PARAMETER	8086		8086-1 (Preliminary)		8086-2		UNITS
		MIN.	MAX.	MIN.	MAX.	MIN.	MAX.	
TCLCL	CLK Cycle Period	200	500	100	500	125	500	ns
TDVCL	Data in Setup Time	30		5		20		ns
TCLAV	Address Valid Delay	10	110	10	50	10	60	ns
TCLRL	$\overline{RD}$ Active Delay	10	165	10	70	10	100	ns

NOTE: Complete timing information in Appendix

(*c*)

FIGURE 7-19 (*continued*) (b) Simplified read waveforms. (c) Timing parameters. (*Intel Corporation*)

74S373 latches to get to the 2716s. The propagation delay of the 74S373s then must be subtracted from the 472 ns to determine how much time is actually available for the t_{ACC} of the 2716. The maximum delay of a 74S373 is 12 ns. As shown in Figure 7-20*a*, subtracting this from the 472 ns leaves 460 ns for the t_{ACC} of the 2716. Now, as we told you in a previous paragraph, the 2716 has a maximum t_{ACC} of 450 ns. Since 450 ns is less than the 460 ns available, you know that the t_{ACC} of the 2716 is acceptable for the SDK-86 operating with a 4.9-MHz clock. You still, however, must check if the values of t_{CE} and t_{OE} for the 2716 are acceptable.

If you look at sheet 1 of the SDK-86 schematics, you should see that the $\overline{CE}$ inputs of the 2716s are connected either to A0 or to $\overline{BHE}$. The timing for these signals is the same as that for the addresses in the preceding section. As shown in Figure 7-20*a*, the time available for t_{CE} of the 2716 will be 460 ns. Since the maximum t_{CE} of the 2716 is 450 ns, you know that this parameter is also acceptable for an SDK-86 operating with a 4.9-MHz clock.

The final parameter to check is t_{OE} of the 2716. According to sheet 1 of the SDK-86 schematics, the $\overline{OE}$ signals for the 2716s are produced by the 3625 decoder. The signals coming to this decoder are A12 through A19, M/$\overline{IO}$, and $\overline{RD}$. Look at the 8086 timing diagram in Figure 7-19*b* to see if you can determine which of these signals arrives last at the 3625. You should find that addresses and M/$\overline{IO}$ are sent out during T_1, but $\overline{RD}$ is not sent out until T_2. As indicated by the arrow from the falling edge of the $\overline{RD}$ signal, $\overline{RD}$ going low causes the address decoder to send an $\overline{OE}$ signal to the 2716 EPROMs. Since $\overline{RD}$ is sent out so much later than addresses, it will be the limiting factor for timing. $\overline{RD}$ going low and the EPROM returning valid data must occur within the time of states T_2 and T_3. Now, according

to the timing diagram, RD is sent out from the 8086 within a time TCLRL after the falling edge of the clock at the start of T_2. From the data sheet, the maximum value of TCLRL is 165 ns. As we discussed before, the 8086 requires that valid data arrive on AD0 through AD15 from memory a time TDVCL before the falling edge of the clock at the end of T_3. The minimum value of TDVCL from the data sheet is 30 ns. The time between the end of the TCLRL interval and the start of the TDVCL interval is the time available for the $\overline{OE}$ signal to be produced and for the $\overline{OE}$ signal to turn on the memory. To determine the actual time available for these operations, first compute the time for states T_2 and T_3. For a 4.9-MHz clock, each clock cycle or state will be 204 ns, so the two together total 408 ns. Then subtract the TCLRL of 165 ns and the TDVCL of 30 ns. As shown by the simple diagram in Figure 7-20*b*, this leaves 213 ns available for the decoder delay and the t_{OE} of the 2716. Checking a data sheet for the 3625 would show you that it has a maximum $\overline{CS2}$ to output delay of 30 ns. Subtract this from the available 213 ns to see how much time is left for the t_{OE} of the 2716. The result of this subtraction is 183 ns.

As we indicated in a previous paragraph, the 2716 has a maximum t_{OE} of 120 ns. Since this time is considerably less than the 183 ns available, the 2716 has an acceptable t_{OE} value for operating on the SDK-86 board with a 4.9-MHz clock.

All three times for the 2716 are less than those required by the 8086 for 5-MHz operation, so you know that the devices will work correctly at 4.9 MHz without inserting a WAIT state. You could use a logic analyzer as we described earlier in the chapter to verify the timing on an actual SDK-86 board.

Here's a final point about calculating the time available for t_{ACC}, t_{CE}, and t_{OE} of some device in a system. Suppose that you want to add another pair of 2716 EPROMs in the prototyping area of the SDK-86 board, and you want to enable the outputs of these added devices with the O3 output of the 3625 ROM decoder on sheet 1 of the schematics. The timing for these added devices will be the same as for the previously discussed 2716s, except that the data from the added devices must come back through the 8286 buffers shown on sheet 4 of the SDK-86 schematics. According to an 8286 data sheet, these buffers have a maximum delay of 30 ns. This 30 ns must be subtracted from the times available for t_{ACC}, t_{CE}, and t_{OE}. If you look back at our calculations of the time available for t_{ACC} in Figure 7-20*a*, for example, you will see that we ended up with 460 ns available for t_{ACC}. Subtracting the 30 ns of buffer delay from this leaves only 430 ns, which is considerably less than the maximum t_{ACC} of 450 ns for the 2716. This tells you that, because of the buffer delay, the added 2716s are not fast enough to operate on an SDK-86 board with a 4.9-MHz clock and no WAIT states. To take care of this problem, the SDK-86 is designed so that any access to a memory or I/O device "off board" will cause the selected number of WAIT states to be inserted in the machine cycle. For our example here, selecting one WAIT state with jumper W28 on sheet 2 will give another 204 ns for the data to get from the 2716s to the 8086. This is more than enough time to compensate for the buffer delay, so the added 2716s will work correctly.

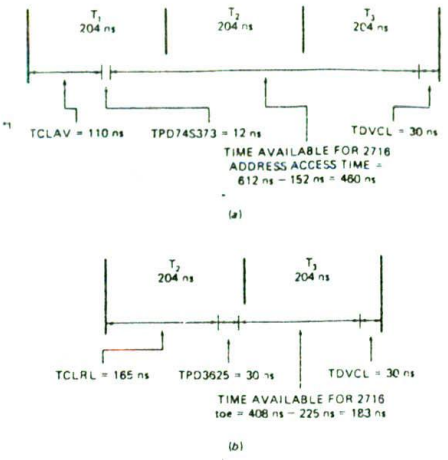


FIGURE 7-20 Calculations of maximum allowable access times for 4.9-MHz 8086. (a) Time for t_{ACC} and t_{RD}. (*b*) Time for t_{OE}.

TROUBLESHOOTING A SIMPLE 8086-BASED MICROCOMPUTER

Now that you have some knowledge of the software and the hardware of a microcomputer system, we can start teaching you how to troubleshoot a simple microcomputer system such as an SDK-86 board. For this section assume that the microcomputer or microprocessor-based instrument previously worked. Later sections of this book will describe how the prototype of a microprocessor-based instrument is developed.

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 $\overline{RD}$, $\overline{WR}$, 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 normal operation. If an external logic gate fails and holds RDY low, the 8086 will go on inserting WAIT states forever, and the buses will be held constant. If the 8086 HOLD input is stuck high or the RST input is held high, the 8086 address/data bus will be floating. Connecting a scope probe to these lines will pull them to ground, so you will see them as constant lows.

If there is activity on the buses, use an oscilloscope to see if the CPU is putting out control signals such as $\overline{RD}$ and $\overline{WR}$. Also, check with your oscilloscope to see if select signals are being generated on the outputs of the ROM, RAM, and port decoders as the system attempts to run its monitor or basic program. If no select signals are being produced, then the address decoder may be bad or the CPU may not be sending out the correct addresses.

After a little practice, you should be able to work through the previously described steps quite quickly. If you have not located the problem at this point, the next step for a system with its ICs in sockets is to systematically substitute known good ICs for those in the nonworking system.

Systematically Substitute Socketed ICs

The easiest case of substitution is that where you have two identical microcomputers, one that works and one that doesn't, and the ICs of both units are in sockets. For this case you can use the working system to test the ICs from the nonworking system. The trick here is to do this in such a way that you don't end up with two systems that do not work! Here's how you do it.

First of all, *do not remove or insert any ICs with the power on!* With the power off, remove the CPU from the good system and put it in a piece of conductive foam. Plug the CPU from the bad system into the now empty socket on the good board and turn on the power. If the good system still works, then the CPU is probably good. Turn off the power and put the CPU back in the bad system. If the good system does not work with the CPU from the bad system, then the CPU is probably bad. Remove it from the good system and bend the pins so that you know it is bad. If the CPU seems bad, you can try replacing it with the CPU you removed from the good system. If you do this, however, it is important that you keep track of which IC came from which system. To do this, we like to mark each IC from the good system with a wide-tip, water-soluble marking pen. We can then rebuild the good system by simply putting back all the marked ICs. The marks on the ICs can easily be removed with a damp cloth.

The procedure from here on is to keep testing ICs from the bad system until you find all the bad ICs. Make sure you turn the power off before you remove or insert any ICs. Be aware that more than one IC may be bad. It is not unusual, for example, for an AC power line surge to wipe out several devices in a system. We usually work

our way out from the CPU to address latches, buffers, decoders, and memory devices. Often the specific symptoms point you to the problem group of ICs without your having to test every IC in the system. If, for example, the system accesses ROM but doesn't access RAM, suspect the RAM decoder. If a system uses buffers on the buses, suspect these devices. Buffers are high-current devices, and they often fail.

Troubleshoot Soldered-in ICs

The approach described in the preceding paragraphs works well if the system ICs are all in sockets and you have two identical systems. However, since sockets add to the cost and unreliability of a system, many small systems put only the CPU and ROMs in sockets. This makes your troubleshooting work harder but not impossible.

Again, if you have two identical systems, one that works and one that doesn't work, you can attempt to run the monitor or basic system program on each and compare signals on the two. A missing or wrong signal may point you to the bad IC or ICs.

If the system works enough to read some instructions from ROM and execute them, you can replace the monitor or basic system ROM with one that contains diagnostic programs which test RAM and I/O devices. A RAM test routine, for example, might attempt to write all 1's to each RAM location and then read each memory location to see if the data was written correctly to that location. If the data read back is not correct, the diagnostic program can stop and in some way indicate the address that it could not write to. If a write of all 1's is successful, then the test routine will try to write all 0's to each memory location. A port test routine might initialize a port for output and then write alternating 1's and 0's to the port over and over again. With an oscilloscope you can see if the port device is getting enabled and if the data is getting to the output of the port device. Another port test routine might try to read a byte of data in from a port over and over so that you can again see if the device is getting enabled and if the data is getting through the device to the system data bus. The technique of using program routines to test hardware is a very important one that you will use many times when you are working with microcomputer systems.

Now, suppose that you have localized the problem to a few ICs that are soldered in. If the problem is one that occurs when the unit gets hot, you might try spraying some cold spray on the ICs, one at a time, to see if you can determine which one has a problem. If this does not find the bad IC or the problem is not heat-related, what you do next is replace these ICs one at a time until the system works correctly. The point we want to stress here is that the cost of these few ICs is probably much less than the cost of the time it would take you to determine just which IC is bad, if you do not have specialized test equipment.

If you do not have special tools available to remove a "through-hole mounted" IC from a printed-circuit board, *do not* attempt to desolder pins with a hand-held solder "slurper." Modern multilayer printed-circuit boards are quite fragile, and these tools can slip and knock a trace right off the board. Instead, use cutters with narrow tips to cut all the leads of the IC next to the body. Since you are going to throw it out anyway, you don't care if you destroy the IC. With the body of the IC out of the way, you can then gently heat each pin individually and use needle-nose pliers to remove it from the PC board. If the hole fills with solder, heat it gently and insert a small wooden toothpick until the solder cools. After you replace each IC, power up the system and see if it now works.

To remove "surface-mount" ICs, use a tool such as that shown in Figure 7-21. This tool sends out a directed blast of hot air which heats all the pins at the same time and allows the IC to be easily removed. To replace the IC, you put some solder paste on the PC board pads for the IC, place the IC carefully in position, and heat the pins with another blast of hot air.

The techniques described in the preceding sections will enable you to troubleshoot many microcomputer systems with a minimum of test equipment. However, specialized test equipment is available to speed up the process and help find complex problems. The following sections describe two of these instruments.

Equipment for Troubleshooting Microcomputers

LOGIC ANALYZER

A logic analyzer can be a powerful tool for debugging difficult problems, but it is important for you to have a perspective on when to use an analyzer in troubleshooting simple systems that previously worked. Generally you can use the techniques described in previous sec-

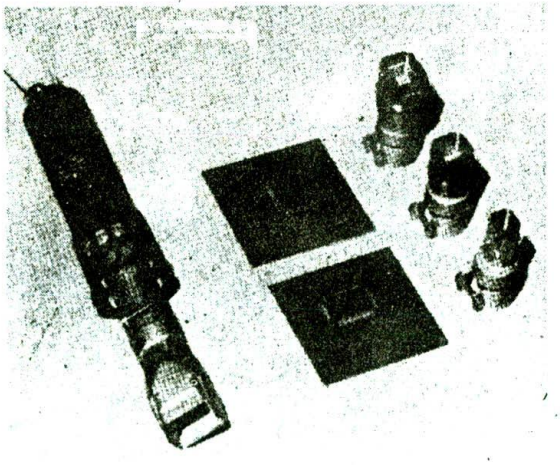


FIGURE 7-21 Leister-Labor S hot-air contactless desoldering and soldering tool for removing and replacing leaded and surface mount components on PC boards. (*Courtesy Brian R. White Co., Inc., Ukiah, California.*)

tions to find and fix a problem in less time than it would take you to connect the logic analyzer, figure out what you should see in a trace, and determine if the trace is correct.

One of the main problems is that in a repair setting you often don't have good documentation on an instrument, so it is difficult to determine what the correct trace should be. Analyzers such as the Tektronix 1230 allow you to store a trace from a functioning instrument in a reference memory. This trace can then be compared with a trace from a nonfunctioning instrument. We have found this feature very helpful in pointing to the source of a problem.

The disassembly feature found in some analyzers is also useful, because it allows you to determine if a microcomputer-based instrument is correctly fetching and executing its basic control program.

Despite the minor difficulties, don't hesitate to use an analyzer when the simple techniques don't seem to be getting you anywhere.

OTHER MICROCOMPUTER TROUBLESHOOTING EQUIPMENT

A logic analyzer is a very powerful troubleshooting tool, but to use it effectively, you need some detailed knowledge and a program listing for the system that you are trying to troubleshoot. If you are working as a repair technician and have to repair several different types of microcomputer systems with poor documentation to work from, most analyzers are not too useful. To make your life easier in this case, "smart" instruments such as the Fluke 9010A Microsystem troubleshooter have been created.

As you can see from the picture of the 9010A in Figure 7-22, it has a keyboard, a display, and an "umbilical" cable with an IC plug on the end. The unit also contains a minicassette tape recorder. For troubleshooting, the 9010A is used as follows.

The microprocessor in a fully functioning unit is removed, and the plug at the end of the cable is inserted in its place. The learn function of the 9010A is then executed. This function finds and maps ROM, RAM, and I/O registers that can be written into and read from. It also computes signatures (checksums) for blocks of ROM. All these parameters are stored in the 9010A's RAM and/or on a minicassette tape. The microprocessor on a malfunctioning unit is then removed and the plug at the end of the umbilical cable inserted in its place. An automatic test function is then executed. In this mode, the 9010A tests the buses, RAM, ROM, ports, power supply, and clock on the malfunctioning system. Any problem found, such as stuck nodes or adjacent trace short circuits, is indicated on the display. The results of this test give some good hints as to the source of the problem. Because of its built-in intelligence, the 9010A can be programmed to do other tests as well.

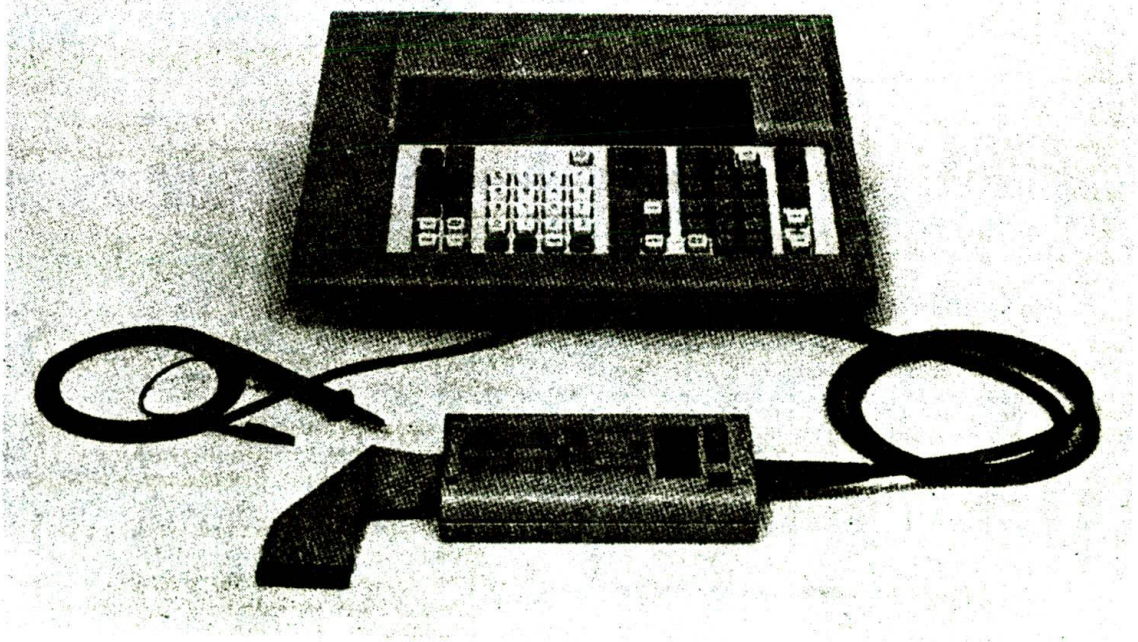


FIGURE 7-22 Fluke 9010A microsystem troubleshooter. (*John Fluke Mfg. Co., Inc.*)

The point of an instrument such as the 9010A is that with it you do not have to be intimately familiar with the programming language and hardware details of a simple microcomputer system in order to troubleshoot it.

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.

Pin functions of 8086:
V_{CC}, $\overline{RD}$, $\overline{WR}$, CLK, ALE, M/$\overline{IO}$, $\overline{LOCK}$, MN/$\overline{MX}$, RESET, NMI, INTR, $\overline{BHE}$, $\overline{DEN}$, DT/$\overline{R}$

8086 RESET response

Maximum and minimum mode of 8086

8086 timing diagram interpretation

State, instruction cycle, machine cycle, WAIT state, RDY signal

Bus activities during read/write

Logic analyzer use: external clock, internal clock, word recognizer, trigger, trace

Bidirectional buffer

General functions: 8284, 8255A, 8251A, 8279, 2716, 2142

SDK-86 schematic: zones, plugs, jacks, resistor packs

Address decoding: ROM decoding, RAM decoding, port decoding

Memory-mapped and direct I/O

8086 memory banks

Timing parameters: t_{ACC}, t_{CL}, t_{OE}, t_{CE}, TCLAV, TCLRL, TDVCL

8086 typical clock frequencies

Troubleshooting steps for a simple 8086-based microcomputer

REVIEW QUESTIONS AND PROBLEMS

1. From what point on the clock waveform is the start of an 8086 state measured?
2. Why are latches required on the AD0–AD15 bus in an 8086 system?
3. What is the purpose of the ALE signal in an 8086 system?
4. Describe the sequence of events on the 8086 data/address bus, the ALE line, the M/$\overline{IO}$ line, and the $\overline{RD}$ line as the 8086 fetches an instruction word.
5. What logic levels will be on the 8086 $\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$ lines when the 8086 is doing a write to a memory location? A read from a port?
6. What is the major difference between an 8086 operating in minimum mode and an 8086 operating in maximum mode?
7. Describe the response an 8086 will make when its RESET (RST) input is asserted high.
8. Why are buffers often needed on the address, data, and control buses in a microcomputer system?
9. *a.* How is an 8086 entered into a WAIT state?
   b. At what point in a machine cycle does an 8086 enter a WAIT state?
   c. What information is on the buses during a WAIT state?
   d. How long is a WAIT state?
   e. How many WAIT states can be inserted in a machine cycle?
   f Why would you want the 8086 to insert a WAIT state?
10. What are the functions of the 8086 DT/$\overline{R}$ and $\overline{DEN}$ signals?
11. What does an arrow going from a transition on one signal waveform to a transition on another tell you?
12. Draw a block diagram of a simple logic analyzer and briefly describe how it operates. Include in your answer the function of the clock and the function of the trigger.
13. What do you use for a logic analyzer clock when you want to make detailed timing measurements?
14. On what signal and what edge of that signal would you clock a logic analyzer, and on what word would you trigger to see each of the following in an 8086 system?
   a. The sequence of addresses output after a RESET.
   b. The sequence of instructions read in after a RESET. (Assume that the first instruction word is 9CEAH.)
   c. Both the addresses sent out and the words read in.

d. What clock qualifier would you use to see a trace of only data read in from ports?

15. How is it possible for a logic analyzer to display data that occurred before the trigger?

16. How are wire-wrap jumpers indicated on a schematic?

17. What is the meaning of /8 on a signal line in a schematic?

18. Describe the two purposes of address decoders in microcomputer systems.

19. A memory device has 15 address lines connected to it and 8 data outputs. What size words and how many words does the device store?

20. Briefly describe the function of the 8255, 8251A, and 8279 devices in the SDK-86 microcomputer system.

21. A group of signal lines in a schematic has the label 2ZB3 next to it. What is the meaning of this label?

22. What is the difference between a connector identified with a J and a connector identified with a P?

23. Describe the purpose of the many small capacitors connected between V_{CC} and ground on microcomputer printed-circuit boards.

24. A 74LS138 decoder has its three SELECT inputs connected to A12, A13, and A14 of the system address bus. It has $\overline{G2A}$ connected to A15, $\overline{G2B}$ connected to $\overline{RD}$, and G1 connected to +5 V. Use an address decoder worksheet to determine what eight ROM address blocks the decoder outputs will select. Why is $\overline{RD}$ used as one of the enables on a ROM decoder?

25. Show a memory map for the ROMs in Problem 24.

26. Use an address decoder worksheet to help you draw a circuit to show how another 74LS138 can be connected to select one of eight 1-Kbyte RAMs starting at address 8000H.

27. Why are there actually many addresses that will select one of the port devices connected to the port decoder in Figure 7-12*a*?

28. Describe memory-mapped I/O and direct I/O. Give the main advantage and main disadvantage of each.

29. *a.* Why is the 8086 memory set up as 2-byte-wide banks?
    b. What logic levels would you find on $\overline{BHE}$ and A0 when an 8086 is writing a byte to address 04274H? When it is writing a word to 04274H?
    c. Describe the 8086 bus operations required to write a word to address 04373H.

30. How does the circuitry on the SDK-86 make sure that you cannot accidentally write a byte or word to ROM?

31. Why is some ROM put at the top of the address space in an 8086 system?

32. *a.* Show the truth table you would use for a 3625 PROM decoder to produce $\overline{CS1}$ signals for 4K × 8 RAMs in an 8086 system. Assume the first RAM starts at address 00000H. Don't forget A0 and $\overline{BHE}$.
    b. Draw the circuit connections for the 3625 decoder PROM and for two of the 4K × 8 RAMs.

33. Use sheets 5 and 7 of the SDK-86 schematics to help you determine for the SDK-86 what logic levels will be on $\overline{BHE}$, A0 to A19, M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ when a word is read from ports FFF8H and FFF9H. Are these ports memory-mapped or direct? What instruction(s) would you use to do this read operation?

34. *a.* How is the $\overline{OFF\ BOARD}$ signal produced on the SDK-86 board?
    b. Describe the purpose of the $\overline{OFF\ BOARD}$ signal.

35. Describe how the 8088 memory is configured. Why doesn't the 8088 need a $\overline{BHE}$ signal?

36. By referring to the 8086 timing diagrams in Figure 7-19*a* and parameters in Appendix A, determine for the 8086-2:
    a. The maximum clock frequency.
    b. The time between CLOCK going low and $\overline{RD}$ going low.
    c. The time for which memory must hold data on the data bus after CLOCK goes low at the start of T_4.
    d. The time that the lower 16 address bits remain on the data bus after ALE goes low.

37. The 27128-25 is a 16K × 8 EPROM with a t_{ACC} of 250 ns maximum, a t_{CE} of 250 ns maximum, and a t_{OE} of 100 ns maximum. Will this device work correctly without WAIT states in an 8-MHz 8086-2 system with circuit connections such as those in the SDK-86 schematics? Assume the address latches have a propagation delay of 12 ns and the decoder has a delay of 30 ns.

38. List the major steps you would take to troubleshoot a microcomputer system such as the SDK-86 which previously worked. Assume all ICs are in sockets.

39. Why is it important to check power supplies with an oscilloscope?

40. Describe how you can keep from mixing up ICs from a good system with those from a bad system when substituting.

41. Write an 8086 routine to test the system RAM in addresses 00200H through 07FFFH.

42. Write a test routine to output alternating 1's and 0's to port FFFAH over and over. With this routine running, you could check with an oscilloscope to see if the port device is getting enabled and is outputting data.

43. Describe the symptoms that an SDK-86 would show for each of the following problems.
    a. Pin 8 of A15 in zone D5 of schematic sheet 2 is stuck low.
    b. The reset key is stuck on.
    c. None of the outputs of A29 in zone D7 of schematic sheet 6 ever goes low.
    d. Pin 6 of A3 in zone A5 of schematic sheet 5 is stuck low.

CHAPTER 8

8086 Interrupts and Interrupt Applications

Most microprocessors allow normal program execution to be interrupted by some external signal or by a special instruction in the program. In response to an interrupt, the microprocessor stops executing its current program and calls a procedure which "services" the interrupt. An IRET instruction at the end of the interrupt-service procedure returns execution to the interrupted program. This chapter introduces you to the 8086 interrupt types, shows you how the microprocessors in the 8086 family respond to interrupts, teaches you how to write interrupt-service procedures, and describes how interrupts are used in a variety of applications.

OBJECTIVES

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

1. Describe the interrupt response of an 8086 family processor.
2. Initialize an 8086 interrupt vector (pointer) table.
3. Write interrupt-service procedures.
4. Describe the operation of an 8254 programmable counter/timer and write the instructions necessary to initialize an 8254 for a specified application.
5. Describe the operation of an 8259A priority interrupt controller and write the instructions needed to initialize an 8259A for a specified application.
6. Call a BIOS procedure using a software interrupt.

8086 INTERRUPTS AND INTERRUPT RESPONSES

Overview

An 8086 interrupt can come from any one of three sources. One source is an external signal applied to the *nonmaskable interrupt* (NMI) input pin or to the *interrupt* (INTR) input pin. An interrupt caused by a signal applied to one of these inputs is referred to as a *hardware interrupt*.

A second source of an interrupt is execution of the Interrupt instruction, INT. This is referred to as a *software interrupt*.

The third source of an interrupt is some error condition produced in the 8086 by the execution of an instruction. An example of this is the divide-by-zero interrupt. If you attempt to divide an operand by zero, the 8086 will automatically interrupt the currently executing program.

At the end of each instruction cycle, the 8086 checks to see if any interrupts have been requested. If an interrupt has been requested, the 8086 responds to the interrupt by stepping through the following series of major actions.

1. It decrements the stack pointer by 2 and pushes the flag register on the stack.
2. It disables the 8086 INTR interrupt input by clearing the interrupt flag (IF) in the flag register.
3. It resets the trap flag (TF) in the flag register.
4. It decrements the stack pointer by 2 and pushes the current code segment register contents on the stack.
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, p. 208, 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

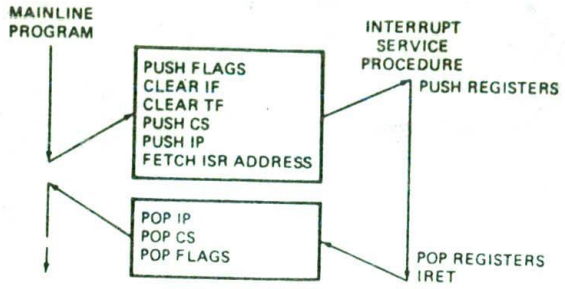


FIGURE 8-1 8086 interrupt response.

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*.

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

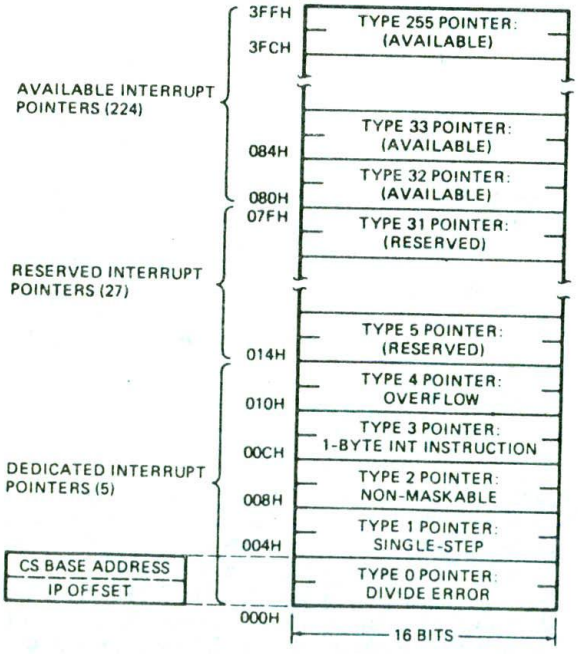


FIGURE 8-2 8086 interrupt-pointer table.

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 Figure 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 in detail and show you how to write a procedure to service that interrupt.

An 8086 Interrupt Response Example—Type 0

Probably the easiest 8086 interrupt to understand is the divide-by-zero interrupt, identified as *type 0* in Figure 8-2. Before we get into the details of the type 0 interrupt response, let's refresh your memory about how the 8086 DIV and IDIV instructions work.

The 8086 DIV instruction allows you to divide a 16-bit unsigned binary number in AX by an 8-bit unsigned number from a specified register or memory location. The 8-bit result (quotient) from this division will be left in the AL register. The 8-bit remainder will be left in the AH register. The DIV instruction also allows you to divide a 32-bit unsigned binary number in DX and AX by a 16-bit number in a specified register or memory location. The 16-bit quotient from this division is left in the AX register, and the 16-bit remainder is left in the DX register. In the same manner, the 8086 IDIV instruction allows you to divide a 16-bit signed number in AX by an 8-bit signed number in a specified register or a 32-bit signed number in DX and AX by a 16-bit signed number from a specified register or memory location.

If the quotient from dividing a 16-bit number is too large to fit in AL or the quotient from dividing a 32-bit number is too large to fit in AX, the result of the division will be meaningless. A special case of this is where an attempt is made to divide a 32-bit number or a 16-bit number by zero. The result of dividing by zero is infinity (actually undefined), which is somewhat too large to fit in AX or AL. Whenever the quotient from a DIV or IDIV operation is too large to fit in the result register, the 8086 will automatically do a type 0 interrupt. In response to this interrupt the 8086 proceeds as follows.

The 8086 first decrements the stack pointer by 2 and copies the flag register to the stack. It then clears IF and TF. Next, it saves the return address on the stack. To do this, the 8086 decrements the stack pointer by 2, pushes the CS value of the return address on the stack, decrements the stack pointer by 2 again, and pushes the IP value of the return address on the stack. The 8086 then gets the starting address of the interrupt-service procedure from the type 0 locations in the

interrupt-vector table. As you can see in Figure 8-2, it gets the new value for CS from addresses 00002H and 00003H and the new value for IP from addresses 00000H and 00001H. After the starting address of the procedure is loaded into CS and IP, the 8086 then fetches and executes the first instruction of the procedure.

At the end of the interrupt-service procedure, an IRET instruction is used to return execution to the interrupted program.

The IRET instruction pops the stored value of IP off the stack and increments the stack pointer by 2. It then pops the stored value of CS off the stack and increments the stack pointer again by 2. Finally, it restores the flags by popping off the stack the values stored during the interrupt response and increments the stack pointer by 2. Remember from the previous paragraph that during its interrupt response, the 8086 disables the INTR and single-step interrupts by clearing IF and TF. If the INTR input and/or the trap interrupt were enabled before the interrupt, they will be enabled upon return to the interrupted program. The reason for this is that flags from the interrupted program were pushed on the stack before IF and TF were cleared by the 8086 in its interrupt response. To summarize, then, IRET returns execution to the interrupted program and restores IF and TF to the state they were in before the interrupt. Now that we have described the type 0 response, we can show you how to write a program to handle this interrupt.

An 8086 Interrupt Program Example

DEFINING THE PROBLEM AND WRITING THE ALGORITHM

In the last chapter we were working mostly with hardware, so instead of jumping directly into the program, let's use this example to review how you go about writing any program.

For the example program here, assume we have four word-sized hexadecimal values stored in memory. We want to divide each of these values by a byte-type scale factor to give a byte-type scaled value. If the result of the division is valid, we want to put the scaled value in an array in memory. If the result of the division is invalid (too large to fit in the 8-bit result register), we want to put 0 in the array for that scaled value. Figure 8-3 shows the algorithm for this program in pseudocode.

As shown in Figure 8-3*a*, the mainline part of this program gets each 16-bit value from memory in turn and divides that value by the 8-bit scale factor. If the result of the division is valid, it is stored in the appropriate memory location; else a 0 is stored in the memory location. Not indicated in the algorithm is how we determine whether the quotient is valid or not. With 8086 family microprocessors, a type 0 interrupt procedure is a handy way to do this.

Remember from the preceding discussion that if the result of the division is too large to fit in the quotient register, AL, then the 8086 will do a type 0 interrupt immediately after the divide instruction finishes. Figure 8-3*b* shows the algorithm for a procedure to service this type 0 interrupt. The main function of this procedure is to set a *flag* which will be checked by the mainline

```
INITIALIZATION LIST

REPEAT
  Get INPUT_VALUE
  Divide by scale factor
  If result valid THEN
    store result as scaled value
  ELSE store zero
UNTIL all values scaled
```

(*a*)

```
Save registers
Set error flag
Restore registers
Return to mainline
```

(*b*)

FIGURE 8-3 Algorithm for divide-by-zero program example. (*a*) Mainline program. (*b*) Interrupt-service procedure.

program. The flag in this case is not one of the flags in the 8086 flag register. The flag here is a bit in a memory location we set aside for this purpose. In the actual program, we give this memory location the name BAD_DIV_FLAG. At the end of the interrupt-service procedure, execution is returned to the interrupted mainline program.

After the divide operation in the mainline program, we check the value of the BAD_DIV_FLAG to determine if the result of the division is valid. If the result of the division was too large, then the 8086 will have done a type 0 interrupt, and the interrupt-service procedure will have set the BAD_DIV_FLAG to a 1. If the result of the division is valid, then the 8086 will not have done the type 0 interrupt, and the BAD_DIV_FLAG will be 0.

This sequence of operations is repeated until all the values have been scaled.

WRITING THE INITIALIZATION LIST

After you have worked out the data structure and the algorithm for a program, the next step is to make an initialization list such as the one shown in Chapter 3. Here is a list for this program.

1. Initialize the interrupt-vector table. In other words, the starting address of our type 0 interrupt service routine must be put in locations 00000H and 00002H.
2. Set up the data segment where the values to be scaled, the scale factor, the scaled values, and the BAD_DIV_FLAG will be put.
3. Initialize the data segment register to point to the base address of the data segment containing the values to be scaled.
4. Set up a stack to store the flags and return address.
5. Initialize the stack segment and stack pointer registers.
6. Initialize a pointer to the start of the data to be scaled, a counter to keep track of how many values have been scaled, and a pointer to the start of the array where the scaled values are to be written.

```
 1                                  ;8086 MAINLINE PROGRAM F8-04A.ASM
 2                                  ;ABSTRACT   : Program scales data values using division.
 3                                  ;PORTS      : None used
 4                                  ;REGISTERS  : Uses CS,DS,ES,SS,SP,SI,AX,BX,CS
 5                                  ;PROCEDURES : Uses BAD_DIV, a type 0 interrupt service procedure
 6                                  ;             Link mailine F8-04A.OBJ with procedure F8-04B.OBJ
 7
 8 0000                             DATA SEGMENT  WORD    PUBLIC
 9 0000  0035 0855 2011 1359            INPUT_VALUES  DW 0035H, 0855H, 2011H, 1359H
10 0008  04*(00)                        SCALED_VALUES DB 4 DUP(0)  ; Answers = 05,ED,00,00
11 000C  09                             SCALE_FACTOR  DB 09
12 000D  00                             BAD_DIV_FLAG  DB 0
13 000E                            DATA ENDS
14
15 0000                            STACK_SEG  SEGMENT STACK
16 0000  64*(0000)                         DW  100 DUP (0)        ; Set up stack of 100 words
17                                         TOP_STACK  LABEL  WORD ; Pointer to top of stack
18 00C8                            STACK_SEG  ENDS
19
20                                 PUBLIC   BAD_DIV_FLAG          ; Make flag available to other modules
21
22 0000                            INT_PROC  SEGMENT  WORD  PUBLIC
23                                     EXTRN  BAD_DIV:FAR         ; Let assembler know procedure BAD_DIV
24 0000                            INT_PROC  ENDS                 ; is in another assembly module
25
26 0000                            CODE SEGMENT WORD  PUBLIC
27                                     ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
28 0000  B8 0000s                  START: MOV  AX, STACK_SEG       ; Initialize stack segment
29 0003  8E D0                            MOV  SS, AX              ; register
30 0005  BC 00C8r                         MOV  SP, OFFSET TOP_STACK ; Initialize stack pointer
31 0008  B8 0000s                         MOV  AX, DATA            ; Initialize data segment
32 000B  8E D8                            MOV  DS, AX              ; register
33                                 ;Store the address for the BAD_DIV routine at address 0000:0000
34                                 ;Address 00000-00003 is where type 0 interrupt gets interrupt
35                                 ;service procedure address. CS at 00002 & 00003, IP at 00000 & 00001
36 000D  B8 0000                          MOV  AX, 0000
37 0010  8E C0                            MOV  ES, AX
38 0012  26: C7 06 0002 0000s             MOV  WORD PTR ES:0002, SEG BAD_DIV
39 0019  26: C7 06 0000 0000e             MOV  WORD PTR ES:0000, OFFSET BAD_DIV
40 0020  BE 0000r                         MOV  SI, OFFSET INPUT_VALUES  ; Initialize pointer for input values
41 0023  BB 0008r                         MOV  BX, OFFSET SCALED_VALUES ; Point BX at start of result array
42 0026  B9 0004                          MOV  CX, 0004                 ; Initialize data value counter
43 0029  8B 04                     NEXT:  MOV  AX, [SI]                 ; Bring a value to AX for divide
44 002B  F6 36 000Cr                      DIV  SCALE_FACTOR             ; Divide by scale factor
45 002F  80 3E 000Dr 01                   CMP  BAD_DIV_FLAG, 01         ; If divide produced valid result
46 0034  75 06                            JNE  OK                       ; then go save scaled value
47 0036  C6 07 00                         MOV  BYTE PTR [BX], 00        ; else load 0 as scaled value
48 0039  EB 03 90                         JMP  SKIP
49 003C  88 07                     OK:    MOV  [BX], AL                 ; Save scaled value
50 003E  C6 06 000Dr 00            SKIP:  MOV  BAD_DIV_FLAG, 0          ; Reset BAD_DIV_FLAG
51 0043  83 C6 02                         ADD  SI, 02                   ; Point at next input value location
52 0046  43                               INC  BX                       ; Point at location for next result
53 0047  E2 E0                            LOOP NEXT                     ; Repeat until all values done
54 0049  90                        STOP:  NOP
55 004A                            CODE   ENDS
56                                        END  START
```

(*a*)

FIGURE 8-4 8086 assembly language program for divide-by-zero example. (*a*) Mainline. (*See also next page.*)

Once you have the algorithm and the initialization list for a program, the next step is to start writing the instructions for the program, so now let's look at the assembly language program for this problem.

ASSEMBLY LANGUAGE PROGRAM AND INTERRUPT PROCEDURE

Figure 8-4 shows our 8086 assembly language program for the mainline and for the type 0 interrupt service procedure. You can use many parts from these examples when you write your own interrupt programs. Also, to help refresh your memory of the PUBLIC and EXTRN directives, we have written the mainline program and the interrupt service procedure as two separate assembly modules.

At the start of the mainline program in Figure 8-4*a*, we declare a segment named DATA for the data that the program will be working with. The WORD in this statement tells the linker-locator to locate this segment on the first available even address. The PUBLIC in this statement tells the linker that this segment can be joined

```
 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                     DATA  SEGMENT WORD PUBLIC
 9                           EXTRN  BAD_DIV_FLAG:BYTE   ; Let assembler know BAD_DIV_FLAG
10 0000                     DATA  ENDS                 ; is in another assembly module
11
12                          PUBLIC  BAD_DIV            ; Make procedure BAD_DIV available
13                                                     ; to other assembly modules
14 0000                     INT_PROC SEGMENT WORD PUBLIC ; Segment for interrupt service procedure
15 0000                     BAD_DIV  PROC FAR          ; Procedure for type 0 interrupt
16                           ASSUME CS:INT_PROC, DS:DATA
17 0000  50                    PUSH  AX                ; Save AX of interrupted program
18 0001  1E                    PUSH  DS                ; Save DS of interrupted program
19 0002  B8 0000s              MOV   AX, DATA          ; Load Data Segment register value
20 0005  8E D8                 MOV   DS, AX            ; needed here
21 0007  C6 06 0000e 01        MOV   BAD_DIV_FLAG, 01  ; Set LSB of BAD_DIV_FLAG byte
22 000C  1F                    POP   DS                ; Restore DS of interrupted program
23 000D  58                    POP   AX                ; Restore AX of interrupted program
24 000E  CF                    IRET                    ; Return to next instruction in interrupted
25 000F                     BAD_DIV  ENDP              ; program
26 000F                     INT_PROC ENDS
27                                  END
```

(b)

FIGURE 8-4 (*continued*) (b) Interrupt-service procedure.

together (concatenated) with segments of the same name from other assembly modules. The input values are words, so we use a DW directive to declare these four values. The scaled values will be bytes, so we use a DB directive to set aside four locations for them. Remember that the DUP(0) in the statement initializes the 4-byte locations to all 0's. As the program executes, the results will be written into these locations. SCALE_FACTOR DB 09H sets aside a byte location for the number by which we are going to be dividing the input values. The advantage of using a DB rather than an EQU directive to declare the scale factor is that with a DB the value of the scale factor can be held in RAM, where it can be changed dynamically in the program as needed. If you use a statement such as SCALE_FACTOR EQU 09H to set a value, you have to reassemble the program to change the value.

Part of the 8086 interrupt response is essentially a far call to the interrupt service procedure. In any program that calls a procedure, we have to set up a stack to store the return address and parameters passed to and from the procedure. The next section of the program declares a stack segment called STACK_SEG. It also establishes a pointer to the next location above the stack with the statement TOP_STACK LABEL WORD. Remember from the examples in Chapter 5 that this label is used to initialize the stack pointer to the next location after the top of the stack.

The next two parts of the program are necessary because we wrote the mainline program and the interrupt service procedure as two separate assembly modules. When the assembler reads through a source program, it 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] copies an input value from memory to the AX register, where it has to be for the divide operation. The DIV SCALE_FACTOR instruction divides the number in AX by 09H, the value we assigned to SCALE_FACTOR previously with a DB directive. The 8-bit quotient from this division will be put in AL, and the 8-bit remainder will be put in AH. If the quotient is too large to fit in AL, then the 8086 will automatically do a type 0 interrupt. For our program here, the 8086 will push the flags on the stack, reset IF and TF, and push the return address on the stack. It will then go to addresses 0000H and 0002H to get the IP and CS values for the start of BAD_DIV, the procedure we wrote to service a type 0 interrupt. It will then execute the BAD_DIV procedure. Now let's look at the procedure in Figure 8-4*b* and see how it works.

The BAD_DIV procedure starts by letting the assembler know that the name BAD_DIV_FLAG represents a variable of type byte and that this variable is defined in a segment called DATA in some other (EXTRN) assembly module. We also tell the assembler that the label BAD_DIV should be made available to other assembly modules (PUBLIC).

Next, we declare a logical segment called INT_PROC. We could have put this procedure in the segment CODE with the mainline program. However, in system programs where there are many interrupt-service procedures, a separate segment is usually set aside for them. The statement BAD_DIV PROC FAR identifies the actual start of the procedure and tells the assembler that both the CS and IP values for this procedure must be saved.

Now, an important operation to do at the start of any interrupt-service procedure is to push on the stack any registers that are used in the procedure. You can then restore these registers by popping them off the stack just before returning to the interrupted program. The interrupted program will then resume with its registers as they were before the interrupt. In the procedure in Figure 8-4*b*, we saved AX and DS. Since we use the same data segment, DATA, in the mainline and in the procedure, you may wonder why we saved DS. The point is that an interrupt-service procedure should be written so that it can be used at any point in a program. By saving the DS value for the interrupted program, this interrupt-service procedure can be used in a program section that does not use DATA as its data segment.

The ASSUME statement tells the assembler the name of the segment to use as a data segment, but remember that it does not load the DS register with a value for the start of that segment. The instructions MOV AX,DATA and MOV DS,AX do this in our procedure.

Finally, we get to the whole point of this procedure with the MOV BAD_DIV_FLAG,01 instruction. This instruction simply sets the least significant bit of the memory location we set aside with a DB directive at the start of the mainline program. Note that in order to access this variable by name, you have to let the assembler know that it is external, and you have to make sure that the DS register contains the segment base for the segment in which BAD_DIV_FLAG is located.

To complete the procedure, we pop the saved registers off the stack and return to the interrupted program. The IRET instruction, remember, is different from the regular RET instruction in that it pops the flag register and the return address off the stack. Note in the program in Figure 8-4*b* that the procedure must be "closed" with an ENDP directive, and the segment must as usual be closed with an ENDS directive.

Now let's look back in the mainline to see what it does with this BAD_DIV_FLAG. Immediately after the DIV instruction, the mainline checks to see if the BAD_DIV_FLAG is set by comparing it with 01. If the BAD_DIV_FLAG was not set by the type 0 interrupt-service procedure, then a jump is made to the MOV [BX],AL instruction. This instruction copies the result of the division in AL to the memory location in SCALED_VALUES pointed to by BX. If BAD_DIV_FLAG was set by a type 0 interrupt, then 0 is put in the memory location in SCALED_VALUES and a jump will be made to the MOV BAD_DIV_FLAG,00 instruction, which resets the BAD_DIV_FLAG. Since this jump passes over the MOV [BX],AL instruction, the invalid result of the division will not be copied into one of the locations in SCALED_VALUES.

After putting the scaled value or 0 in the array and resetting the flag, we get ready to operate on the next input value. The ADD SI,02 instruction increments SI by 2 so that it points to the next 16-bit value in INPUT_VALUES. The INC BX instruction points BX at the next 8-bit location in SCALED_VALUES. The LOOP instruction after these automatically decrements the CX register by 1 and, if CX is not then 0, causes the 8086 to jump to the specified label, NEXT.

The preceding section has shown you how to set up an interrupt-pointer table, how to write an interrupt-service procedure, and how the 8086 responds to a type 0 interrupt. Now we can discuss some of the other types of 8086 interrupts.

8086 Interrupt Types

The preceding sections used the type 0 interrupt as an example of how the 8086 interrupts function. In this section we discuss in detail the different ways an 8086 can be interrupted and how the 8086 responds to each of these interrupts. We discuss these in order, starting with type 0, so that you can easily find a particular discussion when you need to refer back to it. However, as you read through this section, you should not attempt to learn all the details of all the interrupt types at once.

Read through all the types to get an overview, and then focus on the details of the hardware-caused NMI interrupt, the software interrupts produced by the INT instruction, and the hardware interrupt produced by applying a signal to the INTR input pin.

DIVIDE-BY-ZERO INTERRUPT—TYPE 0

As we described in the preceding section, the 8086 will automatically do a type 0 interrupt if the result of a DIV operation or an IDIV operation is too large to fit in the destination register. For a type 0 interrupt, the 8086 pushes the flag register on the stack, resets IF and TF, and pushes the return address (CS and IP) on the stack. It then gets the CS value for the start of the interrupt-service procedure from address 00002H in the interrupt-pointer table and the IP value for the start of the procedure from address 00000H in the interrupt pointer-table.

Since the 8086 type 0 response is automatic and cannot be disabled in any way, you have to account for it in any program where you use the DIV or IDIV instruction. One way is to in some way make sure the result will never be too large for the result register. We showed one way to do this in the example program in Figure 5-27*b*. In that example, you may remember, we first make sure the divisor is not zero, and then we do the division in several steps so that the result of the division will never be too large.

Another way to account for the 8086 type 0 response is to simply write an interrupt-service procedure which takes the desired action when an invalid division occurs. The advantage of this approach is that you don't have the overhead of a more complex division routine in your mainline program. The 8086 automatically does the checking and does the interrupt procedure only if there is a problem.

SINGLE-STEP INTERRUPT—TYPE 1

In a section of Chapter 3 on debugging assembly language programs, we discussed the use of the single-step feature found in some monitor programs and debugger programs. When you tell a system to single-step, it will execute one instruction and stop. You can then examine the contents of registers and memory locations. If they are correct, you can tell the system to go on and execute the next instruction. In other words, when in single-step mode, a system will stop after it executes each instruction and wait for further direction from you. The 8086 trap flag and *type 1* interrupt response make it quite easy to implement a single-step feature in an 8086-based system.

If the 8086 trap flag is set, the 8086 will automatically do a type 1 interrupt after each instruction executes. When the 8086 does a type 1 interrupt, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the type 1 interrupt-service procedure from address 00006H and it gets the IP value for the start of the procedure from address 00004H.

The tasks involved in implementing single stepping are: Set the trap flag, write an interrupt-service procedure which saves all registers on the stack, where they can later be examined or perhaps displayed on the CRT, and load the starting address of the type 1 interrupt-service procedure into addresses 00004H and 00006H. The actual single-step procedure will depend very much on the system on which it is to be implemented. We do not have space here to show you the different ways to do this. We will, however, show you how the trap flag is set or reset, because this is somewhat unusual.

The 8086 has no instructions to directly set or reset the trap flag. These operations are done by pushing the flag register on the stack, changing the trap flag bit to what you want it to be, and then popping the flag register back off the stack. Here is the instruction sequence to set the trap flag.

```
PUSHF                   ; Push flags on stack
MOV BP,SP               ; Copy SP to BP for use as index
OR WORD PTR[BP+0],0100H
                        ; Set TF bit
POPF                    ; Restore flag register
```

To reset the trap flag, simply replace the OR instruction in the preceding sequence with the instruction AND WORD PTR [BP+0], OFEFFH.

NOTE: We have to use [BP+0] because BP cannot be used as a pointer without a displacement. See Figure 3-8.

The trap flag is reset when the 8086 does a type 1 interrupt, so the single-step mode will be disabled during the interrupt-service procedure.

NONMASKABLE INTERRUPT—TYPE 2

The 8086 will automatically do a *type 2* interrupt response when it receives a low-to-high transition on its NMI input pin. When it does a type 2 interrupt, the 8086 will push the flags on the stack, reset TF and IF, and push the CS value and the IP value for the next instruction on the stack. It will then get the CS value for the start of the type 2 interrupt-service procedure from address 0000AH and the IP value for the start of the procedure from address 00008H.

The name *nonmaskable* given to this input pin on the 8086 means that the type 2 interrupt response cannot be disabled (masked) by any program instructions. Because this input cannot be intentionally or accidentally disabled, we use it to signal the 8086 that some condition in an external system must be taken care of. We could, for example, have a pressure sensor on a large steam boiler connected to the NMI input. If the pressure goes above some preset limit, the sensor will send an interrupt signal to the 8086. The type 2 interrupt-service procedure for this case might turn off the fuel to the boiler, open a pressure-relief valve, and sound an alarm.

Another common use of the type 2 interrupt is to save program data in case of a system power failure. Some external circuitry detects when the ac power to the system fails and sends an interrupt signal to the NMI

input. Because of the large filter capacitors in most power supplies, the dc system power will remain for perhaps 50 ms after the ac power is gone. This is more than enough time for a type 2 interrupt-service procedure to copy program data to some RAM which has a battery backup power supply. When the ac power returns, program data can be restored from the battery-backed RAM, and the program can resume execution where it left off. A practice problem at the end of the chapter gives you a chance to write a simple procedure for this task.

•

BREAKPOINT INTERRUPT—TYPE 3

The *type 3* interrupt is produced by execution of the INT 3 instruction. The main use of the type 3 interrupt is to implement a breakpoint function in a system. In Chapter 4 we described the use of breakpoints in debugging assembly language programs. We hope that you have been using them in debugging your programs. When you insert a breakpoint, the system executes the instructions up to the breakpoint and then goes to the breakpoint procedure. Unlike the single-step feature, which stops execution after each instruction, the breakpoint feature executes all the instructions up to the inserted breakpoint and then stops execution.

When you tell most 8086 systems to insert a breakpoint at some point in your program, they actually do it by temporarily replacing the instruction byte at that address with CCH, the 8086 code for the INT 3 instruction. When the 8086 executes this INT 3 instruction, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next mainline instruction on the stack. The 8086 then gets the CS value of the start of the type 3 interrupt-service procedure from address 0000EH and the IP value for the procedure from address 0000CH. A breakpoint interrupt-service procedure usually saves all the register contents on the stack. Depending on the system, it may then send the register contents to the CRT display and wait for the next command from the user, or in a simple system it may just return control to the user. In this case an Examine Register command can be used to check if the register contents are correct at that point in the program.

OVERFLOW INTERRUPT—TYPE 4

The 8086 overflow flag (OF) will be set if the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register or memory location. For example, if you add the 8-bit signed number 01101100 (108 decimal) and the 8-bit signed number 01010001 (81 decimal), the result will be 10111101 (189 decimal). This would be the correct result if we were adding unsigned binary numbers, but it is not the correct signed result. For signed operations, the 1 in the most significant bit of the result indicates that the result is negative and in 2's complement form. The result, 10111101, then actually represents −67 decimal, which is obviously not the correct result for adding +108 and +89.

There are two major ways to detect and respond to an overflow error in a program. One way is to put the Jump if Overflow instruction, JO, immediately after the arithmetic instruction. If the overflow flag is set as a result of the arithmetic operation, execution will jump to the address specified in the JO instruction. At this address you can put an error routine which responds to the overflow in the way you want.

The second way of detecting and responding to an overflow error is to put the *Interrupt on Overflow* instruction, INTO, immediately after the arithmetic instruction in the program. If the overflow flag is not set when the 8086 executes the INTO instruction, the instruction will simply function as an NOP. However, if the overflow flag is set, indicating an overflow error, the 8086 will do a *type 4* interrupt after it executes the INTO instruction.

When the 8086 does a type 4 interrupt, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the interrupt-service procedure from address 00012H and the IP value for the procedure from address 00010H. Instructions in the interrupt-service procedure then perform the desired response to the error condition. The procedure might, for example, set a "flag" in a memory location as we did in the BAD_DIV procedure in Figure 8-4*b*. 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.

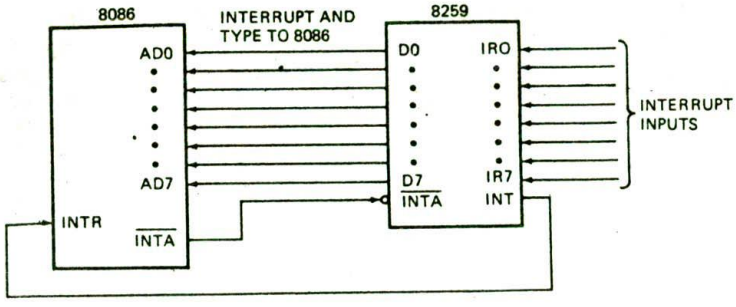


FIGURE 8-5 Block diagram showing an 8259 connected to an 8086.

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 be interrupted. If INTR were not disabled during the first response, the 8086 would be continuously interrupted and would never get to the actual interrupt-service procedure.

The IRET instruction at the end of an interrupt-service procedure restores the flags to the condition they were in before the procedure by popping the flag register off the stack. This will reenable the INTR input. If a high-level signal is still present on the INTR input, it will cause the 8086 to be interrupted again. If you do not want the 8086 to be interrupted again by the same input signal, you have to use external hardware to make sure that the signal is made low again before you reenable INTR with the STI instruction or before the IRET from the INTR service procedure.

When the 8086 responds to an INTR interrupt signal, its response is somewhat different from its response to other interrupts. The main difference is that for an INTR interrupt, the interrupt type is sent to the 8086 from an external hardware device such as the 8259A *priority interrupt controller*, as shown in Figure 8-5. We discuss the 8259A in detail later in the chapter, but here's an introduction.

When an 8259A receives an interrupt signal on one of its IR inputs, it sends an interrupt request signal to the INTR input of the 8086. If the INTR input of the 8086 has been enabled with an STI instruction, the 8086 will respond as shown by the waveforms in Figure 8-6.

The 8086 first does two interrupt-acknowledge ma-

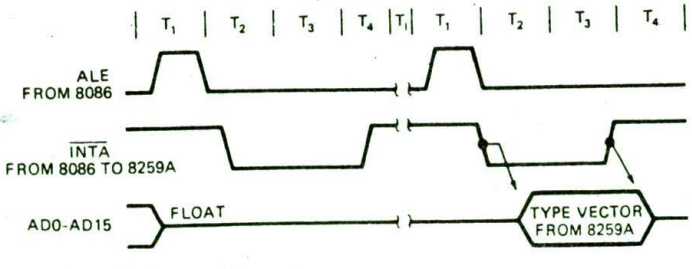


FIGURE 8-6 8086 interrupt-acknowledge machine cycles.

chine cycles, as shown in Figure 8-6. The purpose of these two machine cycles is to get the interrupt type from the external device. At the start of the first interrupt-acknowledge machine cycle, the 8086 floats the data bus lines, AD0–AD15, and sends out an interrupt-acknowledge pulse on its $\overline{\text{INTA}}$ output pin. This pulse essentially tells the 8259A to "get ready." During the second interrupt-acknowledge machine cycle, the 8086 sends out another pulse on its $\overline{\text{INTA}}$ output pin. In response to this second $\overline{\text{INTA}}$ pulse, the 8259A puts the interrupt type (number) on the lower eight lines of the data bus, where it is read by the 8086.

Once the 8086 receives the interrupt type, it pushes the flag register on the stack, clears TF and IF, and pushes the CS and IP values of the next instruction on the stack. It then uses the type it read in from the external device to get the CS and IP values for the interrupt-service procedure from the interrupt-pointer table in memory. The IP value for the procedure will be put at an address equal to 4 times the type number, and the CS value will be put at an address equal to 4 times the type number plus 2, just as is done for the other interrupts.

The advantage of having an external device insert the desired interrupt type is that the external device can "funnel" interrupt signals from many sources into the INTR input pin on the 8086. When the 8086 responds with $\overline{\text{INTA}}$ pulses, the external device can send to the 8086 the interrupt type that corresponds to the source of the interrupt signal. As you will see later, the external device can also prevent an argument if two or more sources send interrupt signals at the same time.

PRIORITY OF 8086 INTERRUPTS

As you read through the preceding discussions of the different interrupt types, the question that may have occurred to you is, What happens if two or more interrupts occur at the same time? The answer to this question is that the highest-priority interrupt will be serviced first, and then the next-highest-priority interrupt will be serviced. Figure 8-7 shows the priorities of the 8086 interrupts as shown in the Intel data book. Some examples will show you what these priorities actually mean.

As a first example, suppose that the INTR input is enabled, the 8086 receives an INTR signal during execution of a Divide instruction, and the divide operation produces a divide-by-zero interrupt. Since the internal interrupts—such as divide error, INT, and INTO — have higher priority than INTR, the 8086 will do a divide error (type 0) interrupt response first. Part of the type 0 interrupt response is to clear IF. This disables the INTR input and prevents the INTR signal from interrupting the higher-priority type 0 interrupt-service procedure. An IRET instruction at the end of the type 0 procedure will restore the flags to what they were before the type 0 response. This will reenable the INTR input, and the 8086 will do an INTR interrupt response. A similar sequence of operations will occur if the 8086 is executing an INT or INTO instruction and an interrupt signal arrives at the INTR input.

INTERRUPT	PRIORITY
DIVIDE ERROR, INT n, INTO	HIGHEST
NMI	
INTR	
SINGLE-STEP	LOWEST

FIGURE 8-7 Priority of 8086 interrupts. (*Intel Corporation*)

As a second example of how this priority works, suppose that a rising-edge signal arrives at the NMI input while the 8086 is executing a DIV instruction, and that the division operation produces a divide error. Since the 8086 checks for internal interrupts before it checks for an NMI interrupt, the 8086 will push the flags on the stack, clear TF and IF, push the return address on the stack, and go to the start of the divide error (type 0) service procedure. However, because the NMI interrupt request is not disabled, the 8086 will then do an NMI (type 2) interrupt response. In other words, the 8086 will push the flags on the stack, clear TF and IF, push the return address on the stack, and go execute the NMI interrupt-service procedure. When the 8086 finishes the NMI procedure, it will return to the divide error procedure, finish executing that procedure, and then return to the mainline program.

To finish our discussion of 8086 interrupt priorities, let's see how the single-step (trap, or type 1) interrupt fits in. If the trap flag is set, the 8086 will do a type 1 interrupt response after every mainline instruction. When the 8086 responds to any interrupt, however, part of its response is to clear the trap flag. This disables the single-step function, so the 8086 will not normally single-step through the instructions of the interrupt-service procedure. The trap flag can be set again in the single-step procedure if single-stepping is desired in the interrupt-service procedure.

Now that we have shown you the different types of 8086 interrupts and how the 8086 responds to each, we will show you a few examples of how the 8086 hardware interrupts are used. Other applications of interrupts will be shown throughout the rest of the book.

HARDWARE INTERRUPT APPLICATIONS

Simple Interrupt Data Input

One of the most common uses of interrupts is to relieve a CPU of the burden of polling. To refresh your memory, polling works as follows.

The strobe or data ready signal from some external device is connected to an input port line on the microcomputer. The microcomputer uses a program loop to read and test this port line over and over until the data ready signal is found to be asserted. The microcomputer then exits the polling loop and reads in the data from the external device. Data can also be output on a polled basis.

The disadvantage of polled input or output is that while the microcomputer is polling the strobe or data

ready signal, it cannot easily be doing other tasks. In systems where the microcomputer must be doing many tasks, polling is a waste of time, so interrupt input and output is used. In this case the data ready or strobe signal is connected to an interrupt input on the microcomputer. The microcomputer then goes about doing its other tasks until it is interrupted by a data ready signal from the external device. An interrupt-service procedure can read in or send out the desired data in a few microseconds and return execution to the interrupted program. The input or output operation then uses only a small percentage of the microprocessor's time.

For our example here, we will connect the key-pressed strobe to the NMI interrupt input of the 8086 on an SDK-86. The NMI input is usually reserved for responding to a power failure or some other catastrophic condition. However, since we are not expecting any catastrophic conditions to befall our SDK-86, we choose to use this input because it does not require an external hardware device to insert the interrupt type as does the INTR input.

Sheet 2 of the SDK-86 schematics in Figure 7-8 shows the circuitry normally connected to the NMI input. This circuitry is designed so that you can produce an NMI interrupt by pressing a key labeled INTR on the hex keypad. When this key is pressed, the input of the 74LS14 inverter will be made low, and the output of the inverter will go high. The low-to-high transition on the NMI input causes the 8086 to automatically do an NMI (type 2) interrupt response.

Figure 8-8 shows how we modified the circuitry for our example here. We removed R22, a 110-Ω resistor, and C33, a 1-μF capacitor, so that the keypad switch can no longer cause an interrupt. We then connected an active low strobe line from an ASCII-encoded keyboard directly to the input of A21, the 74LS14 inverter. When a key on the ASCII keyboard is pressed, the keyboard circuitry will send out the ASCII code for the pressed key on its eight parallel data lines and it will assert the key-pressed strobe line low. The key-pressed strobe going low will cause the NMI input of the 8086 to be asserted high. This will cause the 8086 to do a type 2 interrupt.

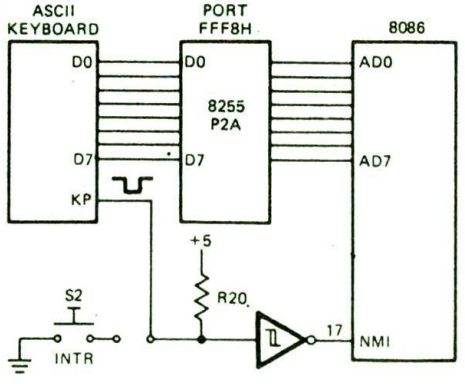


FIGURE 8-8 Circuit modifications for SDK-86 NMI input.

Now let's look at the software considerations for this interrupt example.

The software considerations are very similar to those for the divide-by-zero example in a previous section. As shown in Figure 8-9*a*, p. 218, the mainline program for this example consists mostly of a walk through an initialization list. First, assuming that you are going to read in the ASCII characters from the keyboard and put them in an array in memory, you need to set up a data segment for the array, set up the array, and declare any other variables you are going to use in the program. The statement ASCII_POINTER DW OFFSET ASCII_STRING in the data segment in Figure 8-9*a* sets aside a word location in memory and initializes that location with the offset of the start of the array we declared to put the ASCII characters in. In the procedure we get this pointer, use it to store a character, and increment it to point to the next location in the array. Since this pointer is stored in a named memory location, it can be accessed easily by the procedure, no matter when the interrupt occurs in the mainline program. KEYDONE is a flag which will be set by the interrupt procedure when 100 characters have been read in and stored.

Any interrupt response uses the stack, so next you need to set up a stack. Note that the PUBLIC and EXTRN directives are used so that the mainline program and the interrupt procedure can be in separate assembly modules.

In the code section of the mainline you need to initialize the stack segment register, the stack pointer register, and the data segment register. Finally, you need to initialize the interrupt-vector table by loading address 00008H with the IP value for the start of the type 2 procedure, and address 0000AH with the CS value for the start of the procedure.

The HERE:JMP HERE instruction at the end of the mainline program stimulates a complex mainline program that the 8086 might be executing. The 8086 will execute this instruction over and over until an interrupt occurs. When an interrupt occurs, the 8086 will service the interrupt and then return to execute the HERE:JMP HERE instruction over and over again until the next interrupt. Now let's consider the interrupt procedure.

The algorithm for the interrupt procedure can be simply stated as

```
IF 100 characters not read THEN
        Read character from port
        Mask parity bit
        Put character in array
        Increment array pointer
        Decrement character count
        Return
ELSE Return
```

Note that we used an IF-THEN-ELSE structure rather than a WHILE not 100 characters DO structure, because we want only one character to be read in for each call of the procedure.

Figure 8-9*b*, p. 219, shows the assembly language program for the interrupt-service procedure. After saving AX, BX, CX, and DX on the stack, we check to see if all

```
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                             DATA SEGMENT  WORD  PUBLIC
12 0000  64*(00)                         ASCII_STRING   DB   100 DUP(0)          ; Store for characters
13 0064  0000r                           ASCII_POINTER  DW   OFFSET ASCII_STRING ; Pointer to ASCII_STRING
14 0066  64                              CHARCNT        DB   100                ; Read 100 characters
15 0067  00                              KEYDONE        DB    0                 ; =1 if characters all read
16 0068                             DATA ENDS
17
18 0000                             STACK_SEG SEGMENT
19 0000  64*(0000)                       DW  100 DUP (0)                  ;*Set up stack of 100 words
20                                       TOP_STACK  LABEL  WORD           ; Pointer to top of stack
21 00C8                             STACK_SEG     ENDS
22
23                                  PUBLIC  ASCII_POINTER, CHARCNT, KEYDONE ; Make available to other modules
24                                  EXTRN   KEYBOARD:FAR                    ; Procedure in another assembly module
25
26 0000                             CODE SEGMENT  WORD  PUBLIC
27                                       ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
28 0000  B8 0000s                   START: MOV  AX, STACK_SEG          ; Initialize stack segment register
29 0003  8E D0                             MOV  SS, AX
30 0005  BC 00C8r                          MOV  SP, OFFSET TOP_STACK   ; Initialize stack pointer
31 0008  B8 0000s                          MOV  AX, DATA               ; Initialize data segment register
32 000B  8E D8                             MOV  DS, AX
33                                  ;Store the address for the KEYBOARD routine at address 0000:0008
34                                  ;Address 00008-0000B is where type 2 interrupt gets interrupt
35                                  ;service procedure address. CS at 0000A & 0000B, IP at 00008 & 00009
36 000D  B8 0000                           MOV  AX, 0000
37 0010  8E C0                             MOV  ES, AX
38 0012  26: C7 06 000A 0000s              MOV  WORD PTR ES:000AH, SEG KEYBOARD
39 0019  26: C7 06 0008 0000e              MOV  WORD PTR ES:0008H, OFFSET KEYBOARD
40                                  ;Simulate larger program
41 0020  EB FE                      HERE:  JMP  HERE
42 0022                             CODE   ENDS
43                                         END
```

(*a*)

FIGURE 8-9 Reading characters from an ASCII keyboard on interrupt basis. (*a*) Initialization and mainline. (*See also next page.*)

characters have been read. If CHARCNT is 0, then we just pop the registers and return to the mainline program. If CHARCNT is not 0, we copy the array pointer from its named memory location, ASCII_POINTER, to BX. We then read in the ASCII character from the port that the keyboard is connected to and mask the parity bit of the ASCII character. The MOV [BX],AL instruction next copies the ASCII character to the memory location pointed to by BX. To get the pointer ready for the read and store operation, we increment the stored pointer with the INC ASCII_POINTER instruction. Finally, we restore DX, CX, BX, and AX, and return to the mainline program.

Sitting in a HERE:JMP HERE loop waiting for an interrupt signal may not seem like much of an improvement over polling the key-pressed strobe. However, in a more realistic program, the 8086 would be doing many 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.

```
 1                                ;8086 PROCEDURE F8-09B.ASM called by program F8-09A.ASM
 2                                ;ABSTRACT  : PROCEDURE KEYBOARD
 3                                           ; This procedure reads in ASCII characters from an
 4                                           ; encoded keyboard on an interrupt basis and stores them
 5                                           ; in a buffer in memory.
 6                                ;DESTROYS  : Nothing
 7                                ;PORTS     : Uses input port FFF8H for the keyboard input.
 8
 9 0000                           DATA      SEGMENT     WORD     PUBLIC
10                                 EXTRN     ASCII_POINTER:WORD, CHARCNT:BYTE, KEYDONE:BYTE
11 0000                          DATA      ENDS
12
13                               PUBLIC      KEYBOARD
14
15 0000                          CODE       SEGMENT PUBLIC
16 0000                          KEYBOARD   PROC    FAR
17                                 ASSUME CS:CODE, DS:DATA
18 0000  FB                              STI                    ; Enable 8086 INTR so higher priority
19                                                              ; interrupts can be recognized
20 0001  50                              PUSH AX                ; Save registers used
21 0002  53                              PUSH BX
22 0003  51                              PUSH CX
23 0004  52                              PUSH DX
24 0005  80 3E 0000e 00                  CMP  CHARCNT, 00       ; See if all characters read in
25 000A  74 23                           JZ   EXIT              ; Leave procedure if all done
26 000C  8B 1E 0000e                     MOV  BX, ASCII_POINTER ; Get pointer to buffer
27 0010  BA FFF8                         MOV  DX, 0FFF8H        ; Point at keyboard port
28 0013  EC                              IN   AL, DX            ; Read in ASCII code
29 0014  24 7F                           AND  AL, 7FH           ; Mask parity bit
30 0016  88 07                           MOV  [BX], AL          ; Write character to buffer
31 0018  FF 06 0000e                     INC  ASCII_POINTER     ; Point to next buffer location
32 001C  FE 0E 0000e                     DEC  CHARCNT           ; Reduce character count
33 0020  75 08                           JNZ  NOTDONE           ; If 100 chars not read, clear carry
34 0022  C6 06 0000e 01                  MOV  KEYDONE, 01       ; else set flag to indicate done
35 0027  EB 06 90                        JMP  EXIT
36 002A  C6 06 0000e 00          NOTDONE: MOV  KEYDONE, 00       ; More characters to read so zero flag
37 002F  5A                      EXIT:    POP  DX               ; Restore registers
38 0030  59                              POP  CX
39 0031  5B                              POP  BX
40 0032  58                              POP  AX
41 0033  CF                              IRET                   ; Return to interrupted program
42 0034                          KEYBOARD ENDP
43 0034                          CODE     ENDS
44                                        END
```

(*b*)

FIGURE 8-9 (*continued*) (b) Interrupt-service procedure.

To detect a board coming out of the machine, we use an infrared LED, a phototransistor, and two conditioning gates, as shown in Figure 8-10, p. 220. 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 between the LED and the phototransistor, the light will not reach the phototransistor, and it will turn off. Its collector will go high, and so will the signal to the NMI input of the 8086. The 74LS14 Schmitt trigger inverters are necessary to turn the slow-risetime signal from the phototransistor collector into a signal which meets the risetime requirements of the NMI input on the 8086.

When the 8086 receives the low-to-high signal on its NMI input, it will automatically do a type 2 interrupt response. As we mentioned before, all the type 2 interrupt-service procedure has to do in this case is increment the board count in a named memory location and return to running the machine. This same technique can be used to count people going into a stadium, cows coming in from the pasture, or just about anything else you might want to count.

TIMING APPLICATIONS

In Chapter 4 we showed how a delay loop could be used to set the time between microcomputer operations. In the example there, we used a delay loop to take in data samples at 1-ms intervals. The obvious disadvantage of a delay loop is that while the microcomputer is stuck in the delay loop, it cannot easily be doing other useful work. In many cases a delay loop would be a waste of the microcomputer's valuable time, so we use an interrupt approach.

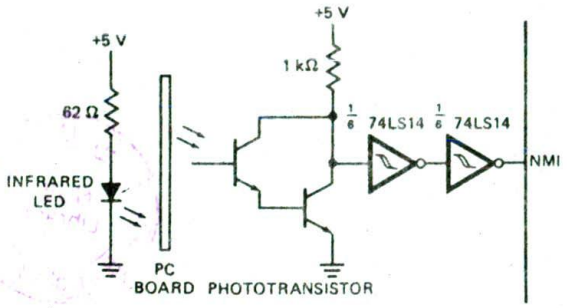


FIGURE 8-10 Circuit for optically detecting presence of an object.

Suppose, for example, that in our 8086-controlled printed-circuit-board-making machine we need to check the pH of a solution approximately every 4 min. If we used a delay loop to count off the 4 min, either the 8086 wouldn't be able to do much else or we would have some difficult calculations to figure out at what points in the program to go check the pH.

To solve this problem, all we have to do is connect a simple 1-Hz pulse source to an interrupt input, as shown in Figure 8-11. This 555 timer circuit is not very accurate, but it is inexpensive, and it is good enough for this application. The 555 timer will send an interrupt signal to the 8086 NMI input approximately once every second. An interrupt procedure is used to keep a count of how many NMI interrupts have occurred. This count is equal to the number of seconds that have passed.

To help you visualize how this works, Figure 8-12 shows the algorithm for this mainline and procedure. In the mainline we set up stack and data segments. In the data segment, we set aside a memory location for the seconds count and initialize that location to the number of seconds that we want to count off. In this case we want 4 min, which is 240 decimal or F0H seconds. Then we initialize the data segment register, stack segment register, and stack pointer register as before.

Each time the 8086 receives an interrupt from the 555 timer, it executes the interrupt-service procedure

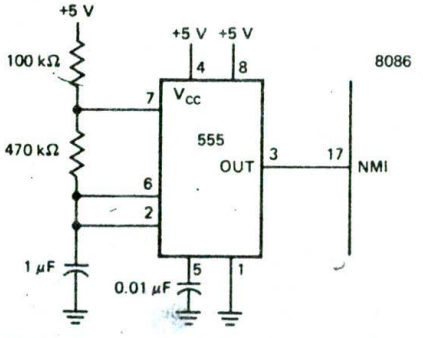


FIGURE 8-11 Inexpensive 1-Hz pulse source for interrupt timing.

```
Initialize
    Interrupt Pointer Table
    Stack and Stack segment Pointer
    Data Segment
    Seconds count to 240 decimal
Wait for Interrupt
```

(a)

```
Save Registers
Decrement Seconds Count
IF Seconds Count = 0 THEN
     Reload Seconds count with 240 decimal
     Call pH read procedure
     Restore registers
     Return to Mainline
ELSE Restore registers
     Return to Mainline
```

(b)

FIGURE 8-12 Algorithm for pH read at 4-min intervals. (*a*) Initialization and mainline. (*b*) Interrupt-service procedure.

for the NMI interrupt. In this procedure we decrement the seconds count in the named memory location and test to see if the count is down to zero yet. If the count is zero, we know that 4 min have elapsed, so we reload the seconds count memory location with 240 and call the procedure which reads the pH of the solution and takes appropriate action if the pH is not correct. If the seconds count is not zero, execution simply returns to the mainline program until the next interrupt from the 555 or from some other source occurs. The advantage of this interrupt approach is that the interrupt-service procedure takes only a few microseconds of the 8086's time once every second. The rest of the time the 8086 is free to run the mainline program.

USING AN INTERRUPT TO PRODUCE A REAL-TIME CLOCK

Another application using a 1-Hz interrupt input might be to generate a real-time clock of seconds, minutes, and hours. The time from this clock can then be displayed and/or printed out on timecards, etc. To generate the clock, a 1-Hz signal is applied to an interrupt input. A seconds count, a minutes count, and an hours count are kept in three successive memory locations. When an interrupt occurs, the seconds count is incremented by 1. If the seconds count is not equal to 60, then execution is simply returned to the mainline program. If the seconds count is equal to 60, then the seconds count is reset to 0 and the minutes count is incremented by 1. If the minutes count is not 60, then execution is simply returned to the mainline. If the minutes count is 60, then the minutes count is reset to 0 and the hours count is incremented by 1. If the hours count is not 13, then execution is simply returned to the mainline. If the hours count is equal to 13, then it is reset to 1 and execution is returned to the mainline. A problem at the end of the chapter asks you to write the algorithm and program for this real-time clock.

The interrupt-service routine for the real-time clock can easily be modified to also keep track of other time measurements such as the 4-min timer shown in the

preceding example. In other words, the single interrupt-service routine can be used to keep track of several different time intervals. By counting a different number of interrupts or applying a different frequency signal to the interrupt input, this technique can be used to time many different tasks in a microcomputer system.

GENERATING AN ACCURATE TIME BASE FOR TIMING INTERRUPTS

The 555 timer that we used for the 4-min timer just described was accurate enough for that application, but for many applications—such as a real-time clock—it is not. For more precise timing, we usually use a signal derived from a crystal-controlled oscillator. The processor clock signal is generated by a crystal-controlled oscillator, so it is stable, but this signal is obviously too high in frequency to drive a processor interrupt input directly. The solution is to divide the clock signal down with an external counter device to the desired frequency for the interrupt input. Most microcomputer manufacturers have a compatible device which can be programmed with instructions to divide an input frequency by any desired number. Besides acting as programmable frequency dividers, these devices have many important uses in microcomputer systems. Therefore, the next section describes how an Intel 8254 Programmable Counter operates, how an 8254 can easily be added to an SDK-86 board, and how an 8254 is used in a variety of interrupt applications. Also in the next section, we use the 8254 discussion to show you the general procedure for initializing any of the programmable peripheral devices we discuss in later chapters.

8254 SOFTWARE-PROGRAMMABLE TIMER/COUNTER

Because of the many tasks that they can be used for in microcomputer systems, programmable timer/counters are very important for you to learn about. As you read through the following sections, pay particular attention to the applications of this device in systems and the general procedure for initializing a programmable device such as the 8254. Read lightly through the discussions of the different counter modes to become aware of the types of problems that the device can solve for you. Later, when you have a specific problem to solve, you can dig into the details of these discussions.

Another important point to make to you here is that the discussions of various devices throughout the rest of this book are not intended to replace the manufacturers' data sheets for the devices. Many of the programmable peripheral devices we discuss are so versatile that each requires almost a small book to describe all the details of its operations. The discussions here are intended to introduce you to the devices, show you what they can be used for, and show you enough details about them that you can do some real jobs with them. After you become familiar with the use of a device in some simple applications, you can read the data sheets to learn further "bells and whistles" that the devices have.

Basic 8253 and 8254 Operation

The Intel 8253 and 8254 each contain three 16-bit counters which can be programmed to operate in several different modes. The 8253 and 8254 devices are pin-for-pin compatible, and they are nearly identical in function. The major differences are as follows:

1. The maximum input clock frequency for the 8253 is 2.6 MHz; the maximum clock frequency for the 8254 is 8 MHz (10 MHz for the 8254-2).
2. The 8254 has a *read-back* feature which allows you to latch the count in all the counters and the status of the counter at any point. The 8253 does not have this read-back feature.

To simplify reading of this section, we will refer only to the 8254. However, you can assume that the discussion also applies to the 8253 except where we specifically state otherwise.

As shown by the block diagram of the 8254 in Figure 8-13, the device contains three 16-bit counters. In some ways these counters are similar to the TTL presettable counters we reviewed in Chapter 1. The big advantage of these counters, however, is that you can load a count in them, start them, and stop them with instructions in your program. Such a device is said to be software-programmable. To program the device, you send count bytes and control bytes to the device just as you would send data to a port device.

If you look along the left side of the block diagram in Figure 8-13, you will see the signal lines used to interface the device to the system buses. A little later we show how these are actually connected in a real system. The main points for you to note about the 8254 at the moment are that it has an 8-bit interface to the data bus, it has a $\overline{CS}$ input which will be asserted by an

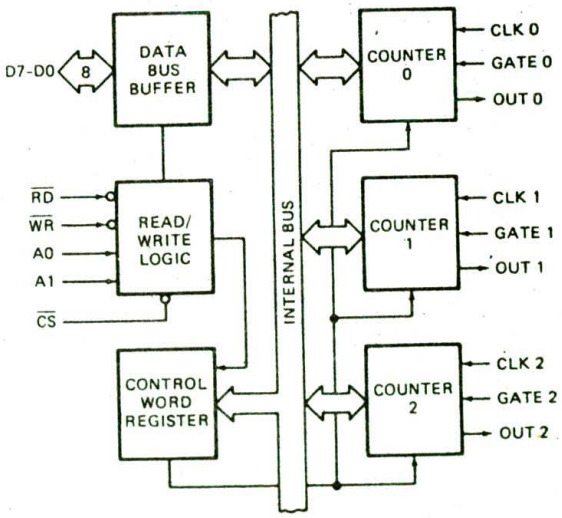


FIGURE 8-13 8254 internal block diagram. (*Intel Corporation*)

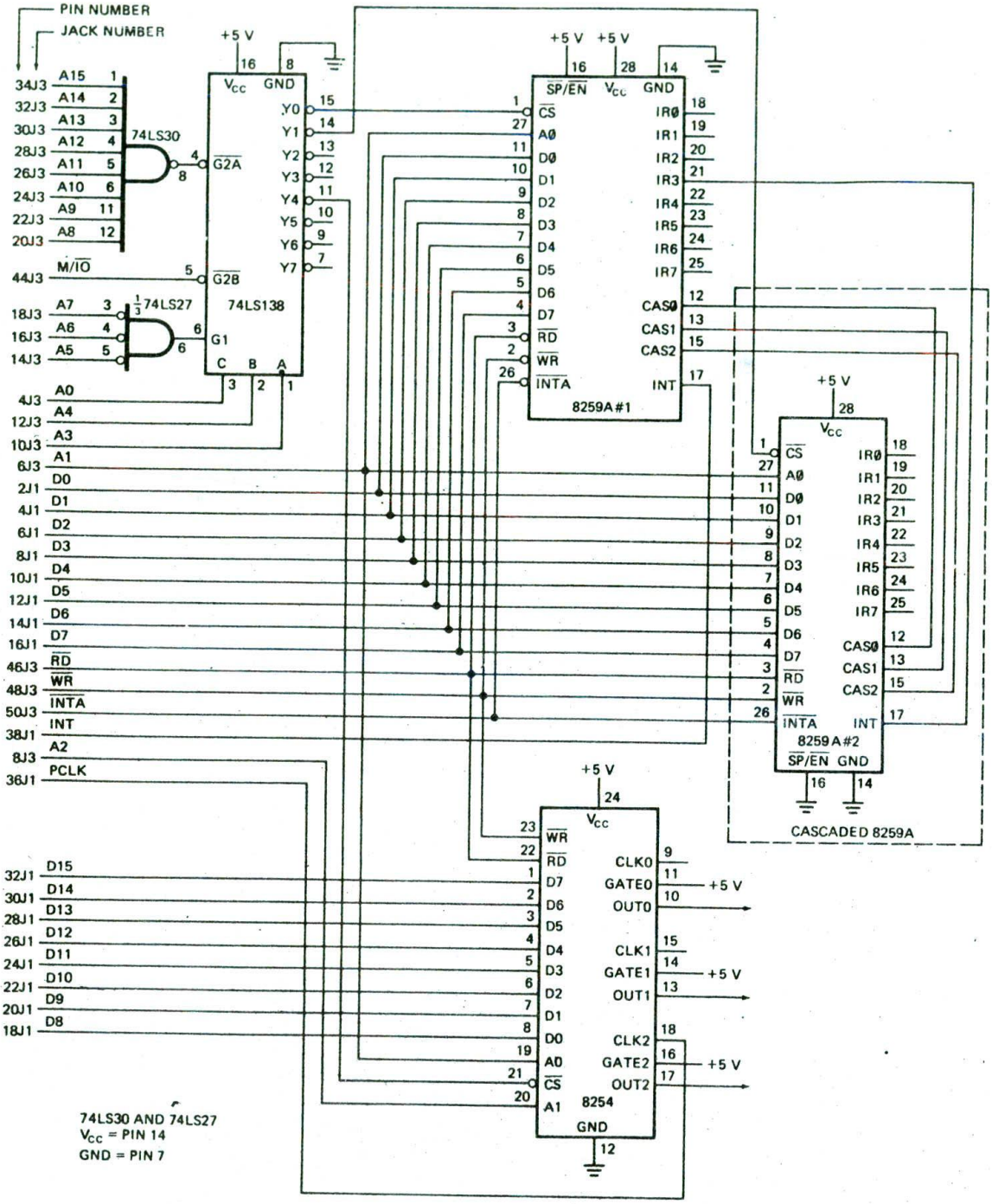


FIGURE 8-14 Circuit showing how to add an 8254 and 8259A(s) to an SDK-86 board.

address decoder when the device is addressed, and it has two address inputs, A0 and A1, to allow you to address one of the three counters or the control word register in the device.

The right side of the 8254 block diagram in Figure 8-13 shows the counter inputs and outputs. You can apply a signal of any frequency from dc to 8 MHz to each of the counter clock inputs, labeled CLK in the diagram.

The GATE input on each counter allows you to start or stop that counter with an external hardware signal. If the GATE input of a counter is high (1), then that counter is enabled for counting. If the GATE input is low, the counter is disabled. The output signal from each counter appears on its OUT pin. Now let's see how a programmable peripheral device such as the 8254 is connected in a system.

System Connections for an 8254 Timer/Counter

An 8254 is a very useful device to have in a microcomputer system, but, in order to keep the cost down, the SDK-86 was not designed with one on the board. Figure 8-14 shows the circuit connections for adding an 8254 Counter and an 8259A Priority Interrupt Controller to an SDK-86 board. We discuss the 8259A in a later section of this chapter.

If you use wire-wrap headers for connectors J1 and J3 on an SDK-86 board, the circuitry shown can easily be wire-wrapped on the prototyping area of the board. Install the WAIT-state jumper to insert one WAIT state. As explained in Chapter 7, a WAIT state is needed because of the added delay of the decoders and buffers.

The 74LS138 in Figure 8-14 is used to produce chip select ($\overline{CS}$) signals for the 8254, the 8259A, and any other I/O devices you might want to add. Let's look first at the circuitry around this device to determine the system base address which selects each device.

In order for any of the outputs of the 74LS138 to be asserted, the G1, $\overline{G2A}$, and $\overline{G2B}$ enable inputs must all be asserted. The G1 input will be asserted (high) if system address lines A5, A6, and A7 are all low. The $\overline{G2A}$ input will be asserted (low) if system address lines A8 through A15 are all high. As shown by the truth table in Figure 8-15, these two inputs therefore will be asserted for a system base address of FF00H. The $\overline{G2B}$ input of the 74LS138 will be asserted (low) if the M/$\overline{IO}$ line is low, as it will be for a port read or write operation.

Now, remember from Chapter 7 that only one of the Y outputs of the 74LS138 will ever be asserted at a time. The output asserted is determined by the 3-bit binary code applied to the A, B, and C select inputs. In the circuit in Figure 8-14, we connected system address line A0 to the C input, address line A4 to the B input, and address line A3 to the A input. The truth table in Figure 8-15 shows the system base addresses that will enable each of the 74LS138 Y outputs. As you will see a little later, system address lines A1 and A2 are used to select internal parts of the 8254 and 8259A.

We connected A0 to the C input so that half of the Y outputs will be selected by even addresses and half of the Y outputs will be selected by odd addresses. We did this so that loading on the two halves of the data bus will be equal as we add peripheral devices such as the 8254 and 8259A. To see how this works, note that the peripheral devices have only eight data lines. For an odd-addressed device we connect these data lines to the upper eight system data lines, and for an even-addressed device, we connect these to the lower eight system data lines. By alternating between odd- and even-selected outputs as we add peripheral devices, we equalize loading on the bus.

As shown by the truth table in Figure 8-15, the system base address of the added 8254 is FF01H. Other connections to the 8254 are the system $\overline{RD}$ and $\overline{WR}$ lines used to enable the 8254 for reading or writing; eight data lines, used to send control bytes, status bytes, and count values between the CPU and the 8254; and system address lines A1 and A2, used to select the control register or one of the three counters in the 8254. Now that you see how an 8254 is connected in a system, we will show you how to initialize an 8254 to do some useful work for you.

Initializing an 8254 Programmable Peripheral Device

When the power is first turned on, programmable peripheral devices such as the 8254 are usually in *undefined 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.

1. 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

A8-A15	A5-A7	A4	A3	A2	A1	A0	M/$\overline{IO}$	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		

FIGURE 8-15 Truth table for 74LS138 address decoder in Figure 8-14.

A1	A0	SELECTS
0	0	COUNTER 0
0	1	COUNTER 1
1	0	COUNTER 2
1	1	CONTROL WORD REGISTER

(*a*)

SYSTEM ADDRESS				8254 PART
F	F	0	1	COUNTER 0
F	F	0	3	COUNTER 1
F	F	0	5	COUNTER 2
F	F	0	7	CONTROL REG

(*b*)

FIGURE 8-16 8254 addresses. (*a*) Internal. (*b*) System.

in Figure 8-15, the system base address of the 8254 in our example here is FF01H.

2. 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 8-16*a* 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 Figure 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.

3. 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 8-16*b* 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.

4. 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.

5. 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 Figure 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 each resulting half in two, and finally divide each resulting quarter in two.) To help keep track of the meaning of each bit of a control word, write under each bit the meaning of that bit. A little later we show you how to do this for an 8254 control word. Documentation of this sort is very valuable when you are trying to debug a program or modify an old program for some new application.

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC – SELECT COUNTER:

SC1	SC0	
0	0	SELECT COUNTER 0
0	1	SELECT COUNTER 1
1	0	SELECT COUNTER 2
1	1	READ-BACK COMMAND (SEE READ OPERATIONS)

RW – READ/WRITE:

RW1	RW0	
0	0	COUNTER LATCH COMMAND (SEE READ OPERATIONS)
0	1	READ/WRITE LEAST SIGNIFICANT BYTE ONLY.
1	0	READ/WRITE MOST SIGNIFICANT BYTE ONLY.
1	1	READ/WRITE LEAST SIGNIFICANT BYTE FIRST, THEN MOST SIGNIFICANT BYTE.

M – MODE:

M2	M1	M0	
0	0	0	MODE 0 – INTERRUPT ON TERMINAL COUNT
0	0	1	MODE 1 – HARDWARE ONE-SHOT
X	1	0	MODE 2 – PULSE GENERATOR
X	1	1	MODE 3 – SQUARE WAVE GENERATOR
1	0	0	MODE 4 – SOFTWARE TRIGGERED STROBE
1	0	1	MODE 5 – HARDWARE TRIGGERED STROBE

BCD:

0	BINARY COUNTER 16-BITS
1	BINARY CODED DECIMAL (BCD) COUNTER (4 DECADES)

NOTE: DON'T CARE BITS (X) SHOULD BE 0 TO INSURE COMPATIBILITY WITH FUTURE INTEL PRODUCTS.

FIGURE 8-17 8254 control word format. (*Intel Corporation*)

6. Finally, send the control word(s) you have made up to the control register address for the device. In the case of the 8254, you also have to send the starting count to each of the counter registers.

Now that you have an overview of the initialization process, let's take a closer look at how you do the last two steps for an 8254.

A separate control word must be sent for each counter that you want to use in the device. However, according to Figure 8-16*a*, the 8254 has only one control register address. The trick here is that the control words for all three counters are sent to the same address in the

device. As shown in Figure 8-17, you use the upper 2 bits of a control word to tell the 8254 which counter you want that control word to initialize. For example, if you are making up a control word for counter 0 in the 8254, you make the SC1 bit of the control word a 0 and the SC0 bit a 0. Later we will explain the meaning of the read-back command, specified by a 1 in each of these bits.

The 16-bit counters in the 8254 are down counters. This means that the number in a counter will be decremented by each clock pulse. You can program the 8254 to count down a loaded number in BCD (decimal) or in binary. If you make the D0 bit of the control word a 0, then the counter will treat the loaded number as a pure binary number. In this case the largest number that you can load in is FFFFH. If you make the D0 bit of the control word a 1, then the largest number you can load in the counter is 9999H, and the counter will count a loaded number down in decimal (BCD). Actually, because of the way the 8254 counts, the "largest" number you can load in for both cases is 0000, but thinking of FFFFH and 9999H makes it easier to remember the difference between the two modes.

Now let's take a brief look at the mode bits (M2, M1, and M0) in the control word format in Figure 8-17. The binary number you put in these bits specifies the effect that the gate input will have on counting and the waveform that will be produced on the OUT pin. For example, if you specify mode 3 for a counter by putting 011 in these 3 bits, the counter will be put in a square-wave mode. In this mode, the output will be high for the first half of the loaded count and low for the second half of the loaded count. When the count reaches 0, the original count is automatically reloaded and the countdown repeated. The waveform on the OUT pin in this mode will then be a square wave with a frequency equal to the input clock frequency divided by the count you wrote to the counter. A little later we will discuss and show applications for some of the six different modes. First, let's finish looking at the control word bits and see how you send the control word and a count to the device.

The RW1 and RW0 bits of the control word are used to specify how you want to write a count to a counter or to read the count from a counter. If you want to load a 16-bit number into a counter, you put 1's in both these bits in the control word you send for that counter. After you send the control word, you send the low byte of the count to the counter address and then send the high byte of the count to the counter address. In a later paragraph we show an example of the instruction sequence to do this. In cases where you only want to load a new value in the low byte of a counter, you can send a control word with 01 in the RW bits and then send the new low byte to the counter. Likewise, if you want to load only a new high byte value in the counter, you can send a control word with 10 in the RW bits, and then send only the new high byte to the counter.

You can read the number in one of the counters at any time. The usual way to do this is to first latch the current count in some internal latches by sending a control word with 00 in the RW bits. Send another control word with 01, 10, or 11 in the RW bits to specify how you want to read out the bytes of the latched count. Then read the count from the counter address.

As a specific example of initializing an 8254, suppose that we want to use counter 0 of the 8254 in Figure 8-14 to produce a stable 78.6-kHz square-wave signal for a UART clock by dividing down the 2.45-MHz PCLK signal available on the SDK-86 board. To do this, we first connect the SDK-86 PCLK signal to the CLK input of counter 0 and tie the GATE input of the counter high to enable it for counting. To produce 78.6 kHz from 2.45 MHz, we have to divide by 32 decimal, so this is the value that we will eventually load into counter 0. First, however, we have to determine the system addresses for the device, make up the control word for counter 0, and send the control word.

As shown in Figure 8-16*b*, the system address for the control register of this 8254 is FF07H. This is where we will send the control word. For our control word we want to select counter 0, so we make the SC1 and SC0 bits both 0's. We want the counter to operate in square-wave mode. This is mode 3, so we make the mode bits of the control word 011. Since we want to divide by 32 decimal, we tell the counter to count down in decimal by making the BCD bit of the control word a 1. This makes our life easier, because we don't have to convert the 32 to binary or hex. Finally, we have to decide how we want to load the count into the counter. Since the count that we need to load in is less than 99, we only have to load the lower byte of the counter. According to Figure 8-17, the RW1 bit should be a 0 and the RW0 bit a 1 for a write to only the lower byte (LSB). The complete control word then is 00010111 in binary. Here are the instructions to send the control word and count to counter 0 of the 8254 in Figure 8-14. Note how the bits of the control word are documented.

```
MOV AL,00010111B  ; Control word for counter 0
                  ; Read/write LSB only, mode 3, BCD countdown
                  ; 00  01  011  1
                  ;             |_____ BCD countdown
                  ;          |_____ Mode 3
                  ;       |_____ R/W LSB only
                  ;  |_____ Select counter 0
MOV DX,0FF07H     ; Point at 8254 control register
OUT DX,AL         ; Send control word
MOV AL,32H        ; Load lower byte of count
MOV DX,0FF01H     ; Point to counter 0 count register
OUT DX,AL         ; Send count to count register
```

Note that since we set the RW bits of the control word for read/write LSB only, we do not have to include instructions to load the MSB of the counter. Programmed in this way, the 8254 will automatically load 0's in the upper byte of the counter.

If you need to load a count that is larger than 1 byte, make the RW bits in the control word both 1's. Send the lower byte of the count as shown above. Then send the high byte of the count to the count register by adding the instructions

```
MOV AL,HIGH_BYTE_OF_COUNT ; Load MSB of count
OUT DX,AL                 ; Send MSB to
                            count register
```

Note that the high byte of the count is sent to the same address that the low byte of the count was sent.

For each counter that you want to use in an 8254, you repeat the preceding series of six or eight instructions with the control word and count for the mode that you want. Before going on with this chapter, review the six initialization steps shown at the start of this section to make sure these are firmly fixed in your mind. In the next section we discuss and show some applications of the different modes in which an 8254 counter can be operated, but we do not have space there to show all the steps for each of the modes.

8254 Counter Modes and Applications

As we mentioned previously, an 8254 counter can be programmed to operate in any one of six different modes. The Intel data book uses timing diagrams such as those in Figure 8-18 to show how a counter functions in each of these modes. Since these waveforms may not be totally obvious to you at first glance, we will work our way through some of them to show you how to interpret them. We will also show some uses of the different counter modes. As you read through this section, don't try to absorb all the details of the different modes. Concentrate on learning to interpret the timing waveforms and on the different types of output signals you can produce with an 8254.

MODE 0—INTERRUPT ON TERMINAL COUNT

First read the Intel notes at the bottom of Figure 8-18; then take a look at the top set of waveforms in the figure. For this first example, the GATE input is held high so that the counter is always enabled for counting. The first dip in the waveform labeled $\overline{WR}$ represents the control word for the counter being written to the 8254. CW = 10 over this dip indicates that the control word written is 10H. According to the control word format in Figure 8-17, this means that counter 0 is being initialized for binary counting, mode 0, and a read/write of only the LSB. After the control word is written to the control register, the output pin of counter 0 will go low. The next dip in the $\overline{WR}$ waveform represents a count of 4 being written to the count register of counter 0. Before this count can be counted down, it must be transferred from the count register to the actual counter. If you look at the count values shown under the OUT waveform in

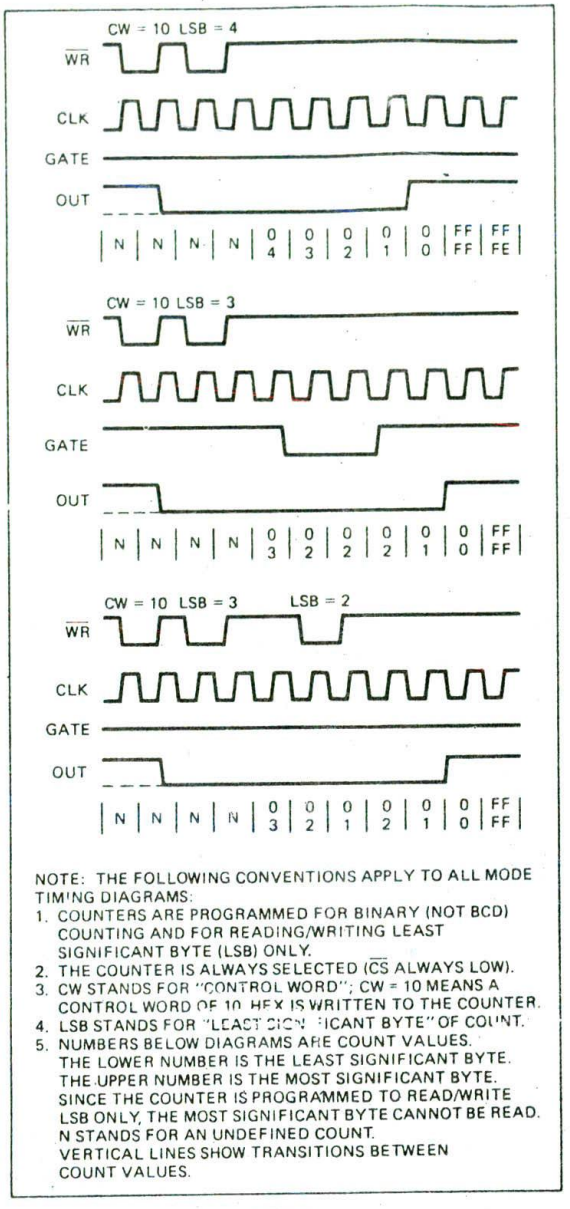


FIGURE 8-18 8254 MODE 0 example timing diagrams. (*Intel Corporation*)

the timing diagram, you should see that the count of 4 is transferred into the counter by the next clock pulse after $\overline{WR}$ goes high. Each clock pulse after this will decrement the count by 1. When the count is decremented to 0, the OUT pin will go high. If you write a count N to a counter in mode 0, the OUT pin will go high after N + 1 clock pulses have occurred. Note that the counter decrements from 0000 to FFFFH on the next clock pulse unless you load some new count into the

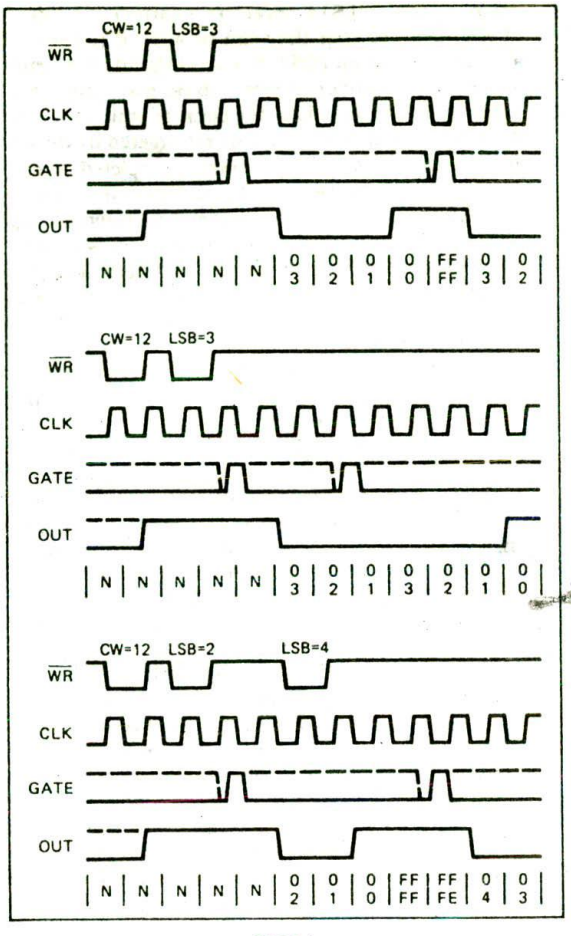


FIGURE 8-19 8254 MODE 1 example timing diagrams. (*Intel Corporation*)

counter. If the OUT pin is connected to an 8259A IR input or the NMI interrupt input of an 8086, then the processor will be interrupted when the counter reaches 0 (terminal count).

The second set of waveforms in Figure 8-18 shows that if the GATE input is made low, the counter value will be held. When the GATE input is made high again, the counter continues to decrement by 1 for each clock pulse.

The third set of waveforms in Figure 8-18 shows that if a new count is written to a counter, the new count will be loaded into the counter on the next clock pulse. Following clock pulses will decrement the new count until it reaches 0.

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 Figure 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 Figure 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 $\overline{WR}$ waveform represents the control word of 12H being sent to the 8254. Use Figure 8-17 to help you determine how this control word initializes the device. You should find

that a control word of 12H programs counter 0 for binary count, mode 1, read/write LSB only. When the control word is written to the 8254, the OUT pin goes high.

The second dip in the $\overline{WR}$ waveform represents writing a count to the counter. Note that, because the GATE input is low, the counter does not start counting down immediately when the count is written, as it does in mode 0. For mode 1, the GATE input functions as a trigger input. When the GATE/trigger input is made high, the count will be transferred from the count register to the actual counter on the next clock pulse. Each following clock pulse will decrement the counter by 1. When the counter reaches 0, the OUT pin will go high again. In other words, if you load a value of N in the counter and trigger the device by making the GATE input high, the OUT pin will go low for a time equal to N clock cycles. The output pulse width is then N times the period of the signal applied to the CLK input. Incidentally, the dashed sections of the GATE waveforms in Figure 8-19 mean that the GATE/trigger input signal can go low again any time during that time interval.

The second set of waveforms in Figure 8-19 demonstrates what is meant by the term *retriggerable*. If another trigger pulse comes before the previously loaded count has been counted down to 0, the original count will be reloaded on the next clock pulse. The countdown will then start over and continue until another trigger occurs or until the count reaches 0. If trigger pulses continue to come before the count is decremented to 0, the OUT pin will remain low.

The bottom set of waveforms in Figure 8-19 shows that if you write a new count to a count register while the OUT pin is low, the new count will not be loaded into the counter and counted down until the next trigger pulse occurs.

For an example of the use of mode 1, we will show you how to make a circuit which produces an interrupt signal if the ac power fails. This circuit could be connected to the NMI input of an 8086 to call an interrupt procedure which saves parameters in battery-backed RAM when the ac power fails.

Figure 8-20 shows a circuit which uses an optical coupler (an LED and a phototransistor packaged together) to produce logic-level pulses at power line frequency. The 74LS14 inverters sharpen the edges of these pulses so that they can be applied to the GATE/trigger input of an 8254. For a 60-Hz line frequency, a pulse will be produced every 16.66 ms. Now what we want to do here is to load the counter with a value such that the counter will always be retriggered by the power line pulses before the countdown is completed. As shown by the second set of waveforms in Figure 8-19, the OUT pin will then stay low and not send an interrupt signal to the NMI input of the 8086. If the ac power fails, no more pulses come in to the 8254 trigger input. The trigger input will be left high, and the countdown will be completed. The 8254 OUT pin will then go high and interrupt the 8086.

To determine the counter value for this application, you just calculate the number of input clock pulses required to produce a countdown time longer than 16.66 ms—for example, 20 ms. If you use the 2.4576-MHz PCLK signal on an SDK-86 board, 20 ms requires 49,152 cycles of PCLK, so this is the number you would load in the 8254 counter. Since this number is too large to load in as a BCD count, you put a 0 in the BCD bit of the control word to tell the 8254 to count the number down in binary. Then you send the count value of C000H to the count register.

MODE 2—TIMED INTERRUPT GENERATOR

In a previous section we described how a real-time clock of seconds, minutes, and hours could be kept in three memory locations by counting interrupts from a 1-Hz pulse source. We also described how the 1-Hz interrupts could be used to measure off other time intervals. The difficulty with using a 1-Hz interrupt signal is that the maximum resolution of any time measurement is 1 s. In other words, if you use a 1-Hz signal, you can only measure times to the nearest second. To improve the resolution of time measurements, most microcomputer systems use a higher-frequency signal such as 1 kHz for a real-time clock interrupt. With a 1-kHz interrupt signal, the time resolution is 1 ms. An 8254 counter operating in mode 2 can be used to produce a stable 1-kHz signal by dividing down the processor clock signal.

Figure 8-21 shows the waveforms for an 8254 counter operating in mode 2. Let's look at the top set of waveforms

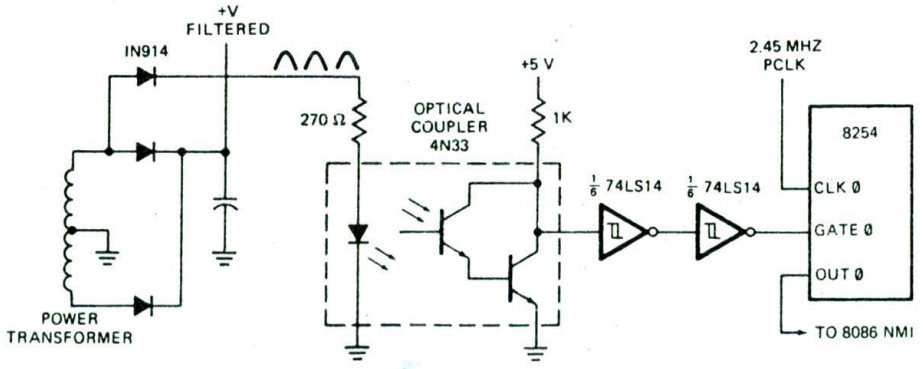


FIGURE 8-20 Circuit to produce logic-level pulses at power line frequency.

first. The two dips in the $\overline{WR}$ waveform represent a control word and the LSB of a count being written to the count register. The next clock pulse after the count is written will transfer the count from the count register to the actual counter. Since the GATE input is high, succeeding clock pulses will count down this value until it reaches 1. When the count reaches 1, the OUT pin, which was previously high, will go low for one clock pulse time. The falling edge of the next clock pulse will cause the OUT pin to go high again and the original count to be loaded into the counter again. Successive clock pulses will cause the countdown and load cycle to repeat over and over. If the counter is loaded with a number N, the OUT pin will go low for one clock cycle every N input clock pulses. The frequency of the output waveform then will be equal to the input clock frequency divided by N.

Now, for a specific example, suppose that you want to produce a 1-kHz signal for a real-time clock from an 8-MHz processor clock signal. To do this, you connect the processor clock signal to the CLK input on one of the 8254 counters and tie the GATE input of that counter high. You initialize that counter for BCD counting, mode 2, and read/write LSB then MSB. Since you want to divide the 8 MHz by 8000 decimal to get 1 kHz, you then write 00H to the counter as the LSB and 80H to the counter as the MSB.

A question that may occur to you at this point is, How do I count seconds if the interrupts are coming in every millisecond? The answer to the question is that you set aside a memory location as a milliseconds counter and initialize that location with 1000 decimal (3E8H). The interrupt-service procedure decrements this count each time an interrupt occurs and checks to see if the count is down to 0 yet. If the count is not 0, then execution is simply returned to the mainline. If the count is down to 0, 1000 interrupts or 1 s has passed. The milliseconds counter location is then reloaded with 3E8H, and the seconds-minutes-hours procedure is called to update the count of seconds, minutes, and hours. An exercise in the accompanying lab manual gives you a chance to develop a real-time clock in this way. Incidentally, the 1-kHz interrupt-service procedure can be used to measure off several different time intervals that are multiples of 1 ms.

The middle set of mode 2 waveforms in Figure 8-21 demonstrates that if the GATE input is made low while the counter is counting, counting will stop. If the GATE input is made high again, the original count will be reloaded into the counter by the next clock pulse. Succeeding clock pulses will decrement the loaded count.

The bottom set of mode 2 waveforms in Figure 8-21 shows that if a new count is written to the count register, this new count will not be transferred to the counter until the previously loaded count has been decremented to 1.

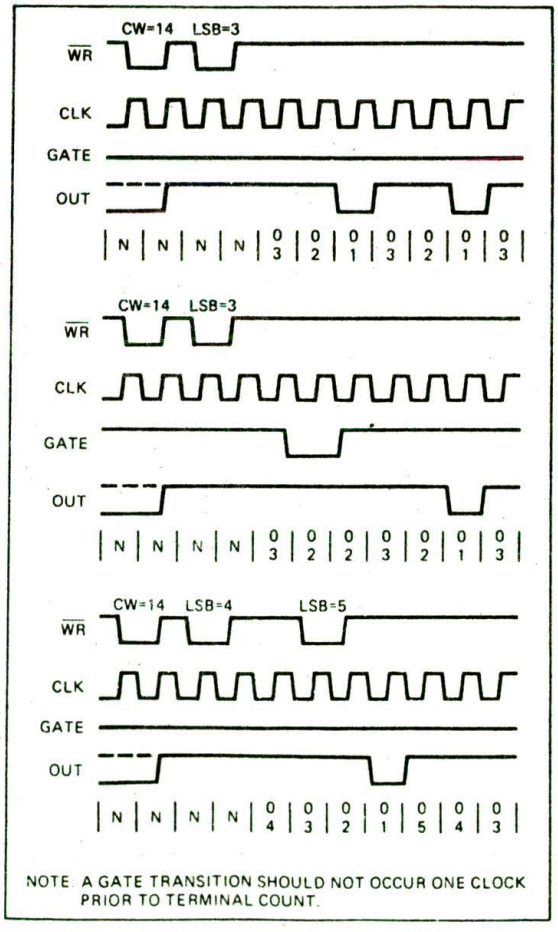
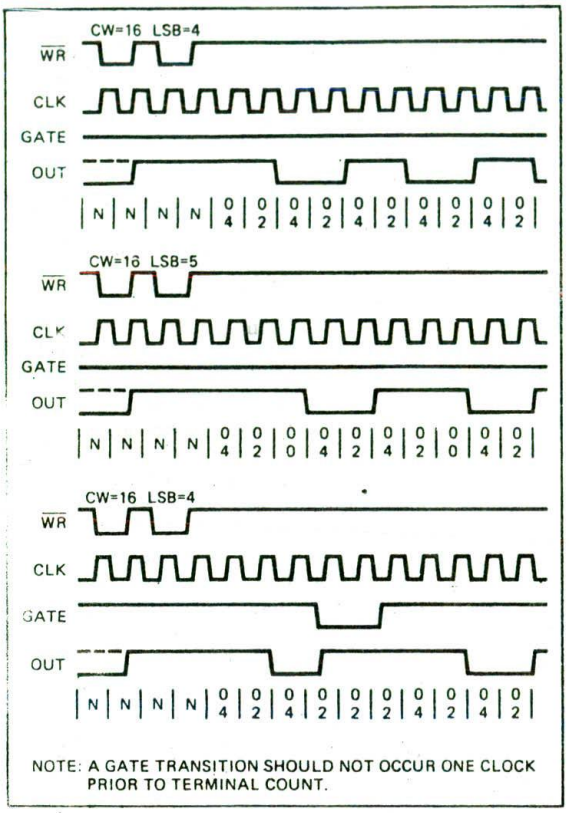


FIGURE 8-21 8254 MODE 2 example timing waveforms. (*Intel Corporation*)

MODE 3—SQUARE-WAVE MODE

If an 8254 counter is programmed for mode 3 and an even number is written to its count register, the waveform on the OUT pin will be a *square wave*. The frequency of the square wave will be equal to the frequency of the input clock divided by the number written to the count register. If an odd number is written to a counter programmed for mode 3, the output waveform will be high for one more clock cycle than it is low, so the waveform will not be quite symmetrical. Figure 8-22, p. 230, shows some example waveforms for mode 3. By now these waveforms should look quite familiar to you.

The top set of waveforms shows that after a control word is written to the control register and a count is written to the count register, the count is transferred to the counter on the next clock pulse. As shown by the count sequence under the OUT waveform, each additional clock pulse decrements the counter by 2. When the count is down to 2, the OUT pin goes low and the original count is reloaded. The OUT pin stays low while the loaded count is again counted down by 2's. When the count is down to 2, the OUT pin goes high again and the original count is again loaded into the counter. The cycle then repeats.



MODE 3

FIGURE 8-22 8254 MODE 3 example timing waveforms. (*Intel Corporation*)

The center set of waveforms in Figure 8-22 shows what happens if an odd number is written to the count register. As you can see from this waveform, the number of clock cycles for each waveform is still equal to the number loaded into the count register. However, as we mentioned before, the clock is high for one more clock cycle than it is low.

The bottom set of waveforms in Figure 8-22 shows that counting stops if the gate is made low at any time. After the GATE input is made high again, the countdown will continue.

Mode 3 can be used for any case where you want a repetitive square-wave-type signal. An 8254 counter operating in mode 3 can be used to generate the baud rate clock for a USART such as the 8251A. Also, mode 3 could be used to generate interrupt pulses for a real-time clock as we described for mode 2. The square-wave signal has the advantage that it is more easily observed with a scope than the narrow pulse produced by mode 2 operation.

Another use of 8254 counters operating in mode 3 is as programmable audio-tone generators. For this application, a high-frequency clock such as the 2.4576-MHz PCLK signal on an SDK-86 board is connected to the counter CLK input, the GATE input is tied high,

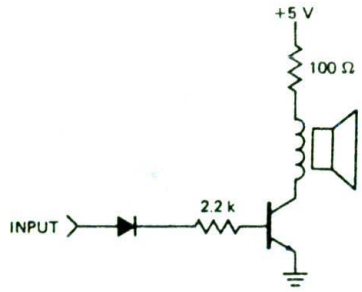


FIGURE 8-23 Audio speaker buffer for 8254 timer output or port.

and the OUT pin is connected to an audio buffer such as that shown in Figure 8-23.

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

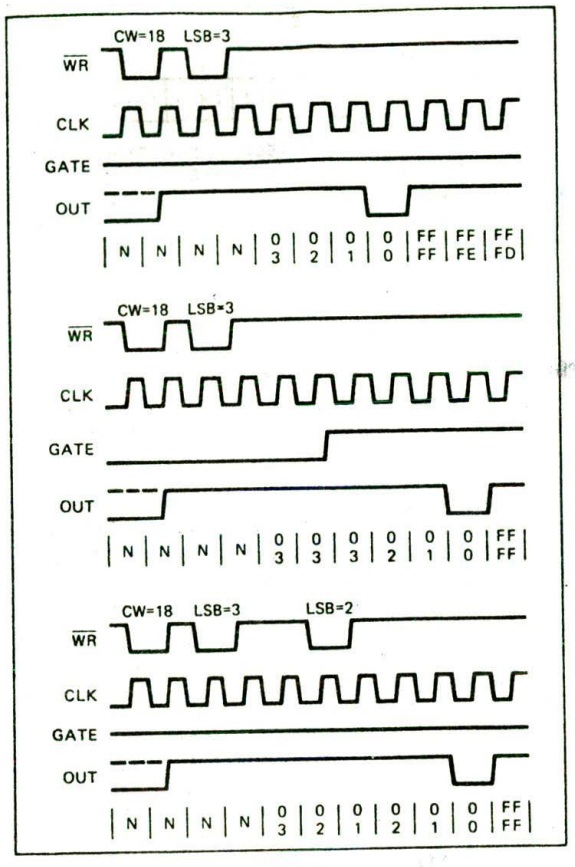
This mode and mode 5 are often confused with mode 1, but there is an obvious difference. Mode 1 is used to produce a low-going pulse that is N clock pulses wide. If you look at the top set of waveforms for mode 4 in Figure 8-24, you should see that mode 4 produces a low-going pulse *after* N + 1 clock 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

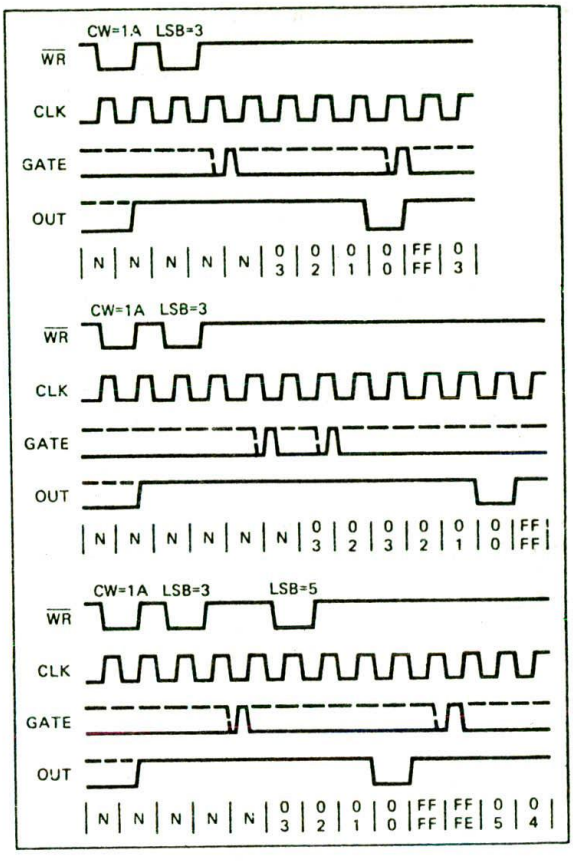
Mode 5 is used where we want to produce a low-going strobe pulse some programmable time interval after a rising-edge trigger signal is applied to the GATE input. This mode is very useful when you want to delay a rising-edge signal by some amount of time.

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



MODE 4

FIGURE 8-24 8254 MODE 4 example timing waveforms. (*Intel Corporation*)



MODE 5

FIGURE 8-25 8254 MODE 5 example timing waveforms. (*Intel Corporation*)

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 Figure 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 counted down. 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.

The bottom set of waveforms in Figure 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.

USING A NONSYSTEM CLOCK WITH AN 8254 IN MODES 2 AND 3

If you are applying a signal which is not derived from the system clock to the CLK input of an 8254 in mode 2 or mode 3, then a small note in the Intel data sheet indicates that the GATE input of the counter must be pulsed low just after the count is written to the counter. An easy way to do this is to connect the GATE input of the counter to an otherwise unused output port pin. You can then pulse the GATE by outputting a low and then outputting a high to that port pin.

READING THE COUNT FROM AN 8254 COUNTER

For many counter applications, we want to be able to read the current count in the counter. Suppose, for example, that we are using an 8254 counter to count the cars coming into a parking lot, as we did in our example for mode 0 in an earlier section. In that case

we used the counter to produce an interrupt when the parking lot was full, so we could shut the gate. Now further suppose that as part of a traffic flow study, we want to find out how many cars have come into the lot by 7:30 A.M. An interrupt-driven real-time clock procedure can, at 7:30 A.M., call a procedure which reads in the current count from the counter. Since the counter was initially loaded with 1000 decimal and is being counted down as cars come in, we can simply subtract the current count from 1000 to determine how many cars have come in.

The counters in an 8254 have latches on their outputs. When you read the count from a counter, what you are actually reading is the data on the outputs of these latches. These latches are normally enabled during counting so that the latch outputs just follow the counter outputs. If you try to read the count while the counter is counting, the count may change between reading the LSB and the MSB. This may give you a strange count. To read a correct count, then, you must in some way stop the counting or latch the current count on the output of the latches. There are three major ways of doing this.

The first is to stop counting by turning off the clock signal or making the GATE input low with external hardware. This method has the disadvantages that it requires external hardware and that a clock pulse which occurs while the clock is disabled will obviously not be counted.

The second way of reading a stable value from a counter is to latch the current count with a counter latch command and then read the latched count. A counter is latched by sending a control word to the control register address in the 8254. If you look at the format for the 8254 control word in Figure 8-17, you should see that a counter latch command is specified by making the RW1 and RW0 bits both 0. The SC1 and SC0 bits specify which counter we want to latch. The lower 4 bits of the control word are "don't cares" for a counter latch command word, so we usually make them 0's for simplicity. As an example, here is the sequence of instructions you would use to latch and read the LSB and MSB from counter 1 of the 8254 in Figure 8-14. We assume that the counter was already programmed for read/write LSB then MSB when the device was initialized. If the counter was programmed for only LSB or only MSB, then only that byte can be read.

```
MOV AL,01000000B    ; Counter 1 latch command
MOV DX,0FF07H       ; Point at 8254 control register
OUT DX,AL           ; Send latch command
MOV DX,0FF03H       ; Point at counter 1 address
IN AL,DX            ; Read LSB of latched count
MOV AH,AL           ; Save LSB of latched count
IN AL,DX            ; Read MSB of latched count
XCHG AH,AL          ; Count now in AX
```

When a counter latch command is sent, the latched count is held until it is read. When the count is read from the latches, the latch outputs return to following the counter outputs.

The third method of reading a stable count from a counter is to latch the count with a read-back command.

A0, A1 = 11 $\overline{CS}$ = 0 $\overline{RD}$ = 1 $\overline{WR}$ = 0

D7	D6	D5	D4	D3	D2	D1	D0
1	1	$\overline{COUNT}$	$\overline{STATUS}$	CNT 2	CNT 1	CNT 0	0

D5: 0 = LATCH COUNT OF SELECTED COUNTERS(S)
D4: 0 = LATCH STATUS OF SELECTED COUNTER(S)
D3: 1 = SELECT COUNTER 2
D2: 1 = SELECT COUNTER 1
D1: 1 = SELECT COUNTER 0
D0: RESERVED FOR FUTURE EXPANSION; MUST BE 0

FIGURE 8-26 8254 read-back control word format.

This method is available in the 8254 but not in the 8253. It is essentially an enhanced version of the counter latch command approach described in the preceding paragraphs.

Figure 8-26 shows the format for the 8254 counter read-back command word. It is sent to the same address that other control words are for a particular 8254. The 1's in bits D7 and D6 identify this as a read-back command word. To latch the count on a counter, you put a 0 in bit D5 of the control word and put a 1 in the bit position that corresponds to that counter in the control word. The advantage of this control word is that you can latch one, two, or all three counters by putting 1's in the appropriate bits. Once a counter is latched, the count is read as shown in the previous example program. After being read, the latch outputs return to following the counter outputs.

If a read-back command word with bit D4 = 0 is sent to an 8254, the status of one or more counters will be latched on the output latches. Consult the Intel data sheet for further information on this latched status.

The preceding sections have shown how 8254 counters can be used to do a wide variety of tasks around microcomputers. Many of these applications produce an interrupt signal which must be connected to an interrupt input on the microprocessor. In the next section we show how a *priority interrupt controller* device, the Intel 8259A, is used to service multiple interrupts.

8259A PRIORITY INTERRUPT CONTROLLER

Previous sections of this chapter show how interrupts can be used for a variety of applications. In a small system, for example, we might read ASCII characters in from a keyboard on an interrupt basis; count interrupts from a timer to produce a real-time clock of seconds, minutes, and hours; and detect several emergency or job-done conditions on an interrupt basis. Each of these interrupt applications requires a separate interrupt input. If we are working with an 8086, we have a problem here because the 8086 has only two interrupt inputs, NMI and INTR. If we save NMI for a power failure interrupt, this leaves only one interrupt input for all the other applications. For applications where we have interrupts from multiple sources, we use an external device called a *priority interrupt controller* (PIC) to "funnel" the interrupt signals into a single interrupt input

on the processor. In this section, we show how a common PIC, the Intel 8259A, is connected in an 8086 system, how it is initialized, and how it is used to handle interrupts from multiple sources.

8259A Overview and System Connections

To show you how an 8259A functions in an 8086 system, we first need to review how the 8086 INTR input works. Remember from Figure 8-5 and a discussion earlier in this chapter that if the 8086 interrupt flag is set and the INTR input receives a high signal, the 8086 will

1. Send out two interrupt acknowledge pulses on its $\overline{\text{INTA}}$ pin to the $\overline{\text{INTA}}$ pin of an 8259A PIC. The $\overline{\text{INTA}}$ pulses tell the 8259A to send the desired interrupt type to the 8086 on the data bus.
2. Multiply the interrupt type it receives from the 8259A by 4 to produce an address in the interrupt vector table.
3. Push the flags on the stack.
4. Clear IF and TF.
5. Push the return address on the stack.
6. Get the starting address for the interrupt procedure from the interrupt-vector table and load that address in CS and IP.
7. Execute the interrupt-service procedure.

Now let's take a little closer look at how the 8259A functions during this process. To start, study the internal block diagram of an 8259A in Figure 8-27. In the figure, first notice the 8-bit data bus and control signal pins in the upper left corner of the diagram. The data bus allows the 8086 to send control words to the 8259A and read a status word from the 8259A. The $\overline{\text{RD}}$ and $\overline{\text{WR}}$ inputs control these transfers when the device is selected by asserting its chip select ($\overline{\text{CS}}$) input low. The 8-bit data bus also allows the 8259A to send interrupt types to the 8086.

Next, in Figure 8-27, observe the eight interrupt inputs labeled IR0 through IR7 on the right side of the diagram. If the 8259A is properly enabled, an interrupt signal applied to any one of these inputs will cause the 8259A to assert its INT output pin high. If this pin is connected to the INTR pin of an 8086 and if the 8086 interrupt flag is set, then this high signal will cause the previously described INTR response.

The $\overline{\text{INTA}}$ input of the 8259A is connected to the $\overline{\text{INTA}}$ output of the 8086. The 8259A uses the first $\overline{\text{INTA}}$ pulse from the 8086 to do some activities that depend on the mode in which it is programmed. When it receives the second $\overline{\text{INTA}}$ pulse from the 8086, the 8259A outputs an interrupt type on the 8-bit data bus, as shown in Figure 8-6. The interrupt type that it sends to the 8086 is determined by the IR input that received an interrupt signal and by a number you send the 8259A when you initialize it. The point here is that the 8259A "funnels" interrupt signals from up to eight different sources into the 8086 INTR input, and it sends the 8086 a specified interrupt type for each of the eight interrupt inputs.

At this point the question that may occur to you is, What happens if interrupt signals appear at, for example, IR2 and IR4 at the same time? In the *fixed-priority mode* that the 8259A is usually operated in, the answer to this question is quite simple. In this mode, the IR0 input has the highest priority, the IR1 input the next highest, and so on down to IR7, which has the lowest priority. What this means is that if two interrupt signals occur at the same time, the 8259A will service the one with

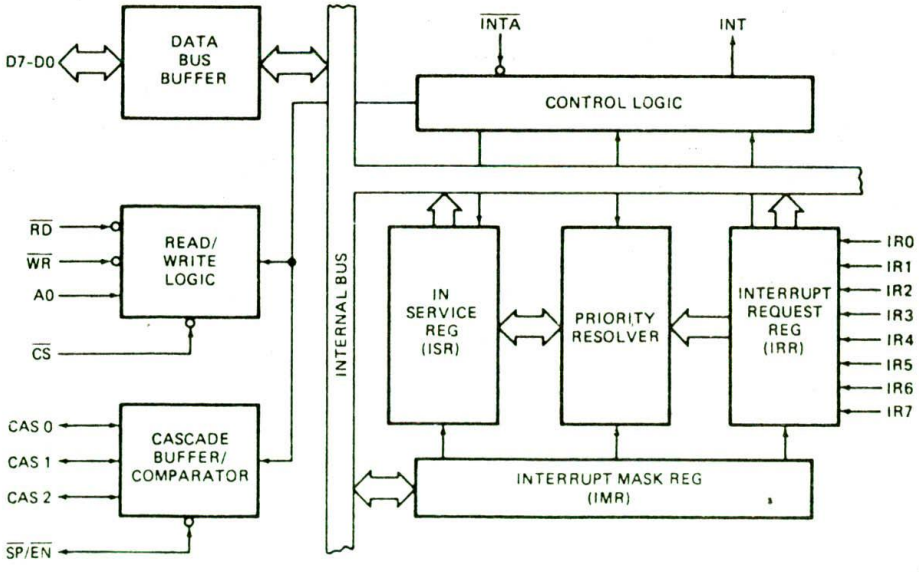


FIGURE 8-27 8259A internal block diagram. (*Intel Corporation*)

the highest priority first, assuming that both inputs are unmasked (enabled) in the 8259A.

Now let's look again at the block diagram of the 8259A in Figure 8-27 so we can explain in more detail how the device will respond to multiple interrupt signals. In the block diagram note the four boxes labeled *interrupt request register* (IRR), *interrupt mask register* (IMR), *in-service register* (ISR), and *priority resolver*.

The interrupt mask register is used to disable (mask) or enable (unmask) individual interrupt inputs. Each bit in this register corresponds to the interrupt input with the same number. You unmask an interrupt input by sending a command word with a 0 in the bit position that corresponds to that input.

The interrupt request register keeps track of which interrupt inputs are asking for service. If an interrupt input has an interrupt signal on it, then the corresponding bit in the interrupt request register will be set.

NOTE: An interrupt signal must remain high on an IR input until after the falling edge of the first $\overline{\text{INTA}}$ 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 $\overline{\text{INTA}}$ 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 Figure 8-28*a*, then this new INT signal will interrupt the 8086 again. When the 8086 sends out a second $\overline{\text{INTA}}$ 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 in-service register so that lower-priority interrupts can be

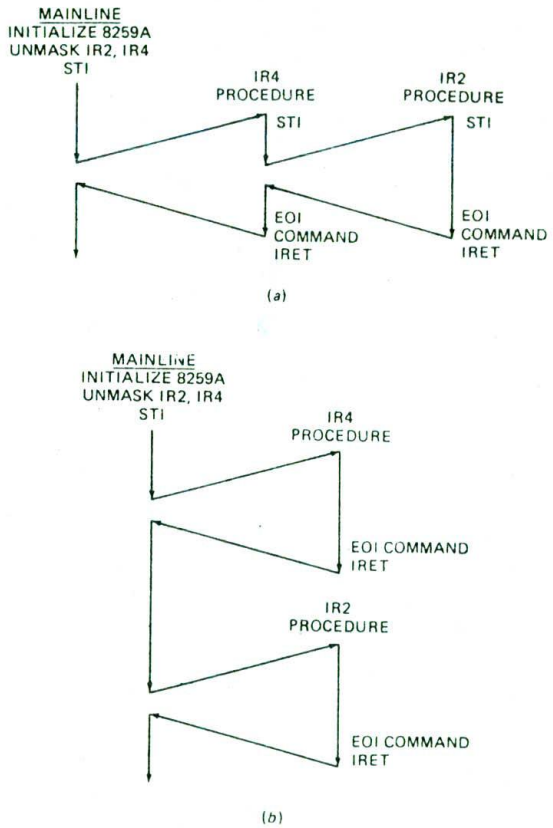


FIGURE 8-28 8259A and 8086 program flow for IR4 interrupt followed by IR2 interrupt. (*a*) Response with INTR enabled in IR4 procedure. (*b*) Response with INTR not enabled in IR4 procedure.

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 Figure 8-28*b*.

We can't describe all the possible cases, but the main point here is that the 8086 and the 8259A can be programmed to respond to interrupt signals from multiple sources in almost any way you want them to. Now, before we can show you how to initialize and write programs for an 8259A, we need to show you more about how one or more 8259As are connected in a microcomputer system.

8259A System Connections and Cascading

Figure 8-14 shows how an 8259A can be added to an SDK-86 board. As shown by the truth table in Figure 8-15, the 74LS138 address decoder will assert the $\overline{CS}$ input of the 8259A when an I/O base address of FF00H is on the address bus. The A0 input of the 8259A is used to select one of two internal addresses in the device. This pin is connected to system address line A1, so the system addresses for the two internal addresses of the 8259A are FF00H and FF02H. The eight data lines of the 8259A are always connected to the lower half of the 8086 data bus because the 8086 expects to receive interrupt types on these lower eight data lines. $\overline{RD}$ and $\overline{WR}$ are connected to the system $\overline{RD}$ and $\overline{WR}$ lines. $\overline{INTA}$ from the 8086 is connected to $\overline{INTA}$ on the 8259A. The interrupt request signal, INT, from the 8259A is connected to the INTR input of the 8086. The multipurpose $\overline{SP/EN}$ pin is tied high because we are using only one 8259A in this system. When just one 8259A is used in a system, the cascade lines (CAS0, CAS1, and CAS2) can be left open. The eight IR inputs are available for interrupt signals. Unused IR inputs should be tied to ground so that a noise pulse cannot accidentally cause an interrupt. In a later section we will show you how to initialize this 8259A, but first we need to show you how more than one 8259A can be added to a system.

The dashed box on the right side of Figure 8-14 shows how another 8259A could be added to the SDK-86 system to give a total of 15 interrupt inputs. If needed, an 8259A could be connected to each of the eight IR inputs of the original 8259A to give a total of 64 interrupt inputs. Note that since the 8086 has only one INTR input, only one of the 8259A INT pins is connected to the 8086 INTR pin. The 8259A connected directly into the 8086 INTR pin is referred to as the *master*. The INT pin from the other 8259A connects into an IR input on the master. This secondary, or *cascaded*, device is referred to as a *slave*. Note that the $\overline{INTA}$ signal from the 8086 goes to both the master and the slave devices.

Each 8259A has its own addresses so that command words can be written to it and status bytes read from it. For the cascaded 8259A in Figure 8-14, the two system I/O addresses will be FF08H and FF0AH.

The cascade pins (CAS0, CAS1, and CAS2) from the master are connected to the corresponding pins of the slave. For the master, these pins function as outputs, and for the slave device, they function as inputs. A further difference between the master and the slave is that on the slave the $\overline{SP/EN}$ pin is tied low to let the device know that it is a slave.

Briefly, here is how the master and the slave work when the slave receives an interrupt signal on one of its IR inputs. If that IR input is unmasked on the slave and if that input is a higher priority than any other interrupt level being serviced in the slave, then the slave will send an INT signal to the IR input of the master. If that IR input of the master is unmasked and if that input is a higher priority than any other IR inputs currently being serviced in the master, then the master will send an INT signal to the 8086 INTR input. If the 8086 INTR is enabled, the 8086 will go through its INTR interrupt procedure and send out two $\overline{INTA}$ pulses to both the master and the slave. The slave ignores the first interrupt acknowledge pulse, but when the master receives the first $\overline{INTA}$ pulse, it outputs a 3-bit slave identification number on the CAS0, CAS1, and CAS2 lines. (Each slave in a system is assigned a 3-bit ID as part of its initialization.) Sending the 3-bit ID number enables the slave. When the slave receives the second $\overline{INTA}$ pulse from the 8086, the slave will send the desired interrupt type number to the 8086 on the lower eight data bus lines.

If an interrupt signal is applied directly to one of the IR inputs on the master, the master will send the desired interrupt type to the 8086 when it receives the second $\overline{INTA}$ pulse from the 8086.

Now that we have given you an overview of how an 8259A operates and how 8259As can be cascaded, the initialization command words for the 8259A should make some sense to you.

Initializing an 8259A

Earlier in this chapter, when we showed you how to initialize an 8254, we listed a series of steps you should go through to initialize any programmable device. To refresh your memory of these very important steps, we will work quickly through them again for the 8259A.

The first step in initializing any device is to find the system base address for the device from the schematic or from a memory map for the system. In order to have a specific example here, we will use the 8259A shown in Figure 8-14. The base address for the 8259A in this system is FF00H.

The next step is to find the internal addresses for the device. For an 8259A the two internal addresses are selected by a high or a low on the A0 pin. In the circuit in Figure 8-14, the A0 pin is connected to system address line A1, so the internal addresses correspond to 0 and 2.

Next, you add the internal addresses to the base address for the device to get the system address for each internal part of the device. The two system addresses for this 8259A then are FF00H and FF02H.

Next, look at Figure 8-29*a* for the format of the command words that must be sent to an 8259A to

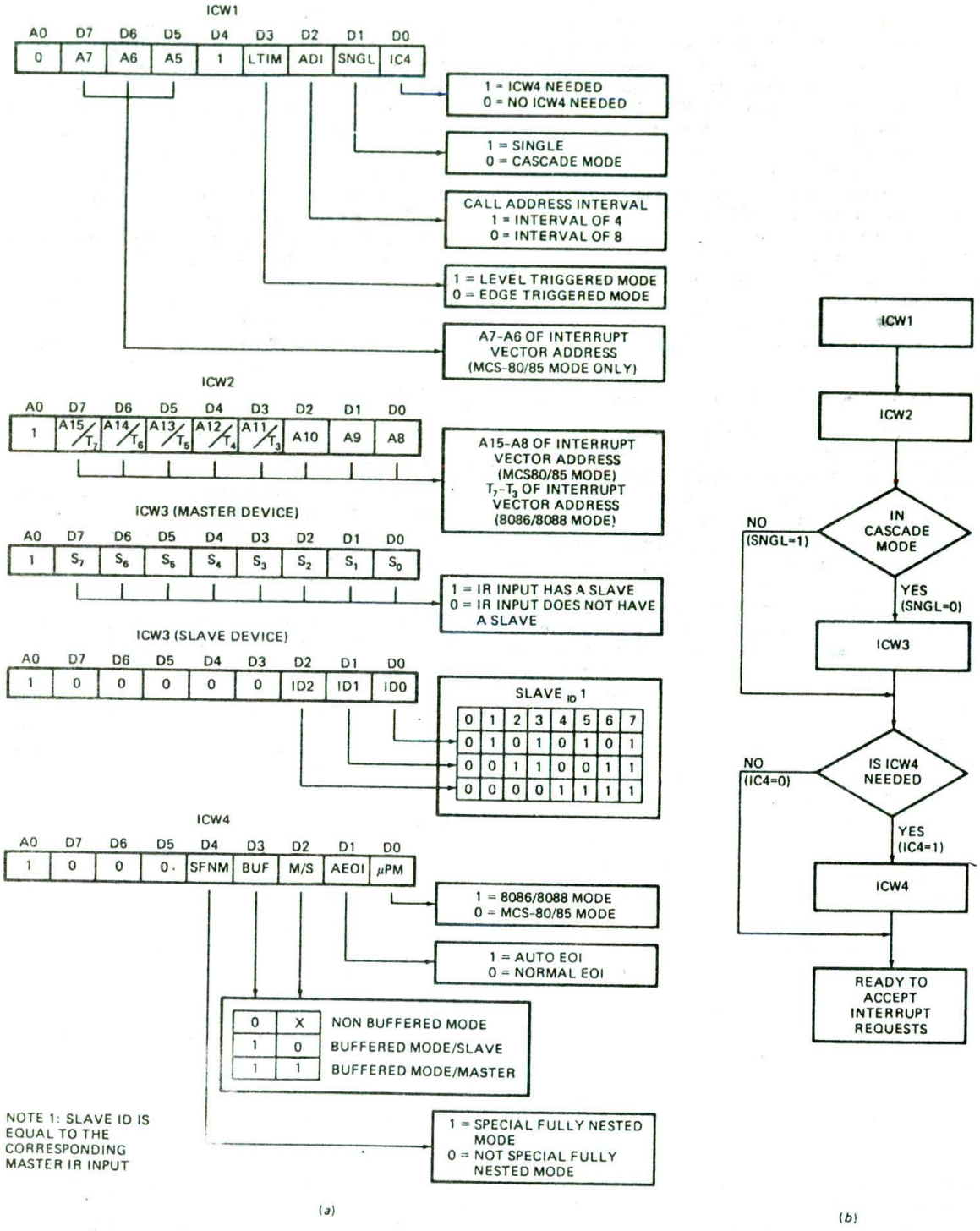


FIGURE 8-29 8259A initialization command word formats and sending order. (a) Formats. (b) Sending order and requirements. (*Intel Corporation*)

initialize it. The sight of all these command words may seem overwhelming at first, but taken one at a time, they are quite straightforward. To help you see which initialization command words (ICWs) are needed for various 8259A applications, Figure 8-29*b* shows this in flowchart form. According to this flowchart, an ICW1 and an ICW2 must be sent to any 8259A in the system. If the system has any slave 8259As (cascade mode), then an ICW3 must be sent to the master, and a different ICW3 must be sent to the slave. If the system is an 8086 or if you want to specify certain special conditions, then you have to send an ICW4 to the master and to each slave. Now let's look at the formats for the different ICWs.

The first thing to notice about the ICW formats in Figure 8-29*a* is that the bit labeled A0 on the left end of each of these is not part of the actual command word. This bit tells you the internal address that the control word must be sent to. The A0 = 0 next to ICW1, for example, tells you that ICW1 must be sent to internal address 0, which for our 8259A corresponds to system address FF00H.

The next step in the initialization procedure is to make up the control words. The least significant bit of ICW1 tells the 8259A whether it needs to look for an ICW4 or not. Since we are using the device in an 8086 system, we need to send ICW4. Therefore we make bit D0 a 1. We only want to use one 8259A for now, so we make bit D1 a 1. When used with an 8086, bit D2 is a don't care, so we make it a 0. Bit D3 is used to specify level-triggered mode or edge-triggered mode. In level-triggered mode, service will be requested whenever a high level is present on an IR input. In edge-triggered mode, a signal on an IR input must go from low to high and stay high until serviced. We usually use the edge-triggered mode so that a signal such as a square wave will not cause multiple interrupts. Making bit D3 a 0 does this. Bit D4 has to be a 1. For operation in an 8086 system, bits D5, D6, and D7 are don't cares, so we make them 0's for simplicity. Therefore, the ICW1 for our example here is 00010011.

In an 8086 system, ICW2 is used to tell the 8259A the type number to send in response to an interrupt signal on the IR0 input. In response to an interrupt signal on some other IR input, the 8259A will automatically add the number of the IR input to this base number and send the result to the 8086 as the type number for that input. Because 8086 interrupt types 0 through 31 are either dedicated or reserved, type 32 (decimal) is the lowest type number available for us to use. If we send the 8259A an ICW2 of 00100000 binary or 32 decimal, the 8259A will send this number as the type to the 8086 in response to an IR0 interrupt. For an interrupt request on the IR1 input, the 8259A will send 00100001 binary or 33 decimal, and for an interrupt request on the IR2 input, the 8259A will send an interrupt type 00100010 binary or 34 decimal. The same pattern continues for interrupt requests on the remaining IR inputs. In any ICW2 you send the 8259A, the lowest three bits must always be 0's because the 8259A automatically supplies these bits to correspond to the number of the IR input.

Since we are not using a slave in this example, we don't need to send an ICW3. If you are using a slave 8259A in a system, you have to send an ICW3 to the master to tell it which IR inputs have slaves. The master has to be told this so that it knows for which IR input signals it has to send out a slave ID number on the CAS0, CAS1, and CAS2 lines. You have to send an ICW3 to a slave 8259A to give it an ID number. The ID number you give a slave is equal to the IR input of the master that its INT output is connected to. When the master sends out an ID number on the CAS lines, the slave will recognize its ID number and output the desired type to the 8086 when it receives an $\overline{\text{INTA}}$ pulse.

For our example here, the only reason we need to send an ICW4 is to let the 8259A know that it is operating in an 8086 system. We do this by making bit D0 of the word a 1. Another interesting bit in this command word is D1, the automatic end-of-interrupt bit. If this bit is set in ICW4, the 8259A will automatically reset the in-service register bit for the interrupt input that is being responded to when the second interrupt-acknowledge pulse is received. The effect of this is that the 8259A will then be able to respond to an interrupt signal on a lower-priority IR input. In other words, a lower-priority interrupt input could then interrupt a higher-priority procedure. Since we don't want automatic end of interrupt, the ICW4 for our example here is 00000001.

In addition to the initialization command words shown in Figure 8-29*a*, the 8259A has a second set of command words called *operation command words*, or OCWs. These are shown in Figure 8-30, p. 238. 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 11111001. 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 in the procedure. The effect of resetting the ISR bit for an interrupt level is that once the bit is reset, the 8259A can respond to interrupt signals of lower priority. In small systems we usually use the nonspecific End-of-Interrupt command word. The OCW2 for this is 00100000. When the 8259A receives this OCW, it will automatically reset the in-service register bit for the IR input currently being serviced. If you want to reset a specific ISR bit, you can send the 8259A an OCW2 with 011 in bits D7, D6, and D5, and the number of the ISR bit you want to reset in the lowest 3 bits of the word. You can also use OCW2 to tell the 8259A to rotate the priorities of the IR inputs so that after an IR input is serviced, it drops to the lowest priority. If you are interested, consult the Intel data sheet for more information on this and on the use of OCW3.

Now that we have made up the required ICWs and OCWs, the next step is to write the instructions to send these command words to the 8259A.

Figure 8-31, p. 239–40, shows an 8086 assembly language program which initializes an 8259A and demonstrates many of the concepts of this chapter. You can use this program as a pattern for writing programs

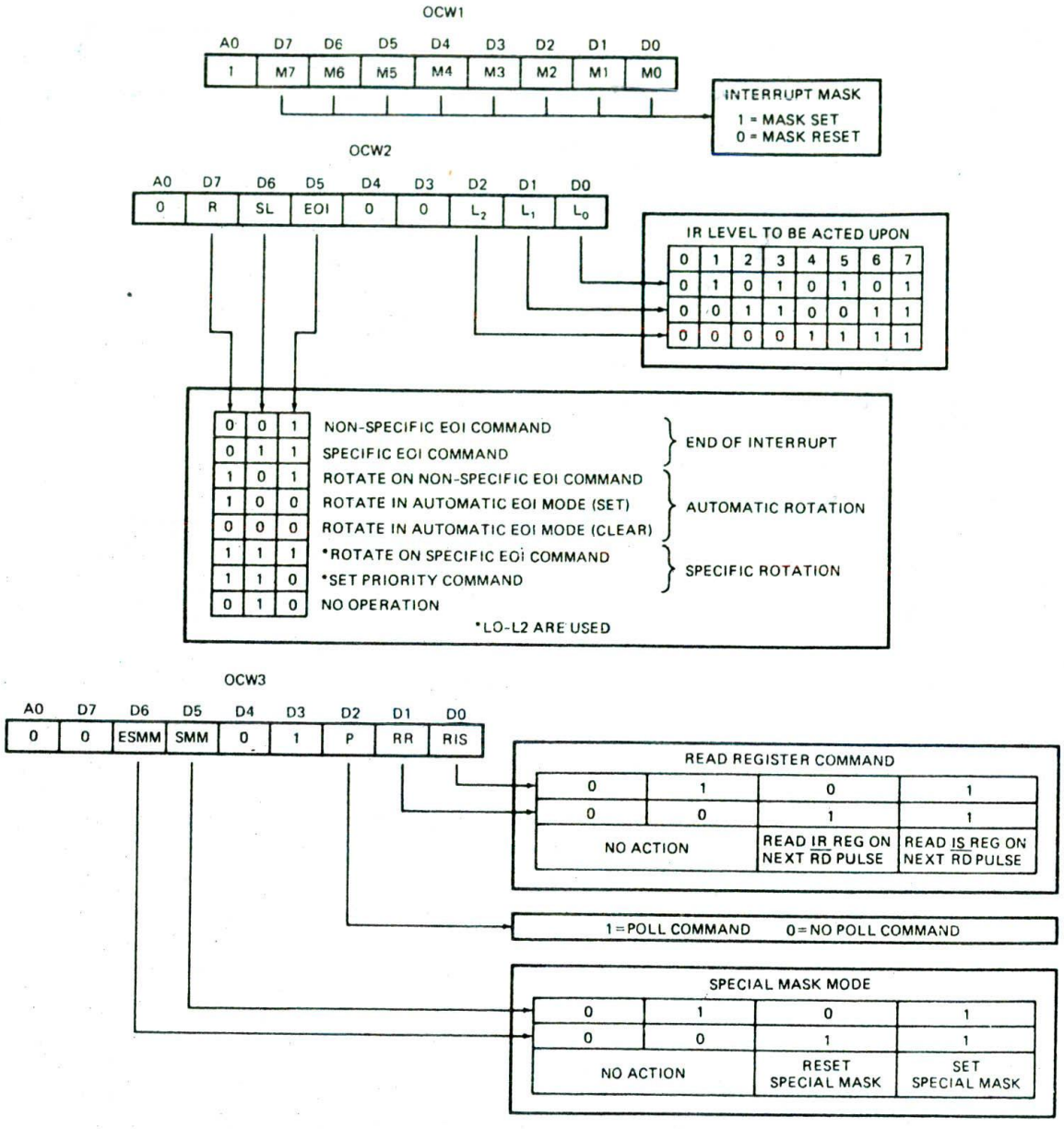


FIGURE 8-30 8259A operational command words. (*Intel Corporation*)

which service one or more interrupts. This program initializes the SDK-86 system in Figure 8-14 for generating a real-time clock of seconds, minutes, and hours from a 1-kHz interrupt signal and for reading ASCII codes from a keyboard on an interrupt basis. This program assumes that the 2.4576-MHz PCLK signal on the board is connected to the CLK input of the 8254 counter 0, the GATE input of the 8254 counter 0 is tied high, and the OUT pin of counter 0 is connected to the IR0 input of the 8259A. The program further assumes that the key-pressed strobe from the ASCII keyboard is connected to the IR2 input of the 8259A.

In the program, we first declare a segment called AINT_TABLE to reserve space for the vectors to the interrupt procedures. The statement TYPE_64 DW 2 DUP(0), for example, sets aside a word space for the offset of the type 64 procedure and a word for the segment base of the procedure. The statement TYPE_65 DW 2 DUP(0) sets aside a word for the offset of the type 65 procedure and a word space for the segment base

```
1                                   ;8086 PROGRAM F8-31.ASM
2                                   ; Program fragment to show the initialization of interrupt jump table,
3                                   ; 8259A priority interrupt controller, and 8254 programmable counter/timer
4
5 0000                              AINT_TABLE  SEGMENT  WORD  PUBLIC
6 0000  02*(0000)                      TYPE_64  DW   2 DUP(0)     ;Reserve space for clock procedure address
7 0004  02*(0000)                      TYPE_65  DW   2 DUP(0)     ;Not used in this program
8 0008  02*(0000)                      TYPE_66  DW   2 DUP(0)     ;Reserve space for keyboard procedure address
9 000C                              AINT_TABLE  ENDS
10
11 0000                             DATA SEGMENT WORD PUBLIC
12 0000  00                             SECONDS   DB   0
13 0001  00                             MINUTES   DB   0
14 0002  00                             HOURS     DB   0
15 0003  03E8                           INT_COUNT DW   03E8H        ;1 kHz interrupt counter
16 0005  64*(00)                        KEY_BUF   DB   100   DUP(0) ;Buffer for 100 ASCII characters
17 0069                             DATA ENDS
18
19 0000                             STACK_SEG  SEGMENT              ;No STACK directive used because
20 0000  64*(0000)                      DW        100   DUP(0)     ; will be using EXE2BIN
21                                      TOP_STACK LABEL WORD
22 00C8                             STACK_SEG     ENDS
23
24 0000                             CODE SEGMENT PUBLIC
25                                     ASSUME CS:CODE, DS:AINT_TABLE, SS:STACK_SEG
26 0000  B8 0000s                       MOV  AX, STACK_SEG              ;Initialize stack
27 0003  8E D0                          MOV  SS, AX                     ;segment register
28 0005  BC 00C8r                       MOV  SP, OFFSET TOP_STACK       ;Initialize stack pointer register
29 0008  B8 0000s                       MOV  AX, AINT_TABLE             ;Initialize data
30 000B  8E D8                          MOV  DS, AX                     ;segment register
31                                  ;Define the addresses for the interrupt service procedures
32 000D  C7 06 0002r 0000s              MOV  TYPE_64+2, SEG CLOCK       ; Put in clock procedure address
33 0013  C7 06 0000r 004Er              MOV  TYPE_64,   OFFSET CLOCK
34 0019  C7 06 000Ar 0000s              MOV  TYPE_66+2, SEG KEYBOARD    ; Put in keyboard procedure address
35 001F  C7 06 0008r 0055r              MOV  TYPE_66,   OFFSET KEYBOARD
36                                  ;Initialize data segment register
37                                      ASSUME DS:DATA
38 0025  B8 0000s                       MOV  AX, DATA
39 0028  8E D8                          MOV  DS, AX
40                                  ;Initialize 8259A priority interrupt controller
41 002A  B0 13                          MOV  AL, 00010011B      ; Edge triggered, single, ICW4
42 002C  BA FF00                        MOV  DX, 0FF00H         ; Point at 8259A control
43 002F  EE                             OUT  DX, AL             ; Send ICW1
44 0030  B0 40                          MOV  AL, 01000000B      ; Type 64 is first 8259A type
45 0032  BA FF02                        MOV  DX, 0FF02H         ; Point at ICW2 address
46 0035  EE                             OUT  DX, AL             ; Send ICW2
47 0036  B0 01                          MOV  AL, 00000001B      ; ICW4, 8086 mode
48 0038  EE                             OUT  DX, AL             ; Send ICW4
49 0039  B0 FA                          MOV  AL, 11111010B      ; OCW1 to unmask IR0 and IR2
50 003B  EE                             OUT  DX, AL             ; Send OCW1
51                                  ;Initialize 8254 counter 0
52 003C  B0 37                          MOV  AL, 00110111B     ; 1 kHz square wave, LSB then MSB, BCD
53 003E  BA FF07                        MOV  DX, 0FF07H        ; Point at 8254 control address
54 0041  EE                             OUT  DX, AL            ; Send counter 0 command word
55 0042  B0 58                          MOV  AL, 58H           ; Load LSB of count
56 0044  BA FF01                        MOV  DX, 0FF01H        ; Point at counter 0 data address
57 0047  EE                             OUT  DX, AL            ; Send LSB of count
58 0048  B0 24                          MOV  AL, 24H           ; Load MSB of count
59 004A  EE                             OUT  DX, AL            ; Send MSB of count
60                                  ;Enable interrupt input of 8086
61 004B  FB                             STI
62 004C  EB FE                      HERE:JMP  HERE             ; wait for interrupt
63
64 004E                             CLOCK  PROC  FAR
65                                  ;     :                     ; Clock procedure instructions
66 004E  B0 20                          MOV  AL, 00100000B     ; OCW2 for non-specific EOI
67 0050  BA FF00                        MOV  DX, 0FF00H        ; Address for OCW2
68 0053  EE                             OUT  DX, AL            ; Send OCW2 for end of interrupt
69 0054  CF                             IRET
70 0055                             CLOCK  ENDP
```

FIGURE 8-31 Assembly language program showing initialization of 8086, 8259A, and 8254 for real-time clock and keyboard interrupt procedures. (*Continued on next page.*)

```
71
72 0055                          KEYBOARD PROC FAR
73                              ;       :                       ; Keyboard procedure instructions
74 0055  B0 20                          MOV  AL, 00100000B      ; OCW2 for non-specific EOI
75 0057  BA FF00                        MOV  DX, 0FF00H         ; Address for OCW2
76 005A  EE                             OUT  DX, AL             ; Send OCW2 for end of interrupt
77 005B  CF                             IRET
78 005C                         KEYBOARD ENDP
79 005C                         CODE ENDS
80                                      END
```

FIGURE 8-31 *(continued)*.

address of the type 65 procedure, etc. As you will soon see, we use program instructions to load the actual starting addresses of the interrupt procedures in these locations.

NOTE: Because of the way the EXE2BIN program works, the AINT_TABLE segment must be first in your program so that it will be located at absolute address 0000:0100H, where it must be for the program to work correctly when downloaded to an SDK-86 board.

The next thing we do in our program is to declare a data segment and set aside some memory locations for seconds count, minutes count, hours count, and 100 characters read in from the keyboard. After the data segment, we set up a stack segment.

At the start of the mainline, we initialize the stack segment register and the stack pointer register. Then we initialize the DS register to point to the interrupt-vector table we set up at the start of the program.

The next four instructions load the addresses of the clock and keyboard procedures in the type 64 and type 66 locations in the interrupt-pointer table.

After we load the interrupt-vector table, we ASSUME DS:DATA and initialize DS to point to the data segment which contains the data for the clock and keyboard.

The next step is to initialize the 8259A as we described in the preceding section. The A0 bit next to ICW1 in Figure 8-29 is a 0, so ICW1 is sent to the lower of the two addresses for the 8259A, FF00H. For the example here we chose type 64 to correspond to an IR0 interrupt, so the needed ICW2 will be 01000000. The A0 bit next to ICW2 in Figure 8-29 is a 1, so ICW2 is sent to the higher of the two addresses for the 8259A, FF02H. Likewise, ICW4 and OCW1 are sent to system address FF02H.

The next section of the mainline program initializes counter 0 of the 8254 for mode 3, BCD countdown, and read/write LSB then MSB. To produce a 1-kHz signal from the 2.4576-MHz PCLK, we then write a count of 2458 to counter 0. This will not give exactly 1 kHz, but it is as close as we can get with this particular input clock frequency. The PCLK frequency for this board was chosen to make baud rate clock frequencies come out exact, not a 1-kHz real-time clock.

Finally, after the timer is initialized, we enable the 8086 INTR input with the STI instruction so that the 8086 can respond to INT signals from the 8259A, and wait for an interrupt with the HERE:JMP HERE instruction.

For the two interrupt-service procedures, we show just the skeletons and the End-of-Interrupt instructions. We leave it to you to write the actual procedures. Note that the interrupt procedures must be declared as far so that the assembler will load both the IP and the CS values in the interrupt-pointer table. Also note the End-of-Interrupt operation at the end of each procedure.

Remember from a previous discussion that when the 8259A responds to an IR signal, it sets the corresponding bit in the ISR. This bit must be reset at some time during or at the end of the interrupt-service procedure so that the priority resolver can respond to future interrupts of the same or lower priority. At the end of our procedures here we do this by sending an OCW2 to the 8259A. The OCW2 of 00100000 that we send tells the 8259A to reset the ISR bit for the IR level that is currently being serviced. This is a nonspecific End-of-Interrupt (EOI) instruction.

SOFTWARE INTERRUPT APPLICATIONS

In an earlier section of the chapter, we described how the 8086 software interrupt instruction INT N can be used to test any type of interrupt procedure. For example, to test a type 64 interrupt procedure without the need for external hardware such as we described in the preceding section, you can just execute the instruction INT 64.

Another important use of software interrupts is to call *B*asic *I*nput *O*utput *S*ystem, or BIOS, procedures in an IBM PC-type computer. These procedures in the system ROMs perform specific input or output functions, such as reading a character from the keyboard, writing some characters to the CRT, or reading some information from a disk.

To call one of these procedures, you load any required parameters in specified registers and execute an INT N instruction. N in this case is the interrupt type which vectors to the desired procedure. You can read the BIOS section of the IBM PC technical reference manual to get all the details of these if you need them, but here's an example of how you might use one of them.

Suppose that, as part of an assembly language program that you are writing to run on an IBM PC-type computer, you want to send some characters to the printer. The INT 17H instruction can be used to call a procedure which will do this.

Figure 8-32 shows the header for the INT 17H procedure from the IBM PC BIOS listing. Note that the DX, AH, and AL registers are used to pass the required

```
;INT 17
;Printer_IO - Provides communication with the printer
;INPUT:
; AH=0 Print the character in AL.
;      On  eturn AH=1 if character could not be
;      printed (Time out). Other bits set as on
;      normal status call.
; AH=1 Initialize printer port.
;      Returns with AH set with printer status
; AH=2 Read the printer status into AH
;      bit   7  6  5  4  3  2  1  0
;                                 └ Time out
;                                └── Unused
;                              1=I/O error
;                            1=selected
;                          1=out of paper
;                        1=acknowledge
;                      1=Not busy
;
; DX =  Printer to be used (0,1,2) corresponding to
;       actual values in Printer_Base area
;
; Data area Printer_Base contains the base address to
;   the printer card(s) available (Located at beginning
;   of data segment 408H absolute, 3 words)
; Data area Print_Tim_Out (byte) may be changed to
;   cause different time out waits. Default=20
;
; REGISTERS: AH is modified, all others unchanged.
```

FIGURE 8-32 Header for INT 17 procedure.
(IBM Corporation)

parameters to the procedure. Also note that the procedure is used for two different operations: initializing the printer port and sending a character to the printer. The operation performed by the procedure is determined by the number passed to the procedure in the AH register. AH = 1 means initialize the printer port, AH = 0 means print the character in AL, and AH = 2 means read the printer status and return it in AH. If an attempt to print a character was not successful for some reason, such as the printer not being turned on, not being selected, or being busy, 01 is returned in AH.

Now work your way through the program example in Figure 8-33 to see how the INT 17H procedure is called to initialize the printer and how it is called over and over to send a text string to the printer. Note that we sent a carriage return character and a linefeed character after the text string because the printer will not print a line until it receives a carriage return.

The main advantage of calling procedures with software interrupts is that you don't need to worry about the absolute address where the procedure actually resides or about trying to link the procedure into your program. All 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

```
 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                          STACK_SEG SEGMENT  STACK
11 0000  C8*(0000)                    DW       200    DUP(0)    ; Set aside 200 words for stack
12                                    STACK_TOP LABEL  WORD      ; Assign name to word above stack top
13 0190                          STACK_SEG ENDS
14
15       = 001E                  CHAR_COUNT EQU  30
16
17 0000                          DATA    SEGMENT
18 0000  48 45 4C 4C 4F 20 54 +       MESSAGE       DB   'HELLO THERE, HOW ARE YOU?'
19       48 45 52 45 2C 20 48 +
20       4F 57 20 41 52 45 20 +
21       59 4F 55 3F
22 0019  0D 0A 0D 0A                  MESSAGE_END   DB   0DH, 0AH, 0DH, 0AH         ; return & line feed
23 001D                          DATA    ENDS
24
25 0000                          CODE    SEGMENT
26                                 ASSUME CS:CODE, SS:STACK_SEG, DS:DATA
27 0000  B8 0000s                START:  MOV  AX, STACK_SEG        ; Initialize stack segment register
28 0003  8E D0                           MOV  SS, AX
29 0005  BC 0190r                        MOV  SP, OFFSET STACK_TOP ; Initialize stack pointer register
30 0008  B8 0000s                        MOV  AX, DATA             ; Initialize data segment register
31 000B  8E D8                           MOV  DS, AX
32 000D  B4 01                           MOV  AH, 01               ; Set up registers to initialize printer
33 000F  BA 0000                         MOV  DX, 0                ; port 0
34 0012  CD 17                           INT  17H                  ; Call procedure to intitialize printer port
35 0014  8D 1E 0000r                     LEA  BX, MESSAGE          ; Get to start of message
36 0018  B9 001E                         MOV  CX, CHAR_COUNT       ; Set up a count variable
37 001B  B4 00                    AGAIN: MOV  AH, 0                ; Load code for procedure to send character
```

FIGURE 8-33 8086 assembly language program for outputting characters to a printer. *(Continued on next page.)*

```
38 001D  8A 07                    MOV  AL, [BX]      ; Load character to be sent into AL
39 001F  CD 17                    INT  17H           ; Use BIOS routine to send character to printer
40 0021  80 FC 01                 CMP  AH, 01H       ; If character not printed then returns AH =1
41 0024  75 04                    JNE  NEXT          ; IF character not printed THEN
42 0026  F9                NOT_RDY:STC               ; Set carry to indicate message not sent
43 0027  EB 05 90                 JMP  EXIT          ; and leave loop
44 002A  F8                NEXT:  CLC                ; ELSE Clear carry flag (character sent)
45 002B  43                       INC  BX            ; Move to address of next character
46 002C  E2 ED                    LOOP AGAIN         ; Send the next character
47 002E  B8 4C00           EXIT:  MOV  AX, 4C00H     ; Graceful exit to DOS
48 0031  CD 21                    INT  21H           ; with function call 4CH
49 0033                    CODE   ENDS
50                                END  START
```

FIGURE 8--33 (*continued*).

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

Nonmaskable interrupt—type 2

Breakpoint interrupt—type 3

Overflow interrupt—type 4

Software interrupts—INT types 0 through 255

INTR interrupts—types 0 through 255

Edge- and level-activated interrupt input

Interrupt priority

Programmable timer/counter devices—8253, 8254

Initialization steps for peripheral devices

Internal addresses

Control words, command words, mode words

8259A priority interrupt controller
- In-service register (ISR)
- Priority resolver
- Interrupt request register (IRR)
- Interrupt mask register (IMR)

BIOS

REVIEW QUESTIONS AND PROBLEMS

1. List and describe in general terms the steps an 8086 will take when it responds to an interrupt.
2. Describe the purpose of the 8086 interrupt-vector table.
3. What addresses in the interrupt-vector table are used for a type 2 interrupt?
4. The starting address for a type 4 interrupt-service procedure is 0010:0082. Show where and in what order this address should be placed in the interrupt-vector table.
5. Address 00080H in the interrupt-vector table contains 4A24H, and address 00082H contains 0040H.
   a. To what interrupt type do these locations correspond?
   b. What is the starting address for the interrupt-service procedure?
6. Briefly describe the condition(s) which cause the 8086 to perform each of the following types of interrupts: type 0, type 1, type 2, type 3, type 4.
7. Why is it necessary to PUSH all registers used in the procedure at the start of an interrupt-service procedure and to POP them at the end of the procedure?
8. Why must you use an IRET instruction rather than the regular RET instruction at the end of an interrupt-service procedure?
9. Show the assembler directive and instructions you would use to initialize the interrupt-pointer table locations for a type 0 procedure called DIV_0_ERROR and a type 2 procedure called POWER_FAIL.
10. *a.* Describe the main use of the 8086 type 1 interrupt.

b. Show the assembly language instructions necessary to set the 8086 trap flag.

11. In a system which has battery-backed RAM for saving data in case of a power failure, the stack is often put in the battery-backed RAM. This makes it easy to save registers and critical program data. Assume that the battery-backed RAM is in the address range of 08000H through 08FFFH. Write an 8086 power failure interrupt-service procedure which

Sets an external battery-backed flip-flop connected to bit 0 of port 28H to indicate that a power failure has occurred.

Saves all registers on the stack.

Saves the stack pointer value for the last entry at location 8000H.

Saves the contents of memory locations 00100H through 003FFH after the saved stack pointer value at the start of the battery-backed memory. (A string instruction might be useful for this.)

Halts.

When the power comes back on, the start-up routine can check the power fail flip-flop. If the flip-flop is set, the start-up procedure can copy the saved data back into its operating locations, initialize the stack segment register, and then get the saved SP value from address 08000H. Using this value, it can restore the pushed registers and return execution to where the power fail interrupt occurred. This is called a "warm start." If we don't want it to do a warm start, we can reset the flip-flop with an external RESET key so the system does a start from scratch, or "cold start."

12. a. Why is the 8086 INTR input automatically disabled when the 8086 is RESET?
    b. How is the 8086 INTR input enabled to respond to interrupts?
    c. What instruction can be used to disable the INTR input?
    d. Why is the INTR input automatically disabled as part of the response to an INTR interrupt?
    e. How is the INTR input automatically reenabled at the end of an INTR interrupt-service procedure?

13. Describe the response an 8086 will make if it receives an NMI interrupt signal during a division operation which produces a divide-by-zero interrupt.

14. The data outputs of an 8-bit analog-to-digital converter are connected to bits D0–D7 of port FFF9H, and the end-of-conversion signal from the A/D converter is connected to the NMI input of an 8086. Write a simple mainline program and an interrupt-service procedure which reads in a byte of data from the converter. If the MSB of the data is a 0, indicating that the value is in range, add the byte to a running total kept in two successive memory locations. If the MSB of data is 1, showing that the value is out of range, ignore the input. After 100 samples have been totaled, divide by 100 to get the average, store this average in another reserved memory location, and reset the total to 0.

15. Write the algorithm and the program for an interrupt-service procedure which turns an LED connected to bit D0 of port FFFAH on for 25 s and off for 25 s. The procedure should also turn a second LED connected to bit D1 of port FFFAH on for 1 min and off for 1 min. Assume that a 1-Hz interrupt signal is connected to the NMI input of an 8086 and that a high on a port bit turns on the LED connected to it.

16. Write the algorithms for a mainline program and an interrupt-service procedure which generate a real-time clock of seconds, minutes, and hours in three memory locations using a 1-Hz signal applied to the NMI input of an 8086. Then write the assembly language programs for the mainline and the procedure. If you are working on an SDK-86 board, there is a procedure in Figure 9-32 that you can add to your program to display the time on the data and address field LEDs of the board. You can use this procedure without needing to understand the details of how it works. To display a word on the data field, simply put the word in the CX register, put 00H in AL, and call the procedure. To display a word on the address field, put the word in CX, 01H in AL, and call the procedure. *Hint:* Clear carry before incrementing a count in AL so that DAA works correctly.

17. In Chapter 5 we discussed using breakpoints to debug programs containing procedures. List the sequence of locations where you would put breakpoints in the example program in Figure 8-9 to debug it if it did not work when you loaded it into memory.

18. Suppose that we add another 8254 to the SDK-86 add-on circuitry shown in Figure 8-14 and that the $\overline{CS}$ input of the new 8254 is connected to the Y5 output of the 74LS138 decoder.
    a. What will be the system base address for this added 8254?
    b. To which half of the 8086 data bus should the eight data lines from this 8254 be connected?
    c. What will be the system addresses for the three counters and the control word register in this 8254?
    d. Show the control word you would use to initialize counter 1 of this device for read/write LSB then MSB, mode 3, and BCD countdown.
    e. Show the sequence of instructions you would use to write this control word and a count of 0356 to the counter.
    f. Assuming that the GATE input is high, when does the counter start counting down in mode 3?

g. Assuming initialization as in parts *d* and *f*, and that a 712-kHz signal is applied to the CLK input of counter 1 in mode 3, describe the frequency, period, and duty cycle of the waveform that will be on the OUT pin of the counter.

h. Describe the effect that a control word of 10010000 sent to this 8254 will have.

19. Show the instructions you would use to initialize counter 2 of the 8254 in Figure 8-14 to produce a 1.2-ms-wide STROBE pulse on its OUT pin when it receives a trigger input on its GATE input.

20. Show the instructions needed to latch and read a 16-bit count from counter 1 of the 8254 in Figure 8-14.

21. Describe the sequence of actions that an 8259A and an 8086, as connected in Figure 8-14, will take when the 8259A receives an interrupt signal on its IR2 input. Assume only IR2 is unmasked in the 8259A and that the 8086 INTR input has been enabled with an STI instruction.

22. Describe the use of the CAS0, CAS1, and CAS2 lines in a system with a cascaded 8259A.

23. Describe the response that an 8259A will make if it receives an interrupt signal on its IR3 and IR5 inputs at the same time. Assume fixed priority for the IR inputs. What response will the 8259A make if it is servicing an IR5 interrupt and an IR3 interrupt signal occurs?

24. Why is it necessary to send an End-of-Interrupt (EOI) command to an 8259A at some time in an interrupt-service routine?

25. Show the sequence of command words and instructions that you would use to initialize an 8259A with a base address of FF10H as follows: edge-triggered; only one 8259A; 8086 system; interrupt type 40 corresponds to IR0 input; normal EOI; nonbuffered mode, not special fully nested mode; IR1 and IR3 unmasked.

26. What is the major advantage of calling BIOS procedures with software interrupts instead of calling them with absolute addresses?

CHAPTER 9

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 LED is always there and ready, so you can send data to it at any time. The timing waveform in Figure 9-1*a*, p. 246, represents this situation. The crossed lines on the waveform represent the time at which a new data byte becomes valid on the output lines of the port. The absence of other waveforms indicates that this output operation is not directly dependent on any other signals.

SIMPLE STROBE I/O

In many applications, valid data is present on an external device only at a certain time, so it must be read in at that time. An example of this is the ASCII-encoded keyboard discussed in Chapter 4. When a key is pressed,

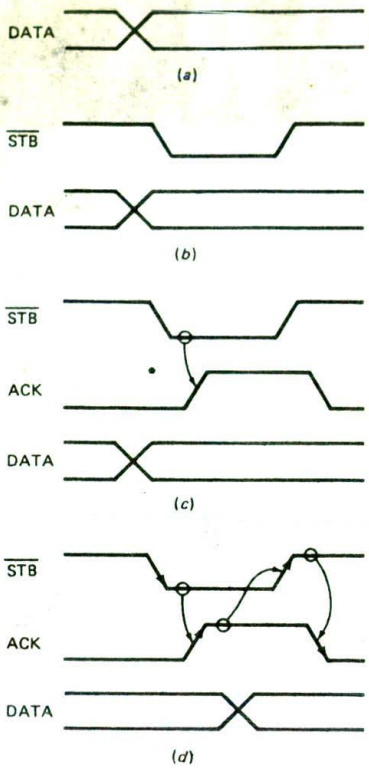


FIGURE 9-1 Parallel data transfer. (*a*) Simple output. (*b*) Simple strobe I/O. (c) Single handshake I/O. (*d*) Double handshake I/O.

circuitry on the keyboard sends out the ASCII code for the pressed key on eight parallel data lines, and then sends out a strobe signal on another line to indicate that valid data is present on the eight data lines. As shown in Figure 4-19, you can connect this strobe line to an input port line and poll it to determine when you can input valid data from the keyboard. Another alternative, described in Chapter 8, is to connect the strobe line to an interrupt input on the processor and have an interrupt service procedure read in the data when the processor receives an interrupt. The point here is that this transfer is time dependent. You can read in data only when a strobe pulse tells you that the data is valid.

Figure 9-1*b* shows the timing waveforms which represent this type of operation. The sending device, such as a keyboard, outputs parallel data on the data lines, and then outputs an $\overline{\text{STB}}$ signal to let you know that valid data is present.

For low rates of data transfer, such as from a keyboard to a microprocessor, a simple strobe transfer works well. However, for higher-speed data transfer this method does not work, because there is no signal which tells the sending device when it is safe to send the next data byte. In other words, the sending system might send data bytes faster than the receiving system could read them. To prevent this problem, a *handshake* data transfer scheme is used.

SINGLE-HANDSHAKE I/O

Figure 9-2 shows the circuit connections and Figure 9-1*c* shows some example timing waveforms for a *handshake data transfer* from a peripheral device to a microprocessor. The peripheral outputs some parallel data and sends an $\overline{\text{STB}}$ signal to the microprocessor. The microprocessor detects the asserted $\overline{\text{STB}}$ signal on a polled or interrupt basis and reads in the byte of data. Then the microprocessor sends an Acknowledge signal (ACK) to the peripheral to indicate that the data has been read and that the peripheral can send the next byte of data. From the viewpoint of the microprocessor, this operation is referred to as a *handshake* or *strobed input*.

These same waveforms might represent a handshake output from a microprocessor to a parallel printer. In this case, the microprocessor outputs a character to the printer and asserts an $\overline{\text{STB}}$ signal to the printer to tell the printer, "Here is a character for you." When the printer is ready, it answers back with the ACK signal to tell the microprocessor, "I got that one; send me another." We will show you much more about printer interfacing in a later section.

The point of this handshake scheme is that the sending device or system is designed so that it does not send the next data byte until the receiving device or system indicates with an ACK signal that it is ready to receive the next byte.

DOUBLE-HANDSHAKE DATA TRANSFER

For data transfers where even more coordination is required between the sending system and the receiving

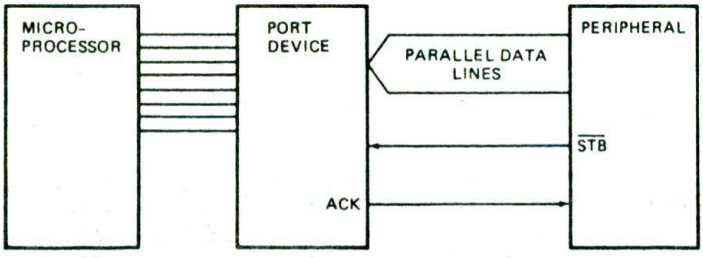


FIGURE 9-2 Signal directions for handshake input data transfer.

system, a *double handshake* is used. The circuit connections are the same as those in Figure 9-2. Figure 9-1*d* shows some example waveforms for a double-handshake input from a peripheral to a microprocessor. Perhaps it will help you to follow these waveforms by thinking of them as a conversation between two people. In these waveforms each signal edge has meaning. The sending device asserts its $\overline{STB}$ line low to ask, "Are you ready?" The receiving system raises its ACK line high to say, "I'm ready." The peripheral device then sends the byte of data and raises its $\overline{STB}$ line high to say, "Here is some valid data for you." After it has read in the data, the receiving system drops its ACK line low to say, "I have the data, thank you, and I await your request to send the next byte of data."

For a handshake output of this type, from a microprocessor to a peripheral, the waveforms are the same, but the microprocessor sends the $\overline{STB}$ signal and the data, and the peripheral sends the ACK signal. In the accompanying laboratory manual we show you how to interface with a speech-synthesizer device using this type of handshake system.

Implementing Handshake Data Transfer

For handshake data transfer, a microprocessor can determine when it is time to send the next data byte on a polled or on an interrupt basis. The interrupt approach is usually used, because it makes better use of the processor's time.

The $\overline{STB}$ or ACK signals for these handshake transfers can be produced on a port pin by instructions in the program. However, this method usually uses too much processor time, so parallel-port devices such as the 8255A have been designed to automatically manage the handshake operation. The 8255A, for example, can be programmed to automatically receive an $\overline{STB}$ signal from a peripheral, send an interrupt signal to the processor, and send the ACK signal to the peripheral at the proper times. The following sections show you how to connect, initialize, and use an 8255A for a variety of handshake and nonhandshake applications.

8255A Internal Block Diagram and System Connections

Figure 9-3, p. 248, shows the internal block diagram of the 8255A. Along the right side of the diagram, you can see that the device has 24 input/output lines. Port A can be used as an 8-bit input port or as an 8-bit output port. Likewise, port B can be used as an 8-bit input port or as an 8-bit output port. Port C can be used as an 8-bit input or output port, as two 4-bit ports, or to produce handshake signals for ports A and B. We will discuss the different modes for these lines in detail a little later.

Along the left side of the diagram, you see the signal lines used to connect the device to the system buses. Eight data lines allow you to write data bytes to a port or the control register and to read bytes from a port or the status register under the control of the $\overline{RD}$ and $\overline{WR}$ lines. The address inputs, A0 and A1, allow you to selectively access one of the three ports or the control register. The internal addresses for the device are: port A, 00; port B, 01; port C, 10; control, 11. Asserting the $\overline{CS}$ input of the 8255A enables it for reading or writing. The $\overline{CS}$ input will be connected to the output of the address decoder circuitry to select the device when it is addressed.

The RESET input of the 8255A is connected to the system reset line so that, when the system is reset, all the port lines are initialized as input lines. This is done to prevent destruction of circuitry connected to port lines. If port lines were initialized as outputs after a power-up or reset, the port might try to output to the output of a device connected to the port. The possible argument between the two outputs might destroy one or both of them. Therefore, all the programmable port devices initialize their port lines as inputs when reset.

We discussed in Chapter 7 how two 8255As can be connected in an 8086 system. Take a look at Figure 7-8 (sheet 5) to refresh your memory of these connections. Note that one of the 8255As is connected to the lower half of the 8086 data bus, and the other 8255A is connected to the upper half of the data bus. This is done so that a byte can be transferred by enabling one device, or a word can be transferred by enabling both devices at the same time. According to the truth table for the I/O port address decoder in Figure 7-16, the A40 8255A on the lower half of the data bus will be enabled for a base address of FFF8H, and the A35 8255A will be enabled for a base address of FFF9H.

Another point to notice in Figure 7-8 is that system address line A1 is connected to the 8255A A0 inputs, and system address line A2 is connected to the 8255A A1 inputs. With these connections, the system addresses for the three ports and the control register in the A40 8255A will be FFF8H, FFFAH, FFFCH, and FFFEH, as shown in Figure 7-16. Likewise, the system addresses for the three ports and the control register of the A35 8255A are FFF9H, FFFBH, FFFDH, and FFFFH.

8255A Operational Modes and Initialization

Figure 9-4, p. 249, 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.

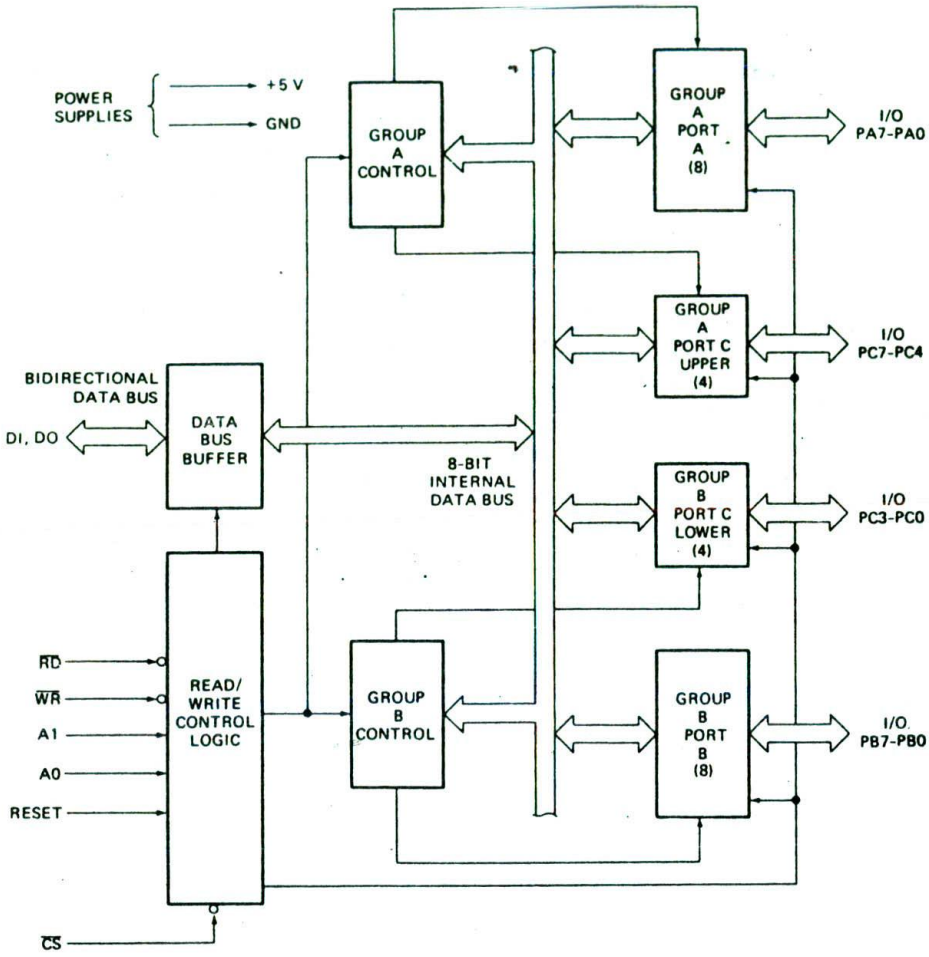


FIGURE 9-3 Internal block diagram of 8255A programmable parallel port device. (*Intel Corporation*)

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 Figure 9-5*a* to tell the device what modes you want the ports to operate in. You use the *bit set/reset control word* format in Figure 9-5*b* when you want to set or reset the output on a pin of port C or when you want to enable the

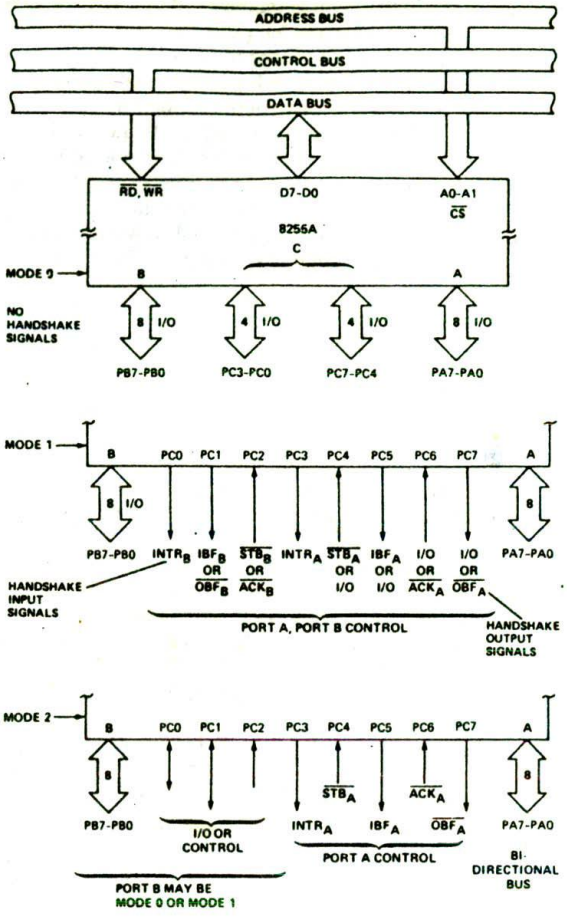


FIGURE 9-4 Summary of 8255A operating modes. (*Intel Corporation*)

interrupt 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 last chapter. As an example for this device, suppose that you want to initialize the 8255A (A40) in Figure 7-8 as follows:

Port B as mode 1 input

Port A as mode 0 output

Port C upper as inputs

Port C bit 3 as output

As we said previously, the base address for the A40 8255A is FFF8H, and the control register address is FFFEH. The next step is to make up the control word by figuring out what to put in each of the little boxes, one bit at a time. Figure 9-6*a*, p. 250, shows the control word which will program the 8255A as desired for this example. The figure also shows how you should document

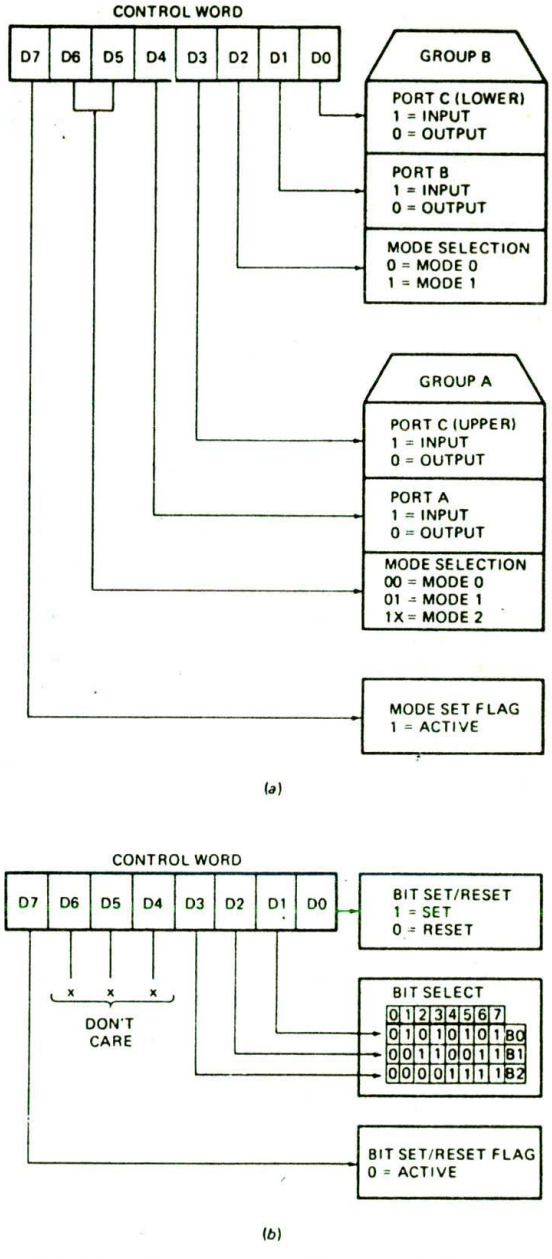


FIGURE 9-5 8255A control word formats. (*a*) Mode-set control word. (*b*) Port C bit set/reset control word.

any control words you make up for use in your programs. Using Figure 9-5*a*, work your way through this word to make sure you see why each bit has the value it does.

To send the control word, you load the control word in AL with a MOV AL,10001110B instruction, point DX at the port address with the MOV DX,0FFFEH instruction, and send the control word to the 8255A control register with the OUT DX,AL instruction.

As an example of how to use the bit set/reset control

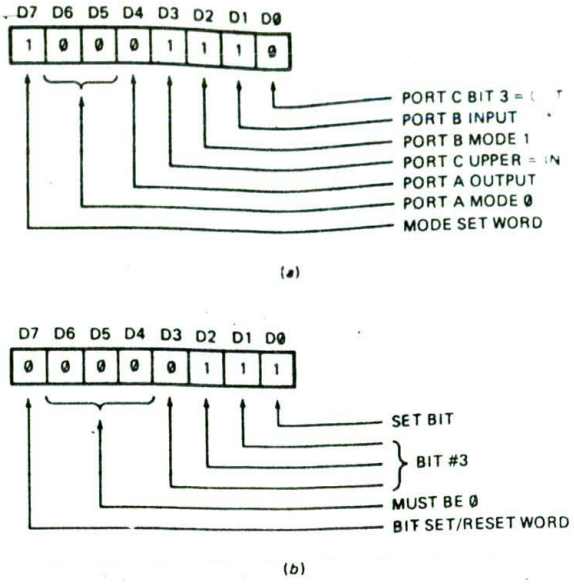


FIGURE 9-6 Control word examples for 8255A. (*a*) Mode-set control word. (*b*) Port C bit set/reset control word to set bit 3.

word, suppose that you want to output a 1 to (set) bit 3 of port C, which was initialized as an output with the mode set control word above. To set or reset a port C output pin, you use the bit set/reset control word shown in Figure 9-5*b*. Make bit D7 a 0 to identify this as a bit set/reset control word, and put a 1 in bit D0 to specify that you want to set a bit of port C. Bits D3, D2, and D1 are used to tell the 8255A which bit you want to act on. For this example you want to set bit 3, so you put 011 in these 3 bits. For simplicity and compatibility with future products, make the other 3 bits of the control word 0's. The result, 00000111B, is shown with proper documentation in Figure 9-6*b*.

To send this control word to the 8255A, simply load it into AL with the MOV AL,00000111B instruction, point DX at the control register address with the MOV DX,0FFFEH instruction if DX is not already pointing there, and send the control word with the OUT DX,AL instruction. As part of the application examples in the following sections, we will show you how you know which bit in port C to set to enable the interrupt output signal for handshake data transfer.

8255A Handshake Application Examples

INTERFACING TO A MICROCOMPUTER-CONTROLLED LATHE

All the machines in the machine shop of our computer-controlled electronics factory operate under microcomputer control. One example of these machines is a lathe which makes bolts from long rods of stainless steel. The cutting instructions for each type of bolt that we need to make are stored on a 3/4-in.-wide teletype-like metal tape. Each instruction is represented by a series of holes in the tape. A tape reader pulls the tape through an optical or mechanical sensor to detect the hole patterns and converts these to an 8-bit parallel code. The microcomputer reads the instruction codes from the tape reader on a handshake basis and sends the appropriate control instructions to the lathe. The microcomputer must also monitor various conditions around the lathe. It must, for example, make sure the lathe has cutting lubricant oil, is not out of material to work on, and is not jammed up in some way. Machines that operate in this way are often referred to as *computer numerical control*, or *CNC, machines*.

Figure 9-7 shows in diagram form how you might use an 8255A to interface a microcomputer to the tape reader and lathe. Later in the chapter, we will show you some of the actual circuitry needed to interface the port pins of the 8255A to the sensors and the high-power motors of the lathe. For now, we want to talk about initializing the 8255A for this application and analyze the timing waveforms for the handshake input of data from the tape reader.

Your first task is to make up the control word which will initialize the 8255A in the correct modes for this application. To do this, start by making a list showing how you want each port pin or group of pins to function. Then put in the control word bits that implement those pin functions. For our example here,

Port A needs to be initialized for handshake input (mode 1) because instruction codes have to be read in from the tape reader on a handshake basis.

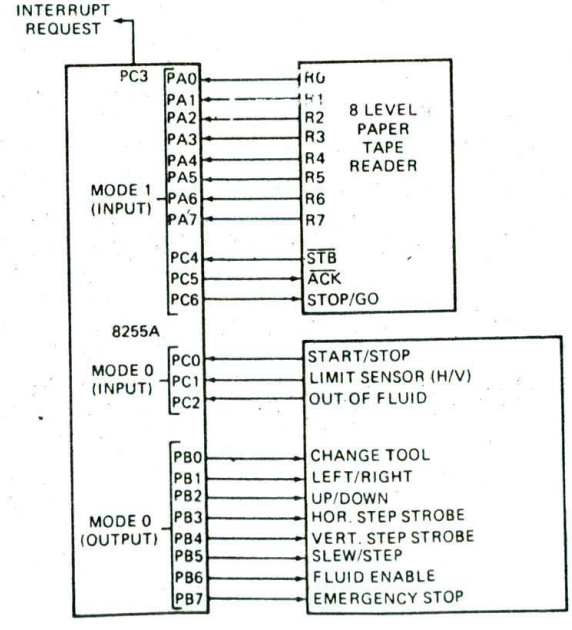


FIGURE 9-7 Interfacing a microprocessor to a tape reader and lathe.

Port B needs to be initialized for simple output (mode 0). No handshaking is needed here because this port is being used to output simple on or off control signals to the lathe.

Port C, bits PC0, PC1, and PC2 are used for simple input of sensor signals from the lathe.

Port C, bits PC3, PC4, and PC5 function as the handshake signals for the data transfer from the tape reader connected to port A.

Port C, bit PC6 is used for output of the STOP/GO signal to the tape reader.

Port C, bit PC7 is not used for this example.

Figure 9-8 shows the control word to initialize the 8255A for these pin functions. You send this word to the control register address of the 8255A as described above.

Before we go on, there is one more point we have to make about initializing the 8255A for this microcomputer-controlled lathe application. In order for the handshake input data transfer from the tape reader to work correctly, the interrupt request signal from bit PC3 has to be enabled. This is done by sending a bit set/reset control word for the appropriate bit of port C. Figure 9-9 shows the port C bit that must be set to enable the interrupt output signal for each of the 8255A handshake modes. For the example here, port A is being used for handshake input, so according to Figure 9-9, port C, bit PC4 must be set to enable the interrupt output for this operation. The bit set/reset control word to do this is 00001001B. You send this bit set/reset control word to the control address of the 8255A.

Handshake data transfer from the tape reader to the 8255A can be stopped by disabling the 8255A interrupt output on port C, pin PC3. This is done by resetting bit PC4 with a bit set/reset control word of 00001000. You will later see another example of the use of this interrupt enable/disable process in Figure 9-16.

As another example of 8255A interrupt output enabling, suppose that you are using port B as a handshake output port. According to Figure 9-9, you need to set bit PC2 to enable the 8255A interrupt output signal. The bit set/reset control word to do this is 00000101.

Now let's talk about how the program for this machine might operate and how the handshake data transfer actually takes place.

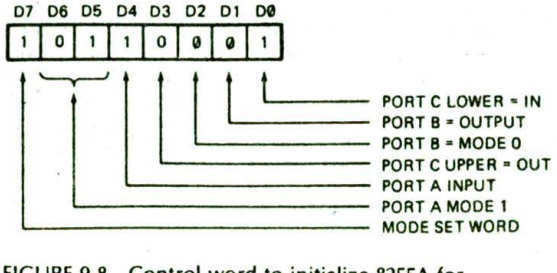


FIGURE 9-8 Control word to initialize 8255A for interface with tape reader and lathe.

	Port C Interrupt Signal Pin Number	To Enable Interrupt Request Set Port C bit
MODE 1		
Port A IN	PC3	PC4
Port B IN	PC0	PC2
Port A OUT	PC3	PC6
Port B OUT	PC0	PC2
MODE 2		
Port A IN	PC3	PC4
Port A OUT	PC3	PC6

FIGURE 9-9 Port C bits to set to enable interrupt request outputs for handshake modes.

After initializing everything, you would probably read port C, bits PC0, PC1, and PC2 to check if the lathe was ready to operate. For any 8255A mode, you read port C by simply doing an input from the port C address. Then you output a start command to the tape reader on bit PC6. This is done with a bit set/reset command. Assuming that you want to reset bit PC6 to start the tape reader, the bit set/reset control word for this is 00001100. When the tape reader receives the Go command, it will start the handshake data transfer to the 8255A. Let's work our way through the timing waveforms in Figure 9-10, p. 252, to see how the data transfer takes place.

The tape reader starts the process by sending out a byte of data to port A on its eight data lines. The tape reader then asserts its $\overline{STB}$ line low to tell the 8255A that a new byte of data has been sent. In response, the 8255A raises its Input Buffer Full (IBF) signal on PC5 high to tell the tape reader that it is ready for the data. When the tape reader detects the IBF signal at a high level, it raises its $\overline{STB}$ signal high again. The rising edge of the $\overline{STB}$ signal has two effects on the 8255A. It first latches the data byte in the input latches of the 8255A. Once the data is latched, the tape reader can remove the data byte in preparation for sending the next data byte. This is shown by the dashed section on the right side of the data waveform in Figure 9-10. Second, if the interrupt signal output has been enabled, the rising edge of the $\overline{STB}$ signal will cause the 8255A to output an Interrupt Request signal to the microprocessor on bit PC3.

The processor's response to the interrupt request will be to go to an interrupt service procedure which reads in the byte of data latched in port A. When the $\overline{RD}$ signal from the microprocessor goes low for this read of port A, the 8255A will automatically reset its Interrupt Request signal on PC3. This is done so that a second interrupt cannot be caused by the same data byte transfer. When the processor raises its $\overline{RD}$ signal high again at the end of the read operation, the 8255A automatically drops its IBF signal on PC5 low again. IBF going low again is the signal to the tape reader that the data transfer is complete and that it can send the next byte of data. The time between when the 8255A sends the Interrupt Request signal and when the processor reads the data byte from port A depends on when the processor gets around to servicing that interrupt. The point here is that this time doesn't matter. The tape reader will not

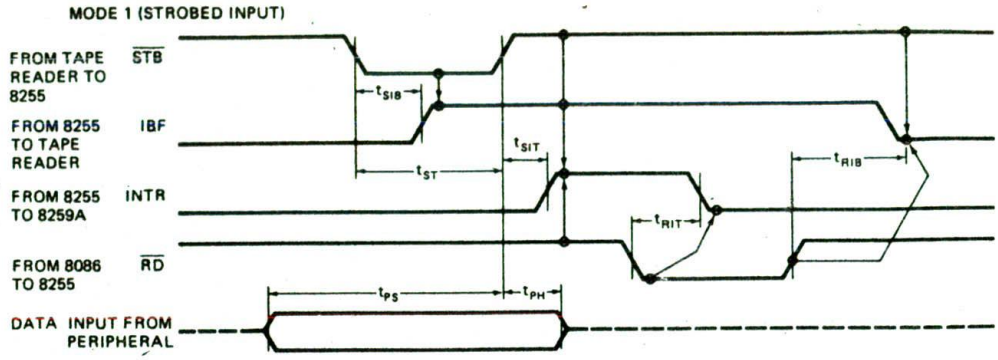


FIGURE 9-10 Timing waveforms for 8255 handshake data input from a tape reader.

send the next byte of data until it detects that the IBF signal has gone low again. The transfer cycle will then repeat for the next data byte.

After the processor reads in the lathe control instruction byte from the tape reader, it will decode this instruction, and output the appropriate control byte to the lathe on port B of the 8255A. The tape reader then sends the next instruction byte. If the instruction tape is made into a continuous loop, the lathe will keep making the specified 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 (0DH), 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 for 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 Figure 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 Figure 9-12, p. 254.

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.)

The rest of the pins on the 36-pin connector fall into two categories: signals sent to the printer to tell it what

SIGNAL PIN NO	RETURN PIN NO.	SIGNAL	DIRECTION	DESCRIPTION
1	19	$\overline{\text{STROBE}}$	IN	STROBE pulse to read data in. Pulse width must be more than 0.5 μs at receiving terminal. The signal level is normally "high"; read-in of data is performed at the "low" level of this signal.
2	20	DATA 1	IN	These signals represent information of the 1st to 8th bits of parallel data respectively. Each signal is at "high" level when data is logical "1" and "low" when logical "0."
3	21	DATA 2	IN	
4	22	DATA 3	IN	
5	23	DATA 4	IN	
6	24	DATA 5	IN	
7	25	DATA 6	IN	
8	26	DATA 7	IN	
9	27	DATA 8	IN	
10	28	$\overline{\text{ACKNLG}}$	OUT	Approximately 5 μs pulse; "low" indicates that data has been received and the printer is ready to accept other data.
11	29	BUSY	OUT	A "high" signal indicates that the printer cannot receive data. The signal becomes "high" in the following cases: 1. During data entry. 2. During printing operation. 3. In "offline" state. 4. During printer error status.
12	30	PE	OUT	A "high" signal indicates that the printer is out of paper.
13	–	SLCT	OUT	This signal indicates that the printer is in the selected state.
14	–	$\overline{\text{AUTO FEED XT}}$	IN	With this signal being at "low" level, the paper is automatically fed one line after printing. (The signal level can be fixed to "low" with DIP SW pin 2-3 provided on the control circuit board.)
15	–	NC		Not used.
16	–	OV		Logic GND level.
17	–	CHASIS-GND	–	Printer chasis GND. In the printer, the chassis GND and the logic GND are isolated from each other.
18	–	NC	–	Not used.
19-30	–	GND	–	"Twisted-Pair Return" signal; GND level.
31	–	$\overline{\text{INIT}}$	IN	When the level of this signal becomes "low" the printer controller is reset to its initial state and the print buffer is cleared. This signal is normally at "high" level, and its pulse width must be more than 50 μs at the receiving terminal.
32		$\overline{\text{ERROR}}$	OUT	The level of this signal becomes "low" when the printer is in "Paper End" state, "Offline" state and "Error" state.
33	–	GND	–	Same as with pin numbers 19 to 30.
34	–	NC	–	Not used.
35				Pulled up to +5 Vdc through 4.7 k-ohms resistance.
36	–	$\overline{\text{SLCT IN}}$	IN	Data entry to the printer is possible only when the level of this signal is "low." (Internal fixing can be carried out with DIP SW 1-8. The condition at the time of shipment is set "low" for this signal.)

Notes:
1. "Direction" refers to the direction of signal flow as viewed from the printer.
2. "Return" denotes "Twisted-Pair Return" and is to be connected at signal-ground level.
   When wiring the interface, be sure to use a twisted-pair cable for each signal and never fail to complete connection on the return side. To prevent noise effectively, these cables should be shielded and connected to the chassis of the system unit.
3. All interface conditions are based on TTL level. Both the rise and fall times of each signal must be less than 0.2 μs.
4. Data transfer must not be carried out by ignoring the $\overline{\text{ACKNLG}}$ or BUSY signal. (Data transfer to this printer can be carried out only after confirming the $\overline{\text{ACKNLG}}$ signal or when the level of the BUSY signal is "low.")

FIGURE 9-11 Pin connections and descriptions for Centronics-type parallel interface to IBM PC and EPSON FX-100 printer. (*IBM Corporation*)

operation to do, and signals from the printer that indicate its status. The major control signals to the printer are $\overline{\text{INIT}}$ on pin 31, which tells the printer to perform its internal initialization sequence, and $\overline{\text{STROBE}}$ on pin 1, which tells the printer, "Here is a character for you." Two additional input pins, pin 14 and pin 36, are usually taken care of inside the printer.

The major status signals output from the printer are

1. The $\overline{\text{ACKNLG}}$ signal on pin 10, which, when low, indicates that the data character has been accepted and the printer is ready for the next character.
2. The BUSY signal on pin 11, which is high if, for some reason such as being out of paper, the printer is not ready to receive a character.
3. The PE signal on pin 12, which goes high if the out-of-paper switch in the printer is activated.
4. The SLCT signal on pin 13, which goes high if the printer is selected for receiving data.
5. The $\overline{\text{ERROR}}$ signal on pin 32, which goes low for a variety of problem conditions in the printer.

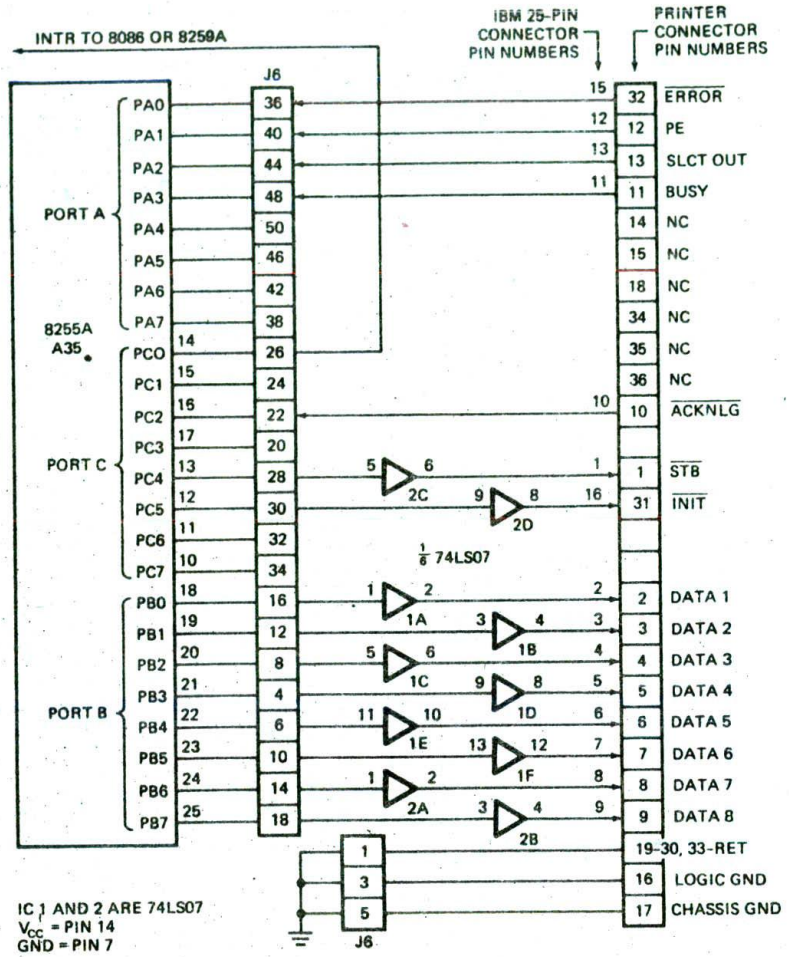


FIGURE 9-12 Circuit for interfacing Centronics-type parallel input printer to 8255A on SDK-86 board.

Figure 9-13 shows the timing waveforms for transferring data characters to an IBM printer using the basic handshake signals. Here's how this works.

Assuming the printer has been initialized, the BUSY signal is checked to see if the printer is ready to receive data. If this signal is low, indicating the printer is ready (not busy), an ASCII code is sent out on the eight parallel data lines. After at least 0.5 μs, the $\overline{\text{STROBE}}$ signal is asserted low to tell the printer that a character has been sent. The $\overline{\text{STROBE}}$ signal going low causes the printer to assert its BUSY signal high. After a minimum time of 0.5 μs, the $\overline{\text{STROBE}}$ signal can be raised high again. Note that the data must be held valid on the data lines for at least 0.5 μs after the $\overline{\text{STROBE}}$ signal is made high.

When the printer is ready to receive the next character, it asserts its $\overline{\text{ACKNLG}}$ signal low for about 5 μs. The rising edge of the $\overline{\text{ACKNLG}}$ signal tells the microcomputer that it can send the next character. At the same time as the rising edge of the $\overline{\text{ACKNLG}}$ signal, the printer also resets the BUSY signal. A low on BUSY is another indication that the printer is ready to accept the next character. Some systems use the $\overline{\text{ACKNLG}}$ signal for the handshake, and some systems use the BUSY signal. Now let's see how you can do this handshake printer interface with an 8255A.

8255A CONNECTIONS AND INITIALIZATION

For this example, we disconnected our printer cable from the printer output and connected it to an 8255A on an SDK-86 board, as shown in Figure 9-12. The 74LS07 open-collector buffers are used on the signal and data lines from the 8255A because the 8255A outputs do not have enough current drive to charge and discharge the capacitance of the connecting cable fast enough. Pull-up resistors for the open-collector outputs of the 74LS07s are built into the printer.

Port B of the 8255A is used for the handshake output data lines. Therefore, as shown in Figure 9-4, bit PC0 functions as the interrupt request output to the 8086.

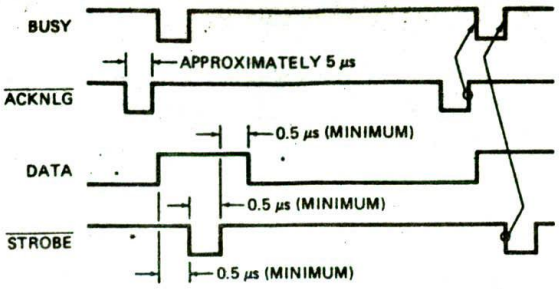


FIGURE 9-13 Timing waveforms for transfer of a data character to a Centronics-type parallel printer such as the IBM-PC or Epson printer. (*IBM Corporation*)

The $\overline{\text{ACKNLG}}$ signal from the printer is connected to the 8255A $\overline{\text{ACK}}$ input on bit PC2. The $\overline{\text{OBF}}$ signal on PC1 of the 8255A would normally be used as the strobe signal for this type of handshake data transfer. Unfortunately, however, it does not have the right timing parameters for this handshake, so it is left unconnected. Therefore, the $\overline{\text{STROBE}}$ input of the printer is connected to bit PC4. The $\overline{\text{STROBE}}$ signal will be generated by a bit set/reset of this pin.

The four printer status signals are connected to port A so the program can read them in to determine the condition of the printer.

Finally, the $\overline{\text{INIT}}$ input of the printer is connected to bit PC5 so that the printer can be initialized under program control.

Now, while the hardware configuration is fresh in your mind, let's look at the control words we have to send to the 8255A for this application.

Figure 9-14*a* shows the mode control word to initialize port B for mode 1 output, port A for mode 0 input, and the upper 4 bits of port C as outputs. Figure 9-14*b* 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 Figure 7-16, port P1A—FFF9H; port P1B—FFFBH; port P1C—FFFDH; and control P1—FFFFH. For that system, then, both control words are output to FFFFH.

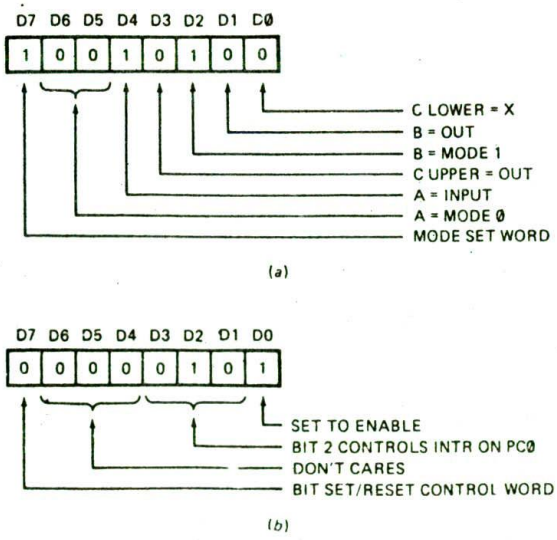


FIGURE 9-14 8255A control words for printer interface. (*a*) Mode control word. (*b*) Bit set/reset control word.

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 Figure 8-14. The higher-priority interrupt inputs on the 8255A are left for a clock interrupt and a keyboard interrupt.

Figure 9-15*a*, p. 256, 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

```
MAINLINE ALGORITHM FOR PRINTER DRIVER

INITIALIZATION
  Set up control block
     Word for storing pointer to ASCII string
     Word for number of charaters in string
  Initialization control words to 8259A
  Unmask 8259A IR6 and any other IR inputs used
  Mode set word to 8255A
  Unmask 8086 INTR input
  Send STROBE high to printer
  Initialize printer (pulse INIT low)
TO SEND ASCII STRING
  Read printer status from port
  IF error THEN
        send message
        exit, terminate program
  Set print done status bit
  Load starting address of string into pointer store
  Load length of string into character counter
  Enable 8255A INTR output
  Wait for interrupt
```

(a)

```
PRINTER DRIVER PROCEDURE ALGORITHM

Save registers
Enable 8086 INTR for higher priority interrupts
Get pointer to string
Get ASCII character from buffer
Send character to printer
Wait 5 μs
Send STROBE low
Wait 5 μs
Send STROBE high
Increment pointer to string
Decrement character counter
IF character count = 0 THEN
   Disable 8255A interrupt request output
Send EOI command to 8259A
Restore registers
Return from interrupt procedure
```

(b)

FIGURE 9-15 Algorithm for printer mainline and interrupt-based printer driver procedure. (a) Mainline steps. (b) Printer driver procedure steps.

mode control word shown in Figure 9-14*a*. A bit set/reset control word is then sent to the 8255A to make the $\overline{\text{STROBE}}$ 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 $\overline{\text{INIT}}$ 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 Figure 9-16*a*. 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 9-15*b* 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 block and used to read a character in from the memory buffer to AL. The character in AL is then output to port B of the 8255A.

From here on, the program follows the timing diagram in Figure 9-13. After sending the character, the program waits at least 0.5 μs, asserts the $\overline{\text{STROBE}}$ input low, waits at least another 0.5 μs, and raises the $\overline{\text{STROBE}}$ line high again. As we said before, the strobe signal must be generated with program instructions because the hardware strobe signal generated by the 8255A does not have the correct timing for this handshake. The data hold parameter in the timing diagram is satisfied because the data byte will be latched on the port B output pins until the next character is sent. Sending of the character is now complete, so the next step is to get ready to send another character.

To do this, the buffer pointer in the control block is incremented by 1, and the character counter in the control block is decremented by 1. If the character counter is not down to 0, there are more characters to send, so the EOI command is sent to the 8259A, the registers are popped off the stack, and execution is returned to the mainline to wait for the next interrupt. If the character counter in the control block is down to 0, all the characters have been sent, so the Interrupt Request output of the 8255A is disabled with a bit set/reset control word. This prevents further interrupt requests from the 8255A until we enable it again to send another buffer of characters to the printer. Work your way through Figure 9-16*b* to see how this algorithm is easily implemented. One part of the program that we do want to expand and clarify is the generation of the $\overline{\text{STROBE}}$ signal with bit PC3.

We could use external hardware to "massage" the $\overline{\text{OBF}}$ signal from the 8255A so it matches the timing and polarity requirements of the receiving device. However,

```
                                ;8086 MAINLINE PROGRAM F9-16A.ASM
                                ;ABSTRACT  : Printer-driver mainline initializes the 8259A and the 8255A
                                           ; on an SDK-86 board so that a message in a buffer can be sent
                                           ; to a printer. It also sets up a control block and initializes
                                           ; all variables used
                                ;REGISTERS : Uses CS,DS,SS,SP,AX,DX,CX,
                                ;PORTS     : SDK-86 port P1A (FFF9H) - used to input status of printer
                                           ;        port P1B (FFFBH) - used to output a character
                                           ;        port P1C used for handshake signals for port B
                                ;PROCEDURES: Uses PRINT_IT used to output characters

0000                            A_INT_TABLE     SEGMENT     WORD
0000  0C*(0000)                     TYPE_64_69 DW  12  DUP(0)     ; Reserved for IR0-IR5
0018  02*(0000)                     TYPE_70    DW   2  DUP(0)     ; IR6 interrupt
001C  02*(0000)                     TYPE_71    DW   2  DUP(0)     ; IR7 interrupt - not used
0020                            A_INT_TABLE     ENDS

0000                            DATA SEGMENT WORD PUBLIC
0000  54 68 69 73 20 69 73 +        MESSAGE_1     DB  'This is the message from the printer driver!'
      20 74 68 65 20 6D 65 +
      73 73 61 67 65 20 66 +
      72 6F 6D 20 74 68 65 +
      20 70 72 69 6E 74 65 +
      72 20 64 72 69 76 65 +
      72 21
002C  0D 0A 0D                                 DB  0DH, 0AH, 0DH     ; Return & line-feed for printer
      = 002F                        MESSAGE_LENGTH EQU ($-MESSAGE_1)    ; Compute length of message
002F  00                            PRINT_DONE    DB  0
0030  0000                          POINTER       DW  00                ; Storage for pointer to MESSAGE_1
0032  00                            COUNTER       DB  0                 ; Counter for length of MESSAGE_1
0033  00                            PRINTER_ERROR DB  0
0034                            DATA ENDS

                                PUBLIC   PRINT_DONE, POINTER, COUNTER, MESSAGE_1
                                EXTRN    PRINT_IT:FAR

0000                            STACK_SEG  SEGMENT
0000  1E*(0000)                          DW     30 DUP(0)
                                     STACK_TOP LABEL  WORD
003C                            STACK_SEG  ENDS

0000                            CODE SEGMENT WORD PUBLIC
                                  ASSUME CS:CODE, DS:A_INT_TABLE, SS:STACK_SEG
                                ;Initialize stack and data segment registers
0000  B8 0000s                          MOV  AX, STACK_SEG              ; Initialize stack
0003  8E D0                             MOV  SS, AX                     ; segment register
0005  BC 003Cr                          MOV  SP, OFFSET STACK_TOP       ; Initialize top of stack
0008  B8 0000s                          MOV  AX, A_INT_TABLE            ; Initialize data
000B  8E D8                             MOV  DS, AX                     ; segment register
                                ;Set up interrupt table and put in address for printer interrupt subroutine
000D  C7 06 001Ar 0000s                 MOV  TYPE_70+2, SEG PRINT_IT
0013  C7 06 0018r 0000e                 MOV  TYPE_70,   OFFSET PRINT_IT
                                ;Initialize data segment register
                                  ASSUME DS:DATA
0019  B8 0000s                          MOV  AX, DATA
001C  8E D8                             MOV  DS, AX
                                ;Initialize 8259A and unmask IR6
001E  BA FF00                           MOV  DX, 0FF00H                 ; Point at 8259A control address
0021  B0 13                             MOV  AL, 00010011B              ; ICW1, edge triggered, single, 8086
0023  EE                                OUT  DX, AL                     ; Send ICW1
0024  BA FF02                           MOV  DX, 0FF02H                 ; Point at ICW2 address
0027  B0 40                             MOV  AL, 01000000B              ; Type 64 is first 8259A type
0029  EE                                OUT  DX, AL                     ; Send ICW2
002A  B0 01                             MOV  AL, 00000001B              ; ICW4, 8086 mode
002C  EE                                OUT  DX, AL                     ; Send ICW4
002D  B0 BF                             MOV  AL, 10111111B              ; OCW1 to unmask IR6
002F  EE                                OUT  DX, AL                     ; Send OCW1
                                ;Initialize 8255A, P1A-mode1 input. P1B-mode0 output. Unused P1C bits-output
0030  BA FFFF                           MOV  DX, 0FFFFH                 ; Control address for 8255A
0033  B0 94                             MOV  AL, 10010100B              ; Control word for above conditions
0035  EE                                OUT  DX, AL                     ; Send control word
0036  FB                                STI                             ; Unmask 8086 INTR interrupt
```

FIGURE 9-16 8086 assembly language program for driver. (a) Mainline.

(Continued)

```
73
74 0037  B0 09                    ;Send strobe high to printer with bit set on PC4
75 0039  EE                              MOV  AL, 00001001B
                                         OUT  DX, AL
76                               ;Initialize printer-pulse INIT low for > 50 useconds (on PC5)
77 003A  B0 00                           MOV  AL, 00001101B              ; Bit set on PC5
78 003C  EE                              OUT  DX, AL                     ; Send INIT high
79 003D  B0 0C                           MOV  AL, 00001100B              ; Bit reset on PC5
80 003F  EE                              OUT  DX, AL                     ; Send INIT low
81 0040  B9 0017                         MOV  CX, 17H                    ; Wait > 50 useconds
82 0043  E2 FE                   PAUSE1: LOOP PAUSE1
83 0045  B0 00                           MOV  AL, 00001101B              ; Bit set on PC5
84 0047  EE                              OUT  DX, AL                     ; Send INIT high again
85                               ;Read printer status from port A, status OK - AL = XXXX0101
86                               ;PA3-BUSY=0, PA2-SLCT=1, PA1-PE=0, PA0-ERROR=1
87 0048  C6 06 0033r 00                 MOV  PRINTER_ERROR, 0           ; Printer OK so far
88 004D  BA FFF9                         MOV  DX, 0FFF9H                 ; Point at port A
89 0050  EC                              IN   AL, DX                     ; Get printer status
90 0051  24 0F                           AND  AL, 0FH                    ; Upper 4 bits not used
91 0053  3C 05                           CMP  AL, 00000101B              ; If status OK then
92 0055  74 14                           JZ   SEND_IT                    ; send it
93                               ;else printer not ready, wait 20 ms and try again
94 0057  B9 16EA                         MOV  CX, 16EAH                  ; Load count for 20 ms
95 005A  E2 FE                   PAUSE:  LOOP PAUSE                      ; and wait
96 005C  EC                              IN   AL, DX                     ; Repeat steps to read status
97 005D  24 0F                           AND  AL, 0FH
98 005F  3C 05                           CMP  AL, 00000101B
99 0061  74 08                           JZ   SEND_IT                    ; If printer not ready then
100 0063 C6 06 0033r 01                  MOV  PRINTER_ERROR, 01          ; set error code and
101 0068 EB 19 90                        JMP  FIN                        ; terminate program
102                              ;else set up pointer to message storage and say print not done yet
103 006B B8 0000r                SEND_IT:MOV  AX, OFFSET MESSAGE_1
104 006E A3 0030r                        MOV  POINTER, AX
105 0071 C6 06 002Fr 00                  MOV  PRINT_DONE, 00
106 0076 C6 06 0032r 2F                  MOV  COUNTER, MESSAGE_LENGTH
107                              ;Enable 8255A interrupt request output on PC0 by setting PC2
108 007B BA FFFF                         MOV  DX, 0FFFFH                 ; Point at port control addr
109 007E B0 05                           MOV  AL, 00000101B              ; Bit set word for PC0 intr
110 0080 EE                              OUT  DX, AL
111                              ;Wait for an interrupt from the printer
112 0081 EB FE                   WT:     JMP  WT
113 0083 90                      FIN:    NOP
114 0084                         CODE    ENDS
115                                      END
```

(a)

FIGURE 9-16 *(Continued)* (a) Mainline.

here we generate the strobe directly under software control.

In the mainline we make the $\overline{\text{STROBE}}$ signal on PC4 high by sending a bit set/reset control word of 00001001 to the control register of the 8255A. In the printer driver procedure a character is sent to the printer with the OUT DX,AL instruction. According to the timing diagram in Figure 9-13, we then want to wait at least 0.5 μs before asserting the $\overline{\text{STROBE}}$ signal low. This is automatically done in the program because the instructions required to assert the strobe low take longer than 0.5 μs. The MOV AL,00001000B instruction requires 4 clock cycles, and the OUT DX,AL instruction requires 8 clock cycles to execute. Assuming a 5-MHz clock (0.2-μs period), these two instructions take 2.4 μs to execute, which is more than required.

Again referring to the timing diagram in Figure 9-13, the $\overline{\text{STROBE}}$ time low must also be at least 0.5 μs. The MOV AL,00001001B instruction takes 4 clock cycles, and the OUT DX,AL instruction takes 8 clock cycles. With a 5-MHz clock, this totals to 2.5 μs, which again is more than enough time for $\overline{\text{STROBE}}$ low. In this case, creating the $\overline{\text{STROBE}}$ signal with software does not use much of the processor's time, so this is an efficient way to do it.

A FEW MORE POINTS ABOUT THE 8255A

Before leaving our discussion of the 8255A, we want to show you a little more about how port C can be used.

Any bits of port C which are programmed as inputs can be read by simply doing a read from the port C address. You can then mask out any unwanted bits of the word read in. If port A and/or port B is programmed in a handshake mode, then some of the bits of a byte read in from port C represent *status information* about the handshake signals. Figure 9-17, p. 260, shows the meaning of the bits read from port C for port A and/or port B in mode 1. Here's how you read this diagram. If port B is initialized as a handshake (mode 1) input port, then bits D0, D1, and D2 read from port C represent the status of the port B handshake signals. Bit D2 will

```
1                                       ;8086 PROCEDURE F9-168.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 (FFFBH) 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 FFFB                                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, OFFFFH          ; 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, 0FF00H          ; 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  9D                                     POPF
49 0034  CF                             *       IRET
50 0035                                 PRINT_IT ENDP
51 0035                                 CODE     ENDS
52                                              END
```

(*b*)

FIGURE 9-16 *(Continued)* (*b*) Procedure.

be high if the port B interrupt request output has been enabled. Bit D2 is a copy of the level on the input buffer full (IBF) pin. Bit D3 is a copy of the interrupt request output, so it will be high if port B is requesting an interrupt.

In our previous application examples, we showed how to do handshake data transfer on an interrupt basis to make maximum use of the CPU time. However, in applications where the CPU has nothing else to do while waiting to, for example, read in the next character from some device, then you can save one interrupt input by reading data from the 8255A on a polled basis. To do this for a handshake input operation on port B, you simply 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 Figure 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.

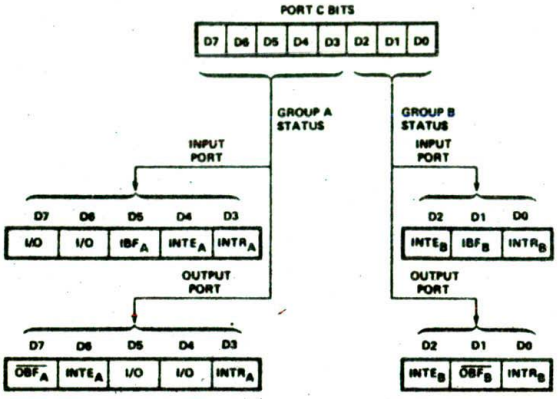


FIGURE 9-17 8255A status word format for mode 1 input and output operations.

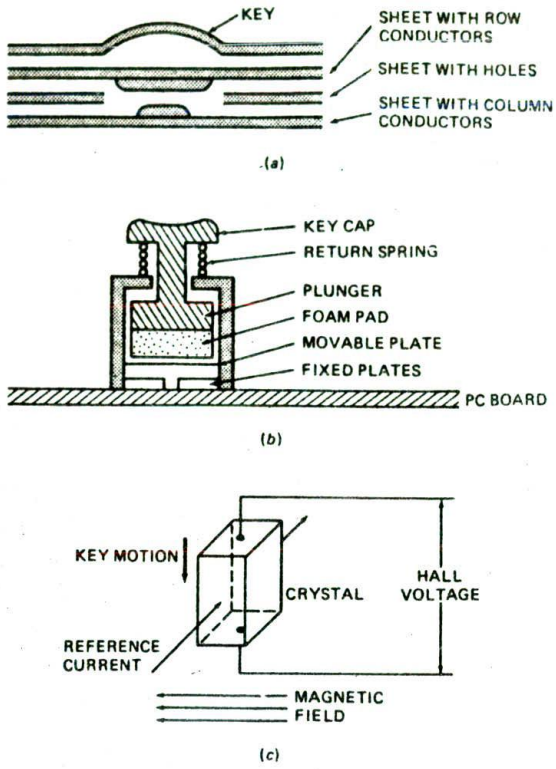


FIGURE 9-18 Keyswitch types. (*a*) Membrane. (*b*) Capacitive. (*c*) Hall effect.

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.

MECHANICAL KEYSWITCHES

In mechanical-switch keys, two pieces of metal are pushed together when you press the key. The actual switch elements are often made of a phosphor-bronze alloy with gold plating on the contact areas. The keyswitch usually contains a spring to return the key to the nonpressed position and perhaps a small piece of foam to help damp out bouncing. Some mechanical keyswitches now consist of a molded silicone dome with a small piece of conductive rubber on the underside. When a key is pressed, the rubber foam shorts two traces on the printed-circuit board to produce the Key Pressed signal.

Mechanical switches are relatively inexpensive, but they have several disadvantages. First, they suffer from *contact bounce*. A pressed key may make and break contact several times before it makes solid contact. Second, the contacts may become oxidized or dirty with age so they no longer make a dependable connection. Higher-quality mechanical switches typically have a rated lifetime of about 1 million keystrokes. The silicone dome type typically last 25 million keystrokes.

MEMBRANE KEYSWITCHES

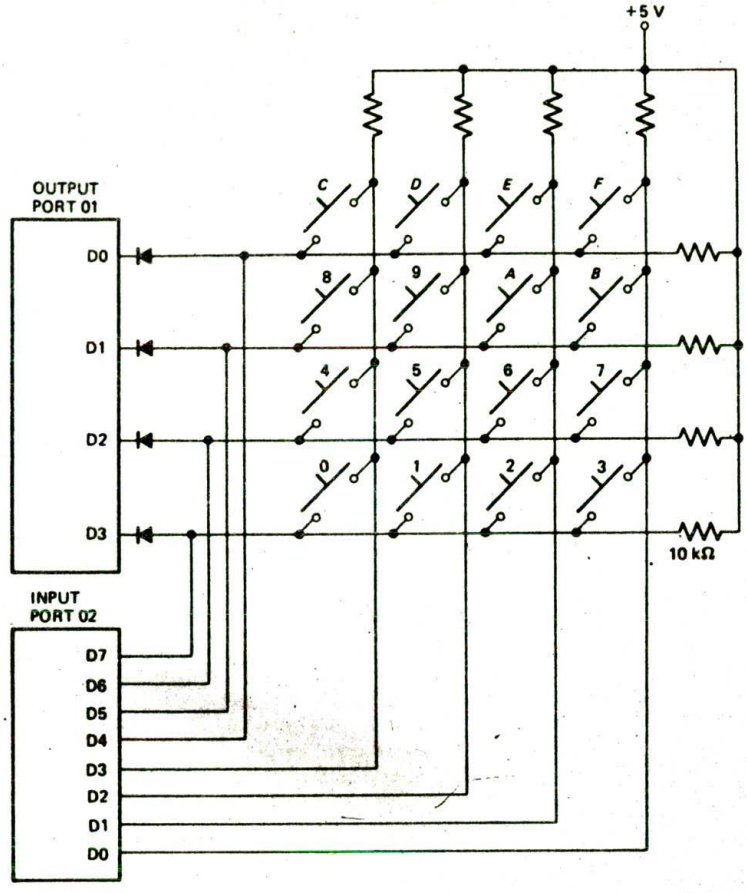
These switches are really just a special type of mechanical switch. They consist of a three-layer plastic or rubber sandwich, as shown in Figure 9-18*a*. The top layer has a conductive line of silver ink running under each row of keys. The middle layer has a hole under each key position. The bottom layer has a conductive line of silver ink running under each column of keys. When you press a key, you push the top ink line through the hole to contact the bottom ink line. The advantage of membrane keyboards is that they can be made as very thin, sealed units. They are often used on cash registers in fast-food restaurants, on medical instruments, and in other messy applications. The lifetime of membrane keyboards varies over a wide range.

CAPACITIVE KEYSWITCHES

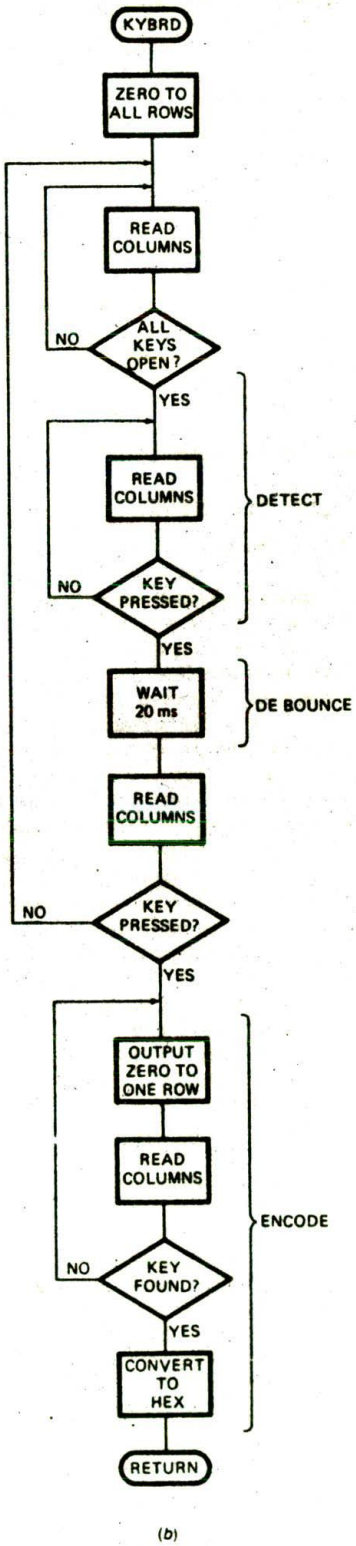
As shown in Figure 9-18*b*, a capacitive keyswitch has two small metal plates on the printed-circuit board and another metal plate on the bottom of a piece of foam. When you press the key, the movable plate is pushed closer to the fixed plate. This changes the capacitance between the fixed plates. Sense amplifier circuitry detects this change in capacitance and produces a logic-level signal that indicates a key has been pressed. The big advantage of a capacitive switch is that it has no mechanical contacts to become oxidized or dirty. A small disadvantage is the specialized circuitry needed to detect the change in capacitance. Capacitive keyswitches typically have a rated lifetime of about 20 million keystrokes.

HALL EFFECT KEYSWITCHES

This is another type of switch which has no mechanical contact. It takes advantage of the deflection of a moving



(a)



(b)

FIGURE 9-19 Detecting a matrix keyboard keypress, debouncing it, and encoding it with a microcomputer. (a) Port connections. (b) Flowchart for procedure.

charge by a magnetic field. Figure 9-18c shows you how this works. A reference current is passed through a semiconductor crystal between two opposing faces. When a key is pressed, the crystal is moved through a magnetic field which has its flux lines perpendicular to the direction of the current flow in the crystal. (Actually, it is easier to move a small magnet past the crystal.) Moving the crystal through the magnetic field causes a small voltage to be developed between two of the other opposing faces of the crystal. This voltage is amplified and used to indicate that a key has been pressed. (Hall effect sensors are also used to detect motion in many electrically controlled machines.) Hall effect keyboards are more expensive because of the more complex switch mechanisms, but they are very dependable and have typical rated lifetimes of 100 million or more keystrokes.

Keyboard Circuit Connections and Interfacing

In most keyboards, the keyswitches are connected in a matrix of rows and columns, as shown in Figure 9-19a.

We will use simple mechanical switches for our examples here, but the principle is the same for other types of switches. Getting meaningful data from a keyboard such as this requires the following three major tasks:

1. Detect a keypress.
2. Debounce the keypress.
3. Encode the keypress (produce a standard code for the pressed key).

The three tasks can be done with hardware, software, or a combination of the two, depending on the application. We will first show you how they can be done with software, as might be done in a microprocessor-based grocery scale where the microprocessor is not pressed for time. Later we describe some hardware devices which do these tasks.

Software Keyboard Interfacing

CIRCUIT CONNECTIONS AND ALGORITHM

Figure 9-19*a* shows how a hexadecimal keypad can be connected to a couple of microcomputer ports so the three interfacing tasks can be done as part of a program. The rows of the matrix are connected to four output-port lines. The column lines of the matrix are connected to four input-port lines. To make the program simpler, the row lines are also connected to four input lines.

When no keys are pressed, the column lines are held high by the pull-up resistors connected to +5 V. Pressing a key connects a row to a column. If a low is output on a row and a key in that row is pressed, then the low will appear on the column which contains that key and can be detected on the input port. If you know the row and the column of the pressed key, you then know which key was pressed, and you can convert this information into any code you want to represent that key. Figure 9-19*b* shows a flowchart for a procedure to detect, debounce, and produce the hex code for a pressed key. This procedure is another example of an I/O driver.

An easy way to detect if any key in the matrix is pressed is to output 0's to all the rows and then check the columns to see if a pressed key has connected a low to a column. In the algorithm in Figure 9-19*b*, we first output lows to all the rows and check the columns over and over until the columns are all high. This is done to make sure a previous key has been released before looking for the next one. In standard keyboard terminology, this is called *two-key lockout*. Once the columns are found to be all high, the program enters another loop, which waits until a low appears on one of the columns, indicating that a key has been pressed. This second loop does the detect task for us. A simple 20-ms delay procedure then does the debounce task.

After the debounce time, another check is made to see if the key is still pressed. If the columns are now all high, then no key is pressed and the initial detection was caused by a noise pulse or a light brushing past a key. If any of the columns are still low, then the assumption is made that it was a valid keypress.

The final task is to determine the row and column of the pressed key and convert this row and column information to the hex code for the pressed key. To get the row and column information, a low is output to one row and the columns are read. If none of the columns is low, the pressed key is not in that row, so the low is rotated to the next row and the columns are checked again. The process is repeated until a low on a row produces a low on one of the columns. The pressed key then is in the row which is low at that time. With the connections shown in Figure 9-19*a*, the byte read in from the input port will contain a 4-bit code which represents the row of the pressed key and a 4-bit code which represents the column of the pressed key. As we show later, a lookup table can be used to easily convert this row-column code to the desired hex value.

Figure 9-20 shows the assembly language program for this procedure. The detect, debounce, and row-detect parts of the program follow the flowchart very closely and should be easy for you to follow. Work your way down through these parts until you reach the ENCODE label; then continue with the discussion here.

CODE CONVERSION

There are two important ways of converting one code to another in a program. The ENCODE portion of this program uses a *compare* technique, which we will discuss in detail here. In a later section on keyboard interfacing with hardware, we will show you the other major code conversion technique, the *XLAT* method.

After the row which produces a low on one of the columns is found, execution jumps to the label ENCODE. The IN AL,DX instruction here reads the row and column codes from the input port. Since this 8-bit code read in represents the pressed key, all that has to be done now is to convert this 8-bit code to the hex code for that key. If we press the D key, for example, we want to exit from the procedure with 0DH in AL.

The conversion is done with the lookup table declared with DBs at the top of Figure 9-20. This table contains the 8-bit keypressed codes for each of the 16 keys. Note that the row-column codes are put in the table in the same order as the hex codes they represent. To convert a row-column code read in from the port, we compare it with each value in the table until we reach the value it matches. For several reasons, we start by comparing a row-column code with the highest entry in the table. A counter is used to keep track of how far down the table we have to go to find a match for a particular input code. Because the entries in the table are in numerical order, the counter will contain the hex code for the pressed key when a match is found. Let's look at the actual program instructions in Figure 9-20 to help you see how this works.

The BX register is used as a counter and as a pointer to one of the codes in the table, so to start we load 000FH in BX. The CMP AL,TABLE[BX] after this compares the code at offset [BX] in the table with the row-column code in AL. Initially, BX contains 000FH, so the row-column code in AL is compared with the row-column code at the highest location in the table. As shown in the data

```
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 (FFFBH) - 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 7B 7D 7E B7 BB BD +         TABLE DB    77H,  7BH,  7DH,  7EH, 0B7H, 0BBH, 0BDH, 0BEH
12       BE
13                                      ;       8     9     A     B     C     D     E     F
14 0008  D7 DB DD DE E7 EB ED +             DB  0D7H, 0DBH, 0DDH, 0DEH, 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                       START: MOV  AX, STACK_SEG        ; Initialize stack
25 0003  8E D0                                 MOV  SS, AX               ; segment register and
26 0005  BC 003Cr                              MOV  SP, OFFSET TOP_STACK ; top of stack
27 0008  B8 0000s                              MOV  AX, DATA
28 000B  8E D8                                 MOV  DS, AX
29                                      ;Initialize ports, mode 0, Port A for output, Ports B & C for input
30 000D  BA FFFF                               MOV  DX, 0FFFFH           ; Put port control address in DX
31 0010  B0 88                                 MOV  AL, 10001011B        ; Code 88H
32 0012  EE                                    OUT  DX, AL               ; Send control word.
33 0013  E8 0001                               CALL KEYBRD
34 0016  90                                    NOP
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 (FFFBH) 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, 0FFFBH           ; 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
```

FIGURE 9-20 Assembly language instructions for keyboard detect, debounce, and encode procedure.

(Continued)

```
 72 0038  24 0F                    AND   AL, 0FH
 73 003A  3C 0F                    CMP   AL, 0FH
 74 003C  74 ED                    JE    WAIT_PRESS
 75                          ;Find the key
 76 003E  B0 FE                    MOV   AL, 0FEH          ; Initialize a row mask with bit 0
 77 0040  8A C8                    MOV   CL, AL            ; low and save the mask
 78 0042  BA FFF9          NEXT_ROW: MOV   DX, 0FFF9H      ; Send out a low on one row
 79 0045  EE                       OUT   DX, AL
 80 0046  BA FFFB                  MOV   DX, 0FFFBH        ; Read columns & check for low
 81 0049  EC                       IN    AL, DX
 82 004A  24 0F                    AND   AL, 0FH           ; Mask out row code
 83 004C  3C 0F                    CMP   AL, 0FH           ; If low in a column then
 84 004E  75 06                    JNE   ENCODE            ; key column found, so encode it
 85 0050  D0 C1                    ROL   CL, 01            ; else rotate mask
 86 0052  8A C1                    MOV   AL, CL
 87 0054  EB EC                    JMP   NEXT_ROW          ; and look at next row
 88                          ;Encode the row/column information
 89 0056  BB 000F          ENCODE:  MOV   BX, 000FH        ; Set up BX as a counter and
 90 0059  EC                        IN    AL, DX           ; read row and column from port
 91 005A  3A 87 0000r      TRY_NEXT: CMP   AL, TABLE[BX]   ; Compare row/col code with table entry
 92 005E  74 08                     JE    DONE             ; Hex code in BX
 93 0060  4B                        DEC   BX               ; Point at next table entry
 94 0061  79 F7                     JNS   TRY_NEXT
 95 0063  B4 01                     MOV   AH, 01           ; Pass an error code in AH
 96 0065  EB 05 90                  JMP   EXIT
 97 0068  8A C3            DONE:    MOV   AL, BL           ; Hex code for key in AL
 98 006A  B4 00                     MOV   AH, 00           ; Put key-valid code in AH
 99 006C  5A               EXIT:    POP   DX               ; Restore calling program
100 006D  59                        POP   CX               ; registers
101 006E  5B                        POP   BX
102 006F  9D                        POPF
103 0070  C3                        RET
104 0071                  KEYBRD    ENDP
105 0071                  CODE      ENDS
106                                 END
```

FIGURE 9-20 (*Continued*)

segment in Figure 9-20, the row-column code at this location in the table is the code for the F key. If the code in AL matches this code, we know the F key was pressed. BX contains 000FH, the hex code for this key. Since we need only the lower 8 bits of BX, the hex code in BL is copied to AL to pass it back to the calling program. AH is loaded with 00H to tell the calling program that this was a valid keypress, and a return is made to the calling program.

If the row-column code in AL doesn't match the table value on the first compare, we decrement BX to point to the code for the E key in the table and do another compare. If a match occurs this time, then we know that the E key was the key pressed and that the hex code for that key, 0EH, is in BL. If we don't get a match on this compare, we cycle through the loop until we get a match or until the row-column code for the pressed key has been compared with all the values in the table. As long as the value in BX is 0 or above after the DEC BX instruction, the Jump if Not Sign instruction, JNS TRY_NEXT, will cause execution to go back to the Compare instruction. If no match is found in the table, BX will decrement from 0 to FFFFH. Since the sign bit is a copy of the MSB of the result after the DEC instruction, the sign bit will then be set. Execution will fall through to an instruction which loads an error code of 01H in AH. We then return to the calling program. The calling program will check AH on return to determine if the contents of AL represent the code for a valid keypress.

ERROR TRAPPING

The concept of detecting some error condition such as "no match found" is called *error trapping*. Error trapping is a very important part of real programs. Even in this simple program, think what might happen with no error trap if two keys in the same row were pressed at exactly the same time and a column code with two lows in it was produced. This code would not match any of the row-column codes in the table, so after all the values in the table were checked, BX would be decremented from 0000H to FFFFH. On the next compare, AL would be compared with a value in memory at offset FFFFH. Since this location is not even in the table, the compare-decrement cycle would continue through 65,536 memory locations until, by chance, the value in a memory location matched the row-column code in AL. The contents of BL at that point would be passed back to the calling routine. The chances are 1 in 256 that this would be the correct value for one of the two pressed keys. Since these are not very good odds, you should put an error trap in a program wherever there is a chance for it to go off to "never-never land" in this way. An error/no-error "flag" can be passed back to the calling program in a register as shown, in a dedicated memory location, or on the stack.

Keyboard Interfacing with Hardware

The previous section described how you can connect a keyboard matrix to a couple of microprocessor ports

and perform the three interfacing tasks with program instructions. For systems where the CPU is too busy to be bothered doing these tasks in software, an external device is used to do them. One example of a MOS device which can do this is the General Instrument AY5-2376, which can be connected to the rows and columns of a keyboard switch matrix. The AY5-2376 independently detects a keypress by cycling a low down through the rows and checking the columns just as we did in software. When it finds a key pressed, it waits a debounce time. If the key is still pressed after the debounce time, the AY5-2376 produces the 8-bit code for the pressed key and sends it out to, for example, a microcomputer port on eight parallel lines. To let the microcomputer know that a valid ASCII code is on the data lines, the AY5-2376 outputs a strobe pulse. The microcomputer can detect this strobe pulse and read in the ASCII code on a polled basis, as we showed in Figure 4-20, or it can detect the strobe pulse on an interrupt basis, as we showed in Figure 8-9. With the interrupt method the microcomputer doesn't have to pay any attention to the keyboard until it receives an interrupt signal, so this method uses very little of the microcomputer's time.

The AY5-2376 has a feature called *two-key rollover.* This means that if two keys are pressed at nearly the same time, each key will be detected, debounced, and converted to ASCII. The ASCII code for the first key and a strobe signal for it will be sent out; then the ASCII code for the second key and a strobe signal for it will be sent out. Compare this with two-key lockout, which we described previously in our discussion of the software method of keyboard interfacing.

DEDICATED MICROPROCESSOR KEYBOARD ENCODERS

Most computers and computer terminals now use detached keyboards with built-in encoders. Instead of using a hardware encoder device such as the AY5-2376, these keyboards use a dedicated microprocessor. Figure 9-21 shows the encoder circuitry for the IBM PC capaci-

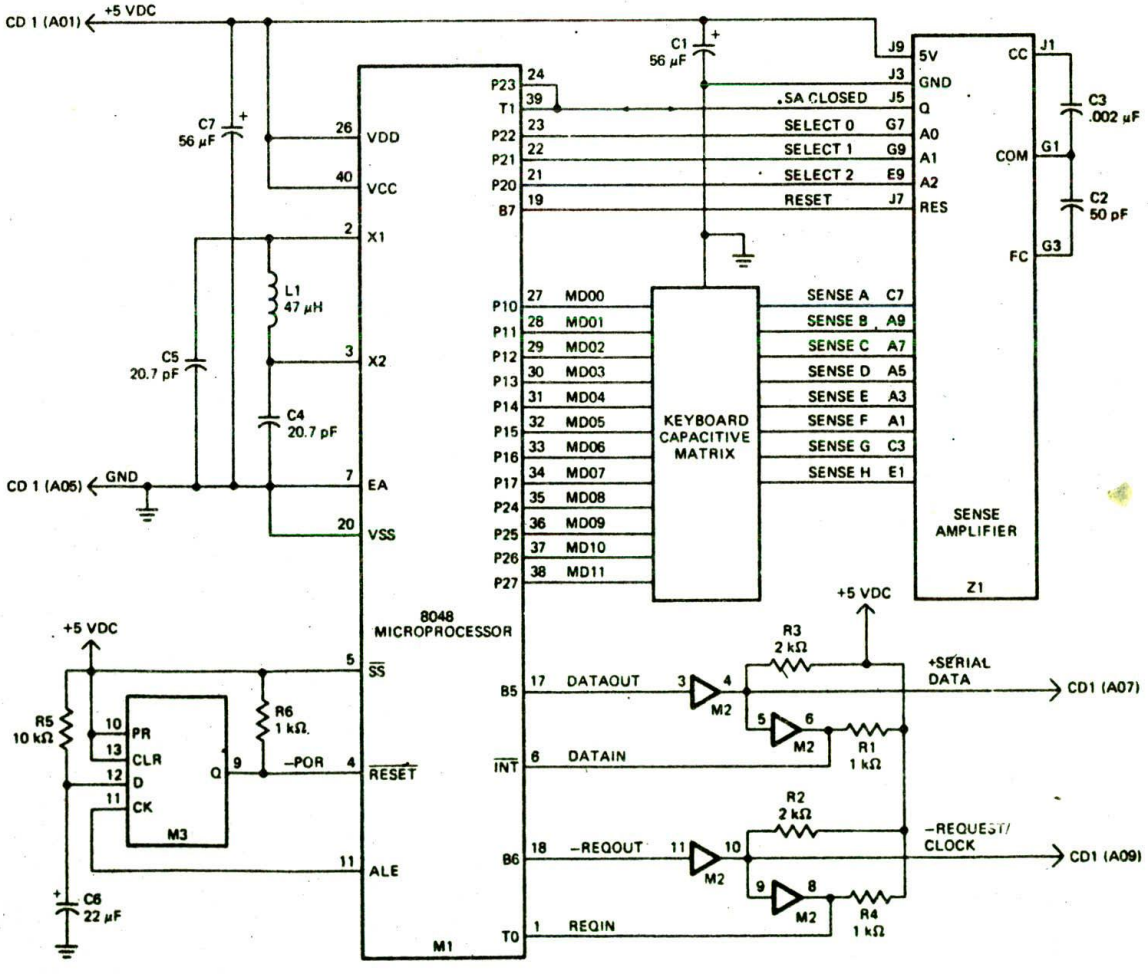


FIGURE 9-21 IBM PC keyboard scan circuitry using a dedicated microprocessor. (*IBM Corporation*)

tive-switch matrix keyboard. The 8048 microprocessor used here contains an 8-bit CPU, a ROM, some RAM, three ports, and a programmable timer/counter. A program stored in the on-chip ROM performs the three keyboard tasks and sends the code for a pressed key out to the computer. To cut down the number of connecting wires, the key code is sent out in serial form rather than in parallel form. Some keyboards send data to the computer in serial form using a beam of infrared light instead of a wire.

Note in Figure 9-21 that a sense amplifier is used to detect the change in capacitance produced when a key is pressed. Also note that the 8048 uses a tuned LC circuit rather than a more expensive crystal to determine its operating clock frequency.

One of the major advantages of using a dedicated microprocessor to do the three keyboard tasks is programmability. Special-function keys on the keyboard can be programmed to send out any code desired for a particular application. By simply plugging in an 8048 with a different lookup table in ROM, the keyboard can be changed from outputting ASCII characters to outputting some other character set.

The IBM keyboard, incidentally, does not send out ASCII codes, but instead sends out a hex "scan" code for each key when it is pressed and a different scan code when that key is released. This double-code approach gives the system software maximum flexibility because a program command can be implemented either when a key is pressed or when it is released.

CONVERTING ONE KEYBOARD CODE TO ANOTHER USING XLAT

Suppose that you are building up a simple microcomputer to control the heating, watering, lighting, and ventilation of your greenhouse. As part of the hardware, you buy a 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.

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 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 Figure 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 Figure 9-20.

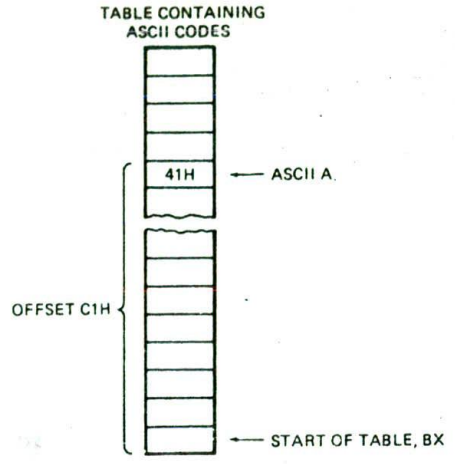


FIGURE 9-22 Memory table setup for using XLAT to convert EBCDIC keycode to ASCII equivalent.

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 of data needs to be displayed, simple digit-type displays are often used. There are several technologies used to make these digit-oriented displays, but we have space here to discuss only the two major types. These are *light-emitting diodes* (LEDs) and *liquid-crystal displays* (LCDs). LCD displays use very low power, so they are often used in portable, battery-powered instruments. LCDs, however, do not emit their own light; they simply change the reflection of available light. Therefore, for an instrument that is to be used in low-light conditions, you have to include a light source for the LCDs or use LEDs, which emit their own light. Starting with LEDs, the following sections show you how to interface these two types of displays to microcomputers.

Interfacing LED Displays to Microcomputers

Alphanumeric LED displays are available in three common formats. For displaying only numbers and hexadecimal letters, simple 7-segment displays such as that shown in Figure 1-4*a* are used.

To display numbers and the entire alphabet, 18-segment displays such as that shown in Figure 9-23*a* or 5 by 7 dot-matrix displays such as that shown in Figure 9-23*b* can be used. The 7-segment type is the least expensive, most commonly used, and easiest to interface with, so we will concentrate first on how to interface with this type. Later we will show the modifications needed to interface with the other types.

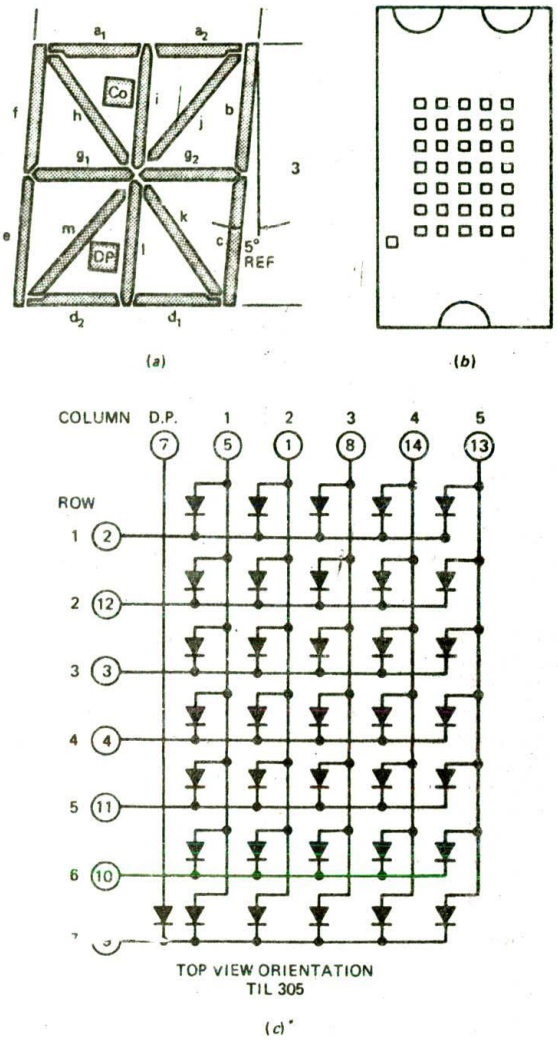


FIGURE 9-23 Eighteen-segment and 5 by 7 matrix LED displays. (*a*) 18-segment display. (*b*) 5 by 7 dot-matrix display format. (*c*) 5 by 7 dot-matrix circuit connections.

DIRECTLY DRIVING LED DISPLAYS

Figure 9-24, p. 268, shows a circuit that you might connect to a parallel port on a microcomputer to drive a single 7-segment, common-anode display. For a common-anode display, a segment is turned on by applying a logic low to it. The 7447 converts a BCD code applied to its inputs to the pattern of lows required to display the number represented by the BCD code. This circuit connection is referred to as a *static display* because current is being passed through the display at all times. Here's how you calculate the value of the current-limiting resistors that have to be connected in series with each segment.

Each segment requires a current of between 5 and 30 mA to light. Let's assume you want a current of 20 mA. The voltage drop across the LED when it is lit is about 1.5 V. The output low voltage for the 7447 is a maximum of 0.4 V at 40 mA, so assume that it is about 0.2 V at 20 mA. Subtracting these two voltage drops from the supply voltage of 5 V leaves 3.3 V across the current-limiting resistor. Dividing 3.3 V by 20 mA gives a value of 168 Ω for the current-limiting resistor. The voltage drops across the LED and the output of the 7447 are not exactly predictable, and the exact current through the LED is not critical as long as we don't exceed its maximum rating. Therefore, a standard value of 150 Ω is reasonable.

SOFTWARE-MULTIPLEXED LED DISPLAYS

The circuit in Figure 9-24 works well for driving just one or two LED digits with a parallel output port. However, this scheme has several problems if you want

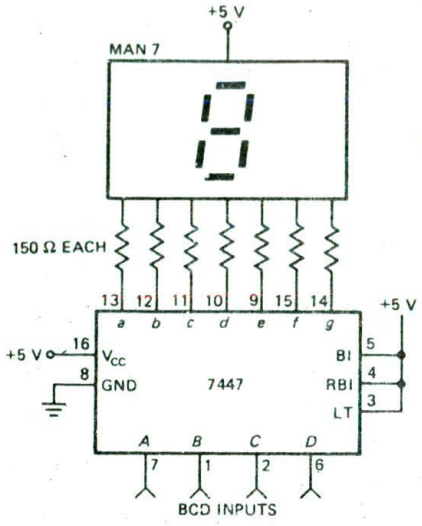


FIGURE 9-24 Circuit for driving single 7-segment LED display with 7447.

to drive, for example, eight digits. The first problem is power consumption. For worst-case calculations, assume that all 8 digits are displaying the digit 8, so all 7 segments are lit. Seven segments times 20 mA per segment gives a current of 140 mA per digit. Multiplying this by 8 digits gives a total current of 1120 mA, or 1.12 A, for the 8 digits! A second problem of the static approach is that each display digit requires a separate 7447 decoder, each of which uses, perhaps, another 13 mA. The current required by the decoders and the LED displays might be several times the current required by the rest of the circuitry in the instrument.

To solve the problems of the static display approach, we use a *multiplex* method. A circuit example is the easiest way to explain to you how this multiplexing works. Figure 9-25 shows a circuit you can add to a couple of microcomputer ports to drive some common-anode LED displays in a multiplexed manner. Note that the circuit has only one 7447 and that the segment outputs of the 7447 are bused in parallel to the segment inputs of all the digits. The question that may occur to you on first seeing this is: Aren't all the digits going to display the same number? The answer is that they would if all the digits were turned on at the same time. The trick of multiplexing displays is that only one display digit is turned on at a time. The PNP transistor in series with the common anode of each digit acts as an on/off switch for that digit. Here's how the multiplexing process works.

The BCD code for digit 1 is first output from port B to the 7447. The 7447 outputs the corresponding 7-segment code on the segment bus lines. The transistor connected to digit 1 is then turned on by outputting a low to the appropriate bit of port A. (Remember, a low turns on a PNP transistor.) All the rest of the bits of port A are made high to make sure no other digits are turned on. After 1 or 2 ms, digit 1 is turned off by outputting all highs to port A. The BCD code for digit 2 is then output to the 7447 on port B, and a word to turn on digit 2 is output on port A. After 1 or 2 ms, digit 2 is turned off and the process is repeated for digit 3. The process is continued until all the digits have had a turn. Then digit 1 and the following digits are lit again in turn. We leave it to you as an exercise at the end of the chapter to write a procedure which is called on an interrupt basis every 2 ms to keep these displays refreshed with some values stored in a table.

With 8 digits and 2 ms per digit, you get back to digit 1 every 16 ms, or about 60 times a second. This refresh rate is fast enough that, to your eye, the digits will each appear to be lit all the time. Refresh rates of 40 to 200 times a second are acceptable.

The immediately obvious advantages of multiplexing the displays are that only one 7447 is required, and only one digit is lit at a time. We usually increase the current per segment to between 40 and 60 mA for multiplexed displays so that they will appear as bright as they would if they were not multiplexed. Even with this increased segment current, multiplexing gives a large saving in power and parts.

> NOTE: If you are calculating the current-limiting resistors for multiplexed displays with increased segment current, check the data sheet for the displays you are using to make sure you are not exceeding their maximum current rating.

The software-multiplexed approach we have just described can also be used to drive 18-segment LED devices and dot-matrix LED devices. For these devices, however, you replace the 7447 in Figure 9-25 with a ROM which generates the required segment codes when the ASCII code for a character is applied to the address inputs of the ROM.

Display and Keyboard Interfacing with the 8279

A disadvantage of the software-multiplexing approach shown here is that it puts an additional burden on the CPU. Also, if the CPU gets involved in doing some lengthy task which cannot be interrupted to refresh the display, only one digit of the display will be left lit. An alternative approach to interfacing multiplexed displays to a microcomputer is to use a *dedicated display controller* such as the Intel 8279. As we show you in the next section, an 8279 independently keeps a bank of 7-segment displays refreshed and performs the three tasks for a matrix keyboard at the same time.

8279 CIRCUIT CONNECTIONS AND OPERATION OVERVIEW

Sheets 7 and 8 of the SDK-86 schematics in Figure 7-8 show the circuit connections for the keypad and the multiplexed 7-segment displays. First let's look at the display circuitry on sheet 8. The displays there are common-anode, and each digit has a PNP transistor switch between its anode and the +5-V supply. A logic low is required to turn on one of these switches. Note

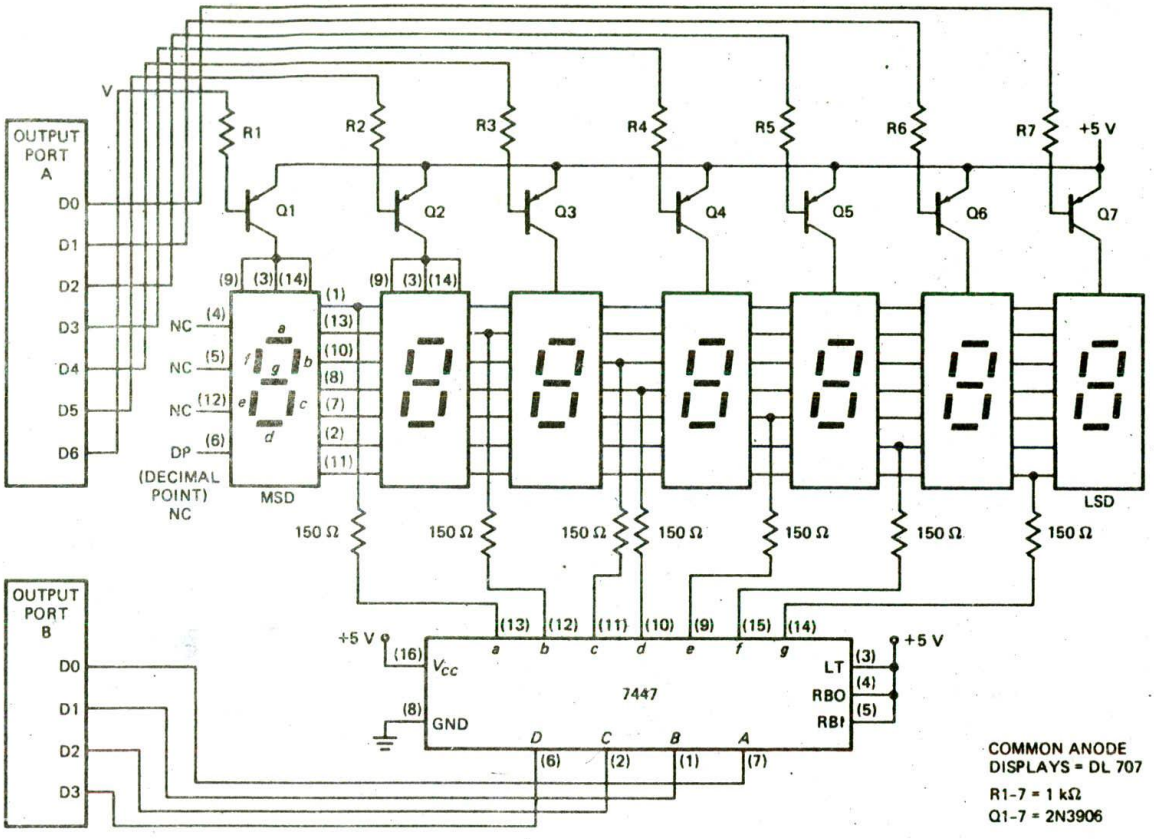


FIGURE 9-25 Circuit for multiplexing 7-segment displays with a microcomputer.

the 22-μF capacitor between +5 V and ground at the top of the schematic. This is necessary to filter out transients caused by switching the large currents to the LEDs off and on.

The segments of each digit are all connected on a common bus. Since these are common-anode displays, a low is needed to turn on a segment. Now let's look at sheet 7 in Figure 7-8 to see how these displays are driven.

The drive for the digit-switch transistors comes from a 7445 BCD-to-decimal decoder. This device is also known as a one-of-ten-low decoder. When a 4-bit BCD code is applied to the inputs of this device, the output corresponding to that BCD number will go low. For example, when the 8279 outputs 0100 or BCD 4, the 7445 output labeled 04 will go low. In the mode used for this circuit, the 8279 outputs a continuous count sequence from 0000 to 1111 over and over. This causes a low to be stepped from output to output of the 7445 in ring counter fashion, turning on each LED digit in turn. Only one output of the 7445 will ever be low at a time, so only one LED digit will be turned on at a time.

The segment bus lines for the displays are connected to the A3–A0 and B3–B0 outputs of the 8279 through some high-current inverting buffers in the ULN2003A. Note that the 22-Ω current-limiting resistors in series with the segment lines are much smaller in value than those we calculated for the static circuit in Figure 9-24. There are two reasons for this. First, there is an additional few tenths of a volt drop across the transistor switch on each anode. 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 Figure 9-26, p. 270, 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 (Figure 7-8, sheet 7) to turn on the first of the digit-driver transistors. The lines SL0 and SL1 in Figure 9-26 represent the SL0 and SL1 lines from the 8279. During this time, the 8279

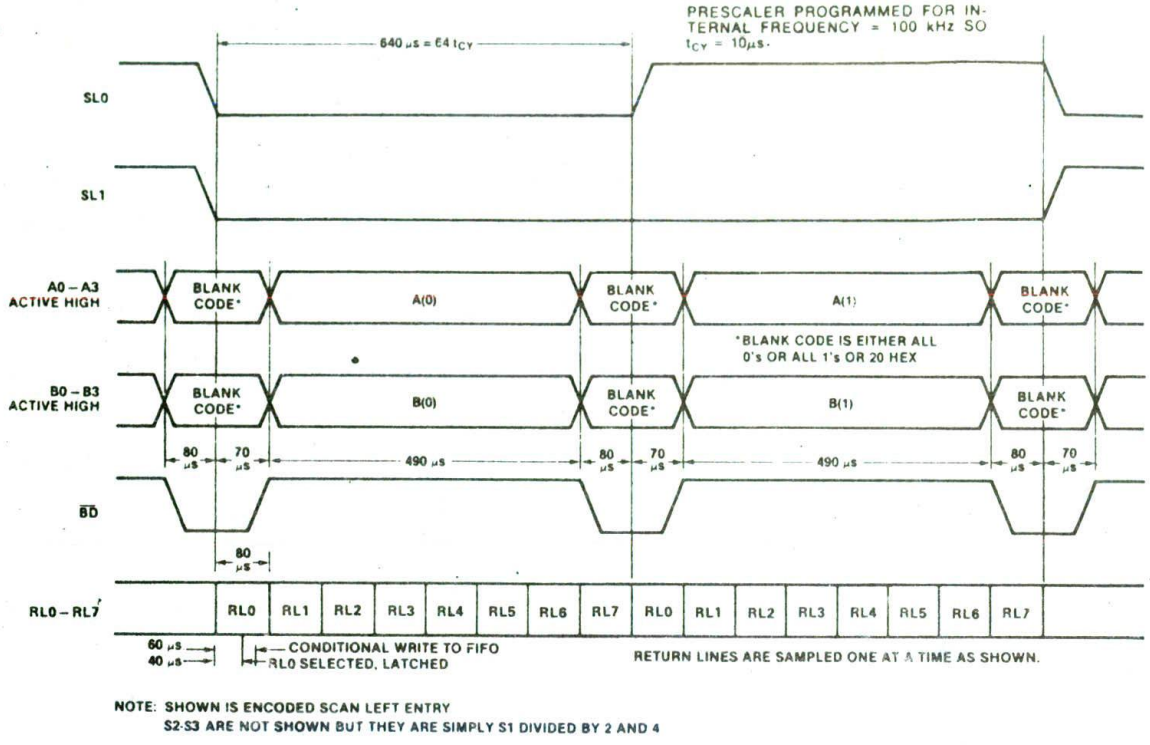


FIGURE 9-26 8279 display refresh timing and keyboard scan timing. (*Intel Corporation*)

outputs on the A3–A0 and B3–B0 segment lines a code which turns off all the segments. For the circuit in Figure 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 key because pressing a key shorts a row to a column. If no low is found on the columns, the low is stepped to the next row and the columns checked again. If a low is found on a column, then, after a debounce time, the column is checked again. If the keypress was valid, a compact code representing the key is constructed. Take a look at the circuit on sheet 7 of Figure 7-8 to see how an 8279 can be connected to do this.

When connected as shown in Figure 7-8, sheet 7, the 74LS156 functions as a one-of-eight-low decoder. In other words, if you apply 011, the binary code for 3, to its inputs, the 74LS156 will output a low on its 2Y3 output. Now remember from the discussion of 8279 display refreshing that the 8279 is outputting a continuous counting sequence from 0000 to 1111 on its SL0–SL3 lines. Applying this count sequence to the inputs

of the 74LS156 will cause it to step a low along its outputs. The 74LS156 then puts a low on one row of the keyboard at a time, as desired.

The column lines of the keyboard are connected to the return lines, RL0–RL7, of the 8279. As a low is put on each row by the scan-line counter and the 74LS156, the 8279 checks these return lines one at a time to see if any of them are low. The bottom line of the timing waveforms in Figure 9-26 shows when the return lines are checked. If the 8279 finds any of the return lines low, indicating a keypress, it waits a debounce time of about 10.3 ms and checks again. If the keypress is still present, the 8279 produces an 8-bit code which represents the pressed key. Figure 9-27 shows the format for the code produced. Three bits of this code represent the number of the row in which the 8279 found the pressed key, and another 3 bits represent the column of the pressed key. For interfacing to full typewriter keyboards the shift and control keys are connected to pins 36 and 37, respectively, of the 8279. The upper 2 bits of the code produced represent the status of these two keys.

After the 8279 produces the 8-bit code for the pressed key, it stores the byte in an internal 8-byte *FIFO* RAM. The term FIFO stands for first in, first out, which means that when you start reading codes from the FIFO, the first code you read out will be that for the first key pressed. The FIFO can store the codes for up to eight pressed keys before overflowing.

When the 8279 finds a valid keypress, it does two things to let you know about it. It asserts its interrupt request pin, IRQ, high, and it increments a FIFO count in an internal status register. You can connect the IRQ output to an interrupt input and detect when the FIFO has a character for you on an interrupt basis, or you can simply check the count in the status word to determine when the FIFO has a code ready to be read. The point here is that once the 8279 is initialized, you don't need to pay any attention to it until you want to send some new characters to be displayed, or until it notifies you that it has a valid keypressed code for you in its FIFO. Now that you have an overview of how the 8279 functions, we will show you how to initialize an 8279 to do all of these wondrous things and more.

INITIALIZING AND COMMUNICATING WITH AN 8279

As we have shown before, the first step in initializing a programmable device is to determine the system base address for the device, the internal addresses, and the system addresses for the internal parts. As an example here, we will use the 8279 on sheet 7 of the SDK-86 schematics in Figure 7-8. Figure 7-16*b* shows that the system base address for this device is FFE8H. The 8279 has only two internal addresses, which are selected by the logic level on its A0 input, pin 21. If the A0 input is low when the 8279 is selected, then the 8279 is enabled for reading data from it or writing data to it. A0 being high selects the internal control/status registers. For the circuit on sheet 7 of Figure 7-8, the A0 input is connected to system address line A1. Therefore, the data address for this 8279 is FFE8H and the control/status address is FFEAH.

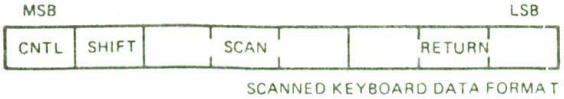


FIGURE 9-27 Format for data word produced by 8279 keyboard encoding.

After you have figured out the system addresses for a device, the next step is to look at the format for the control word(s) you have to send to the device to make it operate in the mode you want. Figure 9-28, p. 272, shows the format for the 8279 control words as they appear in the Intel data book. After you use up your 5-minute "freak-out" time, we will help you decipher these.

One question that may occur to you when you see all these control words is, If the 8279 only has one control register address, how am I going to send it all these different control words? The answer to this is that all the control words are sent to the same control register address, FFEAH for this example. The upper 3 bits of each control word tell the 8279 which control word is being sent. A pattern of 010 in the upper 3 bits of a control word, for example, identifies that control word as a Read FIFO/Sensor RAM control word. Keep Figure 9-28 handy as we discuss this and the other control words.

The first control word you send to initialize the 8279 is the *keyboard/display mode set* word. The bits labeled DD in the control word specify first of all whether you have 8 digits or 16 digits to refresh. If you have eight or fewer displays, make sure to initialize for 8 digits so the 8279 doesn't spend half its time refreshing nonexistent displays. The DD bits in this control word also specify the order in which the characters in the internal 16-byte display RAM will be sent out to the digits. In the left entry mode, the 7-segment code in the first address of the internal display RAM will be sent to the leftmost digit of the display. If you want to display the letters AbCd on the 4 leftmost digits of an 8-digit display, then you put the 7-segment codes for these letters in the first four locations of the display RAM, as shown in Figure 9-29*a*, p. 273. Codes put in higher addresses in the display RAM will be displayed on following digits to the right. In the right entry mode, the first code sent to the display RAM is put in the lowest address. This character will be displayed on the rightmost digit of the display. If a second character is written to the display RAM, it will be put in the second location in the RAM, as shown in Figure 9-29*b*. On the display, however, the new character will be displayed on the rightmost digit, and the previous character will be shifted over to the second position from the right. This is the way the displays of most calculators function as you enter numbers.

Now let's look at the KKK bits of the mode-set control word. The first choice you have to make here if you are using the 8279 with a keyboard is whether you want *encoded scan* or *decoded scan*. You know that for

Keyboard/Display Mode Set

	MSB						LSB
Code	0	0	0	D	D	K	K	K

Where DD is the Display Mode and KKK is the Keyboard Mode

DD

0 0 8 8-bit character display — Left entry

0 1 16 8-bit character display — Left entry*

1 0 8 8-bit character display — Right entry

1 1 16 8-bit character display — Right entry

For description of right and left entry, see Interface Considerations. Note that when decoded scan is set in keyboard mode, the display is reduced to 4 characters independent of display mode set.

KKK

0 0 0 Encoded Scan Keyboard — 2 Key Lockout*

0 0 1 Decoded Scan Keyboard — 2-Key Lockout

0 1 0 Encoded Scan Keyboard — N-Key Rollover

0 1 1 Decoded Scan Keyboard — N-Key Rollover

1 0 0 Encoded Scan Sensor Matrix

1 0 1 Decoded Scan Sensor Matrix

1 1 0 Strobed Input, Encoded Display Scan

1 1 1 Strobed Input, Decoded Display Scan

Program Clock

Code:	0	0	1	P	P	P	P	P

All timing and multiplexing signals for the 8279 are generated by an internal prescaler. This prescaler divides the external clock (pin 3) by a programmable integer. Bits PPPPP determine the value of this integer which ranges from 2 to 31. Choosing a divisor that yields 100 kHz will give the specified scan and debounce times. For instance, if Pin 3 of the 8279 is being clocked by a 2 MHz signal, PPPPP should be set to 10100 to divide the clock by 20 to yield the proper 100 kHz operating frequency.

Read FIFO/Sensor RAM

Code:	0	1	0	AI	X	A	A	A

X = Don't Care

The CPU sets up the 8279 for a read of the FIFO/Sensor RAM by first writing this command. In the Scan Keyboard Mode, the Auto-Increment flag (AI) and the RAM address bits (AAA) are irrelevant. The 8279 will automatically drive the data bus for each subsequent read ($A_0 = 0$) in the same sequence in which the data first entered the FIFO. All subsequent reads will be from the FIFO until another command is issued.

In the Sensor Matrix Mode, the RAM address bits AAA select one of the 8 rows of the Sensor RAM. If the AI flag is set (AI = 1), each successive read will be from the subsequent row of the sensor RAM.

Read Display RAM

Code:	0	1	1	AI	A	A	A	A

The CPU sets up the 8279 for a read of the Display RAM by first writing this command. The address bits AAAA select one of the 16 rows of the Display RAM. If the AI flag is set (AI = 1), this row address will be incremented after each following read *or write* to the Display RAM. Since the same counter is used for both reading and writing, this command sets the next read *or write* address and the sense of the Auto-Increment mode for both operations.

Write Display RAM

Code:	1	0	0	AI	A	A	A	A

The CPU sets up the 8279 for a write to the Display RAM by first writing this command. After writing the command with $A_0 = 1$, all subsequent writes with $A_0 = 0$ will be to the Display RAM. The addressing and Auto-Increment functions are identical to those for the Read Display RAM. However, this command does not affect the source of subsequent Data Reads; the CPU will read from whichever RAM (Display or FIFO/Sensor) which was last specified. If, indeed, the Display RAM was last specified, the Write Display RAM will, nevertheless, change the next Read location.

Display Write Inhibit/Blanking

					A	B	A	B
Code:	1	0	1	X	IW	IW	BL	BL

The IW Bits can be used to mask nibble A and nibble B in applications requiring separate 4-bit display ports. By setting the IW flag (IW = 1) for one of the ports, the port becomes marked so that entries to the Display RAM from the CPU do not affect that port. Thus, if each nibble is input to a BCD decoder, the CPU may write a digit to the Display RAM without affecting the other digit being displayed. It is important to note that bit B_0 corresponds to bit D_0 on the CPU bus, and that bit A_3 corresponds to bit D_7.

If the user wishes to blank the display, the BL flags are available for each nibble. The last Clear command issued determines the code to be used as a "blank." This code defaults to all zeros after a reset. Note that both BL flags must be set to blank a display formatted with a single 8-bit port.

Clear

Code:	1	1	0	C_D	C_D	C_D	C_F	C_A

The C_D bits are available in this command to clear all rows of the Display RAM to a selectable blanking code as follows:

C_D C_D C_D

0 X All Zeros (X = Don't Care)

1 0 AB = Hex 20 (0010 0000)

1 1 All Ones

Enable clear display when = 1 (or by C_A = 1)

During the time the Display RAM is being cleared (~160 μs), it may not be written to. The most significant bit of the FIFO status word is set during this time. When the Display RAM becomes available again, it automatically resets.

If the C_F bit is asserted (C_F = 1), the FIFO status is cleared and the interrupt output line is reset. Also, the Sensor RAM pointer is set to row 0.

C_A, the Clear All bit, has the combined effect of C_D and C_F; it uses the C_D clearing code on the Display RAM and also clears FIFO status. Furthermore, it resynchronizes the internal timing chain.

End Interrupt/Error Mode Set

Code	1	1	1	E	X	X	X	X

X = Don't care

For the sensor matrix modes this command lowers the IRQ line and enables further writing into RAM. (The IRQ line would have been raised upon the detection of a change in a sensor value. This would have also inhibited further writing into the RAM until reset.)

For the N-key rollover mode — if the E bit is programmed to "1" the chip will operate in the special Error mode. (For further details, see Interface Considerations Section.)

FIGURE 9-28 8279 command word formats and bit descriptions. (*Intel Corporation*)

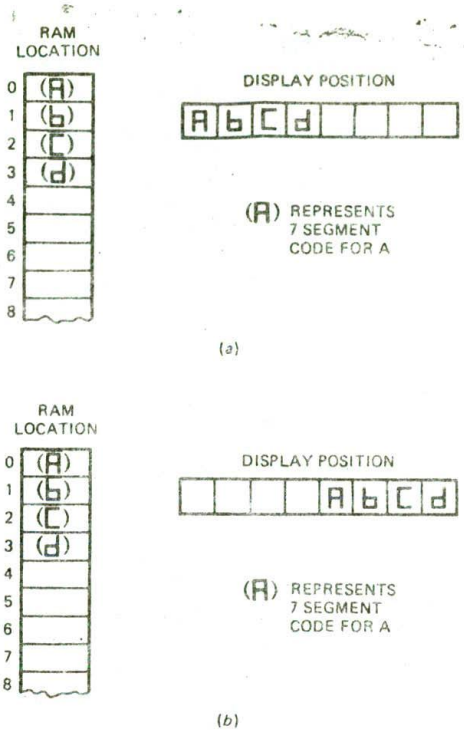


FIGURE 9-29 8279 RAM and display location relationships. (*a*) Left entry. (*b*) Right entry.

scanning a keyboard or turning on digit drivers, you need a pattern of stepping lows. In encoded mode the 8279 puts out a binary count sequence on its SL0–SL3 scan lines, and an external decoder such as the 7445 is used to produce the stepping lows. If you have only 4 digits to refresh, you can program the 8279 in decoded mode. In this mode, the 8279 directly outputs stepping lows on the four scan lines. The second choice you have to make for this control word is whether you want *two-key lockout* or *N-key rollover*. In the two-key mode, one key must be released before another keypress will be detected and processed. In the N-key rollover mode, if two keys are pressed at nearly the same time, both keypresses will be detected and debounced and their codes put in the FIFO RAM in the order the keys were pressed.

In addition to being used to scan a keyboard, the 8279 can also be used to scan a matrix of switch sensors, such as the metal strips and magnetic sensors you see on store windows and doors. In sensor matrix mode, the 8279 scans all the sensors and stores the condition of up to 64 switches in the FIFO RAM. If the condition of any of the switches changes, an IRQ signal is sent out on the IRQ pin. An interrupt service procedure can then sound an alarm and let the guard dogs loose. The return lines of the 8279 can also function as a strobed input port in much the same way as port A or B on an 8255A.

The SDK-86 initializes the 8279 for eight-character display, left entry, encoded scan, two-key lockout. See if you can determine the mode-set control word for these conditions. You should get 00000000.

The next control word you have to send the 8279 is the *program-clock word*. The 8279 requires an internal clock frequency of about 100 kHz. A programmable divider in the 8279 allows you to apply some available frequency, such as the 2.45-MHz PCLK signal, to its clock input and divide this frequency down to the needed 100 kHz. The lower 5 bits of the program-clock control word simply represent the binary number you want to divide the applied clock by. For example, if you want to divide the input clock frequency by 24, you send a control word with 001 in the upper 3 bits and 11000 in the lower 5 bits.

The final control word needed for basic initialization is the *clear* word. You need to send this word to tell the 8279 what code to send to the segments to turn them off while the 8279 is switching from one digit to the next. (Refer to Figure 9-26 and its discussion.) In addition to telling the 8279 what blanking character to use during refresh, this 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 Figure 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.

Figure 9-30, p. 274, 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 Figure 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

```
                    ;INITIALIZATION
MOV DX, OFFEAH      ; Point at 8279 control address
MOV AL, 00000000B   ; 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 all 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, 5BH         ; 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
```

FIGURE 9-30 8086 instructions to initialize SDK-86 8279, write to display RAM, and read FIFO RAM.

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 Figure 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.

Characters can be read from the 8279 on a polled basis as well as on an interrupt basis. To do this, you simply read and test the status word over and over again until bit 0 of the status word becomes a 1. Since the basic SDK-86 does not have an 8259A to receive interrupt inputs, the SDK-86 monitor uses this polling method to tell when the FIFO holds a keypressed code.

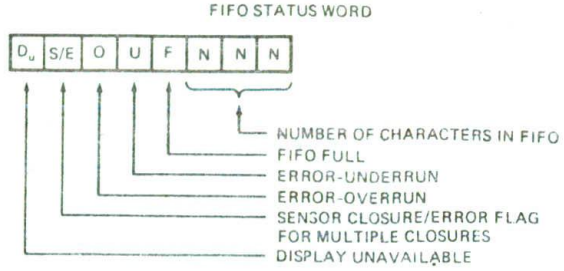


FIGURE 9-31 8279 status word format.

SDK-86 DISPLAY DRIVER PROCEDURE

Figure 9-32 shows an example of an I/O driver which will send the contents of the four nibbles in the CX register to four SDK-86 LED displays. You may have used this procedure for a variety of experiments; now you get to see how it works.

This procedure assumes the 8279 has already been initialized by the SDK-86 monitor program, or as shown in the first part of Figure 9-30. If AL is 0 when this procedure is called, the contents of CX will be displayed on the data field LEDs. If AL is not 0, then the contents of CX will be displayed on the address field LEDs. There are two main points for you to see in this procedure.

The first is the sending of the write-display-RAM control word to the 8279 so we can write to the desired locations in the display RAM. Note that for the data field we write a control word of 90H, which tells the 8279 to put the next data word sent into the first location in the display RAM. Since the 8279 is initialized for left entry, the first location should correspond to the leftmost display digit. However, if you look at sheet 8 of the SDK-86 schematics, you will see that digit 1 (leftmost as far as the 8279 is concerned) is actually the rightmost on the board. This means that for the SDK-86, the position of a 7-segment code in the display RAM corresponds to its position in the display starting from the right! All you have to do is send the 7-segment code for a number you want to display in a particular digit position to the corresponding location in the display RAM.

The next part of the display procedure to take a close look at is the instructions which convert the four hex nibbles in the CX register to the corresponding 7-segment codes for sending to the display RAM. To do this, we first shuffle and mask to get each nibble into a byte by itself. We then use a lookup table and the XLAT instruction to do the actual conversion. Note that when making up 7-segment codes for the SDK-86 board, a high turns on a segment, bit D0 of a display RAM byte represents the "a" segment, bit D6 represents the "g" segment, and bit D7 represents the decimal point. If you are displaying only BCD digits, you can replace the upper six values in the segment code table with values which allow you to blank a digit, display an A or P on a clock, etc. Work your way through the conversion section as a review of using the XLAT instruction.

```
                                ;8086 PROCEDURE F9-32.ASM
                                ;ABSTRACT  : Displays a 4-digit hex or BCD number on LEDs of the SDK-86.
                                ;INPUTS    : Data in CX, control in AL.
                                           ; AL  = 00H  data displayed in data-field of LEDs
                                           ; AL <> 00H  data displayed in address field of LEDs.
                                ;PORTS     : None used
                                ;PROCEDURES: None used
                                ;REGISTERS : Destroys nothing

                                PUBLIC  DISPLAY_IT

0000                            DATA SEGMENT   WORD  PUBLIC
                                           ;      0    1    2    3    4    5    6    7
0000  3F 06 5B 4F 66 6D 7D +    SEVEN_SEG DB   3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H
      07
                                           ;      8    9    A    b    C    d    E    F
0008  7F 6F 77 7C 39 5E 79 +               DB   7FH, 6FH, 77H, 7CH, 39H, 5EH, 79H, 71H
      71
0010                            DATA ENDS

0000                            CODE SEGMENT WORD PUBLIC
                                  ASSUME CS:CODE, DS:DATA
0000                            DISPLAY_IT PROC  FAR
0000  9C                               PUSHF                       ; Save flags
0001  1E                               PUSH DS                     ; Save caller's registers
0002  50                               PUSH AX
0003  53                               PUSH BX
0004  51                               PUSH CX
0005  52                               PUSH DX
0006  BB 0000s                        MOV  BX, DATA                 ; Init DS as needed for procedure
0009  8E DB                           MOV  DS, BX
000B  BA FFEA                         MOV  DX, 0FFEAH               ; Point at 8279 control address
000E  3C 00                           CMP  AL, 00H                  ; If data field required then
0010  74 05                           JZ   DATFLD                   ; load control word for data field
0012  B0 94                           MOV  AL, 94H                  ; else load address-field control word
0014  EB 03 90                        JMP  SEND                     ; Send control word
0017  B0 90                     DATFLD: MOV  AL, 90H                 ; Load control word for data field
0019  EE                        SEND:   OUT  DX, AL                  ; Send control word to 8279
001A  BB 0000r                        MOV  BX, OFFSET SEVEN_SEG     ; Pointer to seven-segment codes
001D  BA FFE8                         MOV  DX, 0FFE8H               ; Point at 8279 display RAM
0020  8A C1                           MOV  AL, CL                   ; Get low byte to be displayed
0022  24 0F                           AND  AL, 0FH                  ; Mask upper nibble
0024  D7                              XLATB                         ; Translate lower nibble to 7-seg code
0025  EE                              OUT  DX, AL                   ; Send to 8279 display RAM
0026  8A C1                           MOV  AL, CL                   ; Get low byte again
0028  B1 04                           MOV  CL, 04                   ; Load rotate count
002A  D2 C0                           ROL  AL, CL                   ; Move upper nibble into low position
002C  24 0F                           AND  AL, 0FH                  ; Mask upper nibble
002E  D7                              XLATB                         ; Translate 2nd nibble to 7-seg code
002F  EE                              OUT  DX, AL                   ; Send to 8279 display RAM
0030  8A C5                           MOV  AL, CH                   ; Get high byte to translate
0032  24 0F                           AND  AL, 0FH                  ; Mask upper nibble
0034  D7                              XLATB                         ; Translate to 7-seg code
0035  EE                              OUT  DX, AL                   ; Send to 8279 display RAM
0036  8A C5                           MOV  AL, CH                   ; Get high byte to fix upper nibble
0038  D2 C0                           ROL  AL, CL                   ; Move upper nibble into low position
003A  24 0F                           AND  AL, 0FH                  ; Mask upper nibble
003C  D7                              XLATB                         ; Translate to 7-seg code
003D  EE                              OUT  DX, AL                   ; 7-seg code to 8279 display RAM
003E  5A                              POP  DX                       ; Restore all registers and flags
003F  59                              POP  CX
0040  5B                              POP  BX
0041  58                              POP  AX
0042  1F                              POP  DS
0043  9D                              POPF
0044  CB                              RET
0045                            DISPLAY_IT ENDP
0045                            CODE    ENDS
                                        END
```

FIGURE 9-32 Procedure to display contents of CX register on SDK-86 LED displays.

INTERFACING TO 18-SEGMENT AND DOT-MATRIX LED DISPLAYS

In the preceding examples we used an 8279 to refresh some 7-segment displays. The 7-segment codes for each digit were stored in successive locations in the display RAM. To display ASCII codes on 18-segment LED displays, you can store the ASCII codes for each digit in the display RAM. (Remember that the A lines are driven from the upper nibble of the display RAM and the B lines are driven by the lower nibble.) An external ROM is used to convert the ASCII codes to the required 18-segment codes and send them to the segment drivers. Strobes for each digit driver are produced just as they are for the 7-segment displays in Figure 7-8. The refreshing of each digit then proceeds just as it does for the 7-segment displays.

Refreshing 5 by 7 dot-matrix LED displays is a little more complex because, instead of lighting an entire digit, you have to refresh one row or one column at a time in each digit. To solve this problem, Beckman Instruments, Hewlett-Packard, and several other companies make large integrated display/driver devices which require you to send only a series of ASCII codes for the characters you want displayed.

Liquid-Crystal Display Operation and Interfacing

LCD OPERATION

Liquid-crystal displays are created by sandwiching a thin (10- to 12-μm) layer of a liquid-crystal fluid between two glass plates. A transparent, electrically conductive film or backplane is put on the rear glass sheet. Transparent sections of conductive film in the shape of the desired characters are coated on the front glass plate. When a voltage is applied between a segment and the backplane, an electric field is created in the region under the segment. This electric field changes the transmission of light through the region under the segment film.

There are two commonly available types of LCD: *dynamic scattering* and *field-effect*. The dynamic scattering type scrambles the molecules where the field is present. This produces an etched-glass-looking light character on a dark background. Field-effect types use polarization to absorb light where the electric field is present. This produces dark characters on a silver-gray background.

Most LCDs require a voltage of 2 or 3 V between the backplane and a segment to turn on the segment. You can't, however, just connect the backplane to ground and drive the segments with the outputs of a TTL decoder, as we did the static LED display in Figure 9-24. The reason for this is that LCDs rapidly and irreversibly deteriorate if a steady dc voltage of more than about 50 mV is applied between a segment and the backplane. To prevent a dc buildup on the segments, the segment-drive signals for LCDs must be square waves with a frequency of 30 to 150 Hz. Even if you pulse the TTL decoder, it still will not work because the output low voltage of TTL devices is greater than 50 mV. CMOS gates are often used to drive LCDs.

Figure 9-33*a* shows how two CMOS gate outputs can be connected to drive an LCD segment and backplane. Figure 9-33*b* shows typical drive waveforms for the backplane and for the on and the off segments. The off (in this case unused) segment receives the same drive signal as the backplane. There is never any voltage between them, so no electric field is produced. The waveform for the on segment is 180° out of phase with the backplane signal, so the voltage between this segment and the backplane will always be +V. The logic for this is quite simple because you only have to produce two signals, a square wave and its complement. To the driving gates, the segment-backplane sandwich appears as a somewhat leaky capacitor. The CMOS gates can easily supply the current required to charge and discharge this small capacitance.

Older and/or inexpensive LCD displays turn on and off too slowly to be multiplexed the way we do LED displays. At 0°C, some LCDs may require as much as 0.5 s to turn on or off. To interface to these types we use a nonmultiplexed driver device. Newer, more expensive LCDs can turn on and off faster, so they are often multiplexed using a variety of techniques. In the following section we show you how to interface a nonmultiplexed LCD display to a microprocessor such as the SDK-86.

INTERFACING A MICROCOMPUTER TO NONMULTIPLEXED LCD DISPLAYS

Figure 9-34 shows how an Intersil ICM7211M can be connected to drive a 4-digit, nonmultiplexed, 7-segment

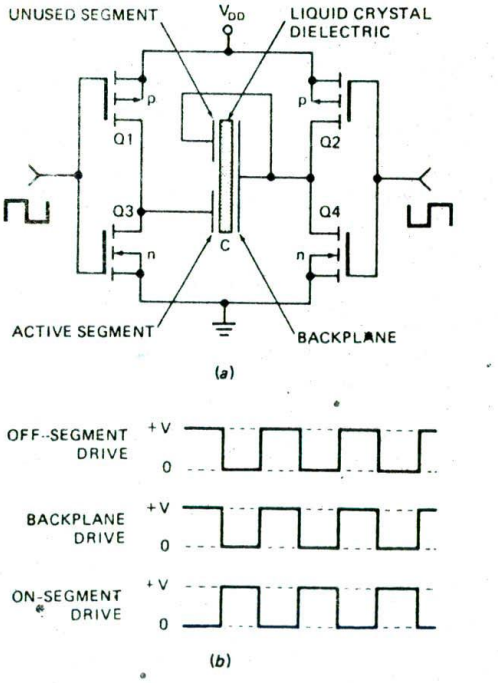


FIGURE 9-33 LCD drive circuit and drive waveforms. (*a*) CMOS drive circuits. (*b*) Segment and backplane drive waveforms.

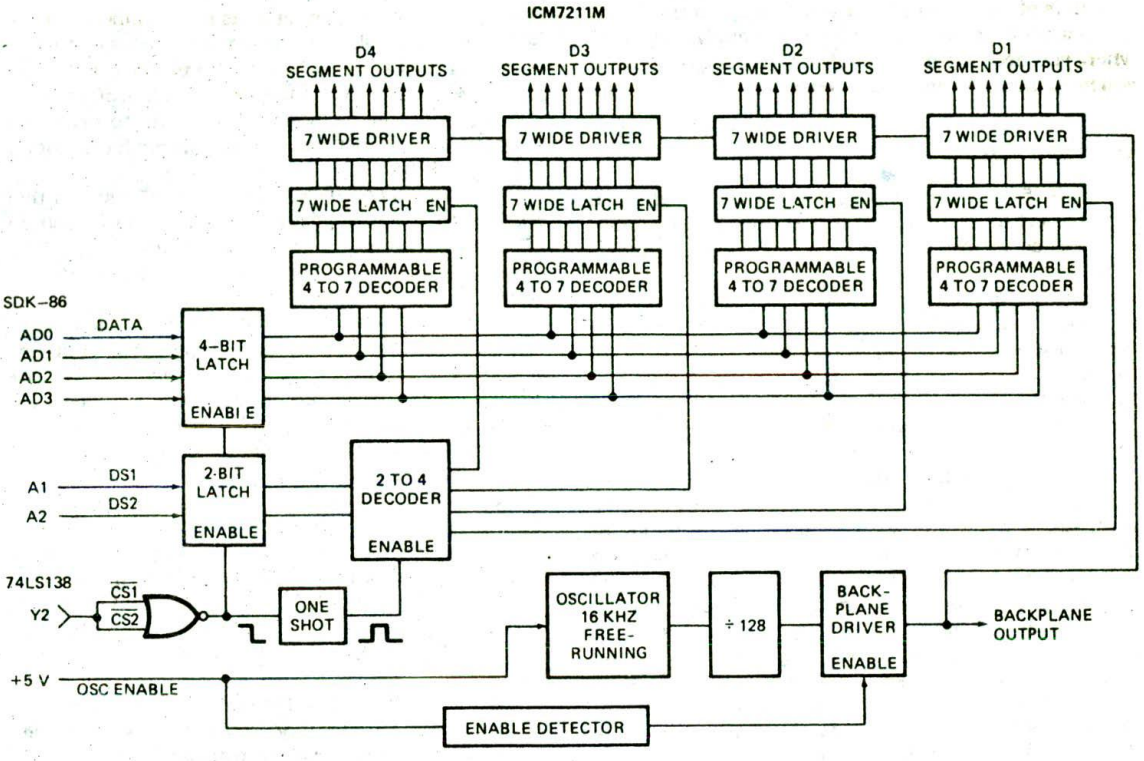


FIGURE 9-34 Circuit for interfacing four LCD digits to an SDK-86 bus using Intersil ICM7211M.

LCD display such as you might buy from your local electronics surplus store. The 7211M inputs can be connected to port pins or directly to microcomputer buses as shown. For our example here, we have connected the $\overline{CS}$ inputs to the Y2 output of the 74LS138 port decoder that we showed you how to add to an SDK-86 board in Figure 8-14. According to the truth table in Figure 8-15, the device will then be addressable as ports with a base address of FF10H. SDK-86 system address line A2 is connected to the digit-select input (DS2), and system address line A1 is connected to the DS1 input. This gives digit 4 a system address of FF10H. Digit 3 will be addressed at FF12H, digit 2 at FF14H, and digit 1 at FF16H. The data inputs are connected to the lower four lines of the SDK-86 data bus. The oscillator input is left open.

To display a character on one of the digits, you simply put the 4-bit hex code for that digit in the lower 4 bits of the AL register and output it to the system address for that digit. The ICM7211M converts the 4-bit hex code to the required 7-segment code. The rising edge of the $\overline{CS}$ input signal causes the 7-segment code to be latched in the output latches for the addressed digit. An internal oscillator automatically generates the segment and backplane drive waveforms shown in Figure 9-33*b*.

For interfacing with LCD displays which can be multiplexed, the Intersil ICM7233 can be used.

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 Figure 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 Figure 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 here, suppose that you want 20 mA to flow through an LED. You then look through your transistor collection to find an NPN transistor which can carry the required current, has a collector-to-emitter breakdown voltage (V_{BCEO}) greater than the applied supply voltage, and can dissipate the power generated by the current flowing through it. We usually keep some inexpensive 2N3904 NPNs and some 2N3906 PNPs on hand for low-current switch applications such as this. Some alternatives are the 2N2222 NPN and the 2N2907 PNP.

When you decide what transistor you are going to use, look up its current gain, h_{FE}, on a data sheet. If you don't have a data sheet, assume a value of 50 for the current gain of small-signal transistors such as these.

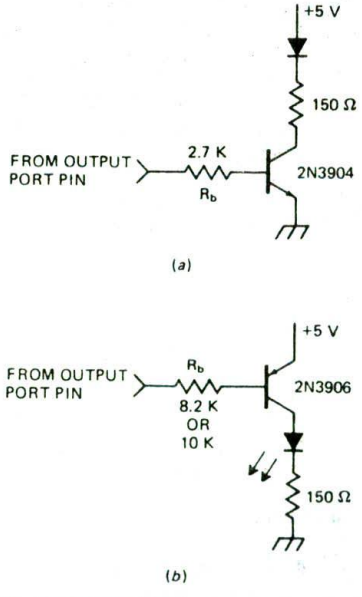


FIGURE 9-35 Transistor buffer circuits for driving LED from 8255A port pin. (a) NPN. (b) PNP.

Remember, current gain, or β, as it is commonly called, is the ratio of collector current to the base current needed to produce that current. To produce a collector current of 20 mA in a transistor with a β of 50 requires a base current of 20 mA/50 or 0.4 mA. To drive this buffer transistor, then, the output port pin has to supply only the 0.4 mA.

The V_{OH}(PER) specification of the 8255A shows that an 8255A peripheral port pin can only source 200 μA (0.2 mA) of current and still maintain a legal TTL-compatible output voltage of 2.4 V! The outputs can source more than 0.2 mA, but if they source more than 0.2 mA, the output high voltage will drop below 2.4 V. You don't care about the output high voltage dropping below 2.4 V except in the unlikely case that you are trying to drive a logic gate input off the same port pin as the transistor. Let's assume an output voltage of 2.0 V for calculating the value of our current-limiting resistor, R_b. The value of this resistor is not very critical as long as it lets through enough base current to drive the transistor. The base of the NPN transistor will be at about 0.7 V when the transistor is conducting, and the output port pin will be at least 2.0 V. This leaves a voltage of 1.3 V across R_b. Dividing the 1.3 V across R_b by the desired base current of 0.4 mA gives an R_b value of 3.25 kΩ. A 2.7-kΩ or 3.3-kΩ resistor will work fine here.

If you chose to use the PNP circuit in Figure 9-35*b*, an output pin on an 8255A could easily sink enough current to drive the base of the transistor. The V_{OL}(PER) specification for an 8255A indicates that an output pin can sink at least 1.7 mA and still have an output low voltage no greater than 0.45 V. The base of the PNP transistor in Figure 9-35*b* will be at about +4.3 V when the transistor is on, and the output of the 8255A will be at about +0.3 V. This means that the R_b in Figure 9-35*b* has about 4 V across it. Dividing this voltage by the required 0.4 mA gives an R_b value of 10 kΩ.

When you need to switch currents larger than about 50 mA on and off with an output port line, a single transistor does not have enough current gain to do this dependably. One solution to this problem is to connect two transistors in a Darlington configuration, as shown in Figure 9-36. A circuit such as this might be used to drive a small solenoid valve which controls the flow of a chemical into our printed-circuit-board-making machine or a small solenoid in the print heads of a dot-matrix printer. The dotted lines around the two transistors in Figure 9-36 indicate that both devices are contained in the same package. Here's how this configuration works.

The output port pin supplies base current to transistor Q1. This base current produces a collector current β times as large in Q1. The collector current of Q1 becomes the base current of Q2 and is amplified by the current gain of Q2. The result of this is that the device acts like a single transistor with a current gain of β Q1 × β Q2 and a base-emitter voltage of about 1.4 V. The internal resistors help turn off the transistors. The TIP110 device we show here has a minimum β of 1000 at 1 A, so if we assume that we need 400 mA to drive the solenoid, then the worst-case current that must be supplied by the

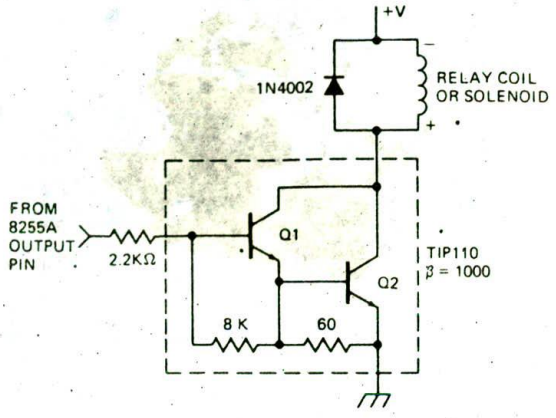


FIGURE 9-36 Darlington transistor used to drive relay coil or solenoid.

output port pin is about 400 mA/1000 or 0.4 mA. As we indicated before, a port pin can easily do this.

If the drive current required for the Darlington is too high for the port output, you can add, for example, a 3.3-kΩ resistor from the transistor base to +5 V to supply an additional milliampere of drive current. The port output can easily sink this additional milliampere of current when it is in the low state. Also, another transistor could be added as a buffer between the output pin and the Darlington input. Note that since the V_{BE} of the Darlington is about 1.4 V, a smaller R_b is needed here. Now let's check out the power dissipation.

According to the data sheet for the TIP110, it comes in a TO-220 package which can dissipate up to 2 W at an ambient temperature of 25°C with no heat sink. With 400 mA flowing through the device, it will have a collector-emitter saturation voltage of about 2 V. Multiplying the current of 400 mA times the voltage drop of 2 V gives us a power dissipation of 0.8 W for our circuit here. This is well within the limits for the device. A rule of thumb that we like to follow is, If the calculated power dissipation for a device such as this is more than half of its 25°C no-heat-sink rating, mount the device on the chassis or a heat sink to make sure it will work on a hot day. If mounted on the appropriate heat sink, the device will dissipate 50 W at 25°C.

One more important point to mention about the circuit in Figure 9-36 is the reverse-biased diode connected across the solenoid coil. You must remember to put in this diode whenever you drive an inductive load such as a solenoid, relay, or motor. Here's why. The basic principle of an inductor is that it fights a change in the current through it. When you apply a voltage to the coil by turning on the transistor, it takes a while for the current to start flowing. This does not cause any major problems. However, when you turn off the transistor, the collapsing magnetic field in the inductor keeps the current flowing for a while. This current cannot flow through the transistor, because it is off. Instead, this current develops a voltage across the inductor with the polarity shown by the + and − signs on the coil in Figure 9-36. This induced voltage, sometimes called inductive "kick," will usually be large enough to break down the transistor if you forget to put in the diode. When the coil is conducting, the diode is reverse-biased, so it doesn't conduct. However, as soon as the induced voltage reaches 0.7 V, the diode turns on and supplies a return path for the induced current. The voltage across the inductor then is clamped at 0.7 V, the voltage across a conducting diode, so the transistor is saved.

Figure 9-37*a* shows how a device called a power MOSFET transistor can be used to drive a solenoid, relay, or motor winding. Power MOSFETS are somewhat more expensive than bipolar Darlingtons, but they have the advantage that they require only a voltage to drive them. The Motorola IRF130 shown here, for example, requires a maximum gate voltage of only 4 V to turn on a drain current of 8 A. Note that this MOSFET circuit also needs a reverse-biased diode across the solenoid to protect the transistor from inductive kick.

Figure 9-37*b* shows a power driver circuit using a newer device called an Isolated-Gate Bipolar Transistor (IGBT). In IGBT data books you may see the device referred to as an IGBT or as an MOSIGT. As you might expect from the schematic symbol, these devices are a compromise between bipolar transistors and MOSFETs. They have the high input impedance and fast switching speed of MOSFETs, and they have the low voltage drop and high current-carrying capacity of bipolar transistors. The Toshiba MG400H1US1, for example, has a collector-emitter breakdown voltage of 500 V and can

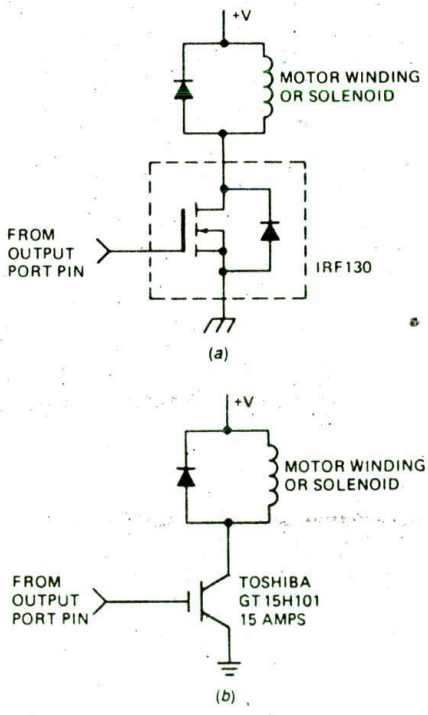


FIGURE 9-37 Circuits for driving solenoid or motor winding. (a) Power MOSFET circuit. (*b*) IGBT circuit.

switch a maximum current of 400 A. A driver device such as the National DS0026, Motorola MMH0026, or Silicon General SG1626 is used to convert the logic signal from an output port to the voltage and current levels required to rapidly switch high-power MOSFETs and IGBTs on and off.

Interfacing to AC Power Devices

To turn 110-V, 220-V, or 440-V ac devices on and off under microprocessor control, we usually use *mechanical* or *solid-state relays*. The control circuitry for both of these types of relay is electrically isolated from the actual switch. This is very important, because if the 110-V ac line gets shorted to the V_{CC} line of a microcomputer, it usually bakes most of the microcomputer's ICs.

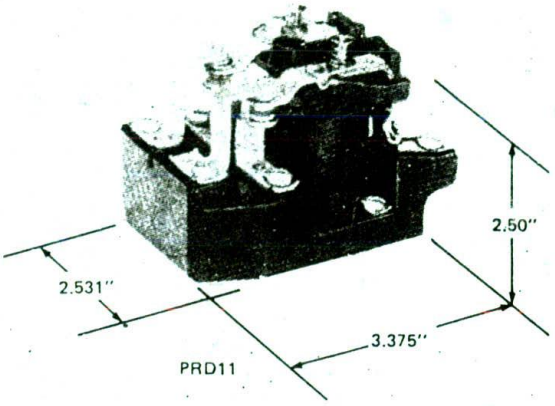
Figure 9-38*a* shows a picture of a mechanical relay. This relay has both normally open and normally closed contacts. When a current is passed through the coil of the relay, the switch arm is pulled down, opening the top contacts and closing the bottom set of contacts. The contacts are rated for a maximum current of 25 A, so this relay could be used to turn on a 1- or 2-hp motor or a large electric heater in one of the machines in our electronics factory. When driven from a 12-V supply, the coil requires a current of about 170 mA. The Darlington circuit shown in Figure 9-36 could easily drive this relay coil from a microcomputer port line.

Mechanical relays, sometimes called contactors, are available to switch currents from milliamperes up to several thousand amperes. Mechanical relays, however, have several serious problems. First of all, when the contacts are opened and closed, arcing takes place between the contacts. This causes the contacts to oxidize and pit, just as the ignition points in older-style cars used to do. 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.

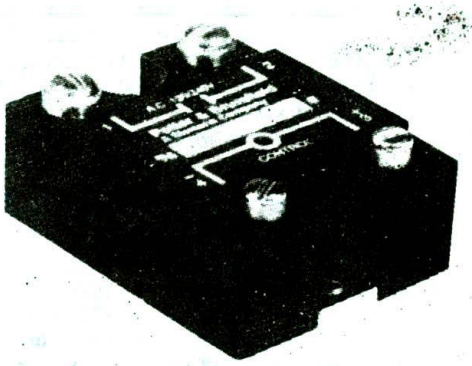
Figure 9-38*b* shows a picture of a solid-state relay which is rated for 25 A at 25°C if mounted on a suitable heat sink. Figure 9-38c shows a block diagram of the circuitry in the device and how it is connected from an output port to an ac load.

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.

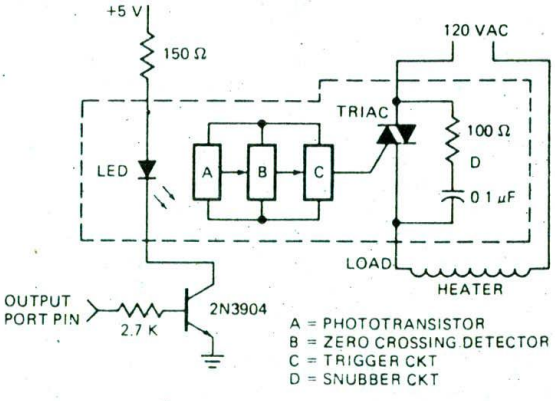
The actual switch in a solid-state relay is a triac. When



(*a*)



(*b*)



(*c*)

FIGURE 9-38 Relays for switching large currents. (*Potter and Brumfield*) (a) Mechanical. (*b*) Solid-state. (c) Internal circuitry for solid-state relay.

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 Figure 9-38*c*. 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 *four-phase steppers* here because their drive circuitry is much simpler.

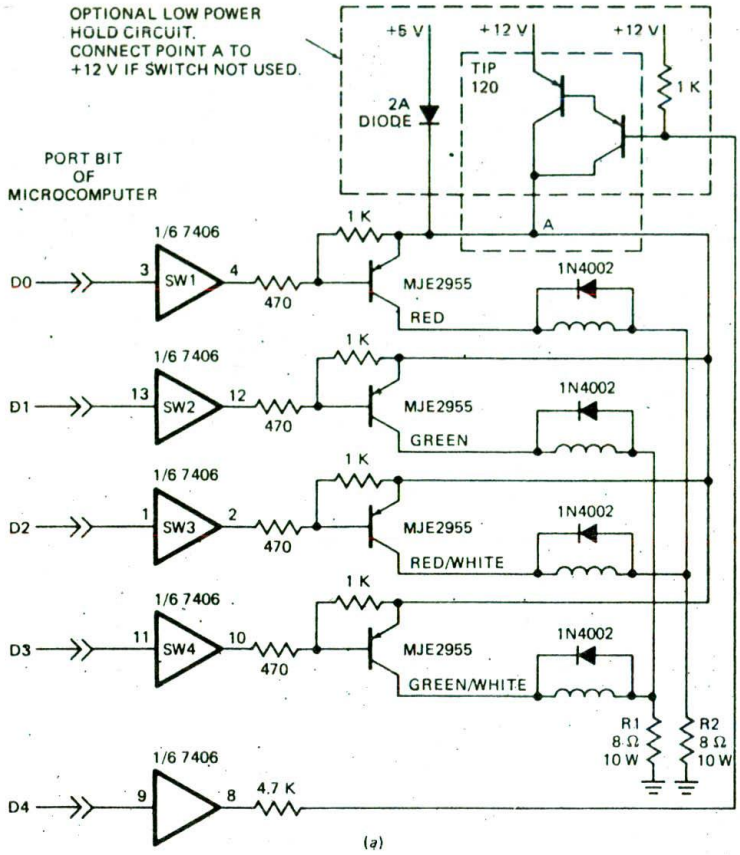
Figure 9-39, p. 282, shows a circuit you can use to interface a small four-phase stepper such as the Superior Electric MO61-FD302, IMC Magnetics Corp. Tormax 200, or a similar, nominal 5-V unit to five microcomputer port lines. If you build up this circuit, bolt some small heat sinks on the MJE2955 transistors 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 low turns on the PNP driver transistor and supplies current to a winding. Figure 9-39*b* shows the switching sequence to step a motor such as this clockwise or counterclockwise. (The directions assume you are facing the end of the motor shaft.) Here's how this works.

Suppose that SW1 and SW2 are turned on. Turning off SW2 and turning on SW4 will cause the motor to rotate one step of 1.8° clockwise. Changing to SW4 and SW3 on will cause the motor to rotate another 1.8° clockwise. Changing to SW3 and SW2 on will cause another step. After that, changing to SW2 and SW1 on again will cause another step clockwise. You can repeat the sequence until the motor has rotated as many steps clockwise as you want. To step the motor counterclockwise, you simply work through the switch sequence in the reverse direction. The motor is held in position between steps by the current through the coils. Figure 9-39*c* shows the switch sequence that can be used to rotate the motor half-steps of 0.9° clockwise or counterclockwise.

A close look at the switch sequence in Figure 9-39*b* shows an interesting pattern. To take the first step clockwise from SW2 and SW1 being on, the pattern of 1's and 0's is simply rotated one bit position around to the right. The 1 from SW1 is rotated around into bit 4. To take the next step, the switch pattern is rotated one more bit position. To step counterclockwise, the switch pattern is rotated left one bit position for each step desired. This rotating pattern can easily be produced with a sequence of 8086 instructions. Suppose that you initially load 00110011 into AL and output this to the switches. (Duplicating the switch pattern in the upper half of AL will make stepping easy.) To step the motor clockwise one step, you just rotate this pattern right one bit position and output it to the switches. To step counterclockwise one step, you rotate the switch pattern left one bit position and output it. You can repeat the rotate and output sequence as many times as needed to produce the desired number of steps.

After you output one step code, you must wait a few milliseconds before you output another step command because the motor can step only so fast. Maximum stepping rates for different types of steppers vary from a few hundred steps per second to several thousand steps per second. To achieve high stepping rates, the stepping rate is slowly increased to the maximum and then decreased as the desired number of steps is approached.



(a)

STEP	SWITCH				
	SW4	SW3	SW2	SW1	
1	0	0	1	1	CW
2	1	0	0	1	
3	1	1	0	0	
4	0	1	1	0	
1	0	0	1	1	CCW

1 = SWITCH ON

(b)

EIGHT-STEP INPUT SEQUENCE
(HALF-STEP MODE)

STEP	SW4	SW3	SW2	SW1
1	OFF	OFF	ON	ON
2	OFF	OFF	OFF	ON
3	ON	OFF	OFF	ON
4	ON	OFF	OFF	OFF
5	ON	ON	OFF	OFF
6	OFF	ON	OFF	OFF
7	OFF	ON	ON	OFF
8	OFF	OFF	ON	OFF
1	OFF	OFF	ON	ON

(c)

FIGURE 9-39 Four-phase stepper motor interface circuit and stepping waveforms. (a) Circuit. (b) Full-step drive signal order. (c) Half-step drive signal order.

When you step a stepper motor to a new position, it tends to oscillate around the new position before settling down. A common software technique to damp out this oscillation is to first send the pattern to step the motor toward the new position. When the motor has rotated part of the way to the new position, a word to step the motor backward is output for a short time. This is like putting the brakes on. The step-forward word is then sent again to complete the step to the next position. The timing for the damping command must be determined experimentally for each motor and load.

Before we go on, here are a couple of additional points about the circuit in Figure 9-39*a*, in case you want to add a stepper to your robot or some other project. First of all, don't forget the clamp diodes across each winding to save the transistors from inductive kick. Second, we need to explain the function of the current-limiting resistors, R1 and R2. The motor we used here has a nominal voltage rating of 5.5 V. This means that we could have designed the circuit to operate with a voltage of about 6.5 V on the emitters of the driver transistors (5.5 V for the motor plus 1 V for the drop across the transistor). For low stepping rates, this would work fine. However, for higher stepping rates and more torque while stepping, we use a higher supply voltage and current-limiting resistors, as shown. The point of this is that by adding series resistance, we decrease the L/R time constant. This allows the current in the windings to change more rapidly. For the motor we used, the current per winding is 0.88 A. Since only one winding on each resistor is ever on at a time, 6.5 V/0.88 A gives a resistor value of 7.4 Ω. To be conservative, we used 8-Ω, 10-W resistors. The optional transistor switch and diode connection to the +5-V supply are used as follows. When the motor is not stepping, the switch to +12 V is off, so the motor is held in position by the current from the +5-V supply. Before you send a step command, you turn on the transistor to +12 V to give the motor more current for stepping. When stepping is done, you turn off the switch to +12 V, and drop back to the +5-V supply. This cuts the power dissipation.

In small printers, one or more dedicated microprocessors are used to control the various operations in the printer. In this case, the microprocessors have plenty of time to control the print-head and line-feed stepper motors in software, as we described above. For applica-

tions where the main microcomputer is too busy to be bothered with controlling a stepper directly, a smart stepper controller device, such as the Sprague UCN-5804B shown in Figure 9-40, can be used in place of the circuitry in Figure 9-39. This device is manufactured with a combination of CMOS and bipolar technology, so it has high input impedance and high output current drive capability. The device contains a shift resister to produce the step patterns, the power driver transistors, and the clamp diodes. Control inputs allow you to specify half-step or full-step operation, step direction, and type of motor. To step the motor, a pulse or series of pulses is applied to the STEP input. A programmable counter such as the 8254 we discussed earlier in the chapter could be programmed to send a desired number of pulses to the controller.

For applications where steps of 0.9° are not small enough, a technique called "microstepping" is used to produce as many as 25,000 steps per revolution. For microstep control, each winding is driven with the output of a D/A converter instead of with on/off switches. This means that the current through a winding can have a range of values instead of just zero or maximum. If the current ratios in the four windings are changed slightly, the motor will take a tiny step. Microstepping is much more complex to implement, but it produces very smooth and precise motion.

OPTICAL MOTOR SHAFT ENCODERS

In order to control the machines in our electronics factory, the microcomputers in these machines often need information about the position, direction of rotation, and speed of rotation of various motor shafts. The microcomputer, of course, needs this information in digital form. The circuitry which produces this digital information from each motor for the microcomputer is called a *shaft encoder*. There are two basic types of shaft encoders, *absolute* and *incremental*. Here's how these two types work.

Absolute Encoders

Absolute encoders have a binary-coded disk such as the one shown in Figure 9-41, p. 284, on the rotating shaft. Light sections of the disk are transparent, and dark sections are opaque. An LED is mounted on one side of each track, and a phototransistor is mounted on the other side of each track, opposite the LED. Outputs from the four phototransistors will produce one of the binary codes shown in Figure 9-41. The phototransistor outputs can be conditioned with Schmitt-trigger buffers and connected to input port lines. Each code represents an absolute angular position of the shaft in its rotation. With a 4-bit disk, 360° are divided up into 16 parts, so the position of the shaft can be determined to the nearest 22.5°. With an 8-bit disk, the position of the disk can be determined to the nearest 360°/256, or 1.4°.

Note that the codes in Figure 9-41 follow a *Gray-code* sequence rather than a normal binary count sequence. Using Gray code reduces the size of the largest possible error in reading the shaft position to the value of the least significant bit. If the disk used straight binary code, the largest possible error would be the value of the

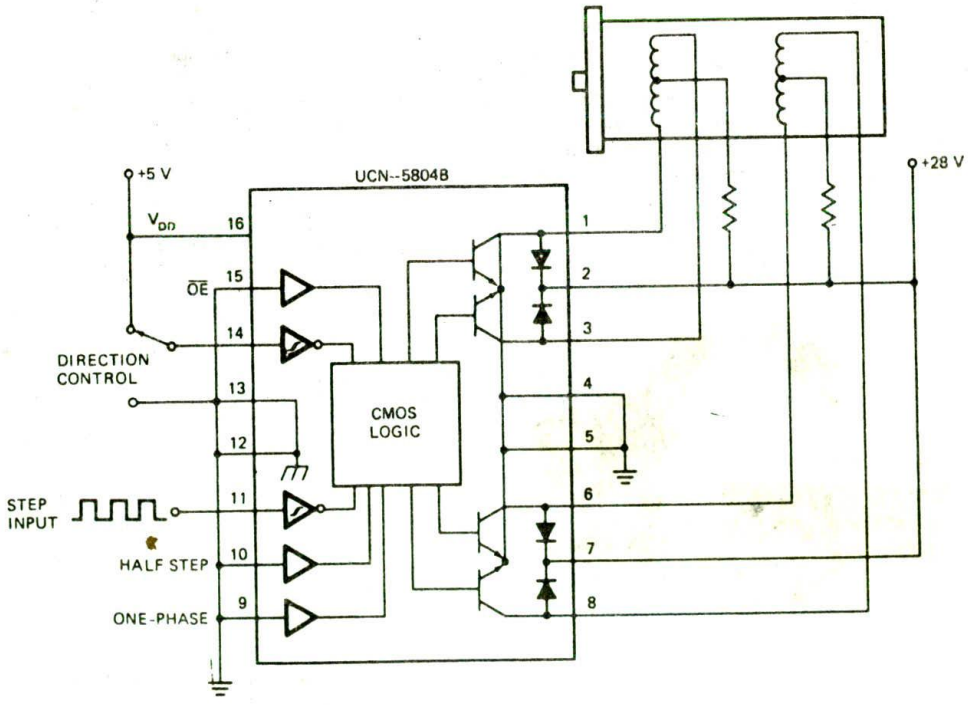


FIGURE 9-40 UCN-5804B stepper motor driver.

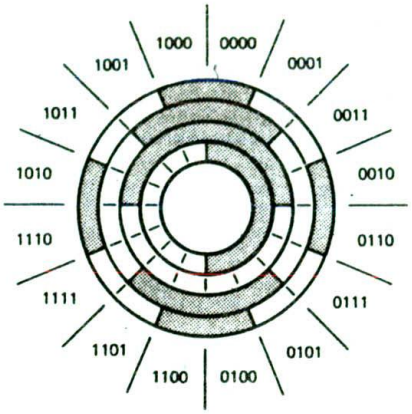


FIGURE 9-41 Gray-code optical-encoder dish used to determine angular position of a rotating shaft.

most significant bit. Look at the parallel listings of binary and Gray codes in Table 1-1 to help you see why this is the case.

To start, assume that a binary-encoded disk was used and that the disk was rotating from position 0111 (7) to position 1000 (8). Now suppose that the detectors pick up the change to 000 on the least significant 3 bits, but don't pick up the change to 1 on the most significant bit. The output code would then be 0000 instead of the desired 1000. This is an error equal to the value of the MSB. Now, while this is fresh in your mind, look across the table at the same position change for the Gray-code encoder. The Gray code for position 7 is 0100, and the Gray code for position 8 is 1100. Note that only 1 bit changes for this transition. If you look at the Gray-code table closely, you will see that this is the case for all the transitions. This means that if a detector fails to pick up the new bit value during a transition, the resulting code will always be the code for the preceding position. This represents a maximum error equal to the value of the LSB.

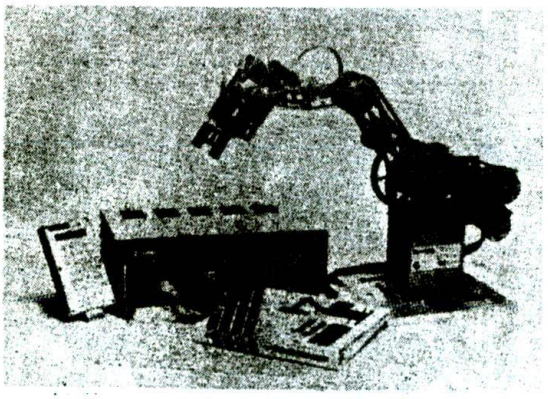


FIGURE 9-42 Rhino XR robotics system. (*Rhino Robots Incorporated*)

Absolute encoding using a Gray-code disk has the advantage that each position is represented by a specific code which can be directly read in by the microcomputer. Disadvantages of absolute encoding are the multiple detectors needed, the multiple lines required, and the difficulty keeping track of position during multiple rotations. Incremental encoders solve some of these problems.

Incremental Encoders

An incremental encoder produces a pulse for each increment of shaft rotation. Figure 9-42 shows an early version of the Rhino XR-2 robot arm, which uses incremental encoders to determine the position and direction of rotation for each of its motors. For this encoder, a metal disk with two tracks of slotted holes is mounted on each motor shaft. An LED is mounted on one side of each track of holes, and a phototransistor is mounted opposite the LED on the other side of the disk. Each phototransistor produces a train of pulses as the disk is rotated. The pulses are passed through Schmitt-trigger buffers to sharpen their edges.

The top part of Figure 9-43 shows a section of the encoder disk straightened out so it is easier to see the pulses produced as it rotates. The two tracks of slotted holes are 90° out of phase with each other, so as the disk is rotated, the waveforms shown at the bottom of Figure 9-43 will be produced by the phototransistors for rotation in one direction. Rotation in the other direction will shift the phase of the waveforms 180°, so that the B waveform leads the A waveform by 90° instead of lagging it by 90°. Now the question is, How do you get position, speed, and direction information from these waveforms?

You can determine the speed of rotation by simply counting the number of pulses from one detector in a fixed time interval, such as 1 s. As we described in Chapter 8, you can use a programmable timer and an

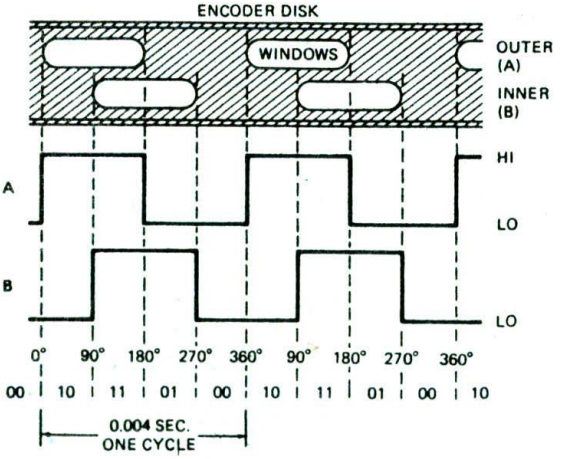


FIGURE 9-43 Optical-encoder disk slot pattern and output waveforms.

interrupt procedure to count off intervals of 1 s. If you connect the output of the detector to another interrupt input, you can use another interrupt procedure to count the number of holes that pass by in a 1-s interval. Each track has six holes, so six pulses will be produced for each revolution. Some simple arithmetic will give you the speed in revolutions per minute (rpm).

You can determine the direction of rotation with hardware or with software. For the hardware approach, connect the A signal to the D input of a D flip-flop and the B signal to the clock input of the flip-flop. The rising edge of the B signal will clock the level of the A signal at that point through the flip-flop to its Q output. If you look at the waveforms in Figure 9-43, you should see that the Q output will be high for rotation in the direction shown. You should also be able to convince yourself that the Q output will be low for rotation in the other direction.

To determine the direction of rotation with software, you can detect the rising edge of the B signal on a polled or an interrupt basis and then read the logic level on the A signal. As shown in the waveforms, the A signal being high when B goes high represents rotation in one direction, and the A signal being low when B goes high represents rotation in the opposite direction.

To determine the position of the motor shaft, you simply count off how many holes the motor has moved from some "home" position. On the Rhino robot arm a small mechanical switch on each axis is activated when the arm is in its starting, or home, position. When you turn on the power, the motor controller/driver box automatically moves the arm to this home position. To move the arm to some new position, you calculate the number of holes each motor must rotate to get the arm to that position. For each motor, you then send the controller a command which tells it which direction to rotate that motor and how many holes to rotate it. The controller will drive the motor the specified number of holes in the specified direction. If you then manually rotate the encoder wheel or some heavy load moves the arm and rotates the encoder disk, the controller will detect the change in position of the disk and drive the motor back to its specified position. This is an example of digital *feedback control*, which is easily done with a microcomputer. The Rhino controller uses an 8748 single-chip microcomputer to interpret and carry out the commands you send it. Commands are sent to the controller in the serial ASCII form described at the start of Chapter 13.

Incidentally, you may wonder at this point why the designers of the Rhino arm did not use stepper motors such as those we described in a previous section. The answers are: Stepper motors are much more expensive than the simple dc motors used, and if a stepper motor is forced back a step by a sudden load change, there is no way to know about it and correct for it unless it has an external encoder. Also, the dc motor-encoder approach better demonstrates the method used in large commercial robots.

In the Rhino robot arm, each motor drives its section of the arm through a series of gears. Gearing the motor down reduces the force that the motor has to exert and makes the exact position of the motor shaft less critical. Therefore, for the Rhino, six sets of slots in the encoder disk are sufficient. However, for applications where a much more accurate indication of shaft position is needed, a self-contained shaft encoder such as the Hewlett-Packard HEDS-5000 is attached to the motor shaft. These encoders have two track-encoder disks with 500 tiny radial slits per track. The waveforms produced are the same as those shown for the Rhino encoder in Figure 9-43, but at a much higher frequency for the same motor speed.

Another common application for optical encoders is to produce digital information about the distance and direction that a computer mouse is moved. The Logictech™ mouse, for example, uses one optical encoder disk to produce pulses for vertical motion and another optical encoder disk to produce pulses for horizontal motion. As you move the mouse around on your desk, the rubber ball on the bottom of the mouse rotates the encoder disks. Data from the encoders is processed and sent to the microcomputer. The microcomputer uses the data from the mouse to move the on-screen cursor to the desired location.

Optical encoders in their many different forms are an important part of a large number of microcomputer-controlled machines.

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.

Simple input and output

Strobed I/O

Single-handshake I/O

Double-handshake data transfer

8255A initialization
- Mode 0, mode 1, mode 2
- Mode definition control word
- Set/reset control word

Computer numerical control (CNC) machines

Centronics parallel printer standard

I/O driver
- Control block
- Sentinel

Keyswitches—mechanical, capacitive, Hall effect

Detect, debounce, and encode a keyboard

Two-key lockout, two-key rollover

Code conversion using compare method

Code conversion using XLAT method

Error trapping

LED interfacing
- Direct drive
- Software-multiplexed display
- 8279 hardware display controller

8279 display and keyboard operation
- Encoded and decoded scan
- Keyboard/display mode set control word
- Clear control word
- Write-display control word

LCD interfacing
- Dynamic scattering display
- Field-effect display
- Backplane drive

Relays
- Mechanical
- Solid-state
- Electromagnetic interference
- Zero-point switching
- RC snubber circuit

Four-phase stepper motor drive

Shaft encoders—absolute and incremental

REVIEW QUESTIONS AND PROBLEMS

1. Why must data be sent to a printer on a handshake basis?
2. For the double-handshake data transfer in Figure 9-1*d*,
   a. Indicate which signal is asserted by the sender and which signal is asserted by the receiver.
   b. Describe the meaning of each of the signal transitions.
3. Why are the port lines of programmable port devices automatically put in the input mode when the device is first powered up or reset?
4. An 8255A has a system base address of FFF9H. What are the system addresses for the three ports and the control register for this 8255A?
5. *a.* Show the mode set control word needed to initialize an 8255A as follows: Port A—handshake input; Port B—handshake output; Port C—bits PC6 and PC7 as outputs.
   b. Show the bit set/reset control word needed to initialize the port A interrupt request and the port B interrupt request.
   c. Show the assembly language instructions you would use to send these control words to the 8255A in problem 4.
   d. Show the additional instruction you need if you want the handshake to be done on an interrupt basis through the IR3 input of the 8259A in Figure 8-14.
   e. Show the instructions you would use to put a high on port C, bit PC6 of this device.
6. Describe the exchange of signals between the tape reader, 8255A, and 8086 in Figure 9-7 as a byte of data is transferred from the tape reader to the microprocessor.
7. When connecting peripheral devices such as printers, terminals, etc., to a computer, why is it very important to connect the logic ground and the chassis ground together only at the computer?
8. Describe the function and direction of the following signals in a Centronics parallel printer interface.
   a. $\overline{\text{STROBE}}$
   b. $\overline{\text{ACKNLG}}$
   c. $\overline{\text{BUSY}}$
   d. $\overline{\text{INIT}}$
9. Modify the printer driver procedure in Figure 9-16 so that it stops sending characters to the printer when it finds a sentinel character of 03H, instead of using the counter approach.
10. Would the software method of generating the $\overline{\text{STROBE}}$ signal to the printer in Figure 9-16 still work if you tried to run the program with an 8-MHz 8086?
11. Show the instructions you would use to read the status byte from the 8255A in Question 5.
12. Describe the three major tasks needed to get meaningful information from a matrix keyboard.
13. Describe how the compare method of code conversion in Figure 9-20 works.
14. Why is error trapping necessary in real programs? Describe how the error trap in the program in Figure 9-20 works.
15. Assume that the rows of the circuit shown in Figure 9-44 are connected to ports FFF8H and the 74148 is connected to port FFFAH of an SDK-86 board. The 74148 will output a low on its $\overline{\text{GS}}$ output if a low is applied to any of its inputs. The way the keyboard is wired, the $\overline{\text{A2}}$, $\overline{\text{A1}}$, and $\overline{\text{A0}}$ outputs will have a 3-bit binary code for the column in which a low appears. Use the algorithm and discussion of Figure 9-20 to help you write a procedure which detects a keypress, debounces the keypress, and determines the row number and column number of the pressed key. The procedure should then combine the row code, column code, shift bit, and control bit into a single byte in the form: control, shift, row code, column code. The XLAT instruction can then be used to convert this code byte to ASCII for return to the calling program. *Hint:* Use a DB directive to make up the table of ASCII codes.

    Why is the XLAT approach more efficient than the compare technique for this case?

    NOTE: For test purposes, the keyboard matrix can be simulated by building the diodes, resistors, and 74148 on a prototyping board and using a jumper wire to produce a "keypress."
16. *a.* Calculate the value of the current-limiting resistor needed in series with each segment of a 7-

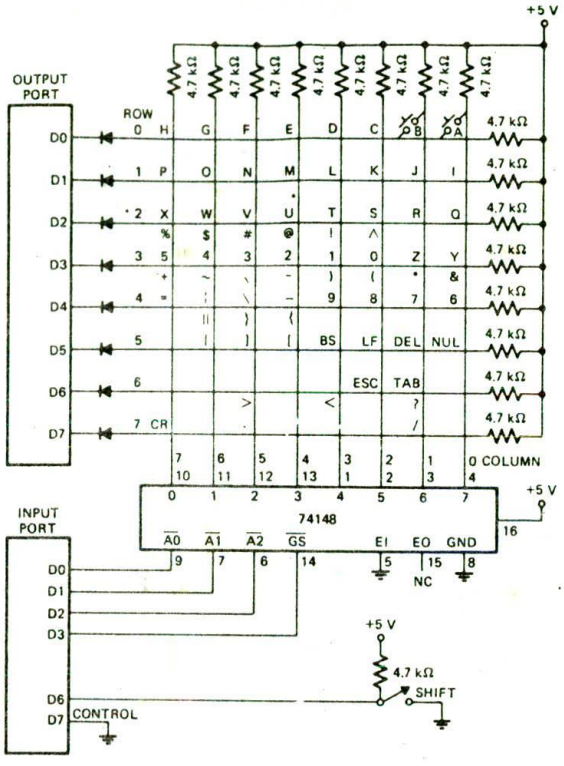


FIGURE 9-44 Interface circuitry for unencoded matrix keyboard for problem 15.

segment display driven by a 7447 if you want 40 mA per segment.

b. Approximately how much current is being pulsed through each LED segment on the SDK-86 board?

17. *a.* Write the algorithm for a procedure which refreshes the multiplexed LED displays shown in Figure 9-25. Assume that the procedure will be called every 2 ms by an interrupt signal to IR4 of an 8259A.

b. Write the assembly language instructions for the display refresh procedure. Since this procedure is called on an interrupt basis, all display parameters should be kept in named memory locations. If you have time, you can add the circuitry shown in Figure 9-25 to your microcomputer so you can test your program.

18. Figure 9-45, p. 288, shows a circuit for an 8 by 8 matrix of LEDs that you can connect to a couple of ports on your microcomputer to produce some interesting displays. The principle here is to output a 1 to port B for each LED you want turned on in the top row and then output a 1 to the D0 bit of port A to turn on that row. After 2 ms, you output the pattern you want in the second row to port B and a 1 to bit 1 of port A to turn on the second row. The process is repeated until all rows are done and then started over.

The row patterns can be kept in a table in memory. If you want to display a sequence of letters, you can display the contents of one table for a few seconds and then switch to another table containing the second letter. Using the rotate instruction, you can produce some scrolled displays. *Hint:* The wiring required to build the LED matrix can be reduced by using an IC 5 by 7 dot-matrix LED display such as the Texas Instruments TIL305.

Write the algorithm and program for an interrupt procedure (called every 2 ms) to refresh these displays.

19. You are assigned the job of fixing several SDK-86 boards with display problems. For each of the following problems, describe a possible cause of the problem and tell where you would look with an oscilloscope to check out your theory. Use the circuit on sheet 7 of Figure 7-8 to help you.

a. A segment never lights.

b. The leftmost digit of the data field never lights.

c. All the displays show dim "eights."

20. *a.* Show the command words and assembly language instructions necessary to initialize an 8279 at address 80H and 82H as follows: 16-character display, left entry, encoded-scan keyboard, N-key rollover; 1-MHz input clock divided to 100 kHz; blanking character FFH.

b. Show the 8279 instructions necessary to write 99H to the first location in the display RAM and autoincrement the display RAM pointer.

c. Show the assembly language instructions necessary to read the first byte from the 8279 FIFO RAM.

d. Determine the 7-segment codes you would have to send to the SDK-86 8279 to display the letters HELP on the data field display. Remember that D0 of the byte sent = B0 and D7 of the byte sent = A3.

e. Show the sequence of instructions you can send to the 8279 of the SDK-86 board to blank the entire display.

21. Write a procedure which polls the LSB of the 8279 status register on the SDK-86 board until it finds a key pressed, then reads the keypressed code from the FIFO RAM to AL and returns.

22. Why must the backplane and segment-line signals be pulsed for LCD displays?

23. Draw a circuit you could attach to an 8255A port B pin to drive a 1-A solenoid valve from a +12-V supply. You want a high on the port pin to turn on the solenoid.

24. Why must reverse-biased diodes always be placed across inductive devices when you are driving them with a transistor?

25. *a.* Define the terms MOSFET and IGBT.

b. What is a major advantage of these devices over bipolar Darlingtons when driving a high-power load with a microcomputer port line?

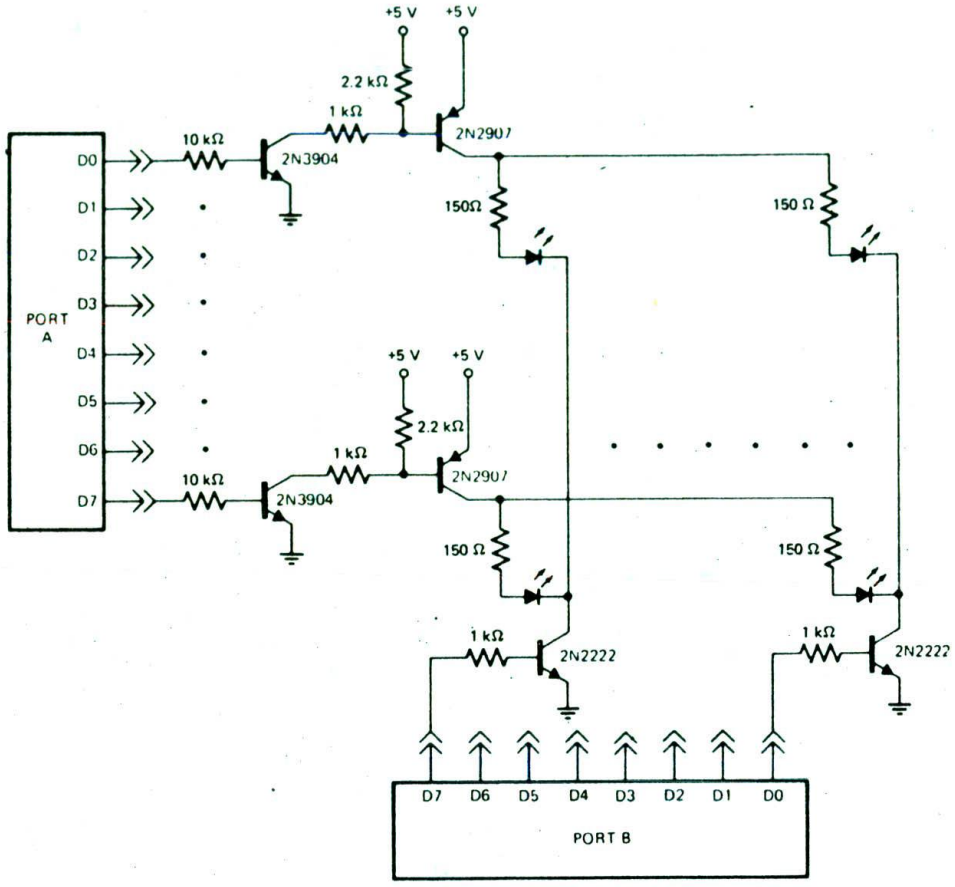


FIGURE 9-45 8 by 8 LED matrix circuitry for problem 18.

26. *a.* What are the major disadvantages of mechanical relays?
b. How do solid-state relays solve these problems?

27. *a.* How is electrical isolation between the control input and the output circuitry achieved in a solid-state relay?
b. Describe the function of the zero-crossing detector used in better-quality solid-state relays.
c. Why is a snubber circuit required across the triac of a solid-state relay when you are driving inductive loads?

28. Write the algorithm and the program for an 8086 procedure to drive the stepper motor shown in Figure 9-39. Assume the desired direction of rotation is passed to the procedure in AL (AL = 1 is clockwise, AL = 0 is counterclockwise) and the number of steps is passed to the procedure in CX. Also assume full-step mode, as shown in Figure 9-39*b*. Don't forget to delay 20 ms between step commands!

29. *a.* Why is Gray code, rather than straight binary code, used on many absolute-position shaft encoders?
b. If a Gray-code wheel has six tracks and each track represents 1 binary bit, what is its angular resolution?

30. *a.* Look at the encoder disk on the Rhino arm in Figure 9-42. Do the waveforms in Figure 9-43 represent clockwise or counterclockwise rotation of the motor shaft as seen from the gear end of the motor, which is what you care about?
b. Assume the A signal shown in Figure 9-43 is connected to bit D0 and the B signal is connected to bit D1 of port FFF8H. Write a procedure which determines the direction of rotation and passes a 1 back in AL for clockwise rotation and a 0 back in AL for counterclockwise rotation.
c. DC motors, such as those on the Rhino arms, are rotated clockwise by passing a current through them in one direction and rotated counterclockwise by passing a current through them in the opposite direction. Assume you

have a motor controller that responds to a 2-bit control word as follows:

00 = hold 01 = rotate clockwise
11 = hold 10 = rotate counterclockwise

. Write the algorithm and program for a procedure to rotate a motor. The number of holes is passed to the procedure in CX; the direction of rotation is determined by the value in AL. AL = 1 is clockwise; AL = 0 is counterclockwise.

CHAPTER 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.

REVIEW OF OPERATIONAL-AMPLIFIER CHARACTERISTICS AND CIRCUITS

Basic Operational-Amplifier Characteristics

Figure 10-1*a* shows the schematic symbol for an op amp. Here are the important points for you to remember about the basic op amp. First, the pins labeled +V and −V represent the power-supply connections. The voltages applied to these pins will usually be +15 V and −15 V, or +12 V and −12 V. The op amp also has two signal inputs. The input labeled with a − sign is called the inverting input, and the input labeled with a + sign is called the noninverting input. The + and − on these inputs have nothing to do with the power supply voltages. These signs indicate the phase relationship between a signal applied to that input and the result that signal produces on the output. If, for example, the noninverting input is made more positive than the inverting input, the voltage on the output will move in a positive direction. In other words, if a signal is applied to the noninverting input, the output signal will be in phase with the input signal. If the inverting input is made more positive than the noninverting input, the output signal will be inverted, or 180° out of phase with the input signal.

The ratio of the voltage out from an amplifier circuit to the input voltage is called *voltage gain*, A_V. In symbols,

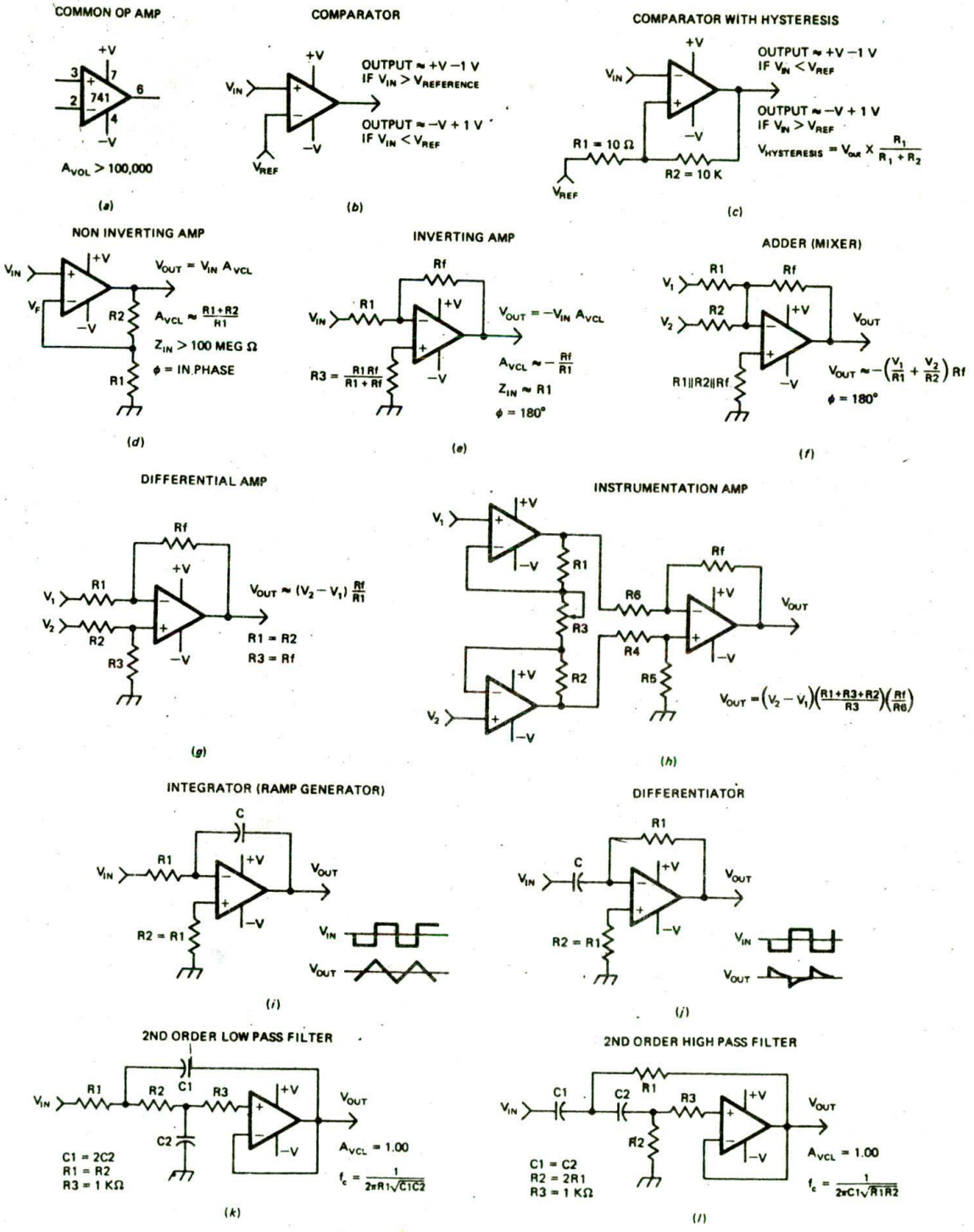


FIGURE 10-1 Overview of commonly used op-amp circuits. (*a*) Common op amp. (*b*) Comparator. (c) Comparator with hysteresis. (*d*) Noninverting amp. (e) Inverting amp. (*f*) Adder (mixer). (*g*) Differential amp. (*h*) Instrumentation amp. (*i*) Integrator (ramp generator). (*j*) Differentiator. (*k*) Second-order low-pass filter. (*l*) Second-order high-pass filter.

$A_V = V_{OUT}/V_{IN}$. The A_V for an op amp is typically 100,000 or more. (The number is variable with temperature and from device to device.) Another useful way of saying this is that an op amp amplifies the *difference* in voltage between these two inputs by 100,000 or more. Now let's see how much the output changes for a given input signal, and see how an op amp is used as a comparator.

Op-Amp Circuits and Applications

OP AMPS AS COMPARATORS

We said previously that an op amp amplifies the difference in voltage between its inputs by 100,000 or more. Suppose that you power an op amp with +15 V and −15 V, tie the inverting input of the op amp to ground, and apply a signal of +0.01 V dc to the noninverting input. The op amp will attempt to amplify this signal by 100,000 and produce the result on its output. An input signal of 0.01 V times a gain of 100,000 predicts an output voltage of 100 V. The maximum positive voltage the op-amp output can go to, however, is a volt or two less than the positive supply voltage, so this is as far as it goes. A common way of expressing this is to say the op-amp output "goes into saturation" at about +13 V.

Now suppose that you apply a signal of −0.01 V to the noninverting input. The output will now try to go to −100 V as fast as it can. The output, however, goes into saturation at about −13 V, so it stops there.

In this circuit the op amp effectively compares the input voltage with the voltage on the inverting input and gives a high or low output, depending on the result of the comparison. If the input is more than a few microvolts above the reference voltage on the inverting input, the output will be high (+13 V). If the input voltage is a few microvolts more negative than the reference voltage, the output will be low (−13 V). An op amp used in this way is called a *comparator*. Figure 10-1*b* shows how a comparator is usually labeled. The reference voltage applied to the inverting input does not have to be ground (0 V). An input voltage can be compared to any voltage within the input range specified for the particular op amp.

As you will see throughout this chapter, comparators have many applications. We might, for example, connect a comparator to a temperature sensor on the boiler in our electronics factory. When the voltage from the temperature sensor goes above the voltage on the reference input of the comparator, the output of the comparator will change state and send an interrupt signal to the microprocessor controlling the boiler. Commonly available comparators such as the LM319 have TTL-compatible outputs which can be connected directly to microcomputer ports or interrupt inputs.

Figure 10-1*c* shows another commonly used comparator circuit. Note in this circuit that the reference signal is applied to the noninverting input, and the input voltage is applied to the inverting input. This connection simply inverts the output state from those in the previous circuit. Note also in Figure 10-1*c* the positive-feedback resistors from the output to the noninverting input. This feedback gives the comparator a characteristic called *hysteresis*. Hysteresis means that the output voltage changes at a different input voltage when the input is going in the positive direction than it does when the input voltage is going in the negative direction. If you have a thermostatically controlled furnace in your house, you have seen hysteresis in action. The furnace, for example, may turn on when the room temperature drops to 65° F and then not turn off until the temperature reaches 68° F. Hysteresis is the difference between the two temperatures. Without this hysteresis, the furnace would be turning on and off rapidly if the room temperature were near 68° F. Another situation where hysteresis saves the day is the case where you have a slowly changing signal with noise on it. Hysteresis prevents the noise from causing the comparator output to oscillate as the input signal gets close to the reference voltage.

To determine the amount of hysteresis in a circuit such as that in Figure 10-1*c*, assume $V_{REF} = 0$ V and $V_{OUT} = 13$ V. A simple voltage-divider calculation will tell you that the noninverting input is at about 13 mV. The voltage on the inverting input of the amplifier will have to go more positive than this before the comparator will change states. Likewise, if you assume V_{OUT} is −13 V, the noninverting input will be at about −13 mV, so the voltage on the inverting input of the amplifier will have to go below this to change the state of the output. The hysteresis of this comparator is 26 mV.

NONINVERTING AMPLIFIER OP-AMP CIRCUIT

When operating in open-loop mode (no feedback to the inverting input), an op amp has a very high, but unpredictable, gain. This is acceptable for use as a comparator, but not for use as a predictable amplifier. Figure 10-1*d* shows one way negative feedback is added to an op amp to produce an amplifier with stable, predictable gain. First of all, notice that the input signal in this circuit is applied to the noninverting input, so the output will be in phase with the input. Second, note that a fraction of the output signal is fed back to the inverting input. Now, here's how this works.

To start, assume that V_{IN} is 0 V, V_{OUT} is 0 V, and the voltage on the inverting input is 0. Now, suppose that you apply a +0.01-V dc signal to the noninverting input. Since the 0.1-V difference between the two inputs will be amplified by 100,000, the output will head toward +100 V as fast as it can. However, as the output goes positive, some of the output voltage will be fed back to the inverting input through the resistor divider. This feedback to the inverting input will decrease the difference in voltage between the two inputs. To make a long story short, the circuit quickly reaches a predictable balance point where the voltage on the inverting input, V_F, is very, very close to the voltage on the noninverting input, V_{IN}. For a 1.0-V dc output, this equilibrium voltage difference might be about 10 μV. If you assume that the voltages on the two inputs are equal, then predicting the output voltage for a given input voltage is simply a voltage-divider problem. $V_{OUT} = V_{IN}$ (R1 + R2)/R1. If R2 = 99 kΩ and R1 = 1 kΩ, then $V_{OUT} = V_{IN} \times 100$. For a 0.01-V input signal, the output voltage will be 1.00 V.

The voltage gain of a circuit with feedback is called its *closed-loop gain*. The closed-loop gain, A_{VCL}, for this circuit is equal to the simple resistor ratio, (R1 + R2)/R1.

To see another advantage of feeding some of the output signal back to the inverting input, let's see what happens when the load connected to the output of the op amp changes and draws more current from the output. The output voltage will temporarily drop because of the increased load. Part of this voltage drop will be fed back to the inverting input, increasing the difference in voltage between the two inputs. The increased difference will cause the op amp to drive its output harder to correct for the increased load. The feedback then causes the op amp to at least partially compensate for the increased load on its output.

Feedback which causes an amplifier to oppose a change on its output is called *negative feedback*. Because of the negative feedback, the op amp will work day and night to keep its output stabilized and its two inputs at nearly the same voltage! This is probably the most important point you need to know to analyze or troubleshoot an op-amp circuit with negative feedback. Draw a box around this point in your mind so you don't forget it.

The noninverting circuit we have just discussed is used mostly as a *buffer* because it has a very high *input impedance*, Z_{IN}. This means that it will not load down a sensor or some other device you connect to its input. If it uses a bipolar-transistor input op amp, the circuit in Figure 10-1*d* will have an input impedance greater than 100 MΩ. If a FET input op amp such as the National LF356 is used, the input impedance will be about 10^{12} Ω.

INVERTING AMPLIFIER OP-AMP CIRCUIT

Figure 10-1*e* shows a somewhat more versatile amplifier circuit using negative feedback. Note that in this circuit, the noninverting input is tied to ground with a resistor, and the signal you want to amplify is applied to the inverting input through a resistor. Since the signal is applied to the inverting input, the output signal will be 180° out of phase with the input signal. For this circuit, resistor Rf supplies the negative feedback which keeps the two inputs at nearly the same voltage. Since the noninverting input is tied to ground, the op amp will sink or source whatever current is needed to hold the inverting input also at zero volts. Because the op amp holds the inverting input at zero volts, this node is referred to as a *virtual ground*.

The voltage gain of this circuit is also determined by the ratio of two resistors. The A_{VCL} for this circuit at low frequencies is equal to −Rf/R1. You can derive this for yourself by just thinking of the two resistors as a voltage divider with V_{IN} at one end, 0 V in the middle, and V_{OUT} on the other end. If V_{IN} is positive, then V_{OUT} must be negative because current cannot flow from positive to ground to positive again. The minus sign in the gain expression is another way of indicating that the output is inverted from the input. The input impedance Z_{IN} of this circuit is approximately R1 because the amplifier end of this resistor is held at 0 V by the op amp.

One additional characteristic of op-amp circuits that we need to refresh in your mind before going on to other op-amp circuits is *gain-bandwidth product*. As we indicated previously, an op amp may have an open-loop dc gain of 100,000 or more. At higher frequencies, the gain decreases until, at some frequency, the open-loop gain drops to 1. Figure 10-2*a* shows an open-loop voltage gain versus frequency graph for a common op amp such as a 741. The frequency at which the gain is 1 is referred to on data sheets as the *unity-gain bandwidth* or the *gain-bandwidth product*. A common value for this is 1 MHz. The bandwidth of an amplifier circuit with negative feedback times the low-frequency closed-loop gain will be equal to this value. For example, if an op amp with a gain-bandwidth product of 1 MHz is used to build an amplifier circuit with a closed-loop gain of 100, the bandwidth of the circuit, f_c, will be about 1 MHz/100 or 10 kHz, as shown in Figure 10-2*b*.

The point here is that the gain-bandwidth product of the op amp limits the maximum frequency that an amplifier circuit can amplify.

OP-AMP ADDER CIRCUIT

Figure 10-1*f* shows a commonly used variation of the inverting amplifier described in the previous section. This circuit adds together or mixes two or more input signals. Here's how it works.

Remember from the previous discussion that in an inverting circuit, the op amp holds the inverting input at 0 V or virtual ground. The current through each of the input resistors will be the same as if it were connected

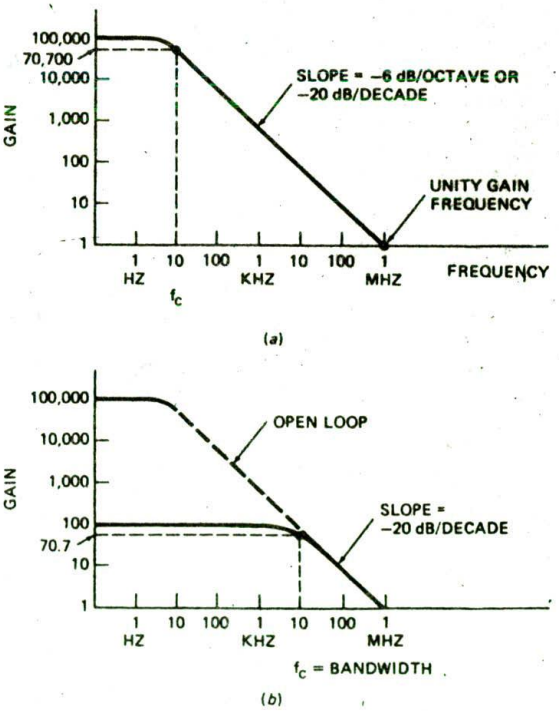


FIGURE 10-2 (*a*) Open-loop gain versus frequency response of 741 op amp. (*b*) Gain versus frequency response of 741 op-amp circuit with closed-loop gain of 100.

to ground. Input voltage V_1 produces a current through R1 to this point, and input voltage V_2 causes a current through R2 to this point. The two currents add together at the virtual ground. In this circuit the virtual ground is often called the *summing point*. The op amp pulls the sum of the two currents through resistor Rf to hold the inverting input at 0 V. The left end of Rf is at 0 V, so the output voltage is the voltage across Rf. This is equal to the sum of the currents times the value of Rf, or $V_1/R1 + V_2/R2 \times Rf$. A circuit such as this is used to "mix" audio signals and to sum binary-weighted currents in a D/A converter. Although the circuit in Figure 10-1*f* shows only two inputs, an adder can have any number of inputs.

SIMPLE DIFFERENTIAL-INPUT AMPLIFIER CIRCUIT

As we show later, many sensors have two output signal lines with a dc voltage of several volts on each signal line. The dc voltage present on both signal leads is referred to as a *common-mode voltage*. The actual signal you need to amplify from these sensors is a difference in voltage of a few millivolts between the two signal lines. If you try to use a standard inverting or noninverting amplifier circuit to do this, the large dc voltage will be amplified along with the small difference voltage you need to amplify. Figure 10-1*g* shows a simple circuit which, for the most part, solves this problem without using coupling capacitors to block the dc. The analysis of this circuit is beyond the space we have here, but basically the resistors on the noninverting input hold this input at a voltage near the common-mode dc voltage. The amplifier holds the inverting input at the same voltage. If the resistors are matched carefully, the result is that only the difference in voltage between V_2 and V_1 will be amplified. The output signal will consist of only the amplified difference in voltage between the input signals. We say that the common-mode signal has been *rejected*.

AN INSTRUMENTATION AMPLIFIER CIRCUIT

Figure 10-1*h* shows an op-amp circuit used in applications that need a greater rejection of the common-mode signal than is provided by the simple differential circuit in Figure 10-1*g*. The first two op amps in this circuit buffer the differential signals and give some amplification. The output op amp removes the common-mode voltage and provides further amplification. Another way of describing the function of the output op amp is to say that it converts the signal from a differential signal to a single-ended signal. Instrumentation amplifier circuits such as this are available in single packages.

AN OP-AMP INTEGRATOR CIRCUIT

Figure 10-1*i* shows an op-amp circuit that can be used to produce linear voltage ramps. A dc voltage applied to the input of this circuit will cause a constant current of $V_{IN}/R1$ to flow into the virtual-ground point. This current flows onto one plate of the capacitor. In order to hold the inverting input at ground, the op-amp output must pull the same current from the other plate of the capacitor. The capacitor then is getting charged by the constant current $V_{IN}/R1$. Basic physics tells you that the voltage across a capacitor being charged by a constant current is a *linear ramp*. Note that because of the inverting amplifier connection, a positive input voltage will cause the output to ramp negative. Also note that some provision must be made to prevent the amplifier output from ramping into *saturation*.

The circuit is called an *integrator* because it produces an output voltage proportional to the integral, or "sum," of the current produced by an input voltage over a period of time. The waveforms in Figure 10-1*i* show the circuit response for a pulse-input signal.

AN OP-AMP DIFFERENTIATOR CIRCUIT

Figure 10-1*j* shows an op-amp circuit which produces an output signal proportional to the rate of change of the input signal. With the input voltage to this circuit at 0 or some other steady dc voltage, the output will be at 0. If a new voltage is applied to the input, the voltage across the capacitor cannot change instantly, so the inverting input will be pulled away from 0 V. This will cause the op amp to drive its output in a direction to charge the capacitor and pull the inverting input back to zero. The waveforms in Figure 10-1*j* show the circuit response for a pulse-input signal. The time required for the output to return to zero is determined by the time constant of R1 and C.

OP-AMP ACTIVE FILTERS

In many control applications, we need to filter out unwanted low-frequency or high-frequency noise from the signals read in from sensors. This could be done with simple RC filters, but *active filters* using op amps give much better control over filter characteristics. There are many different filter configurations using op amps. The main points we want to refresh here are the meanings of the terms *low-pass* filter, *high-pass* filter, and *bandpass* filter and how you identify the type when you find one in a circuit you are analyzing.

A low-pass filter amplifies or passes through low frequencies, but at some frequency determined by circuit values, the output of the filter starts to decrease. The frequency at which the output is down to 0.707 of the low-frequency value is called the *critical frequency* or *breakpoint*. Figure 10-3*a* shows a graph of gain versus frequency for a low-pass filter with the critical frequency, f_C, labeled. Note that above the critical frequency the gain drops off rapidly. For a first-order filter such as a single R and C, the gain decreases by a factor of 10 for each increase of 10 times in frequency (-20 dB/decade). For a second-order filter, the gain decreases by a factor of 100 for each increase of 10 times in frequency.

Figure 10-1*k* shows a common op-amp circuit for a second-order low-pass filter. The way you recognize this as a low-pass filter is to look for a dc path from the input to the noninverting input of the amplifier. If the dc path is present, as it is in Figure 10-1*k*, you know that the amplifier can amplify dc and low frequencies. Therefore, it is a low-pass filter with a response such as that shown in Figure 10-3*a*.

For contrast, look at the circuit for the second-order high-pass filter in Figure 10-1*k*. Note that in this circuit,

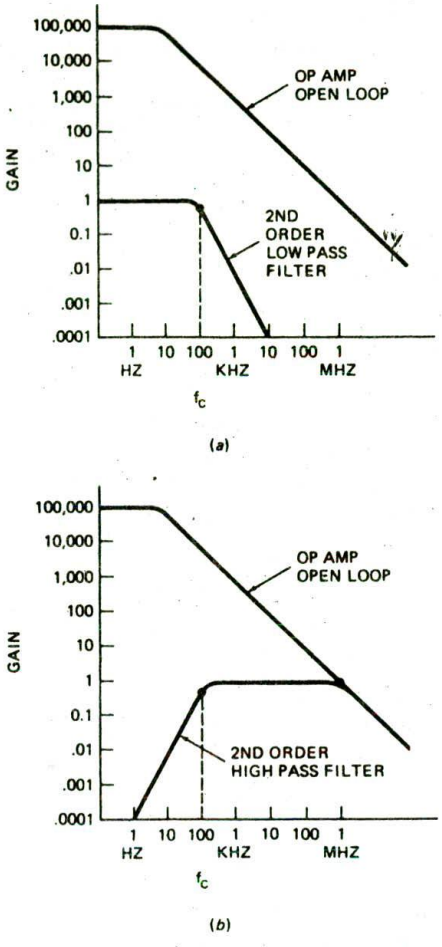


FIGURE 10-3 Gain versus frequency response for second-order low-pass and high-pass filters. (*a*) Low-pass. (*b*) High-pass.

the dc component of an input signal cannot reach the noninverting input, because of the two capacitors in series with that input. Therefore, this circuit will not amplify dc and low-frequency signals. Figure 10-3*b* shows the graph of gain versus frequency for a high-pass filter such as this. Note that the gain-bandwidth product of the op amp limits the high-frequency response of the circuit.

For the low-pass circuit in Figure 10-1*k*, the gain for the flat part of the response curve is 1, or unity, because the output is fed back directly to the inverting input. At the critical frequency, f_c, the gain will be 0.707, and above 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 R1 and R2 are equal and that the value of C1 is twice the value of C2. R3 is simply a damping resistor. The positive feedback supplied by C1 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.

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 R2 is twice the value of R1, the critical frequency is determined by the formula shown next to Figure 10-1*l*. Again, R3 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 Figure 10-4*a*. A glass window allows light to fall on a zigzag

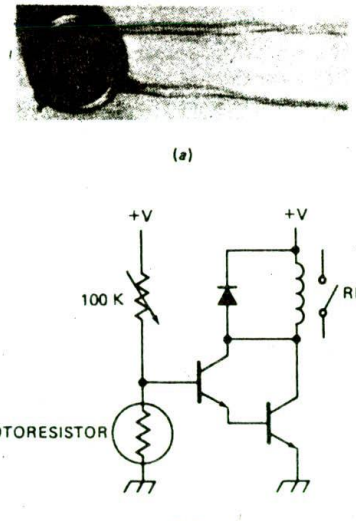


FIGURE 10-4 (*a*) Cadmium sulfide photocell. (*Clairex Electronics*) (*b*) Light-controller relay circuit using a photocell.

pattern of cadmium sulfide or cadmium selenide whose resistance depends on the amount of light present. The resistance of the CL905 varies from about 15 MΩ when in the dark to about 15 kΩ when in a bright light. Photoresistors such as this do not have a very fast response time and are not stable with temperature, but they are inexpensive, durable, and sensitive. For these reasons, they are usually used in applications where the light measurement need not be precise. The devices placed on top of streetlights to turn them on when it gets dark, for example, contain a photoresistor, a transistor driver, and a mechanical relay, as shown in Figure 10-4*b*. As it gets dark, the resistance of the photoresistor goes up. This increases the voltage on the base of the transistor until, at some point, it turns on. This turns on the transistor driving the relay, which in turn switches on the lamp.

Another device used to sense the amount of light present is a photodiode. If light is allowed to fall on the junction of a specially constructed silicon diode, the reverse leakage current of the diode increases linearly as the amount of light falling on it increases. A circuit such as that shown in Figure 10-5 can be used to convert this small leakage current to a proportional voltage. Note that in this circuit a negative reference voltage is applied to the noninverting input of the amplifier. The op amp will then produce this same voltage on its inverting input, reverse-biasing the photodiode. The op amp will pull the photodiode leakage current through Rf to produce a proportional voltage on the output of the amplifier. For a typical photodiode such as the HP 5082-4203 shown, the reverse leakage current varies from near 0 μA to about 100 μA, so with the 100-kΩ Rf, an output voltage of about 0 to 10 V will be produced. The circuit will work without any reverse bias on the diode, but with the reverse bias, the diode responds faster to changes in light. An LM356 FET input amplifier is used here because it does not require an input bias current.

A photodiode circuit such as this might be used to determine the amount of smoke being emitted from a smokestack. To do this, a gallium arsenide infrared LED is put on one side of the smokestack, and the photodetector circuit is put on the other. Since smoke absorbs light, the amount of light arriving at the photodetector is a measure of the amount of smoke present. An infrared LED is used here because the photodiode is most sensitive to light wavelengths in the infrared region.

Still another useful light-sensitive device is a solar cell. Common solar cells are simply large, very heavily doped silicon PN junctions. Light shining on the solar cell causes a reverse current to flow, just as in the photodiode. Because of the large area and the heavy doping in the solar cell, however, the current produced is milliamperes rather than microamperes. The cell functions as a light-powered battery. Solar cells can be connected in a series-parallel array to produce a solar power supply.

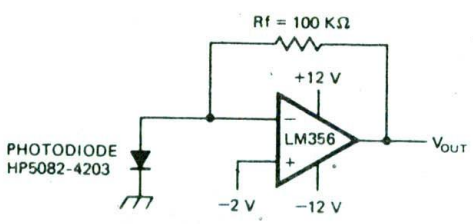


FIGURE 10-5 Photodiode circuit to measure light intensity.

Light meters in cameras, photographic enlargers, and our printed-circuit-board-making machine use solar cells. The current from the solar cell is a linear function of the amount of light falling on the cell. A circuit such as the one in Figure 10-5 can be used to convert the output current to a proportional voltage. Because of the larger output current, the value of R_f is decreased to a much smaller value, depending on the output current of the cell. The noninverting input of the amplifier is connected to ground because reverse biasing is not needed with solar cells. The frequency response to light (spectral response) of solar cells has been tailored to match the output of the sun. Therefore, they are ideal in photographic applications where we want a signal proportional to the total light from the sun.

Temperature Sensors

Again, there are many types of temperature sensors. The four types we discuss in some detail here are semiconductor devices, thermocouples, RTDs, and thermistors.

SEMICONDUCTOR TEMPERATURE SENSORS

The two main types of semiconductor temperature sensors are temperature-sensitive voltage sources and temperature-sensitive current sources. An example of the first type is the National LM35, which we show the circuit connections for in Figure 10-6*a*. The voltage output from this circuit increases by 10 mV for each degree Celsius that its temperature is increased. If the output is connected to a negative reference voltage, V_S, as shown, the sensor will give a meaningful output for a temperature range of −55 to +150° C. The output is adjusted to 0 V for 0° C. The output voltage can be amplified to give the voltage range you need for a particular application. In a later section of this chapter, we show another circuit using the LM35 temperature sensor. The accuracy of this device is about 1° C.

Another common semiconductor temperature sensor is a temperature-dependent current source, such as the Analog Devices AD590. The AD590 produces a current of 1 μA/°K. Figure 10-6*b* shows a circuit which converts this current to a proportional voltage. In this circuit the current from the sensor, I_T, is passed through an approximately 1-kΩ resistor to ground. This produces a voltage which changes by 1mV/°K. The AD580 in the circuit is a precision voltage reference used to produce a reference voltage of 273.2 mV. With this voltage applied to the inverting input of the amplifier, the amplifier output will be at zero volts for 0° C. The advantage of a current-source sensor is that voltage drops in long

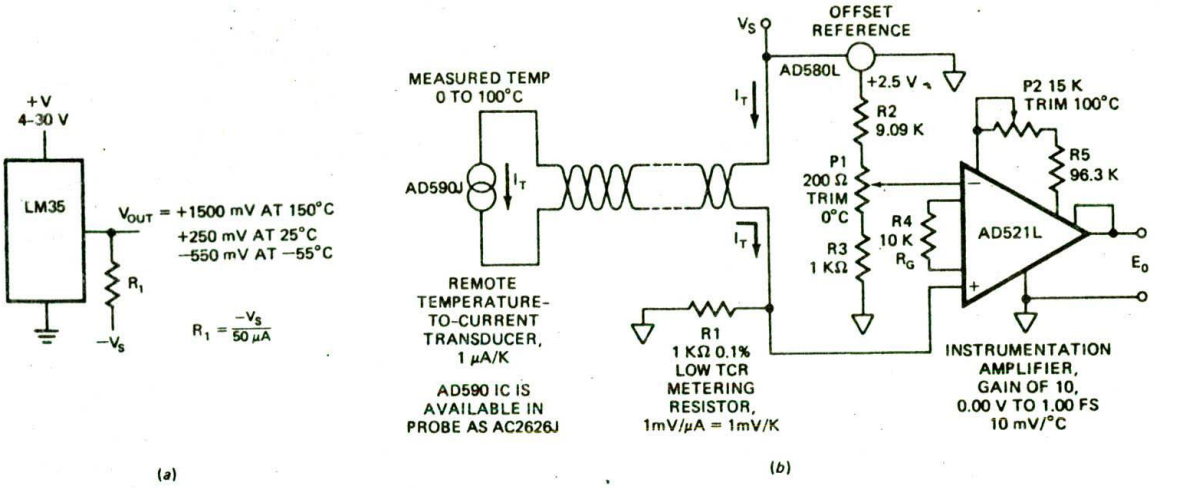


FIGURE 10-6 Semiconductor temperature-sensor circuits. (*a*) LM35 temperature-dependent voltage source. (*b*) AD590 temperature-dependent current source. (*Analog Devices Incorporated*)

connecting wires do not have any effect on the output value. If the gain and offset are carefully adjusted, the accuracy of the circuit in Figure 10-6*b* is ±1° C using an AD590K part.

THERMOCOUPLES

Whenever two different metals are put in contact, a small voltage is produced between them. The voltage developed depends on the type of metals used and the temperature. Depending on the metals, the developed voltage increases between 7 and 75 μV for each degree Celsius increase in temperature. Different combinations of metals are useful for measuring different temperature ranges. A thermocouple junction made of iron and constantan, commonly called a type J thermocouple, has a useful temperature range of about −184 to +760° C. A junction of platinum and an alloy of platinum and 13 percent rhodium has a useful range of 0 to about 1600° C. Thermocouples can be made small, rugged, and stable; however, they have three major problems which must be overcome.

The first of these is the fact that the output is very small and must be amplified a great deal to bring it up into range where it can, for example, drive an A/D converter.

Second, as shown in Figure 10-7, a reference junction made of the same metals must be connected in series with the junction being used to make the measurement. Note that the reference junction is connected in the reverse direction from the measuring junction. This is done so that the output connecting wires are both constantan. The thermocouples formed by connecting these wires to the copper wires going to the amplifier will then cancel out. The input voltage to the amplifier will be the difference between the voltages across the two thermocouples. If we simply amplify this voltage, however, there is a problem if the temperature of both thermocouples is changing. The problem is that it is impossible to tell which thermocouple caused a change in output voltage. One cure for this is to put the reference junction in an ice bath or a small oven to hold it at a constant temperature. This solution is usually inconvenient, so instead a circuit such as that in Figure 10-7 is used to compensate electronically for changes in the temperature of the reference junction.

As we discussed in a previous section, the AD590 shown here produces a current proportional to its temperature. The AD590 is attached to the reference thermocouple so that they are both at the same temperature. The current from the AD590, when passed through the resistor network, produces a voltage which compensates for changes in the reference thermocouple with temperature. The signal to the amplifier then is propor-

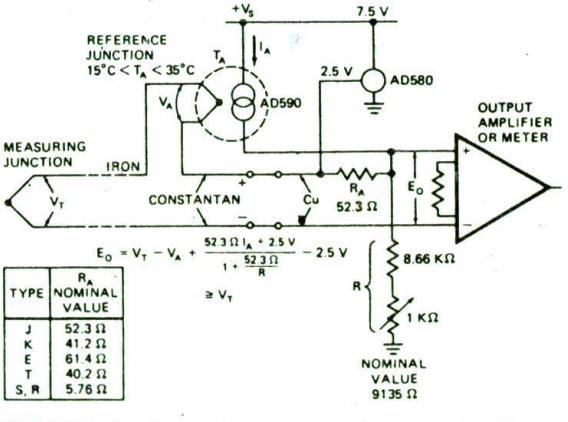


FIGURE 10-7 Circuit showing amplification and cold-junction compensation for thermocouple. (*Analog Devices Incorporated*)

tional only to changes in the sensor thermocouple. Canceling out the effects of ambient temperature variations on the reference junction is referred to as *cold-junction compensation*. The table in Figure 10-7 shows the values of R_A which will provide cold-junction compensation for common types of thermocouples. An instrumentation amplifier such as that in Figure 10-1*h* is usually used for this application.

The third problem with thermocouples is that their output voltages do not change linearly with temperature. This can be corrected with analog circuitry which changes the gain of an amplifier according to the value of the signal. However, when a thermocouple is used with a microcomputer-based instrument, the correction can be easily done using a lookup table in ROM. An A/D converter converts the voltage from the thermocouple to a digital value. The digital value is then used as a pointer to a ROM location which contains the correct temperature for that reading.

RTDS AND THERMISTORS

Resistance temperature detectors (RTDs) and thermal sensitive resistors (thermistors) are two other commonly used types of temperature sensors. Both of these types are essentially resistors which change value with a change in temperature. RTDs consist of a wire or a thin film of platinum or a nickel wire. The response of RTDs is nonlinear, but they have excellent stability and repeatability. Therefore, they are often used in applications where very precise temperature measurement is needed. RTDs are useful for measures in the range of −250 to +850° C. A circuit such as that in Figure 10-8 can be used to convert the change in resistance of the RTD to a proportional voltage. Op amp A1 in this circuit produces a precise reference voltage of −6.25 V. This voltage produces a precise current at the inverting input of A2. Op amp A2 pulls this current through the RTD to produce a voltage proportional to the resistance of the RTD. The resistance of an RTD increases with an increase in temperature.

Thermistors consist of semiconductor material whose resistance decreases nonlinearly with temperature. Devices with 25° C resistance of tens of ohms to millions of ohms are available for different applications. Thermistors are relatively inexpensive, have very fast response times, and are useful in applications where precise measurement is not required. A circuit similar to that in Figure 10-8 can be used to produce a voltage proportional to the resistance of the thermistor.

Force and Pressure Transducers

To convert force or pressure (force/area) to a proportional electrical signal, the most common methods use *strain gages* or *linear variable differential transformers* (LVDTs). Both of these methods involve moving something. This why we refer to them as *transducers* rather than as sensors. Here's how strain gages work.

STRAIN GAGES AND LOAD CELLS

A strain gage is a small resistor whose value changes when its length is changed. It may be made of thin wire, thin foil, or semiconductor material. Figure 10-9*a* shows a simple setup for measuring force or weight with strain gages. One end of a piece of spring steel is attached to a fixed surface. A strain gage is glued on the top of the flexible bar. The force or weight to be measured is applied to the unattached end of the bar. As the applied force bends the bar, the strain gage is stretched, increasing its resistance. Since the amount that the bar is bent is directly proportional to the applied force, the change in resistance will be proportional to the applied force. If a current is passed through the strain gage, then the change in voltage across the strain gage will be proportional to the applied force.

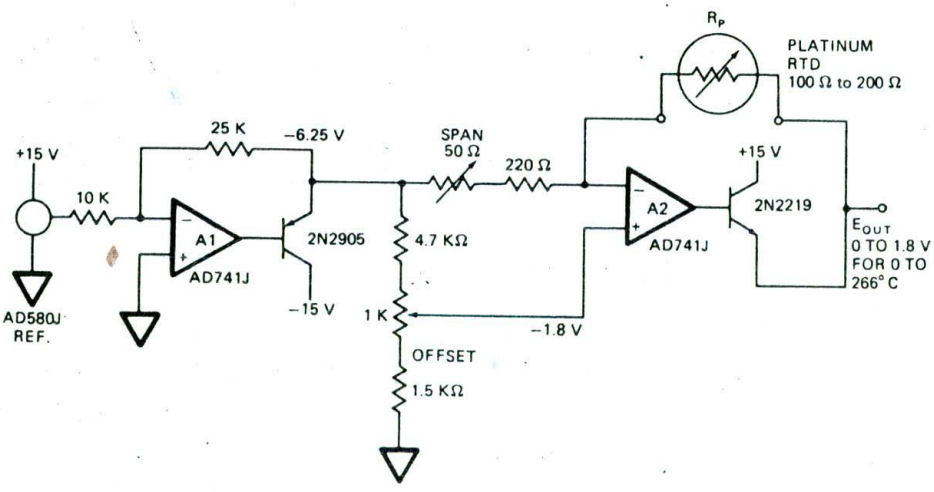


FIGURE 10-8 100-Ω RTD connected to perform temperature measurements in the range 0°C to 266°C. (*Analog Devices Incorporated*)

Unfortunately, the resistance of the strain-gage element also changes with temperature. To compensate for this problem, two strain-gage elements mounted at right angles, as shown in Figure 10-9*b*, are often used. Both of the elements will change resistance with temperature, but only element A will change resistance appreciably with applied force. When these two elements are connected in a balanced-bridge configuration, as shown in Figure 10-9*c*, any change in the resistance of the elements due to temperature will have no effect on the differential output of the bridge. However, as force is applied, the resistance of the element under strain will change and produce a small differential output voltage. The full-scale differential output voltage is typically 2 or 3 mV for each volt of excitation voltage applied to the top of the bridge. For example, if 10 V is applied to the top of the bridge, the full-load output voltage will be 20 or 30 mV. This small signal can be amplified with a differential amplifier or an instrumentation amplifier.

Strain-gage bridges are used in many different forms to measure many different types of force and pressure. If the strain-gage bridge is connected to a bendable beam structure, as shown in Figure 10-9*a*, the result is called a *load cell* and is used to measure weight. Figure 10-10 shows a 10-lb load cell that might be used in a microprocessor-controlled delicatessen scale or postal scale. Larger versions can be used to weigh barrels being filled or even trucks.

If a strain-gage bridge is mounted on a movable diaphragm in a threaded housing, the output of the bridge will be proportional to the pressure applied to the diaphragm. If a vacuum is present on one side of the diaphragm, then the value read out will be a measure of the absolute pressure. If one side of the diaphragm is open, then the output will be a measure of the pressure relative to atmospheric pressure. If the two sides of the diaphragm are connected to two different pressure sources, then the output will be a measure of the differential pressure between the two sides. Figure 10-11 shows a Sensym LX1804GBZ pressure transducer which measures pressures in the range of 0 to 15 lb/in^2. A transducer such as this might be used to measure blood pressure in a microcomputer-based medical instrument.

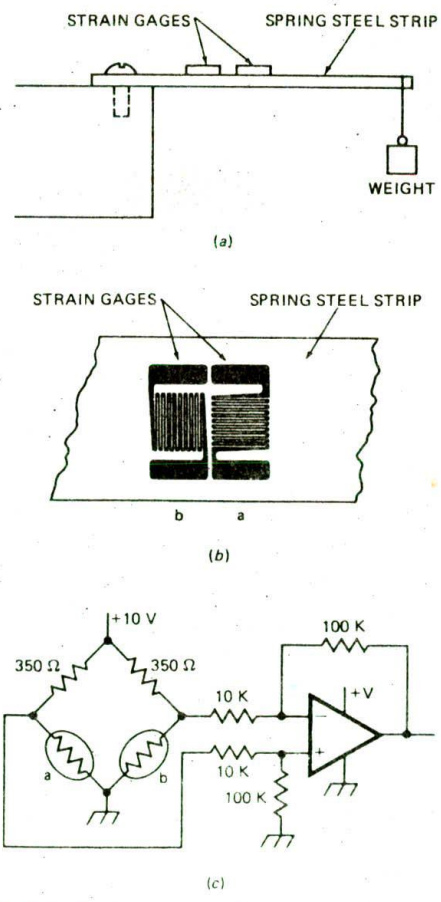


FIGURE 10-9 Strain gages used to measure force. (*a*) Side view. (*b*) Top view (expanded). (*c*) Circuit connections.

LINEAR VARIABLE DIFFERENTIAL TRANSFORMERS

An *LVDT* is another type of transducer often used to measure force, pressure, or position. Figure 10-12 shows the basic structure of an LVDT. It consists of three coils of wire wound on the same form and a movable iron core. An ac excitation signal of perhaps 20 kHz is applied to the primary. The secondaries are connected such that the voltage induced in one opposes the voltage induced in the other. If the core is centered, then the induced voltages are equal and cancel each other, so there is no net output voltage. If the coil is moved off center, coupling to one secondary coil will be stronger, so that the coil will produce a greater output voltage. The result will be a net output voltage. The phase relationship between the output signal and the input signal is an indication of which direction the core moved from the center position. The amplitude of the output signal is linearly proportional to how far the core moves from the center position.

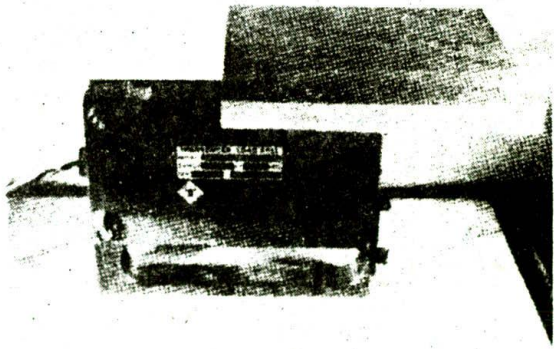


FIGURE 10-10 Photograph of load-cell transducer used to measure weight. (*Transducers, Incorporated*)

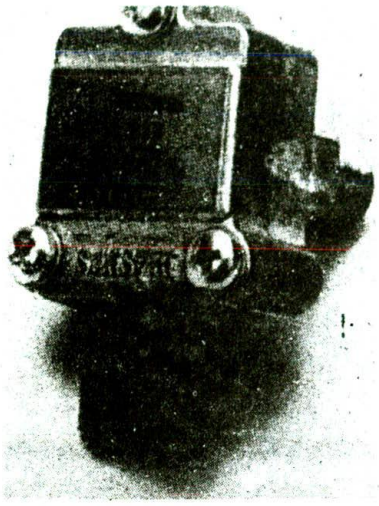


FIGURE 10-11 LX1804GBZ pressure transducer. (*Sensym, Incorporated*)

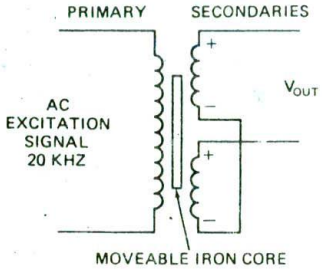


FIGURE 10-12 Linear variable differential transformer (LVDT) structure.

An LVDT can be used directly in this form to measure displacement or position. If you add a spring so that a force is required to move the core, then the voltage out of the LVDT will be proportional to the force applied to the core. In this form, the LVDT can be used in a load cell for an electronic scale. Likewise, if a spring is added and the core of the LVDT is attached to a diaphragm in a threaded housing, the output from the LVDT will be proportional to the pressure exerted on the diaphragm. We do not have the space here to show the ac-interface circuitry required for an LVDT.

Flow Sensors

If we are going to control the flow rate of some material in our electronics factory, we need to be able to measure it. Depending on the material, flow rate, and temperature, we use different methods.

One method used is to put a paddle wheel in the flow, as shown in Figure 10-13*a*. The rate at which the paddle wheel turns is proportional to the rate of flow of a liquid or gas. An optical encoder can be attached to the shaft of the paddle wheel to produce digital information as to how fast the paddle wheel is turning.

A second common method of measuring flow is with a *differential pressure transducer*, as shown in Figure 10-13*b*. A wire mesh or screen is put in the pipe to create some resistance. Flow through this resistance produces a difference in pressure between the two sides of the screen. The pressure transducer gives an output proportional to the difference in pressure between the two sides of the resistance. In the same way that the voltage across an electrical resistor is proportional to the flow of current through the resistor, the output of the pressure transducer is proportional to the flow of a liquid or gas through the pipe.

Other Sensors

As we mentioned previously, the number of different types of sensors is very large. In addition to the types we have discussed, there are sensors to measure pH, concentration of various gases, thickness of materials, presence of an object (proximity), and just about anything else you might want to measure. Often you can use commonly available transducers in creative ways to solve a particular application problem you have. Suppose, for example, that you need to accurately determine the level of a liquid in a large tank. To do this, you could install a pressure transducer at the bottom of the tank. The pressure in a liquid is proportional to the height of the liquid in the tank, so you can easily convert a pressure reading to the desired liquid height. The point here is to check out what is available and then be creative.

4- to 20-mA Current Loops

In the preceding discussions, we showed how op amps can be used to convert output signals to voltages in a range that can be applied to the input of an A/D

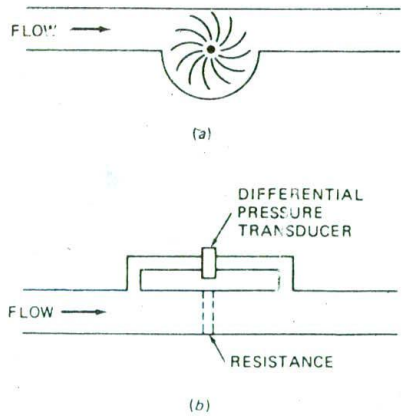


FIGURE 10-13 Flow sensors. (*a*) Paddle wheel. (*b*) Differential pressure.

converter. In many industrial applications where the sensor is a long distance from the A/D converter, however, the signals from the sensors or transducers are converted to currents instead of voltages. Sending a signal as a current has the advantages that the signal amplitude is not affected by resistance, induced-voltage noise, or voltage drops in a long connecting line. A common range of currents used to represent analog signals in industrial environments is 4 to 20 mA. A current of 4 mA represents a zero output, and a current of 20 mA represents the full-scale value. The reason the current range is offset from zero is so that a current of zero is left to represent an open circuit. At the receiving end of the line, a resistor or a simple op-amp circuit is used to convert the current to a proportional voltage which can be applied to the input of the A/D converter.

D/A CONVERTER OPERATION, INTERFACING, AND APPLICATIONS

In the previous sections of this chapter we have discussed how we use sensors to get electrical signals proportional to pressure, temperature, etc, and how we use op amps to amplify and filter these electrical signals. The next logical step would be to show you how to use an A/D converter to get these signals into digital form that a microcomputer can work with. However, since D/A converters are simpler and since several types of A/D converters have D/As as part of their circuitry, we will discuss D/As first.

D/A Converter Operation and Specifications

OPERATION

The purpose of a digital-to-analog converter is to convert a binary word to a proportional current or voltage. To see how this is done, let's look at the simple 4-input adder circuit in Figure 10-14.

Since the noninverting input of the op amp is grounded, the op amp will work day and night to hold the inverting input also at 0 V. Remember that the inverting input in this circuit is referred to as the summing point. When one of the switches is closed, a current will flow from -5 V (V_{REF}) through that resistor to the summing point. The op amp will pull the current on through the feedback resistor to produce a proportional output voltage. If you close switch D0, for example, a current of 0.05 mA will flow into the summing point. In order to pull this current through the feedback resistor, the op amp must put a voltage of 0.05 mA × 10 kΩ or 0.5 V on its output. If you also close switch D1, it will send another 0.1 mA into the summing point. In order to pull the sum of the currents through the feedback resistor, the op amp has to output a voltage of 0.15 mA × 10 kΩ or 1.5 V.

The point here is that the binary-weighted resistors produce binary-weighted currents which are summed by the op amp to produce a proportional output voltage. The binary word applied to the switches produces a proportional output voltage. Technically the output voltage is "digital" because it can only have certain fixed values, just as the display on a digital voltmeter can. However, the output simulates an analog signal, so we refer to it as analog. Switch D3 in Figure 10-14 represents the most significant bit because closing it produces the largest current. Note that since V_{REF} is negative, the output will go positive as switches are closed.

As you see here, the heart of a D/A converter is a set of binary-weighted current sources which can be switched on or off according to a binary word applied to its inputs. Since these current sources are usually inside an IC, we don't need to discuss the different ways the binary-weighted currents can be produced. The op-amp circuit in Figure 10-14 converts the sum of the currents to a proportional voltage.

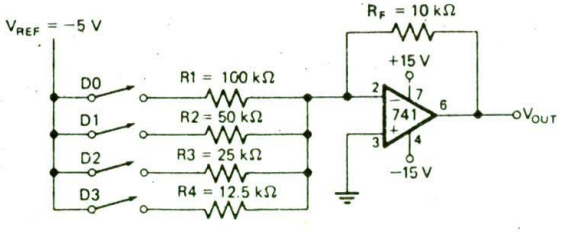
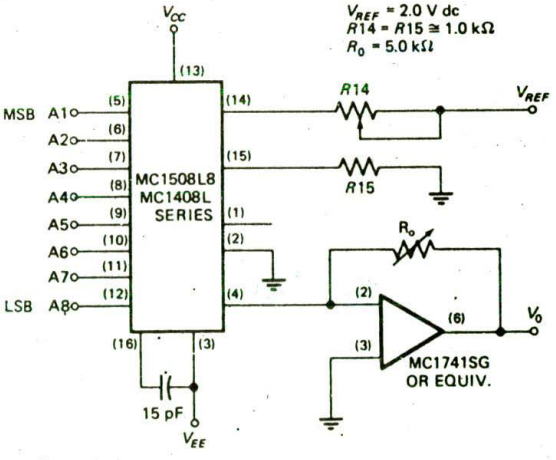


FIGURE 10-14 Simple 4-bit D/A converter.

D/A CHARACTERISTICS AND SPECIFICATIONS

Figure 10-15 shows the circuit for an inexpensive IC D/A converter with an op-amp circuit as a current-to-



Theoretical V_0

$$V_0 = \frac{V_{REF}}{R14}(R_0)\left\{\frac{A1}{2}+\frac{A2}{4}+\frac{A3}{8}+\frac{A4}{16}+\frac{A5}{32}+\frac{A6}{64}+\frac{A7}{128}+\frac{A8}{256}\right\}$$

ADJUST V_{REF}, $R14$ OR R_0 SO THAT V_0 WITH ALL DIGITAL INPUTS AT HIGH LEVEL IS EQUAL TO 9.961 V

$$V_0 = \frac{2\text{ V}}{1\text{ k}\Omega}(5\text{ k}\Omega)\left\{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\frac{1}{128}+\frac{1}{256}\right\}$$

$$= 10\text{ V}\left\{\frac{255}{256}\right\} = 9.961\text{ V}$$

FIGURE 10-15 Motorola MC1408 8-bit D/A with current-to-voltage converter.

voltage converter. We will use this circuit for our discussion of D/A characteristics.

The first characteristic of a D/A converter to consider is *resolution*. This is determined by the number of bits in the input binary word. A converter with 8 binary inputs, such as the one in Figure 10-15, has 2^8 or 256 possible output levels, so its resolution is 1 part in 256. As another example, a 12-bit converter has a resolution of 1 part in 2^{12} or 4096. Resolution is sometimes expressed as a percentage. The resolution of an 8-bit converter expressed as a percentage is (1/256) × 100 percent or about 0.39 percent.

The next D/A characteristic to determine is the *full-scale output voltage*. For the converter in Figure 10-15, the current for all the switches is supplied by V_{REF} through R14. The current output from pin 4 of the D/A is pulled through R_o to produce the output voltage. The formula for the output voltage is shown under the circuit in Figure 10-15. In the equation the term A1, for example, represents the condition of the switch for that bit. If a switch is closed, allowing a current to flow, put a 1 in that bit. If a switch is open, put a 0 in that bit. As we also show in Figure 10-15, if all the switches are closed, the output will be 10 V × (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 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 ±½ 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 ±½ 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 overshoot the correct value and "ring" for a while before finally settling down to the correct value. The time the output takes to get within ±½ 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 Figure 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 Figure 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

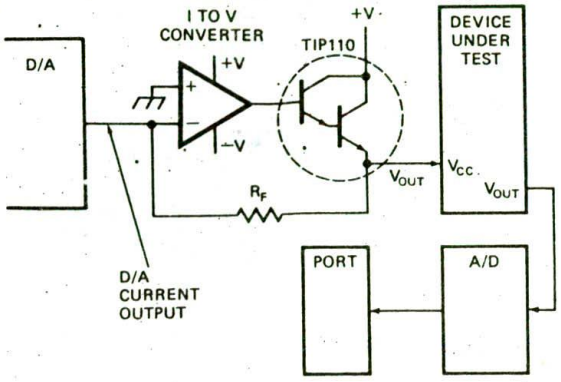


FIGURE 10-16 High-power buffer for D/A output.

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.

So far we have talked about using an 8-bit D/A with a microprocessor. Interfacing an 8-bit converter involves simply connecting the inputs of the converter to an output port or, for some D/As, simply connecting the inputs to the buses as you would a port device. Now, suppose that for some application you need 12 bits of resolution, so you need to interface a 12-bit converter. If you are working with a system which has an 8-bit data bus, your first thought might be to connect the lower 8 inputs of the 12-bit converter to one output port and the upper four inputs to another port. You could send the lower 8 bits with one write operation and the upper 4 bits with another write operation. However, the time between the two writes introduces a potential problem in this approach.

Suppose, for example, that you want to change the output of a 12-bit converter from 0000 1111 1111 to 0001 0000 0000. When you write the lower 8 bits, the output will go from 0000 1111 1111 to 0000 0000 0000. When you write the upper 4 bits, the output will go back up to the desired 0001 0000 0000. The point here is that for the time between the two writes the output will go to an unwanted value. In many systems this could be disastrous. The cure for this problem is to put latches on the input lines. The latches can be loaded separately and then strobed together to pass all 12 bits to the D/A converter at the same time.

Many currently available D/A converters contain built-in latches to make this easier. Figure 10-17*a* shows a block diagram of the National DAC1230- and DAC1208-type 12-bit converters. Note the internal latches and the register. The DAC1230 series of parts has the upper 4 input bits connected to the lower 4 bits so that the 12 bits can be written with two write operations from an 8-bit port or data bus such as that of the 8088 microprocessor. The DAC1208 series of parts has the upper 4 data inputs available separately so they can be connected directly to the bus in a system which has a 16-bit data bus, as shown in Figure 10-17*a*. If, for example, you want to connect a DAC1208 converter to

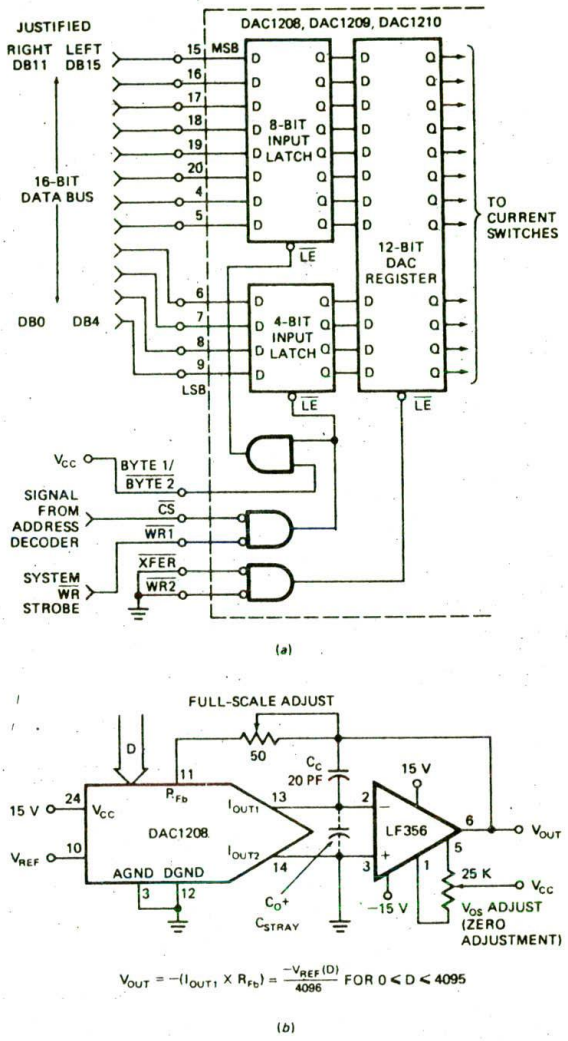


FIGURE 10-17 (*a*) National DAC1208 12-bit D/A input block diagram showing internal latches. (*b*) Analog circuit connections.

an SDK-86 board, you can simply connect the DAC1208 data inputs to the lower 12 data bus lines, connect the $\overline{CS}$ input to an address decoder output, connect the $\overline{WR1}$ input to the system $\overline{WR}$ line, and tie the $\overline{WR2}$ and $\overline{XPER}$ inputs to the ground. The BYTE1/$\overline{BYTE}$ input is tied high. You then write words to the converter just as if it were a 16-bit port. The timing parameters for the DAC1208 are acceptable for an 8086 operating with a clock frequency of 5 MHz or less. For higher 8086 clock frequencies, you would have to add a one-shot or other circuitry that inserts a WAIT state each time you write to the D/A. Here are a few notes about the analog connections for these devices.

These D/A converters require a precision voltage reference. The circuit in Figure 10-17*b* uses a −10.000-V

reference. The D/A converters have a current output, so an op amp is used to convert the D/A output current to a proportional voltage. A FET input amplifier is used here because the input bias current of a bipolar input amp might affect the accuracy of the output. The DAC1208 and DAC1230 have built-in feedback resistors which match the temperature characteristics of the internal current-divider resistors, so all you have to add externally is a 50-Ω resistor for "tweaking" purposes. With a −10.000-V reference as shown, the output voltage will be equal to (the digital input word/4096) × (+10.000 V). Note that the D/A has both a digital ground and an analog ground. To avoid getting digital noise in the analog portions of the circuit, these two should be connected together only at the power supply.

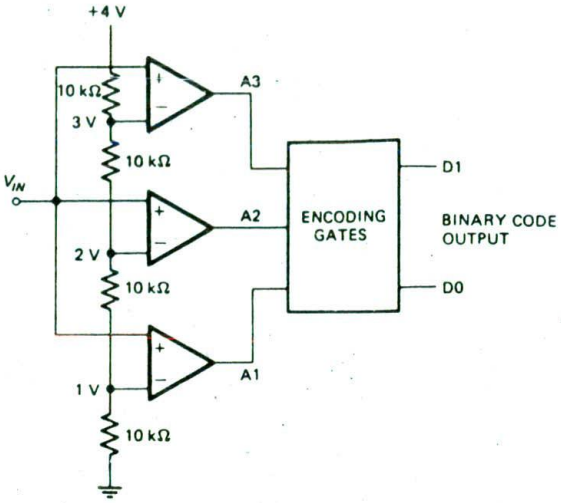


FIGURE 10-18 Parallel comparator A/D converter.

A/D CONVERTER SPECIFICATIONS, TYPES, AND INTERFACING

A/D Specifications

The function of an A/D converter is to produce a digital word which represents the magnitude of some analog voltage or current. The specifications for an A/D converter are very similar to those for a D/A converter. The resolution of an A/D converter refers to the number of bits in the output binary word. An 8-bit converter, for example, has a resolution of 1 part in 256. Accuracy and linearity specifications have the same meanings for an A/D converter as they do for a D/A converter. Another important specification for an A/D converter is its *conversion time*. This is simply the time it takes the converter to produce a valid output binary code for an applied input voltage. When we refer to a converter as *high-speed*, we mean that it has a short conversion time. There are many different ways to do an A/D conversion, but we have space here to review only three commonly used methods, which represent a wide variety of conversion times.

A/D Converter Types

PARALLEL COMPARATOR A/D CONVERTER

Figure 10-18 shows a circuit for a 2-bit A/D converter using *parallel comparators*. A voltage divider sets reference voltages on the inverting inputs of each of the comparators. The voltage at the top of the divider chain represents the full-scale value for the converter. The voltage to be converted is applied to the noninverting inputs of all the comparators in parallel. If the input voltage on a comparator is greater than the reference voltage on the inverting input, the output of the comparator will go high. The outputs of the comparators then give us a digital representation of the voltage level of the input signal. With an input voltage of 2.6 V, for example, the outputs of comparators A1 and A2 will be high.

The major advantage of a parallel, or *flash*, A/D converter is its speed of conversion, which is simply the propagation delay time of the comparators. The output code from the comparators is not a standard binary code, but it can be converted to any desired code with some simple logic. The major disadvantage of a flash A/D is the number of comparators needed to produce a result with a reasonable amount of resolution. The 2-bit converter in Figure 10-18 requires three comparators. To produce a converter with N bits of resolution, you need $(2^N - 1)$ comparators. For an 8-bit conversion, then, you need 255 comparators, and for a 10-bit flash converter, you need 1023 comparators. Single-package flash converters are available from TRW for applications in which the high speed is required, but they are relatively expensive. Flash converters which can do an 8-bit conversion in under 10 ns are currently available.

DUAL-SLOPE A/D CONVERTERS

Figure 10-19*a* shows a functional block diagram of a *dual-slope* A/D converter. This type of converter is often used as the heart of a digital voltmeter because it can give a large number of bits of resolution at a low cost. Here's how the converter in Figure 10-19 works.

To start, the control circuitry resets all the counters to zero and connects the input of the integrator to the input voltage to be converted. If you assume the input voltage is positive, then this will cause the output of the integrator to ramp negative, as shown in Figure 10-19*b*. As soon as the output of the integrator goes a few microvolts below ground, the comparator output will go high. The comparator output being high enables the AND gate and lets the 1-MHz clock into the counter chain. After some fixed number of counts, typically 1000, the control circuitry switches the input of the integrator to a negative reference voltage and resets all the counters to zero. With a negative input voltage, the integrator output will ramp positive, as shown in the right-hand side of Figure 10-19*b*. When the integrator output crosses 0 V, the comparator output will drop low and shut off the clock signal to the counters. The number of counts required for the integrator output to

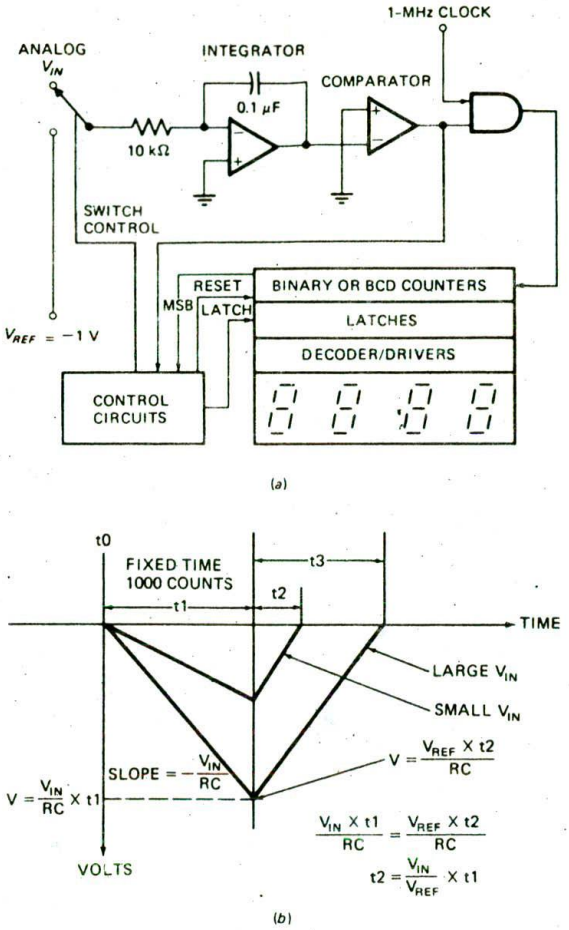


FIGURE 10-19 Dual-slope A/D converter. (*a*) Circuit. (*b*) Integrator output waveform.

get back to zero is directly proportional to the input voltage. For the circuit shown in Figure 10-19*a*, an input signal of +2 V, for example, produces a count of 2000. Because the resistor and the capacitor on the integrator are used for both the input voltage integrate and the reference integrate, small variations in their value with temperature do not have any effect on the accuracy of the conversion.

Complete slope-type A/D converters are readily available in single IC packages. One example is the Intersil ICL7136, which contains all the circuitry for a 3½-digit A/D converter and all the interface circuitry needed to drive a 3½-digit LCD. Another example is the Intersil ICL7135, which contains all the circuitry for a 4½-digit A/D converter and has a multiplexed BCD output. Note that, because of the usual use of this type of converter, we often express its resolution in terms of a number of digits. The full-scale reading for a 3½-digit converter is 1999, so the resolution corresponds to about 1 part in 2000. A two-chip set, the Intersil ICL8068 and ICL7104-16, contains all the circuitry for a slope-type 16-bit binary output A/D converter.

The main disadvantage of slope-type converters is their slow speed. A 4½-digit unit may take 300 ms to do a conversion.

SUCCESSIVE-APPROXIMATION A/D CONVERTERS

Figure 10-20, p. 306, shows a circuit for an 8-bit *successive-approximation* converter which uses readily available parts. The heart of this converter is a successive-approximation register (SAR) such as the MC14549, which functions as follows.

On the first clock pulse at the start of a conversion cycle, the SAR outputs a high on its most-significant bit to the MC1408 D/A converter. The D/A converter and the amplifier convert this to a voltage and apply it to one input of a comparator. If this voltage is higher than the input voltage on the other input of the comparator, the comparator output will go low and tell the SAR to turn off that bit because it is too large. If the voltage from the D/A converter is less than the input voltage, then the 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 Figure 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 Figure 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 V instead of 0 V to +10 V so that sine waves and other ac signals can be input directly to the converter to be digitized.

The National ADC1280 is a single-chip 12-bit successive-approximation converter which does a conversion in about 22 μs. Datel and Analog Devices have several 12-bit converters with conversion times of about 1 μs.

Several commonly available successive-approximation A/D converters have analog multiplexers on their inputs. The National ADC0816, for example, has a 16-input multiplexer in front of the A/D converter. This allows the one converter to digitize any one of 16 input signals. The input channel to be digitized is determined by a 4-bit address applied to the address inputs of the device. An A/D converter with a multiplexer on its inputs is often called a *data acquisition system*, or DAS. Later in this chapter we show an application of a DAS in a factory control system.

Before we go on to discuss A/D interfacing, we need to make a few comments about common A/D output codes.

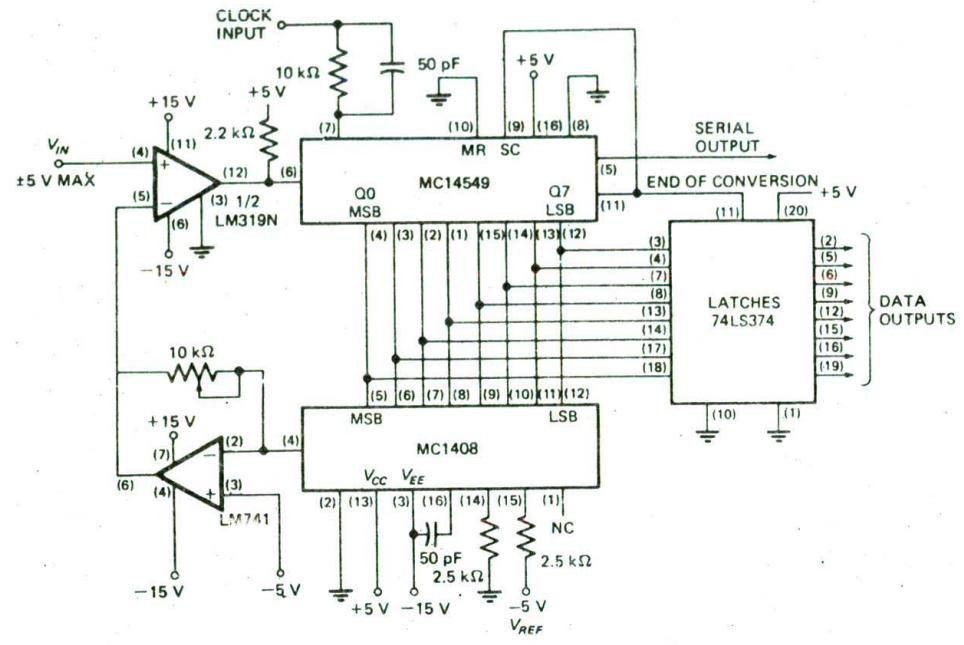


FIGURE 10-20 Successive-approximation A/D converter circuit.

A/D OUTPUT CODES

For convenience in different applications, A/D converters are available with several different, somewhat confusing, output codes. The best way to make sense out of these different codes is to see them all together with representative values, as shown in Figure 10-21. The values shown here are for an 8-bit converter, but you can extend them to any number of bits.

For an A/D converter with only a positive input range (*unipolar*), a straight binary code or inverted binary code is usually used. If the output of an A/D converter is going to drive a display, then it is convenient to have the output coded in BCD. For applications where the input range of the converter has both a negative and a positive range (*bipolar*), we usually use offset-binary coding. As you can see in Figure 10-21, the values of 00000000 to 11111111 are simply shifted downward so that 00000000 represents the most negative input value and 10000000 represents an input value of zero. This coding scheme has the advantage that the 2's complement representation can be produced by simply inverting the most significant bit. Some bipolar converters output the digital value directly in 2's complement form.

UNIPOLAR BINARY CODES

VALUE	10 VOLTS FULL SCALE	BINARY (BIN)	COMPLEMENTARY BINARY (CB)	INVERTED BINARY (IB)	INVERTED COMPLEMENTARY BINARY (ICB)
+FS −1 LSB	9.9609	1111 1111	0000 0000		
+½ FS	5.0000	1000 0000	0111 1111		
+½FS −1 LSB	4.9609	0111 1111	1000 0000		
+1 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
−½ FS + 1 LSB	−4.9609			0111 1111	1000 0000
−½ FS	−5.0000			1000 0000	0111 1111
− FS + 1 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 LSB	9.9	1001 1001	0110 0110		
+½ FS	5.0	0101 0000	1010 1111		
+1 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
−½ FS	−5.0			0101 0000	1010 1111
−FS +1 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 LSB	4.9609	1111 1111	0000 0000	0111 1111
+1 LSB	0.0391	1000 0001	0111 1110	0000 0001
ZERO	0.0000	1000 0000	0111 1111	0000 0000
−1 LSB	−0.0391	0111 1111	1000 0000	1111 1111
−FS +1 LSB	−4.9609	0000 0001	1111 1110	1000 0001
−FS	−5.0000	0000 0000	1111 1111	1000 0000

FIGURE 10-21 Common A/D output codes.

Interfacing Different Types of A/D Converters to Microcomputers

INTERFACING TO PARALLEL-COMPARATOR A/D CONVERTERS

In any application where a parallel comparator converter is used, the converter is most likely going to be producing digital output values much faster than a microcomputer

could possibly read them in. Therefore, separate circuitry is used to bypass the microprocessor and load a set of samples from the converter directly into a series of memory locations. The microprocessor can later perform the desired operation on the samples. Bypassing the microprocessor in this way is called *direct memory access*, or DMA. The basic principle of DMA is that an external controller IC tells the microprocessor to float its buses. When the microprocessor does this, the DMA controller takes control of the buses and allows data to be transferred directly from the A/D converter to successive memory locations. We discuss DMA in detail in the next chapter.

INTERFACING TO SLOPE-TYPE A/D CONVERTERS

Most of the commonly available slope-type converters were designed to drive 7-segment displays in, for example, a digital voltmeter. Therefore, they usually output data in a multiplexed BCD or 7-segment form. Figure 10-23 shows how you can connect the multiplexed BCD outputs of an inexpensive 3½-digit slope converter, the MC14433, to a microprocessor port. In the section of the chapter where Figure 10-23 is located, we use this converter as part of a microcomputer-based scale. The BCD data is output from the converter on lines Q0 through Q3. A logic high is output on one of the digit strobe lines, DS1 through DS4, to indicate when the BCD code for the corresponding digit is on the Q outputs. The MC14433 converter shown in Figure 10-23 outputs the BCD code for the most significant digit and then outputs a high on the DS1 pin. After a period of time, it outputs the BCD code for the next most significant digit and outputs a high on the DS2 pin. After all 4 digits have been put out, the cycle repeats.

To read in the data from this converter, the principle is simply to poll the bit corresponding to a strobe line until you find it high, read in the data for that digit, and put the data in a reserved memory location for future reference. After you have read the BCD code for one digit, you poll the bit which corresponds to the strobe line for the next digit until you find it high, read the code for that digit, and put it in memory. Repeat the process until you have the data for all the digits. The A/D converter in Figure 10-23 is connected to do continuous conversions, so you can call the procedure to read in the value from the A/D converter at any time.

Frequency counters, digital voltmeters, and other test instruments often have multiplexed BCD outputs available on their back panel. With the connections and procedure we have just described, you can use these instruments to input data to your microcomputer.

INTERFACING A SUCCESSIVE-APPROXIMATION A/D CONVERTER

Successive-approximation A/D converters usually have outputs for each bit. The code output on these lines is usually straight binary or offset binary. You can simply connect the parallel outputs of the converter to the required number of input port pins and read in the converter output under program control. In addition to the data lines, there are two other successive-approximation A/D converter signal lines you need to interface to the microcomputer for the data transfer. The first of these is a START CONVERT signal which you output from the microcomputer to the A/D to tell it to do a conversion for you. The second signal is an EOC signal which the A/D converter outputs to indicate that the conversion is complete and that the word on the outputs is valid. Here are the program steps you use to get a data sample from this type of converter.

First, you pulse the START CONVERT input for a time required by the particular converter. Then you detect the EOC signal going low on a polled or interrupt basis. You then read in the digitized value from the parallel outputs of the converter. In a later section of this chapter we show a detailed example of this for the National ADC0808 converter.

A MICROCOMPUTER-BASED SCALE

So far in this book we have shown you how a basic microcomputer functions and how to interface a wide variety of devices to the basic microcomputer. Now it's time to show you how some of these pieces are put together to make a microcomputer-based instrument. The first instrument we have chosen is a "smart" scale such as you might see at the checkout stand in your local grocery store.

Overview of Smart-Scale Operation

Figure 10-22, p. 308, shows a block diagram of our smart scale. A load cell converts the applied weight of, for example, a bunch of carrots to a proportional electrical signal. This small signal is amplified and converted to a digital value which can be read in by the microprocessor and sent to the attached display. The user then enters the price per pound with the keyboard, and this price per pound is shown on the display. When the user presses the compute key on the keyboard, the microprocessor multiplies the weight times the price per pound and displays the computed price. After holding the price display long enough for the user to read it, the scale goes back to reading in the weight and displaying it. To save the user from having to type the computed price into the cash register, an output from the scale could be connected directly into the cash register circuitry. Also, a speech synthesizer could be added to verbally tell the customer the weight, price per pound, and total price.

Smart scales such as this have many applications other than weighing carrots. A modified version of this scale is used in company mail rooms to weigh packages and calculate the postage required to send them to different postal zones. The output of the scale is usually connected to a postage meter, which then automatically prints out the required postage sticker. Another application of smart scales is to count coins in a bank or gambling casino. For this application the user simply enters the type of coin being weighed. A conversion factor in the program computes the total number of coins and the total dollar amount. Still another application of a scale such as this is in packaging items for sale. Suppose, for example, that we are manufacturing wood-

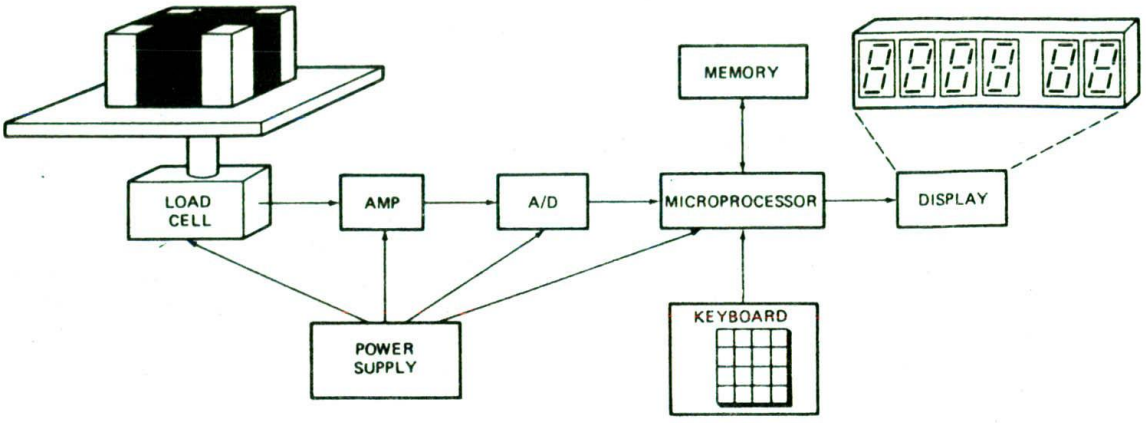


FIGURE 10-22 Block diagram of microcomputer-based smart scale.

arrows and that we want to package 100 of them per box. We can pass the boxes over the load cell on a conveyor belt and fill them from a chute until the weight, and therefore the count, reaches some entered value. The point here is that the combination of intelligence and some simple interface circuitry gives you an instrument with as many uses as your imagination can come up with.

Smart-Scale Input Circuitry

Figure 10-10 shows a picture of the Transducers, Inc. Model C462–10#-10P1 strain gage load cell we used when we built this scale. We added a piece of plywood to the top of the load cell to keep the carrots from falling off. This load cell has an accuracy of about 1 part in 2000, or 0.01 lb over the 0- to 20-lb range for which it was designed.

As shown in Figure 10-23, the load cell consists of four 350-Ω resistors connected in a bridge configuration. A stable 10.00-V excitation voltage is applied to the top of the bridge. With no load on the cell, the outputs from the bridge are at about the same voltage, 5 V. When a load is applied to the bridge, the resistance of one of the lower resistors will be changed. This produces a small differential output voltage from the bridge. The maximum differential output voltage for this 10-lb load cell is 2 mV per volt of excitation, so with 10.00 V excitation as shown, the maximum differential-output voltage is 20 mV.

To amplify this small differential signal, we use a National LM363 instrumentation amplifier. This device contains all the circuitry shown for the instrumentation amplifier in Figure 10-1*h*. The closed-loop gain of the amplifier is programmable with jumpers on pins 2, 3, and 4 for fixed values of 5, 100, and 500. We have jumpered it for a gain of 100 so that the 20-mV maximum signal from the load cell will give a maximum voltage of 2.00 V to the A/D converter input. A precision voltage divider on the output of the amplifier divides this signal in half so that a weight of 10.00 lb produces an output voltage of 1.000 V. This scaling simplifies the display of the weight after it is read into the microprocessor. The 0.1-μF capacitor between pins 15 and 16 of the amplifier reduces the bandwidth of the amplifier to about 7.5 Hz. This removes 60 Hz and any high-frequency noise that might have been induced in the signal lines.

The MC14433 A/D converter used here is an inexpensive dual-slope device intended for use in 3½-digit digital voltmeters, etc. Because the load cell output changes slowly, a fast converter isn't needed here. The voltage across an LM329 6.9-V precision reference diode is amplified by IC4 to produce the 10.00-V excitation voltage for the load cell and a 2.000-V reference for the A/D. With a 2.000-V reference voltage, the full-scale input voltage for the A/D is 2.000 V. Conversion rate and multiplexing frequency for the converter are determined by an internal oscillator and R11. An R11 of 300 kΩ gives a clock frequency of 66 kHz, a multiplex frequency of 0.8 kHz, and about four conversions per second. Accuracy of the converter is ±0.05 percent and ± 1 count, which is comparable to the accuracy of the load cell. In other words, the last digit of the displayed weight may be off by 1 or 2 counts. As we described in a previous section, the output from this converter is in multiplexed BCD form.

An Algorithm for the Smart Scale

Figure 10-24 shows the flowchart for our smart scale. Note that, as indicated by the double-ended boxes in the flowchart, most major parts of the program are written as procedures. This is an example of the structured, modular programming approach we have stressed throughout the book. Here's how it all works.

The output of the A/D is in multiplexed BCD form. The converter outputs the BCD code for a digit on its Q0–Q3 lines and outputs the strobe for that digit on the corresponding digit strobe line, DS1–DS4. To read the data for a digit, that digit strobe is polled until it goes high; then the BCD code for that digit is read in. After the four BCD values are read in from the converter, a display procedure is called to display these values on the address field displays of the SDK-86. The letters "Lb" are displayed in the data field displays.

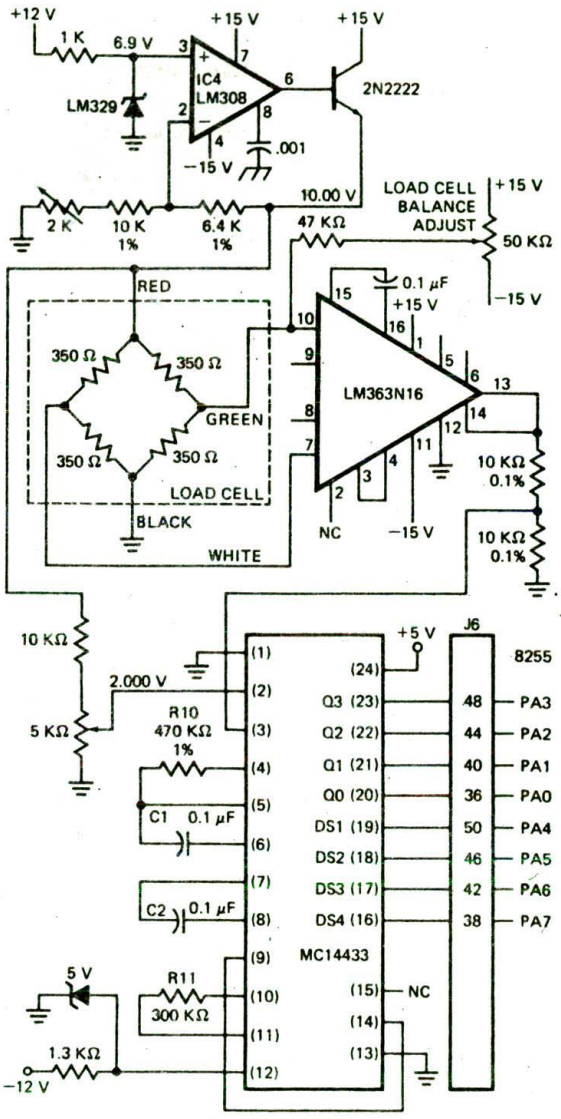


FIGURE 10-23 Circuit diagram for load-cell interface circuitry and A/D converter for smart scale.

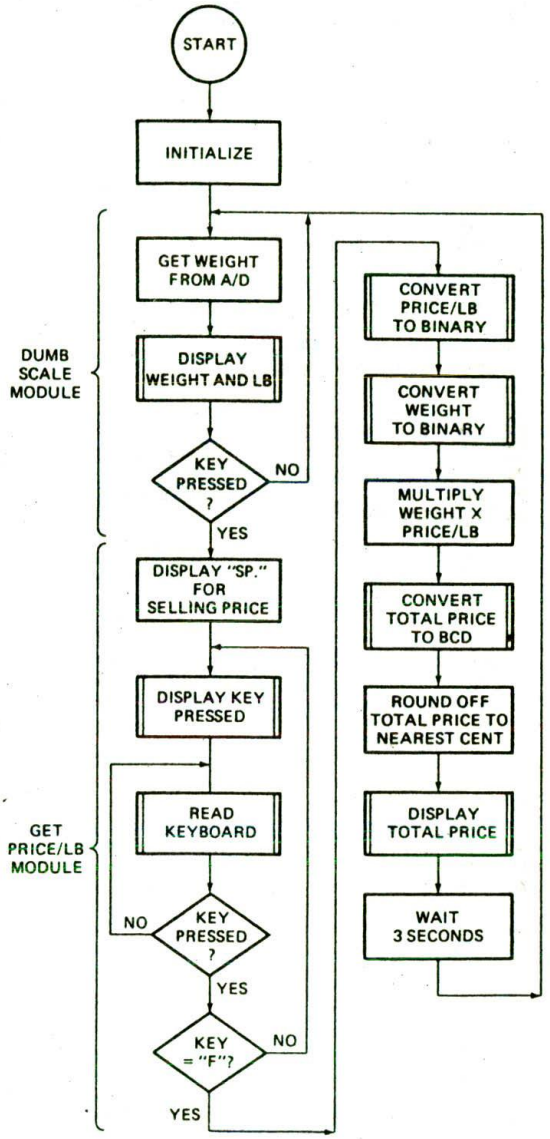


FIGURE 10-24 Flowchart for smart-scale program.

Next, a check is made to see if any keys have been pressed by the user. If a key has been pressed, the letters "SP," which represent selling price, are displayed in the address field. Keycodes are read from the 8279 as entered and displayed on the data field display. Keys can be pressed until the desired price per pound shows on the display. When a nonnumeric key is pressed, it is assumed that the entered price per pound is correct, and the program goes on to compute the total price.

Computing the price involves multiplying the weight in BCD form times the price per pound in BCD form. It is not easy to do a BCD × BCD multiply directly, so we took an alternate route to get there. We converted both the weight and the price per pound to their binary equivalents and then multiplied the binary numbers. Another procedure converts the binary result of the multiplication to BCD. The BCD result is rounded to the nearest cent and displayed in the data field. The letters "Pr" are displayed in the address field to indicate that this is the total price. After a few seconds the program goes back to reading and displaying weight over and over, until a key is pressed.

The Microprocessor-Based Scale Program

Figure 10-25, p. 310–15, shows the complete program for our microprocessor-based scale. It is important for you not to be overwhelmed by a multipage program such

```
 1                                      ;8086 PROGRAM F10-25.ASM
 2                                      ;ABSTRACT  : Program for smart scale
 3                                      ;PORTS     : Uses SDK-86 port P1A (FFF9H) for input
 4                                      ;PROCEDURES: READ_KEY, DISPLAY_IT, PACK, EXPAND, CONVERT2BIN, BINCVT
 5
 6 0000                                 DATA SEGMENT  WORD  PUBLIC
 7 0000  04*(00)                        WEIGHT_BUFFER  DB  4 DUP(0)            ; Space for unpacked BCD weight
 8 0004  04*(00)                        SELL_PRICE     DB  4 DUP(0)            ; Space for unpacked price/pound
 9 0008  04*(00)                        PRICE_TOTAL    DB  4 DUP(0)            ; Space for total price to display
10 000C  0000                           BINARY_WEIGHT  DW  0                   ; Space for converted weight
11 000E  0B 10 14 14                    LB             DB  0BH, 10H, 14H, 14H  ; b, L, blank, blank
12 0012  12 11 14 14                    S_P            DB  12H, 11H, 14H, 14H  ; P, S, blank, blank
13 0016  13 12 14 14                    PR             DB  13H, 12H, 14H, 14H  ; r, P, blank, blank
14
15                                      ;                0    1    2    3    4    5    6    7
16 001A  3F 06 5B 4F 66 6D 7D +         SEVEN_SEG  DB   3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H
17       07
18                                      ;                8    9    A    b    C    d    E    F
19 0022  7F 6F 77 7C 39 5E 79 +                    DB   7FH, 6FH, 77H, 7CH, 39H, 5EH, 79H, 71H
20       71
21                                      ;                L    S    P    r  blank  H
22 002A  38 6D 73 50 00 76 *                       DB   38H, 6DH, 73H, 50H, 00H, 76H
23 0030                                 DATA ENDS
24
25 0000                                 STACK_SEG SEGMENT
26 0000  28*(0000)                             DW       40 DUP(0)
27                                      STACK_TOP      LABEL     WORD
28 0050                                 STACK_SEG      ENDS
29
30 0000                                 CODE  SEGMENT WORD  PUBLIC
31                                           ASSUME CS:CODE, DS:DATA, SS:STACK_SEG
32                                      ;Initialize data & stack segment registers
33 0000  B8 0000s                       START:   MOV   AX, DATA
34 0003  8E D8                                   MOV   DS, AX
35 0005  B8 0000s                                MOV   AX, STACK_SEG
36 0008  8E D0                                   MOV   SS, AX
37 000A  BC 0050r                                MOV   SP, OFFSET STACK_TOP
38                                      ;8279 initialized at power-up of SDK-86 for 8 character display,
39                                      ; left entry encoded scan, 2-key lockout.
40 000D  BA FFEA                                 MOV   DX, 0FFEAH              ; Point at 8279 control address
41 0010  B0 00                                   MOV   AL, 00H                 ; Control word for above conditions
42 0012  EE                                      OUT   DX, AL                  ; Send control word
43 0013  B0 38                                   MOV   AL, 00111000B           ; Clock word for divide by 24
44 0015  EE                                      OUT   DX, AL
45 0016  B0 C0                                   MOV   AL, 11000000B           ; Clear display character is all 0's
46 0018  EE                                      OUT   DX, AL
47                                      ;Dumb scale start
48 0019  B9 0004                        RDWT:    MOV   CX, 04H                 ; Zero out weight buffer
49 001C  BB 0000r                                MOV   BX, OFFSET WEIGHT_BUFFER
50 001F  C6 07 00                       NEXT1:   MOV   BYTE PTR[BX], 00H
51 0022  43                                      INC   BX
52 0023  E2 FA                                   LOOP  NEXT1
53 0025  B9 0004                                 MOV   CX, 04H                 ; Zero out price/pound buffer
54 0028  BB 0004r                                MOV   BX, OFFSET SELL_PRICE
55 002B  C6 07 00                       NEXT2:   MOV   BYTE PTR[BX], 00H
56 002E  43                                      INC   BX
57 002F  E2 FA                                   LOOP  NEXT2
58                                      ;Get weight from A/D converter and display.
59 0031  BB 0003r                                MOV   BX,OFFSET WEIGHT_BUFFER+3; MSD Position in weight buffer
60 0034  BA FFF9                                 MOV   DX, 0FFF9H              ; Point at A/D port
61 0037  EC                             DS1:     IN    AL, DX                  ; Read byte from A/D
62 0038  24 10                                   AND   AL, 10H                 ; Check for MSD strobe high
63 003A  74 FB                                   JZ    DS1                     ; Loop till high
64 003C  EC                                      IN    AL, DX                  ; Read MSD data from A/D
65 003D  24 0F                                   AND   AL, 0FH                 ; Mask strobe bits
66 003F  3C 04                                   CMP   AL, 04H                 ; See if MSD in bit 3 is a one
67 0041  74 06                                   JE    LOAD1                   ; Yes, go load 01H in buffer
68 0043  C6 07 14                                MOV   BYTE PTR[BX], 14H       ; No, load code for blank
69 0046  EB 04 90                                JMP   NXTCHR
70 0049  C6 07 01                       LOAD1:   MOV   BYTE PTR[BX], 01H
71 004C  4B                             NXTCHR:  DEC   BX                      ; Point to next buffer location
72 004D  EC                             DS2:     IN    AL, DX                  ; Poll for digit 2 strobe
```

FIGURE 10-25 Assembly language program for smart scale. (*Continued on pages 311–15.*)

```
 73 004E  24 20          AND   AL, 20H
 74 0050  74 FB          JZ    DS2
 75 0052  EC             IN    AL, DX              ; Read digit 2 from A/D
 76 0053  24 0F          AND   AL, 0FH             ; Mask strobe bits
 77 0055  88 07          MOV   [BX], AL            ; Digit 2 BCD to buffer
 78 0057  4B             DEC   BX                  ; Point at next buffer location
 79 0058  EC     DS3:    IN    AL, DX              ; Poll for digit 3 from A/D
 80 0059  24 40          AND   AL, 40H
 81 005B  74 FB          JZ    DS3
 82 005D  EC             IN    AL, DX              ; Read digit 3 from A/D
 83 005E  24 0F          AND   AL, 0FH             ; Mask strobe bits
 84 0060  88 07          MOV   [BX],AL             ; Digit 3 to buffer
 85 0062  4B             DEC   BX                  ; Point to next buffer location
 86 0063  EC     DS4:    IN    AL, DX              ; Poll for digit 4 (LSD)
 87 0064  24 80          AND   AL, 80H
 88 0066  74 FB          JZ    DS4
 89 0068  EC             IN    AL, DX              ; Read digit 4 from A/D
 90 0069  24 0F          AND   AL, 0FH             ; Mask strobe bits
 91 006B  88 07          MOV   [BX], AL            ; Digit 4 BCD to buffer
 92                     ;Display weight on address field of SDK-86
 93 006D  BB 0000r       MOV    BX, OFFSET WEIGHT_BUFFER   ; Point at stored weight
 94 0070  B0 01          MOV    AL, 01H            ; Specifies address field
 95 0072  B4 01          MOV    AH, 01H            ; Specifies decimal point
 96 0074  E8 00CF        CALL   DISPLAY_IT
 97 0077  BB 000Er       MOV    BX, OFFSET LB      ; Point at Lb string
 98 007A  B0 00          MOV    AL, 00             ; Specifies data field
 99 007C  B4 00          MOV    AH, 00             ; Specifies no decimal point
100 007E  E8 00C5        CALL   DISPLAY_IT
101                     ;Check if key has been pressed
102 0081  BA FFEA        MOV    DX, 0FFEAH         ; Point at 8279 status address
103 0084  EC             IN     AL, DX             ; Read 8279 FIFO status
104 0085  24 01          AND    AL, 01H            ; See if FIFO has keycode
105 0087  75 02          JNZ    GETKEY             ; Yes, go read it
106 0089  EB 8E          JMP    RDWT               ; No, go get weight and display
107 008B  B0 40  GETKEY: MOV    AL, 01000000B      ; Control word for read FIFO
108 008D  EE             OUT    DX, AL             ; Send to 8279
109 008E  BA FFE8        MOV    DX, 0FFE8H         ; Point at 8279 data address
110 0091  EC             IN     AL, DX             ; Read code from FIFO
111 0092  3C 09          CMP    AL, 09H            ; Check if legal keycode (number)
112 0094  76 02          JBE    OK                 ; Go on if below or equal 9
113 0096  EB 81          JMP    RDWT               ; Else ignore, read weight again
114                     ;Read in and display price/pound
115 0098  BB 0004r  OK:  MOV    BX, OFFSET SELL_PRICE ; Point at price per pound buffer
116 009B  88 07          MOV    [BX], AL           ; Keycode to buffer
117 009D  B0 00          MOV    AL, 00             ; Specify data field for display
118 009F  B4 01          MOV    AH, 01             ; Specify decimal point
119 00A1  E8 00A2        CALL   DISPLAY_IT
120 00A4  BB 0012r       MOV    BX, OFFSET S_P     ; Point at SP string
121 00A7  B0 01          MOV    AL, 01             ; Specify address field
122 00A9  B4 00          MOV    AH, 00             ; Specify no decimal point
123 00AB  E8 0098        CALL   DISPLAY_IT
124 00AE  E8 0083 NXTKEY: CALL  READ_KEY           ; Wait for next keypress
125 00B1  3C 09          CMP    AL, 09H            ; See if more price or command
126 00B3  77 1F          JA     COMPUTE            ; Go compute total price
127 00B5  BB 0004r       MOV    BX, OFFSET SELL_PRICE ; Point at price per pound buffer
128 00B8  8A 4F 02       MOV    CL, [BX+2]         ; Shift contents of buffer one
129 00BB  88 4F 03       MOV    [BX+3], CL         ; position left and insert new
130 00BE  8A 4F 01       MOV    CL, [BX+1]         ; keycode
131 00C1  88 4F 02       MOV    [BX+2], CL
132 00C4  8A 0F          MOV    CL, [BX]
133 00C6  88 4F 01       MOV    [BX+1], CL
134 00C9  88 07          MOV    [BX], AL
135 00CB  B0 00          MOV    AL, 00             ; Specify data field
136 00CD  B4 01          MOV    AH, 01             ; Specify decimal point
137 00CF  E8 0074        CALL   DISPLAY_IT
138 00D2  EB DA          JMP    NXTKEY             ; Keep reading and shifting keys
139                                                 ; until command key pressed
140                     ;Compute total price
141 00D4  BB 0000r COMPUTE: MOV  BX,OFFSET WEIGHT_BUFFER; Point at weight buffer for pack
142 00D7  80 7F 03 14    CMP    BYTE PTR[BX+3], 14H ; See if MSD of weight = 0
143 00DB  75 04          JNE    NOTZER
144 00DD  C6 47 03 00    MOV    BYTE PTR[BX+3], 00  ; Yes, load 0 in place of blank code
145 00E1  E8 009B NOTZER: CALL  PACK               ; Pack BCD weight into word
```

FIGURE 10-25 *(Continued)*

```
146 00E4  E8 00D8                     CALL    CONVERT2BIN             ; Convert to 16 bit binary in AX
147 00E7  A3 000Cr                    MOV     BINARY_WEIGHT, AX       ;  and save
148 00EA  BB 0004r                    MOV     BX, OFFSET SELL_PRICE   ; Point at price per pound for pack
149 00ED  E8 008F                     CALL    PACK                    ; Pack BCD price into AX for convert
150 00F0  E8 00CC                     CALL    CONVERT2BIN             ; Convert price to 16-bit binary in AX
151 00F3  F7 26 000Cr                 MUL     BINARY_WEIGHT           ; Price per pound in AX x binary weight
152                                                                   ; total price result in DX:AX
153 00F7  8B D8                       MOV     BX, AX                  ; Prepare for convert to BCD
154 00F9  E8 0104                     CALL    BINCVT                  ; Packed BCD price result in DX:BX
155                                 ;Round off price to nearest cent and display
156 00FC  80 FB 49                    CMP     BL, 49H                 ; Carry set if BL >49H
157 00FF  B0 00                       MOV     AL, 00                  ; Clear AL, keep carry
158 0101  12 C7                       ADC     AL, BH                  ; Add any carry to next digit
159 0103  27                          DAA                             ; Keep in BCD format
160 0104  8A D8                       MOV     BL, AL                  ; Save lower two digits of price
161 0106  B0 00                       MOV     AL, 00                  ; Clear AL, save carry
162 0108  12 C2                       ADC     AL, DL                  ; Propagate carry to upper digits
163 010A  27                          DAA                             ; Keep in BCD form
164 010B  8A E0                       MOV     AH, AL                  ; Position upper digits for EXPAND
165 010D  8A C3                       MOV     AL, BL                  ; Position lower digits for EXPAND
166 010F  BB 0008r                    MOV     BX, OFFSET PRICE_TOTAL  ; Point at buffer for expanded BCD
167 0112  E8 0084                     CALL    EXPAND                  ; Unpack BCD for DISPLAY_IT procedure
168 0115  B0 00                       MOV     AL, 00                  ; Display total price on data field
169 0117  B4 01                       MOV     AH, 01                  ;  with decimal point
170 0119  E8 002A                     CALL    DISPLAY_IT
171 011C  BB 0016r                    MOV     BX, OFFSET PR           ; Point at price/lb string
172 011F  B0 01                       MOV     AL, 01                  ; Display in address field
173 0121  B4 00                       MOV     AH, 00                  ; without decimal point
174 0123  E8 0020                     CALL    DISPLAY_IT
175                                 ;Delay a few seconds
176 0126  B9 FFFF                     MOV     CX,0FFFFH               ; Delay a few seconds
177 0129  BB 000A                   CNTDN1: MOV     BX, 000AH
178 012C  4B                        CNTDN2: DEC     BX
179 012D  75 FD                             JNZ     CNTDN2
180 012F  E2 F8                             LOOP    CNTDN1
181                                 ;Go read next weight
182 0131  E9 FEE5                           JMP     RDWT                    ; Jump back to dumb scale
183
184                                 ;============= PROCEDURES USED IN SMART SCALE PROGRAM ====================
185
186                                 ;====================== PROCEDURE READ_KEY ==============================
187                                 ;ABSTRACT  :Reads the SDK-86 keyboard - polls the status register of the
188                                 ;           8279 on the SDK-86 board until it finds a key pressed. It then
189                                 ;           reads the keypressed code from the FIFO RAM to AL and exits
190                                 ;REGISTERS: Destroys AL - returns with character read in AL
191
192 0134                           READ_KEY PROC  NEAR
193 0134  52                               PUSH  DX
194 0135  BA FFEA                           MOV   DX, 0FFEAH              ; Point at 8279 control address
195 0138  EC                         NO_KEY: IN    AL, DX                ; Get FIFO status
196 0139  24 01                           AND   AL, 00000001B           ; Mask all but LSB, high if key in FIFO
197 013B  74 FB                           JZ    NO_KEY                  ; Loop until a key is pressed
198 013D  B0 40                           MOV   AL, 01000000B           ; Control word for read FIFO
199 013F  EE                              OUT   DX, AL                  ; Send control word
200 0140  BA FFE8                         MOV   DX, 0FFE8H              ; Point at 8279 data address
201 0143  EC                              IN    AL, DX                  ; Read character in FIFO ram
202 0144  5A                             POP   DX
203 0145  C3                             RET
204 0146                           READ_KEY ENDP
205
206                                 ;==================== PROCEDURE DISPLAY_IT ============================
207                                 ;ABSTRACT: Displays characters on the SDK-86 display. The data is sent to
208                                 ;INPUT   : AL=0 for data field
209                                 ;          AL=1 for address field
210                                 ;          AH=0 for no decimal point
211                                 ;          AH=1 for decimal point between second & third digit
212                                 ;          BX= offset of buffer containing 7-seg codes of the four
213                                 ;              characters to be displayed
214 0146                           DISPLAY_IT  PROC  NEAR
215 0146  9C                              PUSHF                         ; Save flags and registers
216 0147  50                              PUSH  AX
217 0148  53                              PUSH  BX
218 0149  51                              PUSH  CX
```

FIGURE 10-25 (*Continued*)

```
219 014A  52                          PUSH  DX
220 014B  56                          PUSH  SI
221 014C  BA FFEA                     MOV   DX, 0FFEAH          ; Point at 8279 control address
222 014F  3C 00                       CMP   AL, 00H             ; See if data field required
223 0151  74 05                       JZ    DATFLD              ; Yes, load control word for data field
224 0153  B0 94                       MOV   AL, 94H             ; No, load address-field control word
225 0155  EB 03 90                    JMP   SEND                ; Go send control word
226 0158  B0 90                 DATFLD: MOV   AL, 90H           ; Load control word for data field
227 015A  EE                    SEND:   OUT   DX, AL            ; Send control word to 8279
228 015B  B1 04                       MOV   CL, 04H             ; Counter for number of characters
229 015D  8B F3                       MOV   SI, BX              ; Free BX for use with XLAT
230 015F  BB 001Ar                    MOV   BX, OFFSET SEVEN_SEG ; Pointer to seven-segment codes
231 0162  BA FFE8                     MOV   DX, 0FFE8H          ; Point at 8279 display RAM
232 0165  8A 04                 AGAIN:  MOV   AL, [SI]          ; Get character to be displayed
233 0167  D7                          XLATB                     ; Translate to 7-seg code
234 0168  80 F9 02                    CMP   CL, 02H             ; See if digit that gets decimal point
235 016B  75 07                       JNE   MORE                ; No, go send digit
236 016D  80 FC 01                    CMP   AH, 01H             ; Yes, see if decimal point specified
237 0170  75 02                       JNE   MORE                ; No, go send character
238 0172  0C 80                       OR    AL, 80H             ; Yes, OR in decimal point
239 0174  EE                    MORE:   OUT   DX, AL            ; Send 7-seg code to 8279 display RAM
240 0175  46                          INC   SI                  ; Point to next character
241 0176  E2 ED                       LOOP  AGAIN               ; until all four characters sent
242 0178  5E                          POP   SI
243 0179  5A                          POP   DX                  ; Restore all registers and flags
244 017A  59                          POP   CX
245 017B  5B                          POP   BX
246 017C  58                          POP   AX
247 017D  9D                          POPF
248 017E  C3                          RET
249 017F                        DISPLAY_IT  ENDP
250
251                             ;========================= PROCEDURE PACK ============================
252                             ;ABSTRACT: Converts four unpacked BCD digits pointed to by BX to
253                             ;          four packed BCD digits in AX
254                             ;DESTROYS: AX
255
256 017F                        PACK    PROC  NEAR
257 017F  9C                          PUSHF                     ; Save flags and registers
258 0180  53                          PUSH  BX
259 0181  51                          PUSH  CX
260 0182  8A 07                       MOV   AL, [BX]            ; First BCD digit to AL
261 0184  B1 04                       MOV   CL, 04H             ; Counter for rotate
262 0186  D2 47 01                    ROL   BYTE PTR[BX+1], CL  ; Position second BCD digit
263 0189  02 47 01                    ADD   AL, [BX+1]          ; First 2 digits in AL
264 018C  8A 67 02                    MOV   AH, [BX+2]          ; Third digit to AH
265 018F  D2 47 03                    ROL   BYTE PTR[BX+3], CL  ; Position fourth digit
266 0192  02 67 03                    ADD   AH, [BX+3]          ; Second two digits now in AH
267 0195  5B                          POP   BX
268 0196  59                          POP   CX
269 0197  9D                          POPF
270 0198  C3                          RET
271 0199                        PACK    ENDP
272
273                             ;========================== PROCEDURE EXPAND ==========================
274                             ;ABSTRACT:  Expands a packed BCD number in AX to 4 unpacked BCD
275                             ;           digits in a buffer pointed to by BX
276
277 0199                        EXPAND  PROC NEAR
278 0199  9C                          PUSHF
279 019A  50                          PUSH  AX
280 019B  53                          PUSH  BX
281 019C  51                          PUSH  CX
282 019D  88 07                       MOV   [BX],AL             ; Move first 2 BCD digits to buffer
283 019F  80 27 0F                    AND   BYTE PTR[BX],0FH    ; Mask off upper digit
284 01A2  B1 04                       MOV   CL, 04H             ; Counter for rotates
285 01A4  D2 C8                       ROR   AL, CL              ; Position digit 2 in low nibble
286 01A6  24 0F                       AND   AL, 0FH             ; Mask upper nibble
287 01A8  88 47 01                    MOV   [BX+1], AL          ; Digit 2 to buffer
288 01AB  88 67 02                    MOV   [BX+2], AH          ; Second 2 BCD digits to buffer
289 01AE  80 67 02 0F                 AND   BYTE PTR[BX+2],0FH  ; Mask off upper digit
290 01B2  D2 CC                       ROR   AH, CL              ; Position digit 4 in low nibble
291 01B4  80 E4 0F                    AND   AH, 0FH             ; Mask upper nibble
```

FIGURE 10-25 (*Continued*)

```
292 01B7  88 67 03                MOV   [BX+3], AH              ; Digit 4 to buffer
293 01BA  59                      POP   CX
294 01BB  5B                      POP   BX
295 01BC  58                      POP   AX
296 01BD  9D                      POPF
297 01BE  C3                      RET
298 01BF                    EXPAND   ENDP
299
300                         ;======================== PROCEDURE CONVERT2BIN ========================
301                         ;ABSTRACT: Converts a 4 digit BCD number in AX register into its binary
302                         ;          (HEX) equivalent. It returns the result in the AX register
303                         ;DESTROYS: AX register
304
305       = 03E8                THOU EQU   3E8H                 ; 1000 = 3E8H
306 01BF                    CONVERT2BIN    PROC NEAR
307 01BF  9C                      PUSHF                         ; Save flags and registers
308 01C0  53                      PUSH  BX
309 01C1  52                      PUSH  DX
310 01C2  51                      PUSH  CX
311 01C3  57                      PUSH  DI
312 01C4  8B D8                   MOV   BX, AX                  ; Copy number into BX
313 01C6  8A C4                   MOV   AL, AH                  ; Place for upper 2 digits
314 01C8  8A FB                   MOV   BH, BL                  ; Place for lower 2 digits
315                         ;Split up numbers so that we have one digit in each register
316 01CA  B1 04                   MOV   CL, 04                  ; Nibble count for rotate
317 01CC  D2 CC                   ROR   AH, CL                  ; Digit 1 in correct place
318 01CE  D2 CF                   ROR   BH, CL                  ; Digit 3 in correct place
319 01D0  25 0F0F                 AND   AX, 0F0FH
320 01D3  81 E3 0F0F              AND   BX, 0F0FH               ; Mask upper nibbles of each digit
321                         ;Copy AX into CX so that can use AX for multiplication
322 01D7  8B C8                   MOV   CX, AX
323 01D9  B8 0000                 MOV   AX, 0000H
324                         ;Now multiply each number by its place value
325 01DC  8A C5                   MOV   AL, CH                  ; Multiply byte in AL * word
326 01DE  BF 03E8                 MOV   DI, THOU                ; No immediate multiplication
327 01E1  F7 E7                   MUL   DI                      ; Digit 1 * 1000
328                         ;Result in DX and AX. Because BCD digit not >9 result in AX only
329                         ;Zero DX and add BL because that digit needs no multiplication for
330                         ;place value. Then add the result in AX for digit 4
331 01E3  BA 0000                 MOV   DX, 0000H
332 01E6  02 D3                   ADD   DL, BL                  ; Add digit 1
333 01E8  03 D0                   ADD   DX, AX                  ; Add digit 4
334                         ;Continue with multiplications
335 01EA  B8 0064                 MOV   AX, 0064H               ; Byte * byte result in AX
336 01ED  F6 E1                   MUL   CL                      ; Digit 2 * 100
337 01EF  03 D0                   ADD   DX, AX                  ; Add digit 3
338 01F1  B8 000A                 MOV   AX, 000AH               ; Byte * byte result in AX
339 01F4  F6 E7                   MUL   BH
340 01F6  03 D0                   ADD   DX, AX                  ; Add digit 2
341 01F8  8B C2                   MOV   AX, DX                  ; Put result in correct place
342 01FA  5F                      POP   DI
343 01FB  59                      POP   CX
344 01FC  5A                      POP   DX                      ; Restore registers
345 01FD  5B                      POP   BX
346 01FE  9D                      POPF
347 01FF  C3                      RET
348 0200                    CONVERT2BIN   ENDP
349
350                         ;======================== PROCEDURE BINCVT ============================
351                         ;ABSTRACT: Converts a 24-bit binary number in DL and BX to a packed
352                         ;          BCD equivalent in DX:BX
353                         ;INPUTS:   DL, BX - 24 BIT BINARY NUMBER
354                         ;OUTPUTS:  DX, BX - PACKED BCD RESULT
355                         ;CALLS:    CNVT1
356                         ;DESTROYS: DX and BX
357
358 0200                    BINCVT    PROC NEAR
359 0200  9C                      PUSHF                         ; Save registers and flags
360 0201  50                      PUSH  AX
361 0202  51                      PUSH  CX
362 0203  B6 19                   MOV   DH, 19H                 ; Bit counter for 24 bits
363 0205  E8 001B                 CALL  CNVT1                   ; Produce 2 LS BCD digits in CH
364 0208  8A CD                   MOV   CL, CH                  ; Save in CL
```

FIGURE 10-25 (*Continued*)

```
365 020A  B6 19                    MOV   DH, 19H              ; Bit counter for 24 BITS
366 020C  E8 0014                  CALL  CNVT1               ; Produce next 2 BCD digits in CH
367 020F  51                       PUSH  CX                  ; Save lower 4 BCD digits on stack
368 0210  B6 19                    MOV   DH, 19H             ; Bit counter for 24 bits
369 0212  E8 000E                  CALL  CNVT1               ; Produce next 2 BCD digits in CH
370 0215  8A CD                    MOV   CL, CH              ; Position in CL
371 0217  B6 19                    MOV   DH,19H              ; Set bit counter for 24 bits
372 0219  E8 0007                  CALL  CNVT1               ; Produce last 2 BCD digits in CH
373 021C  8B D1                    MOV   DX,CX               ; Position 4 MS BCD DIGITS for return
374 021E  5B                       POP   BX                  ; Four LS BCD digits back from stack
375 021F  59                       POP   CX                  ; for return
376 0220  58                       POP   AX
377 0221  9D                       POPF
378 0222  C3                       RET
379 0223                   BINCVT  ENDP
380
381                        ;========================== PROCEDURE CNVT1 ============================
382 0223                   CNVT1   PROC  NEAR
383 0223  32 C0                    XOR   AL, AL              ; Clear AL and carry as workspace
384 0225  8A E8                    MOV   CH, AL              ; Clear CH
385 0227  32 C0            CNVT2:  XOR   AL, AL              ; Clear AL and CARRY
386 0229  FE CE                    DEC   DH                  ; Decrement bit counter
387 022B  75 01                    JNZ   CONTINUE            ; Do all bits
388 022D  C3                       RET                       ; Done if DH down to zero
389 022E  D1 D3            CONTINUE:RCL  BX, 1               ; BX left one bit, MSB to carry
390 0230  D0 D2                    RCL   DL, 1               ; MSB from BX to LSB of DL, MSB of DL to carry
391 0232  8A C5                    MOV   AL, CH              ; Move BCD digit being built to AL
392 0234  12 C0                    ADC   AL, AL              ; Double AL and add carry from DL shift
393 0236  27                       DAA                       ; Keep result in BCD form
394 0237  8A E8                    MOV   CH, AL              ; Put back in CH for next time through
395 0239  73 EC                    JNC   CNVT2               ; No carry from DAA, continue
396 023B  83 D3 00                 ADC   BX, 0000H           ; If carry, propagate to BX and DL
397 023E  80 D2 00                 ADC   DL, 00H             ;  for future terms
398 0241  EB E4                    JMP   CNVT2               ; Continue conversion
399 0243                   CNVT1   ENDP
400 0243                   CODE    ENDS
401                                END
```

FIGURE 10-25 (*Continued*)

as this. If you use the 5-minute rule and work your way through this program one module at a time, you should pick up some more useful programming techniques and procedures you can use in your programs.

Three 4-byte buffers set up at the start of the program are used to store the unpacked BCD values of the weight, the price per pound, and the computed total price. These buffers will be used to pass values to the display procedure. The SEVEN_SEG table in the data segment contains the 7-segment codes for BCD digits, hex digits, and some letters we use to indicate which value is being displayed. In the display procedure you will see how these codes are accessed.

After initializing everything, the program polls the digit strobe for the most significant digit from the A/D converter. Since this A/D converter is a 3½-digit unit, the MSD can be only a 0 or a 1. The value for this digit is sent in the third bit (bit 2) of the 4-bit digit read in. If this bit is a 1, then 01 is loaded into the buffer location. If the bit is a 0, then the value which will access the 7-segment code for a blank (14H) is loaded into the buffer location. Each of the other digit strobes is then polled in turn, and the values for those digits are read in. When all the BCD digits for the weight are in the WEIGHT_BUFFER, the display procedure is called to show the weight on the address field.

To use the display procedure we wrote for this program, you first load a 0 or a 1 into AL to specify data field or address field and a 1 or a 0 in AH to specify a decimal point in the middle of the display or no decimal point. You then load BX with the offset of the memory buffer containing the unpacked codes for the digits to be displayed. A program loop in the display procedure uses the XLAT instruction and the SEVEN_SEG table to convert these codes to the required 7-segment values and send the values to the 8279 display RAM. For displaying the weight, BX is simply loaded with the offset of WEIGHT_BUFFER, AL is loaded with 01 to display the weight in the address field, and AH is loaded with 01 to insert a decimal point at the appropriate place.

To display the letters Lb in the data field, BX is loaded with the offset of the string named LB, and the display procedure is called. Again, the XLAT instruction loop converts the codes from the LB string to the required 7-segment codes and sends them out of the 8279 display RAM. The codes in the string named LB represent the offsets from the start of the SEVEN_SEG table for the desired 7-segment codes. For example, the 7-segment code for a P is at offset 12H in the SEVEN_SEG table. Therefore, if you want to display a P, you put 12H in the appropriate location in the character string in memory. The XLAT instruction will then use the value 12H to access the 7-segment code for P in the SEVEN_SEG table.

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:

$$b7 \times 2^7 + b6 \times 2^6 + b5 \times 2^5 + b4 \times 2^4 + b3 \times 2^3 + b2 \times 2^2 + b1 \times 2^1 + b0$$

This can be shuffled around and expressed as

$$\text{Binary number} = ((((((2b7 + b6)\,2 + b5)\,2 + b4)\,2 + b3)\,2 + b2)\,2 + b1)\,2 + b0$$

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 Figure 10-25 produces two BCD digits of the result at a time by calling the subprocedure CNVT1. Figure 10-26 shows a flowchart for the operation

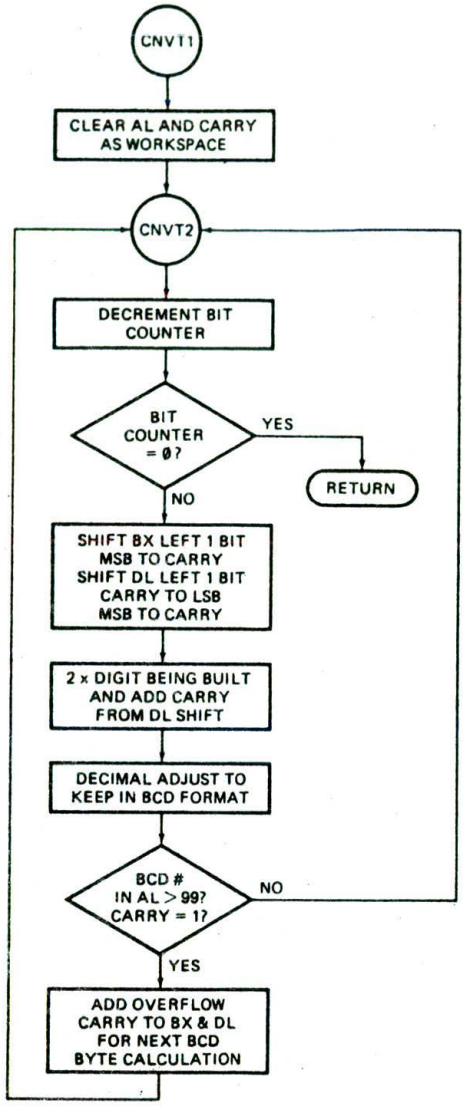


FIGURE 10-26 Flowchart for CNVT1 subprocedure.

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 Figure 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.

Finally, after delaying a few seconds to give the operator time to read the price, execution returns to the "dumb-scale" portion of the program and starts over.

A question that may occur to you when reading a long program such as this is, How do you decide which parts of the program to keep in the mainline and which parts to write as procedures? There is no universal agreement on the answer to this question. The general guidelines we follow are to write a program section as a procedure if it is going to be used more than once in the program, it is reusable (could be used in other programs), it is so lengthy (more than 1 page) that it clutters up the conceptual flow of the main program, or it is an essentially independent section. The disadvantage of using too many procedures is the time and overhead required for each procedure call. As you write more programs, you will arrive at a balance that feels comfortable to you. The following section shows you another long program example which was written in a highly modular manner so that it can easily be expanded. This example should further help you see when and how to use procedures.

A MICROCOMPUTER-BASED INDUSTRIAL PROCESS-CONTROL SYSTEM

Overview of Industrial Process Control

One area in which microprocessors and microcomputers have had a major impact is *industrial process control*. Process control involves first measuring system variables such as motor speed, temperature, the flow of reactants, the level of a liquid in a tank, the thickness of a material, etc. The output of the controller then adjusts the value of each variable until it is equal to a predetermined value called a *set point*. The system controller must maintain each variable as close as possible to its set-point value, and it must compensate as quickly and accurately as possible for any change in the variable caused by, for example, increased load on a motor. A simple example will show the traditional approach to control of a process variable and explain some of the terms used in control systems.

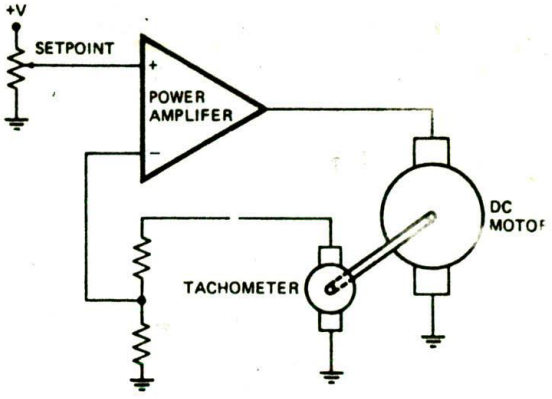


FIGURE 10-27 Circuit for controlling speed of dc motor using feedback from tachometer.

The circuit in Figure 10-27 shows an analog approach to controlling the speed of a dc motor. Attached to the shaft of the motor is a dc generator, or *tachometer*, which puts out a voltage proportional to the speed of the motor. The output voltage is typically a few volts per 1000 rpm. A fraction of the output voltage from the tachometer is fed back to the inverting input of the power amplifier driving the motor. A positive voltage is applied to the noninverting input of the amplifier as a set point. When the power is turned on, the motor accelerates until the voltage fed back from the tachometer to the inverting input of the amplifier is nearly equal to the set-point voltage.

If the load on the motor is increased, the motor will initially slow down, and the voltage output from the tachometer will decrease. This will increase the difference in voltage between the inputs of the amplifier and cause it to drive more current to the motor. The increased current will increase the speed of the motor to nearly the speed it had before the increased load was added. A similar reaction takes place if the load on the motor is decreased.

Using negative feedback to control a system such as this is often called *servo control*. A control loop of this type keeps the motor speed quite constant for applications where the load on the motor does not change much. Some hard-disk drive motors and high-quality phonograph turntables use this method of speed control.

For applications in which the load and/or the set point changes drastically, there are several potential problems. The first of these is overshoot when you change the set point. Figure 10-28*a*, p. 318, shows an example of this. In this case the variable—motor speed, for example—overshoots the new set point and bounces up and down for a while. The time it takes the bouncing to settle within a specified error range or error band is called the

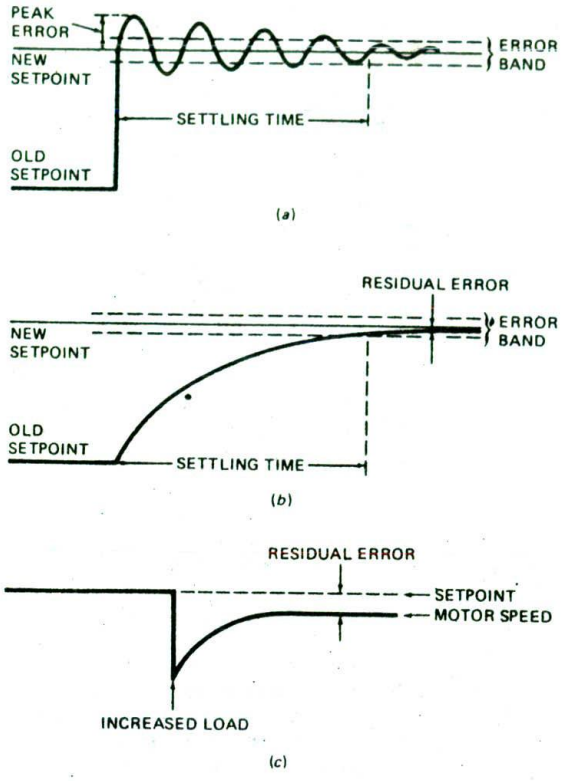


FIGURE 10-28 Overshoot and undershoot of system when set point or load is changed. (a) Overshoot. (b) Undershoot. (c) Load change.

settling time. This type of response is referred to as *underdamped* and is similar to the response that a car with bad shock absorbers will make when it hits a bump. The ringing can be prevented by adding damping to the system. However, if too much damping is added, the response to a change in set point may look like that shown in Figure 10-28*b*. This type response is referred to as an *overdamped response*. The difficulty with this type response is that it takes a long time for the variable to reach the new set point. For best performance, the damping must be custom designed for a particular system.

Another problem in any process control system is *residual error*. Figure 10-28*c* shows the response a control system such as the motor speed controller in Figure 10-27 will have when more load is added on the motor. The motor initially slows down, so the voltage out of the tachometer decreases. As we said before, this increases the voltage difference between the amplifier inputs and causes the amplifier output to increase. Increased amplifier output increases the speed of the motor and thereby the output from the tachometer. When the system reaches equilibrium, however, there is some noticeable difference between the set point and the voltage fed back from the tachometer. It is this difference or residual error which is amplified by the gain of the amplifier to produce the additional drive for the motor. For stability reasons, the gain of many control systems cannot be too high. Therefore, even if you adjust the speed of a motor, for example, to be exactly at a given speed for one load, when you change the load there will always be some residual error between the set point and the actual output.

To help solve these problems, circuits with more complex feedback are used. Figure 10-29 shows a circuit which represents the different types of feedback commonly used. First note in this circuit that the output power amplifier is an adder with four inputs. The current supplied to the summing point of the adder by the set-point input produces the basic output drive current. The other three inputs do not supply any current unless there is a difference between the set point and the feedback voltage from the tachometer. Amplifier 1 is another adder whose function is to compare the set-point voltage with the feedback voltage from the tachometer. Let's assume the two input resistors, R1 and R2, are equal. Since the set-point voltage is negative and the voltage from the tachometer is positive, there will be no net current through the feedback resistor of the amplifier if the two voltages are equal in magnitude. In other words, if the speed of the motor is at its set-point value, the output of amplifier 1 will be zero, and amplifiers 2, 3, and 4 will contribute no current to the summing junction of the power amp.

Now, suppose that you add more load on the motor, slowing it down. The tachometer voltage is no longer equal to the set-point voltage, so amplifier 1 now has some output. This error signal on the output produces three types of feedback to the summing junction of the power amp.

Amplifier 1 produces simple dc feedback proportional to the difference between the set point and the tachometer output. This is exactly the same effect as the voltage divider on the tachometer output in Figure 10-27. *Proportional feedback*, as this is called, will correct for most of the effect of the increased load, but, as we discussed before, there will always be some residual error.

The cure for residual error is to use some *integral feedback*. Amplifier 3 in Figure 10-29 provides this type of feedback. Remember from a previous discussion that this circuit produces a ramp on its output whenever a voltage is applied to its input. For the example here, the integrator will ramp up or ramp down as long as there is any error signal present on its input. By ramping up and down just a tiny bit about the set point, the integrator can eliminate most of the residual error. Too much integral feedback, however, will cause the output to oscillate up and down. Also, feedback only slowly affects the output because the error signal must be present for some time before the integrator has much output.

To improve the response time of the system, amplifier 4 in Figure 10-29 supplies a third type of feedback called *derivative feedback*. Derivative feedback is a signal proportional to the rate of change of the error signal. If the load on the system is suddenly changed, the derivative amplifier circuit will give a quick shot of feedback

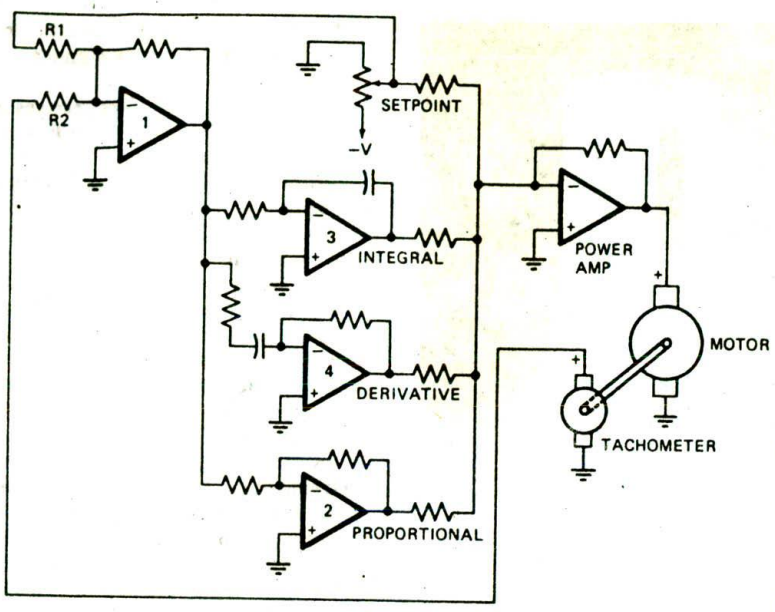


FIGURE 10-29 Circuit showing proportional, integral, and derivative feedback control.

to try to correct the error. When the error signal is first applied to the differentiator circuit, the capacitor in series with the input is not charged, so it acts like a short circuit. This initially lets a large current flow, so the amplifier has a sizable output. As the capacitor charges, the current decreases, so the feedback from the differentiator decreases. The differentiator essentially gives the amplifier a quick pulse of feedback to help correct for the increased load. Too much derivative feedback can cause the system to overshoot and oscillate.

The point here is that by using a combination of some or all of these types of feedback, a given feedback-controlled system can be adjusted for optimum response to changes in load or set point. Process control loops that use all three types of feedback are called *proportional integral derivative* or PID control loops. Because process variables change much more slowly than the microsecond operation of a microcomputer, a microcomputer with some simple input and output circuitry can perform all the functions of the analog circuitry in Figure 10-29 for several PID loops.

Figure 10-30 shows a block diagram of a microcomputer-based process-control system. Data acquisition systems convert the analog signals from various sensors to digital values that can be read in and processed by the microcomputer. A keyboard and display in the

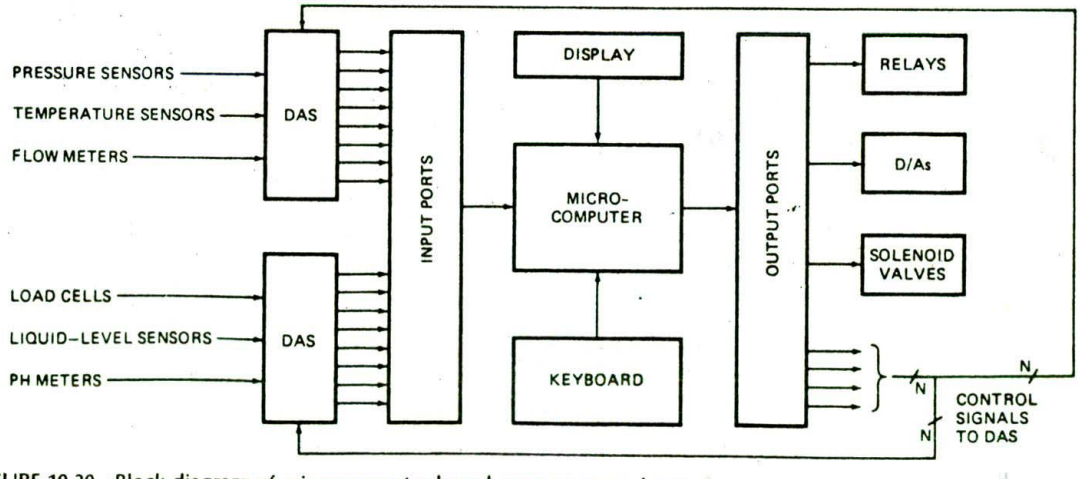


FIGURE 10-30 Block diagram of microcomputer-based process control system.

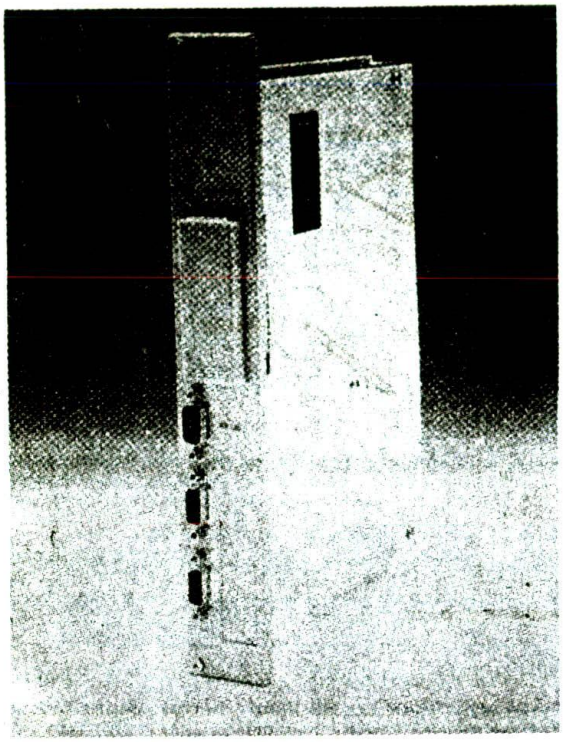


FIGURE 10-31 Photograph of Texas Instruments' programmable controller.

system allow the user to enter set-point values, to read the current values of process variables, and to issue commands. Relays, D/A converters, solenoid valves, and other actuators are used to control process variables under program direction. A programmable timer in the system determines the rate at which control loops are serviced.

Microcomputer-based process-control systems range from a small programmable controller such as the one shown in Figure 10-31, which might be used to control a machine on a factory floor, to a large minicomputer used to control an entire fractionating column in an oil refinery. To show you how these microcomputer-based control systems work, here's an example system you can build and experiment with.

AN 8086-BASED PROCESS-CONTROL SYSTEM

Program Overview

Figure 10-32 shows in flowchart form one way in which the program for a microcomputer-based control system with eight PID loops can be structured. After power is turned on, a mainline or *executive program* initializes ports, initializes the timer, and initializes process variables to some starting values. The executive program then sits in a loop waiting for a user command from the keyboard or a clock "tick" from the timer. The keyboard strobe signal and the clock signal are each connected to interrupt inputs.

When the microcomputer receives an interrupt from the timer, it goes to a procedure which determines whether it is time to service the next control loop. The interrupt procedure does this by counting interrupts in the same way as the real-time clock we described in Chapter 8 does. If you program the timer to produce a pulse every 1 ms, and you want the controller to service another loop every 20 ms, for example, you can simply have the interrupt procedure count 20 interrupts before going on to update the next loop. Once 20 interrupts have been counted down, the program falls into a decision structure which determines which loop is to be updated next. Every 20 ms a new loop is updated, so with eight loops, each loop gets updated every 160 ms. This system is an example of a *time-slice* system, because each loop gets a 20-ms "slice" of time every 160 ms.

An important point here is that the microcomputer services each loop at regular intervals instead of simply updating all eight loops, one loop right after another. This is done so that the timing for each loop is independent of the timing for the other loops. Therefore, a change in the internal timing for one loop will not affect the timing in the other loops.

Each PID loop is controlled by an independent procedure. For our example system here, we have space to show the implementation of only one loop, the control of the temperature of a tank of liquid in, perhaps, our printed-circuit-board-making machine. You could write other similar control-loop procedures to control pH, flow, light exposure timing, motor speed, etc.

Figure 10-32c shows the flowchart for our temperature-controller PID loop. Note that we use lower-level procedures to implement most steps in the basic PID procedure. These low-level operations are written as procedures so that they can be used in other PID loops. Also, using procedures here maintains the top-down program structure we have been trying to get you to use in your programs. Work your way through Figure 10-32 until you clearly see the program levels. After we look at the hardware of the system, we will dig into the details of the actual program.

Hardware for Control Systems and Temperature Controller

To build the hardware for this project, we started by adding an 8254 programmable timer and an 8259A priority interrupt controller to an SDK-86 board, as shown in Figure 8-14. The timer is initialized to produce 1-kHz clock ticks. The 8259A provides interrupt inputs for the clock-tick interrupts and for keyboard interrupts. We built the actual temperature sensing and detecting circuitry on a separate prototyping board and connected it to some ports on the SDK-86 with a ribbon cable. Figure 10-33, p. 322, shows this analog interface circuitry.

The temperature-sensing element in the circuit is an LM35 precision Celsius temperature sensor. The voltage between the output pin and the ground pin of this device will be 0 V at 0° C and will increase by 10 mV for

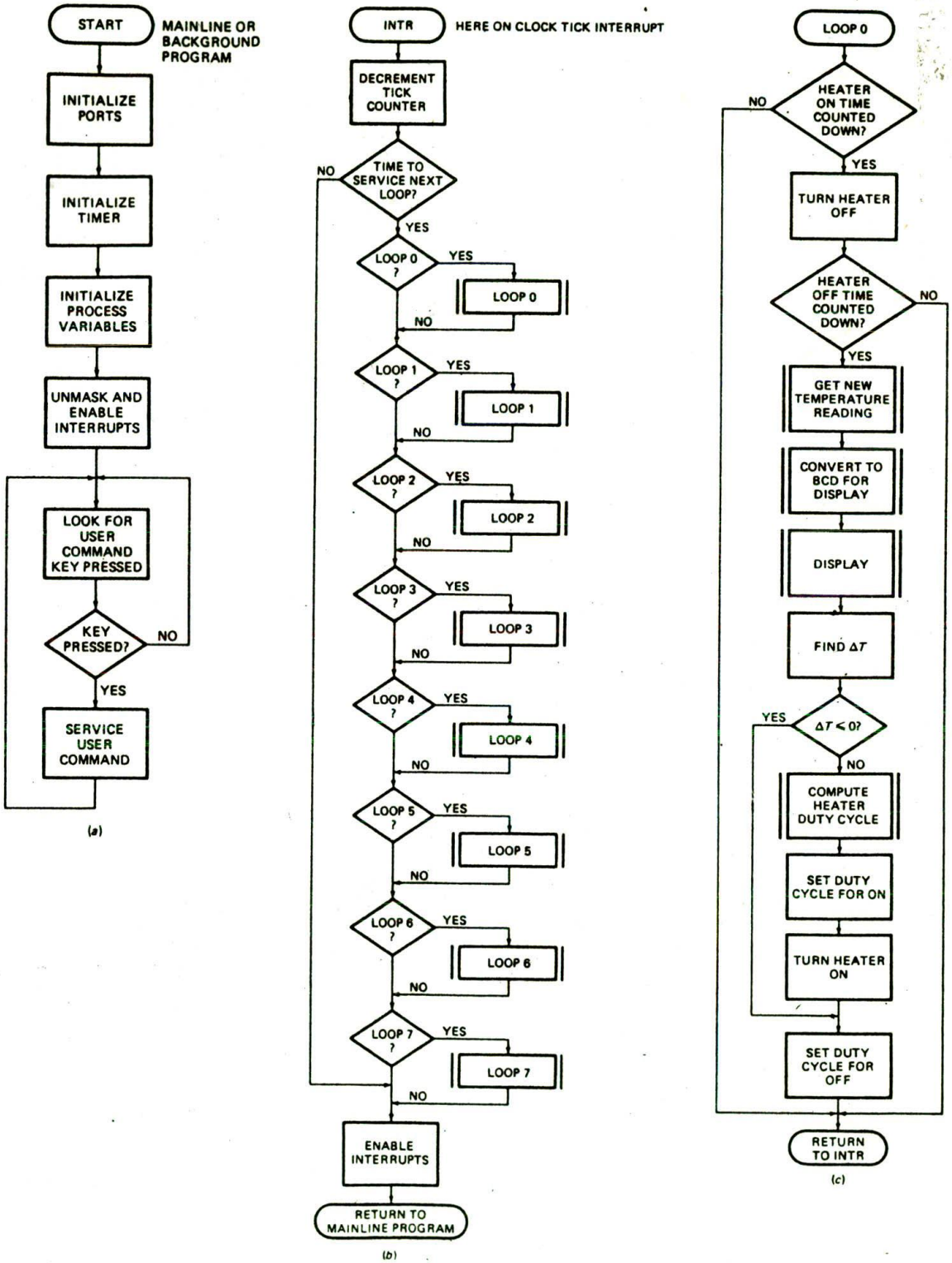


FIGURE 10-32 Flowchart for microcomputer-based process control system. (a) Mainline or executive. (b) Loop selector. (c) Temperature-control loop.

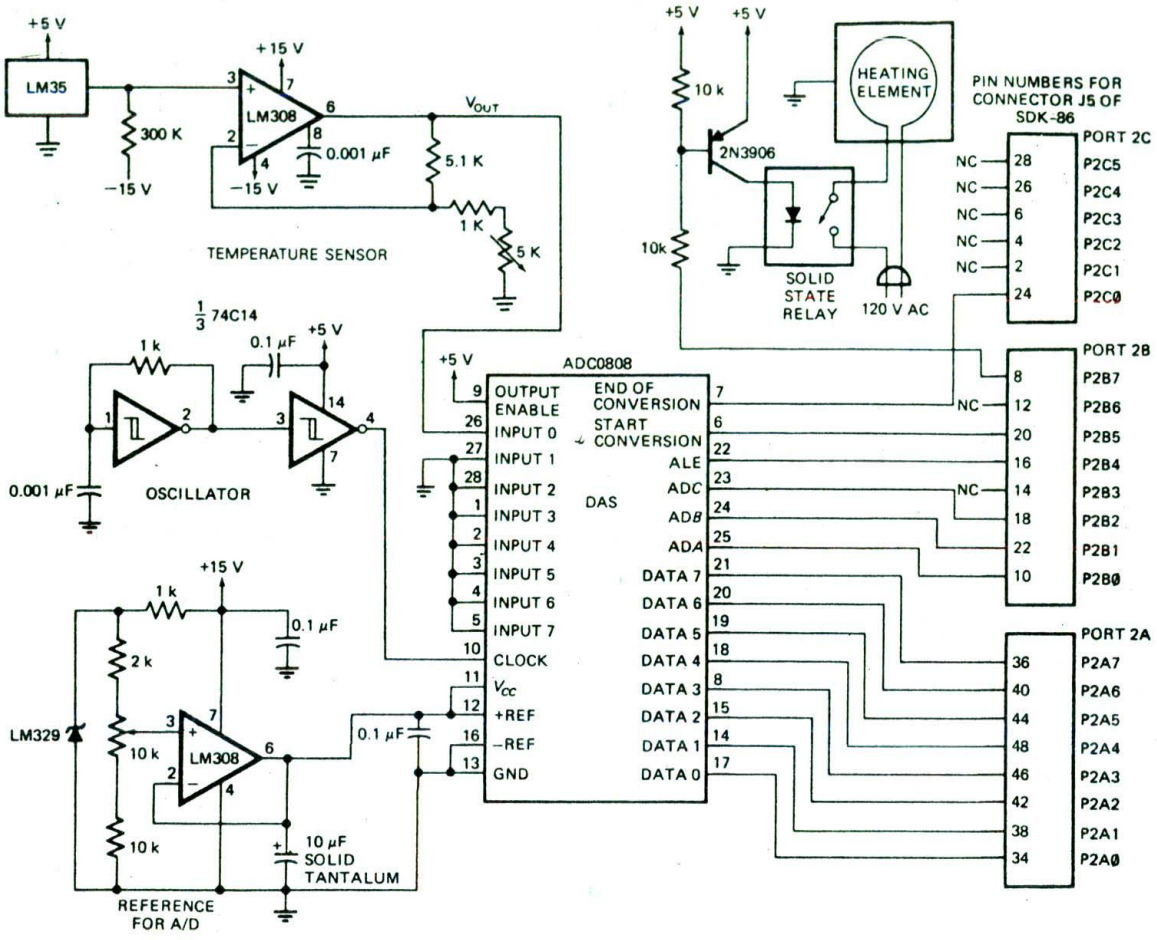


FIGURE 10-33 Temperature-sensing and heater-control circuitry for microcomputer-based controller.

each increase of 1° C above that. The 300-kΩ resistor connecting the output of the LM35 to −15 V allows the output to go negative for temperatures below 0° C. (If you are operating with ±12-V supplies, use a 240-kΩ resistor.) This makes the circuit able to measure temperatures over the range of −55 to +150° C. For our application here, we use only the positive part of the output range, but we thought you might find this circuit useful for some of your other projects. An LM308 buffers and amplifies the signal from the sensor by 2 so that the signal uses a greater part of the input range of the A/D converter. This improves the noise immunity and resolution.

The ADC0808 A/D converter used here is an 8-input *data acquisition system*. You tell the device which input signal you want digitized with a 3-bit address you send to the ADC, ADB, and ADA inputs. This 8-input device was chosen so that other control loops could be added later. Some Schmitt-trigger inverters in a 74C14 are connected as an oscillator to produce a 300-kHz clock for the DAS. The voltage drop across an LM329 low-drift zener is buffered by an LM308 amplifier to produce a V_{CC} and a V_{REF} of 5.12 V for the A/D converter. With this reference voltage, the A/D converter will have 256 steps of 20 mV each. Since the temperature sensor signal is amplified by 2, each degree Celsius of temperature change will produce an output change of 20 mV, or one step on the A/D converter. This gives us a resolution of 1° C, which is about equal to the typical accuracy of the sensor. The advantage of using V_{REF} as the V_{CC} for the device is that this voltage will not have the switching noise that the digital V_{CC} line has. The control inputs and data outputs of the A/D converter are simply connected to SDK-86 ports as shown.

Figure 10-34 shows the timing waveforms and parameters for the ADC0808. Note the sequence in which control signals must be sent to the device. The 3-bit address of the desired input channel is first sent to the multiplexer inputs. After at least 50 ns, the ALE input is sent high. After another 2.5μs, the START CONVERSION input is sent high and then low. Then the ALE input is brought low again. When the END OF CONVER-

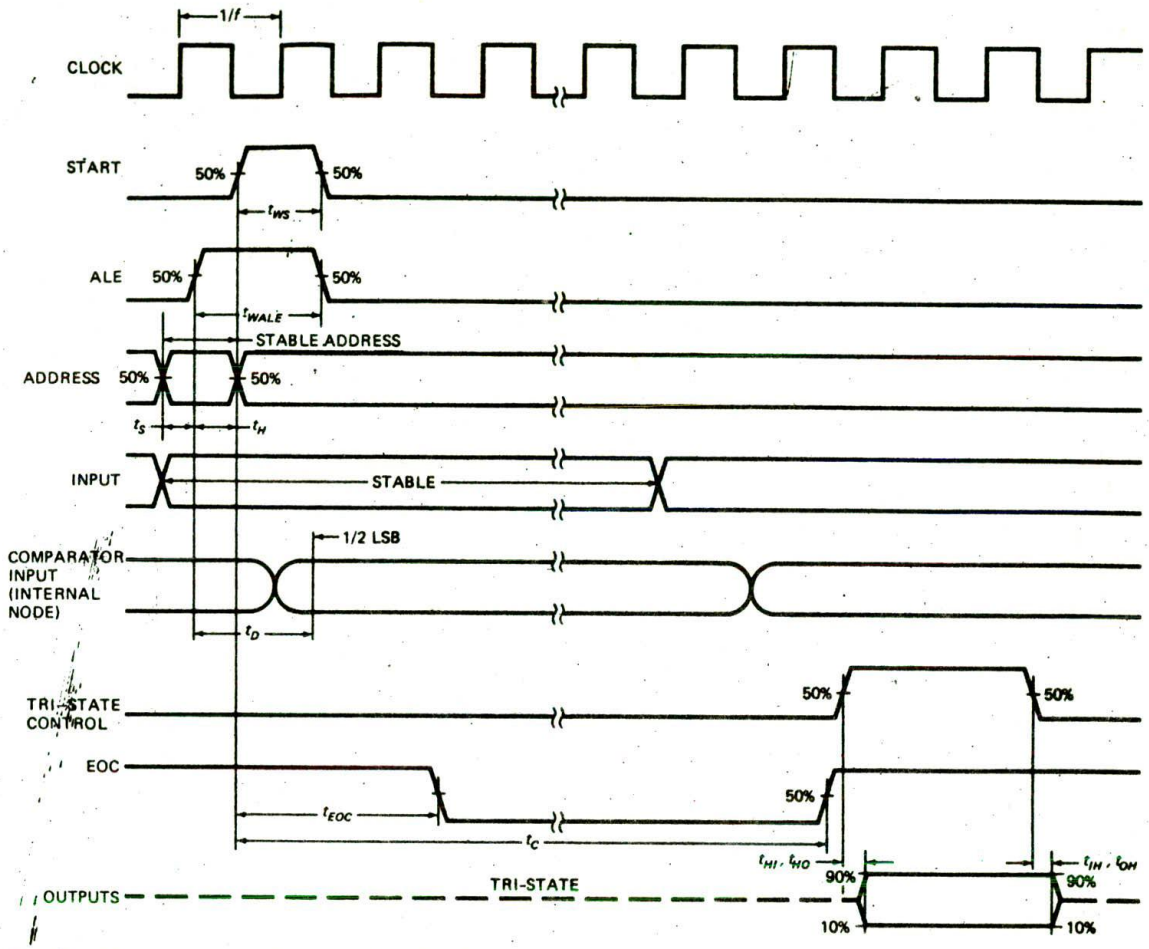


FIGURE 10-34 Timing waveforms for the ADC0808 data acquisition system.

SION signal from the A/D converter is found to be high, the 8-bit data value which represents the temperature can be read in.

To control the power delivered to the heater, we used a 25-A, 0-V turn-on, solid-state relay, such as the Potter Brumfield unit described in Chapter 9. With this relay we can control a 120- or 240-V ac-powered hot plate or immersion heater. To control the amount of heat put out by the heater, we vary the duty cycle of pulses sent to the relay.

For very low power applications, a D/A converter and a power amplifier could be used to drive the heater. However, in high-power applications this is not very practical because the power amplifier may dissipate as much or more power than the load. For example, an amplifier intended to control a 5000-W heater over its full range must be able to dissipate more than 5000 W. The D/A-converter approach has the added disadvantage that it cannot directly use the available ac line voltage.

The driver transistor on the input of the solid-state relay supplies the drive for the relay, isolates the port pin from the relay, and holds the relay in the off position when the power is first turned on. Port pins, remember, are in a floating state after a reset, so some method must be used to hold external circuitry in a known state until the port is initialized and the desired value is output to the port. Now that you know how the hardware is connected, we can explain the operation of the controller program.

The Controller System Program

THE MAINLINE OR EXECUTIVE SECTION

Figure 10-35, pp. 324–29, shows the assembly language program for our controller system. Refer to the flowchart in Figure 10-32 as you work your way through this program. The mainline or executive part of the program starts by initializing port FFFAH for output, the 8259A to receive interrupt inputs from the timer and the keyboard, and the 8254 to produce a 1-kHz square wave on its counter 0 output. In Chapter 8 we described all

```
                                ;8086 MAINLINE PROGRAM F10-35a.ASM       -   MODULE 1
                                ;ABSTRACT: Program for controller system. Services eight process
                                ;          control loops on a rotating basis. Program written to run on an
                                ;          Intel SDK-86 board. Timing for the control loops is generated on
                                ;          an interrupt basis by an on-board 8254 timer. Control-loop 0 in
                                ;          the program controls the temperature of a water bath.
                                ;PORTS:    Uses port P2B (FFFAH) as output
                                ;          bits 7 = heater, bits 6,3 = not connected, bit 5 = start conversion
                                ;          bit 4 = ALE,      bits 2,1,0 = channel address
                                ;          Uses port P2A (FFF8H) as data input
                                ;          Uses port P2C (FFFCH) as end-of-conversion input from A/D
                                ;PROCEDURES: Uses CLOCK_TICK - interrupt service procedure
                                ;                KEYBOARD   - interrupt service procedure (empty)

0000                            INT_PROC    SEGMENT WORD     PUBLIC
                                    EXTRN   CLOCK_TICK:FAR, KEYBOARD:FAR
0000                            INT_PROC    ENDS

                                PUBLIC      COUNTER, TIMEHI, TIMELO, LOOPNUM, CURTEMP, SETPOINT

0000                            AINT_TABLE      SEGMENT  WORD     PUBLIC
0000  02*(0000)                      TYPE_64   DW     2 DUP(0)   ;Reserve space for clock-tick proc addr IR0
0004  02*(0000)                      TYPE_65   DW     2 DUP(0)   ;Not used in this program - IR1
0008  02*(0000)                      TYPE_66   DW     2 DUP(0)   ;Reserve space for keyboard proc addr - IR2
000C                            AINT_TABLE      ENDS

0000                            DATA SEGMENT    WORD  PUBLIC
0000  00                            COUNTER    DB    00          ;Counter for number of interrupts
0001  01                            TIMEHI     DB    01          ;Heater relay - time on
0002  01                            TIMELO     DB    01          ;Heater relay - time off
0003  00                            LOOPNUM    DB    00          ;Temp storage for loop counter
0004  00                            CURTEMP    DB    00          ;Current temperature
0005  3C                            SETPOINT   DB    60          ;Setpoint temperature
0006                            DATA ENDS

0000                            STACK_SEG  SEGMENT                ;No STACK directive because using EXE2BIN
0000  28*(0000)                           DW 40      DUP(0)      ;so can then download code to SDK-86
                                    TOP_STACK LABEL     WORD
0050                            STACK_SEG  ENDS

0000                            CODE SEGMENT WORD  PUBLIC
                                    ASSUME CS:CODE, DS:AINT_TABLE, SS:STACK_SEG
                                ;Initialize stack segment, stack pointer, and data segment registers
0000  B8 0000s                      MOV   AX, STACK_SEG
0003  8E D0                         MOV   SS, AX
0005  BC 0050r                      MOV   SP, OFFSET TOP_STACK
0008  B8 0000s                      MOV   AX, AINT_TABLE
000B  8E D8                         MOV   DS, AX
                                ;Define the addresses for the interrupt service procedures
000D  C7 06 0002r 0000s             MOV   TYPE_64+2, SEG CLOCK_TICK     ;Put in clock-tick proc addr
0013  C7 06 0000r 0000e             MOV   TYPE_64,   OFFSET CLOCK_TICK
0019  C7 06 000Ar 0000s             MOV   TYPE_66+2, SEG KEYBOARD       ;Put in keyboard proc addr
001F  C7 06 0008r 0000e             MOV   TYPE_66,   OFFSET KEYBOARD
                                ;Initialize data segment register
                                    ASSUME DS:DATA
0025  B8 0000s                      MOV   AX, DATA
0028  8E D8                         MOV   DS, AX
                                ;Initialize port P2B (FFFA) as output - mode 0, P2A & P2C as inputs - mode 0
002A  BA FFFE                       MOV   DX, OFFFEH            ;Point DX at port control addr
002D  B0 99                         MOV   AL, 10011001B         ;Mode control word for above conditions
002F  EE                            OUT   DX, AL                ;Send control word
                                ;Initialize 8259A, edge triggered, single, ICW4
0030  B0 13                         MOV   AL, 00010011B
0032  BA FF00                       MOV   DX, 0FF00H            ;Point at 8259A control
0035  EE                            OUT   DX, AL                ;Send ICW1
0036  B0 40                         MOV   AL, 01000000B         ;Type 64 is first 8259A type (IR0)
0038  BA FF02                       MOV   DX, 0FF02H            ;Point at ICW2 address
003B  EE                            OUT   DX, AL                ;and send ICW2
003C  B0 01                         MOV   AL, 00000001B         ;ICW4, 8086 mode
003E  EE                            OUT   DX, AL                ;Send ICW4
003F  B0 FE                         MOV   AL, 11111110B         ;OCW1 to unmask IR0 only leave IR2 masked
0041  EE                            OUT   DX, AL                ;because not used & send OCW1
```

FIGURE 10-35 8086 assembly language program for process control system (*continued on pp. 325–29*). (*a*, pp. 324–5) Module 1—mainline. (*b*, pp. 325–6) Module 2—interrupt-service procedures. (*c*, pp. 326–7) Module 3—loop service procedures. (*d*, pp. 327–29) Module 4—utility procedures.

```
                                ;Initialize 8254 counter 0 for 1-kHz output, LSB then MSB, square wave, BCD
0042  B0 37                         MOV   AL, 00110111B
0044  BA FF07                       MOV   DX, 0FF07H            ;Point 8254 control addr
0047  EE                           OUT   DX, AL                ;Send counter 0 command word
0048  B0 58                        MOV   AL, 58H               ;Load LSB of count
004A  BA FF01                       MOV   DX, 0FF01H            ;Point at counter 0 data addr
004D  EE                           OUT   DX, AL                ;Send LSB of count
004E  B0 24                        MOV   AL, 24H               ;Load MSB of count
0050  EE                           OUT   DX, AL                ;Send MSB of count
                                ;Initialize variables
0051  C6 06 0005r 3C              MOV   SETPOINT, 3CH         ;Initialize final temp at 60°
0056  C6 06 0000r 14              MOV   COUNTER,  14H         ;Intialize time counter
005B  C6 06 0003r 00              MOV   LOOPNUM,  00H         ;Start at first loop
0060  C6 06 0001r 01              MOV   TIMEHI,   01H
0065  C6 06 0002r 01              MOV   TIMELO,   01H
006A  C6 06 0004r 00              MOV   CURTEMP,  00H
                                ;Enable interrupt input of 8086
006F  FB                           STI
0070  EB FE                     HERE:JMP   HERE                 ; Wait for interrupt, if required,
0072  90                           NOP                          ; can put more instructions here
0073                            CODE ENDS
                                     END
```

(a)

```
                                ;8086 MODULE 2 PROCEDURES: F10-35B.ASM
                                ;ABSTRACT: Module 2 contains the interrupt service subroutines for Module 1.

                                PUBLIC CLOCK_TICK, KEYBOARD

0000                            DATA SEGMENT   WORD   PUBLIC      ;Tell assembler where to find
0000  00000000se                LOOP_ADDR_TABLE  DD     LOOP0     ;loop addresses used in this module
0004  00000000se                                 DD     LOOP1
0008  00000000se                                 DD     LOOP2
000C  00000000se                                 DD     LOOP3
0010  00000000se                                 DD     LOOP4
0014  00000000se                                 DD     LOOP5
0018  00000000se                                 DD     LOOP6
001C  00000000se                                 DD     LOOP7
                                EXTRN     COUNTER:BYTE, LOOPNUM:BYTE
0020                            DATA ENDS

                                ;Tell assembler where to find procedures used in this module
0000                            CODE SEGMENT WORD PUBLIC
                                EXTRN     LOOP0:FAR, LOOP1:FAR, LOOP2:FAR, LOOP3:FAR
                                EXTRN     LOOP4:FAR, LOOP5:FAR, LOOP6:FAR, LOOP7:FAR
0000                            CODE ENDS

0000                            INT_PROC SEGMENT WORD PUBLIC      ;Segment for interrupt service procedures
                                  ASSUME  CS:INT_PROC, DS:DATA

                                ;8086 INTERRUPT PROCEDURE CALLED CLOCK_TICK
                                ;ABSTRACT:   Services process control loops. Calls 1 of 8 process
                                ;            control loops on a rotating basis.
                                ;PORTS USED: None
                                ;PROCEDURES: Calls LOOP0,LOOP1,LOOP2,LOOP3,LOOP4,LOOP5,LOOP6,LOOP7
                                ;REGISTERS : Saves all

0000                            CLOCK_TICK  PROC     FAR
0000  50                           PUSH    AX                   ;Save registers
0001  53                           PUSH    BX
0002  52                           PUSH    DX
0003  1E                           PUSH    DS                   ;Save DS of interrupted program
0004  FB                           STI                          ;Enable higher interrupts if any
0005  B0 20                        MOV     AL, 00100000B        ;OCW2 for nonspecific EOI
0007  BA FF00                      MOV     DX, 0FF00H           ;Load address for OCW2
000A  EE                           OUT     DX, AL
000B  B8 0000s                     MOV     AX, DATA             ;Load DS needed here
000E  8E D8                        MOV     DS, AX
0010  FE 0E 0000e                 DEC     COUNTER              ;Decrement interrupt counter
0014  75 20                        JNZ     EXIT2                ;Not zero yet, go wait
0016  C6 06 0000e 14               MOV     COUNTER, 20          ;If zero, reset tick counter to 20
001B  B7 00                        MOV     BH, 00               ;Load BX with number of loop to
```

FIGURE 10-35 (*Continued*)

```
001D  8A 1E 0000e             MOV     BL, LOOPNUM          ;service and service that loop
0021  FF 9F 0000r             CALL    DWORD PTR LOOP_ADDR_TABLE[BX]
0025  80 06 0000e 04          ADD     LOOPNUM, 04          ;Point at next loop address
002A  80 3E 0000e 20          CMP     LOOPNUM, 20H         ;Was this the last loop?
002F  75 05                   JNE     EXIT2                ;No, exit
0031  C6 06 0000e 00          MOV     LOOPNUM, 00          ;Yes, get back to first loop
0036  1F                    EXIT2:POP DS                   ;Restore registers
0037  5A                      POP     DX
0038  5B                      POP     BX
0039  58                      POP     AX
003A  CF                      IRET
003B                        CLOCK_TICK  ENDP

                            ;DUMMY INTERRUPT PROCEDURE TO SERVICE KEYBOARD
003B                        KEYBOARD PROC FAR
                            ;     :                        ;Keyboard procedure intructions
003B  B0 20                   MOV     AL, 00100000B        ;OCW2 for non-specific EOI
003D  BA FF00                 MOV     DX, 0FF00H           ;Load address for OCW2
0040  EE                      OUT     DX, AL               ;and send OCW2 for end of interrupt
0041  CF                      IRET
0042                        KEYBOARD    ENDP

0042                        INT_PROC    ENDS
                                        END
```

(*b*)

```
                            ;8086 MODULE 3 PROCEDURES: F10-35C.ASM
                            ;ABSTRACT: Module 3 contains the procedures to service each loop

0000                        DATA SEGMENT WORD    PUBLIC
                                EXTRN   TIMEHI :BYTE, TIMELO  :BYTE  ;Imported into this
                                EXTRN   CURTEMP:BYTE, SETPOINT:BYTE  ;module from the mainline
0000                        DATA ENDS

                            PUBLIC LOOP0, LOOP1, LOOP2, LOOP3, LOOP4, LOOP5, LOOP6, LOOP7

0000                        CODE SEGMENT WORD PUBLIC
                            EXTRN DISPLAY_IT  : NEAR          ; These procedures can be
                            EXTRN A_D_READ    : NEAR          ; found in MODULE 4 which
                            EXTRN BINCVT      : NEAR          ; will be linked this module
0000                        CODE ENDS                         ; and MODULES 1 and 2

0000                        CODE SEGMENT WORD PUBLIC
                              ASSUME  CS:CODE, DS:DATA

                            ;8086 PROCEDURE - LOOP0
                            ;ABSTRACT :  This procedure services the temperature controller
                            ;REGISTERS:  Destroys none
                            ;PORTS:      Uses bit 7 of P2B (FFFAH) as output port control heater.
                            ;PROCEDURES: Uses DISPLAY_IT, A_D_READ, BINCVT from Module 4

0000                        LOOP0 PROC    FAR
0000  9C                      PUSHF                         ;Save registers
0001  50                      PUSH    AX
0002  53                      PUSH    BX
0003  51                      PUSH    CX
0004  52                      PUSH    DX
0005  FE 0E 0000e             DEC     TIMEHI              ;Decrement time for heater on
0009  75 50                   JNZ     EXIT                ;Return to interrupt procedure
000B  C6 06 0000e 01          MOV     TIMEHI, 01          ;Reset time high to fall through value
0010  BA FFFA                 MOV     DX, 0FFFAH          ;Point at output port P2B &
0013  B0 80                   MOV     AL, 80H             ;turn off heater
0015  EE                      OUT     DX, AL
0016  FE 0E 0000e             DEC     TIMELO              ;Decrement time for heater off
001A  75 3F                   JNZ     EXIT                ;Return to interrupt procedure
001C  B3 00                   MOV     BL, 00              ;Load channel address (0)
001E  E8 0000e                CALL    A_D_READ            ;Do A/D conversion
0021  A2 0000e                MOV     CURTEMP, AL         ;Save current temperature
0024  E8 0000e                CALL    BINCVT              ;Convert to BCD
0027  8A C8                   MOV     CL, AL              ;Put result in CX to display
0029  B5 00                   MOV     CH, 00
002B  B0 00                   MOV     AL, 00              ;temp in data field of SDK-86
```

FIGURE 10-35 (*Continued*)

```
47 002D  E8 0000e                      CALL    DISPLAY_IT
48 0030  A0 0000e                      MOV     AL, SETPOINT        ;Get setpoint temperature
49 0033  2A 06 0000e                   SUB     AL, CURTEMP         ;Get temperature & subtract from setpoint
50 0037  76 18                         JBE     DONE                ;Heater off if above or equal setpoint
51 0039  8A D0                         MOV     DL, AL              ;Save temperature difference
52 003B  B8 0064                       MOV     AX, 0064H           ;Compute new TIMELO
53 003E  F6 F2                         DIV     DL                  ; 0064/error, quotient is value
54 0040  A2 0000e                      MOV     TIMELO, AL          ; for new time low
55 0043  C6 06 0000e 04                MOV     TIMEHI, 04          ;Set time high for 4 loops
56 0048  B0 00                         MOV     AL, 00
57 004A  BA FFFA                       MOV     DX, 0FFFAH          ;Point at output port
58 004D  EE                            OUT     DX,AL               ;Turn on heater
59 004E  EB 0B 90                      JMP     EXIT
60 0051  C6 06 0000e 01        DONE:   MOV     TIMEHI, 01H         ;Fall through value for time high
61 0056  C6 06 0000e 7F                MOV     TIMELO, 7FH         ;Long off value for time low
62 005B  5A                    EXIT:   POP     DX                  ;Loop serviced - return to
63 005C  59                            POP     CX                  ; interrupt service procedure
64 005D  5B                            POP     BX
65 005E  58                            POP     AX
66 005F  9D                            POPF
67 0060  CB                            RET
68 0061                        LOOP0   ENDP
69
70                             ;DUMMY PROCEDURES FOR OTHER LOOPS INSERTED HERE
71 0061                        LOOP1   PROC FAR
72                             ;       :                           ;Instructions for this loop
73 0061  CB                            RET
74 0062                        LOOP1       ENDP
75
76 0062                        LOOP2   PROC FAR
77                             ;       :                           ;Instructions for this loop
78 0062  CB                            RET
79 0063                        LOOP2   ENDP
80
81 0063                        LOOP3   PROC FAR
82                             ;       :                           ;Instructions for this loop
83 0063  CB                            RET
84 0064                        LOOP3   ENDP
85
86 0064                        LOOP4   PROC FAR
87                             ;       :                           ;Instructions for this loop
88 0064  CB                            RET
89 0065                        LOOP4   ENDP
90
91 0065                        LOOP5   PROC FAR
92                             ;       :                           ;Instructions for this loop
93 0065  CB                            RET
94 0066                        LOOP5   ENDP
95
96 0066                        LOOP6   PROC FAR
97                             ;       :                           ;Instructions for this loop
98 0066  CB                            RET
99 0067                        LOOP6   ENDP
100
101 0067                       LOOP7   PROC FAR
102                            ;       :                           ;Instructions for this loop
103 0067  CB                           RET
104 0068                       LOOP7   ENDP
105
106 0068                       CODE    ENDS
107                                    END
```

(c)

```
  1                            ;8086 MODULE 4 PROCEDURES:   F10-35D.ASM
  2                            ;ABSTRACT : Module 4 contains the service procedures needed by the loop modules
  3
  4                            PUBLIC DISPLAY_IT, A_D_READ, BINCVT ; Make procedures available to other modules
  5
  6 0000                       DATA SEGMENT WORD PUBLIC
  7                                       ;       0    1    2    3    4    5    6    7
  8 0000  3F 06 5B 4F 66 6D 7D + SEVEN_SEG DB     3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H
  9       07
```

FIGURE 10-35 (*Continued*)

```
                                ;     8   9   A   b   C   d   E   F
0008 7F 6F 77 7C 39 5E 79 +     DB    7FH, 6FH, 77H, 7CH, 39H, 5EH, 79H, 71H
     71
0010                            DATA ENDS
0000                            CODE  SEGMENT WORD PUBLIC
                                   ASSUME CS:CODE, DS:DATA

                                ;8086 PROCEDURE DISPLAY_IT
                                ;ABSTRACT:  Displays a 4-digit hex or BCD number on LEDs of the SDK-86
                                ;INPUTS:    Data in CX, control in AL.
                                ;           AL =  00H  data displayed in data-field of LEDs
                                ;           AL <> 00H, data displayed in address field of LEDs.
                                ;PORTS:     Uses none
                                ;PROCEDURES:Uses none
                                ;REGISTERS: Saves all registers and flags

0000                            DISPLAY_IT  PROC  NEAR
0000 9C                             PUSHF                       ;Save flags
0001 1E                             PUSH    DS                  ;Save caller's registers
0002 50                             PUSH    AX
0003 53                             PUSH    BX
0004 51                             PUSH    CX
0005 52                             PUSH    DX
0006 BB 0000s                       MOV     BX, DATA            ;Initialize DS as needed for procedure
0009 8E DB                          MOV     DS, BX
000B BA FFEA                        MOV     DX, 0FFEAH          ;Point at 8279 control address
000E 3C 00                          CMP     AL, 00H             ;See if data field required
0010 74 05                          JZ      DATFLD              ;Yes, load control word for data field
0012 B0 94                          MOV     AL, 94H             ;No, load address-field control word
0014 EB 03 90                       JMP     SEND                ;Send control word
0017 B0 90                  DATFLD:MOV      AL, 90H             ;Load control word for data field
0019 EE                     SEND:  OUT      DX, AL              ;Send control word to 8279
001A BB 0000r                       MOV     BX, OFFSET SEVEN_SEG ;Pointer to seven-segment codes
001D BA FFE8                        MOV     DX, 0FFE8H          ;Point at 8279 display RAM
0020 8A C1                          MOV     AL, CL              ;Get low byte to be displayed
0022 24 0F                          AND     AL, 0FH             ;Mask upper nibble
0024 D7                             XLATB                       ;Translate lower nibble to 7-seg code
0025 EE                             OUT     DX, AL              ;Send to 8279 display RAM
0026 8A C1                          MOV     AL, CL              ;Get low byte again
0028 B1 04                          MOV     CL, 04              ;Load rotate count
002A D2 C0                          ROL     AL, CL              ;Move upper nibble into low position
002C 24 0F                          AND     AL, 0FH             ;Mask upper nibble
002E D7                             XLATB                       ;Translate 2nd nibble to 7-seg code
002F EE                             OUT     DX, AL              ;Send to 8279 display RAM
0030 8A C5                          MOV     AL, CH              ;Get high byte to translate
0032 24 0F                          AND     AL, 0FH             ;Mask upper nibble
0034 D7                             XLATB                       ;Translate to 7-seg code
0035 EE                             OUT     DX, AL              ;Send to 8279 display RAM
0036 8A C5                          MOV     AL, CH              ;Get high byte to fix upper nibble
0038 D2 C0                          ROL     AL, CL              ;Move upper nibble into low position
003A 24 0F                          AND     AL, 0FH             ;Mask upper nibble
003C D7                             XLATB                       ;Translate to 7-seg code
003D EE                             OUT     DX, AL              ;7-seg code to 8279 display RAM
003E 5A                             POP     DX                  ;Restore all registers and flags
003F 59                             POP     CX
0040 5B                             POP     BX
0041 58                             POP     AX
0042 1F                             POP     DS
0043 9D                             POPF
0044 C3                             RET
0045                            DISPLAY_IT   ENDP

                                ;8086 PROCEDURE A_D_READ
                                ;ABSTRACT:  Controls A/D converter
                                ;PORTS:     Uses Port P2A for input from A/D
                                ;           Port P2B, bit 7 = heater, bit 5 = start conversion
                                ;                     bit 4 = ALE     bits 2,1,0 = channel address
                                ;           Port P2C  bit 0 = end of conversion
                                ;INPUTS:    Channel address for A/D in BL
                                ;OUTPUTS:   A/D data in AL
                                ;REGISTERS: DESTROYS AL & BL

0045                            A_D_READ    PROC     NEAR
```

FIGURE 10-35 (*Continued*)

```
 83 0045  9C                    PUSHF
 84 0046  52                    PUSH    DX
 85 0047  B0 80                 MOV     AL, 80H          ;Control for heater off
 86 0049  0A C3                 OR      AL, BL           ;Combine with channel address
 87 004B  BA FFFA               MOV     DX, 0FFFAH       ;Point at P2B, output port
 88 004E  EE                    OUT     DX, AL           ; send
 89 004F  B0 90                 MOV     AL, 90H          ;Send ALE, keep heater on
 90 0051  0A C3                 OR      AL, BL           ;Keep channel address on
 91 0053  EE                    OUT     DX, AL
 92 0054  B0 B0                 MOV     AL, 0B0H         ;Send start of conversion
 93 0056  0A C3                 OR      AL, BL           ;Keep channel address on
 94 0058  EE                    OUT     DX, AL
 95 0059  B0 80                 MOV     AL, 80H          ;Turn off ALE and start
 96 005B  0A C3                 OR      AL, BL           ;Keep channel address
 97 005D  EE                    OUT     DX, AL
 98 005E  BA FFFC               MOV     DX, 0FFFCH       ;Point at port P2C
 99 0061  EC              EOCL: IN      AL, DX           ;Wait for end of conversion
100 0062  D0 D8                 RCR     AL, 01           ; to go low
101 0064  72 FB                 JC      EOCL
102 0066  EC              EOCH: IN      AL, DX           ;Wait for end of conversion
103 0067  D0 D8                 RCR     AL, 01           ; to go high
104 0069  73 FB                 JNC     EOCH
105 006B  BA FFF8               MOV     DX, 0FFF8H       ;Point at port P2A
106 006E  EC                    IN      AL, DX           ;Read data from A/D
107 006F  5A                    POP     DX
108 0070  9D                    POPF
109 0071  C3                    RET
110 0072                  A_D_READ      ENDP
111
112                       ;8086 PROCEDURE BINCVT
113                       ;ABSTRACT: Converts 8-bit binary number in AL to packed BCD equivalent in AL
114                       ;INPUTS:   AL - 8-bit binary number
115                       ;OUTPUTS:  AL - packed BCD result
116
117 0072                  BINCVT PROC  NEAR
118 0072  9C                    PUSHF                    ;Save registers and flags
119 0073  51                    PUSH    CX
120 0074  B4 09                 MOV     AH,09H           ;Bit counter for 8 bits
121 0076  8A C8                 MOV     CL, AL           ;Save binary in CL
122 0078  B5 00                 MOV     CH, 00           ;Clear CH for use as buffer
123 007A  32 C0           CNVT2: XOR    AL, AL           ;Clear AL and carry
124 007C  FE CC                 DEC     AH               ;Decrement bit counter
125 007E  75 03                 JNZ     GO_ON            ;Do all bits
126 0080  EB 0C 90              JMP     HOME             ;Done if AH down to zero
127 0083  D0 D1           GO_ON: RCL    CL,1             ;MSB from CL to carry
128 0085  8A C5                 MOV     AL, CH           ;Move BCD digit being built to AL
129 0087  12 C0                 ADC     AL, AL           ;Double AL and add carry from CL shift
130 0089  27                    DAA                      ;Keep result in BCD form
131 008A  8A E8                 MOV     CH, AL           ;Put back in CH for next time through
132 008C  EB EC                 JMP     CNVT2            ;Continue conversion
133 008E  8A C5           HOME: MOV     AL, CH           ;BCD in AL for return
134 0090  59                    POP     CX               ;Restore registers
135 0091  9D                    POPF
136 0092  C3                    RET
137 0093                  BINCVT ENDP
138
139 0093                  CODE   ENDS
140                             END
```

(*d*)

FIGURE 10-35 (*Continued*)

these operations in detail, so we won't dwell on them here. Also in the mainline we initialize some process variables. We will explain these initializations later when they will have more meaning.

After enabling the 8086 INTR input with an STI instruction, the program then enters a loop and waits for an interrupt from the user via the keyboard, or an interrupt from the timer. The keyboard-interrupt procedure would normally contain a command recognizer and subprocedures to implement commands which allow the user to change set points, stop a process, or examine the value of process variables at any time. Due to severe space limitations, we can't show here the implementation of the keyboard interrupt procedure, but we will show you how the timer interrupt procedure and the example PID loop procedure work.

THE CLOCK_TICK INTERRUPT HANDLER

As we said before, the 8254 is programmed to send a pulse to an interrupt input of the 8259A every millisec-

ond. When a clock interrupt occurs, execution goes to the CLOCK_TICK procedure. At the start of this procedure, we simply decrement an interrupt "tick counter" kept in a memory location called COUNTER. In the initialization, this counter was set to 20 decimal or 14H. If the counter is not down to 0, execution is simply returned to the wait loop in the mainline. If the tick counter is now down to 0, the clock-tick counter is reset to 20, and one of the loop procedures is called to service the next loop. It is important that this clock-tick procedure be reentrant because if one of the loop procedures takes more than the time between clock ticks (1 ms), the CLOCK_TICK procedure will be reentered before its first use is completed. The procedure is made reentrant by pushing all registers used in the procedure and by immediately resetting the clock-tick counter to 20. If a loop procedure takes longer than 1 ms and the clock_tick procedure is called again, the tick counter will be decremented by 1 and execution returned to the interrupted loop procedure.

Selecting one of the loop procedures is an example of the CASE or nested IF-THEN-ELSE programming structure described in Chapter 3. To implement this structure, we use a powerful programming technique called a *call table*. Here's how it works.

The starting addresses of all the loop procedures are put in a table called LOOP_ADDR_TABLE, as shown at the start of module 2 in Figure 10-35*b*. The names LOOP0, LOOP1, LOOP2, etc., are the names of the procedures to service each of the loops. Since these procedures are FAR, the DD directive is used to reserve space for the IP and CS of each. When this program module is linked and loaded into memory, the instruction pointer and code segment addresses for each of the loop procedures will be loaded into this table.

To point to the desired loop procedure address in the table, we use a variable called LOOPNUM. During initialization LOOPNUM is loaded with 00H. When it is time to service the first loop, the value in LOOPNUM is loaded into BX. The CALL DWORD PTR LOOP_ADDR_TABLE [BX] instruction then gets the address of the LOOP0 procedure from the call table and goes to that address. For the first access to the table, BX is zero, so the first address in the table is used to call LOOP0 procedure.

When execution returns from the LOOP0 procedure to the interrupt handler, we add 4 to the LOOPNUM. This is done so that LOOP1 will be called the next time the tick counter is counted down to zero. LOOPNUM must be incremented by 4 because each address in the call table uses 4 bytes. When all loops have been serviced, LOOPNUM is set back to 0 so LOOP0 will be serviced again. Now let's look at the actual temperature-control loop.

THE TEMPERATURE-CONTROLLER PROCEDURE

As we said previously, the amount of heat output by the heater is controlled by the duty cycle of a pulse waveform sent to the solid-state relay. The time on for the output waveform to the solid-state relay is determined by counting down a value called TIMEHI. The time off for this waveform is determined by counting down a value called TIMELO. At start-up the mainline program initializes TIMEHI and TIMELO to 01H, so that the first time the LOOP0 procedure is called both of these are decremented to 0, and execution falls through to the A/D conversion procedure. This is done to get a temperature value which can be compared with the set-point value. The difference between the set-point value and the actual temperature will be used to set the value of TIMEHI and TIMELO for the next time the LOOP0 procedure is called.

The part of the program which controls the A/D converter is written as a separate procedure so that it can be used in other PID loops. The A/D converter has eight input channels, so as part of the interaction with the A/D converter, we have to tell it which channel to digitize. The number of the A/D channel that we want to digitize is passed to the A/D conversion procedure in the AL register. The procedure then sends out this channel number to the A/D converter and generates the control waveforms shown in Figure 10-34. Since the timing for these waveforms is in the range of microseconds, we chose to generate the waveforms with program instructions rather than use an 8255A in handshake mode. The binary value for the temperature is returned in AL.

Upon return, the binary value of the temperature is stored in a memory location called CURTEMP for future reference. For testing purposes, we wanted to display the temperature on the address field of the SDK-86 display. To do this, the binary value for the temperature is converted to a BCD value using a reduced version of the binary-to-BCD procedure from the scale program earlier in this chapter and the display routine from Chapter 9.

After the current temperature is displayed, it is compared with the set-point temperature to see if the heater needs to be turned on. If the temperature is at or above the set point, TIMEHI is loaded with fall-through value and TIMELO is loaded with a large number.

If the temperature is below the set point, we call a procedure, DUTY_CYCLE, which computes the correct values for TIMEHI and TIMELO based on the difference between the set point and the current temperature. In a more critical application, a complex PID algorithm might be used for this procedure. For our example here, however, we have used simple proportional feedback. To further simplify the calculations, a fixed value of 4 was used for TIMEHI. The thinking for the value of TIMELO then goes as follows.

If the difference in temperature is large, then TIMELO should be small so the heater is on for a longer duty cycle. If the difference in temperature is small, then the value of TIMELO should be large so the heater has a short duty cycle. Experimentally we found that a good first approximation for our system was (difference in temperature) × TIMELO = 100 decimal (64H). For example, if the difference in temperature is 20° (14H), then 64H/14H gives a value of 5 for TIMELO. The values for TIMEHI and TIMELO are returned in their named memory locations. Upon return to the main loop procedure, we send a control word which turns on the heater. Execution then jumps to EXIT.

When execution returns to loop 0 again after 160 ms,

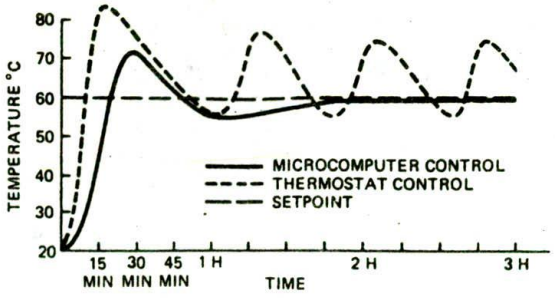


FIGURE 10-36 Temperature versus time responses for thermostat-controlled and microcomputer-controlled heaters.

TIMEHI will be decremented. If TIMEHI is not yet down to 0 after the decrement, then we simply adjust a few things and return. If TIMEHI is 0 after the decrement, the heater is turned off, and TIMELO is decremented. TIMELO is then decremented every time loop 0 is serviced (every 160 ms) until TIMELO reaches 0. When TIMELO gets counted down to 0, a new A/D conversion is done, and a new feedback value for TIMELO is calculated.

An important point here is that the part of the program that determines the feedback is separate from the rest of the program, so it can be easily altered without changing the rest of the program. All that needs to be changed in this procedure is the value of TIMEHI, the value of TIMELO, and the rate at which these change in response to a difference in temperature to produce proportional, integral, and derivative feedback control.

TEMPERATURE CONTROLLER RESPONSE

The dotted line in Figure 10-36 shows the temperature versus time response of our system with traditional thermostat control, which is often called *on-off control* or "bang-bang" control. As you can see, with thermostat control the temperature initially overshoots the set point a great deal and then oscillates over a wide range around the set point. The solid line in Figure 10-36 shows the response of the system operating with our temperature controller program. The initial overshoot was caused by the large thermal inertia of the hot plate we used. The overshoot and the residual error of about 1° could be eliminated by using a more complex feedback algorithm. This example should make you aware of the advantages of computer feedback control.

DEVELOPING THE PROTOTYPE OF A MICROCOMPUTER-BASED INSTRUMENT

The first step in developing a new instrument is to very carefully define exactly what you want the instrument to do. The next step is to decide which parts of the instrument you want to do in hardware and which parts you want to do in software. You then can decide how you want to do each of these.

For the software, you will break the overall programming job down into modules which can be individually tested and debugged, as we have described previously. Likewise, the best way to develop the hardware is in small parts which can be individually tested and debugged. To give you a specific example of how to do this, here's how we developed the SDK-86-based factory-control system described in the preceding section.

The basic SDK-86 does not have a timer to produce 1-kHz clock ticks or a priority interrupt controller to handle keyboard and clock-tick interrupts. Therefore, we first added these two devices and some address decoder circuitry to the SDK-86, as shown in Figure 8-14. To test this circuitry we used a short program which wrote a byte to the starting address for the timer over and over again. We ran this test program with an emulator such as the Applied Microsystems ES1800 shown in Figure 3-17. With the program running, we used a scope to check if the $\overline{CS}$ input of the timer was getting asserted. It was, so we knew that the address decoding circuitry was working correctly.

We then connected the 2.45-MHz PCLK signal to the clock inputs of all three timers in the 8254 and wrote the instructions needed to initialize the three timers for 1-kHz square-wave outputs. Even though we need only one timer here, it was very little additional work to check the other two for future reference. Hurrah, the timers worked the first time; now on to the 8259A priority interrupt controller (PIC).

Testing the 8259A was a little more complex because we had to provide an interrupt signal, initialize the 8259A, initialize the interrupt vector table in low RAM, and provide a location for execution to go to when the PIC received an interrupt. We used the 1-kHz clock tick from the timer as the interrupt signal to the 8259A. For 8259A initialization and the interrupt jump table initialization, we used the instructions in the mainline program in Figure 10-35. For the test-interrupt procedure, we actually used a real-time clock and display procedure that we developed for examples in previous chapters. We used these so that we could see if the interrupt mechanism was working correctly by watching the displays on the SDK-86 count off seconds. This again shows the advantage of writing programs as separate, reusable modules. Note in the program in Figure 10-35 that we initialize the 8259A before we initialize and start the timer. When we first wrote a test program to test an 8259A and an 8254, we did this in the reverse order. When we ran the test program with the emulator, the system would accept only one interrupt and then hang up. We did a trace with the emulator and found that execution was returning from the interrupt procedure to the WAIT loop in the mainline program properly, but it was not recognizing the next interrupt. Careful reading of the 8259A data sheet showed us that we had to initialize the 8259A *before* we started sending it interrupt signals, or it would not respond correctly to the nonspecific EOI command that we used at the end of the interrupt procedure to reset the 8259A's in-service register.

After the interrupt mechanism was working correctly, we wrote the interrupt procedure which implements the decision structure shown in Figure 10-32*b*. Initially we made all eight loops dummy loops to test the basic

structure. By inserting breakpoints with the emulator, we were able to see whether execution was getting to each of the eight loops. When all this was working, we went on to build and test the temperature-control section.

For the temperature-control section, we first built the analog circuitry and tested it. Then we wrote a small program to read the temperature from the A/D converter and display the result on the SDK-86 displays. Initially then, the loop 0 procedure simply read in the temperature, displayed it in binary (hex) form, and returned. This worked the first time, so we went on to add the binary-to-BCD conversion routine and run the result with the emulator. This was a previously written and tested module, and when it was added, the result worked fine.

Next we added a couple of instructions to turn the heater on during one execution of loop 0 and turn the heater off during the next time through loop 0. We then used an oscilloscope to check that the solid-state relay was getting turned on and off correctly.

Finally, we added the actual duty cycle and control instructions and sat back waiting for the system to heat up a big container of water for tea.

The actual development cycle will obviously be somewhat different for every instrument developed. The main points here are to develop and test both the hardware and the software in small modules. To speed up the debugging process, take the time to learn to use all or most of the power of the emulator and system you are working with.

ROBOTICS AND EMBEDDED CONTROL

In recent years the term *robot* has become a "buzzword" in the media and in many people's minds. Science fiction movies have helped us form an image of robots as mobile, rational companions. Robots, however, have many forms, and in operation they are simply a combination of feedback control systems such as we described in the previous section. This is why we have not included a chapter dedicated just to robotics. The controller for the Rhino robot arm shown in Figure 9-42, for example, uses optical encoders to detect the position of its different joints, motors (actuators) to move each joint to a desired position, and a microcomputer to control the motors based on feedback from the sensors. Large industrial robots such as those that weld or spray-paint cars may also use tactile or visual sensors, and the actuators may be hydraulic or pneumatic, but the control principle is the same. A microcomputer or several microcomputers use feedback from the various sensors to control one or more actuators.

Most of you have probably used some simple robots around your home without realizing it. One example is an electric garage door opener which starts to open or close when you tell it to and then stops when a sensor indicates that it is open or closed as desired. Common household examples of microcomputer-controlled robots are a microwave oven with a temperature probe, a programmable sewing machine, a remote-control stereo system, etc.

Smart machines such as these usually use specially designed microprocessors called *embedded controllers* instead of a general-purpose microprocessor such as the 8086 which we used in the scale and the factory controller examples. The main differences of these embedded controlled microprocessors is that they have additional functions included on the chip with the basic CPU and they have special instructions for working with individual bits in a word. In the following sections we give you an overview of a few common embedded controller families. Consult the appropriate Intel handbooks for additional details when you need them.

The Intel 8051 Embedded Controller Family

Figure 10-37*a* shows a block diagram of a basic 8051 family controller and Figure 10-37*b* summarizes some of the features of the members of the family. These controllers are 8-bit units which can address up to 64 Kbytes of memory. All of the family members have some RAM on the chip, and different members of the family have some ROM or EPROM also included on the chip. As shown in Figure 10-37*b*, members of this family also have programmable timers and priority interrupt controllers included on the chips.

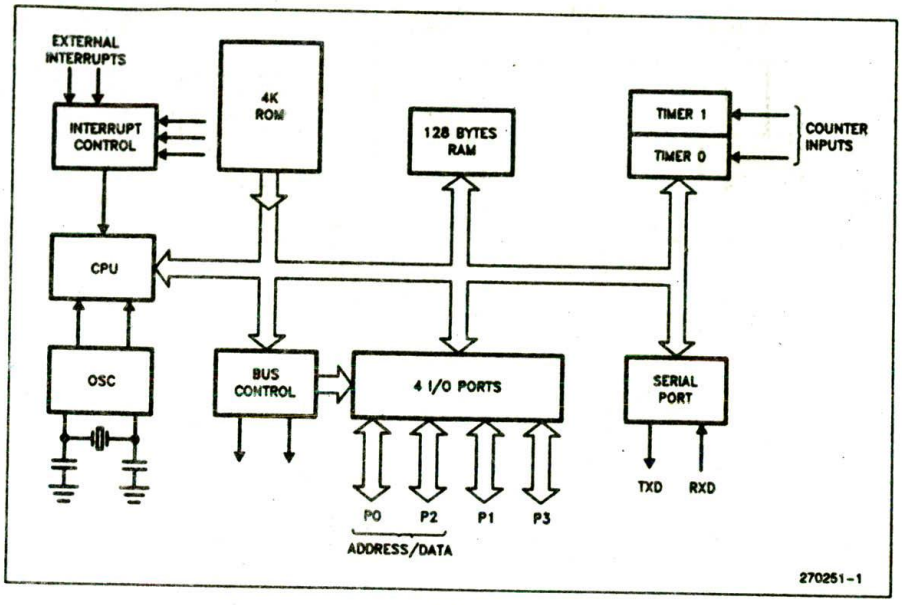
If an application does not require any memory other than that included on the chip, then all four parts are available for use as input or output ports. If additional memory is needed, then port 0 and port 2 can be programmed to function as a multiplexed address/data bus. When used with external memory, two lines on port 3 are used to generate the $\overline{RD}$ and $\overline{WR}$ signals.

The devices in the 8051 family also contain serial data interface circuitry. When this feature is used, two pins on port 3 function as the RxD and TxD lines. Figure 10-37*c* shows the special uses of the port 3 pins.

The Intel 8096 Embedded Controller Family

Figure 10-38*a*, p. 334, shows a block diagram for the Intel 8096 family of 16-bit microcontrollers. One of the most important features of the members of this family is the 232-byte register file and the register ALU (RALU) shown in the center of Figure 10-38*a*. Instead of using a single accumulator register as the 8086 does, the ALU in the 8096 family devices can perform most operations on any of the registers in the register file. This structure is referred to as *register-to-register* architecture. The large number of registers makes it possible to have many data bytes in registers where they can be very quickly accessed. Also, since the contents of any register can be output to a port, the I/O "bottleneck" of 8086-type processors is eliminated. The 8086, remember, requires that data be output from or input to AL/AX.

Other features found in all the 8096 family devices are five ports which can be programmed for use in a variety of ways. In addition to their use as standard ports, ports 3 and 4 can be used as a multiplexed address/data bus to access external memory and ports. Port 2 can be programmed for use as a serial port and/or an output for the pulse-width-modulated signal. The



(*a*)

Device	Internal Memory		Timers/ Event Counters	Interrupts
	Program	Data		
8052AH	8K x 8 ROM	256 x 8 RAM	3 x 16-Bit	6
8051AH	4K x 8 ROM	128 x 8 RAM	2 x 16-Bit	5
8051	4K x 8 ROM	128 x 8 RAM	2 x 16-Bit	5
8032AH	none	256 x 8 RAM	3 x 16-Bit	6
8031AH	none	128 x 8 RAM	2 x 16-Bit	5
8031	none	128 x 8 RAM	2 x 16-Bit	5
8751H	4K x 8 EPROM	128 x 8 RAM	2 x 16-Bit	5
8751H-8	4K x 8 EPROM	128 x 8 RAM	2 x 16-Bit	5

(*b*)

Port 3 also serves the functions of various special features of the MCS-51 Family, as listed below:

Port Pin	Alternative Function
P3.0	RXD (serial input port)
P3.1	TXD (serial output port)
P3.2	$\overline{\text{INT0}}$ (external interrupt 0)
P3.3	$\overline{\text{INT1}}$ (external interrupt 1)
P3.4	T0 (Timer 0 external input)
P3.5	T1 (Timer 1 external input)
P3.6	$\overline{\text{WR}}$ (external data memory write strobe)
P3.7	$\overline{\text{RD}}$ (external data memory read strobe)

(*c*)

FIGURE 10-37 8051 family. (*a*) Block diagram. (*b*) Family features. (*c*) Port pin uses. (*Intel Corporation*)

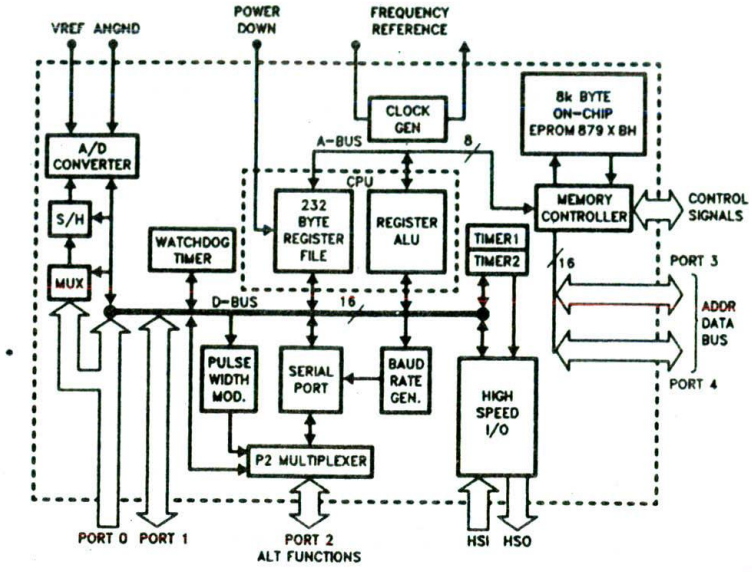
PWM signal is software programmable and can be used to control the speed of a small motor or the duty cycle of a heater, as we described earlier in the chapter. The 8096 family devices also have two programmable counters and 21 hardware and software interrupt types. Note that both the clock generator and the baud-rate generator are included in the basic architecture.

Figure 10-38*b* shows the numbering for the different members of the 8096 family so you can see the options available in different parts. As you can see, devices are available with 8 Kbytes of internal EPROM, 8 Kbytes of internal mask-programmed ROM, or no internal ROM. Also note that some members of the 8096 family contain a 10-bit successive-approximation A/D converter. As shown in Figure 10-38*c*, this A/D has a sample-and-hold and an 8-input analog multiplexer on its input. This allows it to digitize any of eight input signals under program control.

The 8096 instructions are designed for fast operations on registers and for easily working with individual bits in data words. The 8096 also has multiply and divide instructions. From the scale and temperature controller examples earlier in the chapter, you should see that these features optimize the devices for use in hardware control applications.

The 80186 and 80188 Microprocessors

The 8051 and 8096 embedded controllers we described in the preceding sections have different instruction sets and very different architectures from the 8086, which

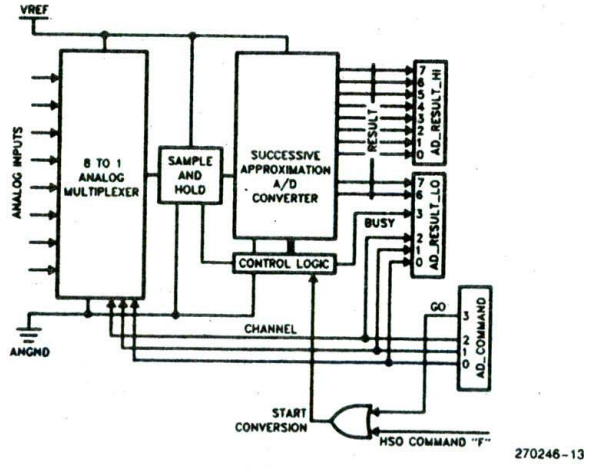


(*a*)

The MCS®-96 Family Nomenclature

		Without A/D	With A/D
ROMless 809XBH	48 Pin		C80958H · Ceramic DIP P80958H · Plastic DIP
	68 Pin	A8096BH · Ceramic PGA N8096BH · PLCC	A8097BH · Ceramic PGA N8097BH · PLCC
ROM 839XBH	48 Pin		C8395BH · Ceramic DIP P8395BH · Plastic DIP
	68 Pin	A8396BH · Ceramic PGA N8396BH · PLCC	A8397BH · Ceramic PGA N8397BH · PLCC
EPROM 879XBH	48 Pin		C8795BH · Ceramic DIP
	68 Pin	A8796BH · Ceramic PGA R8796BH · Ceramic LCC	A8797BH · Ceramic PGA R8797BH · Ceramic LCC

(*b*)



(*c*)

FIGURE 10-38 8096 family. (*a*) Block diagram. (*b*) Family features. (*c*) A/D converter diagram. (*Intel Corporation*)

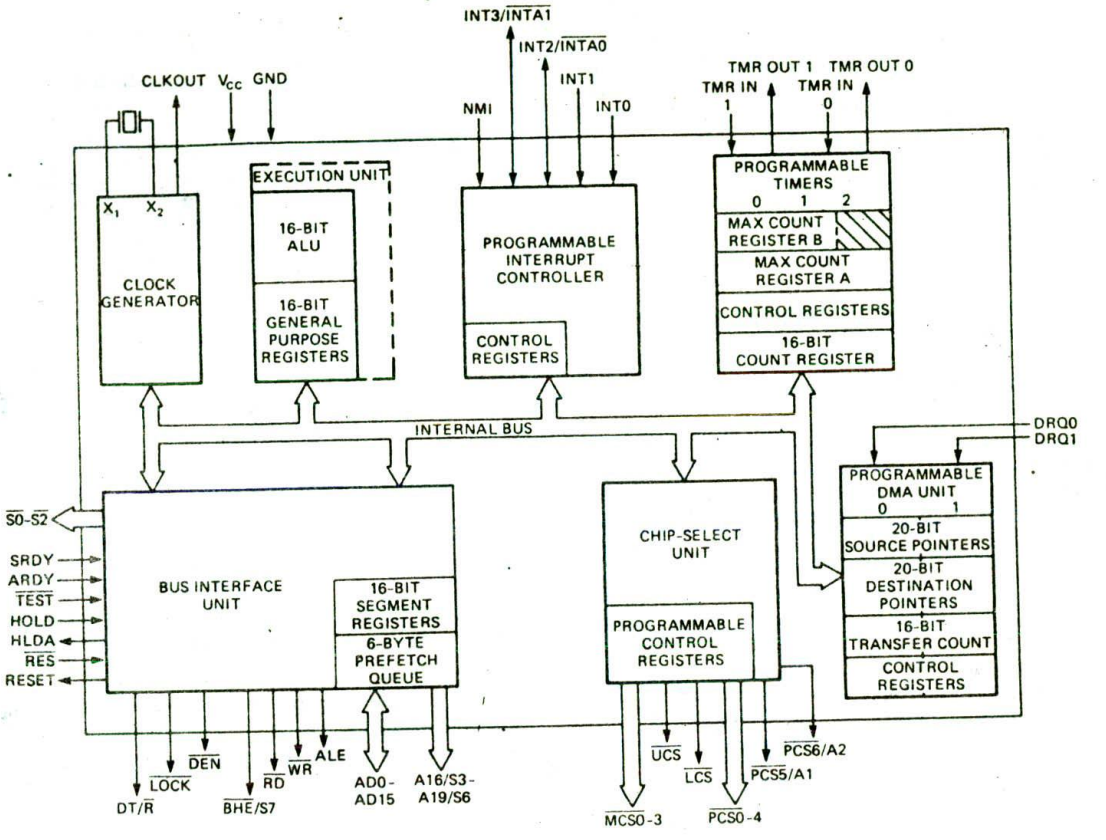


FIGURE 10-39 80186 Internal block diagram. (*Intel Corporation*)

we have used for examples in this book. The 80186 and 80188 are 16-bit processors commonly used for embedded control applications, and they have basically the same instructions set as the 8086. However, as shown in Figure 10-39, they contain some peripheral functions as well as a CPU.

The architecture and instruction set of the 80188 are identical to those of the 80186 except that the 80188 has only an 8-bit data bus instead of the 16-bit data bus that the 80186 has. With this in mind, we will use the 80186 to represent both the 80186 and the 80188 in our discussions here.

The 80186 has the same bus-interface unit and execution unit as the 8086 which we discussed previously, so there is nothing new there for you. Unlike the 8086, however, the 80186 has the clock generator built in so that all you have to add is an external crystal. Also note that the 80186 does not have a pin labeled MN/$\overline{MX}$. The 80186 is packaged in a 68-pin leadless package, so it has enough pins to send out both the minimum-mode-type signals $\overline{RD}$ and $\overline{WR}$ and the S0–S3 status signals which can be connected to external bus controller ICs for maximum-mode systems. Now let's look at the four peripheral chip function blocks in the 80186.

First is a priority interrupt controller which has up to four interrupt inputs, INT0, INT1, INT2/$\overline{INTA0}$, and INT3/$\overline{INTA1}$ as well as an NMI interrupt input. If the four INT inputs are programmed in their internal mode, then a signal applied to one of them will cause the 80186 to push the return address on the stack and vector directly to the start of the interrupt service procedure for that interrupt. The INT2/$\overline{INTA0}$ and INT3/$\overline{INTA1}$ pins can be programmed to be used as interrupt inputs, or they can be programmed to function as interrupt acknowledge outputs. This mode is used to interface with external 8259As. The interrupt request line from an external 8259A is connected to, for example, the 80186 INT0 input, and the 80186 INT2/$\overline{INTA0}$ pin is connected to the interrupt acknowledge input of the 8259A. When the 8259A receives an interrupt request, it asserts the INT0 input of the 80186. When the 8259A receives interrupt acknowledge signals from the INT2/$\overline{INTA0}$ pin, its sends the desired interrupt type to the 80186 on the data bus.

Next to look at in the block diagram is the built-in address decoder, referred to in the drawing as the chip select unit. This unit can be programmed to produce an active low chip select signal when a memory address in the specified range or a port address in a specified range is sent out. Six memory address chip select signals are available: $\overline{LCS}$, $\overline{UCS}$, and $\overline{MCS0}$ through $\overline{MCS3}$. The *lower-chip select* signal, $\overline{LCS}$, will be asserted by ad-

dresses between 00000H and some address which you specify in a control word. The specified ending address can be anywhere between 1K and 256K. The highest address that will assert the *upper-chip select* signal, UCS, is fixed at FFFFFH. The lowest address for this block of memory is again programmable by some bits you put in a control word. The size of the upper memory block can be anywhere between 1K and 256K. Finally, there are four *middle-chip select* lines, MCS0 through MCS3. Each of these four is asserted by an address in a block of memory in the middle range of memory. Both the starting address and the size of the four blocks can be specified for this middle-range block. The specified size of blocks can be anywhere from 2K to 128K.

In addition to producing memory chip select signals, the 80186 can be programmed to produce up to seven peripheral chip select signals on its PCS0 through PCS4, PCS5/A1, and PCS6/A2 pins. You program a base address for these I/O addresses in a control word. PCS0 will be asserted when this base address is output during an IN or an OUT instruction. The other PCS outputs will be asserted by addresses at intervals of 128 bytes above the base address.

Now let's look at the programmable DMA unit in the 80186. As you can see from the block diagram in Figure 10-39, the DMA unit has two DMA request inputs, DRQ0 and DRQ1. These inputs allow external devices such as disk controllers, CRT controllers, etc. to request use of the microcomputer address and data bus so that data can be transferred directly from memory to the peripheral or from the peripheral to memory without going through the CPU. In the next chapter we show you the details of how a DMA controller manages this transfer. For each DMA channel, the 80186 has a full 20-bit register to hold the address of the source of the DMA transfer, a 20-bit register to hold the destination address, and a 16-bit counter to keep track of how many words or bytes have been transferred. DMA transfers can be from memory to memory, from I/O to I/O, or between I/O and memory.

Finally, let's look at the three 16-bit programmable counter/timers in the 80186. The inputs and outputs of counters 0 and 1 are available on pins of the 80186. These two counters can be used to divide down the frequency of external signals, produce programmed-width pulses, etc., just as you do with the counters in an 8254. You can also internally direct the processor clock to the input of one of these counter inputs by clearing the appropriate bit in a control word. The input of the third number in the 80186 is internally connected to the processor clock.

As you can see from the preceding discussion, the 80186 contains many of the peripheral chip functions needed in a medium-complexity microcomputer system. In order to use these integrated peripherals, you have to initialize them just as you do external peripherals. If you have to work with an 80186, you can find the formats for these words in the 80186 data sheet, and work out the control words you need for your particular application on a bit-by-bit basis, just as you do for the separate peripherals. You may also find Intel Application Note 186, *Introduction to the 80186 Microprocessor*, helpful.

The 10 additional instructions that the 80186 has are as follows:

ENTER	—	Enter a procedure
LEAVE	—	Leave a procedure
BOUND	—	Check if an array index in a register is in range of array
INS	—	Input string byte or string word
OUTS	—	Output string byte or string word
PUSHA	—	Push all registers on stack
POPA	—	Pop all registers off stack
PUSH immediate	—	Push immediate number on stack
IMUL destination register, source, immediate	—	Immediate × source to destination
SHIFT/ROTATE destination, immediate	—	Shift register or memory contents specified immediate number of times

The 80960 Embedded Controller

The 80960 family controllers are 32-bit devices which are an evolutionary step up from the 8096 family. They are built with a register-to-register architecture for fast processing, and they contain code and data caches. In Chapter 11 we explain how these caches also help speed up program execution.

The devices in the 80960 family also contain a floating-point processor which performs mathematical operations on 80-bit floating-point numbers. In the next chapter we show you how a floating point processor such as this works.

DIGITAL SIGNAL PROCESSING AND DIGITAL FILTERS

The term *digital signal processing*, or *DSP*, is a very general term used to describe any system which takes samples of a signal with an A/D converter, processes the samples with a microcomputer, and outputs the computed results to a D/A converter or some other device. The process-control system and temperature controller we described earlier in the chapter is one example of digital signal processing. Other applications of DSP include antiskid braking and engine-control systems on automobiles, speech recognition and synthesis systems, and contrast enhancement of images sent back from satellites and planet probes. When most people think of DSP, however, the thought that probably comes to mind first is a special one called a *digital filter*.

A section at the start of this chapter showed how op amps can be used to build high-pass and low-pass filter circuits. In this section of the chapter, we show how filtering of a signal can also be done by taking samples of the signal with an A/D converter, performing mathe-

matical operations on the samples with a microcomputer, and outputting the results to a D/A converter. This digital filter approach can easily produce a filter response which is difficult, if not impossible, to produce with analog circuitry. The digital approach has the further advantage that the filter response can be changed under program control.

To most people it is not intuitively obvious how an A/D converter, microcomputer, and D/A converter can produce the same effect on a signal as, for example, an RC low-pass filter. Before we can show you how digital filters work, we need to review some basic signal relationships and analog filter characteristics.

Time-Domain and Frequency-Domain View of a Square Wave

There are two ways of producing or describing a waveform such as the square wave. One way is with a circuit such as that in Figure 10-40*a*. If the switch is repeatedly flipped up for one-half the period and down for one-half the period, the output waveform will be a square wave centered around 0 V. This way of producing or describing a square wave is referred to as the *time-domain* method.

The second way of producing a square wave of a given frequency is by adding together a series of sine-wave signals which have just the right amplitude and phase relationships. This method of producing or describing a square wave is called the *frequency-domain* method. Figure 10-40*b* shows a circuit which generates a square wave by adding sine-wave signals.

Remember from basic electric circuits that when voltage sources are connected in series, the output voltage at any time is the sum of the individual voltages. The lowest-frequency sine-wave signal added here has the same frequency as the desired square-wave signal. Added to this is a signal with a frequency 3 times the frequency of the fundamental frequency, a signal with a frequency 5 times the frequency of the fundamental frequency, a signal with a frequency 7 times the fundamental frequency, etc. These multiples of the fundamental frequency are called *harmonics*. As we said before, the added harmonics must have the right amplitude and phase relationships to produce a square wave when added. Figure 10-40*c* shows the required phase and relative amplitude relationships.

To help you further visualize this addition process, Figure 10-40*d* shows the resultant waveform that will be produced by adding just the fundamental frequency and the third harmonic. It is somewhat difficult to show, but the more harmonics you add, the more the resultant waveform approaches a square wave.

Mathematically, the equation for this square wave can be expressed in terms of a fundamental frequency and harmonics as

$$v(t) = \frac{4}{\pi}\sin(2\pi ft) + \frac{1}{3}\sin 3(2\pi ft) + \frac{1}{5}\sin 5(2\pi ft) + \frac{1}{7}\sin 7(2\pi ft) + \cdots$$

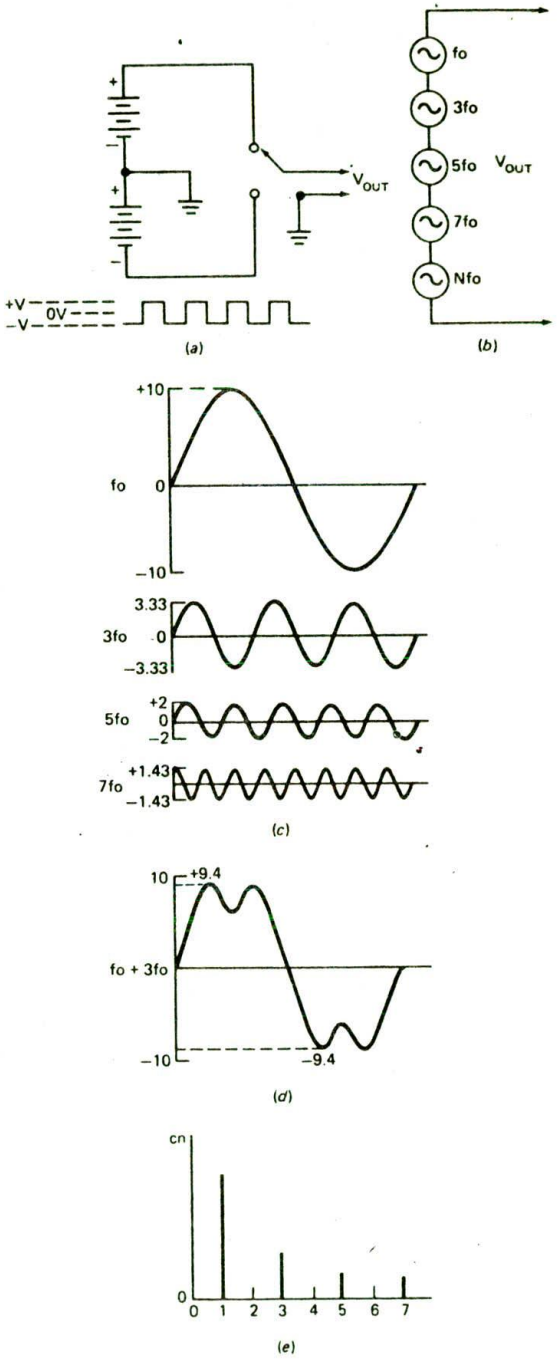


FIGURE 10-40 (*a*) Time-domain method of producing a square wave. (*b*) Frequency-domain method of producing a square wave. (*c*) Amplitude and phase relationships of first, third, fifth, and seventh harmonics which produce a square wave when added. (*d*) Addition of first and third harmonics. (*e*) Amplitude versus frequency graph (frequency spectrum) for square wave.

The terms in the right-hand side of this equation are referred to as a *Fourier series*. As it turns out, any periodic waveform can be described with a Fourier series, but for now, we will just stay with a square wave.

When we want to graphically represent the frequency components in a signal, it is very messy to draw waveforms such as those in Figure 10-40*c*. Therefore, we usually use an amplitude-versus-frequency graph, such as that in Figure 10-40*e*. From this graph you can easily see that the frequency components of our square wave are a fundamental frequency with a relative amplitude of 1, a third harmonic with a relative amplitude of $\frac{1}{3}$, a fifth harmonic with a relative amplitude of $\frac{1}{5}$, a seventh harmonic with a relative amplitude of $\frac{1}{7}$, etc. Later we will use this graph to help describe the effect of a low-pass filter on a square-wave signal.

Time-Domain View and Frequency-Domain View of a Low-Pass Filter

If a square wave is passed through the simple RC circuit shown in Figure 10-41*a*, the output waveform will look like that in Figure 10-41*b*. The time-domain explanation for this output waveform is that it takes time for the capacitor to charge through the resistor, so the risetime and falltime of the output signal will be increased. As you may remember, the 10% to 90% risetime for the output signal is $t_R = 2.2RC$.

Now, if you think of the square wave as a combination of harmonically related sine waves, as shown in Figure 10-40*e*, you can easily describe the operation of the circuit in the frequency domain. In the frequency domain, this circuit acts as a low-pass filter. This means that it passes the fundamental frequency but reduces the amplitude of the harmonics. As we showed you in Figure 10-40*d*, adding harmonics to the fundamental frequency is what produces the square wave, so reducing the amplitude of the harmonics will change the output waveform. For the simple RC circuit in Figure 10-41*a*, the critical frequency, or in other words the frequency at which a sine-wave input signal will be attenuated to 0.707 of its input value, is $f_C = 1/(2\pi\ RC)$. Above the critical frequency the output decreases by a factor of 10 for each increase of 10 in frequency. This means that the upper harmonics will be attenuated more than the lower harmonics, and mathematically it can be shown that the result is the waveform in Figure 10-41*b*.

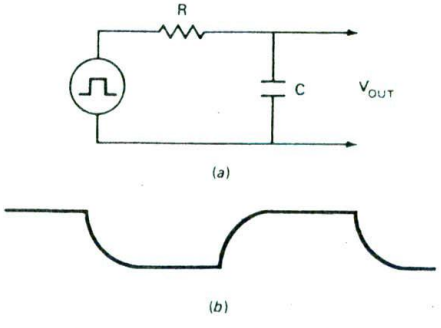


FIGURE 10-41 (*a*) Simple RC circuit. (*b*) Output waveform from RC circuit.

You can pull the time-domain view of this circuit and the frequency-domain view into the same equation as follows:

$$t_R = 2.2RC \qquad \text{so } RC = \frac{t_R}{2.2}$$

$$f_C = \frac{1}{2\pi\ RC} \qquad \text{so } f_C = \frac{2.2}{2\pi\ t_R}$$

$$f_C = \frac{0.35}{t_R}$$

The point of the preceding discussions was to show you the two equivalent ways of describing the operation of a filter circuit. When we are designing analog filters, such as those made with simple resistors and capacitors, we usually think and work in the frequency domain. When we are designing digital filters, we usually think and work in the time domain. Now let's see how you can use an A/D, microcomputer, and D/A converter to function as a digital filter which modifies the waveform/frequency composition of a signal.

Digital Filters

The basic principle of a digital filter is to take continuous samples of the input waveform with the A/D converter, process the samples with the microcomputer, and output the processed results to the D/A converter. The processing done by the microcomputer determines the filter response.

If the samples are simply read in from the A/D converter and output directly to the D/A converter, then the output signal will be almost identical to the input signal. If the samples are read in from the A/D converter and held in memory for some time before being output to the D/A converter, then the output signal will simply be a delayed version of the input signal.

Now, to make it more interesting, suppose that we read in, for example, 100 samples from the A/D converter and use some mathematical algorithm to compute an output value based on these samples. When we take in the next sample from the A/D converter, we throw out the oldest sample and use the latest 100 samples to compute the next output value. The output value at any time then will be a sort of "average" of the last 100 samples. To help visualize this, you might think of the process as sliding a window of 100 samples along the waveform and using some algorithm to compute an "average" of the samples.

If the window is positioned so that all 100 samples come from a section of the square wave where the waveform is at $-V$, as shown in Figure 10-42*a*, then the computed output value will be $-V$. As the window slides to the right so that it includes some samples from the high section, as shown in Figure 10-42*b*, the "average" will increase, so the computed output value will increase. The risetime of the output or the rate at which the output value increases is determined by the weight given to new samples versus the weight given to old samples in computing the "average." Using a low-

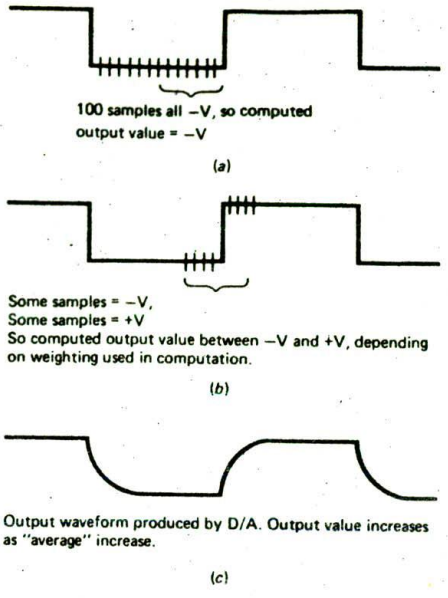


FIGURE 10-42 Effective sample position on computed output value.

pass filter weighting, the output waveform will look like that in Figure 10-42c. The algorithm used to compute the output values determines the characteristics of the output waveform.

The two basic algorithms commonly used for computing the output values are the *finite impulse response* or FIR type and the *infinite impulse response* or IIR type. Figure 10-43*a* shows a functional diagram of the operation of an FIR-type filter. The box containing Z^{-1} represents a delay of one sample interval time. Circles containing an X represent a multiplication operation, and the letters to the left of each circle represent the numbers or coefficients that the term will be multiplied by. Y_0 represents the value of the current sample from the A/D, Y_1 represents the value of the previous sample from the A/D, and Y_2 represents the value of the sample before that. Here's how this works. The output value V at any time is produced by summing the (current sample × some coefficient) + (the previous sample × some coefficient) + (the sample before that × some coefficient), etc. To do all this with a microprocessor requires the simple operations of saving previous samples, multiplying, and adding.

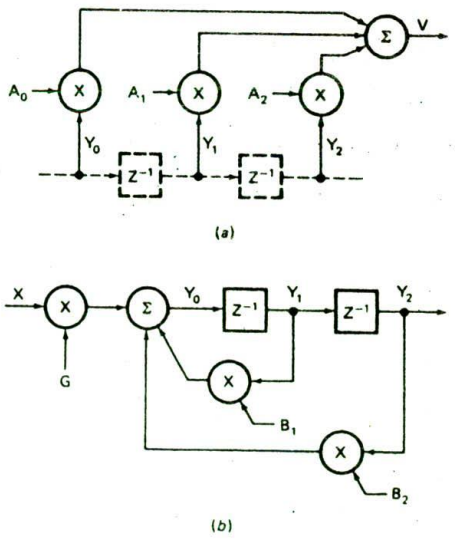


FIGURE 10-43 Digital filter algorithms. (a) FIR. (b) IIR.

Figure 10-43*b* shows a functional diagram for an IIR digital filter. Here again the blocks containing Z^{-1} represent a delay of one sample time. The value of the current sample from the A/D converter is represented by the X at the left of the diagram. The Y_0 point represents the output from the microprocessor to the D/A converter. Note that for an IIR filter, it is this output value which is saved to be used in computing feedback terms for future samples. In the FIR type, remember, the samples from the A/D converter were saved directly for future use. The output for an IIR type is produced by summing (the current sample × a calculated coefficient) + (the previous output value × a calculated coefficient) + (the output value before that × a calculated coefficient), etc.

FIR filters are easier to design, but they may require many terms to produce a given filter response. IIR filters require fewer stages, but they have to be carefully designed so that they do not become oscillators. After we take a look at the special hardware commonly used to implement digital filters, we will describe how computer-based tools are used to calculate the coefficients for FIR and IIR filters.

Digital Filter Hardware

As we said before, the basic parts of a digital filter are an A/D converter, a microcomputer, and a D/A converter. For very low-speed applications the microprocessor used in the microcomputer can be a general-purpose device, such as the 8086 we have used for other applications throughout the book. For many real-time applications such as digital speech processing, however, a general-purpose microprocessor is not nearly fast enough. There are several reasons for this.

1. The architecture of general-purpose machines is mostly memory-based, so most operands must be fetched from memory. The memory access time then adds to the processing time.
2. The Von Neuman architecture of general-purpose microprocessors uses the same bus for instructions and data. This means that data cannot be fetched until the code fetch is completed.
3. The multiply and add operations needed in most digital filter applications each require several clock cycles to execute in a general-purpose machine because the internal hardware is not optimized for

these operations. Since many computations are needed to produce each output value, the time required for these instructions severely limits the sampling rate and the maximum frequency the filter can handle.

To solve these and other problems, several companies have designed microprocessors which have the specific features needed for digital signal processing applications. The leading examples of these types of processors are the TMS320CXX family devices from Texas Instruments. Currently the five generations in this family are the TMS320C1X, the TMS320C2X, the TMS320C3X, the TMS320C4X, and the TMS320C5X devices. These devices have a wide variety of features, but here are some of the common features.

1. Sizable amounts of on-chip registers, ROM, and RAM, so data and instructions can be accessed very quickly.
2. Separate buses for code words and for data words. This approach is commonly referred to as *Harvard architecture*. As you can see in Figure 10-44, the TMS320CXX devices' implementation of Harvard architecture has an address bus and a data bus for program words, an address bus and a data bus for data words, and even an address bus and a data bus for direct memory access (DMA) by an external device. These parallel buses allow instructions and data to be fetched at the same time.
3. An optimized multiplier which, depending on the specific device, can perform a 16 × 16- or a 32 × 32-bit multiply in one clock cycle. For the TMS320C50 device, a clock cycle can be as short as 35 ns.
4. A 32-bit barrel shifter which can shift an operand any number of bits in one clock cycle.
5. A 16-bit or a 32-bit CPU which maintains precision during the chain calculations needed in most digital filter applications.
6. An instruction set optimized for DSP applications. For example, the single TMS320C30 instruction MPYI3 <srcA>, <srcB>, <dst1> || ADDI3 <srcC>, <srcD>, <dst2> will multiply two specified operands (srcA × srcB) and add two different operands (srcC + srcD). Perhaps you can see how an instruction such as this would be useful in implementing the computations for an FIR filter such as we described before. The TMS320C30 also has a "zero-overhead" loop instruction which can be used to quickly repeat an operation some number of times.
7. Some devices in the family also have built-in floating-point processors which can directly perform operations on numbers in floating-point format. (In the next chapter we show you how the 8087 floating-point processor works.)

Figure 10-45 shows a block diagram of a complete digital filter system using one of the TMS320C25 family parts. Note that a simple analog low-pass filter is put in series with the input. Remember the *sampling theorem*, which states that the highest-frequency signal which can be digitized and reconstructed is one which contains two samples per cycle. If higher frequencies are digitized, alias frequencies will be generated when the signal is reconstructed with a D/A converter. This low-pass filter on the input helps prevent aliasing.

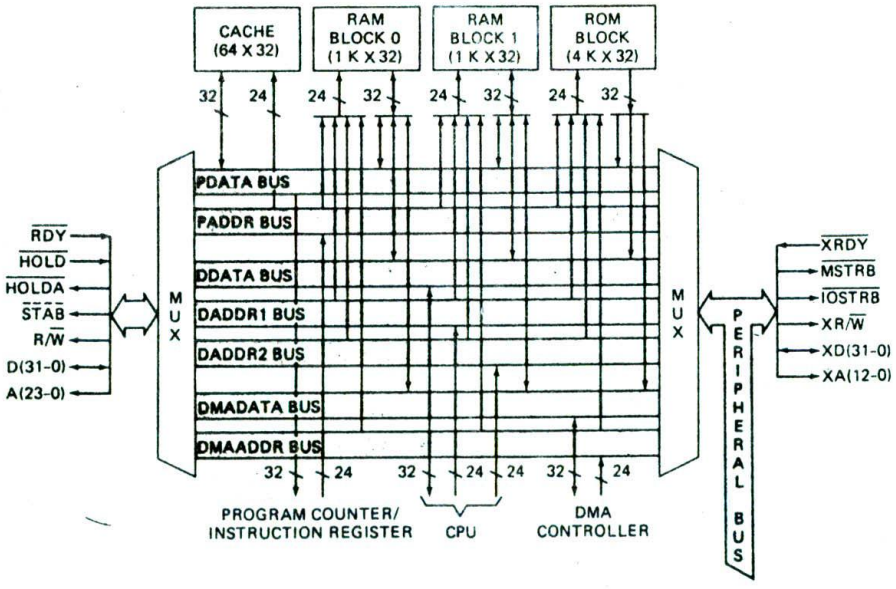


FIGURE 10-44 The TMS320CXX device's implementation of Harvard architecture. (*Texas Instruments Inc.*)

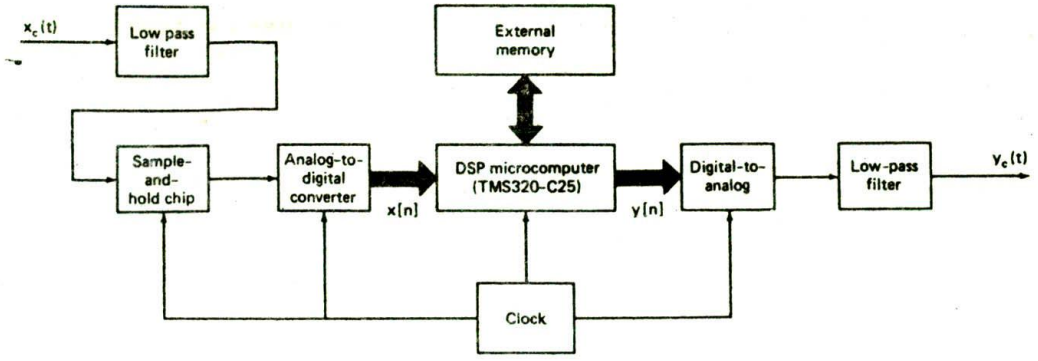


FIGURE 10-45 Block diagram of a TMS320C25 DSP-based filter.

After the anti-alias filter a sample-and-hold is used to keep the value on the input of the A/D constant during conversion. A simple low-pass analog filter is connected to the output of the D/A converter to "smooth" the output signal.

For development purposes and experimentation with a PC-type microcomputer, the DSP-16 board from Ariel Corp. has two 16-bit, 50-kHz A/D converters, a 40-MHz TMS320C25, and two 16-bit 50-kHz D/A converters. The dual channels allow two signals to be processed at the same time. Other boards allow different combinations of sampling rate and number of channels.

Digital Filter Software and Development Tools

As perhaps you can guess from the algorithms in Figure 10-43, developing the program for a digital filter involves two main tasks. The first task is to determine the coefficients by which the terms in the equation will be multiplied to implement the desired filter. The second task is to write a program which reads in values from the A/D converter, computes an output value, and sends the completed value to the D/A converter at the right time.

Several DOS-compatible software packages are available to help perform these tasks. Examples are the Digital Filter Design Package-2 (DFDP2) from Atlanta Signal Processors Inc. and the Filter Design and Analysis System (FDAS2) from Momentum Data Systems. When given the desired filter type, break frequencies, and attenuation rates, these packages tell you if the desired filter can be implemented and generate the required coefficients. After the coefficients are generated, another module in these software packages can be used to produce the assembly language program for the DSP microprocessor. To give you an example of how simple the actual program is, Figure 10-46 shows a procedure which implements a bandpass filter on the TMS320C25 in Figure 10-45. The RPTK 68 instruction in this procedure causes the following instruction to be repeated 68 times. The MACD FDATA+>FD00,*– that is repeated 68 times will multiply a data memory value by a program memory value, add the result to an accumulator, and decrement the pointer to point to the next operands. These two instructions then do most of the work of computing an output value.

Once the coefficients and program for a digital filter have been produced, the next step is to test the result.

```
DELAR   EQU     1               *DELAY AR REGISTER
*
FILTER
        LARP    DELAR           *POINT TO THE DELAY INDEX REGISTER
        LRLK    DELAR,2000      *INDEX POINTS TO Z-0 (INPUT)
        LAC     VSAMPL,15       *GET & SCALE INPUT
        SACH    *               *SAVE SCALED INPUT

        MPYK    0               *P = 0
        ZAC                     *AC = 0
        LRLK    DELAR,ZLAST     *INDEX POINTS TO Z-N
        RPTK    68
        MACD    FDATA+>FD00,*-  *MULTIPLY,ACCUM.and DELAY
        APAC                    *FORM RESULT
        SACH    VSAMPL,0        *SAVE OUTPUT
        RET                     *RETURN
*
        PEND
        END
```

FIGURE 10-46 TMS320C25 digital filter program. (*R.W. Schafer, "The Math Behind the MIPS: DSP Basics." Electronic Design, September 1988*)

One way to do this is with a PC-compatible board such as the Ariel unit described before. Another method is to use an emulator such as the Texas Instruments XDS1000. This emulator consists of a PC board which plugs into an IBM PC-compatible computer and a buffer pod which plugs into the prototype hardware. As with other emulators we have discussed, it allows you to load programs, set breakpoints, do traces, etc. A software package and an emulator allow you to quickly design, test, and debug a digital filter system.

Switched Capacitor Digital Filters

For simple filter designs, another type of digital filter called a *switched capacitor filter* implements digital filtering without the need for the A/D and D/A converters. An example of this type of device is the National MF10. In this type of filter an input signal is sampled on a capacitor. The signal is passed on to other capacitors, and fractions of the outputs from these capacitors are summed to produce an analog output signal directly. Switched capacitor filters are less expensive, but they do not give the degree of programmability that the microprocessor-based filters do.

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.

Op amp

Comparator

Hysteresis

Noninverting amplifier

Inverting amplifier

Virtual ground

Gain-bandwidth product

Unity-gain bandwidth

Adder circuit—summing point

Differential amplifier

Common-mode signal, common-mode rejection

Instrumentation amplifier

Op-amp integrator circuit

Linear ramp
  Saturation

Op-amp differentiator

Op-amp active filters

Low-pass filter, high-pass filter, bandpass filter
  Critical frequency or breakpoint
  Second-order low-pass filter, second-order high-pass filter

Photodiode, solar cell

Temperature-sensitive voltage sources

Temperature-sensitive current sources

Thermocouples, cold-junction compensation

Force and pressure transducers

Strain gage, LVDT, load cell

Flow sensors—paddle wheel, differential pressure transducer

D/A converters
  Resolution
  Full-scale output voltage
  Maximum error
  Linearity
  Settling time

A/D converters
  Conversion time
  Sample and hold
  Sampling theorem
  Quantizing error

A/D conversion methods
  Parallel-comparator A/D converter
  Dual-slope A/D converter
  Successive-approximation A/D converter
  Data acquisition system

Direct memory access, DMA

Set point

Servo control

Settling time, underdamped and overdamped responses

Residual error

Proportional-integral-derivative control loop, PID

Time-slice system

On/off control

Robotics

Embedded controllers

80186, 80188

Digital signal processing

Time-domain and frequency-domain view of a square wave

Digital filter operation

Finite impulse response (FIR) filter algorithm

Infinite impulse response (IIR) filter algorithm

Switched capacitor filter

REVIEW QUESTIONS AND PROBLEMS

1. *a.* A comparator circuit such as the one in Figure 10-1*b* is powered by ±15 V, the inverting input is tied to +5 V, and the noninverting input is at +5.3 V. About what voltage will be on the output of the comparator?
   b. An amplifier circuit, such as the one in Figure 10-1*d*, has R1 = 10 kΩ and R2 = 190 kΩ. Calculate the closed-loop voltage gain for the circuit and the V_{out} that will be produced by a V_{in} of 0.030 V. What voltage would you measure on the inverting input? What would be the gain of the circuit if R2 = 0 Ω?
   c. An amplifier circuit, such as the one in Figure 10-1*e*, is built with an R1 of 15 kΩ and an R_f of 75 kΩ. Calculate the closed-loop voltage gain for the circuit and the output voltage for an input voltage of 0.73 V. What voltage will you always measure on the inverting input of this circuit?
   d. A differential amplifier, such as the one in Figure 10-1*g*, is built with R1 = R2 = 100 kΩ and R_f = R = 1 MΩ. V_1 = 4.9 V, and V_2 = 5.1 V. Calculate the output voltage and polarity.
   e. Describe the main advantage of the instrumentation amplifier in Figure 10-1*h* over the simple differential amplifier in Figure 10-1*g*.
   f. If the amplifier used in the circuit in part *b* has a gain-bandwidth product of 1 MHz, what will be the closed-loop bandwidth of the circuit?
2. Draw a circuit showing how a light-dependent resistor can be connected to a comparator so the output of the comparator changes state when the resistance of the LDR is 10 kΩ.
3. For the photodiode amplifier circuit in Figure 10-5, what voltage will you measure on the inverting input of the amplifier? Why is it important to use an FET input amplifier for this circuit? Which direction are electrons flowing through the photodiode?
4. In what application might you use a temperature-dependent current device such as the AD590 rather than a temperature-dependent voltage device such as the LM35?
5. Why must thermocouples be cold-junction compensated in order to make accurate measurements? How can the nonlinearity of a thermocouple be compensated for?
6. Why are strain gages usually connected in a bridge configuration? Why do you use a differential amplifier to amplify the signal from a strain gage bridge?
7. Calculate the full-scale output voltage for the simple D/A converter in Figure 10-14.
8. What is the resolution of a 13-bit D/A converter? If the converter has a full-scale output of 10.000 V, what is the size of each step? What will be the actual maximum output voltage of this converter? What accuracy should this converter have to be consistent with its resolution?
9. Why must a 12-bit D/A converter have latches on its inputs if it is to be connected to 8-bit ports or an 8-bit data bus?
10. Describe the operation of a flash-type A/D converter. What are its main advantages and disadvantages?
11. For the dual-slope A/D converter in Figure 10-19, what will be the displayed count for an input voltage of 2.372 V? What is the resolution of a 4½-digit slope-type A/D converter expressed in bits?
12. How many clock cycles does a 12-bit successive-approximation A/D converter take to do a conversion on a 0.1-V input signal? On a 5-V input signal? How does this compare with the number of clock cycles required for a 12-bit dual-slope type?
13. *a.* Assume the inputs of the MC1408 D/A converter in Figure 10-20 are connected to an output port on your microcomputer board and the output of the comparator is connected to bit D0 of an input port. Write the algorithm for a procedure to do an A/D conversion by outputting an incrementing count to the output port.
    b. Write an algorithm for a procedure to do the conversion by the successive-approximation method. Which method will produce a faster result? If the hardware is available, write the programs for these algorithms and compare the times by watching the comparator output with an oscilloscope.
14. Show the detailed algorithm for the procedure you would use to read in the data from a multiplexed BCD output A/D converter such as the MC14433 in Figure 10-23 and assemble the value in a 16-bit register for display.
15. The data sheet for an A/D converter indicates that its output is in offset-binary code. If the converter is set up for a range of −5 to +5 V and the output code is 01011011, what input voltage does this represent? How could you convert this code to 2's complement form after you read the code into your microcomputer?
16. Write a procedure to round a 32-bit BCD number in DX:AX to a 16-bit BCD number in DX.
17. For the scale circuitry in Figure 10-23, what voltage should you measure on the inverting input of the LM308 amplifier? What voltages should you measure on the two inputs of the LM363 amplifier with no load on the scale? What voltage should you measure on the output of the LM363 with no load on the scale?
18. The section of the scale program following the label

NXTKEY in Figure 10-35 moves some bytes around in memory. Rewrite this section of the program using an 8086 string instruction to do the move operations. Which version seems more efficient in this case?

19. Describe how feedback helps hold the value of some variable, such as a motor speed, constant. Refer to Figure 10-27 in your explanation.

20. What problem in a control loop does integral feedback help solve? Why is derivative feedback sometimes added to a control loop?

21. What is the major advantage of a microcomputer-controlled loop over the analog approach shown in Figure 10-29?

22. Suppose that you want to control the speed of a small dc motor, such as the one in Figure 10-27, with LOOP1 of our microcomputer-based process controller.
    a. Show how you would connect the output from the motor's tachometer to the system in Figure 10-33. Also show how you would connect an 8-bit D/A to control the current to the motor.
    b. Write a flowchart for the LOOP1 procedure to control the speed of the motor.
    c. Describe how a lookup table could be used to determine the feedback value.

23. a. Describe how a square wave can be generated by the time-domain method.
    b. Describe how a square wave can be generated by the frequency-domain method.

24. a. Describe the basic operation of a digital filter.
    b. Describe the major difference in how an output value is computed in an FIR digital filter and how it is computed in an IIR filter.

25. a. Why is a general-purpose microprocessor such as the 8086 not suitable for most digital filter applications?
    b. Describe three features designed into a digital signal processing microprocessor such as a TMS320CXX device to speed up processing.

26. a. What is the minimum frequency that a sine-wave signal must be sampled with an A/D converter so that it can be reconstructed with a D/A converter?
    b. Why is an analog low-pass filter often put before the A/D in a digital filter?
    c. What is the purpose of the sample-and-hold circuit on the input of the A/D in Figure 10-45?

27. List the two tasks involved in writing the program for a digital filter.