DSP Software

DSP applications are usually programmed in the same languages as other science and engineering tasks, such as: C, BASIC and assembly. The power and versatility of C makes it the language of choice for computer scientists and other professional programmers. On the other hand, the simplicity of BASIC makes it ideal for scientists and engineers who only occasionally visit the programming world. Regardless of the language you use, most of the important DSP software issues are buried far below in the realm of whirling ones and zeros. This includes such topics as: how numbers are represented by bit patterns, round-off error in computer arithmetic, the computational speed of different types of processors, etc. This chapter is about the things you can do at the *high level* to avoid being trampled by the *low level* internal workings of your computer.

Computer Numbers

Digital computers are very proficient at storing and recalling numbers; unfortunately, this process isn't without error. For example, you instruct your computer to store the number: 1.41421356. The computer does its best, storing the closest number it *can* represent: 1.41421354. In some cases this error is quite insignificant, while in other cases it is disastrous. As another illustration, a classic computational error results from the addition of two numbers with very different values, for example, 1 and 0.00000001. We would like the answer to be 1.00000001, but the computer replies with 1. An understanding of how computers store and manipulate numbers allows you to anticipate and correct these problems *before* your program spits out meaningless data.

These problems arise because a fixed number of bits are allocated to store each number, usually 8, 16, 32 or 64. For example, consider the case where eight bits are used to store the value of a variable. Since there are $2^8 = 256$ possible bit patterns, the variable can only take on 256 different values. This is a fundamental limitation of the situation, and there is nothing we can do about it. The part we *can* control is what value we declare each bit pattern

to represent. In the simplest cases, the 256 bit patterns might represent the integers from 0 to 255, 1 to 256, -127 to 128, etc. In a more unusual scheme, the 256 bit patterns might represent 256 exponentially related numbers: 1, 10, 100, 1000, $\cdots$, 10^{254}, 10^{255}. Everyone accessing the data must understand what value each bit pattern represents. This is usually provided by an algorithm or formula for converting between the represented value and the corresponding bit pattern, and back again.

While many encoding schemes are possible, only two general formats have become common, *fixed point* (also called integer numbers) and *floating point* (also called real numbers). In this book's BASIC programs, fixed point variables are indicated by the % symbol as the last character in the name, such as: I%, N%, SUM%, etc. All other variables are floating point, for example: X, Y, MEAN, etc. When you evaluate the formats presented in the next few pages, try to understand them in terms of their **range** (the largest and smallest numbers they can represent) and their **precision** (the size of the gaps between numbers).

Fixed Point (Integers)

Fixed point representation is used to store *integers*, the positive and negative whole numbers: $\cdots$ -3, -2, -1, 0, 1, 2, 3, $\cdots$. High level programs, such as C and BASIC, usually allocate 16 bits to store each integer. In the simplest case, the $2^{16} = 65,536$ possible bit patterns are assigned to the numbers 0 through 65,535. This is called **unsigned integer** format, and a simplified example is shown in Fig. 4-1 (using only 4 bits per number). Conversion between the bit pattern and the number being represented is nothing more than changing between base 2 (binary) and base 10 (decimal). The disadvantage of unsigned integer is that negative numbers cannot be represented.

Offset binary is similar to unsigned integer, except the decimal values are *shifted* to allow for negative numbers. In the 4 bit example of Fig. 4-1, the decimal numbers are offset by *seven*, resulting in the 16 bit patterns corresponding to the integer numbers -7 through 8. In this same manner, a 16 bit representation would use 32,767 as an offset, resulting in a range between -32,767 and 32,768. Offset binary is not a standardized format, and you will find other offsets used, such 32,768. The most important use of offset binary is in ADC and DAC. For example, the input voltage range of -5v to 5v might be mapped to the digital numbers 0 to 4095, for a 12 bit conversion.

Sign and magnitude is another simple way of representing negative integers. The far left bit is called the **sign bit**, and is made a *zero* for positive numbers, and a *one* for negative numbers. The other bits are a standard binary representation of the absolute value of the number. This results in one wasted bit pattern, since there are two representations for zero, 0000 (positive zero) and 1000 (negative zero). This encoding scheme results in 16 bit numbers having a range of -32,767 to 32,767.

UNSIGNED INTEGER		OFFSET BINARY		SIGN AND MAGNITUDE		TWO'S COMPLEMENT	
Decimal	Bit Pattern	Decimal	Bit Pattern	Decimal	Bit Pattern	Decimal	Bit Pattern
15	1111	8	1111	7	0111	7	0111
14	1110	7	1110	6	0110	6	0110
13	1101	6	1101	5	0101	5	0101
12	1100	5	1100	4	0100	4	0100
11	1011	4	1011	3	0011	3	0011
10	1010	3	1010	2	0010	2	0010
9	1001	2	1001	1	0001	1	0001
8	1000	1	1000	0	0000	0	0000
7	0111	0	0111	0	1000	-1	1111
6	0110	-1	0110	-1	1001	-2	1110
5	0101	-2	0101	-2	1010	-3	1101
4	0100	-3	0100	-3	1011	-4	1100
3	0011	-4	0011	-4	1100	-5	1011
2	0010	-5	0010	-5	1101	-6	1010
1	0001	-6	0001	-6	1110	-7	1001
0	0000	-7	0000	-7	1111	-8	1000

16 bit range:	16 bit range	16 bit range	16 bit range
0 to 65,535	-32,767 to 32,768	-32,767 to 32,767	-32,768 to 32,767

FIGURE 4-1
Common formats for fixed point (integer) representation. Unsigned integer is a simple binary format, but cannot represent negative numbers. Offset binary and sign & magnitude allow negative numbers, but they are difficult to implement in hardware. Two's complement is the easiest to design hardware for, and is the most common format for general purpose computing.

These first three representations are conceptually simple, but difficult to implement in hardware. Remember, when A=B+C is entered into a computer program, some hardware engineer had to figure out how to make the bit pattern representing B, combine with the bit pattern representing C, to form the bit pattern representing A.

Two's complement is the format loved by hardware engineers, and is how integers are usually represented in computers. To understand the encoding pattern, look first at decimal number zero in Fig. 4-1, which corresponds to a binary zero, 0000. As we count upward, the decimal number is simply the binary equivalent (0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011, etc.). Now, remember that these four bits are stored in a register consisting of 4 flip-flops. If we again start at 0000 and begin subtracting, the digital hardware automatically counts in two's complement: 0 = 0000, -1 = 1111, -2 = 1110, -3 = 1101, etc. This is analogous to the odometer in a new automobile. If driven forward, it changes: 00000, 00001, 00002, 00003, and so on. When driven backwards, the odometer changes: 00000, 99999, 99998, 99997, etc.

Using 16 bits, two's complement can represent numbers from -32,768 to 32,767. The left most bit is a 0 if the number is positive or zero, and a 1 if the number is negative. Consequently, the left most bit is called the **sign bit**, just as in sign & magnitude representation. Converting between decimal and two's complement is straightforward for positive numbers, a simple decimal to binary

conversion. For negative numbers, the following algorithm is often used: (1) take the absolute value of the decimal number, (2) convert it to binary, (3) complement all of the bits (ones become zeros and zeros become ones), (4) add 1 to the binary number. For example: -5 → 5 → 0101 → 1010 → 1011. Two's complement is hard for humans, but easy for digital electronics.

Floating Point (Real Numbers)

The encoding scheme for floating point numbers is more complicated than for fixed point. The basic idea is the same as used in scientific notation, where a **mantissa** is multiplied by ten raised to some **exponent**. For instance, 5.4321×10^6, where 5.4321 is the *mantissa* and 6 is the *exponent*. Scientific notation is exceptional at representing very large and very small numbers. For example: 1.2×10^{50}, the number of atoms in the earth, or 2.6×10^{-23}, the distance a turtle crawls in one second, compared to the diameter of our galaxy. Notice that numbers represented in scientific notation are *normalized* so that there is only a single nonzero digit left of the decimal point. This is achieved by adjusting the exponent as needed.

Floating point representation is similar to scientific notation, except everything is carried out in base two, rather than base ten. While several similar formats are in use, the most common is ANSI/IEEE Std. 754-1985. This standard defines the format for 32 bit numbers called **single precision**, as well as 64 bit numbers called **double precision**. As shown in Fig. 4-2, the 32 bits used in single precision are divided into three separate groups: bits 0 through 22 form the mantissa, bits 23 through 30 form the exponent, and bit 31 is the sign bit. These bits form the floating point number, *v*, by the following relation:
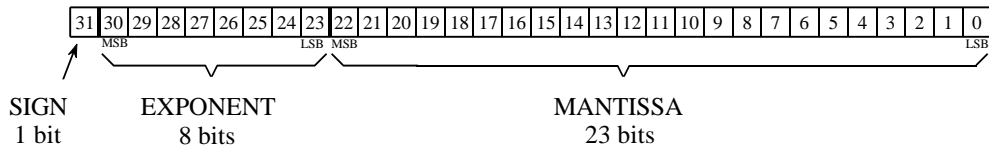
EQUATION 4-1
Equation for converting a bit pattern into a floating point number. The number is represented by *v*, *S* is the value of the sign bit, *M* is the value of the mantissa, and *E* is the value of the exponent.

$$v = (-1)^S \times M \times 2^{E-127}$$

The term: $(-1)^S$, simply means that the sign bit, *S*, is 0 for a positive number and 1 for a negative number. The variable, *E*, is the number between 0 and 255 represented by the eight exponent bits. Subtracting 127 from this number allows the exponent term to run from 2^{-127} to 2^{128}. In other words, the exponent is stored in *offset binary* with an offset of 127.

The mantissa, *M*, is formed from the 23 bits as a *binary fraction*. For example, the decimal fraction: 2.783, is interpreted: $2 + 7/10 + 8/100 + 3/1000$. The binary fraction: 1.0101, means: $1 + 0/2 + 1/4 + 0/8 + 1/16$. Floating point numbers are *normalized* in the same way as scientific notation, that is, there is only one nonzero digit left of the decimal point (called a *binary point* in

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

SIGN — 1 bit

EXPONENT — 8 bits

MANTISSA — 23 bits

Example 1

0 00000111 11000000000000000000000

+ 7 0.75

$+1.75 \times 2^{(7-127)} = +1.316554 \times 10^{-36}$

Example 2

1 10000001 01100000000000000000000

– 129 0.375

$-1.375 \times 2^{(129-127)} = -5.500000$

FIGURE 4-2
Single precision floating point storage format. The 32 bits are broken into three separate parts, the sign bit, the exponent and the mantissa. Equations 4-1 and 4-2 shows how the represented number is found from these three parts. MSB and LSB refer to "most significant bit" and "least significant bit," respectively.

base 2). Since the only nonzero number that exists in base two is 1, the leading digit in the mantissa will always be a 1, and therefore does not need to be stored. Removing this redundancy allows the number to have an additional one bit of precision. The 23 stored bits, referred to by the notation: $m_{22}, m_{21}, m_{21}, \cdots, m_0,$ form the mantissa according to:

EQUATION 4-2
Algorithm for converting the bit pattern into the mantissa, M, used in Eq. 4-1.

$$M = 1.m_{22}m_{21}m_{20}m_{19} \cdots m_2 m_1 m_0$$

In other words, $M = 1 + m_{22}2^{-1} + m_{21}2^{-2} + m_{20}2^{-3} \cdots$. If bits 0 through 22 are all *zeros*, M takes on the value of one. If bits 0 through 22 are all *ones*, M is just a hair under two, i.e., $2 - 2^{-23}$.

Using this encoding scheme, the largest number that can be represented is: $\pm(2 - 2^{-23}) \times 2^{128} = \pm 6.8 \times 10^{38}$. Likewise, the smallest number that can be represented is: $\pm 1.0 \times 2^{-127} = \pm 5.9 \times 10^{-39}$. The IEEE standard reduces this range slightly to free bit patterns that are assigned special meanings. In particular, the largest and smallest numbers allowed in the standard are

$\pm 3.4 \times 10^{38}$ and $\pm 1.2 \times 10^{-38}$, respectively. The freed bit patterns allow three special classes of numbers: (1) ± 0 is defined as all of the mantissa and exponent bits being zero. (2) $\pm \infty$ is defined as all of the mantissa bits being zero, and all of the exponent bits being one. (3) A group of very small *unnormalized* numbers between $\pm 1.2 \times 10^{-38}$ and $\pm 1.4 \times 10^{-45}$. These are lower precision numbers obtained by removing the requirement that the leading digit in the mantissa be a one. Besides these three special classes, there are bit patterns that are not assigned a meaning, commonly referred to as NANs (Not A Number).

The IEEE standard for double precision simply adds more bits to both the mantissa and exponent. Of the 64 bits used to store a double precision number, bits 0 through 51 are the mantissa, bits 52 through 62 are the exponent, and bit 63 is the sign bit. As before, the mantissa is between one and just under two, i.e., $M = 1 + m_{51}2^{-1} + m_{50}2^{-2} + m_{49}2^{-3} \cdots$. The 11 exponent bits form a number between 0 and 2047, with an offset of 1023, allowing exponents from 2^{-1023} to 2^{1024}. The largest and smallest numbers allowed are $\pm 1.8 \times 10^{308}$ and $\pm 2.2 \times 10^{-308}$, respectively. These are incredibly large and small numbers! It is quite uncommon to find an application where single precision is not adequate. You will probably never find a case where double precision limits what you want to accomplish.

Number Precision

The errors associated with number representation are very similar to quantization errors during ADC. You *want* to store a continuous range of values; however, you *can* represent only a finite number of quantized levels. Every time a new number is generated, after a math calculation for example, it must be rounded to the nearest value that can be stored in the format you are using.

As an example, imagine that you allocate 32 bits to store a number. Since there are exactly $2^{32} = 4,294,967,296$ different bit patterns possible, you can represent exactly 4,294,967,296 different numbers. Some programming languages allow a variable called a **long integer**, stored as 32 bits, fixed point, two's complement. This means that the 4,294,967,296 possible bit patterns represent the integers between -2,147,483,648 and 2,147,483,647. In comparison, single precision floating point spreads these 4,294,967,296 bit patterns over the much larger range: -3.4×10^{38} to 3.4×10^{38}.

With fixed point variables, the gaps between adjacent numbers are always *exactly one*. In floating point notation, the gaps between adjacent numbers vary over the represented number range. If we randomly pick a floating point number, the gap next to that number is approximately *ten million times smaller* than the number itself (to be exact, 2^{-24} to 2^{-23} times the number). This is a key concept of floating point notation: large numbers have large gaps between them, while small numbers have small gaps. Figure 4-3 illustrates this by showing consecutive floating point numbers, and the gaps that separate them.

0.00001233862713
0.00001233862804
0.00001233862895
0.00001233862986
$\vdots$

spacing = 0.00000000000091
(1 part in 13 million)

1.000000000
1.000000119
1.000000238
1.000000358
$\vdots$

spacing = 0.000000119
(1 part in 8 million)

1.996093750
1.996093869
1.996093988
1.996094108
$\vdots$

spacing = 0.000000119
(1 part in 17 million)

636.0312500
636.0313110
636.0313720
636.0314331
$\vdots$

spacing = 0.0000610
(1 part in 10 million)

217063424.0
217063440.0
217063456.0
217063472.0

spacing = 16.0
(1 part in 14 million)

FIGURE 4-3
Examples of the spacing between single precision floating point numbers. The spacing between adjacent numbers is always between about 1 part in 8 million and 1 part in 17 million of the value of the number.

The program in Table 4-1 illustrates how **round-off error** (quantization error in math calculations) causes problems in DSP. Within the program loop, two random numbers are added to the floating point variable X, and then subtracted back out again. Ideally, this should do *nothing*. In reality, the round-off error from each of the arithmetic operations causes the value of X to gradually drift away from its initial value. This drift can take one of two forms depending on how the errors add together. If the round-off errors are randomly positive and negative, the value of the variable will randomly increase and decrease. If the errors are predominately of the same sign, the value of the variable will drift away much more rapidly and uniformly.

TABLE 4-1
Program for demonstrating floating point error accumulation. This program initially sets the value of X to 1.000000, and then runs through a loop that should ideally do *nothing*. During each loop, two random numbers, A and B, are added to X, and then subtracted back out. The accumulated error from these additions and subtraction causes X to wander from its initial value. As Fig. 4-4 shows, the error may be random or additive.

```
100  X = 1                  'initialize X
110  '
120  FOR I% = 0 TO 2000
130    A = RND              'load random numbers
140    B = RND              'into A and B
150    '
160    X = X + A            'add A and B to X
170    X = X + B
180    X = X - A            'undo the additions
190    X = X - B
200    '
210    PRINT X              'ideally, X should be 1
220  NEXT I%
230  END
```

FIGURE 4-4
Accumulation of round-off error in floating point variables. These curves are generated by the program shown in Table 4-1. When a floating point variable is repeatedly used in arithmetic operations, accumulated round-off error causes the variable's value to drift. If the errors are both positive and negative, the value will increase and decrease in a random fashion. If the round-off errors are predominately of the same sign, the value will change in a much more rapid and uniform manner.
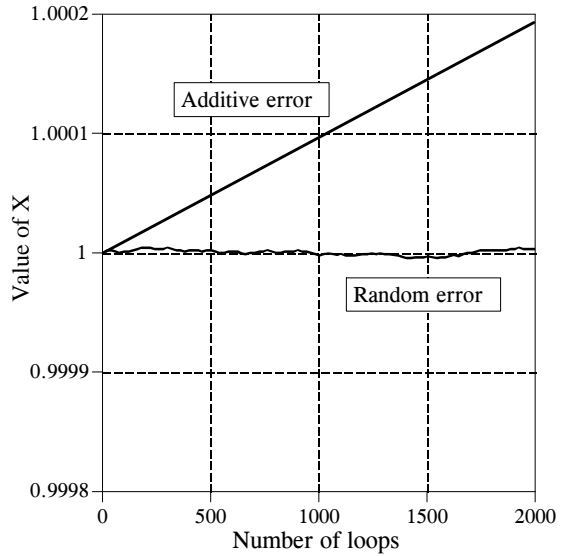

Figure 4-4 shows how the variable, *X*, in this example program drifts in value. An obvious concern is that *additive error* is much worse than *random error*. This is because random errors tend to cancel with each other, while the additive errors simply accumulate. The additive error is roughly equal to the round-off error from a single operation, multiplied by the total number of operations.   In comparison, the random error only increases in proportion to the *square root* of the number of operations. As shown by this example, additive error can be hundreds of times worse than random error for common DSP algorithms.

Unfortunately, it is nearly impossible to control or predict which of these two behaviors a particular algorithm will experience. For example, the program in Table 4-1 generates an additive error. This can be changed to a random error by merely making a slight modification to the numbers being added and subtracted. In particular, the random error curve in Fig. 4-4 was generated by defining: *A = EXP(RND)* and *B = EXP(RND)*, rather than: *A = RND* and *B = RND*. Instead of *A* and *B* being randomly distributed numbers between 0 and 1, they become exponentially distributed values between 1 and 2.718. Even this small change is sufficient to toggle the mode of error accumulation.

Since we can't control which way the round-off errors accumulate, keep in mind the worse case scenario. Expect that every single precision number will have an error of about *one part in forty million, multiplied by the number of operations it has been through*. This is based on the assumption of additive error, and the average error from a single operation being one-quarter of a quantization level. Through the same analysis, every double precision number has an error of about *one part in forty quadrillion, multiplied by the number of operations*.

```
100 'Floating Point Loop Control        100 'Integer Loop Control
110 FOR X = 0 TO 10 STEP 0.01           110 FOR I% = 0 TO 1000
120   PRINT X                           120   X = I%/100
130 NEXT X                              130   PRINT X
                                        140 NEXT I%
```

Program Output:	*Program Output:*
0.00	0.00
0.01	0.01
0.02	0.02
0.03	0.03
⋮	⋮
9.960132	9.96
9.970133	9.97
9.980133	9.98
9.990133	9.99
	10.00

TABLE 4-2
Comparison of floating point and integer variables for loop control. The left hand program controls
the FOR-NEXT loop with a floating point variable, *X*. This results in an accumulated round-off error
of 0.000133 by the end of the program, causing the last loop value, *X* = 10.0, to be omitted. In
comparison, the right hand program uses an integer, *I%*, for the loop index. This provides *perfect*
precision, and guarantees that the proper number of loop cycles will be completed.

Table 4-2 illustrates a particularly annoying problem of round-off error.
Each of the two programs in this table perform the same task: printing 1001
numbers equally spaced between 0 and 10. The left-hand program uses the
floating point variable, *X*, as the loop index. When instructed to execute a
loop, the computer begins by setting the index variable to the starting value
of the loop (0 in this example). At the end of each loop cycle, the step size
(0.01 in the case) is *added* to the index. A decision is then made: are more
loops cycles required, or is the loop completed? The loop ends when the
computer finds that the value of the index is *greater than* the termination
value (in this example, 10.0). As shown by the generated output, round-off
error in the additions cause the value of *X* to accumulate a significant
discrepancy over the course of the loop. In fact, the accumulated error
prevents the execution of the last loop cycle. Instead of *X* having a value
of 10.0 on the last cycle, the error makes the last value of *X* equal to
10.000133. Since *X* is greater than the termination value, the computer
thinks its work is done, and the loop prematurely ends. This missing last
value is a common bug in many computer programs.

In comparison, the program on the right uses an integer variable, *I%*, to
control the loop. The addition, subtraction, or multiplication of two integers
always produces another integer. This means that fixed point notation has
absolutely no round-off error with these operations. Integers are ideal for
controlling loops, as well as other variables that undergo multiple
mathematical operations. The last loop cycle is guaranteed to execute!
Unless you have some strong motivation to do otherwise, always use
integers for loop indexes and counters.

If you must use a floating point variable as a loop index, try to use fractions that are a power of *two* (such as: 1/2, 1/4, 3/8, 27/16), instead of a power of *ten* (such as: 0.1, 0.6, 1.4, 2.3, etc.).   For instance, it would be better to use: FOR X = 1 TO 10 STEP 0.125, rather than: FOR X = 1 to 10 STEP 0.1.   This allows the index to always have an *exact* binary representation, thereby reducing round-off error.   For example, the decimal number: 1.125, can be represented exactly in binary notation: $1.0010000000000000000000000 \times 2^0$.   In comparison, the decimal number: 1.1, falls *between* two floating point numbers: 1.0999999046 and 1.1000000238   (in binary these numbers are: $1.0001100110011001100 \times 2^0$ and $1.0001100110011001101 \times 2^0$). This results in an inherent error each time 1.1 is encountered in a program.

A useful fact to remember: single precision floating point has an *exact* binary representation for every whole number between ±16.8 million (to be exact, $\pm 2^{24}$).   Above this value, the gaps between the levels are larger than one, causing some whole number values to be missed.   This allows floating point whole numbers (between ±16.8 million) to be added, subtracted and multiplied, with no round-off error.

Execution Speed: Program Language

DSP programming can be loosely divided into three levels of sophistication: *Assembly*, *Compiled*, and *Application Specific*. To understand the difference between these three, we need to start with the very basics of digital electronics.   All microprocessors are based around a set of internal binary registers, that is, a group of flip-flops that can store a series of ones and zeros.   For example, the 8088 microprocessor, the core of the original IBM PC, has *four* general purpose registers, each consisting of 16 bits. These are identified by the names: AX, BX, CX, and DX.   There are also *nine* additional registers with special purposes, called: SI, DI, SP, BP, CS, DS, SS, ES, and IP.   For example, IP, the Instruction Pointer, keeps track of where in memory the next instruction resides.

Suppose you write a program to add the numbers: 1234 and 4321.   When the program begins, IP contains the address of a section of memory that contains a pattern of ones and zeros, as shown in Table 4-3.   Although it looks meaningless to most humans, this pattern of ones and zeros contains all of the commands and data required to complete the task.   For example, when the microprocessor encounters the bit pattern: 00000011 11000011, it interpreters it as a command to take the 16 bits stored in the BX register, add them in binary to the 16 bits stored in the AX register, and store the result in the AX register.   This level of programming is called **machine code**, and is only a hair above working with the actual electronic circuits.

Since working in binary will eventually drive even the most patient engineer crazy, these patterns of ones and zeros are assigned names according to the function they perform.   This level of programming is called **assembly**, and an example is shown in Table 4-4.   Although an assembly program is much easier to understand, it is fundamentally the same as programming in

10111001	00000000
11010010	10100001
00000100	00000000
10001001	00000000
00001110	10001011
00000000	00011110
00000000	00000010
10111001	00000000
11100001	00000011
00010000	11000011
10001001	10100011
00001110	00000100
00000010	00000000

TABLE 4-3
A *machine code* program for adding 1234 and 4321. This is the lowest level of programming: direct manipulation of the digital electronics. (The right column is a continuation of the left column).

machine code, since there is a one-to-one correspondence between the program commands and the action taken in the microprocessor. For example: ADD AX, BX translates to: 00000011 11000011. A program called an **assembler** is used to convert the assembly code in Table 4-4 (called the **source code**) into the patterns of ones and zeros shown in Table 4-3 (called the **object code** or **executable code**). This executable code can be directly run on the microprocessor. Obviously, assembly programming requires an extensive understanding of the internal construction of the particular microprocessor you intend to use.

TABLE 4-4
An *assembly* program for adding 1234 and 4321. An *assembler* is a program that converts an assembly program into machine code.

```
MOV CX,1234        ;store 1234 in register CX, and then
MOV DS:[0],CX      ;transfer it to memory location DS:[0]

MOV CX,4321        ;store 4321 in register CX, and then
MOV DS:[2],CX      ;transfer it to memory location DS:[0]

MOV AX,DS:[0]      ;move variables stored in memory at
MOV BX,DS:[2]      ;DS:[0] and DS:[2] into AX & BX

ADD AX,BX          ;add AX and BX, store sum in AX

MOV DS:[4],AX      ;move the sum into memory at DS:[4]
```

Assembly programming involves the direct manipulation of the digital electronics: registers, memory locations, status bits, etc. The next level of sophistication can manipulate *abstract* variables without any reference to the particular hardware. These are called **compiled** or **high-level** languages. A dozen or so are in common use, such as: C, BASIC, FORTRAN, PASCAL, APL, COBOL, LISP, etc. Table 4-5 shows a BASIC program for adding 1234 and 4321. The programmer only knows about the variables A, B, and C, and nothing about the hardware.

TABLE 4-5
A *BASIC* program for adding 1234 and 4321. A *compiler* is a program that converts this type of high-level source code into machine code.

```
100  A = 1234
110  B = 4321
120  C = A+B
130  END
```

A program called a **compiler** is used to transform the high-level source code directly into machine code. This requires the compiler to *assign* hardware memory locations to each of the abstract variables being referenced. For example, the first time the compiler encounters the variable *A* in Table 4-5 (line 100), it understands that the programmer is using this symbol to mean a single precision floating point variable. Correspondingly, the compiler designates four bytes of memory that will be used for nothing but to hold the value of this variable. Each subsequent time that an *A* appears in the program, the computer knows to update the value of the four bytes as needed. The compiler also breaks complicated mathematical expressions, such as: $Y = LOG(X^{COS(Z)})$, into more basic arithmetic. Microprocessors only know how to add, subtract, multiply and divide. Anything more complicated must be done as a series of these four elementary operations.

High-level languages isolate the programmer from the hardware. This makes the programming much easier and allows the source code to be transported between different types of microprocessors. Most important, the programmer who uses a compiled language needs to know *nothing* about the internal workings of the computer. Another programmer has assumed this responsibility, the one who wrote the compiler.

Most compilers operate by converting the *entire* program into machine code *before* it is executed. An exception to this is a type of compiler called an **interpreter**, of which **interpreter BASIC** is the most common example. An interpreter converts a *single line* of source code into machine code, executes that machine code, and then goes on to the next line of source code. This provides an interactive environment for simple programs, although the execution speed is extremely slow (think a factor of 100).

The highest level of programming sophistication is found in **applications** packages for DSP. These come in a variety of forms, and are often provided to support specific hardware. Suppose you buy a newly developed DSP microprocessor to embed in your current project. These devices often have lots of built-in features for DSP: analog inputs, analog outputs, digital I/O, antialias and reconstruction filters, etc. The question is: how do you program it? In the worst case, the manufacturer will give you an *assembler*, and expect you to learn the internal architecture of the device. In a more typical scenario, a C *compiler* will be provided, allowing you to program without being bothered by how the microprocessor actually operates.

In the best case, the manufacturer will provide a sophisticated software package to help in the programming: libraries of algorithms, prewritten routines for I/O, debugging tools, etc. You might simply connect icons to form the desired system in an easy-to-use graphical display. The *things* you manipulate are signal pathways, algorithms for processing signals, analog I/O parameters, etc. When you are satisfied with the design, it is transformed into suitable machine code for execution in the hardware. Other types of applications packages are used with image processing, spectral analysis, instrumentation and control, digital filter design, etc. This is the shape of the future.

The distinction between these three levels can be very fuzzy. For example, most complied languages allow you to directly manipulate the hardware. Likewise, a high-level language with a well stocked library of DSP functions is very close to being an applications package. The point of these three catagories is understand what you are manipulating: (1) hardware, (2) abstract variables, or (3) entire procedures and algorithms.

There is also another important concept behind these classifications. When you use a high-level language, you are relying on the programmer who wrote the compiler to understand the best techniques for hardware manipulation. Similarly, when you use an applications package, you are relying on the programmer who wrote the package to understand the best DSP techniques. Here's the rub: these programmers have never seen the particular problem you are dealing with. Therefore, they cannot always provide you with an optimal solution. As you operate on a *higher* level, expect that the final machine code will be *less* efficient in terms of memory usage, speed, and precision.

Which programming language should you use? That depends on *who* you are and *what* you plan to do. Most computer scientists and programmers use C (or the more advanced C++). Power, flexibility, modularity; C has it all. C is so popular, the question becomes: Why would anyone program their DSP application in something other than C? Three answers come to mind. First, DSP has grown so rapidly that some organizations and individuals are stuck in the mode of other languages, such as FORTRAN and PASCAL. This is especially true of military and government agencies that are notoriously slow to change. Second, some applications require the utmost efficiency, only achievable by assembly programming. This falls into the category of "a little more speed for a lot more work." Third, C is not an especially easy language to master, especially for part time programmers. This includes a wide range of engineers and scientists who occasionally need DSP techniques to assist in their research or design activities. This group often turns to BASIC because of its simplicity.

Why was BASIC chosen for this book? This book is about *algorithms*, not programming style. You should be concentrating on DSP techniques, and not be distracted by the quirks of a particular language. For instance, all the programs in this book have *line numbers*. This makes it easy to describe how the program operates: "line 100 does such-and-such, line 110 does this-and that," etc. Of course, you will probably never use line numbers in your actual programs. The point is, *learning DSP* has different requirements than *using DSP*. There are many books on the market that provide exquisite source code for DSP algorithms. If you are simply looking for prewritten code to copy into your program, you are in the wrong place.

Comparing the execution speed of hardware or software is a thankless task; no matter what the result, the loser will cry that the match was unfair! Programmers who like high-level languages (such as traditional computer scientists), will argue that assembly is only 50% faster than compiled code, but five times more trouble. Those who like assembly (typically, scientists and

hardware engineers) will claim the reverse: assembly is five times faster, but only 50% more difficult to use. As in most controversies, both sides can provide selective data to support their claims.

As a rule-of-thumb, expect that a subroutine written in assembly will be between 1.5 and 3.0 times faster than the comparable high-level program. The only way to know the exact value is to write the code and conduct speed tests. Since personal computers are increasing in speed about 40% every year, writing a routine in assembly is equivalent to about a two year jump in hardware technology.

Most professional programmers are rather offended at the idea of using assembly, and gag if you suggest BASIC. Their rational is quite simple: assembly and BASIC discourage the use of good software practices. Good code should be *portable* (able to move from one type of computer to another), *modular* (broken into a well defined subroutine structure), and *easy to understand* (lots of comments and descriptive variable names). The weak structure of assembly and BASIC makes it difficult to achieve these standards. This is compounded by the fact that the people who are attracted to assembly and BASIC often have little formal training in proper software structure and documentation.

Assembly lovers respond to this attack with a zinger of their own. Suppose you write a program in C, and your competitor writes the same program in assembly. The end user's first impression will be that your program is junk because it is twice as slow. No one would suggest that you write large programs in assembly, only those portions of the program that need rapid execution. For example, many functions in DSP software libraries are written in assembly, and then accessed from larger programs written in C. Even the staunchest software purist will *use* assembly code, as long as they don't have to *write* it.

Execution Speed: Hardware

Computing power is increasing so rapidly, any *book* on the subject will be obsolete before it is published. It's an author's nightmare! The original IBM PC was introduced in 1981, based around the 8088 microprocessor with a 4.77 MHz clock and an 8 bit data bus. This was followed by a new generation of personal computers being introduced every 3-4 years: 8088 → 80286 → 80386 → 80486 → 80586 (Pentium). Each of these new systems boosted the computing speed by a factor of about *five* over the previous technology. By 1996, the clock speed had increased to 200 MHz, and the data bus to 32 bits. With other improvements, this resulted in an increase in computing power of nearly *one thousand* in only 15 years! You should expect *another* factor of one thousand in the *next* 15 years.

The only way to obtain up-to-date information in this rapidly changing field is directly from the manufacturers: advertisements, specification sheets, price lists, etc. Forget books for performance data, look in magazines and your daily

newspaper. Expect that raw computational speed will more than double each two years. Learning about the current state of computer power is simply not enough; you need to understand and track how it is evolving.

Keeping this in mind, we can jump into an overview of how execution speed is limited by computer hardware. Since computers are composed of many subsystems, the time required to execute a particular task will depend on two primary factors: (1) the speed of the individual subsystems, and (2) the time it takes to transfer data between these blocks. Figure 4-5 shows a simplified diagram of the most important speed limiting components in a typical personnel computer. The **Central Processing Unit (CPU)** is the heart of the system. As previously described, it consists of a dozen or so registers, each capable of holding 32 bits (in present generation personnel computers). Also included in the CPU is the digital electronics needed for rudimentary operations, such as moving bits around and fixed point arithmetic.

More involved mathematics is handled by transferring the data to a special hardware circuit called a **math coprocessor** (also called an **arithmetic logic unit**, or **ALU**). The math coprocessor may be contained in the same chip as the CPU, or it may be a separate electronic device. For example, the addition of two floating point numbers would require the CPU to transfer 8 bytes (4 for each number) to the math coprocessor, and several bytes that describe what to do with the data. After a short computational time, the math coprocessor would pass four bytes back to the CPU, containing the floating point number that is the sum. The most inexpensive computer systems don't have a math coprocessor, or provide it only as an option. For example, the 80486DX microprocessor has an internal math coprocessor, while the 80486SX does not. These lower performance systems replace *hardware* with *software*. Each of the mathematical functions is broken into
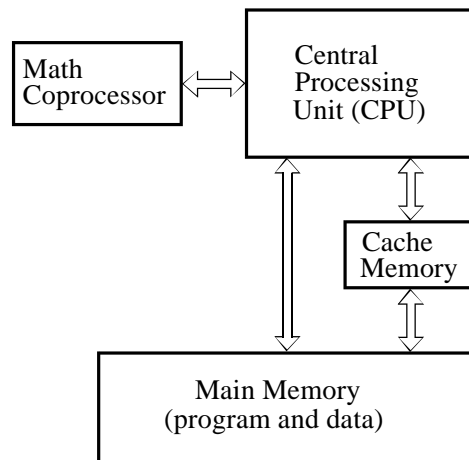
FIGURE 4-5
Architecture of a typical computer system.
The computational speed is limited by: (1)
the speed of the individual subsystems, and
(2) the rate at which data can be transferred
between these subsystems.

elementary binary operations that can be handled directly within the CPU. While this provides the same result, the execution time is much slower, say, a factor of 10 to 20.

Most personal computer software can be used with or without a math coprocessor. This is accomplished by having the compiler generate machine code to handle both cases, all stored in the final executable program. If a math coprocessor is present on the particular computer being used, one section of the code will be run. If a math coprocessor is not present, the other section of the code will be used. The compiler can also be directed to generate code for only one of these situations. For example, you will occasionally find a program that requires that a math coprocessor be present, and will crash if run on a computer that does not have one. Applications such as word processing usually do not benefit from a math coprocessor. This is because they involve moving data around in memory, not the calculation of mathematical expressions. Likewise, calculations involving fixed point variables (integers) are unaffected by the presence of a math coprocessor, since they are handled within the CPU. On the other hand, the execution speed of DSP and other computational programs using floating point calculations can be an order of magnitude different with and without a math coprocessor.

The CPU and main memory are contained in separate chips in most computer systems. For obvious reasons, you would like the main memory to be very large and very fast. Unfortunately, this makes the memory very expensive. The transfer of data between the main memory and the CPU is a very common bottleneck for speed. The CPU *asks* the main memory for the binary information at a particular memory address, and then must *wait* to receive the information. A common technique to get around this problem is to use a **memory cache**. This is a small amount of very fast memory used as a buffer between the CPU and the main memory. A few hundred kilobytes is typical. When the CPU requests the main memory to provide the binary data at a particular address, high speed digital electronics copies a *section* of the main memory around this address into the memory cache. The next time that the CPU requests memory information, it is very likely that it will already be contained in the memory cache, making the retrieval very rapid. This is based on the fact that programs tend to access memory locations that are nearby neighbors of previously accessed data. In typical personnel computer applications, the addition of a memory cache can improve the overall speed by several times. The memory cache may be in the same chip as the CPU, or it may be an external electronic device.

The rate at which data can be transferred between subsystems depends on the number of parallel data lines provided, and the maximum rate that digital signals that can be passed along each line. Digital data can generally be transferred at a much higher rate within a single chip as compared to transferring data between chips. Likewise, data paths that must pass through electrical connectors to other printed circuit boards (i.e., a bus structure) will be slower still. This is a strong motivation for stuffing as much electronics as possible inside the CPU.

A particularly nasty problem for computer speed is *backward compatibility*. When a computer company introduces a new product, say a data acquisition card or a software program, they want to sell it into the largest possible market. This means that it must be compatible with most of the computers currently in use, which could span several generations of technology. This frequently limits the performance of the hardware or software to that of a much older system. For example, suppose you buy an I/O card that plugs into the bus of your 200 MHz Pentium personal computer, providing you with eight digital lines that can transmit and receive data one byte at a time. You then write an assembly program to rapidly transfer data between your computer and some external device, such as a scientific experiment or another computer. Much to your surprise, the maximum data transfer rate is only about 100,000 bytes per second, more than *one thousand* times slower than the microprocessor clock rate! The villain is the ISA bus, a technology that is *backward compatible* to the computers of the early 1980s.

Table 4-6 provides execution times for several generations of computers. Obviously, you should treat these as very rough approximations. If you want to understand *your* system, take measurements on *your* system. It's quite easy; write a loop that executes a *million* of some operation, and use your watch to time how long it takes. The first three systems, the 80286, 80486, and Pentium, are the standard desk-top personal computers of 1986, 1993 and 1996, respectively. The fourth is a 1994 microprocessor designed especially for DSP tasks, the Texas Instruments TMS320C40.

	80286 (12 MHz)	80486 (33 MHz)	PENTIUM (100 MHz)	TMS320C40 (40 MHz)
INTEGER				
A% = B%+C%	1.6	0.12	0.04	
A% = B%-C%	1.6	0.12	0.04	
A% = B%×C%	2.7	0.59	0.13	
A% = B%÷C%	64	9.2	1.5	
FLOATING POINT				
A = B+C	33	2.5	0.50	0.10
A = B-C	35	2.5	0.50	0.10
A = B×C	35	2.5	0.50	0.10
A = B÷C	49	4.5	0.87	0.80
A = SQR(B)	45	5.3	1.3	0.90
A = LOG(B)	186	19	3.4	1.7
A = EXP(B)	246	25	5.5	1.7
A = B^C	311	31	5.3	2.4
A = SIN(B)	262	30	6.6	1.1
A = ARCTAN(B)	168	21	4.4	2.2

TABLE 4-6
Measured execution times for various computers. Times are in microseconds. The 80286, 80486, and Pentium are three generations of personal computers, while the TMS320C40 is a micro-processor specifically designed for DSP tasks. All of the personal computers include a math coprocessor. Use these times only as a general estimate; times on your computer will vary according to the particular hardware and software used.

The Pentium is faster than the 80286 system for four reasons, (1) the greater clock speed, (2) more lines in the data bus, (3) the addition of a memory cache, and (4) a more efficient internal design, requiring fewer clock cycles per instruction.

If the Pentium was a Cadillac, the TMS320C40 would be a Ferrari: less comfort, but blinding speed. This chip is representative of several micro-processors specifically designed to decrease the execution time of DSP algorithms. Others in this category are the Intel i860, AT&T DSP3210, Motorola DSP96002, and the Analog Devices ADSP-2171. These often go by the names: **DSP microprocessor**, **Digital Signal Processor**, and **RISC** (Reduced Instruction Set Computer). This last name reflects that the increased speed results from fewer assembly level instructions being made available to the programmer. In comparison, more traditional microprocessors, such as the Pentium, are called **CISC** (Complex Instruction Set Computer).

DSP microprocessors are used in two ways: as slave modules under the control of a more conventional computer, or as an imbedded processor in a dedicated application, such as a cellular telephone. Some models only handle fixed point numbers, while others can work with floating point. The internal architecture used to obtain the increased speed includes: (1) lots of very fast cache memory contained within the chip, (2) separate buses for the program and data, allowing the two to be accessed simultaneously (called a **Harvard Architecture**), (3) fast hardware for math calculations contained directly in the microprocessor, and (4) a *pipeline* design.

A **pipeline** architecture breaks the *hardware* required for a certain task into several successive stages. For example, the addition of two numbers may be done in three pipeline stages. The first stage of the pipeline does nothing but fetch the numbers to be added from memory. The only task of the second stage is to add the two numbers together. The third stage does nothing but store the result in memory. If each stage can complete its task in a single clock cycle, the entire procedure will take three clock cycles to execute. The key feature of the pipeline structure is that another task can be started before the previous task is completed. In this example, we could begin the addition of *another* two numbers as soon as the first stage is idle, at the end of the first clock cycle. For a large number of operations, the speed of the system will be quoted as one addition per clock cycle, even though the addition of any two numbers requires three clock cycles to complete. Pipelines are great for speed, but they can be difficult to program. The algorithm must allow a new calculation to begin, even though the results of previous calculations are unavailable (because they are still in the pipeline).

Chapters 28 and 29 discuss DSP microprocessors in much more detail. These are amazing devices; their high-power and low-cost will bring DSP to a wide range of consumer and scientific applications. This is one of the technologies that will shape the twenty-first century.

Execution Speed: Programming Tips

While computer hardware and programming languages are important for maximizing execution speed, they are not something you change on a day-to day basis. In comparison, *how you program* can be changed at any time, and will drastically affect how long the program will require to execute. Here are three suggestions.

First, use integers instead of floating point variables whenever possible. Conventional microprocessors, such as used in personal computers, process integers 10 to 20 times faster than floating point numbers. On systems without a math coprocessor, the difference can be 200 to 1. An exception to this is integer division, which is often accomplished by converting the values into floating point. This makes the operation ghastly slow compared to other integer calculations. See Table 4-6 for details.

Second, avoid using functions such as: $\sin(x)$, $\log(x)$, y^x, etc. These transcendental functions are calculated as a series of additions, subtractions and multiplications. For example, the Maclaurin power series provides:

EQUATION 4-3
Maclaurin power series expansion for three transcendental functions. This is how computers calculate functions of this type, and why they execute so slowly.

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \cdots$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots$$

While these relations are infinite in length, the terms rapidly become small enough to be ignored. For example:

$$\sin(1) = 1 - 0.166666 + 0.008333 - 0.000198 + 0.000002 - \cdots$$

These functions require about ten times longer to calculate than a single addition or multiplication (see Table 4-6). Several tricks can be used to bypass these calculations, such as: $x^3 = x \cdot x \cdot x$; $\sin(x) \approx x$, when x is very small; $\sin(-x) = \sin(x)$, where you already know one of the values and need to find the other, etc. Most languages only provide a few transcendental functions, and expect you to derive the others by means of the relations in Table 4-7. Not surprisingly, these derived calculations are even slower.

FUNCTION	EQUATION FOR CALCULATING
Secant (X) =	1/COS(X)
Cosecant (X) =	1/SIN(X)
Cotangent (X) =	1/TAN(X)
Arc Sine (X) =	ATN(X/SQR(1-X*X))
Arc Cosine (X) =	-ATN(X/SQR(1-X*X)) + PI/2
Arc Secant (X) =	ATN(SQR(X*X-1)) + (SGN(X)-1) * PI/2
Arc Cosecant (X) =	ATN(1/SQR(X*X-1)) + (SGN(X)-1) * PI/2
Arc Cotangent (X) =	-ATN(X) + PI/2
Hyperbolic Sine (X) =	(EXP(X)-EXP(-X))/2
Hyperbolic Cosine (X) =	(EXP(X)+EXP(-X))/2
Hyperbolic Tangent (X) =	(EXP(X)-EXP(-X))/(EXP(X)+EXP(-X))
Hyperbolic Secant (X) =	1/HYPERBOLIC COSINE
Hyperbolic Cosecant (X) =	1/HYPERBOLIC SINE
Hyperbolic Cotangent (X) =	1/HYPERBOLIC TANGENT
Arc Hyperbolic Sine (X) =	LOG(X+SQR(X*X+1))
Arc Hyperbolic Cosine (X) =	LOG(X+SQR(X*X-1))
Arc Hyperbolic Tangent (X) =	LOG((1+X) /(1-X))/2
Arc Hyperbolic Secant (X) =	LOG((SQR(1-X*X)+1)/X)
Arc Hyperbolic Cosecant (X) =	LOG(1+SGN(X)*SQR(1+X*X))/X
Arc Hyperbolic Cotangent (X) =	LOG((X+1)/(X-1))/2
$LOG_{10}(X)$ =	LOG(X)/LOG(10) = 0.4342945 LOG(X)
PI =	4*ATN(1) = 3.141592653589794

TABLE 4-7
Calculating rarely used functions from more common ones.   All angles are in radians,
ATN(X) is the arctangent, LOG(X) is the natural logarithm, SGN(X) is the sign of X (i.e.,
-1 for $X \leq 0$, 1 for X>0), EXP(X) is e^x.

Another option is to precalculate these slow functions, and store the values in a **look-up table (LUT)**. For example, imagine an 8 bit data acquisition system used to continually monitor the *voltage* across a resistor. If the parameter of interest is the *power* being dissipated in the resistor, the measured voltage can be used to calculate: $P = V^2/R$. As a faster alternative, the power corresponding to each of the possible 256 voltage measurements can be calculated beforehand, and stored in a LUT. When the system is running, the measured voltage, a digital number between 0 and 255, becomes an index in the LUT to find the corresponding power. Look-up tables can be hundreds of times faster than direct calculation.

Third, learn what is *fast* and what is *slow* on your particular system. This comes with experience and testing, and there will always be surprises. Pay particular attention to *graphics* commands and *I/O*. There are usually several ways to handle these requirements, and the speeds can be tremendously different. For example, the BASIC command: *BLOAD*, transfers a data file directly into a section of memory. Reading the same file into memory byte-by-byte (in a loop) can be 100 times as slow. As another example, the BASIC command: *LINE,* can be used to draw a colored box on the video screen. Drawing the same box pixel-by-pixel can also take 100 times as long. Even putting a *print* statement within a loop (to keep track of what it is doing) can slow the operation by *thousands*!