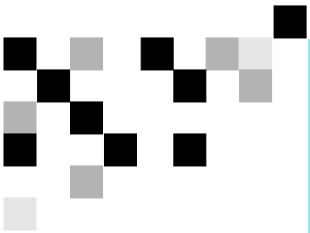


2 DATA REPRESENTATION

2.1 Introduction

In the early days of computing, there were common misconceptions about computers. One misconception was that the computer was only a giant adding machine performing arithmetic operations. Computers could do much more than that, even in the early days. The other common misconception, in contradiction to the first, was that the computer could do "anything." We now know that there are indeed classes of problems that even the most powerful imaginable computer finds intractable with the von Neumann model. The correct perception, of course, is somewhere between the two.

We are familiar with computer operations that are non-arithmetic: computer graphics, digital audio, even the manipulation of the computer mouse. Regardless of what kind of information is being manipulated by the computer, the information must be represented by patterns of 1's and 0's (also known as "on-off" codes). This immediately raises the question of how that information should be described or represented in the machine—this is the **data representation**, or **data encoding**. Graphical images, digital audio, or mouse clicks must all be encoded in a systematic, agreed-upon manner.

We might think of the decimal representation of information as the most natural when we know it the best, but the use of on-off codes to represent information predated the computer by many years, in the form of Morse code.

This chapter introduces several of the simplest and most important encodings: the encoding of signed and unsigned fixed point numbers, real numbers (referred to as **floating point numbers** in computer jargon), and the printing characters. We shall see that in all cases there are multiple ways of encoding a given kind of data, some useful in one context, some in another. We will also take an early look

at computer arithmetic for the purpose of understanding some of the encoding schemes, though we will defer details of computer arithmetic until Chapter 3.

In the process of developing a data representation for computing, a crucial issue is deciding how much storage should be devoted to each data value. For example, a computer architect may decide to treat integers as being 32 bits in size, and to implement an ALU that supports arithmetic operations on those 32-bit values that return 32 bit results. Some numbers can be too large to represent using 32 bits, however, and in other cases, the operands may fit into 32 bits, but the result of a computation will not, creating an **overflow** condition, which is described in Chapter 3. Thus we need to understand the limits imposed on the accuracy and range of numeric calculations by the finite nature of the data representations. We will investigate these limits in the next few sections.

2.2 Fixed Point Numbers

In a fixed point number system, each number has exactly the same number of digits, and the "point" is always in the same place. Examples from the decimal number system would be 0.23, 5.12, and 9.11. In these examples each number has 3 digits, and the decimal point is located two places from the right. Examples from the **binary** number system (in which each digit can take on only one of the values: 0 or 1) would be 11.10, 01.10, and 00.11, where there are 4 binary digits and the binary point is in the middle. An important difference between the way that we represent fixed point numbers on paper and the way that we represent them in the computer is that when fixed point numbers are represented in the computer *the binary point is not stored anywhere*, but only assumed to be in a certain position. One could say that the binary point exists only in the mind of the programmer.

We begin coverage of fixed point numbers by investigating the range and precision of fixed point numbers, using the decimal number system. We then take a look at the nature of number bases, such as decimal and binary, and how to convert between the bases. With this foundation, we then investigate several ways of representing negative fixed point numbers, and take a look at simple arithmetic operations that can be performed on them.

2.2.1 *RANGE AND PRECISION IN FIXED POINT NUMBERS*

A fixed point representation can be characterized by the **range** of expressible numbers (that is, the distance between the largest and smallest numbers) and the

precision (the distance between two adjacent numbers on a number line.) For the fixed-point decimal example above, using three digits and the decimal point placed two digits from the right, the range is from 0.00 to 9.99 inclusive of the endpoints, denoted as [0.00, 9.99], the precision is .01, and the **error** is 1/2 of the difference between two "adjoining" numbers, such as 5.01 and 5.02, which have a difference of .01. The error is thus .01/2 = .005. That is, we can represent any number within the range 0.00 to 9.99 to within .005 of its true or precise value.

Notice how range and precision trade off: with the decimal point on the far right, the range is [000, 999] and the precision is 1.0. With the decimal point at the far left, the range is [.000, .999] and the precision is .001.

In either case, there are only 10^3 different decimal "objects," ranging from 000 to 999 or from .000 to .999, and thus it is possible to represent only 1,000 different items, regardless of how we apportion range and precision.

There is no reason why the range must begin with 0. A 2-digit decimal number can have a range of [00,99] or a range of [-50, +49], or even a range of [-99, +0]. The representation of negative numbers is covered more fully in Section 2.2.6.

Range and precision are important issues in computer architecture because both are finite in the implementation of the architecture, but are infinite in the real world, and so the user must be aware of the limitations of trying to represent external information in internal form.

2.2.2 *THE ASSOCIATIVE LAW OF ALGEBRA DOES NOT ALWAYS HOLD IN COMPUTERS*

In early mathematics, we learned the associative law of algebra:

$$a + (b + c) = (a + b) + c$$

As we will see, the associative law of algebra does not hold for fixed point numbers having a finite representation. Consider a 1-digit decimal fixed point representation with the decimal point on the right, and a range of [-9, 9], with a = 7, b=4, and c=–3. Now $a + (b + c) = 7 + (4 + -3) = 7 + 1 = 8$. But $(a + b) + c = (7 + 4) + -3 = 11 + -3$, but 11 is outside the range of our number system! We have overflow in an intermediate calculation, but the final result is within the number system. This is every bit as bad because the final result will be wrong if an inter-

mediate result is wrong.

Thus we can see by example that the associative law of algebra does not hold for finite-length fixed point numbers. This is an unavoidable consequence of this form of representation, and there is nothing practical to be done except to detect overflow wherever it occurs, and either terminate the computation immediately and notify the user of the condition, or, having detected the overflow, repeat the computation with numbers of greater range. (The latter technique is seldom used except in critical applications.)

2.2.3 *RADIX NUMBER SYSTEMS*

In this section, we learn how to work with numbers having arbitrary bases, although we will focus on the bases most used in digital computers, such as base 2 (binary), and its close cousins base 8 (octal), and base 16 (hexadecimal.)

The **base**, or **radix** of a number system defines the range of possible values that a digit may have. In the base 10 (decimal) number system, one of the 10 values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 is used for each digit of a number. The most natural system for representing numbers in a computer is base 2, in which data is represented as a collection of 1's and 0's.

The general form for determining the decimal value of a number in a radix k fixed point number system is shown below:

$$Value = \sum_{i=-m}^{n-1} b_i \cdot k^i$$

The value of the digit in position i is given by b_i. There are n digits to the left of the radix point and there are m digits to the right of the radix point. This form of a number, in which each position has an assigned weight, is referred to as a **weighted position code**. Consider evaluating $(541.25)_{10}$, in which the subscript 10 represents the base. We have $n = 3$, $m = 2$, and $k = 10$:

$$5 \times 10^2 + 4 \times 10^1 + 1 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} =$$

$$(500)_{10} + (40)_{10} + (1)_{10} + (2/10)_{10} + (5/100)_{10} \quad = (541.25)_{10}$$

Now consider the base 2 number $(1010.01)_2$ in which $n = 4$, $m = 2$, and $k = 2$:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} \quad =$$

$$(8)_{10} + (0)_{10} + (2)_{10} + (0)_{10} + (0/2)_{10} + (1/4)_{10} \quad = (10.25)_{10}$$

This suggests how to convert a number from an arbitrary base into a base 10 number using the **polynomial method**. The idea is to multiply each digit by the weight assigned to its position (powers of two in this example) and then sum up the terms to obtain the converted number. Although conversions can be made among all of the bases in this way, some bases pose special problems, as we will see in the next section.

Note: in these weighted number systems we define the bit that carries the most weight as the **most significant bit (MSB)**, and the bit that carries the least weight as the **least significant bit (LSB)**. Conventionally the MSB is the leftmost bit and the LSB the rightmost bit.

2.2.4 *CONVERSIONS AMONG RADICES*

In the previous section, we saw an example of how a base 2 number can be converted into a base 10 number. A conversion in the reverse direction is more involved. The easiest way to convert fixed point numbers containing both integer and fractional parts is to convert each part separately. Consider converting $(23.375)_{10}$ to base 2. We begin by separating the number into its integer and fractional parts:

$$(23.375)_{10} = (23)_{10} + (.375)_{10}.$$

Converting the Integer Part of a Fixed Point Number—The Remainder Method

As suggested in the previous section, the general polynomial form for representing a binary integer is:

$$b_i \times 2^i + b_{i-1} \times 2^{i-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

If we divide the integer by 2, then we will obtain:

$$b_i \times 2^{i-1} + b_{i-1} \times 2^{i-2} + \ldots + b_1 \times 2^0$$

with a remainder of b_0. As a result of dividing the original integer by 2, we discover the value of the first binary coefficient b_0. We can repeat this process on the remaining polynomial and determine the value of b_1. We can continue iterating the process on the remaining polynomial and thus obtain all of the b_i. This process forms the basis of the **remainder method** of converting integers between bases.

We now apply the remainder method to convert $(23)_{10}$ to base 2. As shown in Figure 2-1, the integer is initially divided by 2, which leaves a remainder of 0 or

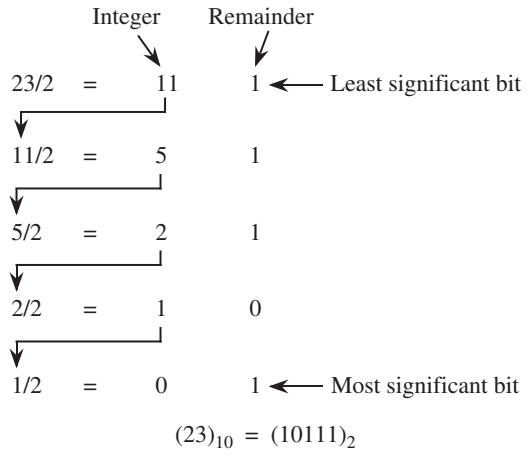


Figure 2-1 A conversion from a base 10 integer to a base 2 integer using the remainder method.

1. For this case, 23/2 produces a quotient of 11 and a remainder of 1. The first remainder is the least significant **b**inary dig**it** (**bit**) of the converted number (the rightmost bit). In the next step 11 is divided by 2, which creates a quotient of 5 and a remainder of 1. Next, 5 is divided by 2, which creates a quotient of 2 and a remainder of 1. The process continues until we are left with a quotient of 0. If we continue the process after obtaining a quotient of 0, we will only obtain 0's for the quotient and remainder, which will not change the value of the converted number. The remainders are collected into a base 2 number in the order shown in Figure 2-1 to produce the result $(23)_{10} = (10111)_2$. In general, we can convert any base 10 integer to any other base by simply dividing the integer by the base to which we are converting.

We can check the result by converting it from base 2 back to base 10 using the polynomial method:

$$(10111)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 16 + 0 + 4 + 2 + 1$$

$$= (23)_{10}$$

At this point, we have converted the integer portion of $(23.375)_{10}$ into base 2.

Converting the Fractional Part of a Fixed Point Number—The Multiplication Method

The conversion of the fractional portion can be accomplished by successively multiplying the fraction by 2 as described below.

A binary fraction is represented in the general form:

$$b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + \ldots$$

If we multiply the fraction by 2, then we will obtain:

$$b_{-1} + b_{-2} \times 2^{-1} + b_{-3} \times 2^{-2} + \ldots$$

We thus discover the coefficient b_{-1}. If we iterate this process on the remaining fraction, then we will obtain successive b_i. This process forms the basis of the **multiplication method** of converting fractions between bases. For the example used here (Figure 2-2), the initial fraction $(.375)_{10}$ is less than 1. If we multiply it by 2, then the resulting number will be less than 2. The digit to the left of the radix point will then be 0 or 1. This is the first digit to the right of the radix point in the converted base 2 number, as shown in the figure. We repeat the process on the fractional portion until we are either left with a fraction of 0, at which point only trailing 0's are created by additional iterations, or we have reached the limit of precision used in our representation. The digits are collected and the result is obtained: $(.375)_{10} = (.011)_2$.

For this process, the multiplier is the same as the target base. The multiplier is 2 here, but if we wanted to make a conversion to another base, such as 3, then we

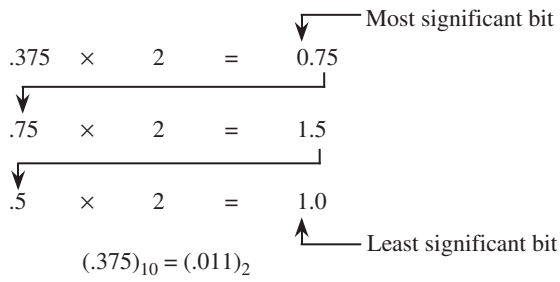


Figure 2-2 A conversion from a base 10 fraction to a base 2 fraction using the multiplication method.

would use a multiplier of 3.[1]

We again check the result of the conversion by converting from base 2 back to base 10 using the polynomial method as shown below:

$$(.011)_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 0 + 1/4 + 1/8 = (.375)_{10}.$$

We now combine the integer and fractional portions of the number and obtain the final result:

$$(23.375)_{10} = (10111.011)_2.$$

Non Terminating Fractions

Although this method of conversion will work among all bases, some precision can be lost in the process. For example, not all terminating base 10 fractions have a terminating base 2 form. Consider converting $(.2)_{10}$ to base 2 as shown in Figure 2-3. In the last row of the conversion, the fraction .2 reappears, and the process repeats *ad infinitum*. As to why this can happen, consider that any non-repeating base 2 fraction can be represented as $i/2^k$ for some integers i and k. (Repeating fractions in base 2 cannot be so represented.) Algebraically,

$$i/2^k = i \times 5^k/(2^k \times 5^k) = i \times 5^k/10^k = j/10^k$$

1. Alternatively, we can use the base 10 number system and also avoid the conversion if we retain a base 2 representation, in which combinations of 1's and 0's represent the base 10 digits. This is known as binary coded decimal (BCD), which we will explore later in the chapter.

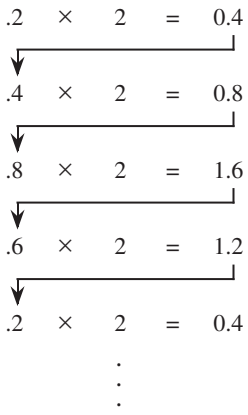


Figure 2-3 A terminating base 10 fraction that does not have a terminating base 2 form.

where j is the integer $i \times 5^k$. The fraction is thus non-repeating in base 10. This hinges on the fact that only non-repeating fractions in base b can be represented as i/b^k for some integers i and k. The condition that must be satisfied for a non-repeating base 10 fraction to have an equivalent non-repeating base 2 fraction is:

$$i/10^k = i/(5^k \times 2^k) = j/2^k$$

where $j = i/5^k$, and 5^k must be a factor of i. For one digit decimal fractions, only $(.0)_{10}$ and $(.5)_{10}$ are non-repeating in base 2 (20% of the possible fractions); for two digit decimal fractions, only $(.00)_{10}$, $(.25)_{10}$, $(.50)_{10}$, and $(.75)_{10}$ are non-repeating (4% of the possible fractions); *etc.* There is a link between relatively prime numbers and repeating fractions, which is helpful in understanding why some terminating base 10 fractions do not have a terminating base 2 form. (Knuth, 1981) provides some insight in this area.

Binary versus Decimal Representations

While most computers use base 2 for internal representation and arithmetic, some calculators and business computers use an internal representation of base 10, and thus do not suffer from this representational problem. The motivation for using base 10 in business computers is not entirely to avoid the terminating fraction problem, however, but also to avoid the conversion process at the input and output units which historically have taken a significant amount of time.

Binary, Octal, and Hexadecimal Radix Representations

While binary numbers reflect the actual internal representation used in most machines, they suffer from the disadvantage that numbers represented in base 2 tend to need more digits than numbers in other bases, (why?) and it is easy to make errors when writing them because of the long strings of 1's and 0's. We mentioned earlier in the Chapter that base 8, **octal radix**, and base 16, **hexadecimal radix**, are related to base 2. This is due to the three radices all being divisible by 2, the smallest one. We show below that converting among the three bases 2, 8, and 16 is trivial, and there are significant practical advantages to representing numbers in these bases.

Binary numbers may be considerably wider than their base 10 equivalents. As a notational convenience, we sometimes use larger bases than 2 that are even multiples of 2. Converting among bases 2, 8, or 16 is easier than converting to and from base 10. The values used for the base 8 digits are familiar to us as base 10 digits, but for base 16 (hexadecimal) we need six more digits than are used in base 10. The letters A, B, C, D, E, F or their lower-case equivalents are commonly used to represent the corresponding values (10, 11, 12, 13, 14, 15) in hexadecimal. The digits commonly used for bases 2, 8, 10, and 16 are summarized in Figure 2-4. In comparing the base 2 column with the base 8 and base 16

Binary (base 2)	Octal (base 8)	Decimal (base 10)	Hexadecimal (base 16)
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Figure 2-4 Values for digits in the binary, octal, decimal, and hexadecimal number systems.

columns, we need three bits to represent each base 8 digit in binary, and we need four bits to represent each base 16 digit in binary. In general, *k* bits are needed to

represent each digit in base 2^k, in which k is an integer, so base $2^3 = 8$ uses three bits and base $2^4 = 16$ uses four bits.

In order to convert a base 2 number into a base 8 number, we partition the base 2 number into groups of three starting from the radix point, and pad the outermost groups with 0's as needed to form triples. Then, we convert each triple to the octal equivalent. For conversion from base 2 to base 16, we use groups of four. Consider converting $(10110)_2$ to base 8:

$$(10110)_2 = (010)_2\ (110)_2 = (2)_8\ (6)_8 = (26)_8$$

Notice that the leftmost two bits are padded with a 0 on the left in order to create a full triplet.

Now consider converting $(10110110)_2$ to base 16:

$$(10110110)_2 = (1011)_2\ (0110)_2 = (B)_{16}\ (6)_{16} = (B6)_{16}$$

(Note that 'B' is a base 16 digit corresponding to 11_{10}. B is not a variable.)

The conversion methods can be used to convert a number from any base to any other base, but it may not be very intuitive to convert something like $(513.03)_6$ to base 7. As an aid in performing an unnatural conversion, we can convert to the more familiar base 10 form as an intermediate step, and then continue the conversion from base 10 to the target base. As a general rule, we use the polynomial method when converting *into* base 10, and we use the remainder and multiplication methods when converting *out* of base 10.

2.2.5 AN EARLY LOOK AT COMPUTER ARITHMETIC

We will explore computer arithmetic in detail in Chapter 3, but for the moment, we need to learn how to perform simple binary addition because it is used in representing signed binary numbers. Binary addition is performed similar to the way we perform decimal addition by hand, as illustrated in Figure 2-5. Two binary numbers *A* and *B* are added from right to left, creating a sum and a carry in each bit position. Since the rightmost bits of *A* and *B* can each assume one of two values, four cases must be considered: 0 + 0, 0 + 1, 1 + 0, and 1 + 1, with a carry of 0, as shown in the figure. The carry into the rightmost bit position defaults to 0. For the remaining bit positions, the carry into the position can be 0

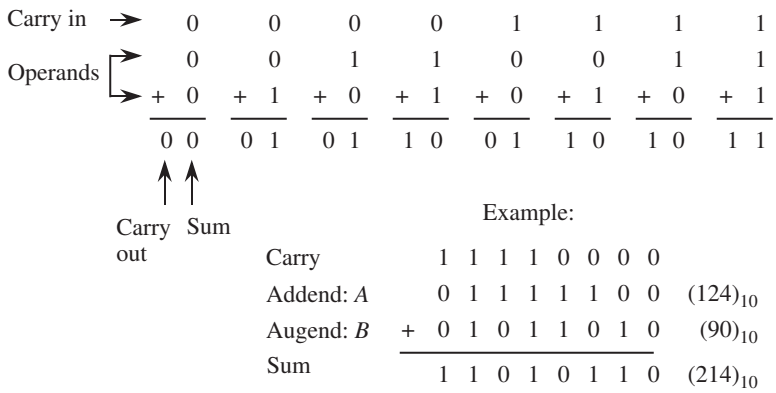


Figure 2-5 Example of binary addition.

or 1, so that a total of eight input combinations must be considered as shown in the figure.

Notice that the largest number we can represent using the eight-bit format shown in Figure 2-5 is $(11111111)_2 = (255)_{10}$ and that the smallest number that can be represented is $(00000000)_2 = (0)_{10}$. The bit patterns 11111111 and 00000000 and all of the intermediate bit patterns represent numbers on the closed interval from 0 to 255, which are all positive numbers. Up to this point we have considered only unsigned numbers, but we need to represent signed numbers as well, in which (approximately) one half of the bit patterns is assigned to positive numbers and the other half is assigned to negative numbers. Four common representations for base 2 signed numbers are discussed in the next section.

2.2.6 *SIGNED FIXED POINT NUMBERS*

Up to this point we have considered only the representation of unsigned fixed point numbers. The situation is quite different in representing *signed* fixed point numbers. There are four different ways of representing signed numbers that are commonly used: sign-magnitude, one's complement, two's complement, and excess notation. We will cover each in turn, using integers for our examples. Throughout the discussion, the reader may wish to refer to Table 2.1 which shows for a 3-bit number how the various representations appear.

Decimal	Unsigned	Sign–Mag.	1's Comp.	2's Comp.	Excess 4
7	111	–	–	–	–
6	110	–	–	–	–
5	101	–	–	–	–
4	100	–	–	–	–
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
+0	000	000	000	000	100
-0	–	100	111	000	100
-1	–	101	110	111	011
-2	–	110	101	110	010
-3	–	111	100	101	001
-4	–	–	–	100	000

Table 2.1: 3-bit Integer Representations

Signed Magnitude

The **signed magnitude** (also referred to as **sign and magnitude**) representation is most familiar to us as the base 10 number system. A plus or minus sign to the left of a number indicates whether the number is positive or negative as in $+12_{10}$ or -12_{10}. In the binary signed magnitude representation, the leftmost bit is used for the sign, which takes on a value of 0 or 1 for '+' or '–', respectively. The remaining bits contain the absolute magnitude. Consider representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format:

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (10001100)_2$$

The negative number is formed by simply changing the sign bit in the positive number from 0 to 1. Notice that there are both positive and negative representations for zero: 00000000 and 10000000.

There are eight bits in this example format, and all bit patterns represent valid numbers, so there are $2^8 = 256$ possible patterns. Only $2^8 - 1 = 255$ different numbers can be represented, however, since +0 and −0 represent the same number.

We will make use of the signed magnitude representation when we look at floating point numbers in Section 2.3.

One's Complement

The **one's complement** operation is trivial to perform: convert all of the 1's in the number to 0's, and all of the 0's to 1's. See the fourth column in Table 2.1 for examples. We can observe from the table that in the **one's complement** representation the leftmost bit is 0 for positive numbers and 1 for negative numbers, as it is for the signed magnitude representation. This negation, changing 1's to 0's and changing 0's to 1's, is known as **complementing** the bits. Consider again representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format, now using the one's complement representation:

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (11110011)_2$$

Note again that there are representations for both +0 and −0, which are 00000000 and 11111111, respectively. As a result, there are only $2^8 - 1 = 255$ different numbers that can be represented even though there are 2^8 different bit patterns.

The one's complement representation is not commonly used. This is at least partly due to the difficulty in making comparisons when there are two representations for 0. There is also additional complexity involved in adding numbers, which is discussed further in Chapter 3.

Two's Complement

The two's complement is formed in a way similar to forming the one's complement: complement all of the bits in the number, but then add 1, and if that addition results in a carry-out from the most significant bit of the number, discard the carry-out. Examination of the fifth column of Table 2.1 shows that in the

two's complement representation, the leftmost bit is again 0 for positive numbers and is 1 for negative numbers. However, this number format does not have the unfortunate characteristic of signed-magnitude and one's complement representations: it has only one representation for zero. To see that this is true, consider forming the negative of $(+0)_{10}$, which has the bit pattern:

$$(+0)_{10} = (00000000)_2$$

Forming the one's complement of $(00000000)_2$ produces $(11111111)_2$ and adding 1 to it yields $(00000000)_2$, thus $(-0)_{10} = (00000000)_2$. The carry out of the leftmost position is discarded in two's complement addition (except when detecting an overflow condition). Since there is only one representation for 0, and since all bit patterns are valid, there are $2^8 = 256$ different numbers that can be represented.

Consider again representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format, this time using the two's complement representation. Starting with $(+12)_{10} = (00001100)_2$, complement, or negate the number, producing $(11110011)_2$. Now add 1, producing $(11110100)_2$, and thus $(-12)_{10} = (11110100)_2$:

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (11110100)_2$$

There is an equal number of positive and negative numbers provided zero is considered to be a positive number, which is reasonable because its sign bit is 0. The positive numbers start at 0, but the negative numbers start at −1, and so the magnitude of the most negative number is one greater than the magnitude of the most positive number. The positive number with the largest magnitude is +127, and the negative number with the largest magnitude is −128. There is thus no positive number that can be represented that corresponds to the negative of −128. If we try to form the two's complement negative of −128, then we will arrive at a negative number, as shown below:

$$\begin{array}{rl} (-128)_{10} = (10000000)_2 & \\ \Downarrow & \\ 01111111 & \\ + \quad 1 & \\ \hline (10000000)_2 & \end{array}$$

The two's complement representation is the representation most commonly used in conventional computers, and we will use it throughout the book.

Excess Representation

In the **excess** or **biased** representation, the number is treated as unsigned, but is "shifted" in value by subtracting the bias from it. The concept is to assign the smallest numerical bit pattern, all zeros, to the negative of the bias, and assign the remaining numbers in sequence as the bit patterns increase in magnitude. A convenient way to think of an excess representation is that a number is represented as the sum of its two's complement form and another number, which is known as the "excess," or "bias." Once again, refer to Table 2.1, the rightmost column, for examples.

Consider again representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format but now using an excess 128 representation. An excess 128 number is formed by adding 128 to the original number, and then creating the unsigned binary version. For $(+12)_{10}$, we compute $(128 + 12 = 140)_{10}$ and produce the bit pattern $(10001100)_2$. For $(-12)_{10}$, we compute $(128 + -12 = 116)_{10}$ and produce the bit pattern $(01110100)_2$:

$$(+12)_{10} = (10001100)_2$$

$$(-12)_{10} = (01110100)_2$$

Note that there is no numerical significance to the excess value: it simply has the effect of shifting the representation of the two's complement numbers.

There is only one excess representation for 0, since the excess representation is simply a shifted version of the two's complement representation. For the previous case, the excess value is chosen to have the same bit pattern as the largest negative number, which has the effect of making the numbers appear in numerically sorted order if the numbers are viewed in an unsigned binary representation. Thus, the most negative number is $(-128)_{10} = (00000000)_2$ and the most positive number is $(+127)_{10} = (11111111)_2$. This representation simplifies making comparisons between numbers, since the bit patterns for negative numbers have numerically smaller values than the bit patterns for positive numbers. This is important for representing the exponents of floating point numbers, in which exponents of two numbers are compared in order to make them equal for addi-

tion and subtraction. We will explore floating point representations in Section 2.3.

2.2.7 *BINARY CODED DECIMAL*

Numbers can be represented in the base 10 number system while still using a binary encoding. Each base 10 digit occupies four bit positions, which is known as **binary coded decimal (BCD)**. Each BCD digit can take on any of 10 values. There are $2^4 = 16$ possible bit patterns for each base 10 digit, and as a result, six bit patterns are unused for each digit. In Figure 2-6, there are four decimal signif-

(a)	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 1	$(+301)_{10}$	Nine's and ten's complement
	$(0)_{10}$	$(3)_{10}$	$(0)_{10}$	$(1)_{10}$		
(b)	1 0 0 1	0 1 1 0	1 0 0 1	1 0 0 0	$(-301)_{10}$	Nine's complement
	$(9)_{10}$	$(6)_{10}$	$(9)_{10}$	$(8)_{10}$		
(c)	1 0 0 1	0 1 1 0	1 0 0 1	1 0 0 1	$(-301)_{10}$	Ten's complement
	$(9)_{10}$	$(6)_{10}$	$(9)_{10}$	$(9)_{10}$		

Figure 2-6 BCD representations of 301 (a) and –301 in nine's complement (b) and ten's complement (c).

icant digits, so $10^4 = 10{,}000$ bit patterns are valid, even though $2^{16} = 65{,}536$ bit patterns are possible with 16 bits.

Although some bit patterns are unused, the BCD format is commonly used in calculators and in business applications. There are fewer problems in representing terminating base 10 fractions in this format, unlike the base 2 representation. There is no need to convert data that is given at the input in base 10 form (as in a calculator) into an internal base 2 representation, or to convert from an internal representation of base 2 to an output form of base 10.

Performing arithmetic on signed BCD numbers may not be obvious. Although we are accustomed to using a signed magnitude representation in base 10, a different method of representing signed base 10 numbers is used in a computer. In the **nine's complement** number system, positive numbers are represented as in ordinary BCD, but the leftmost digit is less than 5 for positive numbers and is 5 or greater for negative numbers. The nine's complement negative is formed by subtracting each digit from 9. For example, the base 10 number +301 is represented as 0301 (or simply 301) in both nine's and ten's complement as shown in

Figure 2-6a. The nine's complement negative is 9698 (Figure 2-6b), which is obtained by subtracting each digit in 0301 from 9.

The **ten's complement** negative is formed by adding 1 to the nine's complement negative, so the ten's complement representation of −301 is then 9698 + 1 = 9699 as shown in Figure 2-6c. For this example, the positive numbers range from 0 – 4999 and the negative numbers range from 5000 to 9999.

2.3 Floating Point Numbers

The fixed point number representation, which we explored in Section 2.2, has a fixed position for the radix point, and a fixed number of digits to the left and right of the radix point. A fixed point representation may need a great many digits in order to represent a practical range of numbers. For example, a computer that can represent a number as large as a trillion[1] maintains at least 40 bits to the left of the radix point since $2^{40} \approx 10^{12}$. If the same computer needs to represent one trillionth, then 40 bits must also be maintained to the right of the radix point, which results in a total of 80 bits per number.

In practice, much larger numbers and much smaller numbers appear during the course of computation, which places even greater demands on a computer. A great deal of hardware is required in order to store and manipulate numbers with 80 or more bits of precision, and computation proceeds more slowly for a large number of digits than for a small number of digits. Fine precision, however, is generally not needed when large numbers are used, and conversely, large numbers do not generally need to be represented when calculations are made with small numbers. A more efficient computer can be realized when only as much precision is retained as is needed.

2.3.1 *RANGE AND PRECISION IN FLOATING POINT NUMBERS*

A **floating point** representation allows a large range of expressible numbers to be represented in a small number of digits by separating the digits used for *precision* from the digits used for *range*. The base 10 floating point number representing Avogadro's number is shown below:

---

1. In the American number system, which is used here, a trillion = 10^{12}. In the British number system, this is a "million million," or simply a "billion." The British "milliard," or a "thousand million" is what Americans call a "billion."

$$+6.023 \times 10^{23}$$

Here, the range is represented by a power of 10, 10^{23} in this case, and the precision is represented by the digits in the fixed point number, 6.023 in this case. In discussing floating point numbers, the fixed point part is often referred to as the **mantissa**, or **significand** of the number. Thus a floating point number can be characterized by a triple of numbers: sign, exponent, and significand.

The range is determined primarily by the number of digits in the exponent (two digits are used here) and the base to which it is raised (base 10 is used here) and the precision is determined primarily by the number of digits in the significand (four digits are used here). Thus the entire number can be represented by a sign and 6 digits, two for the exponent and four for the significand. Figure 2-7 shows how the triple of sign, exponent, significand, might be formatted in a computer.

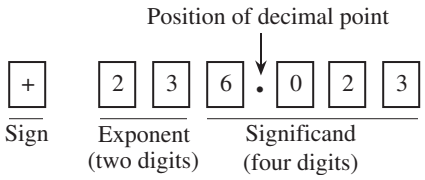


Figure 2-7 Representation of a base 10 floating point number.

Notice how the digits are packed together with the sign first, followed by the exponent, followed by the significand. This ordering will turn out to be helpful in comparing two floating point numbers. The reader should be aware that the decimal point does not need to be stored with the number as long as the decimal point is always in the same position in the significand. (This will be discussed in Section 2.3.2.)

If we need a greater range, and if we are willing to sacrifice precision, then we can use just three digits in the fraction and have three digits left for the exponent without increasing the number of digits used in the representation. An alternative method of increasing the range is to increase the base, which has the effect of increasing the precision of the smallest numbers but decreasing the precision of the largest numbers. The range/precision trade-off is a major advantage of using a floating point representation, but the reduced precision can cause problems, sometimes leading to disaster, an example of which is described in Section 2.4.

2.3.2 NORMALIZATION, AND THE HIDDEN BIT

A potential problem with representing floating point numbers is that the same number can be represented in different ways, which makes comparisons and arithmetic operations difficult. For example, consider the numerically equivalent forms shown below:

$$3584.1 \times 10^0 = 3.5841 \times 10^3 = .35841 \times 10^4.$$

In order to avoid multiple representations for the same number, floating point numbers are maintained in **normalized** form. That is, the radix point is shifted to the left or to the right and the exponent is adjusted accordingly until the radix point is to the left of the leftmost nonzero digit. So the rightmost number above is the normalized one. Unfortunately, the number zero cannot be represented in this scheme, so to represent zero an exception is made. The exception to this rule is that zero is represented as all 0's in the mantissa.

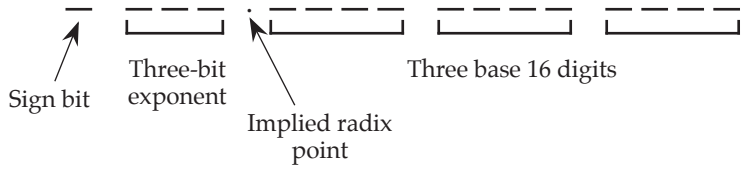
If the mantissa is represented as a binary, that is, base 2, number, and if the normalization condition is that there is a leading "1" in the normalized mantissa, then there is no need to store that "1" and in fact, most floating point formats do *not* store it. Rather, it is "chopped off" before packing up the number for storage, and it is restored when unpacking the number into exponent and mantissa. This results in having an additional bit of precision on the right of the number, due to removing the bit on the left. This missing bit is referred to as the **hidden bit**, also known as a **hidden 1**. For example, if the mantissa in a given format is .11010 after normalization, then the bit pattern that is stored is 1010—the left-most bit is truncated, or hidden. We will see that the IEEE 754 floating point format uses a hidden bit.

2.3.3 REPRESENTING FLOATING POINT NUMBERS IN THE COMPUTER—PRELIMINARIES

Let us design a simple floating point format to illustrate the important factors in representing floating point numbers on the computer. Our format may at first seem to be unnecessarily complex. We will represent the significand in signed magnitude format, with a single bit for the sign bit, and three hexadecimal digits for the magnitude. The exponent will be a 3-bit excess-4 number, with a radix of 16. The normalized form of the number has the hexadecimal point to the left of the three hexadecimal digits.

The bits will be packed together as follows: The sign bit is on the left, followed by the 3-bit exponent, followed by the three hexadecimal digits of the significand. Neither the radix nor the hexadecimal point will be stored in the packed form.

The reason for these rather odd-seeming choices is that numbers in this format can be compared for $=$, $\neq$, $\leq$, and $\geq$ in their "packed" format, which is shown in the illustration below:



Consider representing $(358)_{10}$ in this format.

The first step is to convert the fixed point number from its original base into a fixed point number in the target base. Using the method described in Section 2.1.3, we convert the base 10 number into a base 16 number as shown below:

		Integer	**Remainder**
358/16	=	22	6
22/16	=	1	6
1/16	=	0	1

Thus $(358)_{10} = (166)_{16}$. The next step is to convert the fixed point number into a floating point number:

$$(166)_{16} = (166.)_{16} \times 16^0$$

Note that the form 16^0 reflects a base of 16 with an exponent of 0, and that the number 16 as it appears on the page uses a base 10 form. That is, $(16^0)_{10} = (10^0)_{16}$. This is simply a notational convenience used in describing a floating point number.

The next step is to normalize the number:

$$(166.)_{16} \times 16^0 = (.166)_{16} \times 16^3$$

Finally, we fill in the bit fields of the number. The number is positive, and so we place a 0 in the sign bit position. The exponent is 3, but we represent it in excess 4, so the bit pattern for the exponent is computed as shown below:

	0 1 1	$(+3)_{10}$
Excess 4	+ 1 0 0	$(+4)_{10}$
Excess 4 exponent	1 1 1	

Alternatively, we could have simply computed $3 + 4 = 7$ in base 10, and then made the equivalent conversion $(7)_{10} = (111)_2$.

Finally, each of the base 16 digits is represented in binary as $1 = 0001$, $6 = 0110$, and $6 = 0110$. The final bit pattern is shown below:

0	1 1 1	.	0 0 0 1	0 1 1 0	0 1 1 0
+	3		1	6	6
Sign	Exponent			Fraction	

Notice again that the radix point is not explicitly represented in the bit pattern, but its presence is implied. The spaces between digits are for clarity only, and do not suggest that the bits are stored with spaces between them. The bit pattern as stored in a computer's memory would look like this:

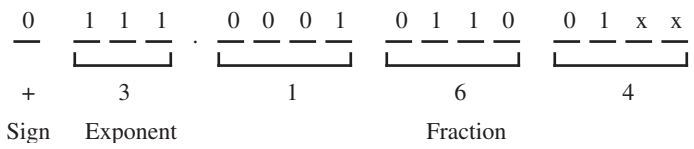
0111000101100110

The use of an excess 4 exponent instead of a two's complement or a signed magnitude exponent simplifies addition and subtraction of floating point numbers (which we will cover in detail in Chapter 3). In order to add or subtract two normalized floating point numbers, the smaller exponent (smaller in degree, not magnitude) must first be increased to the larger exponent (this retains the range), which also has the effect of unnormalizing the smaller number. In order to determine which exponent is larger, we only need to treat the bit patterns as unsigned numbers and then make our comparison. That is, using an excess 4 representa-

tion, the smallest exponent is −4, which is represented as 000. The largest exponent is +3, which is represented as 111. The remaining bit patterns for −3, −2, −1, 0, +1, and +2 fall in their respective order as 001, 010, 011, 100, 101, and 110.

Now if we are given the bit pattern shown above for $(358)_{10}$ along with a description of the floating point representation, then we can easily determine the number. The sign bit is a 0, which means that the number is positive. The exponent in unsigned form is the number $(+7)_{10}$, but since we are using excess 4, we must subtract 4 from it, which results in an actual exponent of $(+7 - 4 = +3)_{10}$. The fraction is grouped in four-bit hexadecimal digits, which gives a fraction of $(.166)_{16}$. Putting it all together results in $(+.166 \times 16^3)_{16} = (358)_{10}$.

Now suppose that only 10 bits are allowed for the fraction in the above example, instead of the 12 bits that group evenly into fours for hexadecimal digits. How does the representation change? One approach might be to round the fraction and adjust the exponent as necessary. Another approach, which we use here, is to simply truncate the least significant bits by **chopping** and avoid making adjustments to the exponent, so that the number we actually represent is:



If we treat the missing bits as 0's, then this bit pattern represents $(.164 \times 16^3)_{16}$. This method of truncation produces a biased error, since values of 00, 01, 10, and 11 in the missing bits are all treated as 0, and so the error is in the range from 0 to $(.003)_{16}$. The bias comes about because the error is not symmetric about 0. We will not explore the bias problem further here, but a more thorough discussion can be found in (Hamacher *et al.*, 1990).

We again stress that whatever the floating point format is, that it be known to all parties that intend to store or retrieve numbers in that format. The Institute of Electrical and Electronics Engineers (IEEE), has taken the lead in standardizing floating point formats. The IEEE 754 floating point format, which is in nearly universal usage, is discussed in Section 2.3.5.

2.3.4 *ERROR IN FLOATING POINT REPRESENTATIONS*

The fact that finite precision introduces error means that we should consider how great the error is (by "error", we mean the distance between two adjacent representable numbers), and whether it is acceptable for our application. As an example of a potential pitfall, consider representing one million in floating point, and then subtracting one million 1's from it. We may still be left with a million if the error is greater than 1.[1]

In order to characterize error, range, and precision, we use the following notation:

b	Base
s	Number of significant *digits* (not bits) in the fraction
M	Largest exponent
m	Smallest exponent

The number of significant digits in the fraction is represented by s, which is different than the number of bits in the fraction if the base is anything other than 2 (for example, base 16 uses four bits for each digit). In general, if the base is 2^k where k is an integer, then k bits are needed to represent each digit. The use of a hidden 1 increases s by one bit even though it does not increase the number of representable numbers. In the previous example, there are three significant digits in the base 16 fraction and there are 12 bits that make up the three digits. There are three bits in the excess 4 exponent, which gives an exponent range of $[-2^2$ to $2^2 - 1]$. For this case, $b = 16$, $s = 3$, $M = 3$, and $m = -4$.

In the analysis of a floating point representation, there are five characteristics that we consider: the number of representable numbers, the numbers that have the largest and smallest magnitudes (other than zero), and the sizes of the largest and smallest gaps between successive numbers.

The number of representable numbers can be determined as shown in Figure

1. Most computers these days will let this upper bound get at least as high as 8 million using the default precision.

2-8. The sign bit can take on two values, as indicated by the position marked

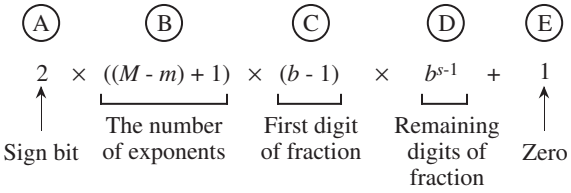


Figure 2-8 Calculation of the number of representable numbers in a floating point representation.

with an encircled "A." The total number of exponents is indicated in position B. Note that not all exponent bit patterns are valid in all representations. The IEEE 754 floating point standard, which we will study shortly, has a smallest exponent of −126 even though the eight-bit exponent can support a number as small as −128. The forbidden exponents are reserved for special numbers, such as zero and infinity.

The first digit of the fraction is considered next, which can take on any value except 0 in a normalized representation (except when a hidden 1 is used) as indicated by $(b - 1)$ at position C. The remaining digits of the fraction can take on any of the b values for the base, as indicated by b^{s-1} at position D. If a hidden 1 is used, then position C is removed and position 4 is replaced with b^s. Finally, there must be a representation for 0, which is accounted for in position E.

Consider now the numbers with the smallest and largest magnitudes. The number with the smallest magnitude has the smallest exponent and the smallest nonzero normalized fraction. There must be a nonzero value in the first digit, and since a 1 is the smallest value we can place there, the smallest fraction is b^{-1}. The number with the smallest magnitude is then $b^m \cdot b^{-1} = b^{m-1}$. Similarly, the number with the largest magnitude has the largest exponent and the largest fraction (when the fraction is all 1's), which is equal to $b^M \cdot (1 - b^{-s})$.

The smallest and largest gaps are computed in a similar manner. The smallest gap occurs when the exponent is at its smallest value and the least significant bit of the fraction changes. This gap is $b^m \cdot b^{-s} = b^{m-s}$. The largest gap occurs when the exponent is at its largest value and the least significant bit of the fraction changes. This gap is $b^M \cdot b^{-s} = b^{M-s}$.

As an example, consider a floating point representation in which there is a sign bit, a two-bit excess 2 exponent, and a three-bit normalized base 2 fraction in which the leading 1 is visible; that is, the leading 1 is not hidden. The representa-

tion of 0 is the bit pattern 000000. A number line showing all possible numbers that can be represented in this format is shown in Figure 2-9. Notice that there is

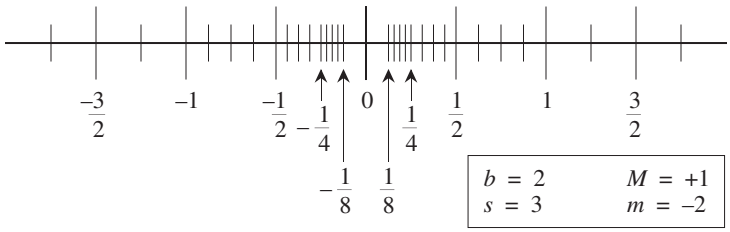


Figure 2-9 A number line showing all representable numbers in a simple floating point format.

a relatively large gap between 0 and the first representable number, because the normalized representation does not support bit patterns that correspond to numbers between 0 and the first representable number.

The smallest representable number occurs when the exponent and the fraction are at their smallest values. The smallest exponent is −2, and the smallest normalized fraction is $(.100)_2$. The smallest representable number is then $b^m \times b^{-1} = b^{m-1} = 2^{-2-1} = 1/8$.

Similarly, the largest representable number occurs when the exponent and the fraction are both at their largest values. The largest fraction occurs when the fraction is all 1's, which is a number that is 2^{-3} less than 1 since there are three digits in the fraction. The largest representable number is then $b^M \times (1 - b^{-s}) = 2^1 \times (1 - 2^{-3}) = 7/4$.

The smallest gap occurs when the exponent is at its smallest value and the least significant bit of the fraction changes, which is $b^m \times b^{-s} = b^{m-s} = 2^{-2-3} = 1/32$.

Similarly, the largest gap occurs when the exponent is at its largest value and the least significant bit of the fraction changes, which is $b^M \times b^{-s} = b^{M-s} = 2^{1-3} = 1/4$.

The number of bit patterns that represent valid numbers is less than the number of possible bit patterns, due to normalization. As discussed earlier, the number of representable numbers consists of five parts, which take into account the sign bit, the exponents, the first significant digit, the remaining digits, and the bit pattern for 0. This is computed as shown below:

$$2 \times ((M - m) + 1) \times (b - 1) \times b^{s-1} + 1$$

$$= 2 \times ((1 - (-2)) + 1) \times (2 - 1) \times 2^{3-1} + 1$$

$$= 33.$$

Notice that the gaps are small for small numbers and that the gaps are large for large numbers. In fact, the relative error is approximately the same for all numbers. If we take the ratio of a large gap to a large number, and compare that to the ratio of a small gap to a small number, then the ratios are the same:

$$\begin{matrix}\text{A large gap} \longrightarrow \\ \text{A large number} \longrightarrow\end{matrix} \quad \frac{b^{M-s}}{b^M \times (1 - b^{-s})} = \frac{b^{-s}}{1 - b^{-s}} = \frac{1}{b^s - 1}$$

and

$$\begin{matrix}\text{A small gap} \longrightarrow \\ \text{A small number} \longrightarrow\end{matrix} \quad \frac{b^{m-s}}{b^m \times (1 - b^{-s})} = \frac{b^{-s}}{1 - b^{-s}} = \frac{1}{b^s - 1}$$

The representation for a "small number" is used here, rather than the smallest number, because the large gap between zero and the first representable number is a special case.

EXAMPLE

Consider the problem of converting $(9.375 \times 10^{-2})_{10}$ to base 2 scientific notation. That is, the result should have the form $x.yy \times 2^z$. We start by converting from base 10 floating point to base 10 fixed point by moving the decimal point two positions to the left, which corresponds to the –2 exponent: .09375. We then convert from base 10 fixed point to base 2 fixed point by using the multiplication method:

.09375	×	2	=	0.1875
.1875	×	2	=	0.375
.375	×	2	=	0.75
.75	×	2	=	1.5

$$.5 \quad \times \quad 2 \quad = \quad 1.0$$

so $(.09375)_{10} = (.00011)_2$. Finally, we convert to normalized base 2 floating point: $.00011 = .00011 \times 2^0 = 1.1 \times 2^{-4}$. ■

2.3.5 THE IEEE 754 FLOATING POINT STANDARD

There are many ways to represent floating point numbers, a few of which we have already explored. Each representation has its own characteristics in terms of range, precision, and the number of representable numbers. In an effort to improve software portability and ensure uniform accuracy of floating point calculations, the IEEE 754 floating point standard for binary numbers was developed (IEEE, 1985). There are a few entrenched product lines that predate the standard that do not use it, such as the IBM/370, the DEC VAX, and the Cray line, but virtually all new architectures generally provide some level of IEEE 754 support.

The IEEE 754 standard as described below must be supported by a computer *system*, and not necessarily by the hardware entirely. That is, a mixture of hardware and software can be used while still conforming to the standard.

2.3.5.1 Formats

There are two primary formats in the IEEE 754 standard: **single precision** and **double precision**. Figure 2-10 summarizes the layouts of the two formats. The

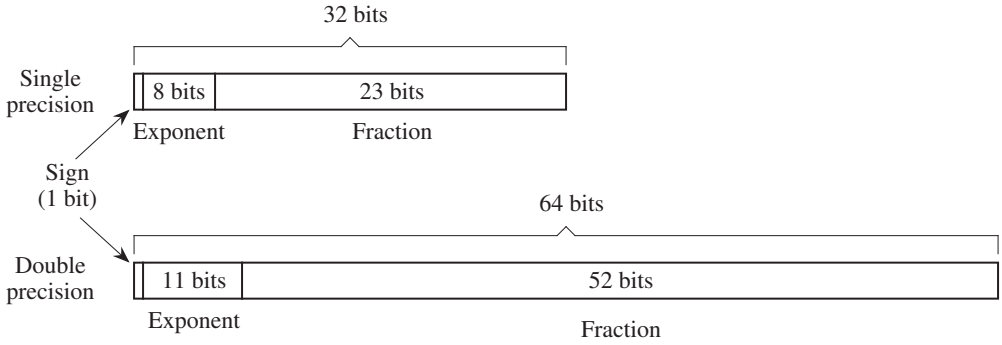


Figure 2-10 Single precision and double precision IEEE 754 floating point formats.

single precision format occupies 32 bits, whereas the double precision format occupies 64 bits. The double precision format is simply a wider version of the

single precision format.

The sign bit is in the leftmost position and indicates a positive or negative number for a 0 or a 1, respectively. The 8-bit excess 127 (*not* 128) exponent follows, in which the bit patterns 00000000 and 11111111 are reserved for special cases, as described below. For double precision, the 11-bit exponent is represented in excess 1023, with 00000000000 and 11111111111 reserved. The 23-bit base 2 fraction follows. There is a hidden bit to the *left* of the binary point, which when taken together with the single-precision fraction form a $23 + 1 = 24$-bit significand of the form 1.fff...f where the fff...f pattern represents the 23-bit fractional part that is stored. The double-precision format also uses a hidden bit to the left of the binary point, which supports a $52 + 1 = 53$ bit significand. For both formats, the number is normalized unless **denormalized** numbers are supported, as described later.

There are five basic types of numbers that can be represented. Nonzero normalized numbers take the form described above. A so-called "clean zero" is represented by the reserved bit pattern 00000000 in the exponent and all 0's in the fraction. The sign bit can be 0 or 1, and so there are two representations for zero: +0 and −0.

Infinity has a representation in which the exponent contains the reserved bit pattern 11111111, the fraction contains all 0's, and the sign bit is 0 or 1. Infinity is useful in handling overflow situations or in giving a valid representation to a number (other than zero) divided by zero. If zero is divided by zero or infinity is divided by infinity, then the result is undefined. This is represented by the **NaN** (not a number) format in which the exponent contains the reserved bit pattern 11111111, the fraction is nonzero and the sign bit is 0 or 1. A NaN can also be produced by attempting to take the square root of −1.

As with all normalized representations, there is a large gap between zero and the first representable number. The denormalized, "dirty zero" representation allows numbers in this gap to be represented. The sign bit can be 0 or 1, the exponent contains the reserved bit pattern 00000000 which represents −126 for single precision (−1022 for double precision), and the fraction contains the actual bit pattern for the magnitude of the number. Thus, there is no hidden 1 for this format. Note that the *denormalized* representation is not an *unnormalized* representation. The key difference is that there is only one representation for each denormalized number, whereas there are infinitely many unnormalized representations.

Figure 2-11 illustrates some examples of IEEE 754 floating point numbers.

	Value	Bit Pattern		
		Sign	Exponent	Fraction
(a)	$+1.101 \times 2^5$	0	1000 0100	101 0000 0000 0000 0000 0000
(b)	-1.01011×2^{-126}	1	0000 0001	010 1100 0000 0000 0000 0000
(c)	$+1.0 \times 2^{127}$	0	1111 1110	000 0000 0000 0000 0000 0000
(d)	+0	0	0000 0000	000 0000 0000 0000 0000 0000
(e)	–0	1	0000 0000	000 0000 0000 0000 0000 0000
(f)	$+\infty$	0	1111 1111	000 0000 0000 0000 0000 0000
(g)	$+2^{-128}$	0	0000 0000	010 0000 0000 0000 0000 0000
(h)	+NaN	0	1111 1111	011 0111 0000 0000 0000 0000
(i)	$+2^{-128}$	0	011 0111 1111	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Figure 2-11 Examples of IEEE 754 floating point numbers in single precision format (a – h) and double precision format (i). Spaces are shown for clarity only: they are not part of the representation.

Examples (a) through (h) are in single precision format and example (i) is in double precision format. Example (a) shows an ordinary single precision number. Notice that the significand is 1.101, but that only the fraction (101) is explicitly represented. Example (b) uses the smallest single precision exponent (–126) and example (c) uses the largest single precision exponent (127).

Examples (d) and (e) illustrate the two representations for zero. Example (f) illustrates the bit pattern for $+\infty$. There is also a corresponding bit pattern for $-\infty$. Example (g) shows a denormalized number. Notice that although the number itself is 2^{-128}, the smallest representable exponent is still –126. The exponent for single precision denormalized numbers is always –126, which is represented by the bit pattern 00000000 and a nonzero fraction. The fraction represents the magnitude of the number, rather than a significand. Thus we have $+2^{-128} = +.01 \times 2^{-126}$, which is represented by the bit pattern shown in Figure 2-11g.

Example (h) shows a single precision NaN. A NaN can be positive or negative. Finally, example (i) revisits the representation of 2^{-128} but now using double precision. The representation is for an ordinary double precision number and so there are no special considerations here. Notice that 2^{-128} has a significand of 1.0, which is why the fraction field is all 0's.

In addition to the single precision and double precision formats, there are also **single extended** and **double extended** formats. The extended formats are not

visible to the user, but they are used to retain a greater amount of internal precision during calculations to reduce the effects of roundoff errors. The extended formats increase the widths of the exponents and fractions by a number of bits that can vary depending on the implementation. For instance, the single extended format adds at least three bits to the exponent and eight bits to the fraction. The double extended format is typically 80 bits wide, with a 15-bit exponent and a 64-bit fraction.

2.3.5.2 Rounding

An implementation of IEEE 754 must provide at least single precision, whereas the remaining formats are optional. Further, the result of any single operation on floating point numbers must be accurate to within half a bit in the least significant bit of the fraction. This means that some additional bits of precision may need to be retained during computation (referred to as **guard bits**), and there must be an appropriate method of rounding the intermediate result to the number of bits in the fraction.

There are four rounding modes in the IEEE 754 standard. One mode rounds to 0, another rounds toward $+\infty$, and another rounds toward $-\infty$. The default mode rounds to the nearest representable number. Halfway cases round to the number whose low order digit is even. For example, 1.01101 rounds to 1.0110 whereas 1.01111 rounds to 1.1000.

2.4 Case Study: Patriot Missile Defense Failure Caused by Loss of Precision

During the 1991-1992 Operation Desert Storm conflict between Coalition forces and Iraq, the Coalition used a military base in Dhahran, Saudi Arabia that was protected by six U.S. Patriot Missile batteries. The Patriot system was originally designed to be mobile and to operate for only a few hours in order to avoid detection.

The Patriot system tracks and intercepts certain types of objects, such as cruise missiles or Scud ballistic missiles, one of which hit a U.S. Army barracks at Dhahran on February 5, 1991, killing 28 Americans. The Patriot system failed to track and intercept the incoming Scud due to a loss of precision in converting integers to a floating point number representation.

A radar system operates by sending out a train of electromagnetic pulses in vari-

ous directions and then listening for return signals that are reflected from objects in the path of the radar beam. If an airborne object of interest such as a Scud is detected by the Patriot radar system, then the position of a **range gate** is determined (see Figure 2-12), which estimates the position of the object being tracked

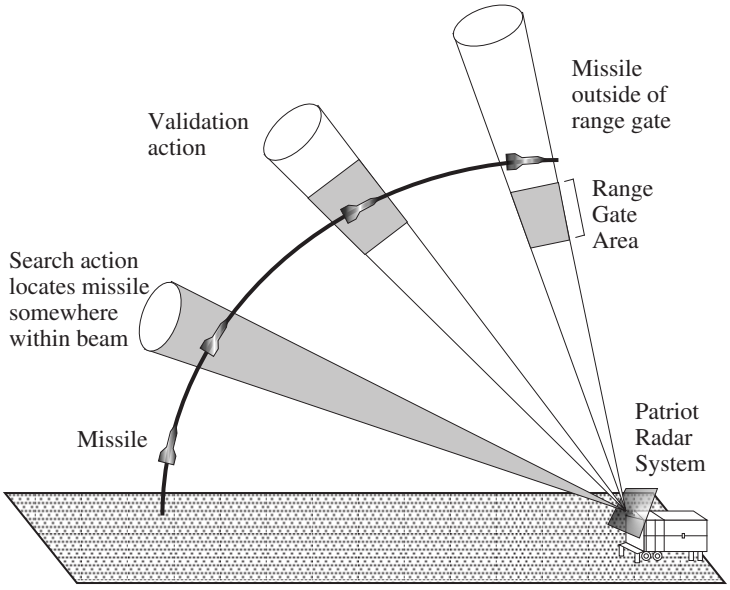


Figure 2-12 Effect of conversion error on range gate calculation.

during the next scan. The range gate also allows information outside of its boundaries to be filtered out, which simplifies tracking. The position of the object (a Scud for this case) is confirmed if it is found within the range gate.

The prediction of where the Scud will next appear is a function of the Scud's velocity. The Scud's velocity is determined by its change in position with respect to time, and time is updated in the Patriot's internal clock in 100 ms intervals. Velocity is represented as a 24-bit floating point number, and time is represented as a 24-bit integer, but both must be represented as 24-bit floating point numbers in order to predict where the Scud will next appear.

The conversion from integer time to real time results in a loss of precision that increases as the internal clock time increases. The error introduced by the conversion results in an error in the range gate calculation, which is proportional to the target's velocity and the length of time that the system is running. The cause of the Dhahran incident, after the Patriot battery had been operating continuously

for over 100 hours, is that the range gate shifted by 687 m, resulting in the failed interception of a Scud.

The conversion problem was known two weeks in advance of the Dhahran incident as a result of data provided by Israel, but it took until the day after the attack for new software to arrive due to the difficulty of distributing bug fixes in a wartime environment. A solution to the problem, until a software fix could be made available, would have been to simply reboot the system every few hours which would have the effect of resetting the internal clock. Since field personnel were not informed of how long was too long to keep a system running, which was in fact known at the time from data provided by Israel, this solution was never implemented. The lesson for us is to be very aware of the limitations of relying on calculations that use finite precision.

2.5 Character Codes

Unlike real numbers, which have an infinite range, there is only a finite number of characters. An entire character set can be represented with a small number of bits per character. Three of the most common character representations, ASCII, EBCDIC, and Unicode, are described here.

2.5.1 *THE ASCII CHARACTER SET*

The American Standard Code for Information Interchange (**ASCII**) is summarized in Figure 2-13, using hexadecimal indices. The representation for each character consists of 7 bits, and all 2^7 possible bit patterns represent valid characters. The characters in positions 00 – 1F and position 7F are special control characters that are used for transmission, printing control, and other non-textual purposes. The remaining characters are all printable, and include letters, numbers, punctuation, and a space. The digits 0-9 appear in sequence, as do the upper and lower case letters[1]. This organization simplifies character manipulation. In order to change the character representation of a digit into its numerical value, we can subtract $(30)_{16}$ from it. In order to convert the ASCII character '5,' which is in position $(35)_{16}$, into the number 5, we compute $(35 - 30 = 5)_{16}$. In

1. As an aside, the character 'a' and the character 'A' are different, and have different codes in the ASCII table. The small letters like 'a' are called **lower case**, and the capital letters like 'A' are called **upper case**. The naming comes from the positions of the characters in a printer's typecase. The capital letters appear above the small letters, which resulted in the upper case / lower case naming. These days, typesetting is almost always performed electronically, but the traditional naming is still used.

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

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

Figure 2-13 The ASCII character code, shown with hexadecimal indices.

order to convert an upper case letter into a lower case letter, we add $(20)_{16}$. For example, to convert the letter 'H ,' which is at location $(48)_{16}$ in the ASCII table, into the letter 'h,' which is at position $(68)_{16}$, we compute $(48 + 20 = 68)_{16}$.

2.5.2 *THE EBCDIC CHARACTER SET*

A problem with the ASCII code is that only 128 characters can be represented, which is a limitation for many keyboards that have a lot of special characters in addition to upper and lower case letters. The Extended Binary Coded Decimal Interchange Code (**EBCDIC**) is an eight-bit code that is used extensively in IBM mainframe computers. Since seven-bit ASCII characters are frequently represented in an eight-bit modified form (one character per byte), in which a 0 or a 1 is appended to the left of the seven-bit pattern, the use of EBCDIC does not

place a greater demand on the storage of characters in a computer. For serial transmission, however, (see Chapter 8), an eight-bit code takes more time to transmit than a seven-bit code, and for this case the wider code does make a difference.

The EBCDIC code is summarized in Figure 2-14. There are gaps in the table, which can be used for application specific characters. The fact that there are gaps in the upper and lower case sequences is not a major disadvantage because character manipulations can still be done as for ASCII, but using different offsets.

2.5.3 *THE UNICODE CHARACTER SET*

The ASCII and EBCDIC codes support the historically dominant (Latin) character sets used in computers. There are many more character sets in the world, and a simple ASCII-to-language-X mapping does not work for the general case, and so a new universal character standard was developed that supports a great breadth of the world's character sets, called **Unicode**.

Unicode is an evolving standard. It changes as new character sets are introduced into it, and as existing character sets evolve and their representations are refined. In version 2.0 of the Unicode standard, there are 38,885 distinct coded characters that cover the principal written languages of the Americas, Europe, the Middle East, Africa, India, Asia, and Pacifica.

The Unicode Standard uses a 16-bit code set in which there is a one-to-one correspondence between 16-bit codes and characters. Like ASCII, there are no complex modes or escape codes. While Unicode supports many more characters than ASCII or EBCDIC, it is not the end-all standard. In fact, the 16-bit Unicode standard is a subset of the 32-bit ISO 10646 Universal Character Set (UCS-4).

Glyphs for the first 256 Unicode characters are shown in Figure 2-15, according to Unicode version 2.1. Note that the first 128 characters are the same as for ASCII.

■ SUMMARY

All data in a computer is represented in terms of bits, which can be organized and interpreted as integers, fixed point numbers, floating point numbers, or characters.

00	NUL	20	DS	40	SP	60	–	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	‘	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	_	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F	|	6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	|
1B	CU1	3B	CU3	5B	$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	·	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	;	7E	=	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	¬	7F	"	9F		BF		DF		FF	

STX	Start of text	RS	Reader Stop	DC1	Device Control 1	BEL	Bell
DLE	Data Link Escape	PF	Punch Off	DC2	Device Control 2	SP	Space
BS	Backspace	DS	Digit Select	DC4	Device Control 4	IL	Idle
ACK	Acknowledge	PN	Punch On	CU1	Customer Use 1	NUL	Null
SOH	Start of Heading	SM	Set Mode	CU2	Customer Use 2		
ENQ	Enquiry	LC	Lower Case	CU3	Customer Use 3		
ESC	Escape	CC	Cursor Control	SYN	Synchronous Idle		
BYP	Bypass	CR	Carriage Return	IFS	Interchange File Separator		
CAN	Cancel	EM	End of Medium	EOT	End of Transmission		
RES	Restore	FF	Form Feed	ETB	End of Transmission Block		
SI	Shift In	TM	Tape Mark	NAK	Negative Acknowledge		
SO	Shift Out	UC	Upper Case	SMM	Start of Manual Message		
DEL	Delete	FS	Field Separator	SOS	Start of Significance		
SUB	Substitute	HT	Horizontal Tab	IGS	Interchange Group Separator		
NL	New Line	VT	Vertical Tab	IRS	Interchange Record Separator		
LF	Line Feed	UC	Upper Case	IUS	Interchange Unit Separator		

Figure 2-14 The EBCDIC character code, shown with hexadecimal indices.

0000	NUL	0020	SP	0040	@	0060	`	0080	Ctrl	00A0	NBS	00C0	À	00E0	à
0001	SOH	0021	!	0041	A	0061	a	0081	Ctrl	00A1	¡	00C1	Á	00E1	á
0002	STX	0022	"	0042	B	0062	b	0082	Ctrl	00A2	¢	00C2	Â	00E2	â
0003	ETX	0023	#	0043	C	0063	c	0083	Ctrl	00A3	£	00C3	Ã	00E3	ã
0004	EOT	0024	$	0044	D	0064	d	0084	Ctrl	00A4	¤	00C4	Ä	00E4	ä
0005	ENQ	0025	%	0045	E	0065	e	0085	Ctrl	00A5	¥	00C5	Å	00E5	å
0006	ACK	0026	&	0046	F	0066	f	0086	Ctrl	00A6	¦	00C6	Æ	00E6	æ
0007	BEL	0027	'	0047	G	0067	g	0087	Ctrl	00A7	§	00C7	Ç	00E7	ç
0008	BS	0028	(	0048	H	0068	h	0088	Ctrl	00A8	¨	00C8	È	00E8	è
0009	HT	0029	)	0049	I	0069	i	0089	Ctrl	00A9	©	00C9	É	00E9	é
000A	LF	002A	*	004A	J	006A	j	008A	Ctrl	00AA	ª	00CA	Ê	00EA	ê
000B	VT	002B	+	004B	K	006B	k	008B	Ctrl	00AB	«	00CB	Ë	00EB	ë
000C	FF	002C	´	004C	L	006C	l	008C	Ctrl	00AC	¬	00CC	Ì	00EC	ì
000D	CR	002D	-	004D	M	006D	m	008D	Ctrl	00AD	–	00CD	Í	00ED	í
000E	SO	002E	.	004E	N	006E	n	008E	Ctrl	00AE	®	00CE	Î	00EE	î
000F	SI	002F	/	004F	O	006F	o	008F	Ctrl	00AF	¯	00CF	Ï	00EF	ï
0010	DLE	0030	0	0050	P	0070	p	0090	Ctrl	00B0	°	00D0	Ð	00F0	¶
0011	DC1	0031	1	0051	Q	0071	q	0091	Ctrl	00B1	±	00D1	Ñ	00F1	ñ
0012	DC2	0032	2	0052	R	0072	r	0092	Ctrl	00B2	²	00D2	Ò	00F2	ò
0013	DC3	0033	3	0053	S	0073	s	0093	Ctrl	00B3	³	00D3	Ó	00F3	ó
0014	DC4	0034	4	0054	T	0074	t	0094	Ctrl	00B4	´	00D4	Ô	00F4	ô
0015	NAK	0035	5	0055	U	0075	u	0095	Ctrl	00B5	µ	00D5	Õ	00F5	õ
0016	SYN	0036	6	0056	V	0076	v	0096	Ctrl	00B6	¶	00D6	Ö	00F6	ö
0017	ETB	0037	7	0057	W	0077	w	0097	Ctrl	00B7	·	00D7	×	00F7	÷
0018	CAN	0038	8	0058	X	0078	x	0098	Ctrl	00B8	¸	00D8	Ø	00F8	ø
0019	EM	0039	9	0059	Y	0079	y	0099	Ctrl	00B9	¹	00D9	Ù	00F9	ù
001A	SUB	003A	:	005A	Z	007A	z	009A	Ctrl	00BA	º	00DA	Ú	00FA	ú
001B	ESC	003B	;	005B	[	007B	{	009B	Ctrl	00BB	»	00DB	Û	00FB	û
001C	FS	003C	<	005C	\	007C	|	009C	Ctrl	00BC	1/4	00DC	Ü	00FC	ü
001D	GS	003D	=	005D	]	007D	}	009D	Ctrl	00BD	1/2	00DD	Ý	00FD	Þ
001E	RS	003E	>	005E	^	007E	~	009E	Ctrl	00BE	3/4	00DE	ý	00FE	þ
001F	US	003F	?	005F	_	007F	DEL	009F	Ctrl	00BF	¿	00DF	ß	00FF	ÿ

NUL	Null	SOH	Start of heading	CAN	Cancel	SP	Space
STX	Start of text	EOT	End of transmission	EM	End of medium	DEL	Delete
ETX	End of text	DC1	Device control 1	SUB	Substitute	Ctrl	Control
ENQ	Enquiry	DC2	Device control 2	ESC	Escape	FF	Form feed
ACK	Acknowledge	DC3	Device control 3	FS	File separator	CR	Carriage return
BEL	Bell	DC4	Device control 4	GS	Group separator	SO	Shift out
BS	Backspace	NAK	Negative acknowledge	RS	Record separator	SI	Shift in
HT	Horizontal tab	NBS	Non-breaking space	US	Unit separator	DLE	Data link escape
LF	Line feed	ETB	End of transmission block	SYN	Synchronous idle	VT	Vertical tab

Figure 2-15 The first 256 glyphs in Unicode, shown with hexadecimal indices.

Character codes, such as ASCII, EBCDIC, and Unicode, have finite sizes and can thus be completely represented in a finite number of bits. The number of bits used

for representing numbers is also finite, and as a result only a subset of the real numbers can be represented. This leads to the notions of range, precision, and error. The range for a number representation defines the largest and smallest magnitudes that can be represented, and is almost entirely determined by the base and the number of bits in the exponent for a floating point representation. The precision is determined by the number of bits used in representing the magnitude (excluding the exponent bits in a floating point representation). Error arises in floating point representations because there are real numbers that fall within the gaps between adjacent representable numbers.

Further Reading

(Hamacher et al., 1990) provides a good explanation of biased error in floating point representations. The IEEE 754 floating point standard is described in (IEEE, 1985). The analysis of range, error, and precision in Section 2.3 was influenced by (Forsythe, 1970). The GAO report (U.S. GAO report GAO/IMTEC-92-26) gives a very readable account of the software problem that led to the Patriot failure in Dhahran. See http://www.unicode.org for information on the Unicode standard.

PROBLEMS

2.1 Given a signed, fixed point representation in base 10, with three digits to the left and right of the decimal point:

a) What is the range? (Calculate the highest positive number and the lowest negative number.)

b) What is the precision? (Calculate the difference between two adjacent numbers on a number line. Remember that the error is 1/2 the precision.)

2.2 Convert the following numbers as indicated, using as few digits in the results as necessary.

a) $(47)_{10}$ to unsigned binary.

b) $(-27)_{10}$ to binary signed magnitude.

c) $(213)_{16}$ to base 10.

d) $(10110.101)_2$ to base 10.

e) $(34.625)_{10}$ to base 4.

2.3 Convert the following numbers as indicated, using as few digits in the results as necessary.

a) $(011011)_2$ to base 10.

b) $(-27)_{10}$ to excess 32 in binary.

c) $(011011)_2$ to base 16.

d) $(55.875)_{10}$ to unsigned binary.

e) $(132.2)_4$ to base 16.

2.4 Convert $.201_3$ to decimal.

2.5 Convert $(43.3)_7$ to base 8 using no more than one octal digit to the right of the radix point. Truncate any remainder by chopping excess digits. Use an ordinary unsigned octal representation.

2.6 Represent $(17.5)_{10}$ in base 3, then convert the result back to base 10. Use two digits of precision to the right of the radix point for the intermediate base 3 form.

2.7 Find the decimal equivalent of the four-bit two's complement number: 1000.

2.8 Find the decimal equivalent of the four-bit one's complement number: 1111.

2.9 Show the representation for $(305)_{10}$ using three BCD digits.

2.10 Show the 10's complement representation for $(-305)_{10}$ using three BCD digits.

2.11 For a given number of bits, are there more representable integers in one's

complement, two's complement, or are they the same?

2.12 Complete the following table for the 5-bit representations (including the sign bits) indicated below. Show your answers as signed base 10 integers.

	5-bit signed magnitude	5-bit excess 16
Largest number		
Most negative number		
No. of distinct numbers		

2.13 Complete the following table using base 2 scientific notation and an eight-bit floating point representation in which there is a three-bit exponent in excess 3 notation (not excess 4), and a four-bit normalized fraction with a hidden '1'. In this representation, the hidden 1 is to the left of the radix point. This means that the number 1.0101 is in normalized form, whereas .101 is not.

Base 2 scientific notation	Floating point representation		
	Sign	Exponent	Fraction
-1.0101×2^{-2}			
$+1.1 \times 2^{2}$			
	0	001	0000
	1	110	1111

2.14 The IBM short floating point representation uses base 16, one sign bit, a seven-bit excess 64 exponent and a normalized 24-bit fraction.

a) What number is represented by the bit pattern shown below?

1 0111111 01110000 00000000 00000000

Show your answer in decimal. Note: the spaces are included in the number for readability only.

b) Represent $(14.3)_6$ in this notation.

2.15 For a normalized floating point representation, keeping everything else

the same but:

a) decreasing the base will increase / decrease / not change the number of representable numbers.

b) increasing the number of significant digits will increase / decrease / not change the smallest representable positive number.

c) increasing the number of bits in the exponent will increase / decrease / not change the range.

d) changing the representation of the exponent from excess 64 to two's complement will increase / decrease / not change the range.

2.16 For parts (a) through (e), use a floating point representation with a sign bit in the leftmost position, followed by a two-bit two's complement exponent, followed by a normalized three-bit fraction in base 2. Zero is represented by the bit pattern: 0 0 0 0 0 0. There is no hidden '1'.

a) What decimal number is represented by the bit pattern: 1 0 0 1 0 0?

b) Keeping everything else the same but changing the base to 4 will: increase / decrease / not change the smallest representable positive number.

c) What is the smallest gap between successive numbers?

d) What is the largest gap between successive numbers?

e) There are a total of six bits in this floating point representation, and there are $2^6 = 64$ unique bit patterns. How many of these bit patterns are valid?

2.17 Represent $(107.15)_{10}$ in a floating point representation with a sign bit, a seven-bit excess 64 exponent, and a normalized 24-bit fraction in base 2. There is no hidden 1. Truncate the fraction by chopping bits as necessary.

2.18 For the following single precision IEEE 754 bit patterns show the numerical value as a base 2 significand with an exponent (*e.g.* 1.11×2^5).

a) 0 10000011 01100000000000000000000

b) 1 10000000 00000000000000000000000

c) 1 00000000 00000000000000000000000

d) 1 11111111 00000000000000000000000

e) 0 11111111 11010000000000000000000

f) 0 00000001 10010000000000000000000

g) 0 00000011 01101000000000000000000

2.19 Show the IEEE 754 bit patterns for the following numbers:

a) $+1.1011 \times 2^5$ (single precision)

b) +0 (single precision)

c) -1.00111×2^{-1} (double precision)

d) −NaN (single precision)

2.20 Using the IEEE 754 single precision format, show the value (not the bit pattern) of:

a) The largest positive representable number (note: ∞ is not a number).

b) The smallest positive nonzero number that is normalized.

c) The smallest positive nonzero number in denormalized format.

d) The smallest normalized gap.

e) The largest normalized gap.

f) The number of normalized representable numbers (including 0; note that ∞ and NaN are not numbers).

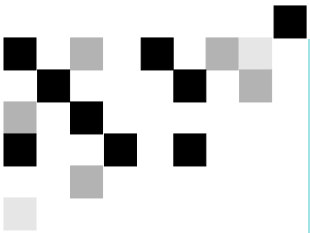
2.21 Two programmers write random number generators for normalized float-

ing point numbers using the same method. Programmer A's generator creates random numbers on the closed interval from 0 to 1/2, and programmer B's generator creates random numbers on the closed interval from 1/2 to 1. Programmer B's generator works correctly, but Programmer A's generator produces a skewed distribution of numbers. What could be the problem with Programmer A's approach?

2.22 A hidden 1 representation will not work for base 16. Why not?

2.23 With a hidden 1 representation, can 0 be represented if all possible bit patterns in the exponent and fraction fields are used for nonzero numbers?

2.24 Given a base 10 floating point number (*e.g.* $.583 \times 10^3$), can the number be converted into the equivalent base 2 form: $.x \times 2^y$ by separately converting the fraction (.583) and the exponent (3) into base 2?



3 ARITHMETIC

3.1 Overview

In the previous chapter we explored a few ways that numbers can be represented in a digital computer, but we only briefly touched upon arithmetic operations that can be performed on those numbers. In this chapter we cover four basic arithmetic operations: addition, subtraction, multiplication, and division. We begin by describing how these four operations can be performed on fixed point numbers, and continue with a description of how these four operations can be performed on floating point numbers.

Some of the largest problems, such as weather calculations, quantum mechanical simulations, and land-use modeling, tax the abilities of even today's largest computers. Thus the topic of high-performance arithmetic is also important. We conclude the chapter with an introduction to some of the algorithms and techniques used in speeding arithmetic operations.

3.2 Fixed Point Addition and Subtraction

The addition of binary numbers and the concept of overflow were briefly discussed in Chapter 2. Here, we cover addition and subtraction of both signed and unsigned fixed point numbers in detail. Since the two's complement representation of integers is almost universal in today's computers, we will focus primarily on two's complement operations. We will briefly cover operations on 1's complement and BCD numbers, which have a foundational significance for other areas of computing, such as networking (for 1's complement addition) and hand-held calculators (for BCD arithmetic.)

3.2.1 TWO'S COMPLEMENT ADDITION AND SUBTRACTION

In this section, we look at the addition of signed two's complement numbers. As we explore the *addition* of signed numbers, we also implicitly cover *subtraction* as well, as a result of the arithmetic principle:

$$a - b = a + (-b).$$

We can negate a number by complementing it (and adding 1, for two's complement), and so we can perform subtraction by complementing and adding. This results in a savings of hardware because it avoids the need for a hardware subtractor. We will cover this topic in more detail later.

We will need to modify the interpretation that we place on the results of addition when we add two's complement numbers. To see why this is the case, consider Figure 3-1. With addition on the real number line, numbers can be as large or as

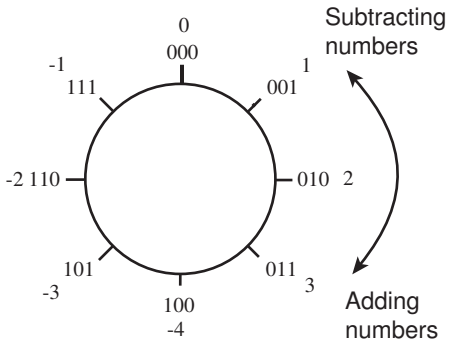


Figure 3-1 Number circle for 3-bit two's complement numbers.

small as desired—the number line goes to $\pm\infty$, so the real number line can accommodate numbers of any size. On the other hand, as discussed in Chapter 2, computers represent data using a finite number of bits, and as a result can only store numbers within a certain range. For example, an examination of Table 2.1 shows that if we restrict the size of a number to, for example, 3 bits, there will only be eight possible two's complement values that the number can assume. In Figure 3-1 these values are arranged in a circle beginning with 000 and proceeding around the circle to 111 and then back to 000. The figure also shows the decimal equivalents of these same numbers.

Some experimentation with the number circle shows that numbers can be added or subtracted by traversing the number circle clockwise for addition and counter-

clockwise for subtraction. Numbers can also be subtracted by two's complementing the subtrahend and adding. Notice that overflow can only occur for addition when the operands ("addend" and "augend") are of the same sign. Furthermore, overflow occurs if a transition is made from +3 to −4 while proceeding around the number circle when adding, or from −4 to +3 while subtracting. (Two's complement overflow is discussed in more detail later in the chapter.)

Here are two examples of 8-bit two's complement addition, first using two positive numbers:

```
   00001010      (+10)10
+  00010111      (+23)10
   --------
   00100001      (+33)10
```

A positive and a negative number can be added in a similar manner:

```
   00000101      (+5)10
+  11111110      (−2)10
   ----------
(1)00000011      (+3)10
```

Discard carry →

The carry produced by addition at the highest (leftmost) bit position is discarded in two's complement addition. A similar situation arises with a carry out of the highest bit position when adding two negative numbers:

```
   11111111      (−1)10
+  11111100      (−4)10
   --------
(1)11111011      (−5)10
```

Discard carry →

The carry out of the leftmost bit is discarded because the number system is **modular**—it "wraps around" from the largest positive number to the largest negative number as Figure 3-1 shows.

Although an addition operation may have a (discarded) carry-out from the MSB, this does not mean that the result is erroneous. The two examples above yield

correct results in spite of the fact that there is a carry-out of the MSB. The next section discusses overflow in two's complement addition in more detail.

Overflow

When two numbers are added that have large magnitudes and the same sign, an **overflow** will occur if the result is too large to fit in the number of bits used in the representation. Consider adding $(+80)_{10}$ and $(+50)_{10}$ using an eight bit format. The result should be $(+130)_{10}$, however, as shown below, the result is $(-126)_{10}$:

$$\begin{array}{rl} 0\,1\,0\,1\,0\,0\,0\,0 & (+80)_{10} \\ +\,0\,0\,1\,1\,0\,0\,1\,0 & (+50)_{10} \\ \hline 1\,0\,0\,0\,0\,0\,1\,0 & (-126)_{10} \end{array}$$

This should come as no surprise, since we know that the largest positive 8-bit two's complement number is $+(127)_{10}$, and it is therefore impossible to represent $(+130)_{10}$. Although the result 10000010_2 "looks" like 130_{10} if we think of it in unsigned form, the sign bit indicates a negative number in the signed form, which is clearly wrong.

In general, if two numbers of opposite signs are added, then an overflow cannot occur. Intuitively, this is because the magnitude of the result can be no larger than the magnitude of the larger operand. This leads us to the definition of two's complement overflow:

> *If the numbers being added are of the same sign and the result is of the opposite sign, then an overflow occurs and the result is incorrect. If the numbers being added are of opposite signs, then an overflow will never occur. As an alternative method of detecting overflow for addition, an overflow occurs if and only if the carry into the sign bit differs from the carry out of the sign bit.*
>
> *If a positive number is subtracted from a negative number and the result is positive, or if a negative number is subtracted from a positive number and the result is negative, then an overflow occurs. If the numbers being subtracted are of the same sign, then an overflow will never occur.*

3.2.2 HARDWARE IMPLEMENTATION OF ADDERS AND SUBTRACTORS

Up until now we have focused on algorithms for addition and subtraction. Now we will take a look at implementations of simple adders and subtractors.

Ripple-Carry Addition and Ripple-Borrow Subtraction

In Appendix A, a design of a four-bit ripple-carry adder is explored. The adder is modeled after the way that we normally perform decimal addition by hand, by summing digits in one column at a time while moving from right to left. In this section, we review the **ripple-carry adder**, and then take a look at a **ripple-borrow subtractor**. We then combine the two into a single addition/subtraction unit.

Figure 3-2 shows a 4-bit ripple-carry adder that is developed in Appendix A. Two

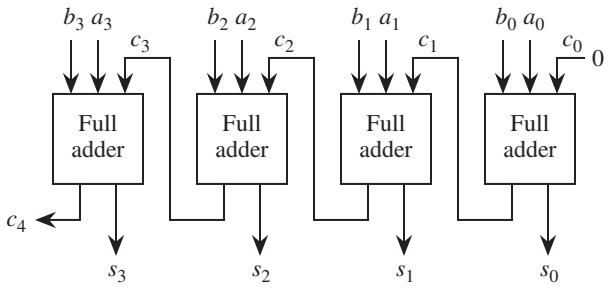


Figure 3-2 Ripple-carry adder.

binary numbers A and B are added from right to left, creating a sum and a carry at the outputs of each full adder for each bit position.

Four 4-bit ripple-carry adders are cascaded in Figure 3-3 to add two 16-bit numbers. The rightmost full adder has a carry-in of 0. Although the rightmost full adder can be simplified as a result of the carry-in of 0, we will use the more general form and force c_0 to 0 in order to simplify subtraction later on.

Subtraction of binary numbers proceeds in a fashion analogous to addition. We can subtract one number from another by working in a single column at a time, subtracting digits of the **subtrahend** b_i, from the **minuend** a_i, as we move from right to left. As in decimal subtraction, if the subtrahend is larger than the minuend or there is a borrow from a previous digit then a borrow must be propagated

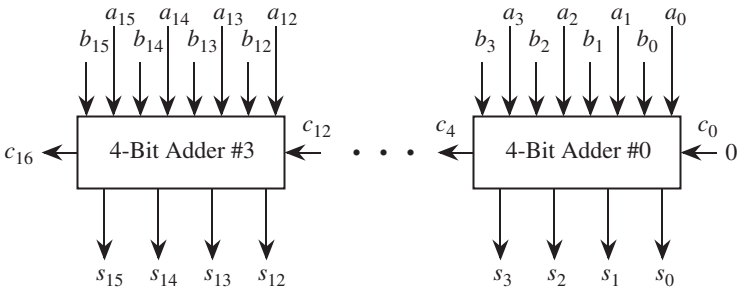


Figure 3-3 A 16-bit adder is made up of a cascade of four 4-bit ripple-carry adders.

to the next most significant bit. Figure 3-4 shows the truth table and a "black-box" circuit for subtraction.

a_i	b_i	bor_i	$diff_i$	bor_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

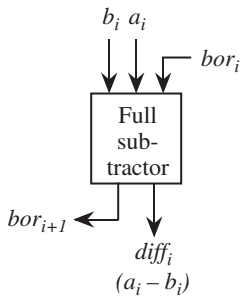


Figure 3-4 Truth table and schematic symbol for a ripple-borrow subtractor.

Full subtractors can be cascaded to form **ripple-borrow** subtractors in the same manner that full adders are cascaded to form ripple-carry adders. Figure 3-5 illus-

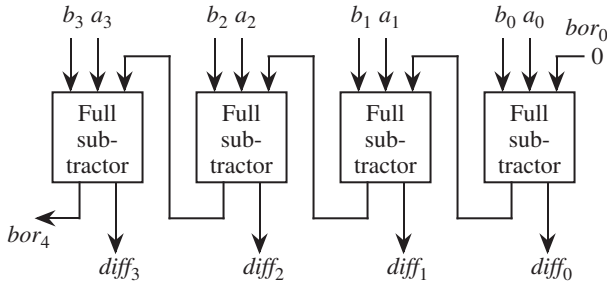


Figure 3-5 Ripple-borrow subtractor.

trates a four-bit ripple-borrow subtractor that is made up of four full subtractors.

As discussed above, an alternative method of implementing subtraction is to form the two's complement negative of the subtrahend and *add* it to the minuend. The circuit that is shown in Figure 3-6 performs both addition and subtrac-

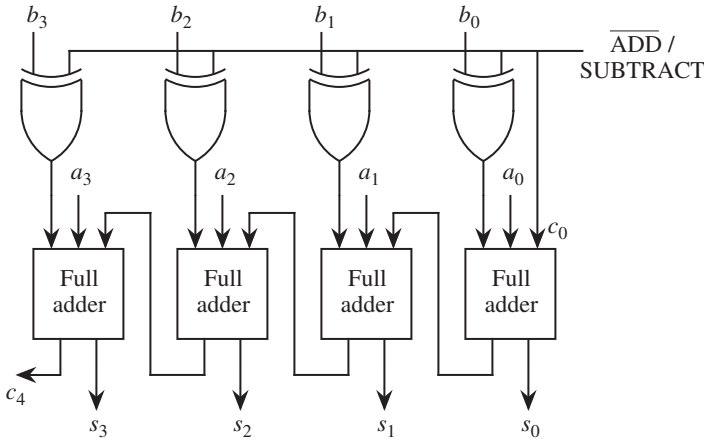


Figure 3-6 Addition / subtraction unit.

tion on four-bit two's complement numbers by allowing the b_i inputs to be complemented when subtraction is desired. An $\overline{\text{ADD}}$/SUBTRACT control line determines which function is performed. The bar over the ADD symbol indicates the ADD operation is active when the signal is low. That is, if the control line is 0, then the a_i and b_i inputs are passed through to the adder, and the sum is generated at the s_i outputs. If the control line is 1, then the a_i inputs are passed through to the adder, but the b_i inputs are one's complemented by the XOR gates before they are passed on to the adder. In order to form the two's complement negative, we must add 1 to the one's complement negative, which is accomplished by setting the *carry_in* line (c_0) to 1 with the control input. In this way, we can share the adder hardware among both the adder and the subtractor.

3.2.3 *ONE'S COMPLEMENT ADDITION AND SUBTRACTION*

Although it is not heavily used in mainstream computing anymore, the one's complement representation was used in early computers. One's complement addition is handled somewhat differently from two's complement addition: the carry out of the leftmost position is not discarded, but is added back into the least significant position of the integer portion as shown in Figure 3-7. This is

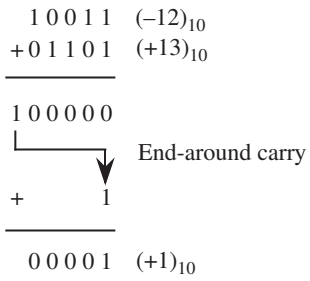


Figure 3-7 An example of one's complement addition with an end-around carry.

known as an **end-around carry**.

We can better visualize the reason that the end-around carry is needed by examining the 3-bit one's complement number circle in Figure 3-8. Notice that the

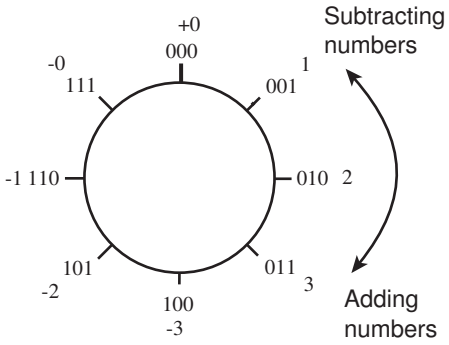


Figure 3-8 Number circle for a three-bit signed one's complement representation.

number circle has two positions for 0. When we add two numbers, if we traverse through both −0 and +0, then we must compensate for the fact that 0 is visited twice. The end-around carry advances the result by one position for this situation.

Notice that the distance between −0 and +0 on the number circle is the distance between two integers, and is *not* the distance between two successive representable numbers. As an illustration of this point, consider adding $(5.5)_{10}$ and $(-1.0)_{10}$ in one's complement arithmetic, which is shown in Figure 3-9. (Note that we can also treat this as a subtraction problem, in which the subtrahend is negated by complementing all of the bits, before adding it to the minuend.) In

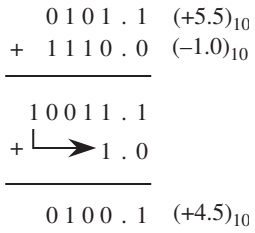


Figure 3-9 The end-around carry complicates addition for non-integers.

order to add $(+5.5)_{10}$ and $(-1.0)_{10}$ and obtain the correct result in one's complement, we add the end-around carry into the one's position as shown. This adds complexity to our number circle, because in the gap between +0 and –0, there are valid numbers that represent fractions that are less than 0, yet they appear on the number circle before –0 appears. If the number circle is reordered to avoid this anomaly, then addition must be handled in a more complex manner.

The need to look for two different representations for zero, and the potential need to perform another addition for the end-around carry are two important reasons for preferring the two's complement arithmetic to one's complement arithmetic.

3.3 Fixed Point Multiplication and Division

Multiplication and division of fixed point numbers can be accomplished with addition, subtraction, and shift operations. The sections that follow describe methods for performing multiplication and division of fixed point numbers in both unsigned and signed forms using these basic operations. We will first cover unsigned multiplication and division, and then we will cover signed multiplication and division.

3.3.1 *UNSIGNED MULTIPLICATION*

Multiplication of unsigned binary integers is handled similar to the way it is carried out by hand for decimal numbers. Figure 3-10 illustrates the multiplication process for two unsigned binary integers. Each bit of the multiplier determines whether or not the multiplicand, shifted left according to the position of the multiplier bit, is added into the product. When two unsigned *n*-bit numbers are multiplied, the result can be as large as 2*n* bits. For the example shown in Figure 3-10, the multiplication of two four-bit operands results in an eight-bit product. When two signed *n*-bit numbers are multiplied, the result can be as large as only

```
                1 1 0 1    (13)10   Multiplicand M
              × 1 0 1 1    (11)10   Multiplier Q
              ---------
                1 1 0 1
              1 1 0 1                Partial products
            0 0 0 0
          1 1 0 1
          ---------------
1 0 0 0 1 1 1 1    (143)10  Product P
```

Figure 3-10 Multiplication of two unsigned binary integers.

$2(n\text{-}1)+1 = (2n\text{-}1)$ bits, because this is equivalent to multiplying two $(n\text{-}1)$-bit unsigned numbers and then introducing the sign bit.

A hardware implementation of integer multiplication can take a similar form to the manual method. Figure 3-11 shows a layout of a multiplication unit for

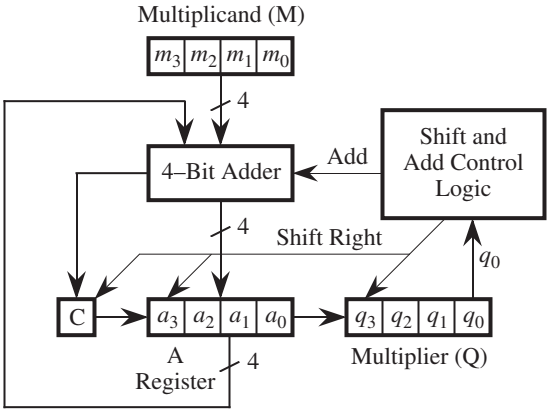


Figure 3-11 A serial multiplier.

four-bit numbers, in which there is a four-bit adder, a control unit, three four-bit registers, and a one-bit carry register. In order to multiply two numbers, the multiplicand is placed in the M register, the multiplier is placed in the Q register, and the A and C registers are cleared to zero. During multiplication, the rightmost bit of the multiplier determines whether the multiplicand is added into the product at each step. After the multiplicand is added into the product, the multiplier and the A register are simultaneously shifted to the right. This has the effect of shifting the multiplicand to the left (as for the manual process) and exposing the next bit of the multiplier in position q_0.

Figure 3-12 illustrates the multiplication process. Initially, C and A are cleared,

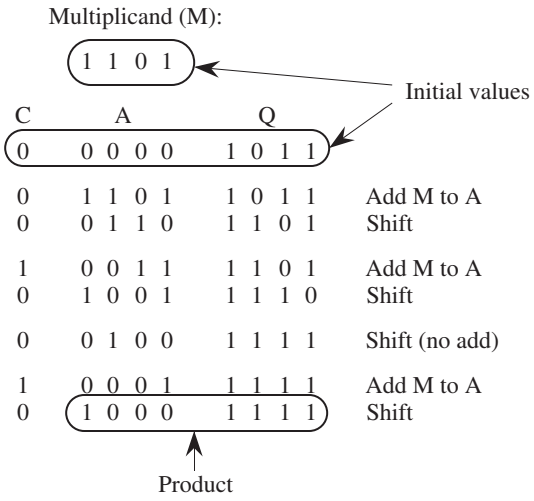


Figure 3-12 An example of multiplication using the serial multiplier.

and M and Q hold the multiplicand and multiplier, respectively. The rightmost bit of Q is 1, and so the multiplier M is added into the product in the A register. The A and Q registers together make up the eight-bit product, but the A register is where the multiplicand is added. After M is added to A, the A and Q registers are shifted to the right. Since the A and Q registers are linked as a pair to form the eight-bit product, the rightmost bit of A is shifted into the leftmost bit of Q. The rightmost bit of Q is then dropped, C is shifted into the leftmost bit of A, and a 0 is shifted into C.

The process continues for as many steps as there are bits in the multiplier. On the second iteration, the rightmost bit of Q is again 1, and so the multiplicand is added to A and the C/A/Q combination is shifted to the right. On the third iteration, the rightmost bit of Q is 0 so M is not added to A, but the C/A/Q combination is still shifted to the right. Finally, on the fourth iteration, the rightmost bit of Q is again 1, and so M is added to A and the C/A/Q combination is shifted to the right. The product is now contained in the A and Q registers, in which A holds the high-order bits and Q holds the low-order bits.

3.3.2 *UNSIGNED DIVISION*

In longhand binary division, we must successively attempt to subtract the divisor from the dividend, using the fewest number of bits in the dividend as we can. Figure 3-13 illustrates this point by showing that $(11)_2$ does not "fit" in 0 or 01,

```
      0 0 1 0  R 1
    ---------------
1 1 | 0 1 1 1
      1 1
    -------
      0 1
```

Figure 3-13 Example of base 2 division.

but *does* fit in 011 as indicated by the pattern 001 that starts the quotient.

Computer-based division of binary integers can be handled similar to the way that binary integer multiplication is carried out, but with the complication that the only way to tell if the dividend does not "fit" is to actually do the subtraction and test if the remainder is negative. If the remainder is negative then the subtraction must be "backed out" by adding the divisor back in, as described below.

In the division algorithm, instead of shifting the product to the right as we did for multiplication, we now shift the quotient to the left, and we subtract instead of adding. When two n-bit unsigned numbers are being divided, the result is no larger than n bits.

Figure 3-14 shows a layout of a division unit for four-bit numbers in which there

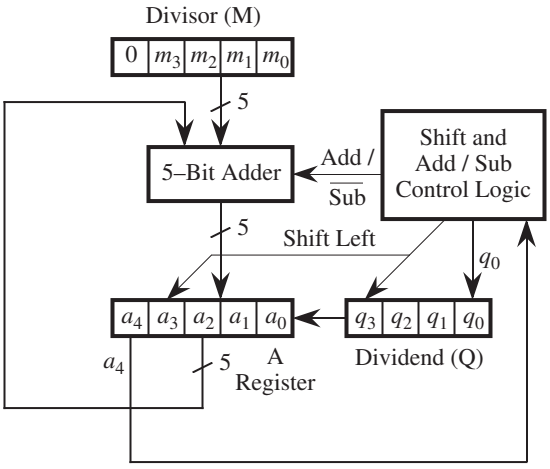


Figure 3-14 A serial divider.

is a five-bit adder, a control unit, a four-bit register for the dividend Q, and two five-bit registers for the divisor M and the remainder A. Five-bit registers are used for A and M, instead of 4-bit registers as we might expect, because an extra bit is

needed to indicate the sign of the intermediate result. Although this division method is for unsigned numbers, subtraction is used in the process and negative partial results sometimes arise, which extends the range from −16 through +15, thus there is a need for 5 bits to store intermediate results.

In order to divide two four-bit numbers, the dividend is placed in the Q register, the divisor is placed in the M register, and the A register and the high order bit of M are cleared to zero. The leftmost bit of the A register determines whether the divisor is added back into the dividend at each step. This is necessary in order to restore the dividend when the result of subtracting the divisor is negative, as described above. This is referred to as **restoring division**, because the dividend is restored to its former value when the remainder is negative. When the result is not negative, then the least significant bit of Q is set to 1, which indicates that the divisor "fits" in the dividend at that point.

Figure 3-15 illustrates the division process. Initially, A and the high order bit of M are cleared, and Q and the low order bits of M are loaded with the dividend and divisor, respectively. The A and Q registers are shifted to the left as a pair and the divisor M is subtracted from A. Since the result is negative, the divisor is added back to restore the dividend, and q_0 is cleared to 0. The process repeats by shifting A and Q to the left, and by subtracting M from A. Again, the result is negative, so the dividend is restored and q_0 is cleared to 0. On the third iteration, A and Q are shifted to the left and M is again subtracted from A, but now the result of the subtraction is not negative, so q_0 is set to 1. The process continues for one final iteration, in which A and Q are shifted to the left and M is subtracted from A, which produces a negative result. The dividend is restored and q_0 is cleared to 0. The quotient is now contained in the Q register and the remainder is contained in the A register.

3.3.3 *SIGNED MULTIPLICATION AND DIVISION*

If we apply the multiplication and division methods described in the previous sections to signed integers, then we will run into some trouble. Consider multiplying −1 by +1 using four-bit words, as shown in the left side of Figure 3-16. The eight-bit equivalent of +15 is produced instead of −1. What went wrong is that the sign bit did not get extended to the left of the result. This is not a problem for a positive result because the high order bits default to 0, producing the correct sign bit 0.

A solution is shown in the right side of Figure 3-16, in which each partial prod-

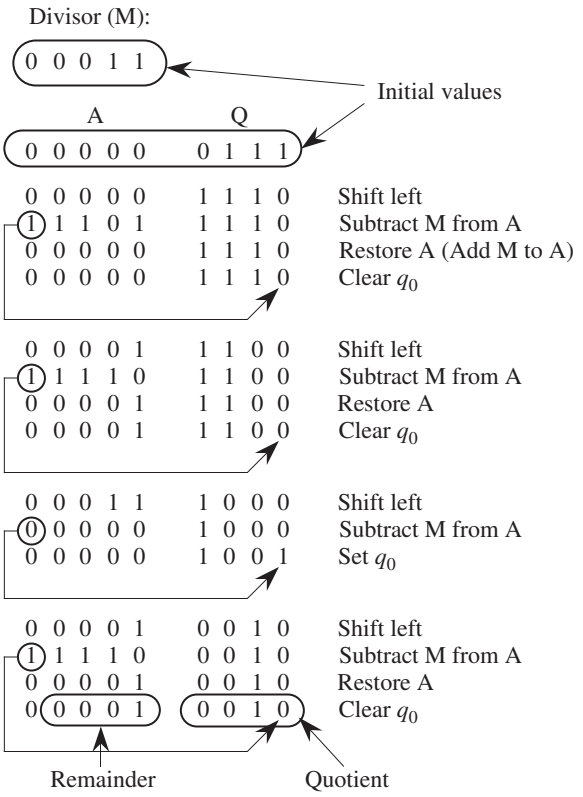


Figure 3-15 An example of division using the serial divider.

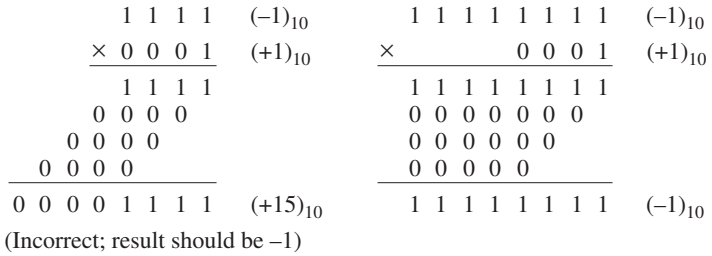


Figure 3-16 Multiplication of signed integers.

uct is extended to the width of the result, and only the rightmost eight bits of the result are retained. If both operands are negative, then the signs are extended for both operands, again retaining only the rightmost eight bits of the result.

Signed division is more difficult. We will not explore the methods here, but as a

general technique, we can convert the operands into their positive forms, perform the division, and then convert the result into its true signed form as a final step.

3.4 Floating Point Arithmetic

Arithmetic operations on floating point numbers can be carried out using the fixed point arithmetic operations described in the previous sections, with attention given to maintaining aspects of the floating point representation. In the sections that follow, we explore floating point arithmetic in base 2 and base 10, keeping the requirements of the floating point representation in mind.

3.4.1 *FLOATING POINT ADDITION AND SUBTRACTION*

Floating point arithmetic differs from integer arithmetic in that exponents must be handled as well as the magnitudes of the operands. As in ordinary base 10 arithmetic using scientific notation, the exponents of the operands must be made equal for addition and subtraction. The fractions are then added or subtracted as appropriate, and the result is normalized.

This process of adjusting the fractional part, and also rounding the result can lead to a loss of precision in the result. Consider the unsigned floating point addition ($.101 \times 2^3 + .111 \times 2^4$) in which the fractions have three significant digits. We start by adjusting the *smaller* exponent to be equal to the *larger* exponent, and adjusting the fraction accordingly. Thus we have $.101 \times 2^3 = .010 \times 2^4$, losing $.001 \times 2^3$ of precision in the process. The resulting sum is

$$(.010 + .111) \times 2^4 = 1.001 \times 2^4 = .1001 \times 2^5,$$

and rounding to three significant digits, $.100 \times 2^5$, and we have lost another 0.001×2^4 in the rounding process.

Why do floating point numbers have such complicated formats?

We may wonder why floating point numbers have such a complicated structure, with the mantissa being stored in signed magnitude representation, the exponent stored in excess notation, and the sign bit separated from the rest of the magnitude by the intervening exponent field. There is a simple explanation for this structure. Consider the complexity of performing floating point arithmetic in a computer. Before any arithmetic can be done, the number must be unpacked

from the form it takes in storage. (See Chapter 2 for a description of the IEEE 754 floating point format.) The exponent and mantissa must be extracted from the packed bit pattern before an arithmetic operation can be performed; after the arithmetic operation(s) are performed, the result must be renormalized and rounded, and then the bit patterns are re-packed into the requisite format.

The virtue of a floating point format that contains a sign bit followed by an exponent in excess notation, followed by the magnitude of the mantissa, is that two floating point numbers can be compared for >, <, and = without unpacking. The sign bit is most important in such a comparison, and it appropriately is the MSB in the floating point format. Next most important in comparing two numbers is the exponent, since a change of ± 1 in the exponent changes the value by a factor of 2 (for a base 2 format), whereas a change in even the MSB of the fractional part will change the value of the floating point number by less than that.

In order to account for the sign bit, the signed magnitude fractions are represented as integers and are converted into two's complement form. After the addition or subtraction operation takes place in two's complement, there may be a need to normalize the result and adjust the sign bit. The result is then converted back to signed magnitude form.

3.4.2 *FLOATING POINT MULTIPLICATION AND DIVISION*

Floating point multiplication and division are performed in a manner similar to floating point addition and subtraction, except that the sign, exponent, and fraction of the result can be computed separately. If the operands have the same sign, then the sign of the result is positive. Unlike signs produce a negative result. The exponent of the result before normalization is obtained by adding the exponents of the source operands for multiplication, or by subtracting the divisor exponent from the dividend exponent for division. The fractions are multiplied or divided according to the operation, followed by normalization.

Consider using three-bit fractions in performing the base 2 computation: $(+.101 \times 2^{2}) \times (-.110 \times 2^{-3})$. The source operand signs differ, which means that the result will have a negative sign. We add exponents for multiplication, and so the exponent of the result is $2 + -3 = -1$. We multiply the fractions, which produces the product .01111. Normalizing the product and retaining only three bits in the fraction produces $-.111 \times 2^{-2}$.

Now consider using three-bit fractions in performing the base 2 computation:

$(+.110 \times 2^5) / (+.100 \times 2^4)$. The source operand signs are the same, which means that the result will have a positive sign. We subtract exponents for division, and so the exponent of the result is 5 – 4 = 1. We divide fractions, which can be done in a number of ways. If we treat the fractions as unsigned integers, then we will have 110/100 = 1 with a remainder of 10. What we really want is a contiguous set of bits representing the fraction instead of a separate result and remainder, and so we can scale the dividend to the left by two positions, producing the result: 11000/100 = 110. We then scale the result to the right by two positions to restore the original scale factor, producing 1.1. Putting it all together, the result of dividing $(+.110 \times 2^5)$ by $(+.100 \times 2^4)$ produces $(+1.10 \times 2^1)$. After normalization, the final result is $(+.110 \times 2^2)$.

3.5 High Performance Arithmetic

For many applications, the speed of arithmetic operations are the bottleneck to performance. Most supercomputers, such as the Cray, the Tera, and the Intel Hypercube are considered "super" because they excel at performing fixed and floating point arithmetic. In this section we discuss a number of ways to improve the speed of addition, subtraction, multiplication, and division.

3.5.1 *HIGH PERFORMANCE ADDITION*

The ripple-carry adder that we reviewed in Section 3.2.2 may introduce too much delay into a system. The longest path through the adder is from the inputs of the least significant full adder to the outputs of the most significant full adder. The process of summing the inputs at each bit position is relatively fast (a small two-level circuit suffices) but the carry propagation takes a long time to work its way through the circuit. In fact, the propagation time is proportional to the number of bits in the operands. This is unfortunate, since more significant figures in an addition translates to more time to perform the addition. In this section, we look at a method of speeding the carry propagation in what is known as a **carry lookahead adder**.

In Appendix B, reduced Boolean expressions for the sum (s_i) and carry outputs (c_{i+1}) of a full adder are created. These expressions are repeated below, with subscripts added to denote the relative position of a full adder in a ripple-carry adder:

$$s_i = \bar{a}_i\bar{b}_i c_i + \bar{a}_i b_i \bar{c}_i + a_i \bar{b}_i \bar{c}_i + a_i b_i c_i$$