To deal with negative numbers, we recognize that the remainder is defined by This is because the remainder is defined by

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

Consider the following examples of integer division with all possible combinations of signs of D and V:

$$\begin{array}{llllll} D = 7 & V = 3 & \Rightarrow & Q = 2 & R = 1 \\ D = 7 & V = -3 & \Rightarrow & Q = -2 & R = 1 \\ D = -7 & V = 3 & \Rightarrow & Q = -2 & R = -1 \\ D = -7 & V = -3 & \Rightarrow & Q = 2 & R = -1 \end{array}$$

The reader will note from Figure 9.17 that $(-7)/(3)$ and $(7)/(-3)$ produce different remainders. We see that the magnitudes of Q and R are unaffected by the input signs and that the signs of Q and R are easily derivable form the signs of D and V. Specifically, $\text{sign}(R) = \text{sign}(D)$ and $\text{sign}(Q) = \text{sign}(D) \times \text{sign}(V)$. Hence, one way to do twos complement division is to convert the operands into unsigned values and, at the end, to account for the signs by complementation where needed. This is the method of choice for the restoring division algorithm [PARH00].

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.

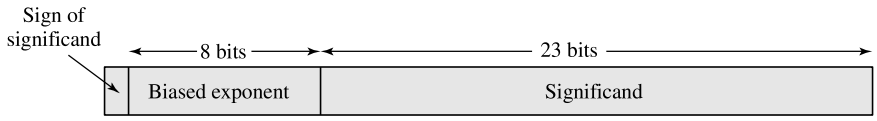
For decimal numbers, we get around this limitation by using scientific notation. Thus, 976,000,000,000,000 can be represented as 9.76×10^{14}, and 0.0000000000000976 can be represented as 9.76×10^{-14}. What we have done, in effect, is dynamically to slide the decimal point to a convenient location and 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



(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

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 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

[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.

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

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

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 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 $(2 - 2^{-23}) \times 2^{128}$

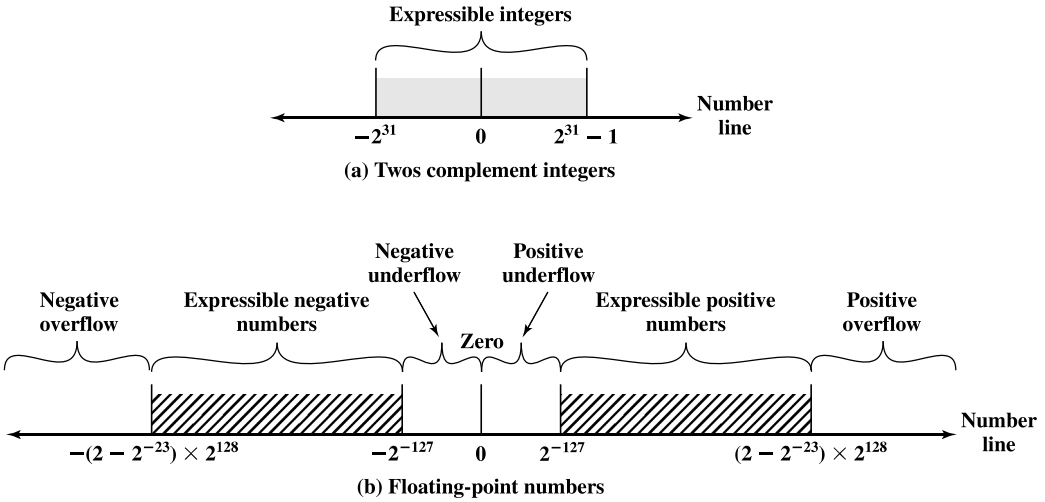


Figure 9.19 Expressible Numbers in Typical 32-Bit Formats

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
- 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. In practice, most floating-point numbers that one would wish to represent are represented only approximately. However, for moderate sized integers, the representation is exact.

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.

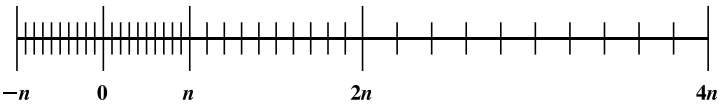


Figure 9.20 Density of Floating-Point Numbers

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.

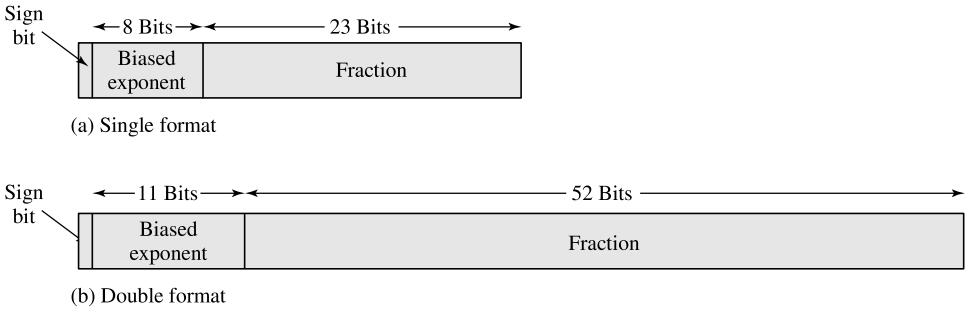


Figure 9.21 IEEE 754 Formats

Table 9.3 IEEE 754 Format Parameters

Parameter	Format			
	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

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 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.

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	$f \neq 0$	$-2^{e-1022}(0.f)$

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.

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.

Table 9.5 Floating-Point Numbers and Arithmetic Operations

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

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 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, 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.

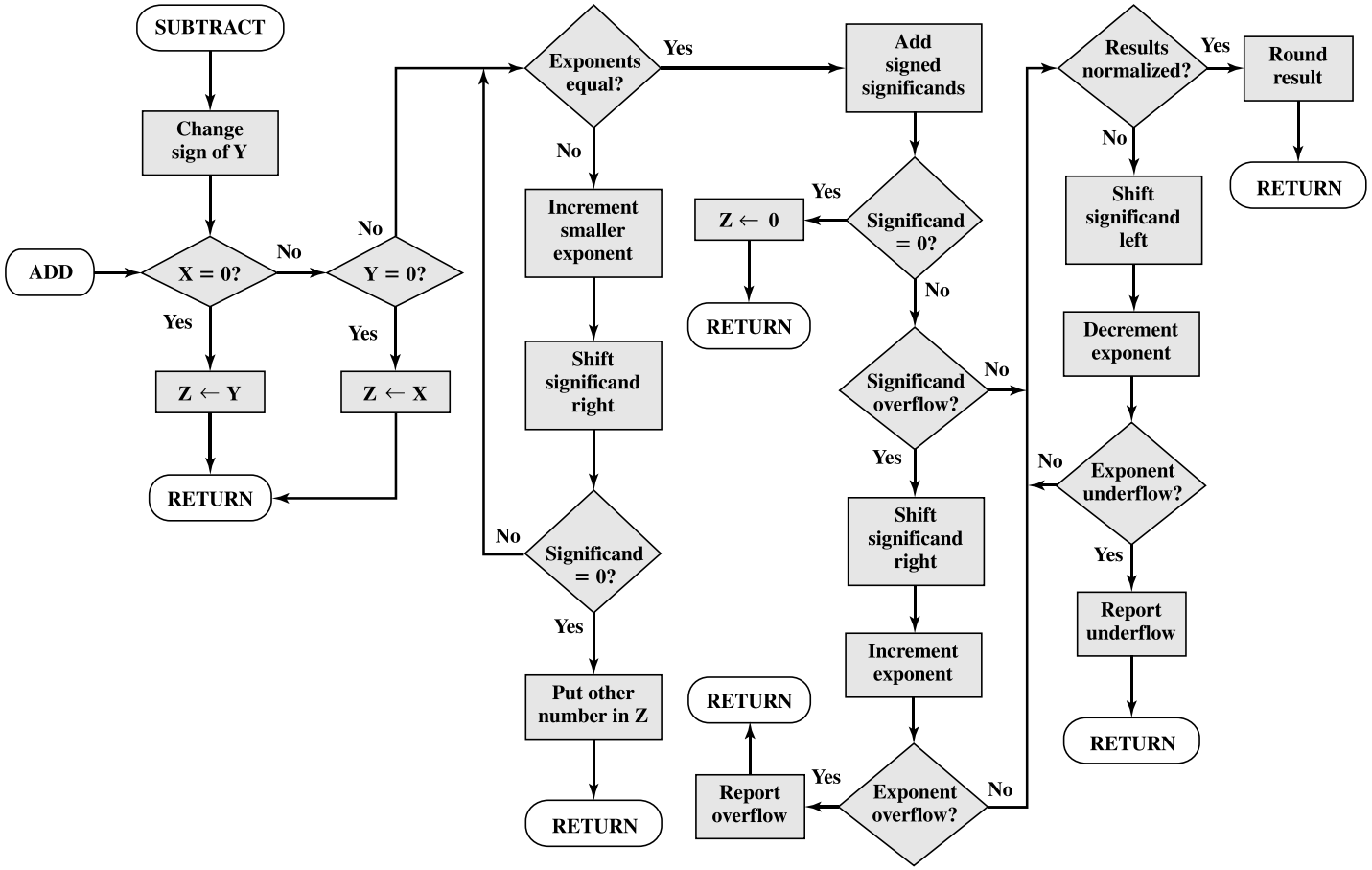


Figure 9.22 Floating-Point Addition and Subtraction ($Z \leftarrow Z \pm Y$)

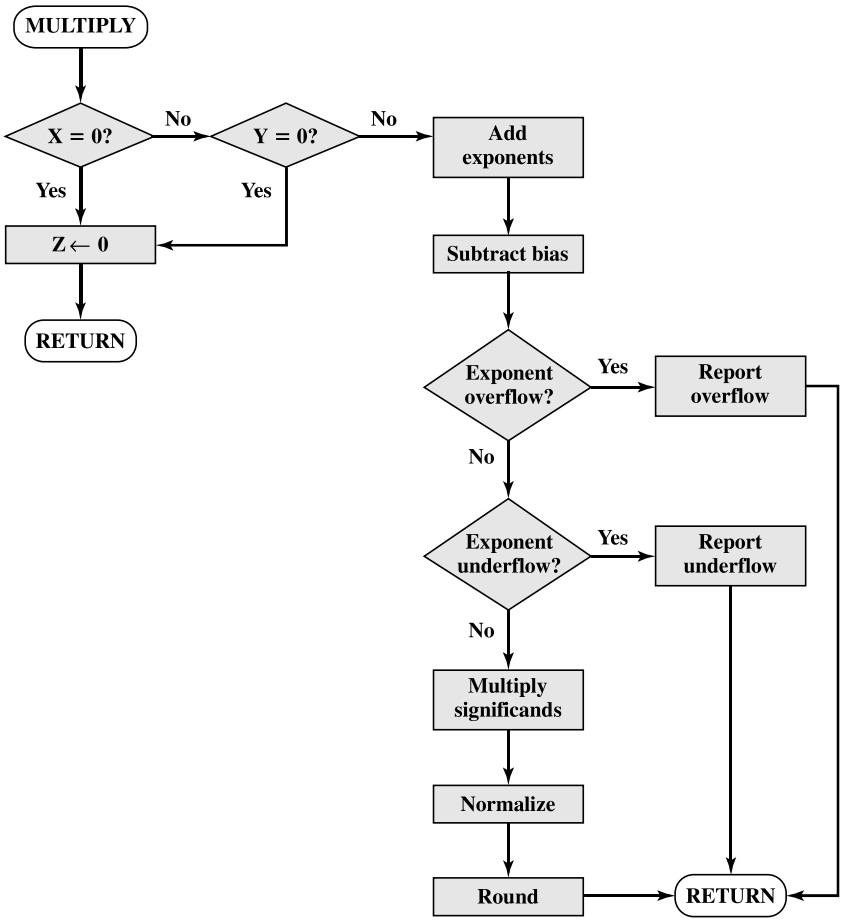


Figure 9.23 Floating-Point Multiplication ($Z \leftarrow X \times Y$)

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 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.

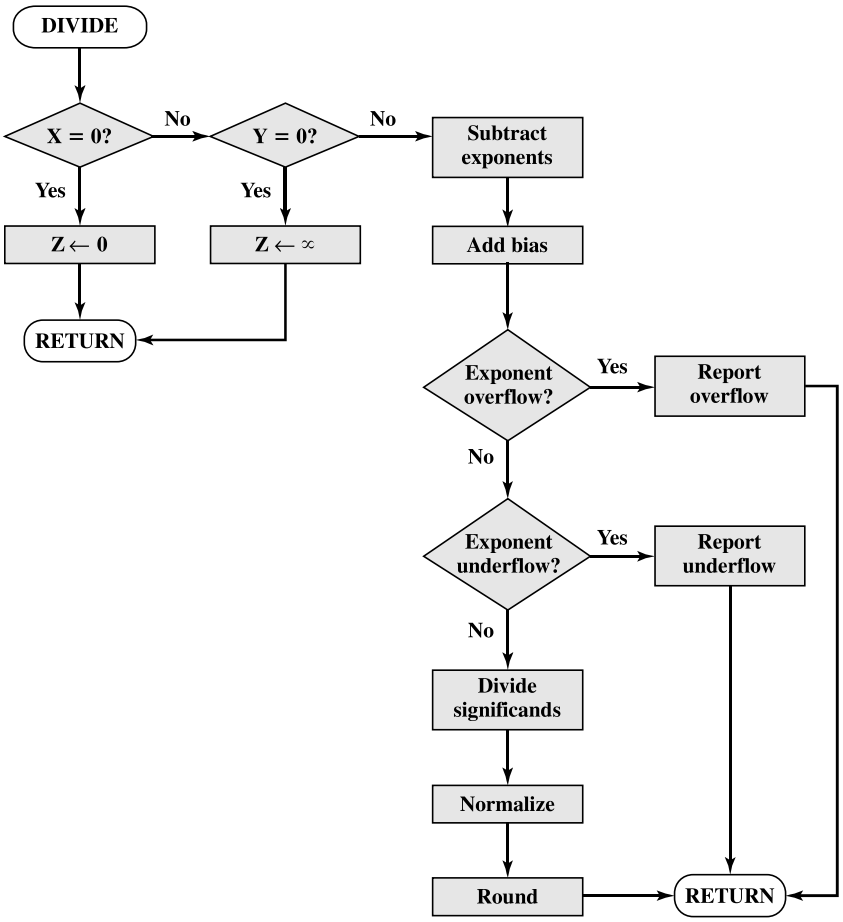


Figure 9.24 Floating-Point Division (Z ← X/Y)

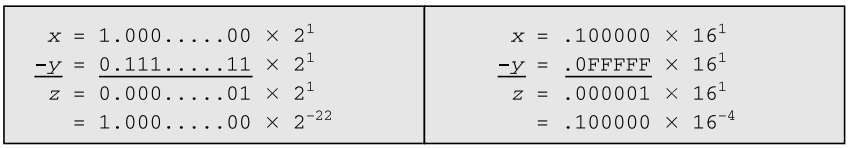
After the product is calculated, the result is then normalized and rounded, as was done for addition and subtraction. Note that normalization could result in exponent underflow.

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.

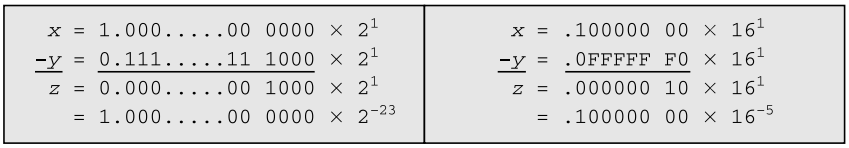
Precision Considerations

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



(a) Binary example, without guard bits

(c) Hexadecimal example, without guard bits



(b) Binary example, with guard bits

(d) Hexadecimal example, with guard bits

Figure 9.25 The Use of Guard Bits

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 $x = 1.00\ldots00 \times 2^1$ and $y = 1.11\ldots11 \times 2^0$. 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.
- **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:

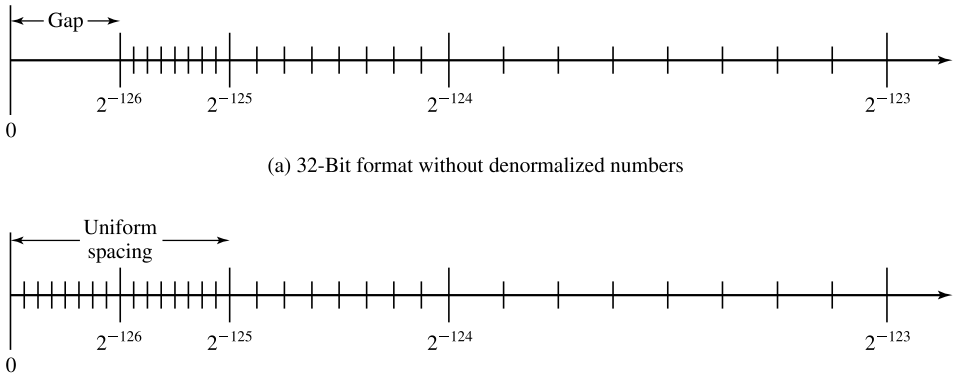
$$\begin{aligned} 5 + (+\infty) &= +\infty & 5 \div (+\infty) &= +0 \\ 5 - (+\infty) &= -\infty & (+\infty) + (+\infty) &= +\infty \\ 5 + (-\infty) &= -\infty & (-\infty) + (-\infty) &= -\infty \\ 5 - (-\infty) &= +\infty & (-\infty) - (+\infty) &= -\infty \\ 5 \times (+\infty) &= +\infty & (+\infty)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

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.

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 $x < 0$



(a) 32-Bit format without denormalized numbers

(b) 32-Bit format with denormalized numbers

Figure 9.26 The Effect of IEEE 754 Denormalized Numbers

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.

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 $2^{23} - 1$ 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 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 SITES

[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