Iran University of Science and Technology
Computer Faculty

INTRODUCTION TO

80x86 Assembly Language

And

Computer Architecture

Introduction

- Chapter 1 – **Representing Data in Computer**
- Chapter 2 – **Parts of a Computer System**
- Chapter 3 – **Elements of Assembly Language**
- Chapter 4 – **Basic Instructions**
- Chapter 5 – **Branching and Looping**
- Chapter 6 – **Procedures**

Introduction

- Chapter 7 – **String Operations**
- Chapter 8 – **Bit Manipulation**
- Chapter 9 – **Assembly Process**
- Chapter 10 – **Floating-Point Arithmetic**
- Chapter 11 – **Decimal Arithmetic**
- Chapter 12 – **Input/Output**

Chapter 1 – Representing Data in a Computer

1-1 Binary and Hexadecimal Numbers

- When dealing with a computer at the machine level, you must be more concerned with how data are stored.
- Often you have the job of converting data from one representation to another.

- **1-1 Binary and Hexadecimal Numbers**

	1		1		0		1	
One	8	One	4	No	2	One	1	= 13

1-1 Binary and Hexadecimal Numbers

Using Hexadecimal Number

The positions in hexadecimal numbers correspond to powers of 16. From right to left, they are 1's, 16's, 256's, etc. The value of the hex number 9D7A is 40314 in decimal since

9	×	4096	36864	[4096=16^3]
+ 13	×	256	3328	[D is 13, 256=16^2]
+ 7	×	16	112	
+ 10	×	1	10	[A is 10]
	×		= 40314	

Chapter 1 – Representing Data in a Computer

1-1 Binary and Hexadecimal Numbers

Decimal	Hexadecimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Chapter 1 – Representing Data in a Computer

1-1 Binary and Hexadecimal Numbers

- A calculator isn't needed to convert a hexadecimal number to its equivalent binary form. In fact, many binary numbers are too long to be displayed on a typical calculator. Instead, simply substitute four bits for each hex digit. The bits are those found in the third column of page, padded with leading zeros as needed. For example

$$3B8E2_{16} = 11\ 1011\ 1000\ 1110\ 0010$$

Chapter 1 – Representing Data in a Computer

1-1 Binary and Hexadecimal Numbers

- To convert binary numbers to hexadecimal format, reverse the above steps: Break the binary number into groups of four bits, starting from the right, and substitute the corresponding hex digit for each group of four bits. For example,
- **1011011101001101111 = 101 1011 1010 0110 1111 = 5BA6F**
- You have seen how to convert a binary number to an equivalent decimal number. However, instead of converting a long binary number directly to decimal, it is faster to convert it to hex, and then convert the hex number to decimal. Again, using the above 19-bit-long number,
- **10110111010011011112**
- **= 101 1011 1010 0110 1111**
- **=5BA6F16**
- **= 5 × 65536 + 11 × 4096 + 10 × 256 + 6 × 16 + 15 × 1**
- **= 375407_{10}**

1-1 Binary and Hexadecimal Numbers

- The following is an algorithm for converting a decimal number to its hex equivalent. It produces the hex digits of the answer right to left. The algorithm is expressed in pseudocodek.

```
until DecimalNumber = 0 loop
    divide DecimalNumber by 16, getting Quotient and Remainder;
    Remainder (in hex) is the next digit (right to left);
    DecimalNumber := Quotient;
end until;
```

Chapter 1 – Representing Data in a Computer

1-1 Binary and Hexadecimal Numbers

Example:

- Divide 16 into 5876 (DecimalNumber).

367 Quotient the new value for DecimalNumber

$$16\,\overline{)5876}$$

5872

4 Remainder the rightmost digit of the answer

Result so far: 4

- 367 is not zero. Divide it by 16.

22 Quotient the new value for DecimalNumber

$$16\,\overline{)367}$$

352

15 Remainder the second digit of the answer

Result so far: F4

- 22 is not zero. Divide it by 16.

1-1 Binary and Hexadecimal Numbers

Example:

- 22 is not zero. Divide it by 16.

$$\begin{array}{r|l} \overset{1}{16\overline{)22}} & \text{Quotient} \quad \text{the new value for DecimalNumber} \\ \underline{16} & \\ 6 & \text{Remainder} \quad \text{the next digit of the answer} \end{array}$$

Result so far: 6F4

- 1 is not zero. Divide it by 16.

$$\begin{array}{r|l} \overset{0}{16\overline{)1}} & \text{Quotient} \quad \text{the new value for DecimalNumber} \\ \underline{0} & \\ 1 & \text{Remainder} \quad \text{the next digit of the answer} \end{array}$$

Result so far: 16F4

- 0 is zero, so the until loop terminates. The answer is $16F4_{16}$

1-2 Character Codes

Letters, numerals, punctuation marks, and other characters are represented in a computer by assigning a numeric value to each character. Several schemes for assigning these numeric values have been used. The system commonly used with microcomputers is the American Standard Code for Information Interchange (ASCII).

- The ASCII system uses seven bits to represent characters, so that values from 000 0000 to 111 1111 are assigned to characters. This means that 128 different characters can be represented using ASCII codes. The ASCII codes are usually given as hex numbers from 00 to 7F or as decimal numbers from 0 to 127. You can check that the message

Computers are fun.

Chapter 1 – Representing Data in a Computer

1-2 Character Codes

- can be coded in ASCII, using hex numbers, as

43	6F	6D	70	75	74	65	72	73	20	61	72	65	20	66	75	6E	2E
C	o	m	p	u	t	e	r	s		a	r	e		f	u	n	.

- Note that a space, even though it is invisible, has a character code (hex 20).
- Numbers can be represented using character codes. For example, the ASCII codes for the date October 21, 1976 are

4F	63	74	6F	62	65	72	20	32	31	2C	20	31	39	37	36
O	c	t	o	b	e	r		2	1	,		1	9	7	6

1-2 Character Codes

- The ASCII code assignments may seem rather arbitrary, but there are certain patterns. The codes for uppercase letters are contiguous, as are the codes for lowercase letters. The codes for an uppercase letter and the corresponding lowercase letter differ by exactly one bit. Bit 5 ercase letter while other bits are the same. For example,

- **uppercase M codes as $4D_{16}$ = 1001101_2**
- **lowercase m codes as $6D_{16}$ = 1101101_2**

- The printable characters are grouped together from 20_{16} to $7E_{16}$. (A space is considered a printable character.) Numerals 0, 1, ..., 9 have ASCII codes 30_{16}, 31_{16}, ..., 39_{16}, respectively.

- The characters from 00_{16} to $1F_{16}$, along with $7F_{16}$, are known as control characters. For example, the ESC key on an ASCII keyboard generates a hex 1B code. The abbreviation ESC stands for extra services control but most people say "escape."

1-2 Character Codes

- The two ASCII control characters that will be used the most frequently in this book are 0D16 and 0A16, for carriage return (CR) and line feed (LF).
- The $0D_{16}$ code is generated by an ASCII keyboard when the Return or Enter key is pressed. When it is sent to an ASCII display, it causes the cursor to move to the beginning of the current line without going down to a new line.
- When carriage return is sent to an ASCII printer (at least one of older design), it causes the print head to move to the beginning of the line. The line feed code $0A_{16}$ causes an ASCII display to move the cursor straight down, or a printer to roll the paper up one line, in both cases without going to the beginning of the new line.

1-2 Character Codes

- Lesser-used control characters include
- form feed ($0C_{16}$), which causes many printers to eject a page;
- horizontal tab (09_{16}), which is generated by the tab key on the keyboard;
- backspace (08_{16}) generated by the Backspace key; and
- delete ($7F_{16}$) generated by the Delete key. Notice that the Backspace and Delete keys do not generate the same codes.
- The bell character (07_{16}) causes an audible signal when output to the display.

- **Many large computers represent characters using Extended Binary Coded Decimal Information Code (EBCDIC).**

1-2 Character Codes

How numbers are actually represented in a computer:

- using binary integers (often expressed in hex)
- using ASCII codes.

These methods have two problems:

- ✓ the number of bits is limited,
- ✓ it is not clear how to represent a negative number.

Memory is divided in to BYTES, each containing 8 bits.

A single ASCII code is normally stored in a byte. Recall that ASCII codes are seven bits long; the extra (left-hand, or high order) bit is set to 0.

1-2 Character Codes

To solve the first representation problem mentioned, you can simply include the code for a minus sign.

For example, the ASCII codes for the four characters -817 are 2D, 38, 31, and 37.

To solve the first problem, you could always agree to use a fixed number of bytes, perhaps padding on the left with ASCII codes for zeros or spaces.

Alternatively, you could use a variable number of bytes, but agree that the number ends with the last ASCII code for a digit, that is, terminating the string with a nondigit.

Chapter 1 – Representing Data in a Computer

1-2 Character Codes

using internal representations for binary values

you must choose a fixed number of bits for the representation. Most central processing units can do arithmetic on binary numbers having a few chosen lengths. For the Intel 80×86 family, these lengths are 8 bits (a byte), 16 bits (a word), 32 bits (a doubleword), and 64 bits (a quadword).

As an example, look at the word-length binary representation of 697.

- $\mathbf{697_{10} = 1010111001_2 = 0000001010111001_2}$

02	B9

00	00	02	B9

- A good system of representing nonnegative, or unsigned, numbers. This system cannot represent negative numbers. Also, for any given length.

1-3 The 2's complement system

1-3 The 2's complement system is similar to the above scheme for unsigned numbers, but it allows representation of negative numbers. **Numbers will be a fixed length**, so that you might find the "word-length 2's complement representation" of a number. The 2's complement representation for a nonnegative number is almost identical to the unsigned representation; that is, you represent the number in binary with enough leading zeros to fill up the desired length. Only one additional restriction exists-*for a positive number, the left-most bit must be zero*.

This means, for example, that the most positive number that can be represented in word-size 2's complement form is

$$\mathbf{0111111111111111_2 \text{ or } 7FFF_{16} \text{ or } 32767_{10}.}$$

you *cannot* simply change the leading bit from 0 to 1 to get the negative version of a number.

1-3 The 2's complement system

Obtaining Negative numbers

One method is to first express the unsigned number in hex, and then subtract this hex number from 10000_{16} to get the word length representation.

The number you subtract from is, in hex, a 1 followed by the number of 0's in the length of the representation; for example, 100000000_{16} to get the doubleword length representation.

1-3 The 2's complement system

Example:

The word-length 2's complement representation of the decimal number 76 is found by first converting the unsigned number 76 to its hex equivalent 4C, then by subtracting 4C from 10000.

```
1 0 0 0 0          F F F ¹0
    - 4 C            -  4  C
                   ---------
                    F F B 4
```

After borrowing, the subtraction is easy. The units digit is

$$10_{16} - C_{16} = 16_{10} - 12_{10} = 4$$

and the 16's position is

$$F_{16} - 4 = 15_{10} - 4_{10} = 11_{10} = B_{16}$$

Chapter 1 – Representing Data in a Computer

1-3 The 2's complement system

The operation of subtracting a number from 1 followed by an appropriate number of 0's is called **taking the 2's complement**, or **complementing the number**. Thus "2's complement" is used both as the name of a representation system and as the name of an operation.

Since a given 2's complement representation is a fixed length, obviously there is a maximum size number that can be stored in it.

For a word, the largest positive number is 7FFF

Positive numbers written in hex

can be identified by a leading hex digit of 0 through 7.

Negative numbers are distinguished

by a leading bit of 1, corresponding to hex digits of 8 through F.

1-3 The 2's complement system

How do you convert a 2's complement?

First, you must determine the sign of a 2's complement number.

To convert a positive 2's complement number to decimal, just treat it like any unsigned binary number and convert it by hand or with a hex calculator.

For example, the word-length 2's complement number 0D43 represents the decimal number 3395.

1-3 The 2's complement system

Dealing with a negative 2's complement number

one starting with a 1 bit or 8 through F in hex-is a little more complicated. **Note that any time you take the 2's complement of a number and then take the 2's complement of the result, you get back to the original number.**

$$N = 10000 - (10000 - N)$$

For example

$$10000 - (10000 - F39E) = 10000 - C62 = F39E$$

This says again that the 2's complement operation corresponds to negation. Because of this, if you start with a bit pattern representing a negative number, the 2's complement operation can be used to find the positive (unsigned) number corresponding it.

1-3 The 2's complement system

Example :

The word-length 2's complement number **E973** represents a negative value since the sign bit (leading bit) is 1 (E = 1110). Taking the complement finds the corresponding positive number.

$$\mathbf{10000 - E973 = 168D = 5773_{10}}$$

This means that the decimal number represented by E973 is **−5773**.

1-3 The 2's complement system

The word-length 2's complement representations with a leading 1 bit range from 8000 to FFFF. These convert to decimal as follows:

$$10000 - 8000 = 8000 = 32768_{10}$$

so 8000 is the representation of 32768. Similarly,

$$10000 - FFFF = 1$$

so FFFF is the representation of −1.

Recall that the largest positive decimal integer that can be represented as a word-length 2's complement number is 32767; the range of decimal numbers that can be represented in word-length 2's complement form is

−32768 to 32767.

1.4 Addition and Subtraction of 2's Complement Numbers

To add two 2's complement numbers, simply add them as if they were unsigned binary numbers.

The 80×86 architecture uses the same addition instructions for unsigned and signed numbers.

Example1:

0A07	2567
+ 01D3	+ 467
0BDA	3034

The answer is correct in this case since $BDA_{16} = 3034_{10}$.

1.4 Addition and Subtraction of 2's Complement Numbers

Now, 0206 and FFB0 are added.

0206	518	518
+ FFB0	+ (-80)	+ 65456
101B6	438	65974

These are, of course, positive as unsigned numbers, but interpreted as 2's complement signed numbers, 0206 is a positive number and FFB0 is negative. This means that there are two decimal versions of the addition problem. There certainly appears to be a problem since it will not even fit in a word. In fact, 101B6 is the hex version of 65974.

However, if the numbers are interpreted as signed and you ignore the extra 1 on the left, then the word 01B6 is the 2's complement representation of the decimal number 438.

Chapter 1 – Representing Data in a Computer

1.4 Addition and Subtraction of 2's Complement Numbers

Now FFE7 and FFF6 are added, both negative numbers in a signed interpretation. Again, both signed and unsigned decimal interpretations are shown.

FFE7	**(-25)**	**65511**
+ FFF6	**+ (-10)**	**+ 65526**
1FFDD	**-35**	**131037**

Again, the sum in hex is too large to fit in two bytes, but if you throw away the extra 1, then FFDD is the correct word-length 2's complement representation of 35.

Each of this addition and the previous one have a carry out of the usual high order position into an extra digit. The remaining digits give the correct 2's complement representation. The remaining digits are not always the correct 2's complement sum, however.

1.4 Addition and Subtraction of 2's Complement Numbers

Consider the addition of the following two positive numbers:

483F	18495
+ 645A	+ 25690
AC99	44185

There was no carry out of the high order digit, but the signed interpretation is plainly incorrect since AC99 represents the *negative* number 21351. Intuitively, what went wrong is that the decimal sum 44185 is bigger than the maximal value 32767 that can be stored in the two bytes of a word. However, when these numbers are interpreted as unsigned, the sum is correct.

1.4 Addition and Subtraction of 2's Complement Numbers

The following is another example showing a "wrong" answer, this time resulting from adding two numbers that are negative in their signed interpretation.

E9FF	(-5633)	59903
+ 8CF0	+ (-29456)	+ 36080
176EF	- 35089	95983

This time there is a carry, but the remaining four digits 76 EF cannot be the right-signed answer since they represent the *positive* number 30447. Again, intuition tells you that something had to go wrong since -32768 is the most negative number that can be stored in a word.

Chapter 1 – Representing Data in a Computer

1.4 Addition and Subtraction of 2's Complement Numbers

In the above "incorrect" examples, **overflow occurred**. As a human being, you detect overflow by the incorrect signed answer.

There may be a carry *into* this position and/or a carry *out of* this position into the "extra" bit.

This "carry out" (into the extra bit) is what was called just "carry" above and was seen as the extra hex 1. This table identifies when overflow does or does not occur. The table can be summarized by saying that overflow occurs when the number of carries into the sign position is different from the number of carries out of the sign position.

Carry into sign bit	**Carry out of sign bit**	**Overflow?**
No	No	No
No	Yes	Yes
Yes	No	Yes
yes	Yes	no

1.4 Addition and Subtraction of 2's Complement Numbers

Each of the above addition examples is now shown again, this time in binary. Carries are written above the two numbers.

```
                   111
   0000 1010 0000 0111          0A07
 + 0000 0001 1101 0011        + 01D3
   -------------------          ----
   0000 1011 1101 1010          0BDA
```

This example has no carry into the sign position and no carry out, so there is no overflow.

1.4 Addition and Subtraction of 2's Complement Numbers

```
   1 1111 11
     0000 0010 0000 0110          0206
   + 1111 1111 1011 0000        + FFB0
   1 0000 0001 1011 0110         101B6
```

This example has a carry into the sign position and a carry out, so there is no overflow.

```
   1 1111 1111 11   11
     1111 1111 1110 0111          FFE7
   + 1111 1111 1111 0110        + FFF6
   1 1111 1111 1101 1101         1FFDD
```

Again, there is both a carry into the sign position and a carry out, so there is no overflow.

1.4 Addition and Subtraction of 2's Complement Numbers

```
    1         1111 11
    0100 1000 0011 1111          483F
  + 0110 0100 0101 1010        + 645A
    ___________________          ____
    1010 1100 1001 1001          AC99
```

Overflow does occur in this addition since there is a carry into the sign position, but no carry out.

```
  1    1    11 111
    1110 1001 1111 1111          E9FF
  + 1000 1100 1111 0000        + 8CF0
    ___________________          ____
  1 0111 0110 1110 1111         176EF
```

There is also overflow in this addition since there is a carry out of the sign bit, but no carry in.

1.4 Addition and Subtraction of 2's Complement Numbers

In a computer, subtraction *a* − *b* of numbers *a* and *b* is usually performed by taking the 2's complement of *b* and adding the result to *a*. This corresponds to adding the negation of *b*. For example, for the decimal subtraction 195 - 618 = -423,

```
  00C3          00C3
- 026A         +FD96      (FD96 is 2's complement of 026A)
                FE59      (The hex digits FE59 do represent −423)
```

Looking at the above addition in binary

```
             11       11
      0000 0000 1100 0011
    + 1111 1101 1001 0110
      1111 1110 0101 1001
```

Notice that there was no carry in the addition. However, this subtraction did involve a borrow. **A borrow occurs in the subtraction *a* − *b* when *b* is larger than *a* as *unsigned* numbers.**

Chapter 1 – Representing Data in a Computer

1.4 Addition and Subtraction of 2's Complement Numbers

Computer hardware can detect a borrow in subtraction by looking at whether on not a carry occurred in the corresponding addition. If there is no carry in the addition, then there is a borrow in the subtraction. If there is a carry in the addition, then there is no borrow in the subtraction.

Here is one more example. Doing the decimal subtraction 985 - 411 = 574,

```
                            1 1111 1111 1        1
   03D9          03D9         0000 0011 1101 1001
 - 019B         +FE65       + 1111 1110 0110 0101
              1 023E        1 0000 0010 0011 1110
```

Discarding the extra 1, the hex digits 023E do represent 574. This addition has a carry, so there is no borrow in the corresponding subtraction.

1.4 Addition and Subtraction of 2's Complement Numbers

A computer detects overflow in subtraction by determining whether or not overflow occurs in the corresponding addition problem.

- ✓ If overflow occurs in the addition problem, then it occurs in the original subtraction problem;
- ✓ if it does not occur in the addition, then it does not occur in the original subtraction.

Overflow does occur if you use word-length 2's complement representations to attempt the subtraction −29123 −15447. As a human, you know that the correct answer −44570 is outside the range −32,768 to +32,767. In the computer hardware

```
                                     1    1    11   111   1
      8E3d           8E3D              1000 1110 0011 1101
    - 3C57          +C3A9            + 1100 0011 1010 1001
                    151E6            1 0101 0001 1110 0110
```

There is a carry out of the sign position, but no carry in, so overflow occurs.

1.4 Addition and Subtraction of 2's Complement Numbers

Although examples in this section have use word-size 2's complement representations, the same techniques apply when performing addition or subtraction with byte-size, doubleword-size, or other size 2's complement numbers.

Chapter 1 – Representing Data in a Computer

1.5 Other Systems for Representing Numbers

section introduces three additional schemes, 1's complement, binary coded decimal (BCD), and floating point.

- ✓ The 1's complement system is an alternative scheme for representing signed integers. (not the Intel 80×86 family)
- ✓ Binary coded decimal and floating point forms are used in 80×86 computers

The primary reason for introducing them here is to illustrate that there are many alternative representations for numeric data, each valid when used in the correct context.

1.5 Other Systems for Representing Numbers

The 1's complement system is similar to 2's complement.

- ✓ A fixed length is chosen for the representation and
- ✓ A positive integer is simply the binary form of the number, padded with one or more leading zeros on the left to get the desired length.
- ✓ To take the negative of the number, each bit is "complemented".
- ✓ This operation is sometimes referred to as taking the 1's complement of a number.
- ✓ Although it is easier to negate an integer using 1's complement than 2's complement, the 1's complement system has several disadvantages.
- ✓ There are two representations for zero (why?), an awkward situation.

97 $\Rightarrow$ 0110 0001 (61 in hex).

-97 $\Rightarrow$ 1001 1110 (9E in hex),

- ✓ There is a useful connection between taking the 1's complement and taking the 2's complement of a binary number. If you take the 1's complement of a number and then add 1, you get the 2's complement.

1.5 Other Systems for Representing Numbers

In binary coded decimal (BCD) schemes, each decimal digit is coded with a string of bits with fixed length, and these strings are pieced together to form the representation.

One BCD representation of the decimal number **926708** is **1001 0010 0110 0111 0000 1000**.

92	67	08

For purposes of illustration, assume a four-byte representation. For now, without leaving room for a sign, eight binary-coded decimal digits can be stored in four bytes.

Decimal	BCD bit Pattern
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Chapter 1 – Representing Data in a Computer
1.5 Other Systems for Representing Numbers

For purposes of illustration, assume a four-byte representation. For now, without leaving room for a sign, eight binary-coded decimal digits can be stored in four bytes. Given these choices, the decimal number 3691 has the BCD representation

00	00	36	91

Notice that the doubleword 2's complement representation for the same number

00	00	0E	6B

and that the ASCII codes for the four numerals are

33	36	39	31

1.5 Other Systems for Representing Numbers

- ✓ It is not as efficient for a computer to do arithmetic with numbers in a BCD format as with 2's complement numbers.

- ✓ It is usually very inefficient to do arithmetic on numbers represented using ASCII codes.

- ✓ However, ASCII codes are the only method so far for representing a number that is not an integer.

1.5 Other Systems for Representing Numbers

Floating point schemes store numbers in a form that corresponds closely to scientific notation.

(IEEE is the abbreviation for the Institute of Electrical and Electronics Engineers.)

First, 78.375 must be converted to binary. In binary, the positions to the right of the binary point correspond to negative powers of two Since 0.375 = 3/8 = 1/4 + 1/8 = .012 + .0012, 0.37510 = 0.0112. The whole part 78 is 1001110 in binary, so

$$78.375_{10} = 1001110.011_2$$

1.5 Other Systems for Representing Numbers

Next this is expressed in binary scientific notation with the mantissa written with 1 before the radix point.

$$1001110.011_2 = 1.001110011 \times 2^6$$

The notation here is really mixed; it would be more proper to write 2^6 as 10^{110}, but it is more convenient to use the decimal form.

- ✓ left bit 0 for a positive number (1 means negative)
- ✓ 1000 0101 for the exponent. This is the actual exponent of 6, plus a bias of 127, with the sum, 133, in 8 bits.
- ✓ 00111001100000000000000, the fraction expressed with the leading 1 removed and padded with zeros on the right to make 23 bits
- ✓ The entire number is then 010000101 00111001100000000000000. Regrouping gives 0100 0010 1001 1100 1100 0000 0000 0000, or, in hex

42	9C	C0	00

1.5 Other Systems for Representing Numbers

To summarize, the following steps are used to convert a decimal number to IEEE single format:

- ✓ **The leading bit of the floating point format is 0 for a positive number and 1 for a negative number.**
- ✓ **Write the unsigned number in binary.**
- ✓ **Write the binary number in binary scientific notation $f_{23}.f_{22} \ldots f_0$ 2^e, where f_{23} = 1. There are 24 fraction bits, but it is not necessary to write trailing 0's.**
- ✓ **Add a bias of 127_{10} to the exponent *e*. This sum, in binary form, is the next 8 bits of the answer, following the sign bit. (Adding a bias is an alternative to storing the exponent as a signed number.)**
- ✓ **The fraction bits $f_{22}f_{21} \ldots f_0$ form the last 23 bits of the floating point number. The leading bit f_{23} (which is always 1) is dropped.**

Chapter Summary

- ✓ All data are represented in a computer using electronic signals. These can be interpreted as patterns of binary digits (bits). These bit patterns can be thought of as binary numbers. Numbers can be written in decimal, hexadecimal, or binary forms.
- ✓ For representing characters, most microcomputers use ASCII codes. One code is assigned for each character, including nonprintable control characters.
- ✓ Integer values are represented in a predetermined number of bits in 2's complement form; a positive number is stored as a binary number (with at least one leading zero to make the required length), and the pattern for a negative number can be obtained by subtracting the positive form from a 1 followed by as many 0's as are used in the length. A 2's complement negative number always has a leading 1 bit. A hex calculator, used with care, can simplify working with 2's complement numbers.
- ✓ Addition and subtraction are easy with 2's complement numbers. Since the length of a 2's complement number is limited, there is the possibility of a carry, a borrow, or overflow.
- ✓ Other formats in which numbers are stored are 1's complement, binary coded decimal (BCD), and floating point.

End of Chapter1