arithmetic operations (add, subtract, multiply, and divide), square root, logarithms, and trigonometric functions. Other types of coprocessors may be attached to the 68020 in similar fashion. Later members of the 680X0 family take advantage of advances in VLSI to integrate a floating-point (co)processor into the CPU chip.

Other design features. Like the IBM System/360-370 and the ARM6, the CPU has a supervisor state intended for operating system use and a user state for application programs. As Figures 3.11 and 3.12 indicate, certain "privileged" control registers and instructions can be used only in the supervisor state. User and supervisory programs are thus clearly separated—for example, they employ different stack pointers—thereby improving system security. 680X0-based computers are also designed to allow easy implementation of *virtual memory*, whereby the operating system makes the main memory appear larger to user programs than it really is. Hardware support for virtual memory is provided by the 68851 memory management unit (MMU), another 680X0 coprocessor.

Provided they meet certain independence conditions, up to three 68020 instructions can be processed simultaneously in pipeline fashion. This pipelining is complicated by the fact that instruction lengths and execution times vary, a problem that RISCs try to eliminate. Another speedup feature found in the 68020 is a small instruction-only cache (I-cache). The 68020 prefetches instructions from main memory while the system bus is idle; the instructions can subsequently be read much more quickly from the on-chip cache than from the off-chip main memory. An unusual feature of the 68020 noted in Figure 3.11 is its use of two levels of microprogramming to implement the CPU's control logic. For the manufacturer, this feature increases design flexibility while reducing IC area compared with conventional (one-level) microprogrammed control.

3.2 DATA REPRESENTATION

The basic items of information handled by a computer are instructions and data. We now examine the methods used to represent such information, focusing on the formats for numerical data.

3.2.1 Basic Formats

Figure 3.15 shows the fundamental division of information into instructions (operation or control words) and data (operands). Data can be further subdivided into numerical and nonnumerical. In view of the importance of numerical computation, computer designs have paid a great deal of attention to the representation of numbers. Two main number formats have evolved: fixed-point and floating-point. The binary fixed-point format takes the form $b_A b_B b_C \ldots b_K$, where each b_i is 0 or 1 and a binary point is present in some fixed but implicit position. A floating-point number, on the other hand, consists of a pair of fixed-point numbers M,E, which denote the number $M \times B^E$, where B is a predetermined base. The many formats used to encode fixed-point and floating-point numbers will be examined later in

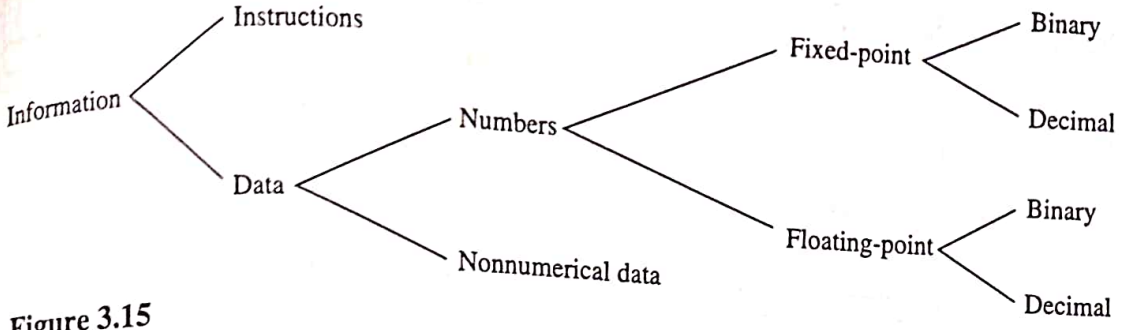


Figure 3.15
The basic information types.

the chapter. Nonnumerical data usually take the form of variable-length character strings encoded in one of several standard codes, such as ASCII (American Standards Committee on Information Exchange) code.

Word length. Information is represented in a digital computer by means of binary words, where a *word* is a unit of information of some fixed length n. An n-bit word allows up to 2^n different items to be represented. For example, with $n = 4$, we can encode the 10 decimal digits as follows:

$$\begin{array}{lllll} 0 = 0000 & 1 = 0001 & 2 = 0010 & 3 = 0011 & 4 = 0100 \\ 5 = 0101 & 6 = 0110 & 7 = 0111 & 8 = 1000 & 9 = 1001 \end{array} \qquad (3.7)$$

To encode alphanumeric symbols or *characters,* 8-bit words called *bytes* are commonly used. As well as being able to encode all the standard keyboard symbols, a byte allows efficient representation of decimal numbers that are encoded in binary according to (3.7). A byte can store two decimal digits with no wasted space. Most computers have the 8-bit byte as the smallest addressable unit of information in their main memories. The CPU also has a standard word size for the data it processes. Word size is typically a multiple of 8, common CPU word sizes being 8, 16, 32, and 64 bits.

No single word length is suitable for representing every kind of information encountered in a typical computer. Even within a single domain such as a computer's instruction set, we often find several different word sizes. For example, instructions such as load and store that reference memory need long address fields. Instructions whose operands are all in the CPU need not contain memory addresses and so can be shorter. The precision of a number word is determined by its length; it is common therefore to have numbers of various sizes. Figure 3.16 gives a sampling of data sizes used by the Motorola 680X0. As here, the term *word* is often restricted to mean a 32-bit (4 byte) word. (680X0 literature refers to 32-bit words with the nonstandard term *long word.*) Fixed-point numbers come in lengths of 1, 2, 4, or more bytes. Floating-point numbers also come in several lengths, the shortest (single precision) number being one word (32 bits) long.

The circuits of a CPU must be carefully designed to permit various information formats to coexist smoothly. For example, if instruction length varies, as is the case in many CISC microprocessors, the program control unit must be designed to determine an instruction's length from its opcode and to fetch a variable number of instruction bytes from memory. It must also increment the program counter by a

Bits	Name	Illustration	Typical uses
1	Bit		Status flag. Logic variable.
8	Byte		Smallest addressable memory item. Binary-coded decimal digit pair.
16	Halfword		Short fixed-point number. Short address (offset). Short instruction.
32	Word		Fixed- or floating-point number. Memory address. Instruction.
64	Double word		Long instruction. Double-precision floating-point number.

Figure 3.16
Some information formats of the Motorola 680X0 microprocessor series.

variable amount to obtain the address of the next consecutive instruction. Thus while the ARM6 has instructions of length 4 bytes only, the 68020's instructions range in length from 2 to 10 bytes.

Instruction sets commonly have features to make it easy to apply instructions to nonstandard-length operands. An example is the add-with-carry (ADC) instruction and its counterpart subtract with carry, which enable add and subtract instructions to apply to long fixed-point numbers by adding them in short segments and propagating carries from segment to segment. Suppose, for example, that we want to add two unsigned 64-bit (double word) binary integers A and B using the ARM6 instruction set (Figure 3.10), which is designed to add 32-bit words. Let A be placed in registers R0 and R1, with the right (least significant) half of A in R0. Similarly, let B be placed in registers R2 and R3, with its right half in R2. We first apply the ADD instruction with inputs R0 and R2 and place the resulting sum in R4. We also instruct ADD to activate the status flags, which requires an *S* suffix to the ARM6 opcode, changing it to ADDS. (In most other computers the flags are set automatically by all data-processing instructions.) ADDS results in the carry flag C assuming the value of the carry-out bit produced by the addition R0 + R2. Then we apply the ADC (add with carry) instruction with inputs R1 and R3 to compute the sum R1 + R3. In the following ARM6 code, the final sum $A + B$ is placed in R4 and R5.

HDL format	ARM6 assembly-language format		Narrative format (comment)
C.R4 := R0 + R2	ADDS	R4,R0,R2	Add right words and store carry signal C.
R5 : = R1 + R3 + C	ADC	R5,R1,R3	Add left words plus C.

Storage order. A small but important aspect of data representation is the way in which the bits of a word are indexed. We will usually follow the convention illustrated in Figure 3.17, where the right-most bit is assigned the index 0 and the bits are labeled in increasing order from right to left. The advantage of this convention is that when the word is interpreted as an unsigned binary integer, the low-order indexes correspond to the numerically less significant bits and the high-order indexes correspond to the numerically more significant bits. Similarly, we label the

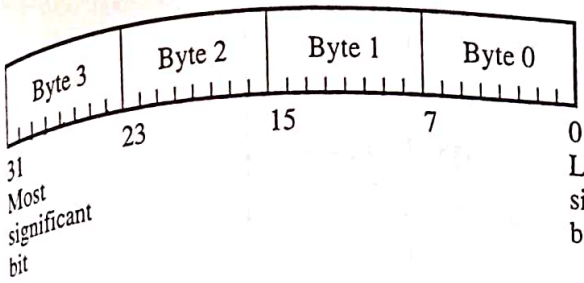


Figure 3.17
Indexing convention for the bits and bytes of a word.

bytes of a word from right to left, with index 0 assigned to the numerically least significant byte. Figure 3.17 therefore shows the format used to store a 4-byte word in a one-word register.

Since words are stored as individually addressable bytes in memory M, a question arises as to the storage order in M of the bytes within each word. Suppose that a sequence $W_0,W_1,\ldots,W_m$ of m 4-byte number words is to be stored. Suppose further that we write W_i as $B_{i,3},B_{i,2},B_{i,1},B_{i,0}$, where as in Figure 3.17, we place the least significant byte $B_{i,0}$ on the right and assign it the lowest index 0. Now the entire sequence can be rewritten as

$$W_0,W_1,\ldots,W_m = B_{0,3},B_{0,2},B_{0,1},B_{0,0},B_{1,3},B_{1,2},B_{1,1},B_{1,0},\ldots, B_{m,3},B_{m,2},B_{m,1},B_{m,0} \tag{3.8}$$

Suppose we store these $4(m + 1)$ bytes in M using the "natural" order defined by (3.8); that is, we assign a sequence of increasing memory addresses

$$adr_0,\ adr_1,\ adr_2,\ adr_3,\ \ldots,\ adr_{4m+2},\ adr_{4m+3}$$

to the bytes as listed in (3.8). This storage sequence, which is illustrated in Figure 3.18*a*, is a byte-storage convention called *big-endian*.[2] It is so named because the most significant (biggest) byte $B_{i,3}$ of word W_i is assigned the lowest address and the least significant byte $B_{i,0}$ is assigned the highest address. In other words, the big-endian scheme assigns the highest address to byte 0. The alternative byte-storage scheme called *little-endian* assigns the lowest address to byte 0. This corresponds to

$$W_0,W_1,\ldots,W_m = B_{0,0},B_{0,1},B_{0,2},B_{0,3},B_{1,0},B_{1,1},B_{1,2},B_{1,3},\ldots,B_{m,0},B_{m,1},B_{m,2},B_{m,3}$$

and is illustrated by Figure 3.18*b*.

Interestingly, computer manufacturers have never agreed on this issue, so both the big-endian and little-endian conventions are in widespread use. For example, the Motorola 680X0 uses the big-endian method, whereas the Intel 80X86 series is little-endian. Some computers including the ARM family can switch between the two endian conventions.

Tags. In the von Neumann computer, instruction and data words are stored together in main memory and are indistinguishable from one another—this is the classic "stored program" concept. An item plucked at random from memory cannot be identified as an instruction or data. Different data types such as fixed-point and floating-point numbers also cannot be distinguished by inspection. A word's type is determined by the way a processor interprets it. In principle, the same word can be treated as an instruction and data at different times, for example, the word X in

[2] The allusion is to an argument appearing in *Gulliver's Travels* on whether an egg should be opened at its big or little end [Cohen 1981].

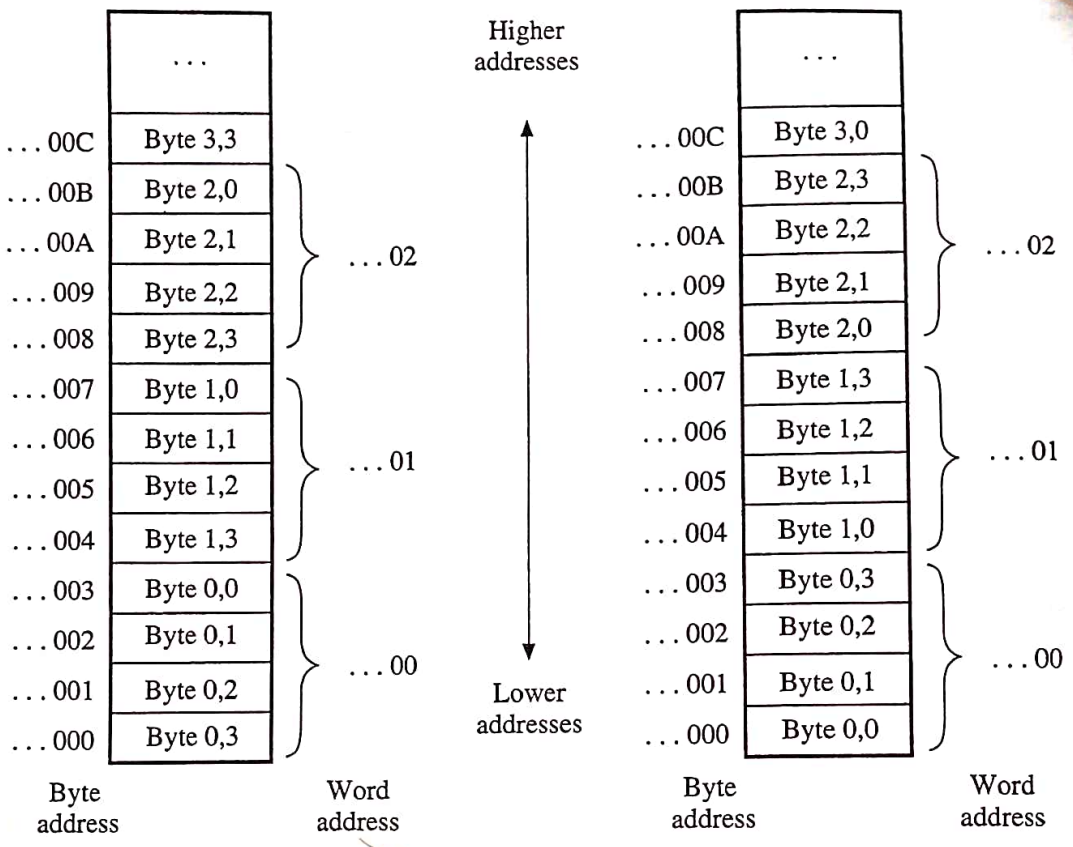


Figure 3.18
Basic byte storage methods: (*a*) big-endian and (*b*) little-endian.

the instruction sequence

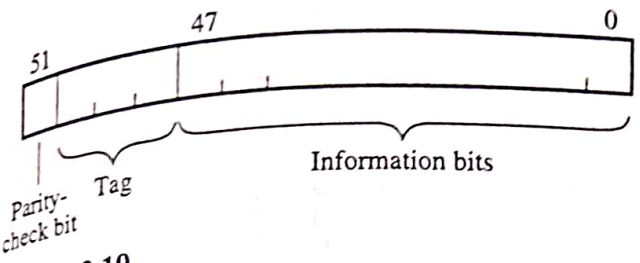
$$X := X + Y;$$

go to X;

It is the programmer's (and compiler's) responsibility to ensure that data are not interpreted as instructions, and vice versa. A reason for this deliberate indistinguishability of data and instructions can be seen in the design of the IAS computer (section 1.2.2). The IAS's address-modify instructions alter stored instructions in main memory. The ability to modify instructions in this way—in effect, treating them as data—is useful when processing indexed variables, as illustrated in Example 1.4. However, this type of instruction modification in memory became obsolete with the introduction of address-indexing hardware.

A few computer designers have argued that the major information types should be assigned formats that identify them [Feustel 1973; Myers 1982]. This can be done by associating with each information word a group of bits, called a *tag*, that identifies the word's type. The tag may be considered as a physical implementation of the **type** declaration found in some high-level programming languages. One of the earliest machines to use tags was the 1960s-vintage Burroughs B6500/7500 series, which employed a 3-bit tag field in every word so that eight word types could be distinguished. The 52-bit word format of the B6500/7500 and the interpretation of its tag appear in Figure 3.19.

Tagging simplifies instruction specification. In conventional, nontagged computers, an instruction's opcode must explicitly or implicitly specify the type of data on which it operates. The PCU must know the operand types in order to route them



Tag	Interpretation
000	Single-precision number.
001	Indirect reference word.
010	Double-precision number.
011	Segment descriptor.
100	Step-index control word.
101	Data descriptor.
110	Uninitialized operand.
111	Instruction.

Figure 3.19
Tagged-word format of the Burroughs B6500/7500 series.

to the proper arithmetic circuits and registers. It is therefore necessary to provide distinct instructions for each data type; for example, add binary word, add binary half-word, add BCD word, add floating-point word, and add floating-point double word. If, on the other hand, tags distinguish the operand types, then a single ADD opcode suffices for all cases. The processor merely has to inspect an operand's tag to determine its type. Furthermore, tag inspection permits the hardware to check for software errors, such as an attempt to add operands whose types are incompatible. Tags have a serious cost disadvantage, however. They increase memory size and add to the system hardware costs without increasing computing performance. This fact has severely restricted the use of tagged architectures.

Error detection and correction. Various factors like manufacturing defects and environmental effects cause errors in computation. Such errors frequently appear when information is being transmitted between two relatively distant points within a computer or is being stored in a memory unit. "Noise" in the communication link can corrupt a bit x that is being sent from A to B so that B receives $\bar{x}$ instead of x. To guard against errors of this type, the information can be encoded so that special logic circuits can detect, and possibly even correct, the errors.

A general way to detect or correct errors is to append special check bits to every word. One popular technique employs a single check bit c_0 called a *parity bit.* The parity bit is appended to an n-bit word $X = (x_0, x_1, \ldots, x_{n-1})$ to form the $(n+1)$-bit word $X^* = (x_0, x_1, \ldots, x_{n-1}, c_0)$; see Figure 3.19. Bit c_0 is assigned the value 0 or 1 that makes the number of ones in X^* even, in the case of *even-parity* codes, or odd, in the case of *odd-parity* codes. In the even-parity case, c_0 is defined by the logic equation

$$c_0 = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1} \tag{3.9}$$

where $\oplus$ denotes EXCLUSIVE-OR, while in the odd-parity case

$$\bar{c}_0 = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}$$

Suppose that the information X is to be transmitted from A to B. The value of c_0 is generated at the source point A using, say, (3.9), and X^* is sent to B. Let B receive the word $X' = (x'_0, x'_1, \ldots, x'_{n-1}, c'_0)$. B then determines the parity of the received word by recomputing the parity bit according to (3.9) thus:

$$c^*_0 = x'_0 \oplus x'_1 \oplus \ldots \oplus x'_{n-1}$$

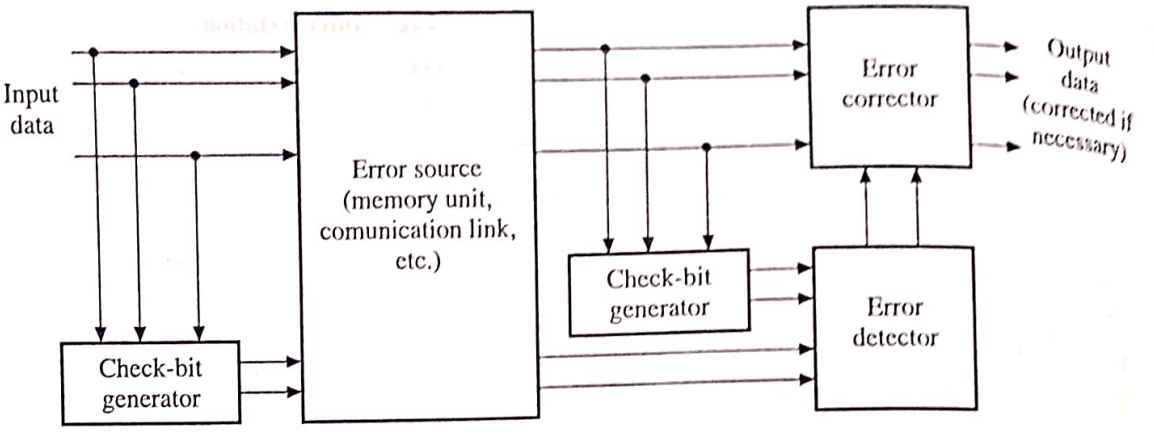


Figure 3.20
Error detection and correction logic.

The received parity bit c'_0 and the reconstituted parity bit c^*_0 are then compared. If $c'_0 \neq c^*_0$, the received information contains an error. In particular, if exactly 1 bit of X^* has been inverted during the transmission process (a single-bit error), then $c'_0 \neq c^*_0$. If $c'_0 = c^*_0$, it can be concluded that no single-bit error occurred, but the possibility of multiple-bit errors is not ruled out. For example, if a 0 changes to 1 and a 1 changes to 0 (a double error), then the parity of X' is the same as that of X^* and the error will go undetected. The parity bit c_0 therefore provides *single-error detection.* It does not detect all multiple errors, much less provide any information about the location of the erroneous bits.

The parity-checking concept can be extended to the detection of multiple errors or to the location of single or multiple errors. These goals are achieved by providing additional parity bits, each of which checks the parity of some subset of the bits in the word X^*. By appropriately overlapping these subsets, the correctness of every bit can be determined. Suppose, for instance, that we can deduce from the parity checks the identity of the bit x_i responsible for a single-bit error. It is then a simple matter to introduce logic circuits to replace x_i by $\bar{x}_i$, thus providing *single-error correction.* Let c be the number of check bits required to achieve single-error correction with n-bit data words. Clearly the check bits have 2^c patterns that must distinguish between $n + c$ possible error locations and the single error-free case. Hence c must satisfy the inequality

$$2^c \geq n + c + 1 \tag{3.10}$$

For $n = 16$, (3.10) implies that $c \geq 5$, while for $n = 32$ we have $c \geq 6$. A variety of practical single-error-correcting parity-check codes meet the lower bound on c implied by (3.10) [Siewiorek and Swarz 1992]. Some of these codes can also detect double errors and so are called *single-error-correcting double-error-detecting* (SECDED) codes. As the main memories of computers have increased in storage capacity and decreased in physical size, they have become more prone to transient failures that are often correctable via SECDED codes. Figure 3.20 shows the structure of a typical error detection and correction scheme used with a computer's main memory.

3.2.2 Fixed-Point Numbers

In selecting a number representation to be used in a computer, the following factors should be taken into account:

- The number types to be represented; for example, integers or real numbers.
- The range of values (number magnitudes) likely to be encountered.
- The precision of the numbers, which refers to the maximum accuracy of the representation.
- The cost of the hardware required to store and process the numbers.

The two principal number formats are fixed-point and floating-point. Fixed-point formats allow a limited range of values and have relatively simple hardware requirements. Floating-point numbers, on the other hand, allow a much larger range of values but require either costly processing hardware or lengthy software implementations.

Binary numbers. The fixed-point format is derived directly from the ordinary (decimal) representation of a number as a sequence of digits separated by a decimal point. The digits to the left of the decimal point represent an integer; the digits to the right represent a fraction. This is *positional notation* in which each digit has a fixed *weight* according to its position relative to the decimal point. If $i > 1$, the ith digit to the left (right) of the decimal point has weight 10^{i-1} (10^{-i}). Thus the five-digit decimal number 192.73 is equivalent to

$$1 \times 10^2 + 9 \times 10^1 + 2 \times 10^0 + 7 \times 10^{-1} + 3 \times 10^{-2}$$

More generally, we can assign weights of the form r^i, where r is the *base* or *radix* of the number system, to each digit.

The most fundamental number representation used in computers employs a base-two positional notation. A binary word of the form

$$b_N \ldots b_3 b_2 b_1 b_0 \,.\, b_{-1} b_{-2} b_{-3} b_{-4} \ldots b_M \tag{3.11}$$

represents the number

$$\sum_{i=M}^{N} b_i 2^i$$

When unclear from the context, the base r being used will be indicated by appending r as a subscript to the number. Thus 1010_2 denotes the binary equivalent of the decimal number 10_{10}, whereas 10_2 denotes 2_{10}. The format of (3.11) is an example of a fixed-point binary number and is used to denote unsigned numbers. Several distinct methods used for representing signed (positive and negative) numbers are discussed below.

Suppose that an n-bit word is to contain a signed binary number. One bit is reserved to represent the sign of the number, while the remaining bits indicate its magnitude. To permit uniform processing of all n bits, the sign is placed in the left-most position, and 0 and 1 are used to denote plus and minus, respectively. This

leads to the format

$$\underbrace{x_{n-1}}_{\text{Sign}}\underbrace{x_{n-2}x_{n-3}\cdots x_2x_1x_0}_{\text{Magnitude}} \tag{3.12}$$

The precision allowed by this format is $n-1$ bits, which is equivalent to $(n-1)\log_2 10$ decimal digits. The binary point is not explicitly represented; instead, it is implicitly assigned to some fixed location in the word. The binary point's position is not very important from the point of view of design. In many situations the numbers being processed are integers, so the binary point is assumed to lie immediately to the right of the least significant bit x_0. Monetary quantities are often expressed as integers; for instance, \$54.30 might be expressed as 5430 cents. Using an n-bit integer format, we can represent all integers N with magnitude $|N|$ in the range $0 \le |N| \le 2^n - 1$. The other most widely used fixed-point format treats (3.12) as a fraction with the binary point lying between x_{n-1} and x_{n-2}. The fraction format denotes numbers with magnitudes in the range $0 \le |N| \le 1 - 2^{-n}$.

Signed numbers. Suppose that both positive and negative binary numbers are to be represented by an n-bit word $X = x_{n-1}x_{n-2}x_{n-3}\ldots x_2x_1x_0$. The standard format for positive numbers is given by (3.12) with a sign bit of 0 on the left and the magnitude to the right in the usual positional notation. This means that each magnitude bit x_i, $0 \le i \le n-2$, has a fixed weight of the form 2^{k+i}, where k depends on the position of the binary point. A natural way to represent negative numbers is to employ the same positional notation for the magnitude and simply change the sign bit x_{n-1} to 1 to indicate minus. Thus with $n = 8$, $+75 = 01001011$, while $-75 = 11001011$. This number code is called *sign magnitude*. Note that humans normally use decimal versions of sign-magnitude code. Nevertheless, operations like subtraction are costly to implement by logic circuits when sign-magnitude codes are used. However, multiplication and division of sign-magnitude numbers is almost as easy as the corresponding operation for unsigned numbers, as Example 2.7 (section 2.3.3) shows.

Several number codes have been devised that use the same representation for positive numbers as the sign-magnitude code but represent negative numbers in different ways. For example, in the *ones-complement* code, $-X$ is denoted by $\bar{X}$, the bitwise logical complement of X. In this code we again have $+75 = 01001011$, but now $-75 = 10110100$. In the *twos-complement* code, $-X$ is formed by adding 1 to the least significant bit of $\bar{X}$ and ignoring any carry bit generated from the most significant (sign) position. If $X = x_{n-1}x_{n-2}\ldots x_0$ is an n-bit binary fraction, $-X$ can be expressed as follows:

$$-X = \underbrace{\bar{x}_{n-1}}_{\text{Implicit binary point}}.\bar{x}_{n-2}\bar{x}_{n-3}\ldots\bar{x}_1\bar{x}_0 + \underbrace{0.00\ldots01}_{\text{Implicit binary point}} \quad \text{(modulo 2)} \tag{3.13}$$

where the use of modulo-2 addition corresponds to ignoring carries from the sign position. If X is an integer, then (3.13) becomes

$$-X = \underbrace{\bar{x}_{n-1}}_{\text{Implicit binary point}}\bar{x}_{n-2}\bar{x}_{n-3}\ldots\bar{x}_1\bar{x}_0. + \underbrace{000\ldots01.}_{\text{Implicit binary point}} \quad \text{(modulo } 2^n\text{)} \tag{3.14}$$

For example, in twos-complement code +75 = 01001011 and –75 = 10110101. Note that in both complement codes x_{n-1} retains its role as the sign bit, but the remaining bits no longer form a simple positional code when the number is negative.

The primary advantage of the complement codes is that subtraction can be performed by logical complementation and addition only. Consider the twos-complement code. To subtract X from Y, just add $-X$ to Y, where $-X$ is obtained by logical complementation and addition of a 1 bit, as in (3.13) and (3.14). As we will see later, the sign bits do not require special treatment; consequently, twos-complement addition and subtraction can be implemented by a simple adder designed for unsigned numbers. Multiplication and division are more difficult to implement if twos-complement code is used instead of sign magnitude. The addition of ones-complement numbers is complicated by the fact that a carry bit from the most significant magnitude bit x_{n-2} must be added to the least significant bit position x_0. Otherwise ones-complement codes are quite similar to twos-complement codes and so will not be considered further.

Figure 3.21 illustrates how integers are represented using all three codes when $n = 4$. These codes are all referred to as *binary* codes to distinguish them from the so-called decimal codes discussed below. Observe that in all cases, 0000 represents zero. Only in the case of twos-complement code, however, is the nega-

Decimal representation	Binary code		
	Sign magnitude	Ones complement	Twos complement
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
–0	1000	1111	0000
–1	1001	1110	1111
–2	1010	1101	1110
–3	1011	1100	1101
–4	1100	1011	1100
–5	1101	1010	1011
–6	1110	1001	1010
–7	1111	1000	1001

Figure 3.21
Comparison of three 4-bit codes for signed binary numbers.

tive (numerical complement) of 0000 also 0000. This unique representation of zero is a significant advantage, for example, in implementing instructions like BNE in Figure 3.13 that test for zero. Consequently, twos-complement code is by far the most popular code for representing signed binary numbers in computers.

Exceptional conditions. If the result of an arithmetic operation involving n-bit numbers is too large (small) to be represented by n bits, *overflow (underflow)* is said to occur. It is generally necessary to detect overflow and underflow, since they may indicate bad data or a programming error. Consider, for example, the addition operation

$$z_{n-1}z_{n-2}\ldots z_0 := x_{n-1}x_{n-2}\ldots x_0 + y_{n-1}y_{n-2}\ldots y_0$$

using n-bit twos-complement operands. Assume that bitwise addition is performed with a carry bit c_i generated by the addition of x_i, y_i, and c_{i-1}. The output bits z_i and c_i can be computed according to the full-adder logic equations

$$z_i = x_i \oplus y_i \oplus c_{i-1}$$

$$c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$$

Let v be a binary variable indicating overflow when $v = 1$. Figure 3.22 shows how the sign bit z_{n-1} and v are determined as functions of the sign bits x_{n-1}, y_{n-1} and the carry bit c_{n-2}. The overflow indicator v is therefore defined by the logic equation

$$v = \bar{x}_{n-1}\bar{y}_{n-1}c_{n-2} + x_{n-1}y_{n-1}\bar{c}_{n-2}$$

If the combinations $(x_{n-1},y_{n-1},c_{n-2}) = (0,0,1)$ and $(1,1,0)$, which make $v = 1$, are removed from the truth table of Figure 3.22, then z_{n-1} is defined correctly for all the remaining combinations by the equation

$$z_{n-1} = x_{n-1} \oplus y_{n-1} \oplus c_{n-2}$$

Consequently, during twos-complement addition the sign bits of the operands can be treated in the same way as the remaining (magnitude) bits.

A related issue in computer arithmetic is *round-off error,* which results from the fact that every number must be represented by a limited number of bits. An

Inputs			Outputs	
x_{n-1}	y_{n-1}	c_{n-2}	z_{n-1}	v
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0

Figure 3.22
Computation of the sign bit z_{n-1} and the overflow indicator v in twos-complement addition.

operation involving n-bit numbers frequently produces a result of more than n bits. For example, the product of two n-bit numbers contains up to $2n$ bits, all but n of which must normally be discarded. Retaining the n most significant bits of the result without modification is called *truncation*. Clearly the resulting number is in error by the amount of the discarded digits. This error can be reduced by a process called *rounding*. One way of rounding is to add $r^j/2$ to the number before truncation, where r^j is the weight of the least significant retained digit. For instance, to round 0.346712 to three decimal places, add 0.0005 to obtain 0.347212 and then take the three most significant digits 0.347. Simple truncation yields the less accurate value 0.346. Successive computations can cause round-off errors to build up unless countermeasures are taken. The number formats provided in a computer should have sufficient precision that round-off errors are of no consequence to most users. It is also desirable to provide facilities for performing arithmetic to a higher degree of precision if required. Such high precision is usually achieved by using several words to represent a single number and writing special subroutines to perform multiword, or *multiple-precision*, arithmetic.

Decimal numbers. Since humans use decimal arithmetic, numbers being entered into a computer must first be converted from decimal to some binary representation. Similarly, binary-to-decimal conversion is a normal part of the computer's output processes. In certain applications the number of decimal-binary conversions forms a large fraction of the total number of elementary operations performed by the computer. In such cases, number conversion should be carried out rapidly. The various binary number codes discussed above do not lend themselves to rapid conversion. For example, converting an unsigned binary number $x_{n-1}x_{n-2}\ldots x_0$ to decimal requires a polynomial of the form

$$\sum_{i=0}^{n-1} x_i 2^{k+i}$$

to be evaluated.

Several number codes exist that facilitate rapid binary-decimal conversion by encoding each decimal digit separately by a sequence of bits. Codes of this kind are called *decimal codes*. The most widely used decimal code is the BCD (*binary-coded decimal*) code. In BCD format each digit d_i of a decimal number is denoted by its 4-bit equivalent $b_{i,3}b_{i,2}b_{i,1}b_{i,0}$ in standard binary form, as in (3.7). Thus the BCD number representing 971 is 100101110001. BCD is a weighted (positional) number code, since $b_{i,j}$ has the weight 10^i2^j. Signed BCD numbers employ decimal versions of the sign-magnitude or complement formats. The 8-bit ASCII code represents the 10 decimal digits by a 4-bit BCD field; the remaining 4 bits of the ASCII code word have no numerical significance.

Two other decimal codes of moderate importance are shown in Figure 3.23. The *excess-three* code can be formed by adding 0011_2 to the corresponding BCD number—hence its name. The advantage of the excess-three code is that it may be processed using the same logic used for binary codes. If two excess-three numbers are added like binary numbers, the required decimal carry is automatically generated from the high-order bits. The sum must be corrected by adding +3. For

Decimal digit	Decimal code			
	BCD	ASCII	Excess-three	Two-out-of-five
0	0000	0011 0000	0011	11000
1	0001	0011 0001	0100	00011
2	0010	0011 0010	0101	00101
3	0011	0011 0011	0110	00110
4	0100	0011 0100	0111	01001
5	0101	0011 0101	1000	01010
6	0110	0011 0110	1001	01100
7	0111	0011 0111	1010	10001
8	1000	0011 1000	1011	10010
9	1001	0011 1001	1100	10100

Figure 3.23
Some important decimal number codes.

example, consider the addition 5 + 9 = 14 using excess-three code.

$$\begin{array}{rrll} & & 1000 = 5 & \\ & + & 1100 = 9 & \\ \hline \text{Carry } 1 & \leftarrow & 0100 & \text{Binary sum} \\ & + & 0011 & \text{Correction} \\ \hline & & 0111 = 4 & \text{Excess-three sum} \end{array}$$

Binary addition of the BCD representations of 5 and 9 results in 1110 and no carry generation. (The binary sum of two BCD numbers can also be corrected to give the proper BCD sum as described later.) Some arithmetic operations are difficult to implement using excess-three code, mainly because it is a *nonweighted* code; that is, each bit position in an excess-three number does not have a fixed weight.

The final decimal code illustrated by Figure 3.23 is the *two-out-of-five* code. Each decimal digit is represented by a 5-bit sequence containing two 1s and three 0s; there are exactly 10 distinct sequences of this type. The particular merit of the two-out-of-five code is that it is single-error detecting, since changing any one bit results in a sequence that does not correspond to a valid code word. Its drawbacks are that it is a nonweighted code and uses 5 rather than 4 bits per decimal digit.

The main advantage of the decimal codes is ease of conversion between the internal computer representation that allows only the symbols 0, 1 and external representations using the 10 decimal symbols 0, 1, 2,..., 9. Decimal codes have two disadvantages.

1. They use more bits to represent a number than the binary codes. Decimal codes therefore require more memory space. An n-bit word can represent 2^n numbers using binary codes; approximately $10^{n/4} = 2^{0.830n}$ numbers can be represented if a 4-bit decimal code such as BCD or excess-three is used.
2. The circuitry required to perform arithmetic using decimal operands is more complex than that needed for binary arithmetic. For example, in adding BCD

numbers bit by bit, a uniform method of propagating carries between adjacent positions is not possible, since the weights of adjacent bits do not differ by a constant factor.

Hexadecimal numbers. One or two other numerical codes are encountered in the design or use of computers. Of particular importance is *hexadecimal* (hex) code, which is characterized by a base $r = 16$ and the use of 16 digits, consisting of the decimal digits 0,1,...,9 augmented by the six digits A,B,C,D,E, and F, which have the numerical values 10, 11, 12, 13, 14, and 15, respectively. The unsigned hexadecimal integer 2FA0C has the interpretation

$$\begin{aligned} &2 \times 16^4 + \mathrm{F} \times 16^3 + \mathrm{A} \times 16^2 + 0 \times 16^1 + \mathrm{C} \times 16^0 \\ &\quad = 2 \times 65{,}536 + 15 \times 4{,}096 + 10 \times 256 + 0 \times 16 + 12 \times 1 \\ &\quad = 195{,}084 \end{aligned}$$

Hence $2\mathrm{FA0C}_{16} = 195{,}084_{10}$.

Hexadecimal code is useful for representing long binary numbers, a consequence of the fact that the base 16 is a power of two. A hexadecimal number is converted to binary simply by replacing each hex digit by the equivalent 4-bit binary form. For example, we can convert $2\mathrm{FA0C}_{16}$ to binary by replacing the first digit 2 by 0010, the second digit F by 1111, the third digit A by 1010, and so on, yielding

$$2\mathrm{FA0C}_{16} = 00101111101000001100_2$$

Conversely, we can convert a binary number to hex form by replacing each four-digit group by the corresponding hex digit. Clearly hexadecimal-binary number conversion is very similar to BCD-binary conversion. By treating any binary word as an unsigned integer, we can easily convert the word to hex form as indicated above. Hex code provides a very convenient shorthand for binary information.

3.2.3 Floating-Point Numbers

The range of numbers that can be represented by a fixed-point number code is insufficient for many applications, particularly scientific computations where very large and very small numbers are encountered. Scientific notation permits us to represent such numbers using relatively few digits. For example, it is easier to write a quintillion as

$$1.0 \times 10^{18} \tag{3.15}$$

than as the 19-bit, fixed-point integer 1 000 000 000 000 000 000. The floating-point codes used in computers are binary (or binary-coded) versions of (3.15).

Basic formats. Three numbers are associated with a floating-point number: a *mantissa* M, an *exponent* E, and a *base* B. The mantissa M is also referred to as the *significand* or *fraction* in the literature. These three components together represent the real number $M \times B^E$. For example, in (3.15) 1.0 is the mantissa, 18 is the exponent, and 10 is the base. For machine implementation the mantissa and exponent are encoded as fixed-point numbers with a base r that is usually 2 or 10. The base B

is also r, or some power of r, for reasons that will become obvious. Since B is a constant, it need not be included in the number code; it is simply built into the circuits that process the numbers. A floating-point number is therefore stored as a word (M,E) consisting of a pair of signed fixed-point numbers: a mantissa M, which is usually a fraction or an integer, and an exponent E, which is an integer. The number of digits in M determines the precision of (M,E); B and E determine its range. With a word size of n bits, 2^n is the most real numbers that (M,E) can represent. Increasing B increases the range of the representable real numbers but results in a sparser distribution of numbers over that range.

As a small example, suppose that M and E are both 3-bit, sign-magnitude integers and $B = 2$. Then M and E can each assume the values ±0, ±1, ±2, and ±3. All binary words of the form $(M,E) = (x00, xxx)$ represent zero, where x denotes either 0 or 1. The smallest nonzero positive number is (001,111), denoting $1 \times 2^{-3} = 0.125$; (101,111) denotes –0.125. The largest representable positive number is (011,011), which denotes $3 \times 2^3 = 24$, while (111,011) denotes the largest negative number –24. Observe that the left-most bit, which is the sign of the mantissa, is also the sign of the floating-point number. Figure 3.24 illustrates the real numbers representable by this 6-bit, floating-point format. As the figure shows, they are sparsely and nonuniformly distributed over the range ±24.

The floating-point representation of most real numbers is only approximate. For instance, the 6-bit format of Figure 3.24 cannot represent the number 1.25; it is approximated by (011,101), representing 1.5, or by either (001,000) or (001,100), representing 1.0. Moreover, the results of most calculations with floating-point arithmetic only approximate the correct result. For example, in the system of Figure 3.24, the exact result (18) of the addition (011,001) + (011,010), which implements 6 + 12, is not representable. The closest representable number, that is, the best approximation to 18, is (010,011) = 16. Overflow occurs in this small system when a result's magnitude exceeds 24, and underflow occurs when a nonzero result has a magnitude less than 0.125. In practice, floating-point numbers must have long mantissas (at least 20 bits), and the results of floating-point operations must be carefully rounded off to minimize the errors inherent in floating-point representation. It is common practice for floating-point processing circuits to include a few extra mantissa digits termed *guard digits* to reduce approximation errors; the guard digits are removed automatically from the final results.

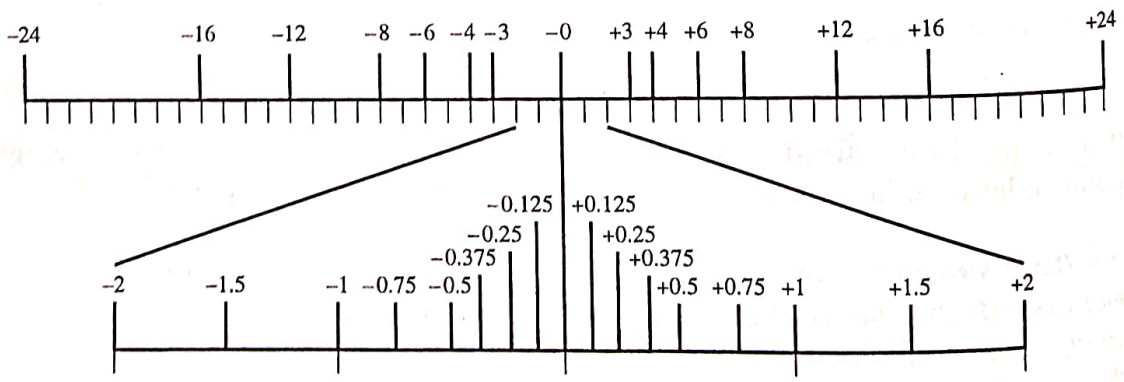


Figure 3.24
The real numbers representable by a hypothetical 6-bit, floating-point format.

Normalization and biasing. Floating-point representation is redundant in the sense that the same number can be represented in more than one way. For example, 1.0×10^{18}, 0.1×10^{19}, 1000000×10^{12}, and 0.000001×10^{24} are possible representations of a quintillion. It is generally desirable to have a unique or *normal* form for each representable number in a floating-point system. Consider the common case where the mantissa is a sign-magnitude fraction and a base of r is used. The mantissa is said to be *normalized* if the digit to the right of the radix point is not zero, that is, no leading zeros appear in the magnitude part of the number. Thus, for example, 0.1×10^{19} is the unique normal form of a quintillion using base 10, a decimal mantissa, and a decimal exponent. A binary fraction in twos-complement code is normalized when the sign bit differs from the bit to its right. This implies that no leading 1s appear in the magnitude part of negative numbers. Normalization restricts the magnitude $|M|$ of a fractional binary mantissa to the range

$$1/2 \leq |M| < 1$$

Normal forms can be defined similarly for other floating-point codes. An unnormalized number is normalized by shifting the mantissa to the right or left and appropriately incrementing or decrementing the exponent to compensate for the mantissa shift.

The representation of zero poses some special problems. The mantissa must, of course, be zero, but the exponent can have any value, since $0 \times B^E = 0$ for all values of E. Often in attempting to compute zero, round-off errors result in a mantissa that is nearly, but not exactly, zero. For the entire floating-point number to be close to zero, its exponent must be a very large negative number $-K$. This requirement suggests that the exponent used for representing zero should be the negative number with the largest magnitude that can be contained in the exponent field of the number format. If k bits are allowed for the exponent including its sign, then 2^k exponent bit patterns are available to represent signed integers, which can range either from -2^{k-1} to $2^{k-1} - 1$ or from $-2^{k-1} + 1$ to 2^{k-1}, so that K is 2^{k-1} or $2^{k-1} - 1$.

A second complication arises from the desirability of representing zero by a sequence of 0-bits only. This convention gives zero the same representation in both fixed- and floating-point formats, which facilitates the implementation of instructions that test for zero. These considerations suggest that floating-point exponents should be encoded in excess-K code similar to the excess-three code of Figure 3.23, where the exponent field E contains an integer that is the desired exponent value plus K. The quantity K is called the *bias,* and an exponent encoded in this way is called a *biased exponent* or *characteristic.* Figure 3.25 shows the possible values of an 8-bit exponent with bias 127 and 128.

Standards. Until the 1980s floating-point number formats varied from one computer family to the next, making it difficult to transport programs between different computers without encountering small but significant differences in such areas as round-off errors. To deal with this problem, the Institute of Electrical and Electronics Engineers (IEEE) sponsored a standard format for 32-bit and larger floating-point numbers, known as the IEEE 754 standard [IEEE 1985], which has been widely adopted by computer manufacturers. Besides specifying the permissible formats for M, E, and B, the IEEE standard prescribes methods for handling round-off errors, overflow, underflow, and other exceptional conditions.

Exponent bit pattern E	Unsigned value	Number represented	
		Bias = 127	Bias = 128
111 . . . 11	255	+128	+127
111 . . . 10	254	+127	+126
. . .		. . .	. . .
100 . . . 01	129	+2	+1
100 . . . 00	128	+1	0
011 . . . 11	127	0	–1
011 . . . 10	126	–1	–2
. . .		. . .	. . .
000 . . . 01	1	–126	–127
000 . . . 00	0	–127	–128

Figure 3.25
Eight-bit biased exponents with bias = 127 (excess-127 code) and bias = 128 (excess-128 code).

EXAMPLE 3.4 THE IEEE 754 FLOATING-POINT NUMBER FORMAT [IEEE 1985; GOLDBERG 1991]. This standard format for 32-bit numbers is illustrated in Figure 3.26. It comprises a 23-bit mantissa field M, an 8-bit exponent field E, and a sign bit S. The base B is two. As in all signed binary number formats, both fixed-point and floating-point, S occupies the left-most bit position. M is a fraction that with S forms a sign-magnitude binary number. For the reasons discussed earlier, floating-point numbers are usually normalized, meaning that the magnitude field should contain no insignificant leading bits. Hence the magnitude part of a normalized sign-magnitude number always has 1 as its most significant digit. There is no need to actually store this leading 1 in floating-point numbers, since it can always be inserted by the arithmetic circuits that process the numbers. Consequently, in the IEEE 754 format the complete mantissa (called the *significand* in the standard) is actually 1.M, where the 1 to the left of the binary point is an implicit or *hidden* leading bit that is not stored with the number. Use of the hidden 1 means that the precision of a normalized number is effectively increased by 1 bit. The exponent representation is the 8-bit excess-127 code of Figure 3.25; hence the actual exponent value is computed as $E - 127$. The base B of the floating-point number is 2, so that a 1-bit left (right) shift of M corresponds to incrementing (decrementing) E by one.

Consequently, a 32-bit floating-point number conforming to the IEEE 754 standard represents the real number N given by the formula

$$N = (-1)^S 2^{E-127}(1.M) \tag{3.16}$$

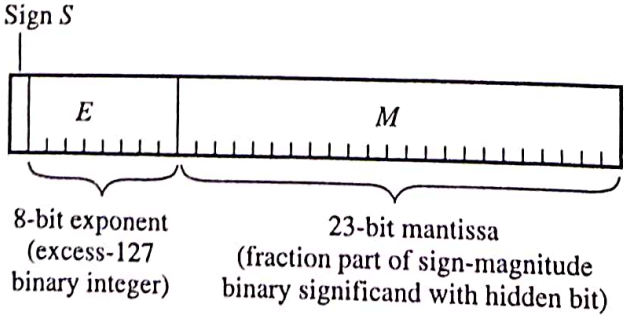


Figure 3.26
IEEE 754 standard 32-bit floating-point number format.

provided $0 < E < 255$. For example, the number $N = -1.5$ is represented by

$$1\ 01111111\ 10000000000000000000000$$

where $S = 1$, $E = 127$, and $M = 0.5$, since from (3.16) we have $N = (-1)^1 2^{127-127}(1.5) = -1.5$. Nonzero floating-point numbers in this format have magnitudes ranging from $2^{-126}(1.0)$ to $2^{+127}(2 - 2^{-23})$, that is, from 1.18×10^{-38} to 3.40×10^{38} approximately. In contrast, 32-bit, fixed-point binary formats for integers can only represent nonzero numbers with magnitudes from 1 to $2^{31} - 1$ (approximately 2.15×10^9). The 64-bit version of the IEEE 754 standard is a straightforward extension of the 32-bit case. It employs an 11-bit exponent E and a 52-bit mantissa M and defines the number

$$N = (-1)^S 2^{E-1023}(1.M) \tag{3.17}$$

where $0 < E < 2047$.

The IEEE floating-point standard addresses a number of subtle problems encountered in floating-point arithmetic. Well-defined formats are specified for the results of overflow, underflow, and other exceptional conditions, which often yield unpredictable and unusable numbers in computers employing other floating-point formats. The IEEE standard's exception formats are intended to set flags in the host processor, which subsequent instructions can use for error control, in many cases with little or no loss of accuracy. If the result of a floating-point operation is not a valid floating-point number, then a special code referred to as *not a number* (NaN) is used. Examples of operations that result in NaNs are dividing zero by zero and taking the square root of a negative number. NaN formats are identified in the standard by $M \neq 0$, and $E = 255$ (32-bit format) or $E = 2047$ (64-bit format).

When overflow occurs, meaning that a number has been produced whose magnitude is too big to represent by the usual format, the result is referred to as *infinity,* or ∞, and is identified by $M = 0$, and $E = 255$ (32-bit format) or $E = 2047$ (64-bit format). The 754 standard stipulates that operations using the floating-point infinities $\pm\infty$ should follow certain properties of infinity in real-number theory, such as $-\infty + N = \infty$ and $-\infty < N < +\infty$ for any finite N. If underflow occurs, implying that a result is nonzero, but too small to represent as a normalized number, it is encoded in a *denormalized*[3] form characterized by $E = 0$ and a significand $0.M$ having a leading 0 instead of the usual leading 1. Denormalization reduces the effect of underflow to a systematic loss of precision equivalent to a small round-off error. Finally, floating-point zero is identified by an all-0 exponent and significand, but the sign S may be 0 or 1. Note that as the tiny denormalized numbers are diminished, they eventually reach zero.

In summary, the number N represented by a 32-bit IEEE-standard, floating-point number has the following set of interpretations.

If $E = 255$ and $M \neq 0$, then $N = \text{NaN}$.
If $E = 255$ and $M = 0$, then $N = (-1)^S \infty$.
If $0 < E < 255$, then $N = (-1)^S 2^{E-127}(1.M)$.
If $E = 0$ and $M \neq 0$, then $N = (-1)^S 2^{E-126}(0.M)$.
If $E = 0$ and $M = 0$, then $N = (-1)^S 0$.

The interpretation of 64-bit and larger floating-point numbers is similar.

[3]The term *unnormalized* applies to numbers with any value of E and a leading 0 instead of a leading 1 associated with their mantissas. Such numbers are encountered only as intermediate results during floating-point computations and are not relevant to the standard.