A

BINARY NUMBERS

The arithmetic used by computers differs in some ways from the arithmetic used by people. The most important difference is that computers perform operations on numbers whose precision is finite and fixed. Another difference is that most computers use the binary rather than the decimal system for representing numbers. These topics are the subject of this appendix.

A.1 FINITE-PRECISION NUMBERS

While doing arithmetic, one usually gives little thought to the question of how many decimal digits it takes to represent a number. Physicists can calculate that there are 10^{78} electrons in the universe without being bothered by the fact that it requires 79 decimal digits to write that number out in full. Someone calculating the value of a function with pencil and paper who needs the answer to six significant digits simply keeps intermediate results to seven, or eight, or however many are needed. The problem of the paper not being wide enough for seven-digit numbers never arises.

With computers, matters are quite different. On most computers, the amount of memory available for storing a number is fixed at the time that the computer is designed. With a certain amount of effort, the programmer can represent numbers two, or three, or even many times larger than this fixed amount, but doing so does not change the nature of this difficulty. The finite nature of the computer forces

us to deal only with numbers that can be represented in a fixed number of digits. We call such numbers **finite-precision numbers**.

In order to study properties of finite-precision numbers, let us examine the set of positive integers representable by three decimal digits, with no decimal point and no sign. This set has exactly 1000 members: 000, 001, 002, 003, ..., 999. With this restriction, it is impossible to express certain kinds of numbers, such as

1. Numbers larger than 999.
2. Negative numbers.
3. Fractions.
4. Irrational numbers.
5. Complex numbers.

One important property of arithmetic on the set of all integers is **closure** with respect to the operations of addition, subtraction, and multiplication. In other words, for every pair of integers i and j, $i + j$, $i - j$, and $i \times j$ are also integers. The set of integers is not closed with respect to division, because there exist values of i and j for which i/j is not expressible as an integer (e.g., 7/2 and 1/0).

Finite-precision numbers are not closed with respect to any of these four basic operations, as shown below, using three-digit decimal numbers as an example:

$600 + 600 = 1200$	(too large)
$003 - 005 = -2$	(negative)
$050 \times 050 = 2500$	(too large)
$007 / 002 = 3.5$	(not an integer)

The violations can be divided into two mutually exclusive classes: operations whose result is larger than the largest number in the set (overflow error) or smaller than the smallest number in the set (underflow error), and operations whose result is neither too large nor too small but is simply not a member of the set. Of the four violations above, the first three are examples of the former, and the fourth is an example of the latter.

Because computers have finite memories and therefore must of necessity perform arithmetic on finite-precision numbers, the results of certain calculations will be, from the point of view of classical mathematics, just plain wrong. A calculating device that gives the wrong answer even though it is in perfect working condition may appear strange at first, but the error is a logical consequence of its finite nature. Some computers have special hardware that detects overflow errors.

The algebra of finite-precision numbers is different from normal algebra. As an example, consider the associative law:

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

Let us evaluate both sides for $a = 700$, $b = 400$, $c = 300$. To compute the left-hand side, first calculate $(b - c)$, which is 100, and then add this amount to a,

yielding 800. To compute the right-hand side, first calculate $(a + b)$, which gives an overflow in the finite arithmetic of three-digit integers. The result may depend on the machine being used but it will not be 1100. Subtracting 300 from some number other than 1100 will not yield 800. The associative law does not hold. The order of operations is important.

As another example, consider the distributive law:

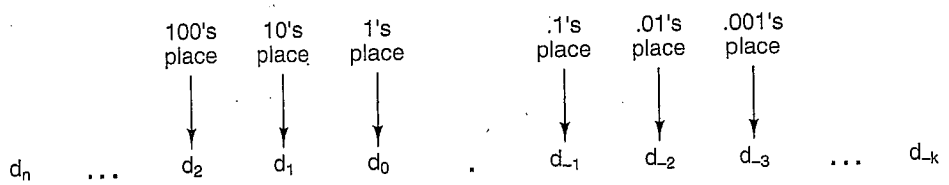
$$a \times (b - c) = a \times b - a \times c$$

Let us evaluate both sides for $a = 5$, $b = 210$, $c = 195$. The left-hand side is 5×15, which yields 75. The right-hand side is not 75 because $a \times b$ overflows.

Judging from these examples, one might conclude that although computers are general-purpose devices, their finite nature renders them especially unsuitable for doing arithmetic. This conclusion is, of course, not true, but it does serve to illustrate the importance of understanding how computers work and what limitations they have.

A.2 RADIX NUMBER SYSTEMS

An ordinary decimal number with which everyone is familiar consists of a string of decimal digits and, possibly, a decimal point. The general form and its usual interpretation are shown in Fig. A-1. The choice of 10 as the base for exponentiation, called the **radix**, is made because we are using decimal, or base 10, numbers. When dealing with computers, it is frequently convenient to use radices other than 10. The most important radices are 2, 8, and 16. The number systems based on these radices are called **binary**, **octal**, and **hexadecimal**, respectively.



$$\text{Number} = \sum_{i=-k}^{n} d_i \times 10^i$$

Figure A-1. The general form of a decimal number.

A radix k number system requires k different symbols to represent the digits 0 to $k - 1$. Decimal numbers are built up from the 10 decimal digits

0 1 2 3 4 5 6 7 8 9

In contrast, binary numbers do not use these ten digits. They are all constructed exclusively from the two binary digits

0 1

Octal numbers are built up from the eight octal digits

0 1 2 3 4 5 6 7

For hexadecimal numbers, 16 digits are needed. Thus six new symbols are required. It is conventional to use the uppercase letters A through F for the six digits following 9. Hexadecimal numbers are then built up from the digits

0 1 2 3 4 5 6 7 8 9 A B C D E F

The expression "binary digit" meaning a 1 or a 0 is usually referred to as a **bit**. Figure A-2 shows the decimal number 2001 expressed in binary, octal, decimal, and hexadecimal form. The number $'7B9$ is obviously hexadecimal, because the symbol B can only occur in hexadecimal numbers. However, the number 111 might be in any of the four number systems discussed. To avoid ambiguity, people use a subscript of 2, 8, 10, or 16 to indicate the radix when it is not obvious from the context.

Binary 1 1 1 1 1 0 1 0 0 0 1

$1\times2^{10}+1\times2^{9}+1\times2^{8}+1\times2^{7}+1\times2^{6}+0\times2^{5}+1\times2^{4}+0\times2^{3}+0\times2^{2}+0\times2^{1}+1\times2^{0}$

1024 + 512 + 256 + 128 + 64 + 0 + 16 + 0 + 0 + 0 + 1

Octal 3 7 2 1

$3\times8^{3}+7\times8^{2}+2\times8^{1}+1\times8^{0}$

1536 + 448 + 16 + 1

Decimal 2 0 0 1

$2\times10^{3}+0\times10^{2}+0\times10^{1}+1\times10^{0}$

2000 + 0 + 0 + 1

Hexadecimal 7 D 1

$7\times16^{2}+13\times16^{1}+1\times16^{0}$

1792 + 208 + 1

Figure A-2. The number 2001 in binary, octal, and hexadecimal.

As an example of binary, octal, decimal, and hexadecimal notation, consider Fig. A-3, which shows a collection of nonnegative integers expressed in each of these four different systems. Perhaps some archaeologist thousands of years from now will discover this table and regard it as the Rosetta Stone to late twentieth century and early twenty-first century number systems.

Decimal	Binary	Octal	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	3	3
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Figure A-3. Decimal numbers and their binary, octal, and hexadecimal equivalents.

A.3 CONVERSION FROM ONE RADIX TO ANOTHER

Conversion between octal or hexadecimal numbers and binary numbers is easy. To convert a binary number to octal, divide it into groups of 3 bits, with the 3 bits immediately to the left (or right) of the decimal point (often called a binary point) forming one group, the 3 bits immediately to their left, another group, and so on. Each group of 3 bits can be directly converted to a single octal digit, 0 to 7, according to the conversion given in the first lines of Fig. A-3. It may be necessary to add one or two leading or trailing zeros to fill out a group to 3 full bits. Conversion from octal to binary is equally trivial. Each octal digit is simply replaced by the equivalent 3-bit binary number. Conversion from hexadecimal to

binary is essentially the same as octal-to-binary except that each hexadecimal digit corresponds to a group of 4 bits instead of 3 bits. Figure A-4 gives some examples of conversions.

Example 1

Hexadecimal	1 9 4 8 . B 6
Binary	0001 1001 0100 1000 . 1011 0110 0
Octal	1 4 5 1 0 . 5 5 4

Example 2

Hexadecimal	7 B A 3 . B C 4
Binary	0111 1011 1010 0011 . 1011 1100 0100
Octal	7 5 6 4 3 . 5 7 0 4

Figure A-4. Examples of octal-to-binary and hexadecimal-to-binary conversion.

Conversion of decimal numbers to binary can be done in two different ways. The first method follows directly from the definition of binary numbers. The largest power of 2 smaller than the number is subtracted from the number. The process is then repeated on the difference. Once the number has been decomposed into powers of 2, the binary number can be assembled with 1s in the bit positions corresponding to powers of 2 used in the decomposition, and 0s elsewhere.

The other method (for integers only) consists of dividing the number by 2. The quotient is written directly beneath the original number and the remainder, 0 or 1, is written next to the quotient. The quotient is then considered and the process repeated until the number 0 has been reached. The result of this process will be two columns of numbers, the quotients and the remainders. The binary number can now be read directly from the remainder column starting at the bottom. Figure A-5 gives an example of decimal-to-binary conversion.

Binary integers can also be converted to decimal in two ways. One method consists of summing up the powers of 2 corresponding to the 1 bits in the number. For example,

$$10110 = 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$$

In the other method, the binary number is written vertically, one bit per line, with the leftmost bit on the bottom. The bottom line is called line 1, the one above it line 2, and so on. The decimal number will be built up in a parallel column next to the binary number. Begin by writing a 1 on line 1. The entry on line n consists of two times the entry on line $n - 1$ plus the bit on line n (either 0 or 1). The entry on the top line is the answer. Figure A-6 gives an example of this method of binary to decimal conversion.

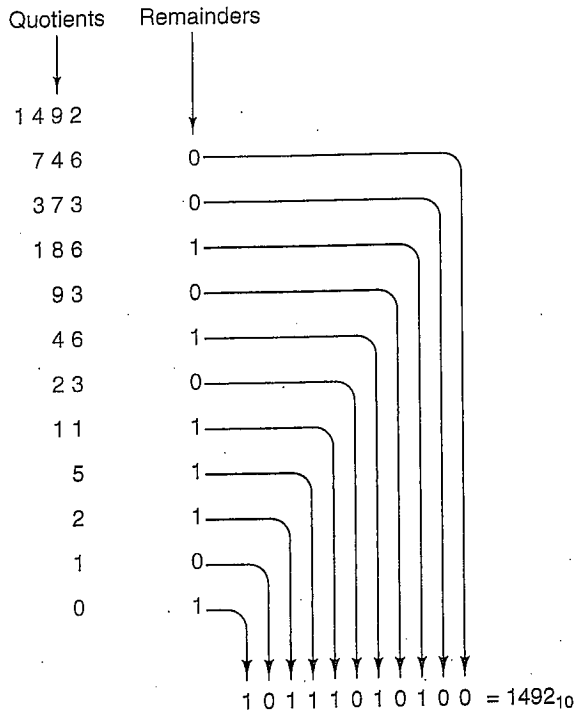


Figure A-5. Conversion of the decimal number 1492 to binary by successive halving, starting at the top and working downward. For example, 93 divided by 2 yields a quotient of 46 and a remainder of 1, written on the line below it.

Decimal-to-octal and decimal-to-hexadecimal conversion can be accomplished either by first converting to binary and then to the desired system or by subtracting powers of 8 or 16.

A.4 NEGATIVE BINARY NUMBERS

Four different systems for representing negative numbers have been used in digital computers at one time or another in history. The first one is called **signed magnitude**. In this system the leftmost bit is the sign bit (0 is + and 1 is −) and the remaining bits hold the absolute magnitude of the number.

The second system, called **one's complement**, also has a sign bit with 0 used for plus and 1 for minus. To negate a number, replace each 1 by a 0 and each 0 by a 1. This holds for the sign bit as well. One's complement is obsolete.

The third system, called **two's complement**, also has a sign bit that is 0 for plus and 1 for minus. Negating a number is a two-step process. First, each 1 is

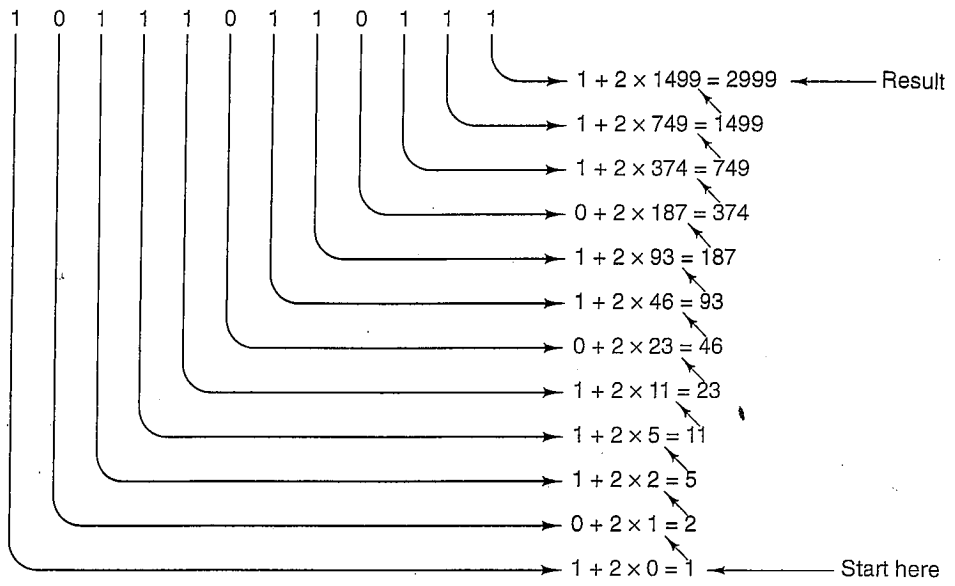


Figure A-6. Conversion of the binary number 101110110111 to decimal by successive doubling, starting at the bottom. Each line is formed by doubling the one below it and adding the corresponding bit. For example, 749 is twice 374 plus the 1 bit on the same line as 749.

replaced by a 0 and each 0 by a 1, just as in one's complement. Second, 1 is added to the result. Binary addition is the same as decimal addition except that a carry is generated if the sum is greater than 1 rather than greater than 9. For example, converting 6 to two's complement is done in two steps:

```
00000110  (+6)
11111001  (–6 in one's complement)
11111010  (–6 in two's complement)
```

If a carry occurs from the leftmost bit, it is thrown away.

The fourth system, which for m-bit numbers is called **excess 2^{m-1}**, represents a number by storing it as the sum of itself and 2^{m-1}. For example, for 8-bit numbers, $m = 8$, the system is called excess 128 and a number is stored as its true value plus 128. Therefore, –3 becomes $-3 + 128 = 125$, and –3 is represented by the 8-bit binary number for 125 (01111101). The numbers from –128 to +127 map onto 0 to 255, all of which are expressible as an 8-bit positive integer. Interestingly enough, this system is identical to two's complement with the sign bit reversed. Figure A-7 gives examples of negative numbers in all four systems.

Both signed magnitude and one's complement have two representations for zero: a plus zero, and a minus zero. This situation is undesirable. The two's complement system does not have this problem because the two's complement of plus

N decimal	N binary	−N signed mag.	−N 1's compl.	−N 2's compl.	−N excess 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Nonexistent	Nonexistent	Nonexistent	10000000	00000000

Figure A-7. Negative 8-bit numbers in four systems.

zero is also plus zero. The two's complement system does, however, have a different singularity. The bit pattern consisting of a 1 followed by all 0s is its own complement. The result is to make the range of positive and negative numbers unsymmetric; there is one negative number with no positive counterpart.

The reason for these problems is not hard to find: we want an encoding system with two properties:

1. Only one representation for zero.
2. Exactly as many positive numbers as negative numbers.

The problem is that any set of numbers with as many positive as negative numbers and only one zero has an odd number of members, whereas m bits allow an even number of bit patterns. There will always be either one bit pattern too many or one bit pattern too few, no matter what representation is chosen. This extra bit

pattern can be used for –0 or for a large negative number, or for something else, but no matter what it is used for it will always be a nuisance.

A.5 BINARY ARITHMETIC

The addition table for binary numbers is given in Fig. A-8.

Addend	0	0	1	1
Augend	+0	+1	+0	+1
Sum	0	1	1	0
Carry	0	0	0	1

Figure A-8. The addition table in binary.

Two binary numbers can be added, starting at the rightmost bit and adding the corresponding bits in the addend and the augend. If a carry is generated, it is carried one position to the left, just as in decimal arithmetic. In one's complement arithmetic, a carry generated by the addition of the leftmost bits is added to the rightmost bit. This process is called an end-around carry. In two's complement arithmetic, a carry generated by the addition of the leftmost bits is merely thrown away. Examples of binary arithmetic are shown in Fig. A-9.

Decimal	1's complement	2's complement
10	00001010	00001010
+ (–3)	11111100	11111101
+7	1 00000110	1 00000111
	carry 1	discarded
	00000111	

Figure A-9. Addition in one's complement and two's complement.

If the addend and the augend are of opposite signs, overflow error cannot occur. If they are of the same sign and the result is of the opposite sign, overflow error has occurred and the answer is wrong. In both one's and two's complement arithmetic, overflow occurs if and only if the carry into the sign bit differs from the carry out of the sign bit. Most computers preserve the carry out of the sign bit, but the carry into the sign bit is not visible from the answer. For this reason, a special overflow bit is usually provided.

B

FLOATING-POINT NUMBERS

In many calculations the range of numbers used is very large. For example, a calculation in astronomy might involve the mass of the electron, 9×10^{-28} grams, and the mass of the sun, 2×10^{33} grams, a range exceeding 10^{60}. These numbers could be represented by

```
000000000000000000000000000000000.0000000000000000000000000009
200000000000000000000000000000000.0000000000000000000000000000
```

and all calculations could be carried out keeping 34 digits to the left of the decimal point and 28 places to the right of it. Doing so would allow 62 significant digits in the results. On a binary computer, multiple-precision arithmetic could be used to provide enough significance. However, the mass of the sun is not even known accurately to five significant digits, let alone 62. In fact few measurements of any kind can (or need) be made accurately to 62 significant digits. Although it would be possible to keep all intermediate results to 62 significant digits and then throw away 50 or 60 of them before printing the final results, doing this is wasteful of both CPU time and memory.

What is needed is a system for representing numbers in which the range of expressible numbers is independent of the number of significant digits. In this appendix, such a system will be discussed. It is based on the scientific notation commonly used in physics, chemistry, and engineering.

B.1 PRINCIPLES OF FLOATING POINT

One way of separating the range from the precision is to express numbers in the familiar scientific notation

$$n = f \times 10^e$$

where f is called the **fraction**, or **mantissa**, and e is a positive or negative integer called the **exponent**. The computer version of this notation is called **floating point**. Some examples of numbers expressed in this form are

$$\begin{array}{lll} 3.14 & = 0.314 \times 10^1 & = 3.14 \times 10^0 \\ 0.000001 & = 0.1 \times 10^{-5} & = 1.0 \times 10^{-6} \\ 1941 & = 0.1941 \times 10^4 & = 1.941 \times 10^3 \end{array}$$

The range is effectively determined by the number of digits in the exponent and the precision is determined by the number of digits in the fraction. Because there is more than one way to represent a given number, one form is usually chosen as the standard. In order to investigate the properties of this method of representing numbers, consider a representation, R, with a signed three-digit fraction in the range $0.1 \le |f| < 1$ or zero and a signed two-digit exponent. These numbers range in magnitude from $+0.100 \times 10^{-99}$ to $+0.999 \times 10^{+99}$, a span of nearly 199 orders of magnitude, yet only five digits and two signs are needed to store a number.

Floating-point numbers can be used to model the real-number system of mathematics, although there are some important differences. Figure B-1 gives a grossly exaggerated schematic of the real number line. The real line is divided up into seven regions:

1. Large negative numbers less than -0.999×10^{99}.
2. Negative numbers between -0.999×10^{99} and -0.100×10^{-99}.
3. Small negative numbers with magnitudes less than 0.100×10^{-99}.
4. Zero.
5. Small positive numbers with magnitudes less than 0.100×10^{-99}.
6. Positive numbers between 0.100×10^{-99} and 0.999×10^{99}.
7. Large positive numbers greater than 0.999×10^{99}.

One major difference between the set of numbers representable with three fraction and two exponent digits and the real numbers is that the former cannot be used to express any numbers in regions 1, 3, 5, or 7. If the result of an arithmetic operation yields a number in regions 1 or 7—for example, $10^{60} \times 10^{60} = 10^{120}$—**overflow error** will occur and the answer will be incorrect. The reason is due to the finite nature of the representation for numbers and is unavoidable. Similarly,

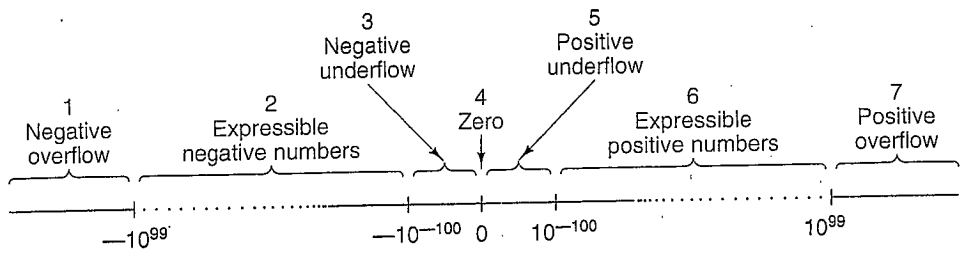


Figure B-1. The real number line can be divided into seven regions.

a result in regions 3 or 5 cannot be expressed either. This situation is called **underflow error**. Underflow error is less serious than overflow error, because 0 is often a satisfactory approximation to numbers in regions 3 and 5. A bank balance of 10^{-102} dollars is hardly better than a bank balance of 0.

Another important difference between floating-point numbers and real numbers is their density. Between any two real numbers, x and y, is another real number, no matter how close x is to y. This property comes from the fact that for any distinct real numbers, x and y, $z = (x + y)/2$ is a real number between them. The real numbers form a continuum.

Floating-point numbers, in contrast, do not form a continuum. Exactly 179,100 positive numbers can be expressed in the five-digit, two-sign system used above, 179,100 negative numbers, and 0 (which can be expressed in many ways), for a total of 358,201 numbers. Of the infinite number of real numbers between -10^{+100} and $+0.999 \times 10^{99}$, only 358,201 of them can be specified by this notation. They are symbolized by the dots in Fig. B-1. It is quite possible for the result of a calculation to be one of the other numbers, even though it is in region 2 or 6. For example, $+0.100 \times 10^{3}$ divided by 3 cannot be expressed *exactly* in our system of representation. If the result of a calculation cannot be expressed in the number representation being used, the obvious thing to do is to use the nearest number that can be expressed. This process is called **rounding**.

The spacing between adjacent expressible numbers is not constant throughout region 2 or 6. The separation between $+0.998 \times 10^{99}$ and $+0.999 \times 10^{99}$ is vastly more than the separation between $+0.998 \times 10^{0}$ and $+0.999 \times 10^{0}$. However, when the separation between a number and its successor is expressed as a percentage of that number, there is no systematic variation throughout region 2 or 6. In other words, the **relative error** introduced by rounding is approximately the same for small numbers as large numbers.

Although the preceding discussion was in terms of a representation system with a three-digit fraction and a two-digit exponent, the conclusions drawn are valid for other representation systems as well. Changing the number of digits in the fraction or exponent merely shifts the boundaries of regions 2 and 6 and changes the number of expressible points in them. Increasing the number of digits in the fraction increases the density of points and therefore improves the accuracy

of approximations. Increasing the number of digits in the exponent increases the size of regions 2 and 6 by shrinking regions 1, 3, 5, and 7. Figure B-2 shows the approximate boundaries of region 6 for floating-point decimal numbers for various sizes of fraction and exponent.

Digits in fraction	Digits in exponent	Lower bound	Upper bound
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Figure B-2. The approximate lower and upper bounds of expressible (unnormalized) floating-point decimal numbers.

A variation of this representation is used in computers. For efficiency, exponentiation is to base 2, 4, 8, or 16 rather than 10, in which case the fraction consists of a string of binary, base-4, octal, or hexadecimal digits. If the leftmost of these digits is zero, all the digits can be shifted one place to the left and the exponent decreased by 1, without changing the value of the number (barring underflow). A fraction with a nonzero leftmost digit is said to be **normalized**.

Normalized numbers are generally preferable to unnormalized numbers, because there is only one normalized form, whereas there are many unnormalized forms. Examples of normalized floating-point numbers are given in Fig. B-3 for two bases of exponentiation. In these examples a 16-bit fraction (including sign bit) and a 7-bit exponent using excess 64 notation are shown. The radix point is to the left of the leftmost fraction bit—that is, to the right of the exponent.

B.2 IEEE FLOATING-POINT STANDARD 754

Until about 1980, each computer manufacturer had its own floating-point format. Needless to say, all were different. Worse yet, some of them actually did arithmetic incorrectly because floating-point arithmetic has some subtleties not obvious to the average hardware designer.

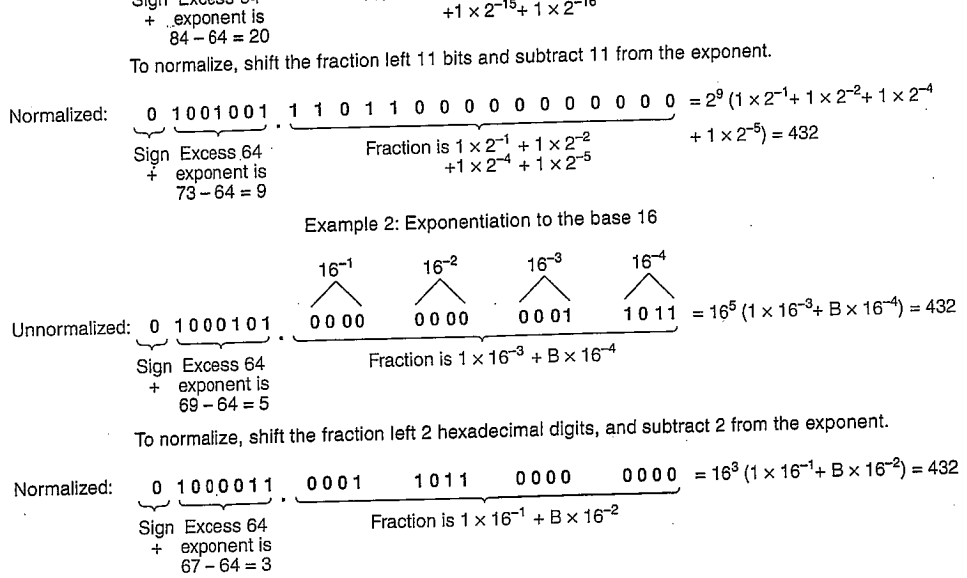


Figure B-3. Examples of normalized floating-point numbers.

To rectify this situation, in the late 1970s IEEE set up a committee to standardize floating-point arithmetic. The goal was not only to permit floating-point data to be exchanged among different computers but also to provide hardware designers with a model known to be correct. The resulting work led to IEEE Standard 754 (IEEE, 1985). Most CPUs these days (including the Intel, SPARC, and JVM ones studied in this book) have floating-point instructions that conform to the IEEE floating-point standard. Unlike many standards, which tend to be wishy-washy compromises that please no one, this one is not bad, in large part because it was primarily the work of one person, Berkeley math professor William Kahan. The standard will be described in the remainder of this section.

The standard defines three formats: single precision (32 bits), double precision (64 bits), and extended precision (80 bits). The extended-precision format is intended to reduce roundoff errors. It is used primarily inside floating-point arithmetic units, so we will not discuss it further. Both the single- and double-precision formats use radix 2 for fractions and excess notation for exponents. The formats are shown in Fig. B-4.

Both formats start with a sign bit for the number as a whole, 0 being positive and 1 being negative. Next comes the exponent, using excess 127 for single

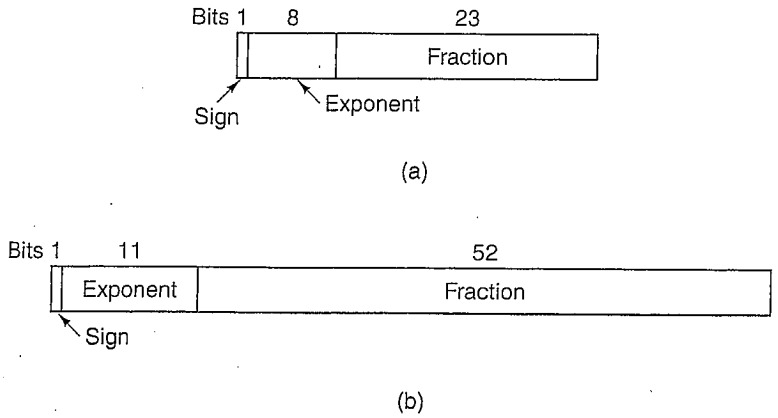


Figure B-4. IEEE floating-point formats. (a) Single precision. (b) Double precision.

precision and excess 1023 for double precision. The minimum (0) and maximum (255 and 2047) exponents are not used for normalized numbers; they have special uses described below. Finally, we have the fractions, 23 and 52 bits, respectively.

A normalized fraction begins with a binary point, followed by a 1 bit, and then the rest of the fraction. Following a practice started on the PDP-11, the authors of the standard realized that the leading 1 bit in the fraction does not have to be stored, since it can just be assumed to be present. Consequently, the standard defines the fraction in a slightly different way than usual. It consists of an implied 1 bit, an implied binary point, and then either 23 or 52 arbitrary bits. If all 23 or 52 fraction bits are 0s, the fraction has the numerical value 1.0; if all of them are 1s, the fraction is numerically slightly less than 2.0. To avoid confusion with a conventional fraction, the combination of the implied 1, the implied binary point, and the 23 or 52 explicit bits is called a **significand** instead of a fraction or mantissa. All normalized numbers have a significand, s, in the range $1 \leq s < 2$.

The numerical characteristics of the IEEE floating-point numbers are given in Fig. B-5. As examples, consider the numbers 0.5, 1, and 1.5 in normalized single-precision format. These are represented in hexadecimal as 3F000000, 3F800000, and 3FC00000, respectively.

One of the traditional problems with floating-point numbers is how to deal with underflow, overflow, and uninitialized numbers. The IEEE standard deals with these problems explicitly, borrowing its approach in part from the CDC 6600. In addition to normalized numbers, the standard has four other numerical types, described below and shown in Fig. B-6.

A problem arises when the result of a calculation has a magnitude smaller than the smallest normalized floating-point number that can be represented in this system. Previously, most hardware took one of two approaches: just set the result to zero and continue, or cause a floating-point underflow trap. Neither of these is

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	−126 to +127	−1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

Figure B-5. Characteristics of IEEE floating-point numbers.

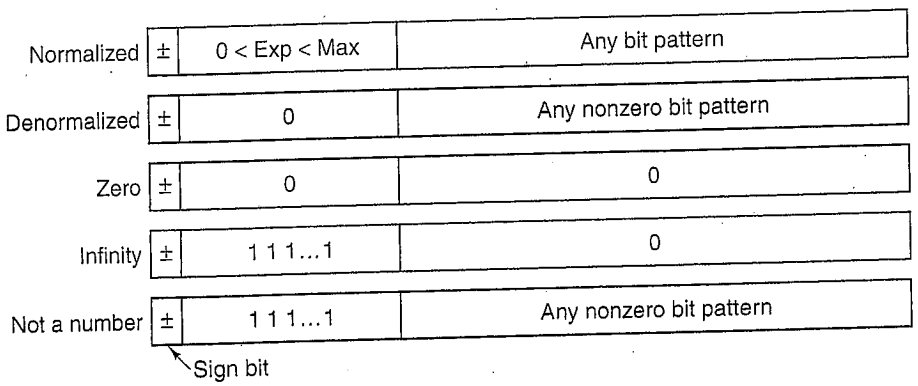


Figure B-6. IEEE numerical types.

really satisfactory, so IEEE invented **denormalized numbers**. These numbers have an exponent of 0 and a fraction given by the following 23 or 52 bits. The implicit 1 bit to the left of the binary point now becomes a 0. Denormalized numbers can be distinguished from normalized ones because the latter are not permitted to have an exponent of 0.

The smallest normalized single precision number has a 1 as exponent and 0 as fraction, and represents 1.0×2^{-126}. The largest denormalized number has a 0 as exponent and all 1s in the fraction, and represents about $0.9999999 \times 2^{-126}$, which is almost the same thing. One thing to note however, is that this number has only 23 bits of significance, versus 24 for all normalized numbers.

As calculations further decrease this result, the exponent stays put at 0, but the first few bits of the fraction become zeros, reducing both the value and the number of significant bits in the fraction. The smallest nonzero denormalized

number consists of a 1 in the rightmost bit, with the rest being 0. The exponent represents 2^{-126} and the fraction represents 2^{-23} so the value is 2^{-149}. This scheme provides for a graceful underflow by giving up significance instead of jumping to 0 when the result cannot be expressed as a normalized number.

Two zeros are present in this scheme, positive and negative, determined by the sign bit. Both have an exponent of 0 and a fraction of 0. Here too, the bit to the left of the binary point is implicitly 0 rather than 1.

Overflow cannot be handled gracefully. There are no bit combinations left. Instead, a special representation is provided for infinity, consisting of an exponent with all 1s (not allowed for normalized numbers), and a fraction of 0. This number can be used as an operand and behaves according to the usual mathematical rules for infinity. For example infinity plus anything is infinity, and any finite number divided by infinity is zero. Similarly, any finite number divided by zero yields infinity.

What about infinity divided by infinity? The result is undefined. To handle this case, another special format is provided, called **NaN (Not a Number)**. It too, can be used as an operand with predictable results.

PROBLEMS

1. Convert the following numbers to IEEE single-precision format. Give the results as eight hexadecimal digits.

   a. 9
   b. 5/32
   c. –5/32
   d. 6.125

2. Convert the following IEEE single-precision floating-point numbers from hex to decimal:

   a. 42E48000H
   b. 3F880000H
   c. 00800000H
   d. C7F00000H

3. The format of single-precision floating-point numbers on the 370 has a 7-bit exponent in the excess 64 system, and a fraction containing 24 bits plus a sign bit, with the binary point at the left end of the fraction. The radix for exponentiation is 16. The order of the fields is sign bit, exponent, fraction. Express the number 7/64 as a normalized number in this system in hex.

4. The following binary floating-point numbers consist of a sign bit, an excess 64, radix 2 exponent, and a 16-bit fraction. Normalize them.

   a. 0 1000000 0001010100000001