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)

$227_{10} = \underline{\ ?\ }_{\text{Binary}}$

Least Significant Binary Digit ↓

Division	Quotient	Remainder	×	=
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

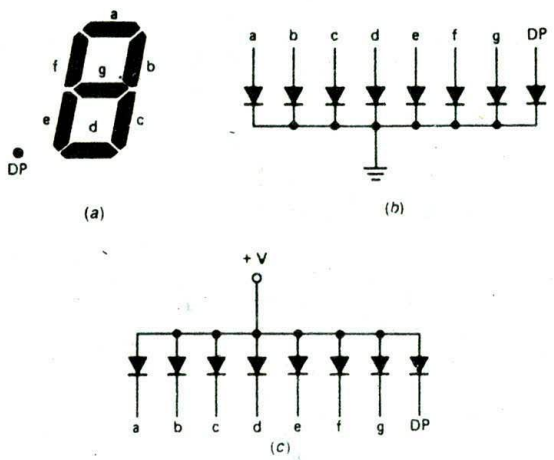


FIGURE 1-4 7-segment LED display. (a) Segment labels. (b) Schematic of common-cathode type. (c) Schematic of common-anode type.

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.

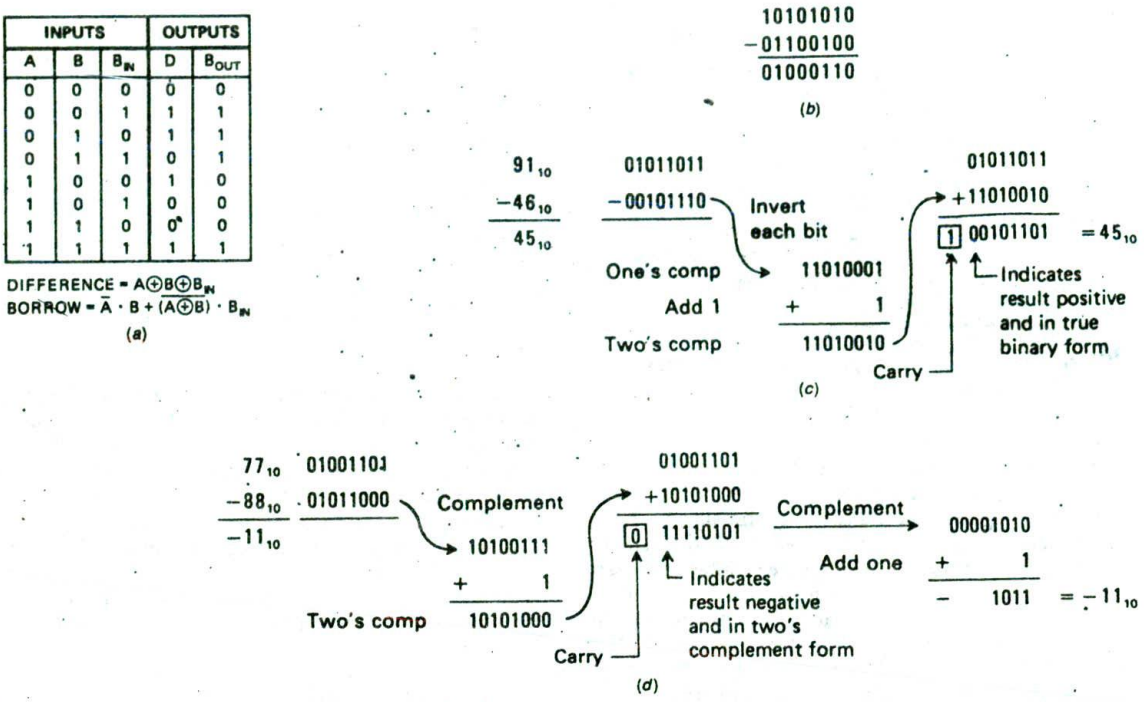


FIGURE 1-8 Binary subtraction. (a) Truth table for 2 bits and borrow. (*b*) Pencil method. (c) 2's-complement positive result. (*d*) 2's-complement negative result.

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               11_10   25_10
              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.

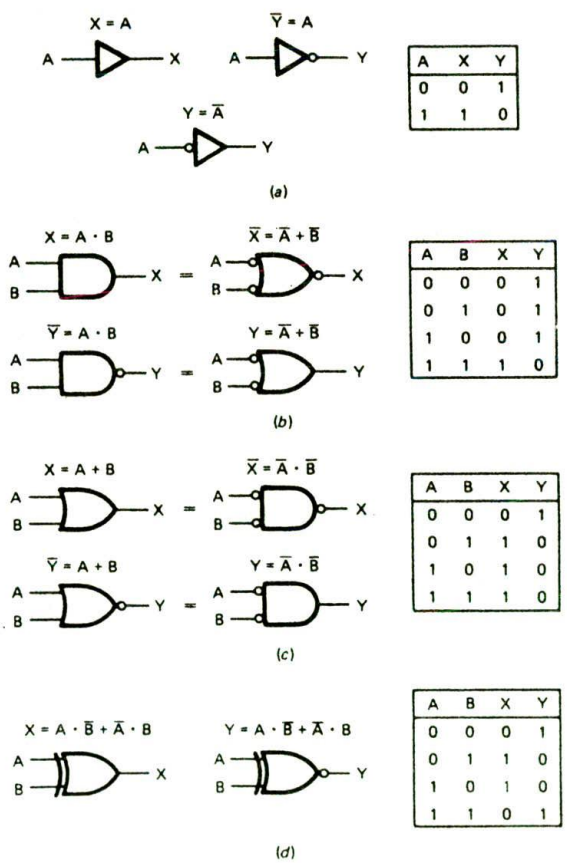
Inverting and Noninverting Buffers

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

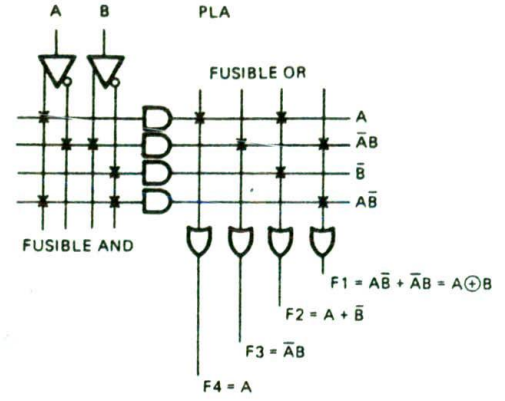
FIGURE 1-15 Buffers and logic gates. (*a*) Buffers. (*b*) AND-NAND. (*c*) OR-NOR. (*d*) Exclusive OR.

Logic Gates

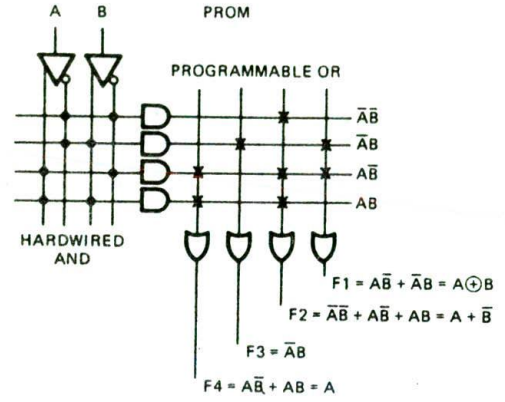
Figure 1-15*b* shows the symbols and truth tables for simple logic gates. A symbol with a flat back and a round front indicates that the device performs the logical *AND* function. This means that the output will be asserted if the A input is asserted *and* the B input is asserted. Again, bubbles or 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.

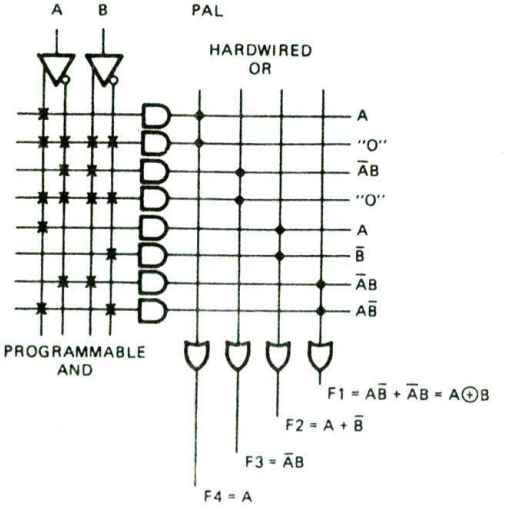
A logic symbol with a curved back indicates that the output of the device will be asserted if the A input is asserted *or* the B input of the device is asserted. Again, bubbles or no bubbles are used to indicate the assertion level for inputs and outputs. Note in Figure 1-15*b* and *c* that each of the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

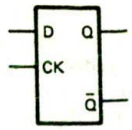
In a *programmable array logic* device or *PAL*, the connections in the OR matrix are fixed and the AND matrix connections are programmable. PALs are often used to implement combinational logic and address decoders in microcomputer systems.

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

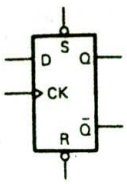
THE D LATCH

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

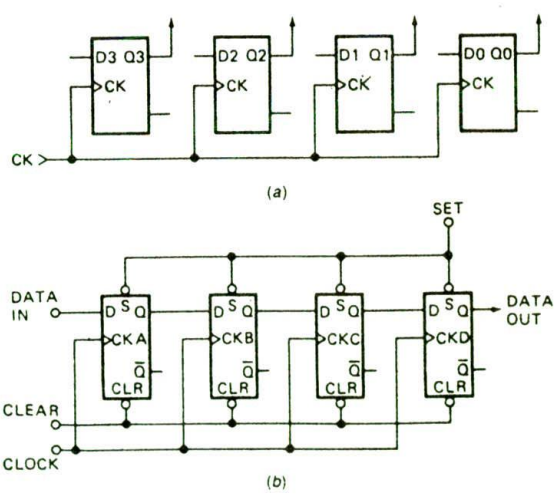
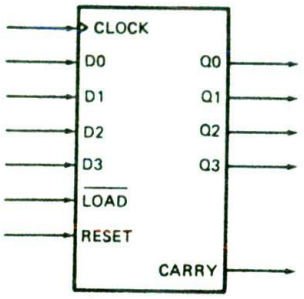


FIGURE 1-18 Registers. (a) Simple data storage. (*b*) Shift register.



(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.

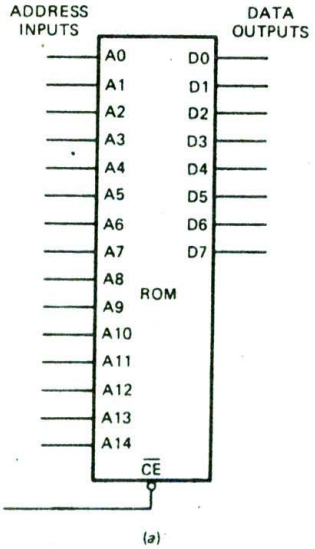
ROMs, RAMs, and Buses

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.

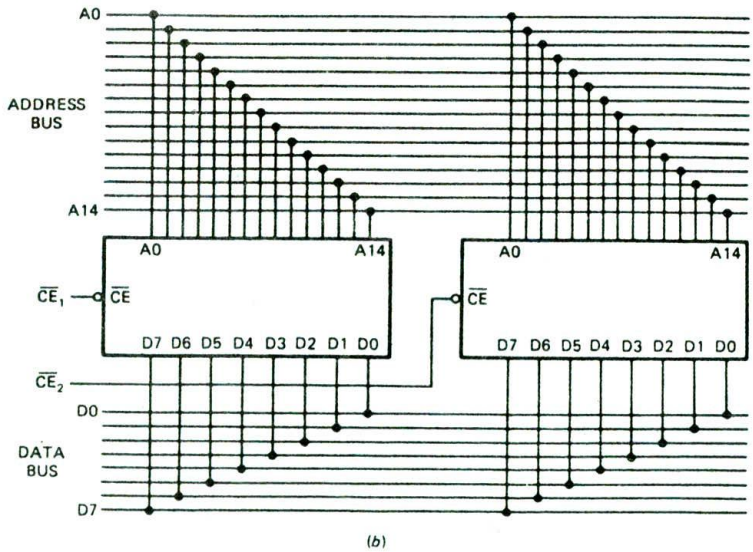
ROMS

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

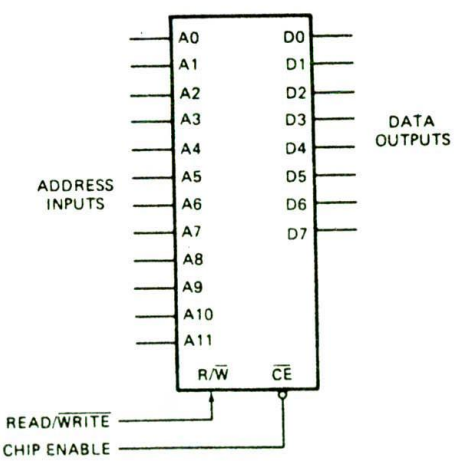


FIGURE 1-21 RAM schematic symbol.

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.

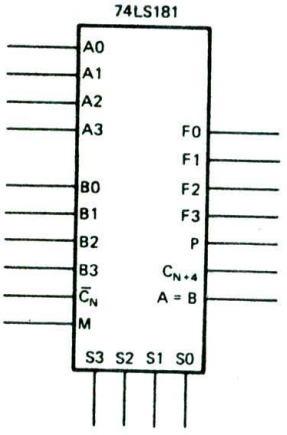
Arithmetic Logic Units

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?

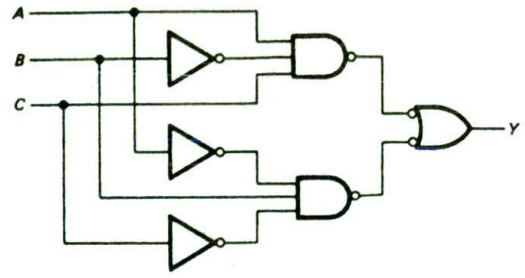


FIGURE 1-23 Circuit for problem 15.

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

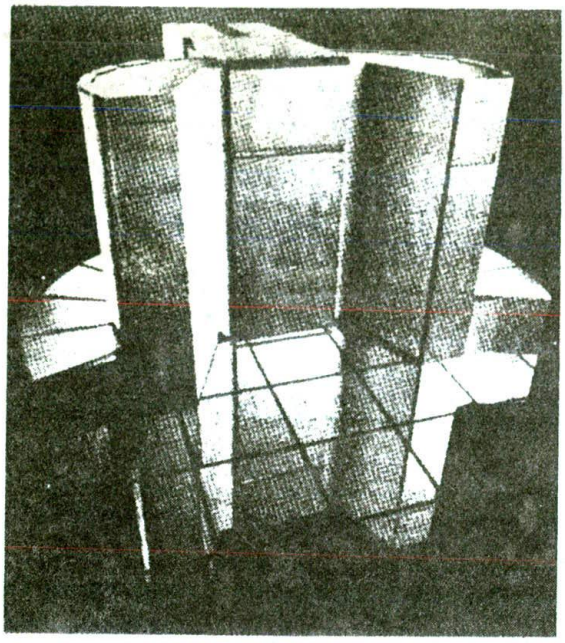
Computers come in a wide variety of sizes and capabilities. The largest and most powerful are often called *mainframes*. Mainframe computers may fill an entire room. They are designed to work at very high speeds with large data words, typically 64 bits or greater, and they have massive amounts of memory. 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

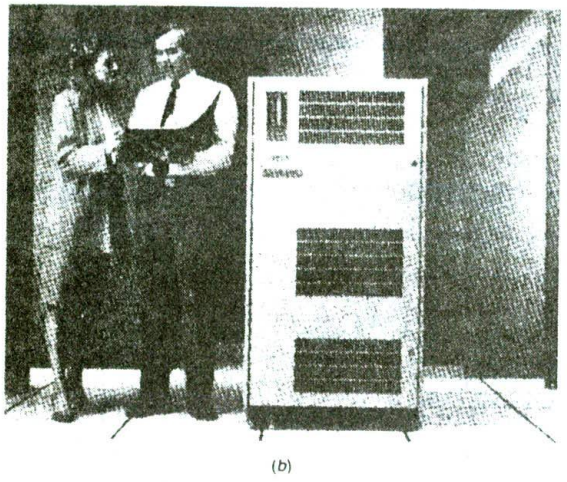
Scaled-down versions of mainframe computers are often called *minicomputers*. The main unit of a minicomputer usually fits in a single rack or box. A minicomputer runs more slowly, works directly with smaller data words (often 32-bit words), and does not have as much memory as a mainframe. Computers 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.

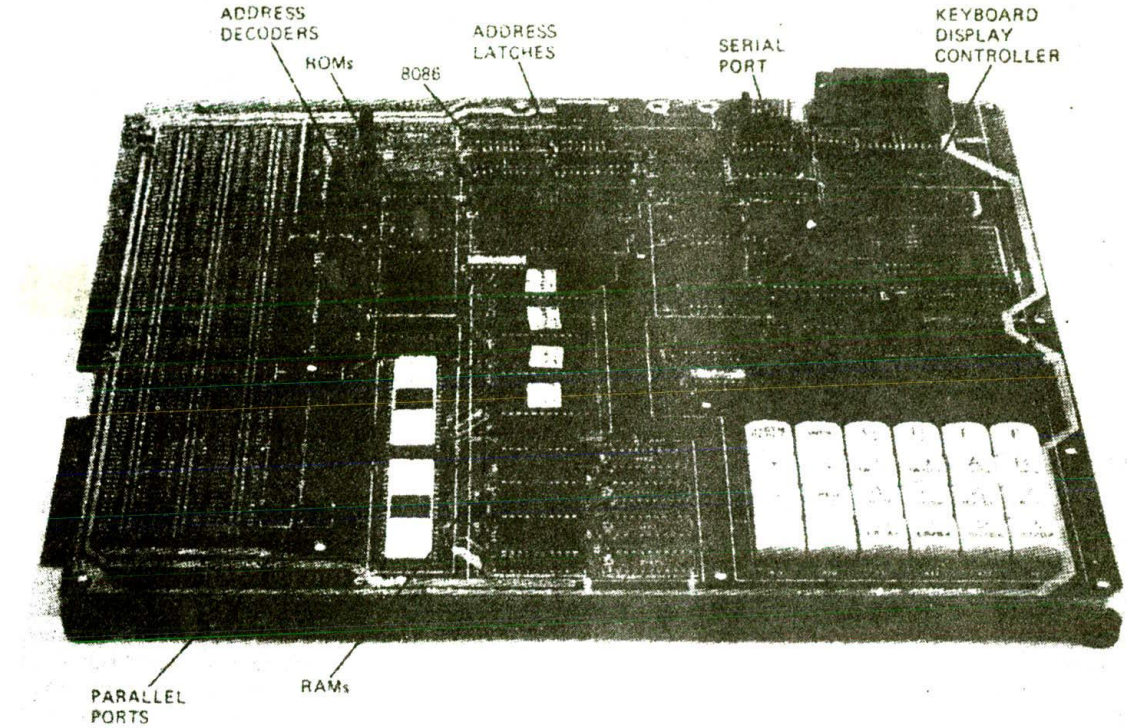
Computerizing an Electronics Factory—Problem

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.

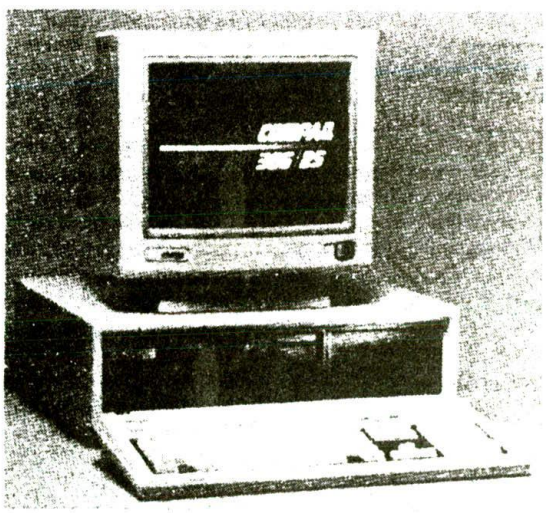
TIMESHARING AND MULTITASKING SYSTEMS

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*

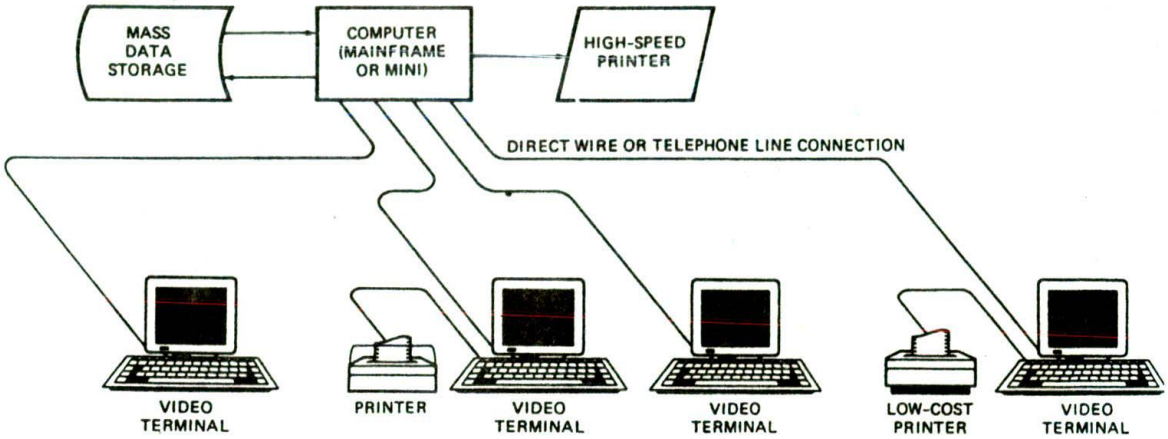


FIGURE 2-3 Block diagram of a computer timesharing system.

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

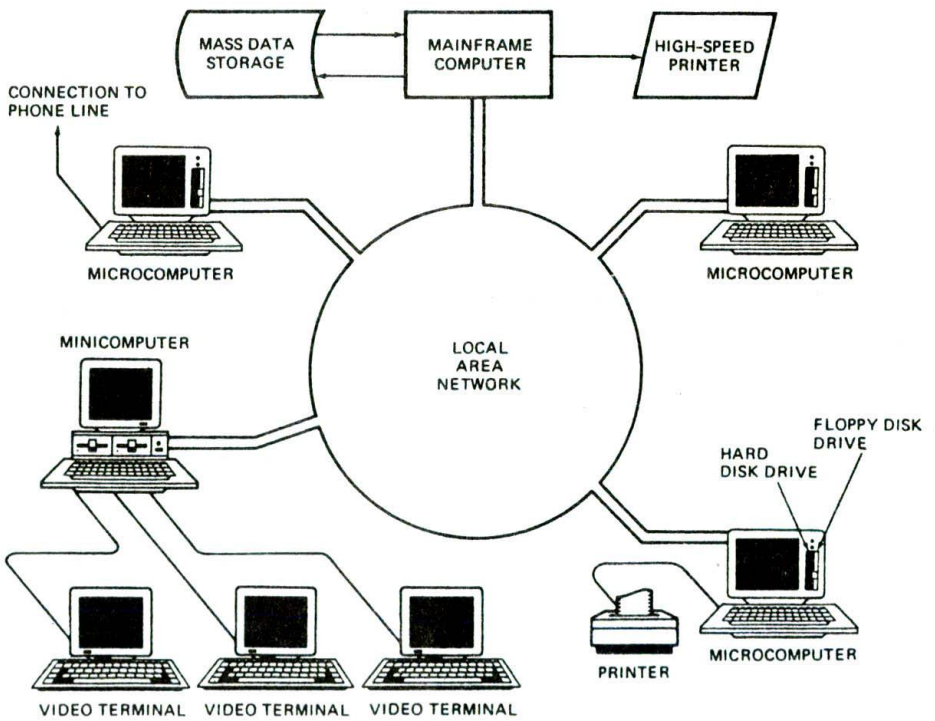


FIGURE 2-4 Block diagram of a distributed processing computer system.

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,

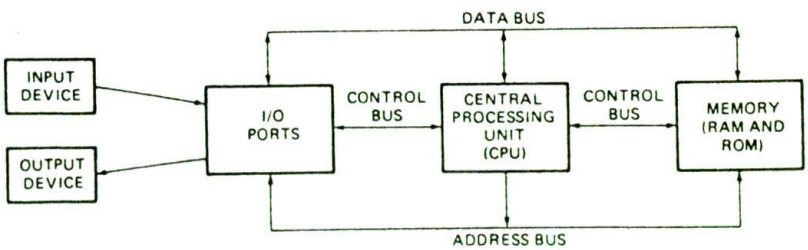


FIGURE 2-5 Block diagram of a simple microcomputer.

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

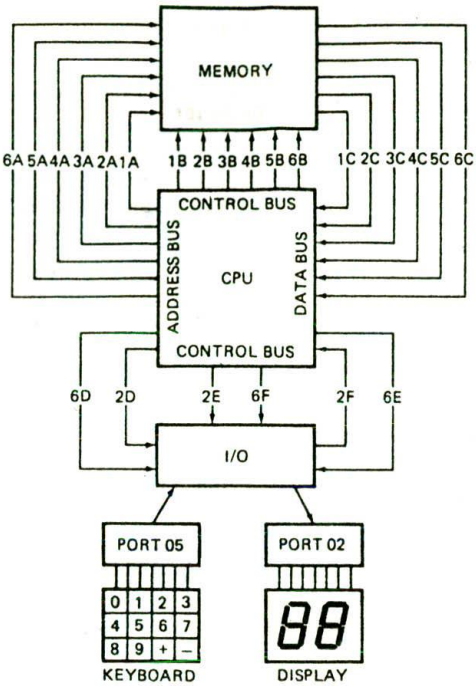
- A computer or microcomputer consists of memory, a CPU, and some input/output circuitry.
- These three parts are connected by the address bus, the data bus, and the control bus.
- The sequence of instructions or program for a computer is stored as binary numbers in successive memory locations.
- The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

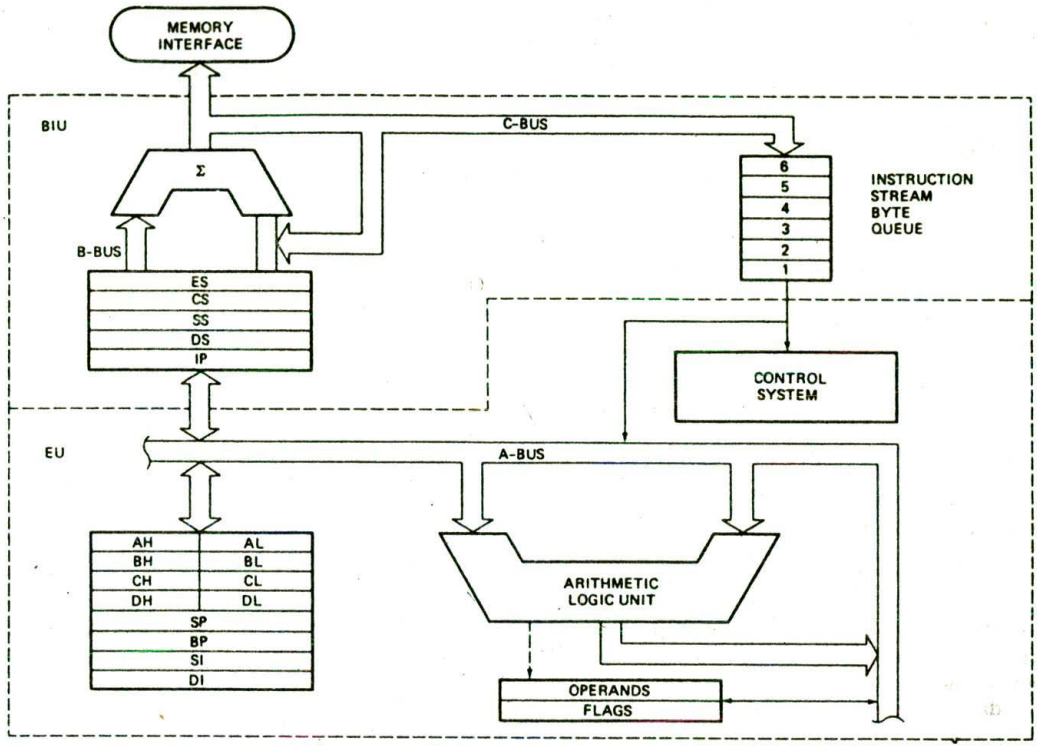


FIGURE 2-7 8086 internal block diagram. (*Intel Corp.*)

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

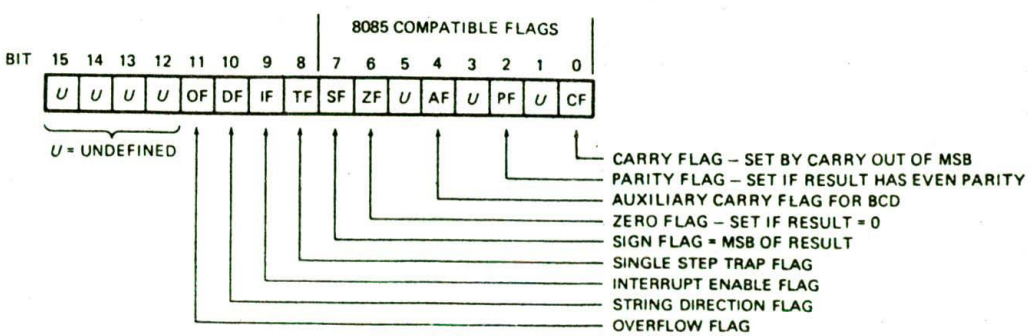


FIGURE 2-8 8086 flag register format. (*Intel Corp.*)

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

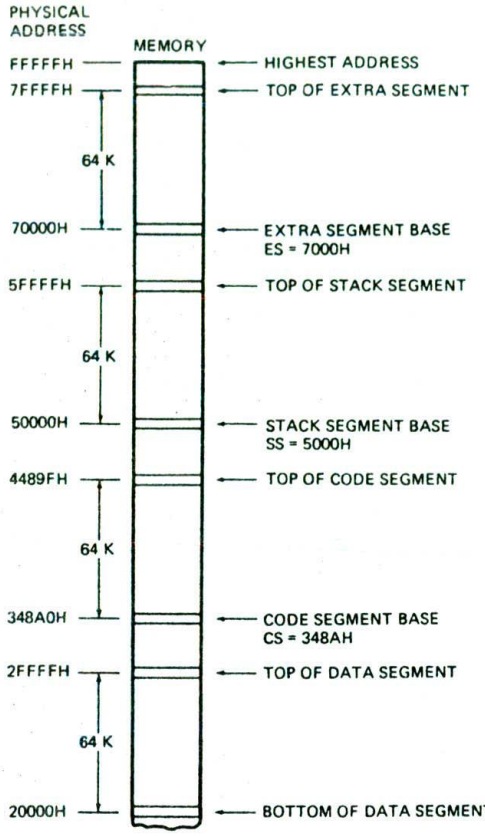


FIGURE 2-9 One way four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086.

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

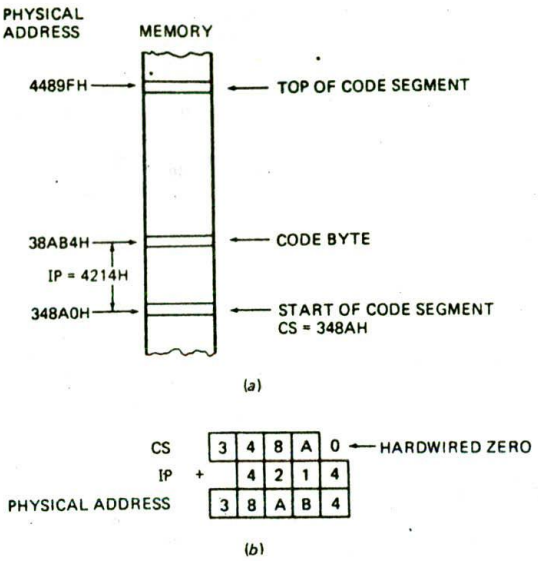


FIGURE 2-10 Addition of IP to CS to produce the physical address of the code byte. (a) Diagram. (b) Computation.

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.

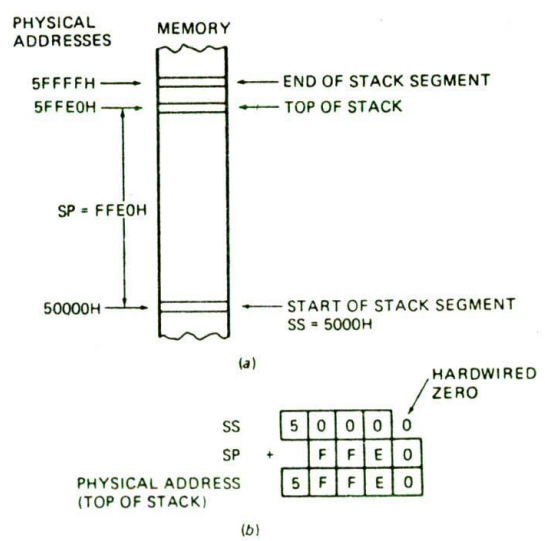


FIGURE 2-11 Addition of SS and SP to produce the physical address of the top of the stack. (a) Diagram. (b) Computation.

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

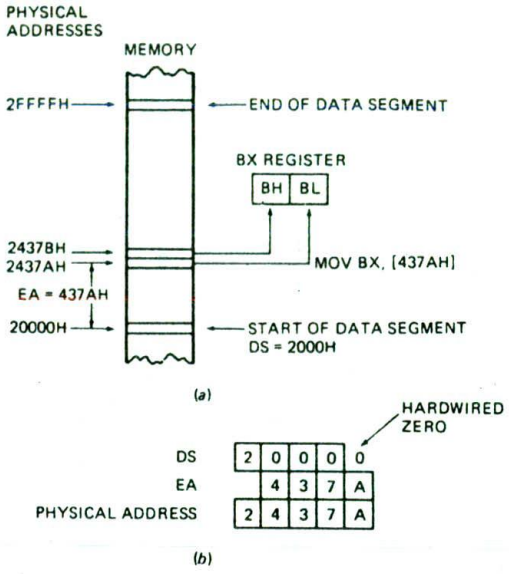


FIGURE 2-13 Addition of data segment register and effective address to produce the physical address of the data byte. (*a*) Diagram. (*b*) Computation.

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*

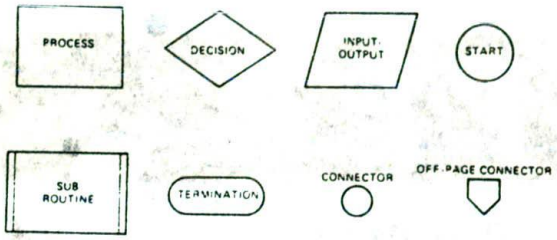


FIGURE 3-1 Flowchart symbols.

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.

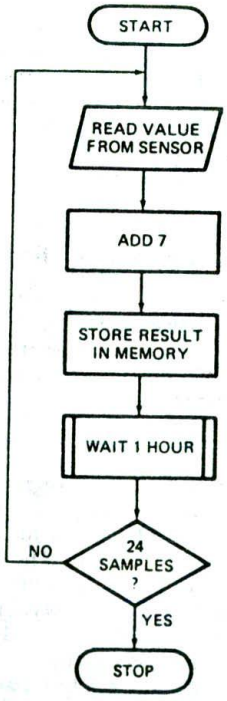


FIGURE 3-2 Flowchart for program to read in 24 data samples from a port, correct each value, and store each in a memory location.

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

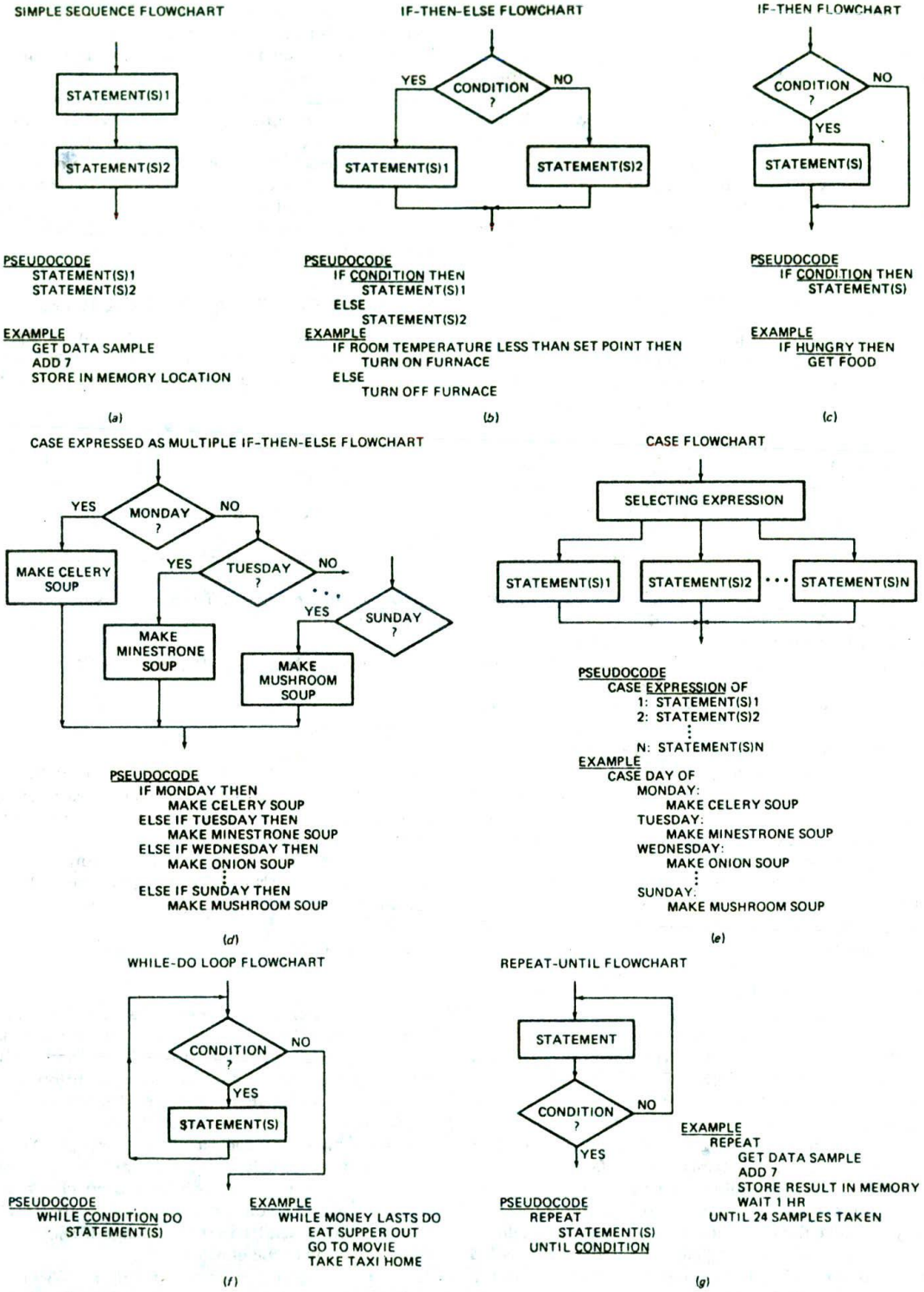


FIGURE 3-3 Standard program structures. (*a*) Sequence. (*b*) IF-THEN-ELSE. (*c*) IF-THEN. (*d*) CASE expressed as nested IF-THEN-ELSE. (*e*) CASE. (*f*) WHILE-DO. (*g*) REPEAT-UNTIL.

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

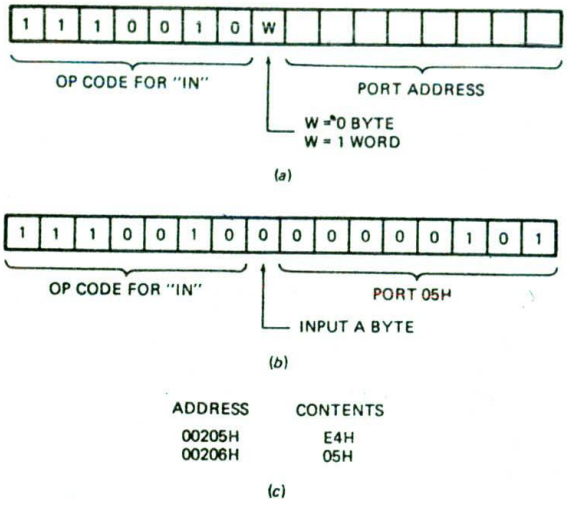


FIGURE 3-5 Coding template for 8086 IN (fixed port) instruction. (*a*) Template. (*b*) Example. (*c*) Hex codes in sequential memory locations.

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.

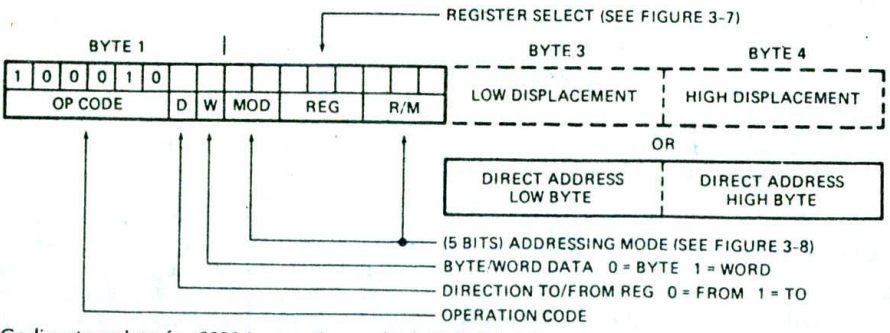


FIGURE 3-6 Coding template for 8086 instructions which MOV data between registers or between a register and a memory location.

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.

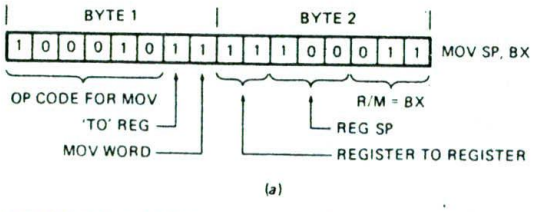
CODING MOV CL, [BX]

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.



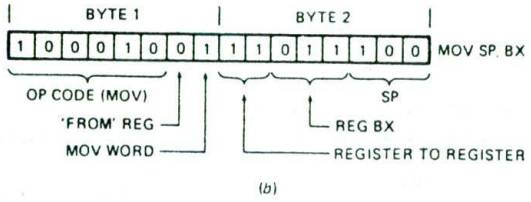


FIGURE 3-9 MOV instruction coding examples. (*a*) MOV SP, BX. (*b*) MOV SP, BX alternative.

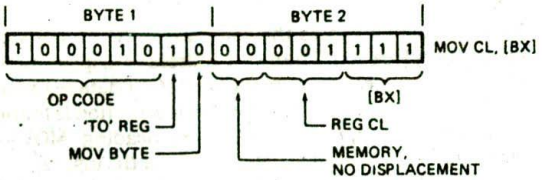


FIGURE 3-10 MOV CL, [BX].

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.

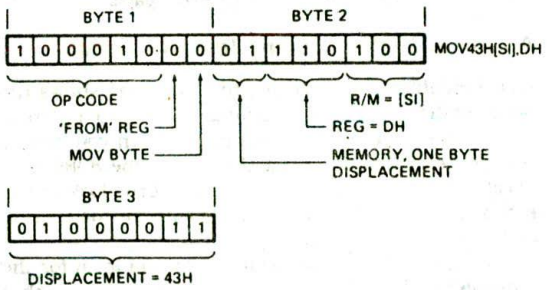


FIGURE 3-11 MOV 43H[SI], DH.

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

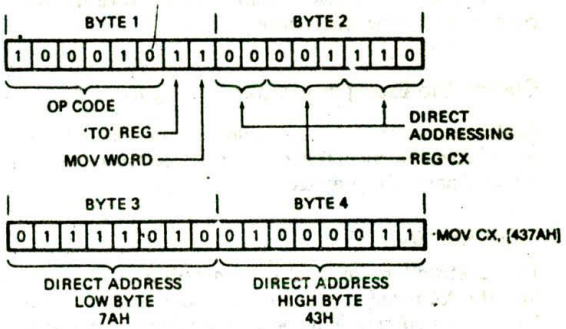


FIGURE 3-12 MOV CX, [437AH].

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

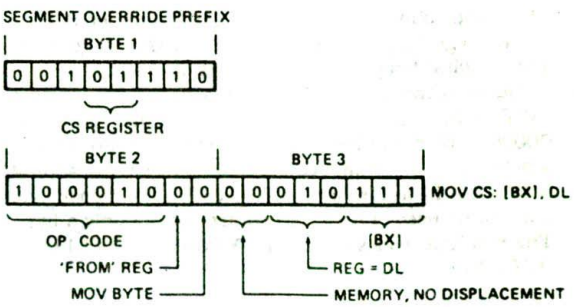


FIGURE 3-13 MOV CS:[BX], DL.

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.

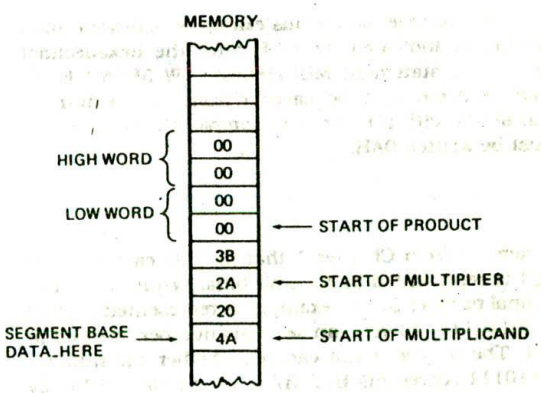


FIGURE 3-15 Data arrangement in memory for multiply program.

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.

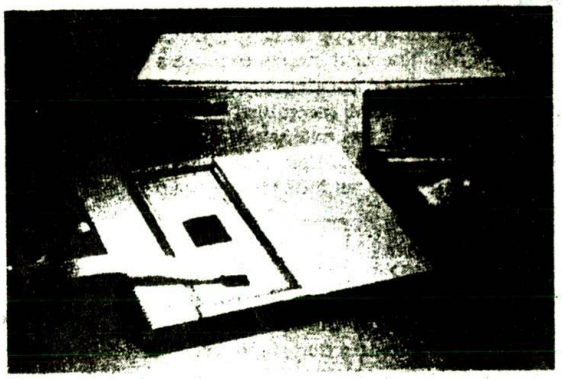


FIGURE 3-17 Applied Microsystems ES 1800 16-bit emulator. *(Applied Microsystems Corp.)*

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

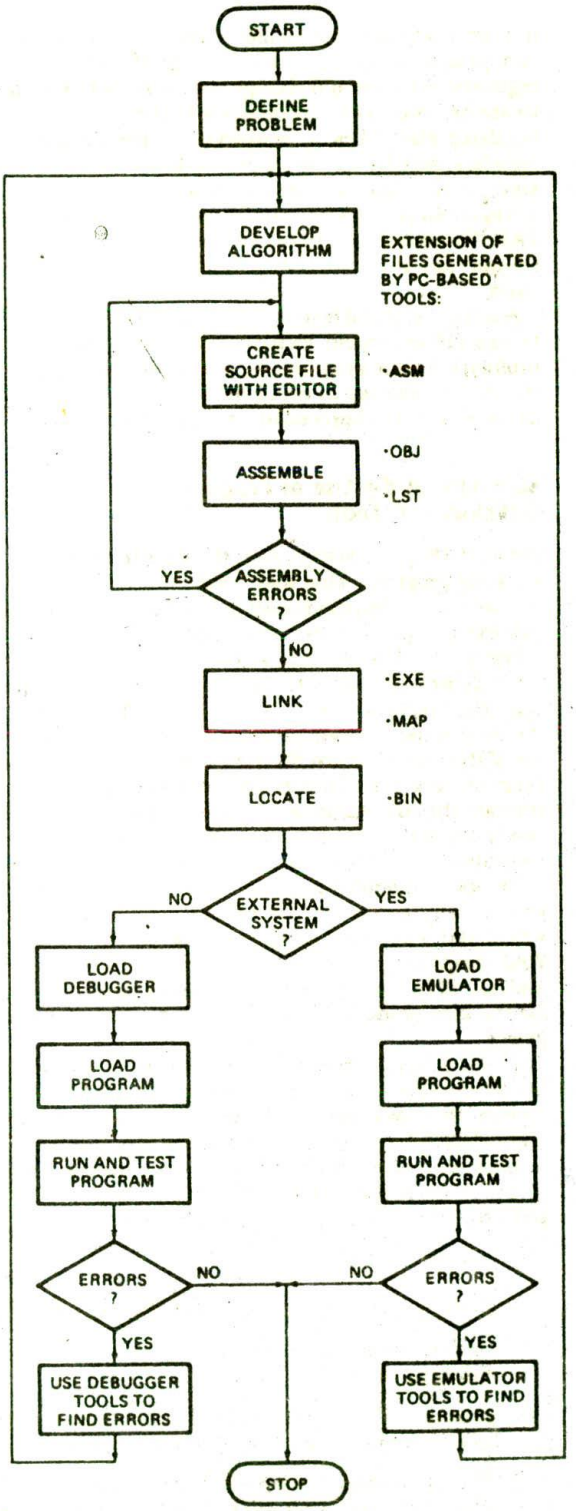


FIGURE 3-18 Program development algorithm (see p. 62).

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

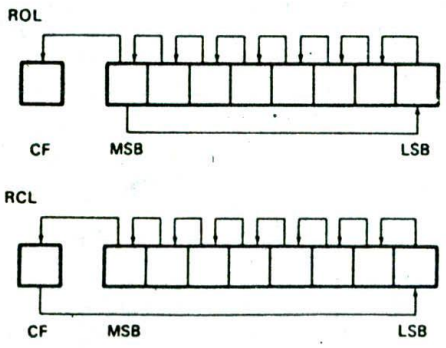


FIGURE 4-4 ROL instruction and RCL instruction operations for byte operands.

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.

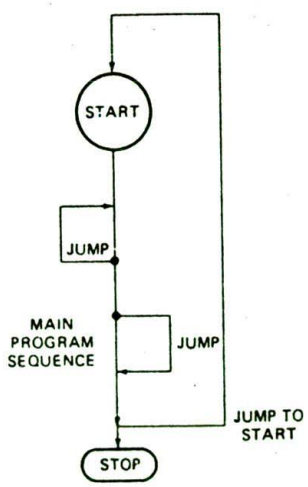


FIGURE 4-6 Change in program flow that can be caused by jump instructions.

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

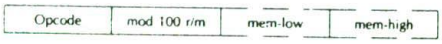
The 8086 Unconditional Jump instruction, JMP, has five different types. Figure 4-7 shows the names and instruction coding templates for these five types. We will first summarize how these five types work to give you an overview; then we will describe in detail the two types you need for your programs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

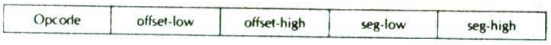
Opcode	Clocks	Operation
E9	15	IP ← IP + Disp16
EB	15	IP ← IP + Disp8 (Disp8 sign-extended)

Within segment or group, Indirect



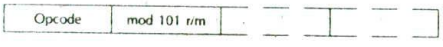
Opcode	Clocks	Operation
FF	11	IP ← Reg16
FF	18+EA	IP ← Mem16

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.

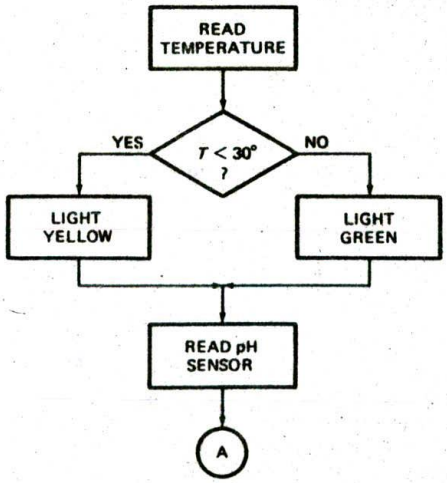
THE 8086 IN AND OUT INSTRUCTIONS

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.

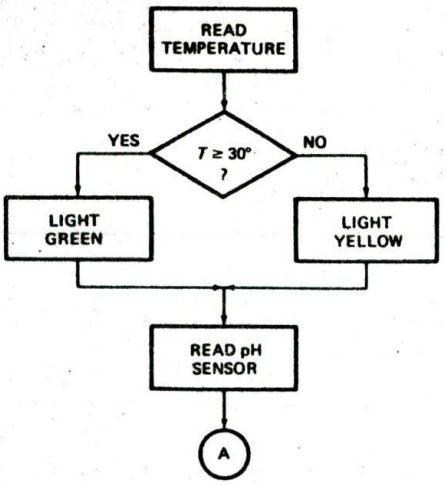
The 8086 also has a fixed-port output instruction and a variable-port output instruction. The fixed-port output instruction has the form OUT port,AL or OUT port,AX. Here again the term *port* represents an 8-bit port address written in the instruction. OUT 0AH,AL, for example, will copy the contents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

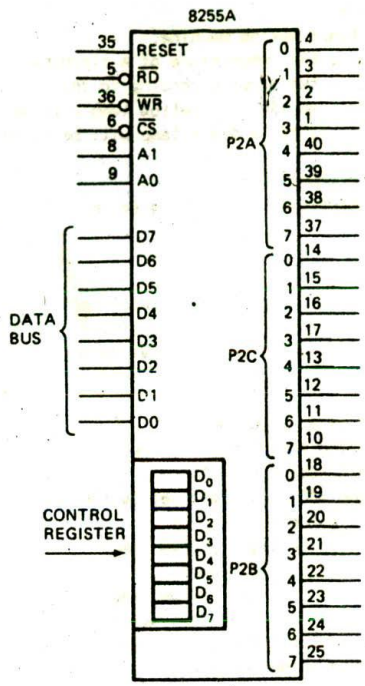


FIGURE 4-13 Block diagram of SDK-86 board's 8255A port.

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.

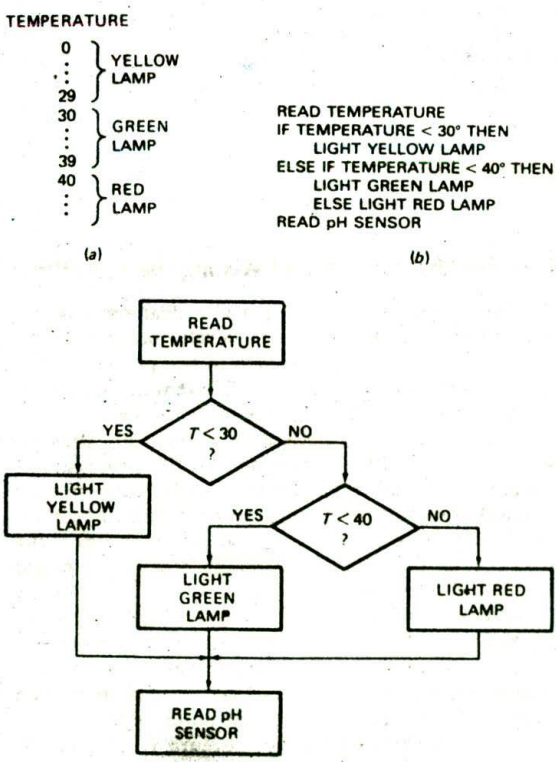


FIGURE 4-15 Algorithm for three-lamp printed-circuit-board-making machine. (*a*) Condition list. (*b*) Pseudocode. (*c*) Flowchart.

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-

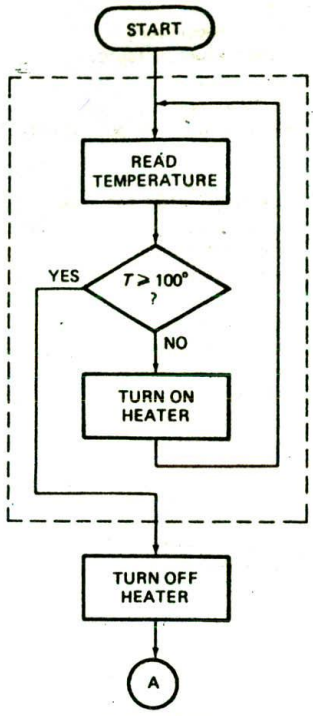


FIGURE 4-17 Flowchart and pseudocode for heater control program.

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

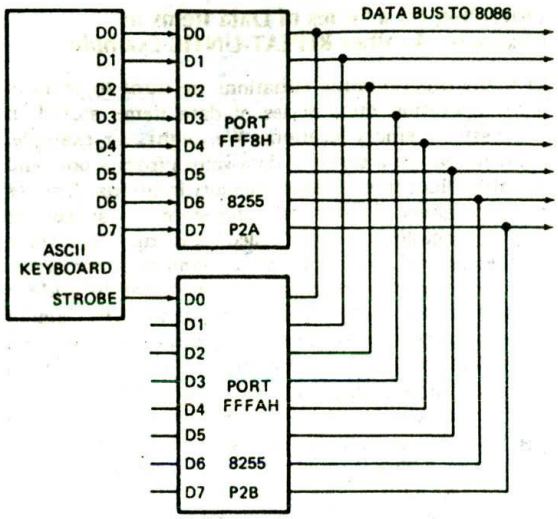


FIGURE 4-19 ASCII-encoded keyboard with strobe connected to microcomputer port.

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:

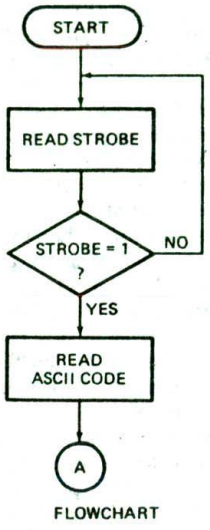
```
LOOK_AGAIN:IN AL, DX
           ROR AL, 1          ; Rotate LSB into carry
           JNC LOOK_AGAIN; If LSB = 0, keep looking
```

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.

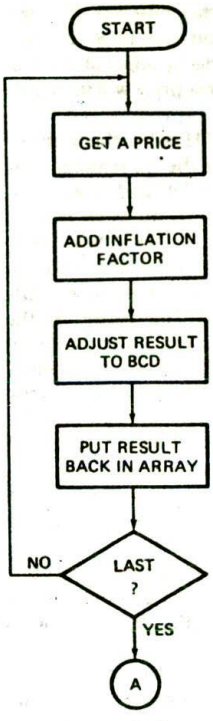
We read one of the BCD prices from memory, add the inflation factor to it, and adjust the result to keep it in BCD format. The new value is then copied back to the array, replacing the old value. After that, a check is made to see whether all the prices have been operated on. If they haven't, then we 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.

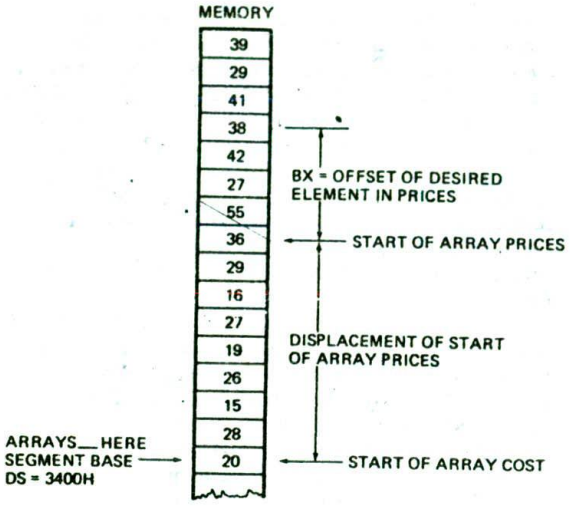


FIGURE 4-22 Data arrangement in memory for "inflate prices" program.

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.

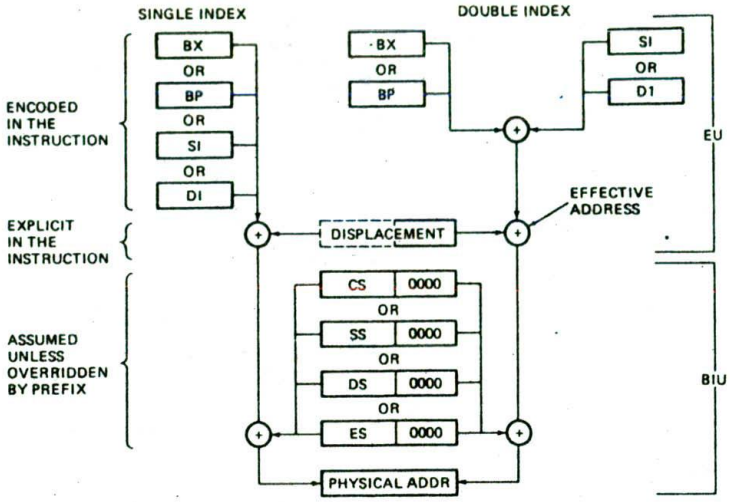


FIGURE 4-24 Summary of 8086 addressing modes.

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.

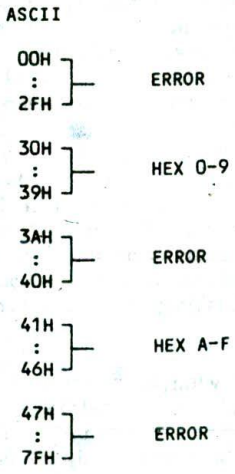


FIGURE 4-29 ASCII chart for Problem 8.

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.

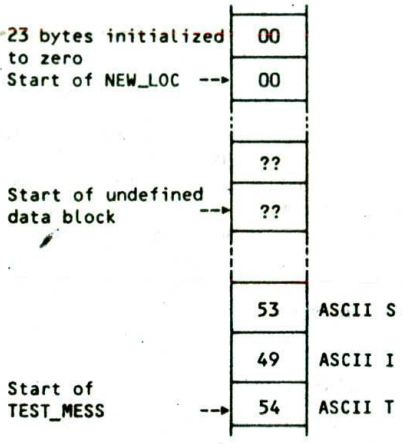
The statement TEST_MESS DB 'TIS TIME FOR A NEW HOME' sets aside 23 bytes of memory and gives the first memory location the name TEST_MESS. This statement will also cause the ASCII codes for the letters enclosed in the single quotes to be written in the reserved memory locations when the program is loaded 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.

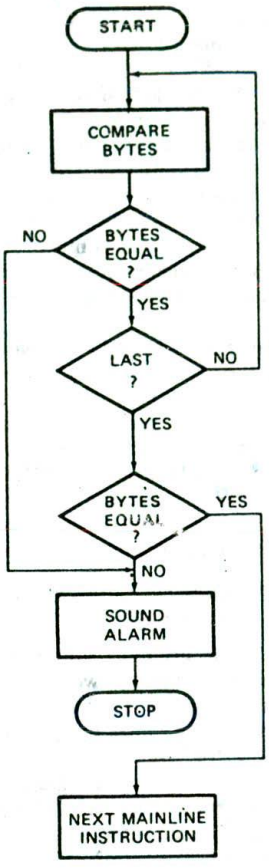
USING THE COMPARE STRING BYTE TO CHECK A PASSWORD

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*

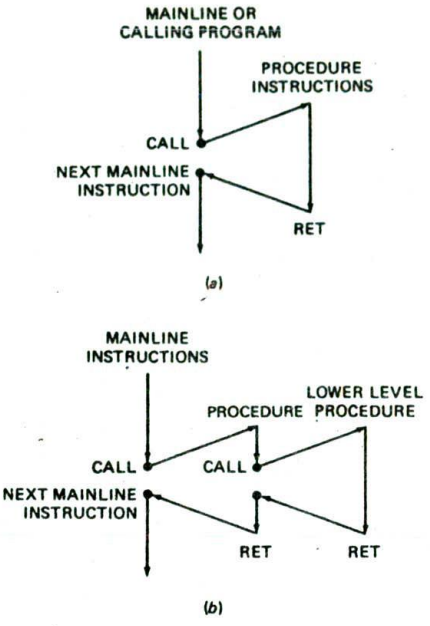


FIGURE 5-4 Program flow to and from procedures. (*a*) Single procedures. (*b*) Nested procedures.

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.

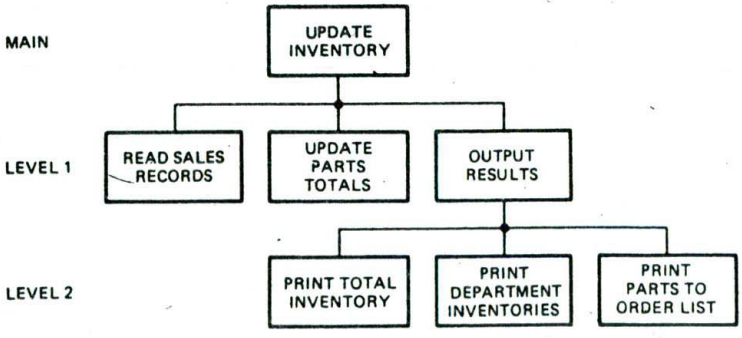


FIGURE 5-5 Hierarchical chart for inventory update program.

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

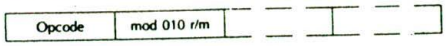
The indirect within-segment CALL instruction is also a near call. When this form of CALL executes, the instruction pointer is replaced with a 16-bit value from a specified register or memory location. As indicated by the MOD-R/M byte in the coding template, the source of the value can be any of the eight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

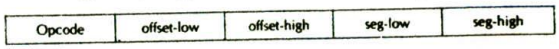
Opcode	Clocks	Operation
E8	19	IP ← IP+Disp16—(SP) ← return link

Within segment or group, Indirect



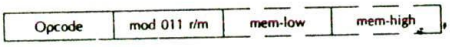
Opcode	Clocks	Operation
FF	16	IP ← Reg16—(SP) ← return link
FF	21+EA	IP ← Mem16—(SP) ← return link

Inter-segment or group, Direct



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

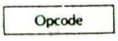
Inter-segment or group, Indirect



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

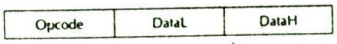
(*a*)

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

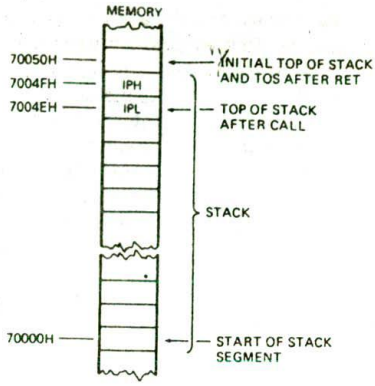


FIGURE 5-7 Stack diagram showing how the return address is pushed onto the stack by CALL.

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.

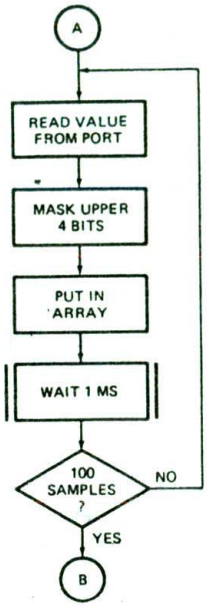
You want to initialize SP so that the first word written to the stack goes to the highest location in the memory you set aside for the stack. All the instructions which write a word to the stack decrement the stack pointer by 2 before writing the word. Therefore, you want the stack pointer to be initially 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

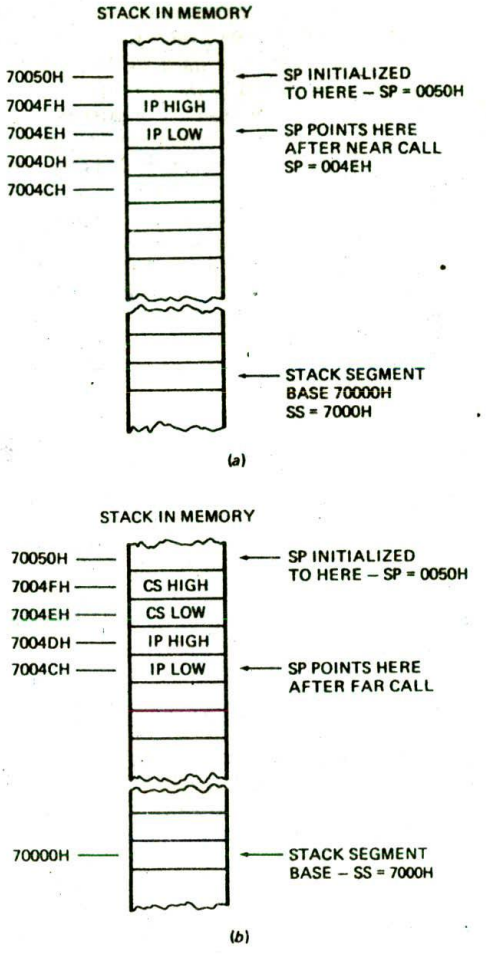


FIGURE 5-11 Stack diagram for program in Figure 5-10. (*a*) For near CALL. (*b*) For far CALL.

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

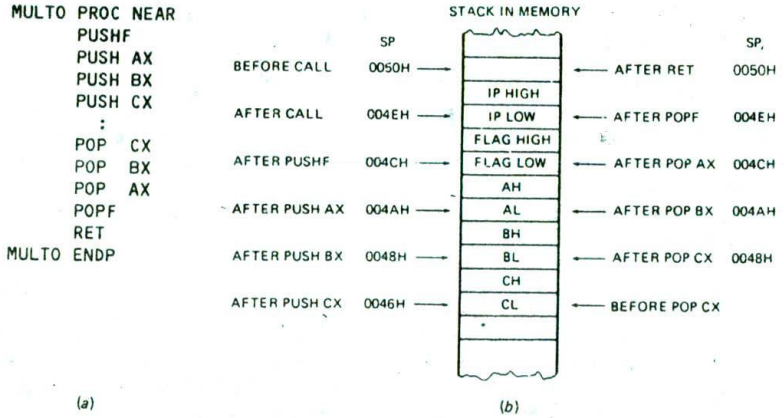


FIGURE 5-12 . Using PUSH and POP instructions. (*a*) Instruction sequence. (*b*) Effect on stack and stack pointer.

```
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

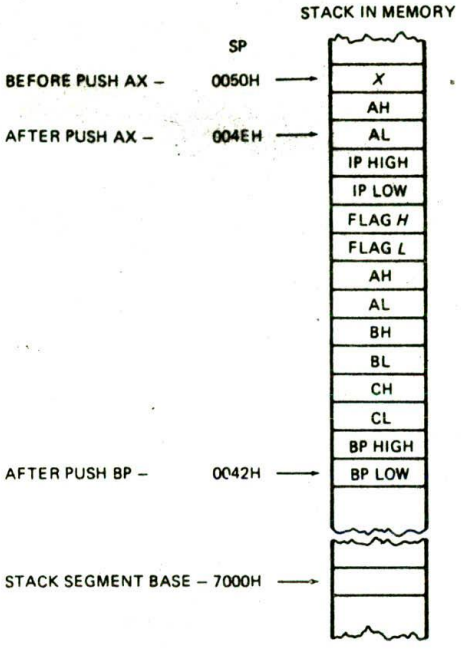


FIGURE 5-18 Stack diagram for program in Figure 5-17.

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.

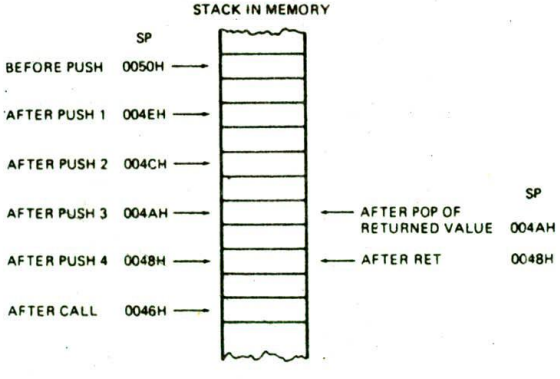


FIGURE 5-19 Stack diagram showing cause of stack overflow.

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.

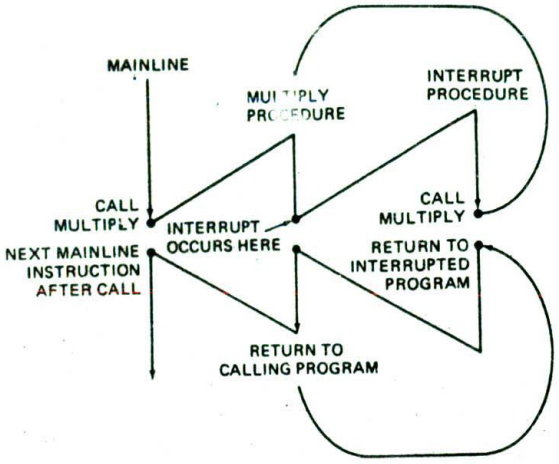


FIGURE 5-20 Program execution flow for reentrant procedure.

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.

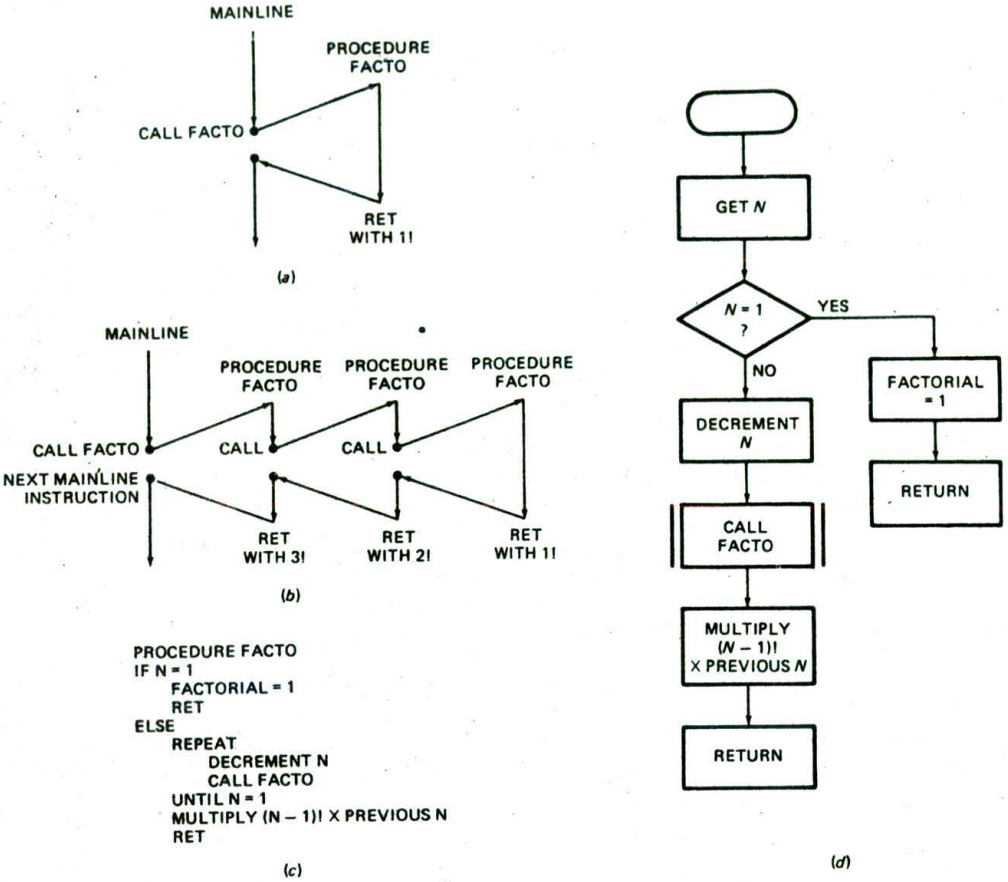


FIGURE 5-21 Algorithm for program to compute factorial for a number N between 1 and 8. (*a*) Flow diagram for N = 1. (*b*) Flow diagram for N = 3. (*c*) Pseudocode. (*d*) Flowchart.

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.

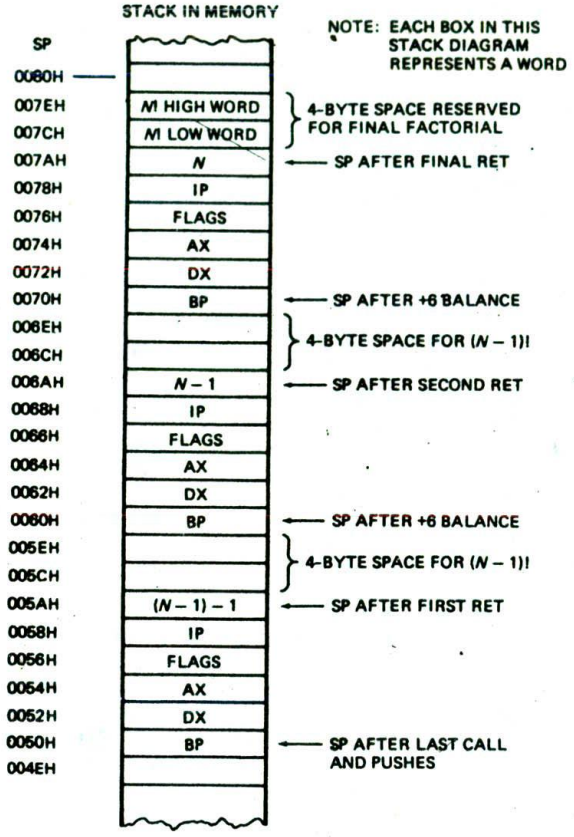


FIGURE 5-23 Stack diagram for program in Figure 5-22 showing contents of stack for N = 3.

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.

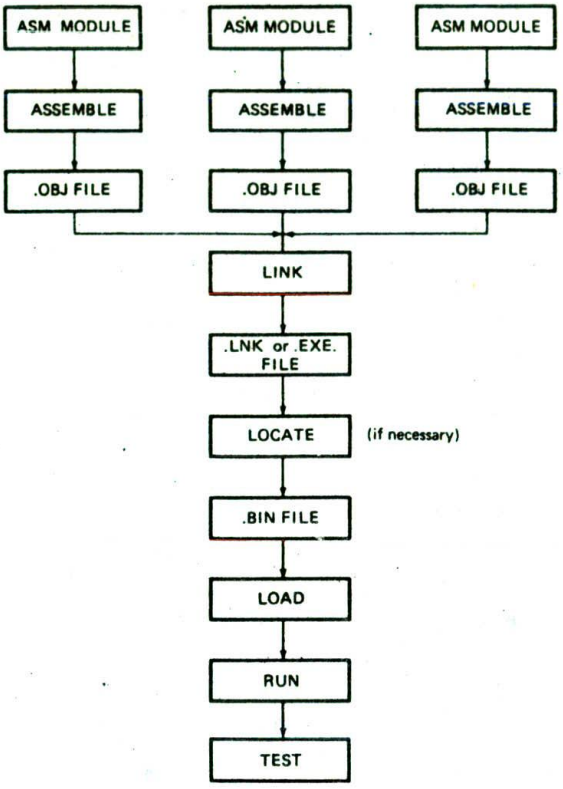


FIGURE 5-25 Chart showing the steps needed to run a program that has been written in modular form.

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

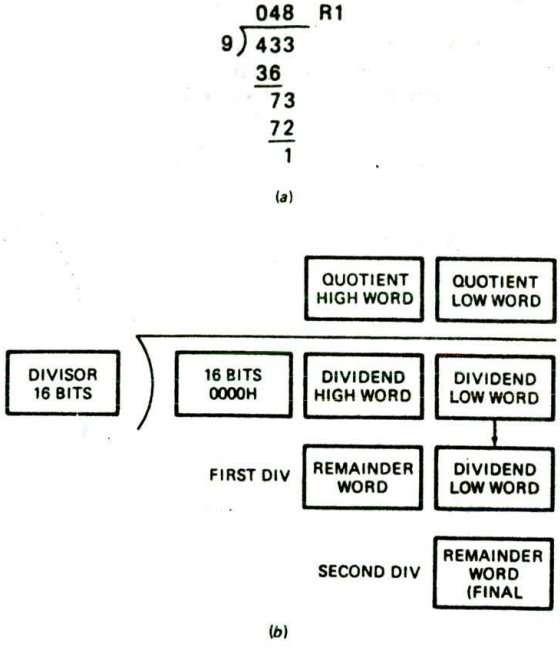


FIGURE 5-26 Algorithm for smart divide procedure. (*a*) Decimal analogy. (*b*) 8086 approach.

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.

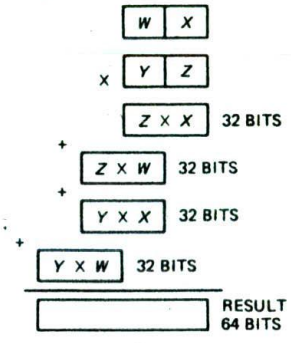


FIGURE 5-28 32-bit by 32-bit multiply method for Problem 9.

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*.