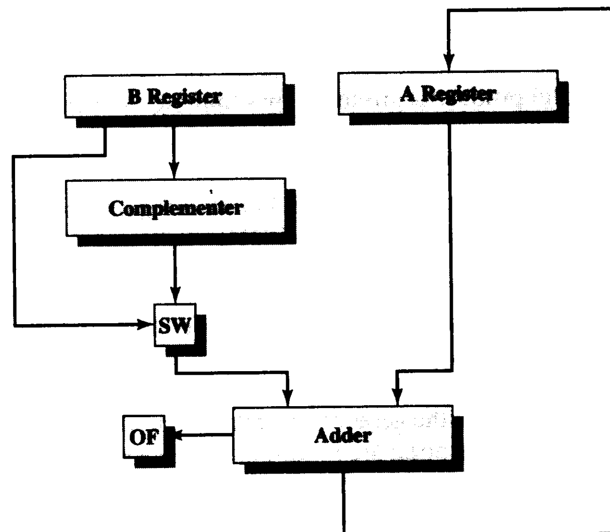
in the upper half of each part of the figure is formed by selecting the appropriate segment of the number line and joining the endpoints. Note that when the numbers are laid out on a circle, the twos complement of any number is horizontally opposite that number (indicated by dashed horizontal lines). Starting at any number on the circle, we can add positive *k* (or subtract negative *k*) to that number by moving *k* positions clockwise, and we can subtract positive *k* (or add negative *k*) from that number by moving *k* positions counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given (overflow).

All of the examples of Figures 9.3 and 9.4 are easily traced in the circle of Figure 9.5.

Figure 9.6 suggests the data paths and hardware elements needed to accomplish addition and subtraction. The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. (A logic implementation of an adder is given in Appendix B.) For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow). For subtraction, the subtrahend (B register) is passed through a twos complementer so that its twos complement is presented to the adder.



OF = Overflow bit
SW = Switch (select addition or subtraction)

Figure 9.6 Block Diagram of Hardware for Addition and Subtraction

1011	**Multiplicand (11)**
×1101	**Multiplier (13)**
1011	
0000	
1011	**Partial products**
1011	
10001111	**Product (143)**

Figure 9.7 Multiplication of Unsigned Binary Integers

Multiplication

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software. A wide variety of algorithms have been used in various computers. The purpose of this subsection is to give the reader some feel for the type of approach typically taken. We begin with the simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation.

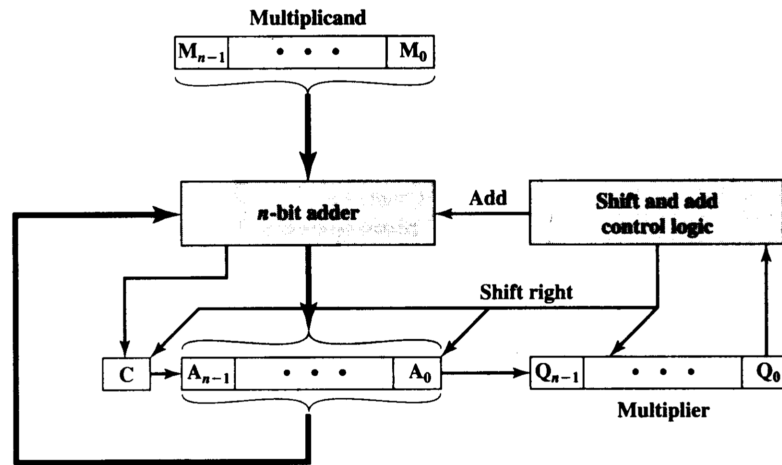
Unsigned Integers Figure 9.7 illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Several important observations can be made:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
4. The multiplication of two n-bit binary integers results in a product of up to $2n$ bits in length (e.g., $11 \times 11 = 1001$).

Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient. First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Figure 9.8a shows a possible implementation employing these measures. The multiplier and multiplicand are loaded into two registers (Q and M). A third register, the A register, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition.

The operation of the multiplier is as follows. Control logic reads the bits of the multiplier one at a time. If Q_0 is 1, then the multiplicand is added to the A register



(a) Block diagram

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

Figure 9.8 Hardware Implementation of Unsigned Binary Multiplication

and the result is stored in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost. If Q_0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier. The resulting $2n$-bit product is contained in the A and Q registers. A flowchart of the operation is shown in Figure 9.9, and an example is given in Figure 9.8b. Note that on the second cycle, when the multiplier bit is 0, there is no add operation.

Twos Complement Multiplication We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers. Consider

$$\begin{array}{r} 1001 \\ +\underline{0011} \\ 1100 \end{array}$$

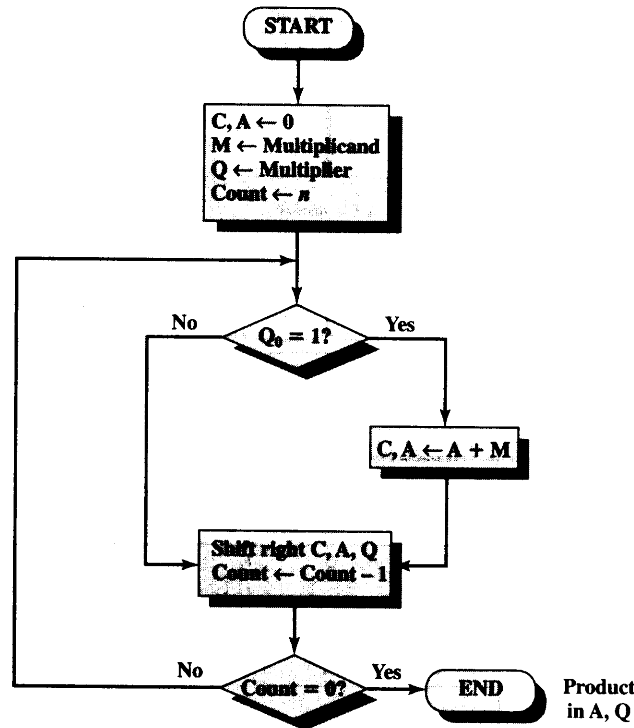


Figure 9.9 Flowchart for Unsigned Binary Multiplication

If these numbers are considered to be unsigned integers, then we are adding 9 (1001) plus 3 (0011) to get 12 (1100). As twos complement integers, we are adding −7 (1001) to 3 (0011) to get −4 (1100).

Unfortunately, this simple scheme will not work for multiplication. To see this, consider again Figure 9.7. We multiplied 11 (1011) by 13 (1101) to get 143 (10001111). If we interpret these as twos complement numbers, we have −5 (1011) times −3 (1101) equals −113 (10001111). This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative. In fact, it will not work if either the multiplicand or the multiplier is negative. To justify this statement, we need to go back to Figure 9.7 and explain what is being done in terms of operations with powers of 2. Recall that any unsigned binary number can be expressed as a sum of powers of 2. Thus,

$$\begin{aligned} 1101 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 2^3 + 2^2 + 2^0 \end{aligned}$$

Further, the multiplication of a binary number by 2^n is accomplished by shifting that number to the left n bits. With this in mind, Figure 9.10 recasts Figure 9.7 to make the generation of partial products by multiplication explicit. The only difference in Figure 9.10 is that it recognizes that the partial products should be viewed as $2n$-bit numbers generated from the n-bit multiplicand.

```
    1011
  × 1101
00001011   1011 × 1 × 2^0
00000000   1011 × 0 × 2^1
00101100   1011 × 1 × 2^2
01011000   1011 × 1 × 2^3
10001111
```

Figure 9.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011. Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).

Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative. The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a $2n$-bit field; the sign bits of the partial products must line up. This is demonstrated in Figure 9.11, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of $9 \times 3 = 27$ proceeds simply. However, if 1001 is interpreted as the twos complement value -7, then each partial product must be a negative twos complement number of $2n$ (8) bits, as shown in Figure 9.11b. Note that this is accomplished by padding out each partial product to the left with binary 1s.

If the multiplier is negative, straightforward multiplication also will not work. The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place. For example, the 4-bit decimal number -3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$$1101 \longleftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing.

There are a number of ways out of this dilemma. One would be to convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two

(a) Unsigned integers		(b) Twos complement integers	
1001	(9)	1001	(−7)
×0011	(3)	×0011	(3)
00001001	1001×2^0	11111001	$(-7) \times 2^0 = (-7)$
00010010	1001×2^1	11110010	$(-7) \times 2^1 = (-14)$
00011011	(27)	11101011	(−21)

Figure 9.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

original numbers differed. Implementers have preferred to use techniques that do not require this final transformation step. One of the most common of these is Booth's algorithm. This algorithm also has the benefit of speeding up the multiplication process, relative to a more straightforward approach.

Booth's algorithm is depicted in Figure 9.12 and can be described as follows. As before, the multiplier and multiplicand are placed in the Q and M registers, respectively. There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}; its use is explained shortly. The results of the multiplication will appear in the A and Q registers. A and Q_{-1} are initialized to 0. As before, control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined. If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0. Following the addition or subtraction, the right shift occurs. In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}. This is required to preserve the sign of the number in A and Q. It is known as an **arithmetic shift**, because it preserves the sign bit.

Figure 9.13 shows the sequence of events in Booth's algorithm for the multiplication of 7 by 3. More compactly, the same operation is depicted in Figure 9.14a.

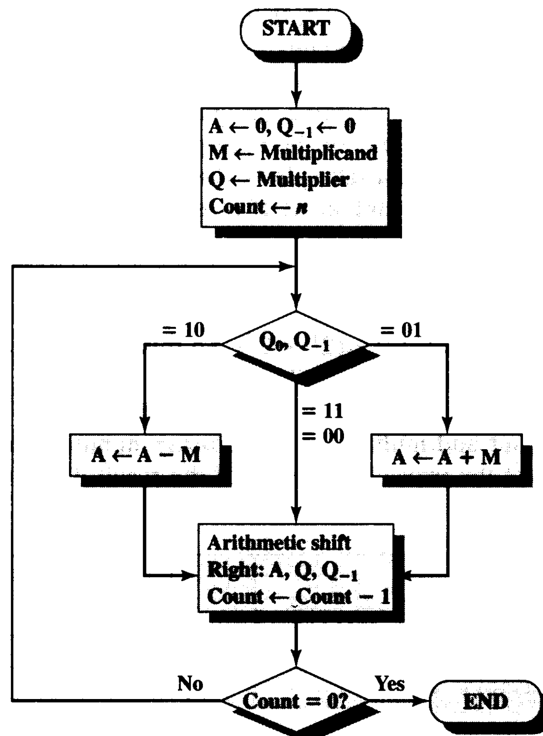


Figure 9.12 Booth's Algorithm for Twos Complement Multiplication

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A - M	First
1100	1001	1	0111	Shift	cycle
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third
0010	1010	0	0111	Shift	cycle
0001	0101	0	0111	Shift	Fourth cycle

Figure 9.13 Example of Booth's Algorithm (7 × 3)

The rest of Figure 9.14 gives other examples of the algorithm. As can be seen, it works with any combination of positive and negative numbers. Note also the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of only one addition or subtraction per block.

Why does Booth's algorithm work? Consider first the case of a positive multiplier. In particular, consider a positive multiplier consisting of one block of 1s surrounded by 0s (for example, 00011110). As we know, multiplication can be achieved by adding appropriately shifted copies of the multiplicand:

$$\begin{aligned} M \times (00011110) &= M \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= M \times (16 + 8 + 4 + 2) \\ &= M \times 30 \end{aligned}$$

The number of such operations can be reduced to two if we observe that

$$2^n + 2^{n-1} + \cdots + 2^{n-K} = 2^{n+1} - 2^{n-K} \tag{9.3}$$

(a)		(b)	
0111		0111	
× 0011	(0)	× 1101	(0)
11111001	1-0	11111001	1-0
0000000	1-1	0000111	0-1
000111	0-1	111001	1-0
00010101	(21)	11101011	(-21)

(a) (7) × (3) = (21) (b) (7) × (−3) = (−21)

(c)		(d)	
1001		1001	
× 0011	(0)	× 1101	(0)
00000111	1-0	00000111	1-0
0000000	1-1	1111001	0-1
111001	0-1	000111	1-0
11101011	(-21)	00010101	(21)

(c) (−7) × (3) = (−21) (d) (−7) × (−3) = (21)

Figure 9.14 Examples Using Booth's Algorithm

$$\begin{aligned} \mathbf{M} \times (00011110) &= \mathbf{M} \times (2^5 - 2^1) \\ &= \mathbf{M} \times (32 - 2) \\ &= \mathbf{M} \times 30 \end{aligned}$$

So the product can be generated by one addition and one subtraction of the multiplicand. This scheme extends to any number of blocks of 1s in a multiplier, including the case in which a single 1 is treated as a block.

$$\begin{aligned} \mathbf{M} \times (01111010) &= \mathbf{M} \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= \mathbf{M} \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1–0) and an addition when the end of the block is encountered (0–1).

To show that the same scheme works for a negative multiplier, we need to observe the following. Let X be a negative number in twos complement notation:

$$\text{Representation of } X = \{1x_{n-2}x_{n-3}\ldots x_1x_0\}$$

Then the value of X can be expressed as follows:

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0) \qquad \textbf{(9.4)}$$

The reader can verify this by applying the algorithm to the numbers in Table 9.2.

The leftmost bit of X is 1, because X is negative. Assume that the leftmost 0 is in the kth position. Thus, X is of the form

$$\text{Representation of } X = \{111\ldots 10x_{k-1}x_{k-2}\ldots x_1x_0\} \qquad \textbf{(9.5)}$$

Then the value of X is

$$X = -2^{n-1} + 2^{n-2} + \cdots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \qquad \textbf{(9.6)}$$

From Equation (9.3), we can say that

$$2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = 2^{n-1} - 2^{k+1}$$

Rearranging,

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = -2^{k+1} \qquad \textbf{(9.7)}$$

Substituting Equation (9.7) into Equation (9.6), we have

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \qquad \textbf{(9.8)}$$

At last we can return to Booth's algorithm. Remembering the representation of X [Equation (9.5)], it is clear that all of the bits from x_0 up to the leftmost 0 are handled properly because they produce all of the terms in Equation (9.8) but (-2^{k+1}) and thus are in the proper form. As the algorithm scans over the leftmost 0 and encounters the next 1 (2^{k+1}) a 1–0 transition occurs and a subtraction takes place (-2^{k+1}). This is the remaining term in Equation (9.8).

As an example, consider the multiplication of some multiplicand by (−6). In twos complement representation, using an 8-bit word, (−6) is represented as 11111010. By Equation (9.4), we know that

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

which the reader can easily verify. Thus,

$$M \times (11111010) = M \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

Using Equation (9.7),

$$M \times (11111010) = M \times (-2^3 + 2^1)$$

which the reader can verify is still M × (−6). Finally, following our earlier line of reasoning,

$$M \times (11111010) = M \times (-2^3 + 2^2 - 2^1)$$

We can see that Booth's algorithm conforms to this scheme. It performs a subtraction when the first 1 is encountered (1–0), an addition when (01) is encountered, and finally another subtraction when the first 1 of the next block of 1s is encountered. Thus, Booth's algorithm performs fewer additions and subtractions than a more straightforward algorithm.

Division

Division is somewhat more complex than multiplication but is based on the same general principles. As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction.

Figure 9.15 shows an example of the long division of unsigned binary integers. It is instructive to describe the process in detail. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder*. From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. As

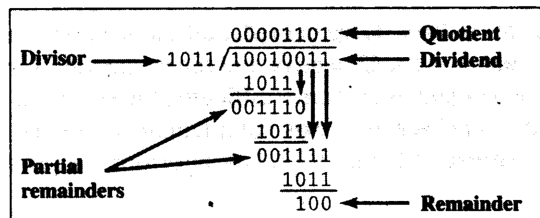


Figure 9.15 Example of Division of Unsigned Binary Integers

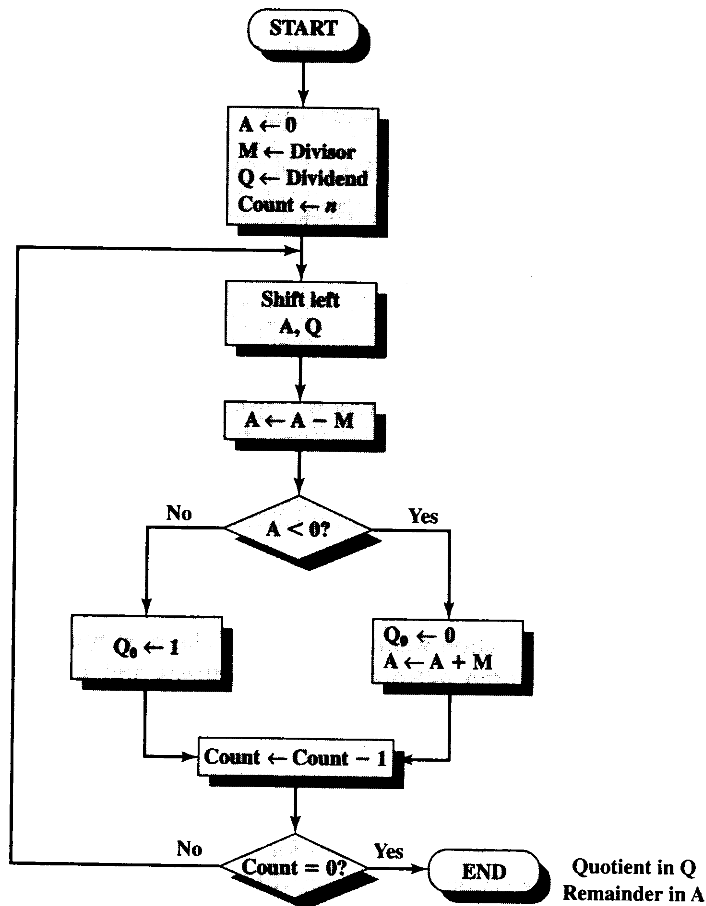


Figure 9.16 Flowchart for Unsigned Binary Division

before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

Figure 9.16 shows a machine algorithm that corresponds to the long division process. The divisor is placed in the M register, the dividend in the Q register. At each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder.[3] If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A to restore the previous value. The count is then decremented, and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.

This process can, with some difficulty, be extended to negative numbers. We give here one approach for twos complement numbers. Several examples of this approach are shown in Figure 9.17. The algorithm can be summarized as follows:

[3]This is subtraction of unsigned integers. A result that requires a borrow out of the most significant bit is a negative result.

A	Q	M = 0011
0000	0111	Initial value
0000	1110	Shift
1101		Subtract
0000	1110	Restore
0001	1100	Shift
1110		Subtract
0001	1100	Restore
0011	1000	Shift
0000		Subtract
0000	1001	Set $Q_0 = 1$
0001	0010	Shift
1110		Subtract
0001	0010	Restore

(a) (7)/(3)

A	Q	M = 1101
0000	0111	Initial value
0000	1110	Shift
1101		Add
0000	1110	Restore
0001	1100	Shift
1110		Add
0001	1100	Restore
0011	1000	Shift
0000		Add
0000	1001	Set $Q_0 = 1$
0001	0010	Shift
1110		Add
0001	0010	Restore

(b) (7)/(−3)

A	Q	M = 0011
1111	1001	Initial value
1111	0010	Shift
0010		Add
1111	0010	Restore
1110	0100	Shift
0001		Add
1110	0100	Restore
1100	1000	Shift
1111		Add
1111	1001	Set $Q_0 = 1$
1111	0010	Shift
0010		Add
1111	0010	Restore

(c) (−7)/(3)

A	Q	M = 1101
1111	1001	Initial value
1111	0010	Shift
0010		Subtract
1111	0010	Restore
1110	0100	Shift
0001		Subtract
1110	0100	Restore
1100	1000	Shift
1111		Subtract
1111	1001	Set $Q_0 = 1$
1111	0010	Shift
0010		Subtract
1111	0010	Restore

(d) (−7)/(−3)

Figure 9.17 Examples of Twos Complement Division

1. Load the divisor into the M register and the dividend into the A, Q registers. The dividend must be expressed as a $2n$-bit twos complement number. Thus, for example, the 4-bit 0111 becomes 00000111, and 1001 becomes 11111001.
2. Shift A, Q left 1 bit position.
3. If M and A have the same signs, perform $A \leftarrow A - M$; otherwise, $A \leftarrow A + M$.
4. The preceding operation is successful if the sign of A is the same before and after the operation.
   a. If the operation is successful or $A = 0$, then set $Q_0 \leftarrow 1$.
   b. If the operation is unsuccessful and $A \neq 0$, then set $Q_0 \leftarrow 0$ and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A. If the signs of the divisor and dividend were the same, then the quotient is in Q; otherwise, the correct quotient is the twos complement of Q.

The reader will note from Figure 9.17 that $(-7)/(3)$ and $(7)/(-3)$ produce different remainders. This is because the remainder is defined by

$$D = Q \times V + R$$

where

D = dividend
Q = quotient
V = divisor
R = remainder

The results of Figure 9.17 are consistent with this formula.

9.4 FLOATING-POINT REPRESENTATION

Principles

With a fixed-point notation (e.g., twos complement) it is possible to represent a range of positive and negative integers centered on 0. By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well.

This approach has limitations. Very large numbers cannot be represented, nor can very small fractions. Furthermore, the fractional part of the quotient in a division of two large numbers could be lost.

For decimal numbers, one gets around this limitation by using scientific notation. Thus, 976,000,000,000,000 can be represented as 9.76×10^{14}, and 0.0000000000000976 can be represented as 9.76×10^{-14}. What we have done, in effect, is dynamically to slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point. This allows a range of very large and very small numbers to be represented with only a few digits.

This same approach can be taken with binary numbers. We can represent a number in the form

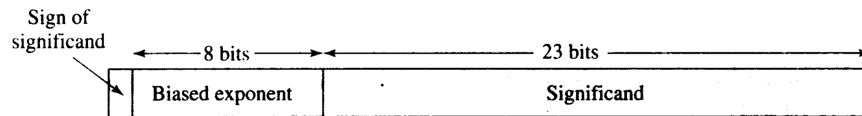
$$\pm S \times B^{\pm E}$$

This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand S
- Exponent E

The **base** B is implicit and need not be stored because it is the same for all numbers. Typically, it is assumed that the radix point is to the right of the leftmost, or most significant, bit of the significand. That is, there is one bit to the left of the radix point.

The principles used in representing binary floating-point numbers are best explained with an example. Figure 9.18a shows a typical 32-bit floating-point format. The leftmost bit stores the **sign** of the number (0 = positive, 1 = negative). The **exponent** value is stored in the next 8 bits. The representation used is known as a **biased representation**. A fixed value, called the bias, is subtracted from the field to



(a) Format

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$
$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$
$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$
$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$

(b) Examples

Figure 9.18 Typical 32-Bit Floating-Point Format

get the true exponent value. Typically, the bias equals $(2^{k-1} - 1)$, where k is the number of bits in the binary exponent. In this case, the 8-bit field yields the numbers 0 through 255. With a bias of 127 $(2^7 - 1)$, the true exponent values are in the range -127 to $+128$. In this example, the base is assumed to be 2.

Table 9.2 shows the biased representation for 4-bit integers. Note that when the bits of a biased representation are treated as unsigned integers, the relative magnitudes of the numbers do not change. For example, in both biased and unsigned representations, the largest number is 1111 and the smallest number is 0000. This is not true of sign-magnitude or twos complement representation. An advantage of biased representation is that nonnegative floating-point numbers can be treated as integers for comparison purposes.

The final portion of the word (23 bits in this case) is the **significand.**[4]

Any floating-point number can be expressed in many ways.

The following are equivalent, where the significand is expressed in binary form:

$$0.110 \times 2^5$$
$$110 \times 2^2$$
$$0.0110 \times 2^6$$

To simplify operations on floating-point numbers, it is typically required that they be normalized. A **normalized number** is one in which the most significant digit of the significand is nonzero. For base 2 representation, a normalized number is therefore one in which the most significant bit of the significand is one. As was mentioned, the typical convention is that there is one bit to the left of the radix point. Thus, a normalized nonzero number is one in the form

$$\pm 1.\mathrm{bbb}\ldots\mathrm{b} \times 2^{\pm E}$$

where b is either binary digit (0 or 1). Because the most significant bit is always one, it is unnecessary to store this bit; rather, it is implicit. Thus, the 23-bit field

[4]The term *mantissa*, sometimes used instead of *significand*, is considered obsolete. Mantissa also means the fractional part of a logarithm, so is best avoided in this context.

is used to store a 24-bit significand with a value in the half open interval [1, 2). Given a number that is not normalized, the number may be normalized by shifting the radix point to the right of the leftmost 1 bit and adjusting the exponent accordingly.

Figure 9.18b gives some examples of numbers stored in this format. For each example, on the left is the binary number; in the center is the corresponding bit pattern; on the right is the decimal value. Note the following features:

- The sign is stored in the first bit of the word.
- The first bit of the true significand is always 1 and need not be stored in the significand field.
- The value 127 is added to the true exponent to be stored in the exponent field.
- The base is 2.

For comparison, Figure 9.19 indicates the range of numbers that can be represented in a 32-bit word. Using twos complement integer representation, all of the integers from -2^{31} to $2^{31} - 1$ can be represented, for a total of 2^{32} different numbers. With the example floating-point format of Figure 9.18, the following ranges of numbers are possible:

- Negative numbers between $-(2 - 2^{-23}) \times 2^{128}$ and -2^{-127}
- Positive numbers between 2^{-127} and $(2 - 2^{-23}) \times 2^{128}$

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(2 - 2^{-23}) \times 2^{128}$, called **negative overflow**
- Negative numbers greater than 2^{-127}, called **negative underflow**
- Zero

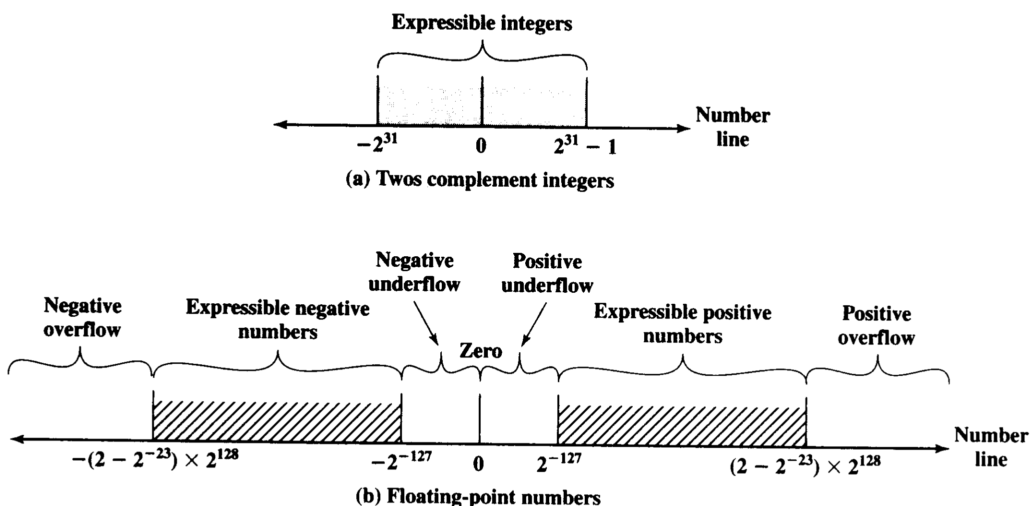


Figure 9.19 Expressible Numbers in Typical 32-Bit Formats

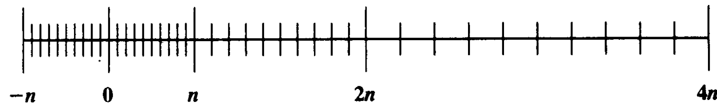


Figure 9.20 Density of Floating-Point Numbers

- Positive numbers less than 2^{-127}, called **positive underflow**
- Positive numbers greater than $(2 - 2^{-23}) \times 2^{128}$, called **positive overflow**

The representation as presented will not accommodate a value of 0. However, as we shall see, actual floating-point representations include a special bit pattern to designate zero. Overflow occurs when an arithmetic operation results in a magnitude greater than can be expressed with an exponent of 128 (e.g., $2^{120} \times 2^{100} = 2^{220}$). Underflow occurs when the fractional magnitude is too small (e.g., $2^{-120} \times 2^{-100} = 2^{-220}$). Underflow is a less serious problem because the result can generally be satisfactorily approximated by 0.

It is important to note that we are not representing more individual values with floating-point notation. The maximum number of different values that can be represented with 32 bits is still 2^{32}. What we have done is to spread those numbers out in two ranges, one positive and one negative.

Also, note that the numbers represented in floating-point notation are not spaced evenly along the number line, as are fixed-point numbers. The possible values get closer together near the origin and farther apart as you move away, as shown in Figure 9.20. This is one of the trade-offs of floating-point math: Many calculations produce results that are not exact and have to be rounded to the nearest value that the notation can represent.

In the type of format depicted in Figure 9.18, there is a trade-off between range and precision. The example shows 8 bits devoted to the exponent and 23 to the significand. If we increase the number of bits in the exponent, we expand the range of expressible numbers. But because only a fixed number of different values can be expressed, we have reduced the density of those numbers and therefore the precision. The only way to increase both range and precision is to use more bits. Thus, most computers offer, at least, single-precision numbers and double-precision numbers. For example, a single-precision format might be 32 bits, and a double-precision format 64 bits.

So there is a trade-off between the number of bits in the exponent and the number of bits in the significand. But it is even more complicated than that. The implied base of the exponent need not be 2. The IBM S/390 architecture, for example, uses a base of 16 [ANDE67b]. The format consists of a 7-bit exponent and a 24-bit significand.

In the IBM base-16 format,

$$0.11010001 \times 2^{10100} = 0.11010001 \times 16^{101}$$

and the exponent is stored to represent 5 rather than 20.

The advantage of using a larger exponent is that a greater range can be achieved for the same number of exponent bits. But remember, we have not increased the number of different values that can be represented. Thus, for a fixed format, a larger exponent base gives a greater range at the expense of less precision.

IEEE Standard for Binary Floating-Point Representation

The most important floating-point representation is defined in IEEE Standard 754, adopted in 1985. This standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs. The standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors.

The IEEE standard defines both a 32-bit single and a 64-bit double format (Figure 9.21), with 8-bit and 11-bit exponents, respectively. The implied base is 2. In addition, the standard defines two extended formats, single and double, whose exact format is implementation dependent. The extended formats include additional bits in the exponent (extended range) and in the significand (extended precision). The extended formats are to be used for intermediate calculations. With their greater precision, the extended formats lessen the chance of a final result that has been contaminated by excessive roundoff error; with their greater range, they also lessen the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format. An additional motivation for the single extended format is that it affords some of the benefits of a double format without incurring the time penalty usually associated with higher precision. Table 9.3 summarizes the characteristics of the four formats.

Not all bit patterns in the IEEE formats are interpreted in the usual way; instead, some bit patterns are used to represent special values. Table 9.4 indicates the values assigned to various bit patterns. The extreme exponent values of all zeros (0) and all ones (255 in single format, 2047 in double format) define special values. The following classes of numbers are represented:

- For exponent values in the range of 1 through 254 for single format and 1 through 2046 for double format, normalized nonzero floating-point numbers are represented. The exponent is biased, so that the range of exponents is -126

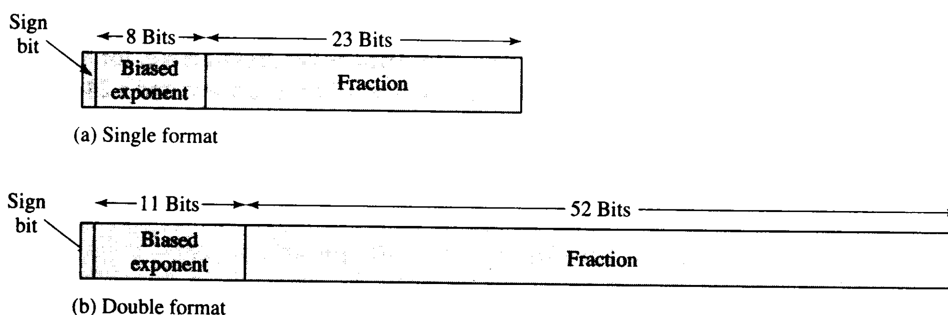


Figure 9.21 IEEE 754 Formats

Table 9.3 IEEE 754 Format Parameters

	Format			
Parameter	**Single**	**Single Extended**	**Double**	**Double Extended**
Word width (bits)	32	≥43	64	≥79
Exponent width (bits)	8	≥11	11	≥15
Exponent bias	127	Unspecified	1023	Unspecified
Maximum exponent	127	≥1023	1023	≥16383
Minimum exponent	−126	≤−1022	−1022	≤−16382
Number range (base 10)	$10^{-38}, 10^{+38}$	Unspecified	$10^{-308}, 10^{+308}$	Unspecified
Significand width (bits)*	23	≥31	52	≥63
Number of exponents	254	Unspecified	2046	Unspecified
Number of fractions	2^{23}	Unspecified	2^{52}	Unspecified
Number of values	1.98×2^{31}	Unspecified	1.99×2^{63}	Unspecified

*Not including implied bit

through +127 for single format and −1022 through +1023. A normalized number requires a 1 bit to the left of the binary point; this bit is implied, giving an effective 24-bit or 53-bit significand (called *fraction* in the standard).

- An exponent of zero together with a fraction of zero represents positive or negative zero, depending on the sign bit. As was mentioned, it is useful to have an exact value of 0 represented.
- An exponent of all ones together with a fraction of zero represents positive or negative infinity, depending on the sign bit. It is also useful to have a representation of infinity. This leaves it up to the user to decide whether to treat overflow as an error condition or to carry the value ∞ and proceed with whatever program is being executed.
- An exponent of zero together with a nonzero fraction represents a denormalized number. In this case, the bit to the left of the binary point is zero and the true exponent is −126 or −1022. The number is positive or negative depending on the sign bit.
- An exponent of all ones together with a nonzero fraction is given the value NaN, which means *Not a Number*, and is used to signal various exception conditions.

The significance of denormalized numbers and NaNs is discussed in Section 9.5.

9.5 FLOATING-POINT ARITHMETIC

Table 9.5 summarizes the basic operations for floating-point arithmetic. For addition and subtraction, it is necessary to ensure that both operands have the same exponent value. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are more straightforward.

Table 9.4 Interpretation of IEEE 754 Floating-Point Numbers

	Single Precision (32 bits)				Double Precision (64 bits)			
	Sign	**Biased exponent**	**Fraction**	**Value**	**Sign**	**Biased exponent**	**Fraction**	**Value**
Positive zero	0	0	0	0	0	0	0	0
Negative zero	1	0	0	−0	1	0	0	−0
Plus infinity	0	255 (all 1s)	0	∞	0	2047 (all 1s)	0	∞
Minus infinity	1	255 (all 1s)	0	$-\infty$	1	2047 (all 1s)	0	$-\infty$
Quiet NaN	0 or 1	255 (all 1s)	$\neq 0$	NaN	0 or 1	2047 (all 1s)	$\neq 0$	NaN
Signaling NaN	0 or 1	255 (all 1s)	$\neq 0$	NaN	0 or 1	2047 (all 1s)	$\neq 0$	NaN
Positive normalized nonzero	0	$0 < e < 255$	f	$2^{e-127}(1.f)$	0	$0 < e < 2047$	f	$2^{e-1023}(1.f)$
Negative normalized nonzero	1	$0 < e < 255$	f	$-2^{e-127}(1.f)$	1	$0 < e < 2047$	f	$-2^{e-1023}(1.f)$
Positive denormalized	0	0	$f \neq 0$	$2^{e-126}(0.f)$	0	0	$f \neq 0$	$2^{e-1022}(0.f)$
Negative denormalized	1	0	$f \neq 0$	$-2^{e-126}(0.f)$	1	0	$f \neq 0$	$-2^{e-1022}(0.f)$

Table 9.5 Floating-Point Numbers and Arithmetic Operations

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$
$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$
$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$
$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$
$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

A floating-point operation may produce one of these conditions:

- **Exponent overflow:** A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as $+\infty$ or $-\infty$.
- **Exponent underflow:** A negative exponent is less than the minimum possible exponent value (e.g., -200 is less than -127). This means that the number is too small to be represented, and it may be reported as 0.
- **Significand underflow:** In the process of aligning significands, digits may flow off the right end of the significand. As we shall discuss, some form of rounding is required.
- **Significand overflow:** The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment, as we shall explain.

Addition and Subtraction

In floating-point arithmetic, addition and subtraction are more complex than multiplication and division. This is because of the need for alignment. There are four basic phases of the algorithm for addition and subtraction:

1. Check for zeros.
2. Align the significands.
3. Add or subtract the significands.
4. Normalize the result.

A typical flowchart is shown in Figure 9.22. A step-by-step narrative highlights the main functions required for floating-point addition and subtraction. We assume a format similar to those of Figure 9.21. For the addition or subtraction operation,

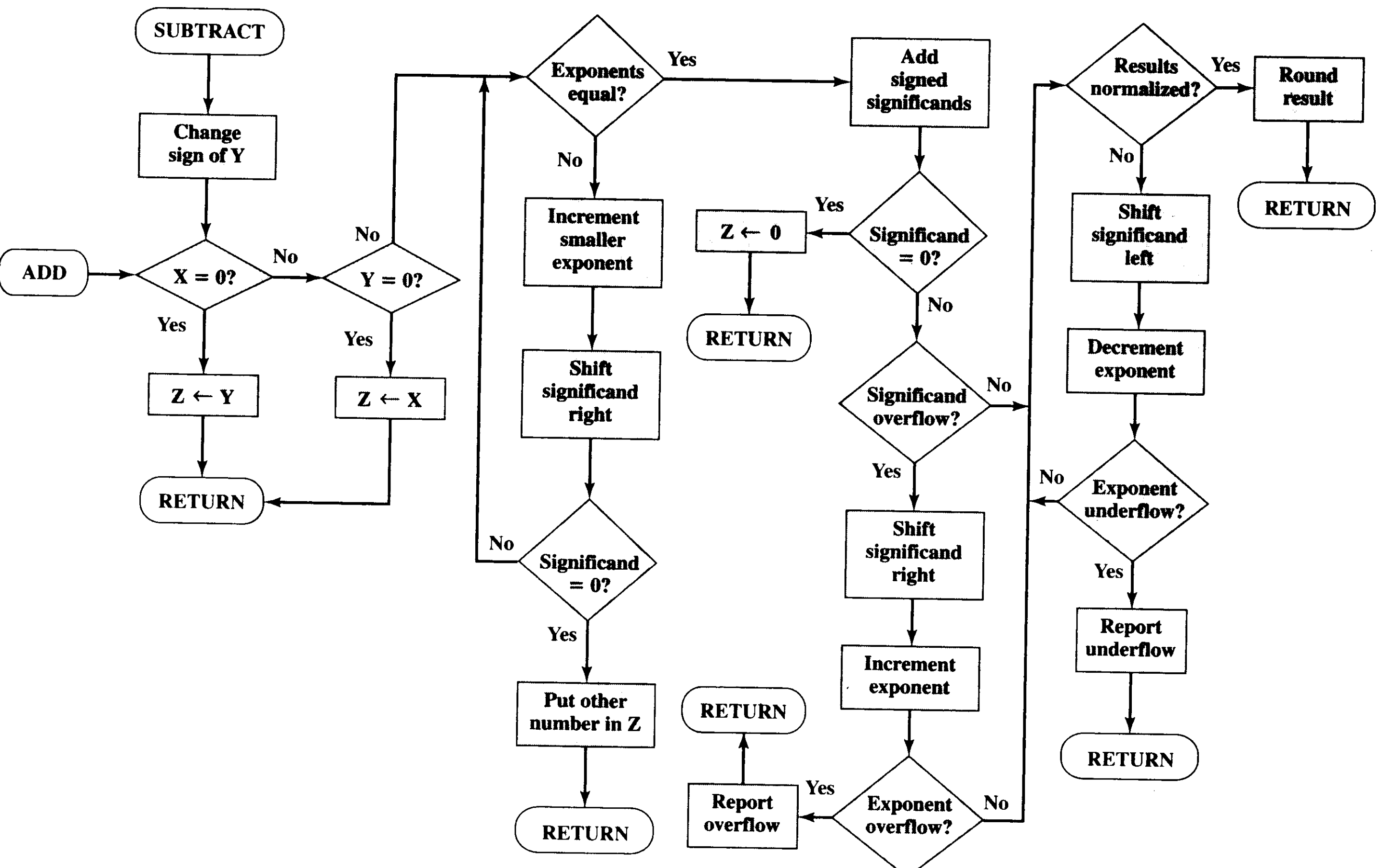


Figure 9.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

the two operands must be transferred to registers that will be used by the ALU. If the floating-point format includes an implicit significand bit, that bit must be made explicit for the operation.

Phase 1: Zero check. Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation. Next, if either operand is 0, the other is reported as the result.

Phase 2: Significand alignment. The next phase is to manipulate the numbers so that the two exponents are equal.

To see the need for aligning exponents, consider the following decimal addition:

$$(123 \times 10^{0}) + (456 \times 10^{-2})$$

Clearly, we cannot just add the significands. The digits must first be set into equivalent positions, that is, the 4 of the second number must be aligned with the 3 of the first. Under these conditions, the two exponents will be equal, which is the mathematical condition under which two numbers in this form can be added. Thus,

$$(123 \times 10^{0}) + (456 \times 10^{-2}) = (123 \times 10^{0}) + (4.56 \times 10^{0}) = 127.56 \times 10^{0}$$

Alignment may be achieved by shifting either the smaller number to the right (increasing its exponent) or shifting the larger number to the left. Because either operation may result in the loss of digits, it is the smaller number that is shifted; any digits that are lost are therefore of relatively small significance. The alignment is achieved by repeatedly shifting the magnitude portion of the significand right 1 digit and incrementing the exponent until the two exponents are equal. (Note that if the implied base is 16, a shift of 1 digit is a shift of 4 bits.) If this process results in a 0 value for the significand, then the other number is reported as the result. Thus, if two numbers have exponents that differ significantly, the lesser number is lost.

Phase 3: Addition. Next, the two significands are added together, taking into account their signs. Because the signs may differ, the result may be 0. There is also the possibility of significand overflow by 1 digit. If so, the significand of the result is shifted right and the exponent is incremented. An exponent overflow could occur as a result; this would be reported and the operation halted.

Phase 4: Normalization. The final phase normalizes the result. Normalization consists of shifting significand digits left until the most significant digit (bit, or 4 bits for base-16 exponent) is nonzero. Each shift causes a decrement of the

exponent and thus could cause an exponent underflow. Finally, the result must be rounded off and then reported. We defer a discussion of rounding until after a discussion of multiplication and division.

Multiplication and Division

Floating-point multiplication and division are much simpler processes than addition and subtraction, as the following discussion indicates.

We first consider multiplication, illustrated in Figure 9.23. First, if either operand is 0, 0 is reported as the result. The next step is to add the exponents. If the exponents are stored in biased form, the exponent sum would have doubled the bias. Thus, the bias value must be subtracted from the sum. The result could be either an exponent overflow or underflow, which would be reported, ending the algorithm.

If the exponent of the product is within the proper range, the next step is to multiply the significands, taking into account their signs. The multiplication is performed

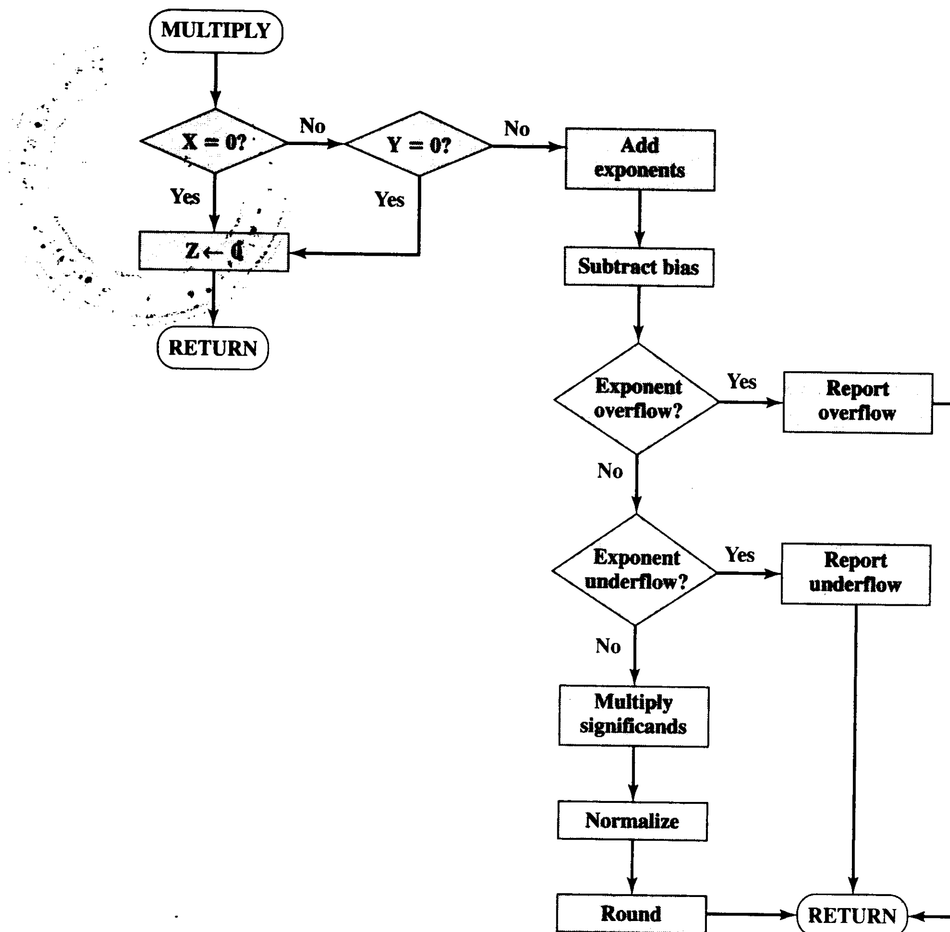


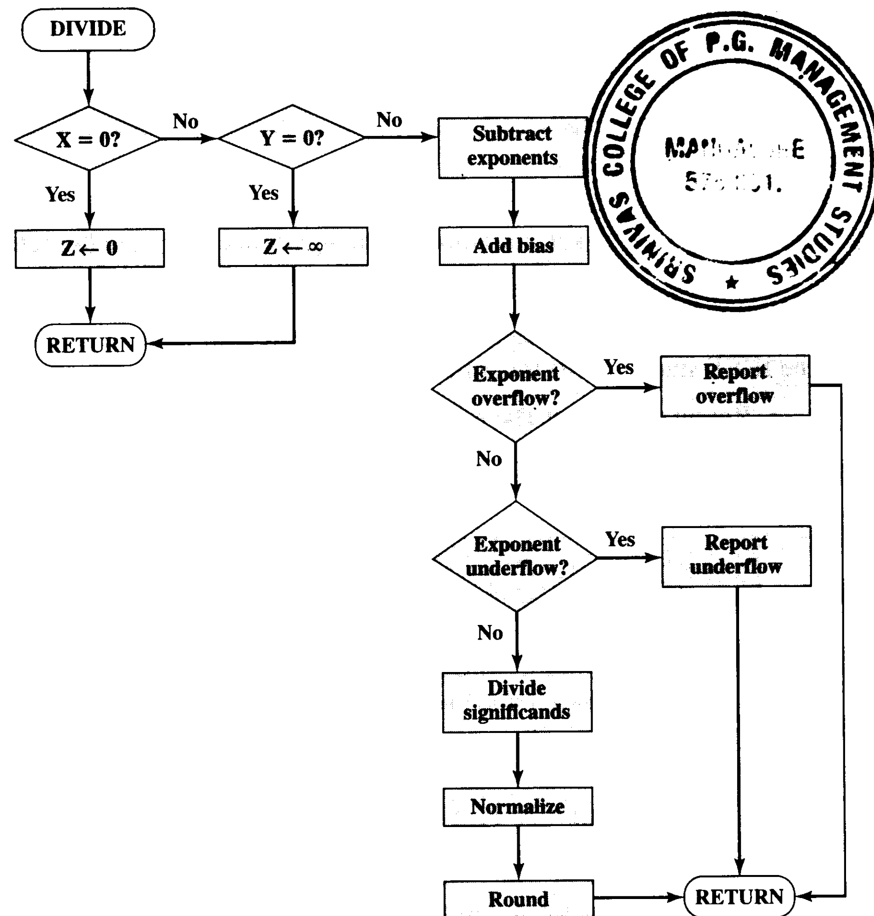
Figure 9.23 Floating-Point Multiplication (Z ← X × Y)

in the same way as for integers. In this case, we are dealing with a sign-magnitude representation, but the details are similar to those for twos complement representation. The product will be double the length of the multiplier and multiplicand. The extra bits will be lost during rounding.

After the product is calculated, the result is then normalized and rounded, as was done for addition and subtraction. Note that normalization could result in exponent underflow.

Finally, let us consider the flowchart for division depicted in Figure 9.24. Again, the first step is testing for 0. If the divisor is 0, an error report is issued, or the result is set to infinity, depending on the implementation. A dividend of 0 results in 0. Next, the divisor exponent is subtracted from the dividend exponent. This removes the bias, which must be added back in. Tests are then made for exponent underflow or overflow.

The next step is to divide the significands. This is followed with the usual normalization and rounding.



SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES

Figure 9.24 Floating-Point Division (Z ← X/Y)

Precision Considerations

Guard Bits We mentioned that, prior to a floating-point operation, the exponent and significand of each operand are loaded into ALU registers. In the case of the significand, the length of the register is almost always greater than the length of the significand plus an implied bit. The register contains additional bits, called guard bits, which are used to pad out the right end of the significand with 0s.

The reason for the use of guard bits is illustrated in Figure 9.25. Consider numbers in the IEEE format, which has a 24-bit significand, including an implied 1 bit to the left of the binary point. Two numbers that are very close in value are $X = 1.00 \ldots 00 \times 2^1$ and $Y = 1.11 \ldots 11 \times 2^0$. If the smaller number is to be subtracted from the larger, it must be shifted right 1 bit to align the exponents. This is shown in Figure 9.25a. In the process, Y loses 1 bit of significance; the result is 2^{-22}. The same operation is repeated in part (b) with the addition of guard bits. Now the least significant bit is not lost due to alignment, and the result is 2^{-23}, a difference of a factor of 2 from the previous answer. When the radix is 16, the loss of precision can be greater. As Figures 9.25c and d show, the difference can be a factor of 16.

Rounding Another detail that affects the precision of the result is the rounding policy. The result of any operation on the significands is generally stored in a longer register. When the result is put back into the floating-point format, the extra bits must be disposed of.

A number of techniques have been explored for performing rounding. In fact, the IEEE standard lists four alternative approaches:

- **Round to nearest:** The result is rounded to the nearest representable number.
- **Round toward $+\infty$:** The result is rounded up toward plus infinity.

(a) Binary example, without guard bits	(c) Hexadecimal example, without guard bits
x = 1.000.....00 × 2^1	x = .100000 × 16^1
$-y$ = 0.111.....11 × 2^1	$-y$ = .0FFFFF × 16^1
z = 0.000.....01 × 2^1	z = .000001 × 16^1
= 1.000.....00 × 2^{-22}	= .100000 × 16^{-4}

(b) Binary example, with guard bits	(d) Hexadecimal example, with guard bits
x = 1.000.....00 0000 × 2^1	x = .100000 00 × 16^1
$-y$ = 0.111.....11 1000 × 2^1	$-y$ = .0FFFFF F0 × 16^1
z = 0.000.....00 1000 × 2^1	z = .000000 10 × 16^1
= 1.000.....00 0000 × 2^{-23}	= .100000 00 × 16^{-5}

Figure 9.25 The Use of Guard Bits

- **Round toward $-\infty$:** The result is rounded down toward negative infinity.
- **Round toward 0:** The result is rounded toward zero.

Let us consider each of these policies in turn. **Round to nearest** is the default rounding mode listed in the standard and is defined as follows: The representable value nearest to the infinitely precise result shall be delivered.

If the extra bits, beyond the 23 bits that can be stored, are 10010, then the extra bits amount to more than one-half of the last representable bit position. In this case, the correct answer is to add binary 1 to the last representable bit, rounding up to the next representable number. Now consider that the extra bits are 01111. In this case, the extra bits amount to less than one-half of the last representable bit position. The correct answer is simply to drop the extra bits (truncate), which has the effect of rounding down to the next representable number.

The standard also addresses the special case of extra bits of the form 10000.... Here the result is exactly halfway between the two possible representable values. One possible technique here would be to always truncate, as this would be the simplest operation. However, the difficulty with this simple approach is that it introduces a small but cumulative bias into a sequence of computations. What is required is an unbiased method of rounding. One possible approach would be to round up or down on the basis of a random number so that, on average, the result would be unbiased. The argument against this approach is that it does not produce predictable, deterministic results. The approach taken by the IEEE standard is to force the result to be even: If the result of a computation is exactly midway between two representable numbers, the value is rounded up if the last representable bit is currently 1 and not rounded up if it is currently 0.

The next two options, **rounding to plus and minus infinity**, are useful in implementing a technique known as interval arithmetic. Interval arithmetic provides an efficient method for monitoring and controlling errors in floating-point computations by producing two values for each result. The two values correspond to the lower and upper endpoints of an interval that contains the true result. The width of the interval, which is the difference between the upper and lower endpoints, indicates the accuracy of the result. If the endpoints of an interval are not representable, then the interval endpoints are rounded down and up, respectively. Although the width of the interval may vary according to implementation, many algorithms have been designed to produce narrow intervals. If the range between the upper and lower bounds is sufficiently narrow, then a sufficiently accurate result has been obtained. If not, at least we know this and can perform additional analysis.

The final technique specified in the standard is **round toward zero**. This is, in fact, simple truncation: The extra bits are ignored. This is certainly the simplest technique. However, the result is that the magnitude of the truncated value is always less

than or equal to the more precise original value, introducing a consistent bias toward zero in the operation. This is a serious bias because it affects every operation for which there are nonzero extra bits.

IEEE Standard for Binary Floating-Point Arithmetic

IEEE 754 goes beyond the simple definition of a format to lay down specific practices and procedures so that floating-point arithmetic produces uniform, predictable results independent of the hardware platform. One aspect of this has already been discussed, namely rounding. This subsection looks at three other topics: infinity, NaNs, and denormalized numbers.

Infinity Infinity arithmetic is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation:

$$-\infty < (\text{every finite number}) < +\infty$$

With the exception of the special cases discussed subsequently, any arithmetic operation involving infinity yields the obvious result.

For example,

$$\begin{array}{ll} 5 + (+\infty) = +\infty & 5 \div (+\infty) = +0 \\ 5 - (+\infty) = -\infty & (+\infty) + (+\infty) = +\infty \\ 5 + (-\infty) = -\infty & (-\infty) + (-\infty) = -\infty \\ 5 - (-\infty) = +\infty & (-\infty) - (+\infty) = -\infty \\ 5 \times (+\infty) = +\infty & (+\infty) - (-\infty) = +\infty \end{array}$$

Quiet and Signaling NaNs A NaN is a symbolic entity encoded in floating-point format, of which there are two types: signaling and quiet. A signaling NaN signals an invalid operation exception whenever it appears as an operand. Signaling NaNs afford values for uninitialized variables and arithmetic-like enhancements that are not the subject of the standard. A quiet NaN propagates through almost every arithmetic operation without signaling an exception. Table 9.6 indicates operations that will produce a quiet NaN.

Note that both types of NaNs have the same general format (Table 9.4): an exponent of all ones and a nonzero fraction. The actual bit pattern of the nonzero fraction is implementation dependent; the fraction values can be used to distinguish quiet NaNs from signaling NaNs and to specify particular exception conditions.

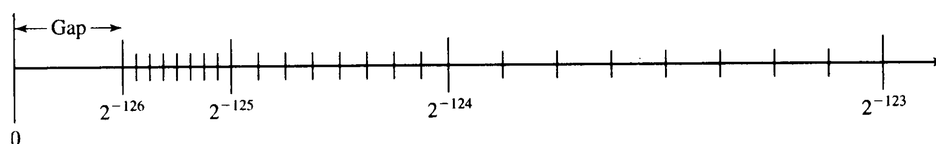
Denormalized Numbers Denormalized numbers are included in IEEE 754 to handle cases of exponent underflow. When the exponent of the result becomes too small (a negative exponent with too large a magnitude), the result is denormalized by right shifting the fraction and incrementing the exponent for each shift until the exponent is within a representable range.

Table 9.6 Operations that Produce a Quiet NaN

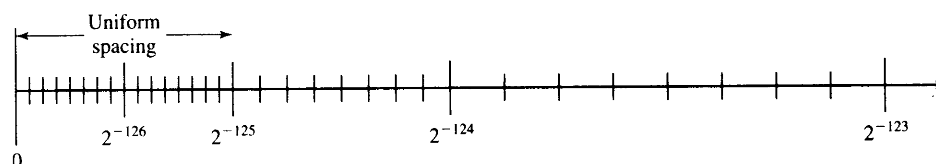
Operation	Quiet NaN Produced by
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$ where $x < 0$

Figure 9.26 illustrates the effect of including denormalized numbers. The representable numbers can be grouped into intervals of the form $[2^n, 2^{n+1}]$. Within each such interval, the exponent portion of the number remains constant while the fraction varies, producing a uniform spacing of representable numbers within the interval. As we get closer to zero, each successive interval is half the width of the preceding interval but contains the same number of representable numbers. Hence the density of representable numbers increases as we approach zero. However, if only normalized numbers are used, there is a gap between the smallest normalized number and 0. In the case of the 32-bit IEEE 754 format, there are 2^{23} representable numbers in each interval, and the smallest representable positive number is 2^{-126}. With the addition of denormalized numbers, an additional 2^{23} numbers are uniformly added between 0 and 2^{-126}.

The use of denormalized numbers is referred to as *gradual underflow* [COON81]. Without denormalized numbers, the gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest



(a) 32-Bit format without denormalized numbers



(b) 32-Bit format with denormalized numbers

Figure 9.26 The Effect of IEEE 754 Denormalized Numbers

representable nonzero number and the next larger number. Gradual underflow fills in that gap and reduces the impact of exponent underflow to a level comparable with roundoff among the normalized numbers.

9.6 RECOMMENDED READING AND WEB SITE

[ERCE04] and [PARH00] are excellent treatments of computer arithmetic, covering all of the topics in this chapter in detail. [FLYN01] is a useful discussion that focuses on practical design and implementation issues. For the serious student of computer arithmetic, a very useful reference is the two-volume [SWAR90]. Volume I was originally published in 1980 and provides key papers (some very difficult to obtain otherwise) on computer arithmetic fundamentals. Volume II contains more recent papers, covering theoretical, design, and implementation aspects of computer arithmetic.

For floating-point arithmetic, [GOLD91] is well named: "What Every Computer Scientist Should Know About Floating-Point Arithmetic." Another excellent treatment of the topic is contained in [KNUT98], which also covers integer computer arithmetic. The following more in-depth treatments are also worthwhile: [OVER01, EVEN00a, OBER97a, OBER97b, SODE96]. [KUCK77] is a good discussion of rounding methods in floating-point arithmetic. [EVEN00b] examines rounding with respect to IEEE 754.

[SCHW99] describes the first IBM S/390 processor to integrate radix-16 and IEEE 754 floating-point arithmetic in the same floating-point unit.

ERCE04 Ercegovac, M., and Lang, T. *Digital Arithmetic.* San Francisco: Morgan Kaufmann, 2004.

EVEN00a Even, G., and Paul, W. "On the Design of IEEE Compliant Floating-Point Units." *IEEE Transactions on Computers*, May 2000.

EVEN00b Even, G., and Seidel, P. "A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication." *IEEE Transactions on Computers*, July 2000.

FLYN01 Flynn, M., and Oberman, S. *Advanced Computer Arithmetic Design.* New York: Wiley, 2001.

GOLD91 Goldberg, D. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys*, March 1991.

KNUT98 Knuth, D. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Reading, MA: Addison-Wesley, 1998.

KUCK77 Kuck, D.; Parker, D.; and Sameh, A. "An Analysis of Rounding Methods in Floating-Point Arithmetic." *IEEE Transactions on Computers.* July 1977.

OBER97a Oberman, S., and Flynn, M. "Design Issues in Division and Other Floating-Point Operations." *IEEE Transactions on Computers*, February 1997.

OBER97b Oberman, S., and Flynn, M. "Division Algorithms and Implementations." *IEEE Transactions on Computers*, August 1997.

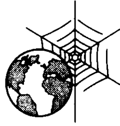
OVER01 Overton, M. *Numerical Computing with IEEE Floating Point Arithmetic.* Philadelphia, PA: Society for Industrial and Applied Mathematics, 2001.

PARH00 Parhami, B. *Computer Arithmetic: Algorithms and Hardware Design.* Oxford: Oxford University Press, 2000.

SCHW99 Schwarz, E., and Krygowski, C. "The S/390 G5 Floating-Point Unit." *IBM Journal of Research and Development*, September/November 1999.

SODE96 Soderquist, P., and Leeser, M. "Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations." *ACM Computing Surveys*, September 1996.

SWAR90 Swartzlander, E., editor. *Computer Arithmetic, Volumes I and II.* Los Alamitos, CA: IEEE Computer Society Press, 1990.



Recommended Web Site:

- **IEEE 754:** The IEEE 754 documents, related publications and papers, and a useful set of links related to computer arithmetic

9.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

arithmetic and logic unit (ALU)	mantissa	quotient
arithmetic shift	minuend	radix point
base	multiplicand	remainder
biased representation	multiplier	rounding
denormalized number	negative overflow	sign bit
dividend	negative underflow	significand
divisor	normalized number	significand overflow
exponent	ones complement representation	significand underflow
exponent overflow	overflow	sign-magnitude representation
exponent underflow	partial product	subtrahend
fixed-point representation	positive overflow	twos complement representation
floating-point representation	positive underflow	
guard bits	product	

Review Questions

9.1 Briefly explain the following representations: sign-magnitude, twos complement, biased.

9.2 Explain how to determine if a number is negative in the following representations: sign-magnitude, twos complement, biased

9.3 What is the sign-extension rule for twos complement numbers?

9.4 How can you form the negation of an integer in twos complement representation?

9.5 In general terms, when does the twos complement operation on an n-bit integer produce the same integer?

9.6 What is the difference between the twos complement representation of a number and the twos complement of a number?

9.7 If we treat 2 twos complement numbers as unsigned integers for purposes of addition, the result is correct if interpreted as a twos complement number. This is not true for multiplication. Why?

9.8 What are the four essential elements of a number in floating-point notation?

9.9 What is the benefit of using biased representation for the exponent portion of a floating-point number?

9.10 What are the differences among positive overflow, exponent overflow, and significand overflow?

9.11 What are the basic elements of floating-point addition and subtraction?

9.12 Give a reason for the use of guard bits.

9.13 List four alternative methods of rounding the result of a floating-point operation.

Problems

9.1 Represent the following decimal numbers in both binary sign/magnitude and twos complement using 16 bits: +512; −29.

9.2 Represent the following twos complement values in decimal: 1101011; 0101101.

9.3 Another representation of binary integers that is sometimes encountered is **ones complement**. Positive integers are represented in the same way as sign magnitude. A negative integer is represented by taking the Boolean complement of each bit of the corresponding positive number.

a. Provide a definition of ones complement numbers using a weighted sum of bits, similar to Equations (9.1) and (9.2).

b. What is the range of numbers that can be represented in ones complement?

c. Define an algorithm for performing addition in ones complement arithmetic.

9.4 Add columns to Table 9.1 for sign magnitude and ones complement.

9.5 Consider the following operation on a binary word. Start with the least significant bit. Copy all bits that are 0 until the first bit is reached and copy that bit, too. Then take the complement of each bit thereafter. What is the result?

9.6 In Section 9.3, the twos complement operation is defined as follows. To find the twos complement of X, take the Boolean complement of each bit of X, and then add 1.

a. Show that the following is an equivalent definition. For an n-bit integer X, the twos complement of X is formed by treating X as an unsigned integer and calculating $(2^n - X)$.

b. Demonstrate that Figure 9.5 can be used to support graphically the claim in part (a), by showing how a clockwise movement is used to achieve subtraction.

9.7 The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$. Find the tens complement of the decimal number13250.

9.8 Calculate (72530 − 13250) using tens complement arithmetic. Assume rules similar to those for twos complement arithmetic.

9.9 Consider the twos-complement addition of two n-bit numbers:

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

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}. Let v be a binary variable indicating overflow when $v = 1$. Fill in the values in the table.

Input	x_{n-1}	0	0	0	0	1	1	1	1
	y_{n-1}	0	0	1	1	0	0	1	1
	c_{n-2}	0	1	0	1	0	1	0	1
Output	z_{n-1}								
	v								

9.10 Assume numbers are represented in 8-bit twos complement representation. Show the calculation of the following:

a. 6 + 13
b. −6 + 13
c. 6 − 13
d. −6 − 13

9.11 Find the following differences using twos complement arithmetic:

a.	**b.**	**c.**	**d.**
111000	11001100	111100001111	11000011
−110011	− 101110	−110011110011	−11101000

9.12 Is the following a valid alternative definition of overflow in twos complement arithmetic?
If the exclusive-OR of the carry bits into and out of the leftmost column is 1, then there is an overflow condition. Otherwise, there is not.

9.13 Compare Figures 9.9 and 9.12. Why is the C bit not used in the latter?

9.14 Given $x = 0101$ and $y = 1010$ in twos complement notation (i.e., $x = 5$, $y = -6$), compute the product $p = x \times y$ with Booth's algorithm.

9.15 Use the Booth algorithm to multiply 23 (multiplicand) by 29 (multiplier), where each number is represented using 6 bits.

9.16 Prove that the multiplication of two n-digit numbers in base B gives a product of no more than $2n$ digits.

9.17 Verify the validity of the unsigned binary division algorithm of Figure 9.16 by showing the steps involved in calculating the division depicted in Figure 9.15. Use a presentation similar to that of Figure 9.17.

9.18 The twos complement integer division algorithm described in Section 9.3 is known as the restoring method because the value in the A register must be restored following unsuccessful subtraction. A slightly more complex approach, known as nonrestoring, avoids the unnecessary subtraction and addition. Propose an algorithm for this latter approach.

9.19 Under computer integer arithmetic, the quotient J/K of two integers J and K is less than or equal to the usual quotient. True or false?

9.20 Divide −145 by 13 in binary twos complement notation, using 12-bit words. Use the algorithm described in Section 9.3.

9.21 **a.** Consider a fixed-point representation using decimal digits, in which the implied radix point can be in any position (e.g., to the right of the least significant digit, to the right of the most significant digit, and so on). How many decimal digits are needed to represent the approximations of both Planck's constant (6.63×10^{-27}) and Avogadro's number (6.02×10^{23})? The implied radix point must be in the same position for both numbers.

b. Now consider a decimal floating-point format with the exponent stored in a biased representation with a bias of 50. A normalized representation is assumed. How many decimal digits are needed to represent these constants in this floating-point format?

9.22 Assume that the exponent e is constrained to lie in the range $0 \leq e \leq X$, with a bias of q, that the base is b, and that the significand is p digits in length.

a. What are the largest and smallest positive values that can be written?
b. What are the largest and smallest positive values that can be written as normalized floating-point numbers?

9.23 Express the following numbers in IEEE 32-bit floating-point format:

a. −1.5	**c.** −1.5	**e.** 1/16
b. −6	**d.** 384	**f.** −1/32

9.24 The following numbers use the IEEE 32-bit floating-point format. What is the equivalent decimal value?

a. 1 10000011 11000000000000000000000
b. 0 01111110 10100000000000000000000
c. 0 10000000 00000000000000000000000

9.25 Consider a reduced 7-bit IEEE floating-point format, with 3 bits for the exponent and 3 bits for the significand. List all 127 values.

9.26 Express the following numbers in IBM's 32-bit floating-point format, which uses a 7-bit exponent with an implied base of 16 and an exponent bias of 64(40 hexadecimal). A normalized floating-point number requires that the leftmost hexadecimal digit be nonzero; the implied radix point is to the left of that digit.

a. 1.0	**c.** 1/64	**e.** −15.0	**g.** 7.2×10^{75}
b. 0.5	**d.** 0.0	**f.** 5.4×10^{-79}	**h.** 65535

9.27 Let 5BCA000 be a floating-point number in IBM format, expressed in hexadecimal. What is the decimal value of the number?

9.28 What would be the bias value for
a. A base-2 exponent (B = 2) in a 6-bit field?
b. A base-8 exponent (B = 8) in a 7-bit field?

9.29 Draw a number line similar to that in Figure 9.19b for the floating-point format of Figure 9.21b.

9.30 Consider a floating-point format with 8 bits for the biased exponent and 23 bits for the significand. Show the bit pattern for the following numbers in this format:
a. −720
b. 0.645

9.31 The text mentions that a 32-bit format can represent a maximum of 2^{32} different numbers. How many different numbers can be represented in the IEEE 32-bit format? Explain.

9.32 Any floating-point representation used in a computer can represent only certain real numbers exactly; all others must be approximated. If A' is the stored value approximating the real value A, then the relative error, r, is expressed as

$$r = \frac{A - A'}{A}$$

Represent the decimal quantity +0.4 in the following floating-point format: base = 2; exponent: biased, 4 bits; significand, 7 bits. What is the relative error?

9.33 If $A = 1.427$, find the relative error if A is truncated to 1.42 and if it is rounded to 1.43.

9.34 When people speak about inaccuracy in floating-point arithmetic, they often ascribe errors to cancellation that occurs during the subtraction of nearly equal quantities. But when X and Y are approximately equal, the difference $X - Y$ is obtained exactly, with no error. What do these people really mean?

9.35 Numerical values A and B are stored in the computer as approximations A' and B'. Neglecting any further truncation or roundoff errors, show that the relative error of the product is approximately the sum of the relative errors in the factors.

9.36 One of the most serious errors in computer calculations occurs when two nearly equal numbers are subtracted. Consider $A = 0.22288$ and $B = 0.22211$. The computer truncates all values to four decimal digits. Thus $A' = 0.2228$ and $B' = 0.2221$.
a. What are the relative errors for A' and B'?
b. What is the relative error for $C' = A' - B'$?

9.37 To get some feel for the effects of denormalization and gradual underflow, consider a decimal system that provides 6 decimal digits for the significand and the smallest normalized number is 10^{-99}. A normalized number has one nonzero decimal digit to the left of the decimal point. Perform the following calculations and denormalize the results. Comment on the results.
a. $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-43})$
b. $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-60})$
c. $(5.67834 \times 10^{-97}) - (5.67812 \times 10^{-97})$

9.38 Show how the following floating-point additions are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $5.566 \times 10^2 + 7.777 \times 10^2$ **b.** $3.344 \times 10^1 + 8.877 \times 10^{-2}$

9.39 Show how the following floating-point subtractions are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $7.744 \times 10^{-3} - 6.666 \times 10^{-3}$ **b.** $8.844 \times 10^{-3} - 2.233 \times 10^{-1}$

9.40 Show how the following floating-point calculations are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $(2.255 \times 10^1) \times (1.234 \times 10^0)$ **b.** $(8.833 \times 10^2) \div (5.555 \times 10^4)$

10.1 MACHINE INSTRUCTION CHARACTERISTICS

The operation of the processor is determined by the instructions it executes, referred to as *machine instructions* or *computer instructions*. The collection of different instructions that the processor can execute is referred to as the processor's *instruction set*.

Elements of a Machine Instruction

Each instruction must contain the information required by the processor for execution. Figure 10.1, which repeats Figure 3.6, shows the steps involved in instruction execution and, by implication, defines the elements of a machine instruction. These elements are as follows:

- **Operation code:** Specifies the operation to be performed (e.g., ADD, I/O). The operation is specified by a binary code, known as the operation code, or **opcode**.
- **Source operand reference:** The operation may involve one or more source operands, that is, operands that are inputs for the operation.
- **Result operand reference:** The operation may produce a result.
- **Next instruction reference:** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

The next instruction to be fetched is located in main memory or, in the case of a virtual memory system, in either main memory or secondary memory (disk). In most cases, the next instruction to be fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction. When an explicit reference is needed, then the main memory or virtual memory address must be supplied. The form in which that address is supplied is discussed in Chapter 11.

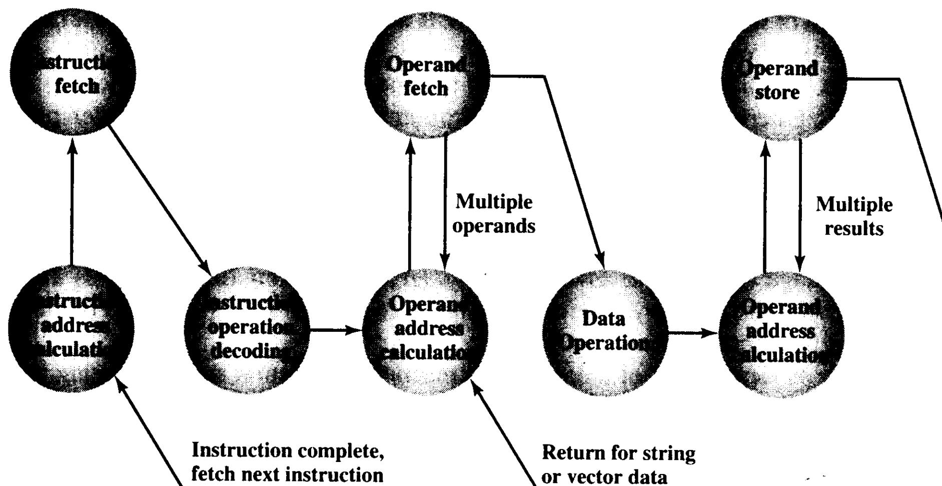


Figure 10.1 Instruction Cycle State Diagram

Source and result operands can be in one of three areas:

- **Main or virtual memory:** As with next instruction references, the main or virtual memory address must be supplied.
- **Processor register:** With rare exceptions, a processor contains one or more registers that may be referenced by machine instructions. If only one register exists, reference to it may be implicit. If more than one register exists, then each register is assigned a unique number, and the instruction must contain the number of the desired register.
- **I/O device:** The instruction must specify the I/O module and device for the operation. If memory-mapped I/O is used, this is just another main or virtual memory address.

Instruction Representation

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. A simple example of an instruction format is shown in Figure 10.2. As another example, the IAS instruction format is shown in Figure 2.2. With most instruction sets, more than one format is used. During instruction execution, an instruction is read into an instruction register (IR) in the processor. The processor must be able to extract the data from the various instruction fields to perform the required operation.

It is difficult for both the programmer and the reader of textbooks to deal with binary representations of machine instructions. Thus, it has become common practice to use a *symbolic representation* of machine instructions. An example of this was used for the IAS instruction set, in Table 2.1.

Opcodes are represented by abbreviations, called *mnemonics,* that indicate the operation. Common examples include

ADD	Add
SUB	Subtract
MPY	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

Operands are also represented symbolically. For example, the instruction

ADD R, Y

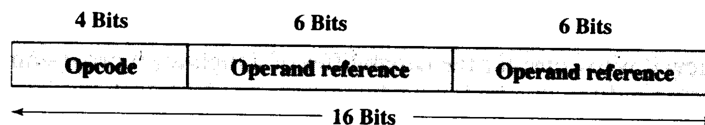


Figure 10.2 A Simple Instruction Format

may mean add the value contained in data location Y to the contents of register R. In this example, Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location, not on its address.

Thus, it is possible to write a machine-language program in symbolic form. Each symbolic opcode has a fixed binary representation, and the programmer specifies the location of each symbolic operand. For example, the programmer might begin with a list of definitions:

$$X = 513$$
$$Y = 514$$

and so on. A simple program would accept this symbolic input, convert opcodes and operand references to binary form, and construct binary machine instructions.

Machine-language programmers are rare to the point of nonexistence. Most programs today are written in a high-level language or, failing that, assembly language, which is discussed at the end of this chapter. However, symbolic machine language remains a useful tool for describing machine instructions, and we will use it for that purpose.

Instruction Types

Consider a high-level language instruction that could be expressed in a language such as BASIC or FORTRAN. For example,

$$X = X + Y$$

This statement instructs the computer to add the value stored in Y to the value stored in X and put the result in X. How might this be accomplished with machine instructions? Let us assume that the variables X and Y correspond to locations 513 and 514. If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:

1. Load a register with the contents of memory location 513.
2. Add the contents of memory location 514 to the register.
3. Store the contents of the register in memory location 513.

As can be seen, the single BASIC instruction may require three machine instructions. This is typical of the relationship between a high-level language and a machine language. A high-level language expresses operations in a concise algebraic form, using variables. A machine language expresses operations in a basic form involving the movement of data to or from registers.

With this simple example to guide us, let us consider the types of instructions that must be included in a practical computer. A computer should have a set of instructions that allows the user to formulate any data processing task. Another way to view it is to consider the capabilities of a high-level programming language. Any program written in a high-level language must be translated into machine language to be executed. Thus, the set of machine instructions must be sufficient to express

any of the instructions from a high-level language. With this in mind we can categorize instruction types as follows:

- **Data processing:** Arithmetic and logic instructions
- **Data storage:** Memory instructions
- **Data movement:** I/O instructions
- **Control:** Test and branch instructions

Arithmetic instructions provide computational capabilities for processing numeric data. *Logic* (Boolean) instructions operate on the bits of a word as bits rather than as numbers; thus, they provide capabilities for processing any other type of data the user may wish to employ. These operations are performed primarily on data in processor registers. Therefore, there must be *memory* instructions for moving data between memory and the registers. *I/O* instructions are needed to transfer programs and data into memory and the results of computations back out to the user. *Test* instructions are used to test the value of a data word or the status of a computation. *Branch* instructions are then used to branch to a different set of instructions depending on the decision made.

We will examine the various types of instructions in greater detail later in this chapter.

Number of Addresses

One of the traditional ways of describing processor architecture is in terms of the number of addresses contained in each instruction. This dimension has become less significant with the increasing complexity of processor design. Nevertheless, it is useful at this point to draw and analyze this distinction.

What is the maximum number of addresses one might need in an instruction? Evidently, arithmetic and logic instructions will require the most operands. Virtually all arithmetic and logic operations are either unary (one source operand) or binary (two source operands). Thus, we would need a maximum of two addresses to reference source operands. The result of an operation must be stored, suggesting a third address, which defines a destination operand. Finally, after completion of an instruction, the next instruction must be fetched, and its address is needed.

This line of reasoning suggests that an instruction could plausibly be required to contain four address references: two source operands, one destination operand, and the address of the next instruction. In practice, four-address instructions are extremely rare. Most instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the program counter).

Figure 10.3 compares typical one-, two-, and three-address instructions that could be used to compute $Y = (A - B)/[C + (D \times E)]$. With three addresses, each instruction specifies two source operand locations and a destination operand location. Because we choose not to alter the value of any of the operand locations, a temporary location, T, is used to store some intermediate results. Note that there are four instructions and that the original expression had five operands.

Three-address instruction formats are not common because they require a relatively long instruction format to hold the three address references. With two-address instructions, and for binary operations, one address must do double duty as

Instruction	Comment
SUB Y, A, B	Y ← A − B
MPY T, D, E	T ← D × E
ADD T, T, C	T ← T + C
DIV Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction	Comment
MOVE Y, A	Y ← A
SUB Y, B	Y ← Y − B
MOVE T, D	T ← D
MPY T, E	T ← T × E
ADD T, C	T ← T + C
DIV Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction	Comment
LOAD D	AC ← D
MPY E	AC ← AC × E
ADD C	AC ← AC + C
STOR Y	Y ← AC
LOAD A	AC ← A
SUB B	AC ← AC − B
DIV Y	AC ← AC ÷ Y
STOR Y	Y ← AC

(c) One-address instructions

Figure 10.3 Programs to Execute $Y = \frac{A - B}{C + (D \times E)}$

both an operand and a result. Thus, the instruction SUB Y, B carries out the calculation Y − B and stores the result in Y. The two-address format reduces the space requirement but also introduces some awkwardness. To avoid altering the value of an operand, a MOVE instruction is used to move one of the values to a result or temporary location before performing the operation. Our sample program expands to six instructions.

Simpler yet is the one-address instruction. For this to work, a second address must be implicit. This was common in earlier machines, with the implied address being a processor register known as the **accumulator** (AC). The accumulator contains one of the operands and is used to store the result. In our example, eight instructions are needed to accomplish the task.

It is, in fact, possible to make do with zero addresses for some instructions. Zero-address instructions are applicable to a special memory organization, called a *stack*. A stack is a last-in-first-out set of locations. The stack is in a known location and, often, at least the top two elements are in processor registers. Thus, zero-address instructions would reference the top two stack elements. Stacks are described in Appendix 10A. Their use is explored further later in this chapter and in Chapter 11.

Table 10.1 summarizes the interpretations to be placed on instructions with zero, one, two, or three addresses. In each case in the table, it is assumed that the address of the next instruction is implicit, and that one operation with two source operands and one result operand is to be performed.

The number of addresses per instruction is a basic design decision. Fewer addresses per instruction result in instructions that are more primitive, requiring a less complex processor. It also results in instructions of shorter length. On the other hand, programs contain more total instructions, which in general results in longer execution times and longer, more complex programs. Also, there is an important

Table 10.1 Utilization of Instruction Addresses (Nonbranching Instructions)

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	AC ← AC OP A
0	OP	T ← (T − 1) OP T

AC = accumulator
T = top of stack
(T − 1) = second element of stack
A, B, C = memory or register locations

threshold between one-address and multiple-address instructions. With one-address instructions, the programmer generally has available only one general-purpose register, the accumulator. With multiple-address instructions, it is common to have multiple general-purpose registers. This allows some operations to be performed solely on registers. Because register references are faster than memory references, this speeds up execution. For reasons of flexibility and ability to use multiple registers, most contemporary machines employ a mixture of two- and three-address instructions.

The design trade-offs involved in choosing the number of addresses per instruction are complicated by other factors. There is the issue of whether an address references a memory location or a register. Because there are fewer registers, fewer bits are needed for a register reference. Also, as we shall see in the next chapter, a machine may offer a variety of addressing modes, and the specification of mode takes one or more bits. The result is that most processor designs involve a variety of instruction formats.

Instruction Set Design

One of the most interesting, and most analyzed, aspects of computer design is instruction set design. The design of an instruction set is very complex because it affects so many aspects of the computer system. The instruction set defines many of the functions performed by the processor and thus has a significant effect on the implementation of the processor. The instruction set is the programmer's means of controlling the processor. Thus, programmer requirements must be considered in designing the instruction set.

It may surprise you to know that some of the most fundamental issues relating to the design of instruction sets remain in dispute. Indeed, in recent years, the level of disagreement concerning these fundamentals has actually grown. The most important of these fundamental design issues include the following:

- **Operation repertoire:** How many and which operations to provide, and how complex operations should be
- **Data types:** The various types of data upon which operations are performed
- **Instruction format:** Instruction length (in bits), number of addresses, size of various fields, and so on

- **Registers:** Number of processor registers that can be referenced by instructions, and their use
- **Addressing:** The mode or modes by which the address of an operand is specified

These issues are highly interrelated and must be considered together in designing an instruction set. This book, of course, must consider them in some sequence, but an attempt is made to show the interrelationships.

Because of the importance of this topic, much of Part Three is devoted to instruction set design. Following this overview section, this chapter examines data types and operation repertoire. Chapter 11 examines addressing modes (which includes a consideration of registers) and instruction formats. Chapter 13 examines the reduced instruction set computer (RISC). RISC architecture calls into question many of the instruction set design decisions traditionally made in commercial computers.

10.2 TYPES OF OPERANDS

Machine instructions operate on data. The most important general categories of data are

- Addresses
- Numbers
- Characters
- Logical data

We shall see, in discussing addressing modes in Chapter 11, that addresses are, in fact, a form of data. In many cases, some calculation must be performed on the operand reference in an instruction to determine the main or virtual memory address. In this context, addresses can be considered to be unsigned integers.

Other common data types are numbers, characters, and logical data, and each of these is briefly examined in this section. Beyond that, some machines define specialized data types or data structures. For example, there may be machine operations that operate directly on a list or a string of characters.

Numbers

All machine languages include numeric data types. Even in nonnumeric data processing, there is a need for numbers to act as counters, field widths, and so forth. An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited. This is true in two senses. First, there is a limit to the magnitude of numbers representable on a machine and second, in the case of floating-point numbers, a limit to their precision. Thus, the programmer is faced with understanding the consequences of rounding, overflow, and underflow.

Three types of numerical data are common in computers:

- Integer or fixed point
- Floating point
- Decimal

We examined the first two in some detail in Chapter 9. It remains to say a few words about decimal numbers.

Although all internal computer operations are binary in nature, the human users of the system deal with decimal numbers. Thus, there is a necessity to convert from decimal to binary on input and from binary to decimal on output. For applications in which there is a great deal of I/O and comparatively little, comparatively simple computation, it is preferable to store and operate on the numbers in decimal form. The most common representation for this purpose is **packed decimal.**[1]

With packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way, with two digits stored per byte. Thus, 0 = 0000, 1 = 0001, ..., 8 = 1000, and 9 = 1001. Note that this is a rather inefficient code because only 10 of 16 possible 4-bit values are used. To form numbers, 4-bit codes are strung together, usually in multiples of 8 bits. Thus, the code for 246 is 0000 0010 0100 0110. This code is clearly less compact than a straight binary representation, but it avoids the conversion overhead. Negative numbers can be represented by including a 4-bit sign digit at either the left or right end of a string of packed decimal digits. For example, the code 1111 might stand for the minus sign.

Many machines provide arithmetic instructions for performing operations directly on packed decimal numbers. The algorithms are quite similar to those described in Section 9.3 but must take into account the decimal carry operation.

Characters

A common form of data is text or character strings. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems. Such systems are designed for binary data. Thus, a number of codes have been devised by which characters are represented by a sequence of bits. Perhaps the earliest common example of this is the Morse code. Today, the most commonly used character code in the International Reference Alphabet (IRA), referred to in the United States as the American Standard Code for Information Interchange (ASCII; see Table 7.1). Each character in this code is represented by a unique 7-bit pattern; thus, 128 different characters can be represented. This is a larger number than is necessary to represent printable characters, and some of the patterns represent *control* characters. Some of these control characters have to do with controlling the printing of characters on a page. Others are concerned with communications procedures. IRA-encoded characters are almost always stored and transmitted using 8 bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection. In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity).

---

[1]Textbooks often refer to this as binary coded decimal (BCD). Strictly speaking, BCD refers to the encoding of each decimal digit by a unique 4-bit sequence. Packed decimal refers to the storage of BCD-encoded digits using one byte for each two digits.

Note in Table 7.1 that for the IRA bit pattern 011XXXX, the digits 0 through 9 are represented by their binary equivalents, 0000 through 1001, in the rightmost 4 bits. This is the same code as packed decimal. This facilitates conversion between 7-bit IRA and 4-bit packed decimal representation.

Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC). EBCDIC is used on IBM mainframes. It is an 8-bit code. As with IRA, EBCDIC is compatible with packed decimal. In the case of EBCDIC, the codes 11110000 through 11111001 represent the digits 0 through 9.

Logical Data

Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometimes useful, however, to consider an *n*-bit unit as consisting of *n* 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be *logical* data.

There are two advantages to the bit-oriented view. First, we may sometimes wish to store an array of Boolean or binary data items, in which each item can take on only the values 1 (true) and 0 (false). With logical data, memory can be used most efficiently for this storage. Second, there are occasions when we wish to manipulate the bits of a data item. For example, if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations. Another example: To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte.

Note that, in the preceding examples, the same data are treated sometimes as logical and other times as numerical or text. The "type" of a unit of data is determined by the operation being performed on it. While this is not normally the case in high-level languages, it is almost always the case with machine language.

10.3 PENTIUM AND POWERPC DATA TYPES

Pentium Data Types

The Pentium can deal with data types of 8 (byte), 16 (word), 32 (doubleword), and 64 (quadword) bits in length. To allow maximum flexibility in data structures and efficient memory utilization, words need not be aligned at even-numbered addresses; doublewords need not be aligned at addresses evenly divisible by 4; and quadwords need not be aligned at addresses evenly divisible by 8. However, when data are accessed across a 32-bit bus, data transfers take place in units of doublewords, beginning at addresses divisible by 4. The processor converts the request for misaligned values into a sequence of requests for the bus transfer. As with all of the Intel 80x86 machines, the Pentium uses the little-endian style; that is, the least significant byte is stored in the lowest address (see Appendix 10B for a discussion of endianness).

The byte, word, doubleword, and quadword are referred to as general data types. In addition, the Pentium supports an impressive array of specific data types that are recognized and operated on by particular instructions. Table 10.2 summarizes these types.

Table 10.2 Pentium Data Types

Data Type	Description
General	Byte, word (16 bits), doubleword (32 bits), and quadword (64 bits) locations with arbitrary binary contents.
Integer	A signed binary value contained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer contained in a byte, word, or doubleword.
Unpacked binary coded decimal (BCD)	A representation of a BCD digit in the range 0 through 9, with one digit in each byte.
Packed BCD	Packed byte representation of two BCD digits; value in the range 0 to 99.
Near pointer	A 32-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.
Bit field	A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to $2^{32} - 1$ bits.
Byte string	A contiguous sequence of bytes, words, or doublewords, containing form zero to $2^{32} - 1$ bytes.
Floating point	See Figure 10.4.

Figure 10.4 illustrates the Pentium numerical data types. The signed integers are in twos complement representation and may be 16, 32, or 64 bits long. The floating-point type actually refers to a set of types that are used by the floating-point unit and operated on by floating-point instructions. The three floating-point representations conform to the IEEE 754 standard.

PowerPC Data Types

The PowerPC can deal with data types of 8 (byte), 16 (halfword), 32 (word), and 64 (doubleword) bits in length. Some instructions require that memory operands be aligned on a 32-bit boundary. In general, however, alignment is not required. One interesting feature of the PowerPC is that it can use either little-endian or big-endian style; that is, the least significant byte is stored in the lowest or highest address (see Appendix 10B for a discussion of endianness).

The byte, halfword, word, and doubleword are general data types. The processor interprets the contents of a given item of data depending on the instruction. The fixed-point processor recognizes the following data types:

- **Unsigned byte:** Can be used for logical or integer arithmetic operations. It is loaded from memory into a general register by zero extending on the left to the full register size.
- **Unsigned halfword:** As for unsigned byte, but for 16-bit quantities.

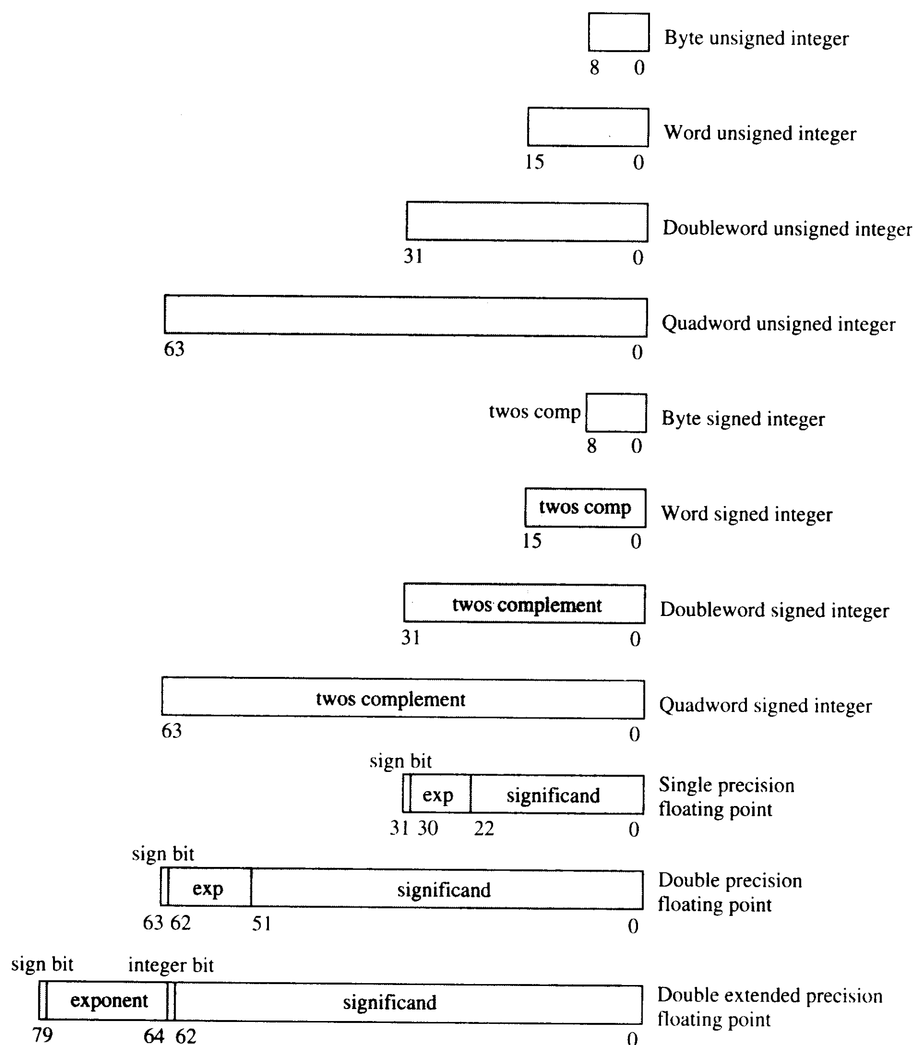


Figure 10.4 Pentium Numeric Data Formats

- **Signed halfword:** Used for arithmetic operations; loaded into memory by sign extending on the left to full register size (i.e., the sign bit is replicated in all vacant positions).
- **Unsigned word:** Used for logical operations and as an address pointer.
- **Signed word:** Used for arithmetic operations.
- **Unsigned doubleword:** Used as an address pointer.
- **Byte string:** From 0 to 128 bytes in length.

In addition, the PowerPC supports the single- and double-precision floating-point data types defined in IEEE 754.

10.4 TYPES OF OPERATIONS

The number of different opcodes varies widely from machine to machine. However, the same general types of operations are found on all machines. A useful and typical categorization is the following:

- Data transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System control
- Transfer of control

Table 10.3 (based on [HAYE98]) lists common instruction types in each category. This section provides a brief survey of these various types of operations, together with a brief discussion of the actions taken by the processor to execute a particular type of operation (summarized in Table 10.4). The latter topic is examined in more detail in Chapter 12.

Table 10.3 Common Instruction Set Operations

Type	Operation Name	Description
Data Transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand

(continued)

Table 10.3 Continued

Type	Operation Name	Description
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT (complement)	Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end
Transfer of Control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued
Input/Output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

Table 10.4 CPU Actions for Various Types of Operations

Data Transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of Control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

Data Transfer

The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things. First, the location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack. Second, the length of data to be transferred must be indicated. Third, as with all instructions with operands, the mode of addressing for each operand must be specified. This latter point is discussed in Chapter 11.

The choice of data transfer instructions to include in an instruction set exemplifies the kinds of trade-offs the designer must make. For example, the general location (memory or register) of an operand can be indicated in either the specification of the opcode or the operand. Table 10.5 shows examples of the most common IBM S/390 data transfer instructions. Note that there are variants to indicate the amount of data to be transferred (8, 16, 32, or 64 bits). Also, there are different instructions for register to register, register to memory, and memory to register transfers. In contrast, the VAX has a move (MOV) instruction with variants for different amounts of data to be moved, but it specifies whether an operand is register or memory as part of the operand. The VAX approach is somewhat easier for the programmer, who has fewer mnemonics to deal with. However, it is also somewhat less compact than the IBM S/390 approach because the location (register versus memory) of each operand must be specified separately in the instruction. We will return to this distinction when we discuss instruction formats, in the next chapter.

In terms of processor action, data transfer operations are perhaps the simplest type. If both source and destination are registers, then the processor simply causes data to be transferred from one register to another; this is an operation internal to

Table 10.5 Examples of IBM S/390 Data Transfer Operations

Operation Mnemonic	Name	Number of Bits Transferred	Description
L	Load	32	Transfer from memory to register
LH	Load Halfword	16	Transfer from memory to register
LR	Load	32	Transfer from register to register
LER	Load (Short)	32	Transfer from floating-point register to floating-point register
LE	Load (Short)	32	Transfer from memory to floating-point register
LDR	Load (Long)	64	Transfer from floating-point register to floating-point register
LD	Load (Long)	64	Transfer from memory to floating-point register
ST	Store	32	Transfer from register to memory
STH	Store Halfword	16	Transfer from register to memory
STC	Store Character	8	Transfer from register to memory
STE	Store (Short)	32	Transfer from floating-point register to memory
STD	Store (Long)	64	Transfer from floating-point register to memory

the processor. If one or both operands are in memory, then the processor must perform some or all of the following actions:

1. Calculate the memory address, based on the address mode (discussed in Chapter 11).
2. If the address refers to virtual memory, translate from virtual to actual memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

Arithmetic

Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably provided for signed integer (fixed-point) numbers. Often they are also provided for floating-point and packed decimal numbers.

Other possible operations include a variety of single-operand instructions; for example,

- **Absolute:** Take the absolute value of the operand.
- **Negate:** Negate the operand.
- **Increment:** Add 1 to the operand.
- **Decrement:** Subtract 1 from the operand.