Chapter 1

Numeric Data Representation

The primary problem in computer arithmetic is the mapping from the infinite number systems of mathematics to the finite representational capability of the machine. Finitude is the principal characteristic of a computer number system. Almost all other considerations are a direct consequence of this finitude. For example, overflow is simply an unsuccessful attempt to map from the infinite to the finite number system.

The common solution to this problem is the use of modular arithmetic. In this scheme, every integer from the infinite number set has one unique representation in a finite system. However, now a problem of multiple interpretations is introduced—that is, in a modulo 8 system, the number 9 is mapped into the number 1. As a result of mapping, the number 1 corresponds in the infinite number system to 1, 9, 17, 25, etc.

1.1 Number Systems

1.1.1 Natural Numbers

The historical need for and the use of numbers was for counting. Even nowadays, the child's numerical development starts with counting. The counting function is accomplished by the infinite set of numbers 1, 2, 3, 4, ... , which are described as natural numbers. These numbers have been used for thousands of years, and yet only in the 19^{th} century were they described precisely by Peano (1858–1932). The following description of Peano's postulates is adapted from Parker [28].

POSTULATE 1: For every natural number x, there is a unique natural number which we call the successor of x and which is denoted by x'.

POSTULATE 2: There is a unique natural number which we call *1*.

POSTULATE 3: The natural number 1 is not the successor of any natural number.

POSTULATE 4: If two natural numbers x and y are such that $x' = y'$, then $x = y$.

POSTULATE 5: (Principle of Mathematical Induction): Let M be a subset of the natural numbers with the following properties:

(a) 1 is a member of M;

(b) For any x that belongs to M, x' also belongs to M.

Then M is the set of natural numbers.

Later on, there will be a description of other number systems (negative, real, rational), and it will be shown that all other number systems can be described in terms of natural numbers. At this point, our attention is on the problem of mapping from the infinite set to a finite set of numbers.

1.1.2 Finitude

Garner [16] has shown that the most important characteristic of machine number systems is finitude. Overflows, underflows, scaling, and complement coding are consequences of this finitude.

1.1.3 Modular Arithmetic—Informal Description

Informally, the infinite set of natural numbers needs to be represented by a finite set of numbers. Arithmetic that takes place within a closed set of numbers is known as *modular arithmetic*. Brennan [4] provides the following examples of modular arithmetic in everyday life: The clock tells time in terms of the closed set (modules) of 12 hours, and the days of the week all fall within modulo 7. If the sum of any two numbers within such a modulus exceeds the modulus, only the remainder number is considered; e.g., eight hours after seven o'clock, the time is three o'clock, since

$$(8 + 7) \text{ modulo } 12 = \text{remainder of } \frac{15}{12} = 3.$$

Seventeen days after Tuesday, the third day of the week, the day is Friday, the sixth day of the week, since

$$(17 + 3) \text{ modulo } 7 = \text{remainder of } \frac{20}{7} = 6.$$

1.1.4 Modular Arithmetic—Formal Description

In modular arithmetic, the property of congruence (having the same remainder) is of particular importance. By definition [33]:

> *If m is a positive integer, then any two integers N and M are* congruent, *modulo m, if and only if there exists an integer K such that*
>
> $$N - M = Km.$$

or

$$N \bmod m \equiv M \bmod m,$$

where m is called the modulus.

Informally, the modulus is the quantity of numbers within which a computation takes place. $(0, 1, 2, 3, \ldots, m-1.)$

Example 1.1

If $m = 256$ and $M = 258$, $N = 514$, then

$$514 \, mod \, 256 = 2 \bmod 256$$

and

$$258 \, mod \, 256 = 2 \bmod 256,$$

i.e., they are congruent *mod* 256, and

$$514 - 258 = 1 \times 256,$$

i.e., K = 1. $\Diamond$

1.1.5 Properties

Congruence has the same properties with respect to the operations of addition, subtraction, and multiplication, or any combination.

If $N = N' \, mod \, m$ and $M = M' \, mod \, m$, then

$$\begin{aligned} (N + M) \bmod m &= (N' + M') \bmod m \\ (N - M) \bmod m &= (N' - M') \bmod m \\ (N * M) \bmod m &= (N' * M') \bmod m \end{aligned}$$

Example 1.2

If $m = 4$, $N' = 11$, $N = 3$, $M' = 5$, $M = 1$; then

$$\begin{aligned} 3 + 1 \, mod \, 4 &= (11 + 5) \, mod \, 4 \equiv 0 \\ 3 - 1 \, mod \, 4 &= (11 - 5) \, mod \, 4 \equiv 2 \\ 3 \times 1 \, mod \, 4 &= (11 \times 5) \, mod \, 4 \equiv 3 \end{aligned}$$

What if N is negative (assume m positive) in the operation $N \, mod \, m$? We could choose from several conventions; for example,

$$-7 \, mod \, 3 \equiv -1 \text{ or } +2,$$

since

$$-7/3 = -2 \text{ quotient}, \ -1 \text{ remainder}$$

or

$$-7/3 = -3 \text{ quotient}, \ +2 \text{ remainder}.$$

For modulus operations, the usual convention is to choose the *least positive residue* (including zero). Unless otherwise specified, we will assume this convention throughout this book, even if the modulus is negative; for example, $-7 \, mod \, -3 = +2$. That is,

$$\frac{-7}{-3} = +3 \text{ quotient}, \ +2 \text{ remainder}.$$

In terms of conventional division, this is surprising, since one might expect

$$\frac{-7}{-3} = +2 \text{ quotient}, \ -1 \text{ remainder}.$$

$\Diamond$

We will distinguish between the two division conventions by referring to the former as *modulus division* and the latter as *signed division*. In signed division, the magnitude of the quotient is independent of the signs of the divisor and dividend. This distinction follows the work of Warren and his colleagues [44].

The division operation is defined as

$$\frac{a}{b} = q + \frac{r}{b},$$

where q is the quotient and r is the remainder. But even the modulus division operation does not extend as simply as the other three operations; for example,

$$\frac{3}{1} \neq \frac{11}{5} \, mod \, 4.$$

Nevertheless, division is a central operation in modular arithmetic. It can be shown that for any modulus division M/m, there is a unique quotient–remainder pair, and the remainder has one of the m possible values $0, 1, 2, \ldots, m-1$. This leads to the concept of *residue class*.

A residue class is the set of all integers having the same remainder upon division by the modulus m. For example, if $m = 4$, then the numbers 1, 5, 9, 13 ... are of the same residue class. Obviously, there are exactly m residue classes, and each integer belongs to one and only one residue class. Thus, the modulus m partitions the set of all integers into m distinct and disjoint subsets called residue classes.

Example 1.3

If $m = 4$, then there are four residue classes which partition the integers:

$$\{\ldots, -8, -4, 0, 4, 8, 12, \ldots\}$$
$$\{\ldots, -7, -3, 1, 5, 9, 13, \ldots\}$$
$$\{\ldots, -6, -2, 2, 6, 10, 14, \ldots\}$$
$$\{\ldots, -5, -1, 3, 7, 11, 15, \ldots\}$$

In conclusion, by not dealing with individual integers but only with the residue class of which an integer is a member, the problem of working with an infinite set is reduced to one of working with a finite set. ♢

1.1.6 Extending Peano's Numbers

Peano's numbers are the natural integers 1, 2, 3, ..., but in real life we deal with more numbers. The historic motivation for the extension can be understood by studying some arithmetic operations. The operations of addition and multiplication (on Peano's numbers) result in numbers that are still described by the original Peano's postulates. However, subtraction of two numbers may result in negative numbers or zero. Thus, the extended set of all integers is

$$-\infty, \ldots, -2, -1, 0, 1, 2, \ldots + \infty,$$

and natural integers are a subset of these integers. The operation of division on integers may result in noninteger numbers; by definition such a number is a rational number, which can be represented exactly as a ratio of two integers. However, if the rational number is to be approximated as a single number, an infinite sequence of digits may be required for such a number, for example, $1/3 = 0.33333\ldots$. Between any two rational numbers, however small but finite their difference, lies an infinite number of other rational numbers and infinitely more numbers which cannot be expressed as rationals. We call these latter numbers *real* numbers and they include such constants as π and e. Real numbers can be viewed as all points along the number axis from $-\infty$ to $+\infty$.

Real numbers need to be represented in a machine with the characteristics of finitude. This is accomplished by approximating real numbers and rational numbers by terminating sequences of digits. Thus, all numbers (real, rational, and integers) can be operated on as if they were integers (provided scaling and rounding are done properly).

1.2 Integer Representation

The data representation to be described here is a weighted positional representation. The development for a weighted system was a particular breakthrough in ancient man's way of counting. While his hearthmate was simmering clams, and children demanding equal portions, to count seventeen shells he may have counted the first ten and marked something in the sand (to indicate 10), then counted the remaining seven shells. If his mark on the sand happened to look like 1, he could easily have generated the familiar weighted positional number system. Our base 10 system (sometimes called the Arabic system) comes to us from North Africa, and is quite an improvement over earlier schemes such as Roman numbers. In a weighted positional system, the number N is the sequence of $m+1$ digits (d_m, $d_{m-1}\ldots$, d_2, d_1, d_0), which in base β can be computed to give $N = d_m \cdot \beta^m + d_{m-1}\ldots\beta^{m-1} + \ldots d_1 \cdot \beta + d_0$. The digit values for d_i may be any integer between 0 and $\beta - 1$. For example, in the familiar decimal system, the base is $\beta = 10$, and the 4-digit number 1736 is:

$$N = 1736 = 1 \times 10^3 + 7 \times 10^2 + 3 \times 10^1 + 6.$$

In the binary system, $\beta = 2$, and the 5-digit number 10010 is:

$$N = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 0 = 18 \quad \text{(base 10)}.$$

Other common number bases are octal (base = 8) and hexadecimal (base = 16).

The leading digit, d_m, is the most significant digit (MSD) or the most significant bit (MSB) for binary base—similarly, d_0 is designated as the least significant digit or bit—(LSD or LSB).

The preceding positional number system does not include a representation of negative numbers. Two methods are commonly used to represent signed numbers [16]:

1. *Magnitude plus sign*: Digits are represented according to the simple positional number system; an additional high-order symbol represents the sign. This code is natural for humans, but unnatural for a modular computer system.
2. *Complement codes*: Two types are commonly used; namely, *radix complement* code (RC) and *diminished radix* complement code (DRC). Complement coding is natural for computers, since no special sign symbology or computation is required. In binary arithmetic (base = 2), the RC code is called *two's complement* and the DRC is called *one's complement.*

1.2.1 Complement Coding

Suppose we had a modular number system with modulus $2M$. We could designate numbers in the range 0 to M as positive, and *treat* numbers $M+1$ to $2M-1$ as negative, since they lie in the same residue class as numbers $-(M-1)$ to -1:

$$-1 \bmod 2M = (2M-1) \bmod 2M;$$

$$-(M-1) \bmod 2M = (2M - M + 1) \bmod 2M.$$

Mapping these negative numbers into large positive residues is called *complement coding.* We deal with $2M - x$ rather than $-x$. But, because both representations are congruent, they produce the same modular results.

Of course, "overflows" are a problem. These are results that appear as correct representations mod$2M$, but are incorrect in our mapped mod$M$ system. If two positive or two negative numbers a and b have sum c, which exceeds $|M|$, overflow occurs and this must be detected.

The form of the modulus affects the coding and the implementation of the system.

1.2.2 Radix Complement Code—Subtraction Using Addition

Suppose N is a positive integer of the form

$$N = d_m \cdot \beta^m + d_{m-1} \cdot \beta^{m-1} + \cdots + d_0.$$

The maximum value N may assume is $\beta^{m+1} - 1$; i.e., where all the digit values (d_i) are equal to $\beta - 1$, their maximum value. Thus, $\beta^{m+1} > N \geq 0$.

Now, suppose we wish to represent $-N$, a negative $m+1$ digit number. We define the radix complement of N as

$$\mathrm{RC}(N) = \beta^{m+1} - N.$$

Clearly, the RC(N) is a nonnegative integer.

For ease of representation, let $n = m+1$; then $\mathrm{RC}(N) = \beta^n - N$. Assume β is even and suppose M and N are n-digit numbers. We wish to compute $M - N$, using the addition operation. M and N may be either positive or negative numbers, so long as

$$\frac{\beta^n}{2} - 1 \geq M,\ N \geq \frac{-\beta^n}{2}.$$

Then

$$M - N$$

is more accurately

$$(M - N)\ mod\ \beta^n,$$

and

$$(M - N)\ mod\ \beta^n = (M\ mod\ \beta^n - N\ mod\ \beta^n)\ mod\ \beta^n;$$

but, if we replace $-N$ with $\beta^n - N$, the equality is unchanged; that is, by taking

$$(M\ mod\ \beta^n + (\beta^n - N)\ mod\ \beta^n)\ mod\ \beta^n,$$

we get

$$M\ mod\ \beta^n - N\ mod\ \beta^n.$$

The computation of $\beta^n - N$ is relatively straightforward. For N less than β^n, let N be represented as $X_m \ldots X_0$, and the operation $\beta^n - N$ is actually:

$$\begin{array}{l} \overbrace{}^{n\ \text{digits}} \\ 1\ 0\ \ 0\ 0\ 0\ \ldots 0 \\ -\ X_m\ X\ X\ X_i\ \ldots\ X_0 \\ \hline \end{array}$$

recall $m = n - 1$.

Now the radix complement of any digit X_i is designated RC(X_i). For all lower order digits which satisfy

$$X_0 = X_1 = \cdots = X_i = 0,$$

the RC(X_i) is

$$\mathrm{RC}(X_0) = \mathrm{RC}(X_1) = \cdots = \mathrm{RC}(X_i) = 0.$$

That is,

$$\begin{array}{r} 0\ 0 \\ -\ 0\ 0 \\ \hline 0\ 0. \end{array}$$

For $X_{i+1} \neq 0$, the first (lower order) nonzero element in N,

$$\mathrm{RC}(X_{i+1}) = \beta - X_{i+1},$$

and for all elements X_j thereafter, $m \geq j \geq i+2$,

$$\mathrm{RC}(X_j) = \beta - 1 - X_j.$$

For example, in a three-position decimal number system, the radix complement of the positive number 245 is $1000 - 245 = 755$. This illustrates that by properly scaling the represented positive and negative numbers about zero, no special treatment of the sign is required. Thus, in radix complement code, the most significant digit indicates the sign of the number. In the base 10 system, the digits 5, 6, 7, 8, 9 (in the most significant position) indicate negative numbers; i.e., the three digits represent numbers from $+499$ to -500, and in the binary system, the digit 1 is an indication of negative numbers.

Example 1.4

$M = +250, N = +245$; compute $M - N$.

$$\begin{array}{rcrl} 250 & \Rightarrow & 250 & \\ -245 & & +755 & \\ \hline & & 1005 & mod\ 1000 \equiv 5 \end{array}$$

$\Diamond$

For the familiar case of even radix, a disadvantage of the radix complement code is the asymmetry around zero; that is, the number of negative numbers is greater by one than the number of positive numbers. But this shortcoming is not a serious one, especially if the number zero is viewed as a positive number; then there are as many positive numbers as there are negative numbers.

The greatest disadvantage of the two's complement number system is the difficulty in converting from positive to negative numbers, and vice versa. This difficulty is the motivation [38] for developing the diminished radix complement code.

1.2.3 Diminished Radix Complement Code

By definition, the diminished radix complement of the previously defined number N, $\mathrm{DRC}(N)$ is $\beta^n - 1 - N$. In a decimal number system, this code is called *nine's complement*, and in binary system, it is called *one's complement*.

The computation of the diminished radix complement (DRC) is simpler than that of the radix complement. Since, if $N\ mod\ b^n = X_{n-1}X_{n-2}\ldots X_0$, then for all X_i $(n-1 \geq i \geq 0)$

$$\mathrm{DRC}(X_i) = \beta - 1 - X_i.$$

Since $\beta - 1$ is the highest valued symbol in a radix β system, no borrows can occur and the DRC digits can be computed independently.

This simplicity of complement computation comes at some expense in arithmetic operation, since the arithmetic logic itself is always mod β^p, (where $p > n$). Consider the computation

$P\,mod\,(\beta^n - 1)$. If P were initially represented as a mod β^p number, or the result of addition or subtraction of two numbers mod β^n, then the conversion to a mod $\beta^n - 1$ number, P', would be

$$\text{If } P < \beta^n - 1 \quad \text{then} \quad P = P'.$$

That is,

$$P\,mod\,\beta^n \equiv P\,mod\,(\beta^n - 1) = P'.$$

If $P > \beta^n - 1$, then P' must be increased by 1 (called the end around carry) for each multiple of $\beta^n - 1$ contained in P. Thus,

$$P' = \left(P + \lfloor \frac{P}{\beta^n - 1} \rfloor\right) \bmod \beta^n.$$

That is, P' is P plus the largest integer contained by $\frac{\mathrm{P}}{\beta^n - 1}$.

(Throughout this book, we use two symbols: $\lceil x \rceil$ and $\lfloor x \rfloor$, respectively the ceiling and the floor of the real number x. *The ceiling function* is defined as the smallest integer that properly contains x; e.g., if $x = 1.33$, then $\lceil x \rceil = \lceil 1.33 \rceil = 2$. *The floor function* is defined as the largest integer contained by x, e.g., if $\lfloor x \rfloor = \lfloor 1.33 \rfloor = 1$.)

Finally, if

$$P = k(\beta^n - 1),$$

k equal to any integer, then

$$P' = 0.$$

Example 1.5

Suppose we have two mod 99 numbers A' and B$'$, having the following operations performed mod 1000, and then corrected to mod 100 and then to a mod 99 result:

(i) $A' = 47$, B$' = 24$; find $(A' + \mathrm{B}')\,mod\,99$.

$$\begin{array}{r} 47 \\ +24 \\ \hline 071 \end{array} \qquad 71\,mod\,100 \equiv 71\,mod\,99 = \text{result.}$$

(ii) $A' = 47$, B$' = 57$; find $(A' + \mathrm{B}')\,mod\,99$.

$$\begin{array}{r} 47 \\ +57 \\ \hline 104 \end{array} \qquad 4\,mod\,100 \equiv 5\,mod\,99 = \text{result.}$$

$$\begin{array}{r} +1 \\ \hline 05 \end{array}$$

(iii) $A' = 47$, B$' = 52$; find $(A' + \mathrm{B}')\,mod\,99$.

$$\begin{array}{r} 47 \\ +52 \\ \hline 099 \end{array} \qquad 99\,mod\,100 \equiv 0\,mod\,99 = \text{result.}$$

$\Diamond$

Since $\beta^n - 1$ is a represented element in n-digit arithmetic (mod β^n arithmetic), we have two representations for zero: $\beta^n - 1$ and 0.

While the problem of $\beta^n - 1$ and β^n modular compatibility will be of interest to us in Chapter 2, the use of the DRC in subtraction provides a more restricted version of this problem. In order to represent negative numbers using the DRC, we will partition the range of β^n representation as follows:

$\beta^n - 1, \ldots\ldots\ldots, \frac{\beta^n}{2}$	$\frac{\beta^n}{2} - 1, \ldots\ldots\ldots, 1, 0$
0 max. neg.	max. pos. 0
negative	positive

Thus, any m-digit ($m = n - 1$) number M must be in the following range:

$$\frac{\beta^n}{2} - 1 \geq M \geq \frac{-\beta^n}{2} + 1.$$

Note that $\frac{\beta^n}{2}$ is congruent to (lies in the same residue class as) $\frac{-\beta^n}{2} + 1$ modulo $\beta^n - 1$, since

$$(\frac{-\beta^n}{2} + 1)\, mod\, (\beta^n - 1) \equiv ((\beta^n - 1) - \frac{\beta^n}{2} + 1)\, mod\, (\beta^n - 1) \equiv (\frac{\beta^n}{2}\, mod\, \beta^n - 1.$$

So long as β has 2 as a factor, there will be a unique set of leading digit identifiers for negative numbers. For example, if $\beta = 10$, a negative (nonpositive) number will have 5, 6, 7, 8, 9 as a leading digit.

Consider the computation $M - N$ using the diminished radix complement (DRC) with mod b^n arithmetic logic to be corrected to mod $\beta^n - 1$. M and N lie within the previously defined range.

$$(M - N)\, mod\, (\beta^n - 1) \equiv (M\, mod\, (\beta^n - 1) - N\, mod\, (\beta^n - 1))\ mod\, (\beta^n - 1).$$

Then

$$M\, mod\, (\beta^n - 1) \equiv M$$

and

$$-N\, mod\, (\beta^n - 1) = \beta^n - 1 - N$$

and

$$M - N = M + \beta^n - 1 - N;$$

that is,

$$M + \mathrm{DRC}(N).$$

Since the basic addition logic is performed mod β^n, we correct the mod β^n difference, D, as follows to find D$'$, the mod $\beta^n - 1$ difference:

$$\mathrm{D} = M + \beta^n - 1 - N.$$

If $D > \beta^n - 1$, then

$$D' = D + 1; \quad \text{i.e., } M - N > 0.$$

If $D < \beta^n - 1$, then

$$D' = D; \quad \text{i.e., } M - N < 0.$$

If $D = \beta^n - 1$, then

$$D' = 0; \quad \text{i.e., } M = N,$$

and *the result is zero* (i.e., one of the two representations).

In summary, in the decimal system $-43 \Rightarrow 99 - 43 = 56$, and in the binary system $-3 \Rightarrow 111 - 011 = 100$. These examples illustrate the advantage of the diminished radix complement code—the ease of initial conversion from positive to negative numbers; the conversion is done by taking the complement of each digit. Of course, in the binary system, the complement is the simple Boolean NOT operation.

A disadvantage of the system is illustrated by taking the complement of zero; for example, in a 3-digit decimal system, the complement of zero $= 999 - 000 = 999$. Thus, the number zero has two representations: 000 and 999. (Note: the complement of the new zero is $999 - 999 = 000$.)

Another disadvantage is that the arithmetic logic may require correction of results (end-around carry)—see Chapter 3.

1.3 Implementation of Integer Operations

For each integer data representation, five operations will be analyzed: addition, subtraction, shifting, multiplication, and division. Most of the discussion will assume binary arithmetic (radix 2).

Addition and subtraction will be treated together, since the subtraction is the same as addition of two numbers of opposite signs. Thus, subtraction is performed by adding the negative of the subtrahend to the minuend. Therefore, the first thing to be addressed is the negation operation in each data representation.

1.3.1 Negation

In a one's complement system, negation is a simple Boolean NOT operation. Negation in a two's complement (TC) system can be viewed as

$$\text{TC}(N) = 2^n - N = (2^n - 1 - N) + (1),$$

where n is the number of digits in the representation. It may look awkward in the equation, but in practice this form is easier to implement, since the first term is the simple one's complement (i.e., NOT operation) and the second term calls for adding one to the least significant bit (LSB). The discussion of one's and two's complement operations follows Stone [38].

1.3.2 Two's Complement Addition

Two's complement addition is performed as if the two numbers were unsigned numbers; that is, no correction is required. However, it is necessary to determine when an overflow occurs. For the two summands B and C, there are four cases to consider:

Case	B	C	Comments
1	Positive	Positive	
2	Negative	Negative	
3	Negative	Positive	$\|B\| > \|C\|$
4	Negative	Positive	$\|B\| < \|C\|$

For positive numbers, the sign bit (the MSB) is zero, and for negative numbers, the sign bit is one. The sign bit is added just like all the other bits. Thus, the sign bit of the final result is made up of the sum of the summands' sign bits plus the carry into the sign bit. In the first case, the sum of the sign bits is zero ($0 + 0 = 0$), and if a carry is generated by the remaining bits, the resultant sign bit will become one. That is, the result overflows (since adding the two positive numbers generated a negative number). The rest of the cases are analyzed in a similar fashion and summarized in the following table:

Case	B	C	Sum of Signs	Carry-in to Sign Bit C_{in}	Carry-out of Sign Bit C_{out}	Overflow	Notes
1a	Pos	Pos	0	0	0	no	
1b	Pos	Pos	0	1	0	yes	
2a	Neg	Neg	0	1	1	no	
2b	Neg	Neg	0	0	1	yes	
3	Neg	Pos	1	0	0	no	$\|B\| > \|C\|$
4	Neg	Pos	1	1	1	no	$\|B\| < \|C\|$

Two observations can be made from the above table: first, it is impossible to overflow the result when the two summands have different signs (this is quite clear intuitively); second, the overflow condition can be stated in terms of the carries in and out of the sign bit—that is, overflow occurs when these carries are different. The Boolean expression for this condition ($\oplus$ is the exclusive OR operation) is:

$$\text{OVERFLOW} = C_{in} \oplus C_{out}.$$

1.3.3 One's Complement Addition

It was mentioned earlier that addition in one's complement representation requires correction. An insight into the reason for correction can be obtained by analyzing the four cases as was done for the two's complement addition (for simplicity, the overflow cases are ignored).

Case 1: Same as two's complement addition, and no correction is required.

Case 2: Adding two negative numbers b, c;

$$\begin{aligned} \text{DRC}(|b|) + \text{DRC}(|c|) &= \text{DRC}(|b| + |c|) \\ \text{where DRC}(|x|) &= 2^n - 1 - |x| \end{aligned}$$

$$\begin{array}{rl} & 2^n \quad -1-|b| \\ +& 2^n \quad -1-|c| \\ \hline & 2^{n+1}-2-(|b|+|c|) \end{array}$$

In modulo 2^n, the number 2^{n+1} is represented by its congruent 2^n. Thus, the sum is $2^n - 2 - (|b| + |c|)$, but it should have the one's complement format $2^n - 1 - (|b| + |c|)$. Therefore, 1 must be added to the LSB to have the correct result.

Case 3: b negative, c positive, $|b| > |c|$.

$$\begin{array}{rl} & 2^n - 1 - |b| \\ + & \qquad |c| \\ \hline & 2^n - 1 - (|b| - |c|) \end{array}$$

This form requires no correction.

Case 4: b negative, c positive, $|b| < |c|$.

$$\mathrm{DRC}(|b|) + |c| = |c| - |b|.$$

$$\begin{array}{rl} & 2^n - 1 - |b| \\ + & \qquad |c| \\ \hline & 2^n - 1 + (|c| - |b|) \end{array}$$

But this result has to be positive, and correction is required. After the correction, the result is $2^n + (|c| - |b|)$, which is congruent to $|c| - |b|$.

The implementation of the correction term is relatively easy. Whenever correction is necessary there is a carry-out of the sign bit. Thus, in hardware the carry-out of the sign bit is added to the LSB (if no correction is required, zero is added to the LSB). The correction term is the end-around carry, and it causes one's complement addition to be slower than two's complement addition.

Overflow detection in one's complement addition is the same as in two's complement addition; that is, $\mathrm{OVR} = C_{\mathrm{in}} \oplus C_{\mathrm{out}}$.

1.3.4 Computing Through the Overflows

This subject is covered in detail by Garner [18]. Here, we just state the main property. In complement-coded arithmetic, it is possible to perform a chain of additions, subtractions, multiplications, or any combination that will generate a final correct (representable) result, even though some of the intermediate results have overflowed.

For example, in 4-bit two's complement representation, where the range of representable numbers is -8 to $+7$, consider the following operation:

$$+5 + 4 - 6 = +3.$$

0101	+5
0100	+4
1001	Overflow
1010	−6
0011	+3 (correct)

1.3.5 Arithmetic Shifts

The arithmetic shifts are discussed as an introduction to the multiplication and division operations. An arithmetic left shift is equivalent to multiplying by the radix (assuming the shifted result does not overflow), and an arithmetic right shift is equivalent to dividing by the radix. In binary, shifting p places is equivalent to multiplying or dividing by 2^p. In left shifts (multiplying), zeros are shifted into the least significant bits, and in right shifts, the sign bit is shifted into the most significant bit (since the quotient will have the same sign as the dividend).

The difference between a logical and an arithmetic shift is important to note. In a logical shift, *all bits* of a word are shifted right or left by the indicated amount with zeros filling unreplaced end bits. In an arithmetic shift, the sign bit is fixed and the sign convention must be observed when filling unreplaced end bits. Thus, a right shift (divide) of a number will fix the sign bit and fill the higher order unreplaced bits with either ones or zeros in accordance with the sign bit. With arithmetic left shift, the lower order bits are filled with zeros regardless of the sign bit. So long as a p place left shift does not cause an overflow—i.e., $2^p *$ original value $\leq$ maximum representable number in the word—arithmetic left shift is the same as logical left shift.

In two's complement right shift, there is an asymmetry between the shifted results of positive and negative numbers:

$$-13/2 \Rightarrow 10011 \text{ right shift } 11001 \quad -7;$$

$$+13/2 \Rightarrow 01101 \text{ right shift } 00110 \quad +6.$$

This, of course, relates to the asymmetry of the two's complement data representation, where the quantity of negative numbers is larger by one than the quantity of positive numbers.

By contrast, the one's complement right shift is symmetrical:

$$-13/2 \Rightarrow 10010 \text{ right shift } 11001 \quad -6;$$

$$+13/2 \Rightarrow 01101 \text{ right shift } 00110 \quad +6.$$

Notice that the asymmetric resultant quotients correspond to modular division—i.e., creating a quotient so that the remainder is always positive. Similarly, symmetric quotients correspond to signed division—the remainder assumes the sign of the dividend.

1.3.6 Multiplication

In unsigned data representation, multiplying two operands, one with n bits and the other with m bits, requires that the result will be $n + m$ bits. If each of the two operands is n bits, then the product has to be $2n$ bits. This, of course, corresponds to the common notion that the multiplication product is a double-length operand. This should be clear from analyzing the multiplication of the two largest representable unsigned operands:

$$P = (2^n - 1) * (2^n - 1) = 2^{2n} - 2^{n+1} + 1 = \underbrace{2^{2n-1} + 2^{2n-1} - 2^{n+1}}_{\text{Positive number}} + 1.$$

Thus, the largest product P is $2^{2n} > P > 2^{2n-1}$, so $2n$ bits are necessary and sufficient to represent it.

In signed numbers, where the MSB of each of the operands is a sign bit, the product should require only $2n - 1$ bits, since the product has only one sign bit. However, in the two's complement code there is one exceptional case: multiplying -2^n by -2^n results in $+2^{2n}$. But this positive number is not representable in $2n - 1$ bits. This latter case is often treated as an overflow, especially in fractional representation where both operand and results are restricted to the range $-1 \leq R < +1$. Thus, multiplying -1 times -1 gives the unrepresentable $+1$.

1.3.7 Division

Division is the most difficult operation of the four basic arithmetic operations. Two properties of the division are the source for this difficulty:

1. Overflow—Even when the dividend is n bits long and the divisor is n bits long, an overflow may occur. A special case is a zero divisor.

2. Inaccurate results—In most cases, dividing two numbers gives a quotient that is an approximation to the actual rational number.

In general, one would like to think of division as the converse operation to multiplication but, by definition:

$$\frac{A}{B} = Q + \frac{R}{B},$$

or

$$A = B * Q + R,$$

where A is the dividend, B is the divisor, Q is the quotient, and R is the remainder. When $R = 0$, there is a subset of cases for which division is the exact converse of multiplication.

In terms of the natural integers (Peano's numbers), all multiplication results are still integers, but only a small subset of the division results are such numbers. The rest of the results are rational numbers, and to represent them accurately a pair of integers is required.

In terms of machine division, the result has to be expressed by one finite number. Going back to the definition of division,

$$\frac{A}{B} = Q + \frac{R}{B},$$

it is observed that the same equation holds true for any desired finite precision.

Example 1.6

In decimal arithmetic, if $A = 1$, $B = 7$, then $1/7$ is computed as follows:

$A/B{=}Q + R/B$	or	$A{=}B * Q + R$	
$1/7{=}0.1 + 0.3/7$	or	$1{=}0.7 + 0.3$	$Q{=}0.1$
$1/7{=}0.14 + 0.02/7$	or	$1{=}0.98 + 0.02$	$Q{=}0.14$
$1/7{=}0.142 + 0.006/7$	or	$1{=}0.994 + 0.006$	$Q{=}0.142$
$1/7{=}0.1428 + 0.0004/7$	or	$1{=}0.9996 + 0.0004$	$Q{=}0.1428$

$\Diamond$

In implementing a simple subtractive division algorithm, one more difficulty of the division becomes evident. Multiplication can be thought of as successive additions, and division is similarly successive subtractions. But while in multiplication it is known how many times to add, in division the quotient digits are not known in advance. It is not absolutely certain how many times it will be necessary to subtract the divisor from a given order of the dividend. Therefore, in these algorithms, which are trial and error processes, it is not known that the divisor has been subtracted a sufficient number of times until it has been subtracted once too often.

One more difficulty in division is the multiplicity of valid results depending upon the sign conventions, e.g., signed vs. modular division. Thus, if one wishes a signed division using the two's complement code, a negative quotient requires a correction by adding one to the least significant bit.

The difficulties encountered in performing division as a trial and error shift and subtract process are eliminated when a different approach to implementation is taken. The division of A/B can be treated as multiplication of A times the reciprocal of $B, (1/B)$. Thus, the problem is reduced to the computation of a reciprocal, which will be discussed in the chapter on division algorithms.

1.4 Floating Point Number Representation

1.4.1 Motivation and Terminology

So far, we have discussed fixed point numbers where the number is written as an integer string of digits and the radix point is a function of the interpretation. The problem with fixed point arithmetic is the lack of dynamic range, which can be illustrated by the following example in the decimal number system.

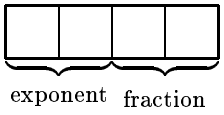
> Assume that there are four decimal digits. Then the dynamic range 9999 to 0 is $\simeq 10,000$. This range is independent of the decimal point position, that is, the dynamic range of 0.9999 to 0.0000 is also $\simeq 10,000$. Since this is a 4-digit number, we may want to represent during the same operation both 9999 and 0.0001; but this is impossible to do in fixed point arithmetic without scaling.

The above example illustrates the motivation for floating point representation: dynamic range. Floating point representation is similar to scientific notation; that is:

$$\text{fraction} \times (\text{radix})^{\text{exponent}}.$$

For example, the number 9999 is expressed as 0.9999×10^4. In a computer with floating point instructions, the radix is implicit, so only the fraction and the exponent need to be represented explicitly.

The floating point format for the above four decimal digits could be like this:



Let us now examine the dynamic range in this representation (assume positive numbers and positive exponents only). Note that *dynamic range* is the ratio of the largest number representation to the smallest (nonzero) number representation.

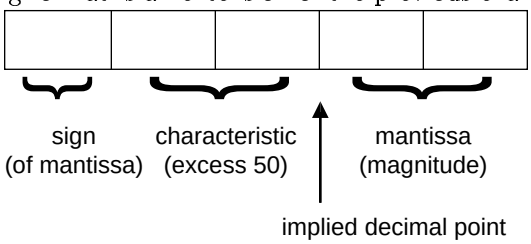
> Smallest (nonzero) number $= 0.01 \times 10^0 = 0.01$.
> Largest number $= 0.99 \times 10^{99}$, approximately 10^{99}.
> Therefore, the dynamic range is $10^{99}/0.01$, approximately 10^{101}.

Thus, in floating point representation, the dynamic range is several orders of magnitude larger then that of fixed point representation.

In practice, floating point numbers may have a negative fraction and negative exponent. There are many formats to represent these numbers, but most of them have the following properties in common:

1. The fraction is an unsigned number called the *mantissa*.
2. The exponent is represented by a *characteristic*, which is an excess code to accommodate both polarities of the exponents. The advantage of excess representation is that zero is represented by all zeros, and therefore smaller numbers (i.e., with increasing negative exponent) uniformly approach zero, thus simplifying compare logic.
3. The sign of the entire number is represented by the most significant bit of the number.

For example, the following format is an extension of the previous example:



The excess code is a method of representing both negative and positive exponents by adding bias to the exponent:

$$\text{bias} = \frac{1}{2}2^S$$

where S = number of exponent bits (with binary or binary-based radix, $\beta = 2^p$). For a non-binary based radix,

$$\text{bias} = \frac{1}{2}\beta^S,$$

where β is the radix and S is the number of exponent digits.

For the above format:

$$\text{bias } = \frac{1}{2}(10)^2 = \frac{1}{2}(100) = 50;$$

and the biased exponent is called the characteristic and is defined as follows:

$$\text{Characteristic} = \text{exponent} + \text{bias}.$$

Example 1.7

Exponent = 2 will result in

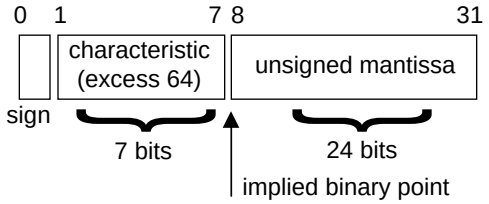
$$\text{Characteristic} = 2 + 50 = 52.$$

The mantissa is the magnitude of the fraction, and its sign is the MSD of the format. Numbers are defined as normalized if the MSD of the mantissa is nonzero. The number zero is represented by a zero mantissa and any characteristic; thus, there is no unique representation for zero. However, by definition, a normalized zero has zero characteristic, and a zero in the sign position. Following are some examples:

$0\ 51\ 78 \rightarrow +.78 \times 10^{1}$	$= +7.8$	
$0\ 52\ 07 \rightarrow +.07 \times 10^{2}$	$= +7.0$	unnormalized
$0\ 47\ 12 \rightarrow +.12 \times 10^{-3}$	$= .00012$	negative exponent
$1\ 51\ 78 \rightarrow -.78 \times 10^{1}$	$= -7.8$	negative number
$0\ 52\ 00 \rightarrow +.00 \times 10^{2}$	$= +0.0$	unnormalized zero
$0\ 00\ 00 \rightarrow +0.0 \times 10^{0}$	$= +0.0$	normalized zero

$\Diamond$

The same approach can be used in binary floating point numbers. Consider a 32-bit word, where 24 bits are the unsigned mantissa, 7 bits are the characteristic, and the MSB is the sign of the number, as follows:



The range of the representable numbers as determined by the exponent in this format is:

The largest exponent is $127 - 64 = 63$ and $2^{+63} \simeq 10^{+19}$.

The smallest exponent is $0 - 64 = -64$ and $2^{-64} \simeq 0.5 \times 10^{-20}$.

1.5 Properties of Floating Point Representation

1.5.1 Lack of Unique Representation

Generally, a floating point number is evaluated by the equation $M \times \beta^e$, where

$$\begin{aligned} M &= \text{mantissa} \\ \beta &= \text{radix} \\ e &= \text{exponent} \end{aligned}$$

In a 5-digit decimal floating point representation, the number 9 can be written as 0.9×10^1 or as 0.09×10^2. The lack of unique representation makes comparison of numbers difficult. Consequently, floating point numbers are usually represented in normalized form, where the mantissa is always represented by a nonzero most significant digit. Obviously, this rule could not apply to the case of zero. Therefore, by definition, normalized zero is represented by all zero digits (which simplifies zero detection circuitry). It is interesting to note that a normalized zero in floating point representation is designed to be identical to the fixed point representation of zero.

1.5.2 Range and Precision

Range is a pair of numbers (smallest, largest) which bounds all representable numbers in a given system. Precision is the gap between any two such representable numbers.

The largest number representable in any normalized floating point system is approximately equal to the radix raised to the power of the maximum positive exponent, and the absolute value of the smallest nonzero number is approximately equal to the radix raised to the power of the maximum negative exponent.

Let us translate the above paragraph into equations.

$$\boxed{max = \beta^{e_{\max}} * M_{\max}}$$

where

$$\begin{aligned} max &= \text{largest representable number} \\ \beta &= \text{Radix} \\ e_{\max} &= \text{largest exponent} \\ M_{\max} &= \text{largest Mantissa.} \end{aligned}$$

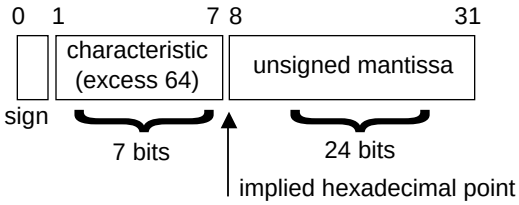
$$\boxed{min = \beta^{e_{\min}} * M_{\min}}$$

where

$$\begin{aligned} min &= \text{smallest nonzero number (absolute value),} \\ e_{\min} &= \text{smallest exponent,} \\ M_{\min} &= \text{smallest Mantissa (normalized).} \end{aligned}$$

Example 1.8

The following IBM System 370 (short) format is similar to the binary format at the end of the last section, except that the IBM radix is 16.



Since

$$\beta = 16,$$

then

$$e_{\max} = 63; \qquad M_{\max} = 1 - 16^{-6}.$$

Therefore, the largest representable number is

$$max = 16^{63} * (1 - 16^{-6}) \simeq 7.23 \times 10^{75}$$

and the smallest positive normalized number is

$$min = 16^{-64}(16^{-1}) \simeq 5.4 \times 10^{-79},$$

since

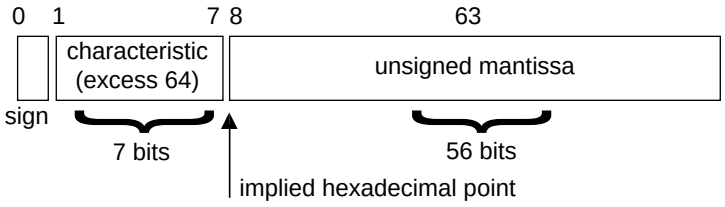
$$e_{\min} = -64; \qquad M_{\min} = 16^{-1}.$$

$\Diamond$

For a given radix, the range is mainly a function of the exponent. By contrast, the precision is a function of the mantissa. *Precision* is the resolution of the system, and it is defined as the minimum difference between two mantissa representations, which is equal to the value of the least significant bit of the mantissa. Precision is defined independently of the exponent; it depends only on the mantissa difference. In the IBM short format, there are 24 bits in the mantissa, therefore, the precision is 16^{-6}(or 2^{-24}) $\simeq 0.6 \times 10^{-7}$, or approximately seven significant decimal digits.

Example 1.9

More precision is obtained by extending the number of bits in the mantissa; for example, in the IBM System 370, one more word is added to the mantissa in its long format, that is, 32 more bits. The mantissa is 56 bits long and the precision is 16^{-14} (or 2^{-56}) $\simeq 10^{-17}$. The format with an extended mantissa is commonly called double precision, but in reality the precision is more than doubled. In the IBM case, this is 17 *vs.* 7 decimal digits.



$$e_{\max} = 63; \qquad M_{\max} = 1 - 16^{-14}.$$

The largest representable number is

$$max = 16^{63} * (1 - 16^{-14}).$$

and the smallest positive normalized number is, as before,

$$min = 16^{-64}(16^{-1}).$$

♢

1.5.3 Mapping Errors: Overflows, Underflows, and Gap

Just as in a fixed point system, the finitude of the machine number system is a problem in floating point representation. In practice, the problem of overflow in a floating point system is much less severe than in a fixed point system, and most business-type applications are never aware of it. For example, the budget of the U.S. Government, while in the trillions of dollars, requires only thirteen decimal digits to represent—well within the capability of the IBM/370 floating point format. By contrast, in many scientific applications [35], the computation results in overflows; for example, $e^{200} > 10^{76}$, therefore, e^{200} cannot be represented in the IBM floating point system. Similarly, $(0.1)^{200}$ cannot be represented either, since $(0.1)^{200} = 10^{-200}$, and the smallest representable number is approximately 10^{-76}. The latter situation is called *underflow*. Thus, mapping from the human infinite number system to a floating point system with finite range may result in an unrepresentable exponent (exponent spill). The exponent spill is called overflow if the absolute value of the result is larger than max, and it is called underflow if the absolute value of the result is smaller than min. In order to allow the computation to proceed in a reasonable manner after an exponent spill, the following approximations can be used: underflow is replaced by a normalized zero, and overflow is replaced by the largest signed representable number. However, the CDC 6600 produces a bit pattern representing ∞, and from that point on this overflow is treated as a genuine ∞, for example, $X \div \infty = 0$. These approximations should not be confused with the computations through overflows in fixed point representation. The latter always produce a correct result, whereas the floating point approximation of overflow and underflow always produces an incorrect result; but this incorrect result may have an acceptable error associated with it. For example, in computing a function using polynomial approximation, some of the last terms may underflow, but by setting them to zero, no significance is lost. The case in point [35] is $\sin X \simeq X$, which for $|X| < .25 \times 16^{-32}$ is good to over 65 hexadecimal digits. So far, we have discussed the consequences of mapping out of range numbers, and have shown that the resulting overflow or underflow is a function of the range; that is, the exponent portion of the floating point number. Now consider the consequences of the fact that the floating point number system can represent only a finite subset of R, the set of real numbers. For simplicity, assume that the set R is within the range of floating point numbers; thus, the error in mapping is a function of the mantissa resolution. Garner [17] has shown that for a base β floating point number system with a p-digit mantissa, the value of the gap between normalized floating point numbers is $\beta^{-p}\beta^{e}$, where e is the value of the exponent. The magnitude of the mapping error is some fraction of the gap value. For example, in the range of 0.5 to 0.999..., the IBM short format has a maximum mapping error (gap) of $2^{-24} \times 16^{0} = 2^{-24} \simeq 10^{-7}$, while the long IBM format reduces the mapping error to $2^{-56} \simeq 10^{-17}$.

1.6 Floating Point Operations

In this section, the four basic arithmetic operations are discussed in just enough detail so that the resulting consequences can be analyzed. All operations assume normalized operands:

$$\text{radix}^{-1} \leq \text{ mantissa} < 1.$$

Example 1.10

In the binary system radix = 2 and $0.5 \leq$ mantissa < 1. $\Diamond$

1.6.1 Addition and Subtraction

Addition and subtraction require that the exponents of the two operands be equal. This alignment is accomplished by shifting the mantissa of the smaller operand to the right, while proportionally increasing its exponent until it is equal to the exponent of the larger number. (In general scientific notation, the alignment could be accomplished by the converse operation, that is, shift the mantissa of the larger number left, while decreasing its exponent. However, this is impossible in a normalized floating point system, since a left-shifted normalized mantissa has to be larger than 1, but $1 - \beta^{-p}$ is the largest representable p-digit mantissa.) After the alignment, the two mantissas are added (or subtracted), and the resultant number, with the common exponent, is normalized. The latter operation is called postnormalization. In the addition operation, the postnormalization is a maximum of one right-shifted digit. Since the range (binary radix) of one mantissa is $0.5 \leq |m_1| < 1$, and the other unnormalized mantissa has the range of $0 \leq |m_2| < 1$, the range of the sum $0.5 \leq |m_1 + m_2| < 2$ may require no shift for $0.5 \leq |m_1 + m_2| < 1$, or may require one position right shift for $1 \leq |m_1 + m_2| < 2$. In the latter case, the exponent is increased by 1. If this results in exponent spill, the postnormalization sets the number to its largest possible value. In subtraction, the maximum shift (for a nonzero result) required on postnormalization is equal to the number of mantissa bits minus one. Subtraction may produce the special case of zero result, whereby the postnormalization, instead of shifting, generates a normalized zero.

1.6.2 Multiplication

Multiplication in floating point is conceptually easier than addition. No alignment is necessary. The mantissas are multiplied as if they were fixed point integers and the exponents are simply added. Postnormalization is made up of one of the following cases.

Case 1. Resultant mantissa is in the range of $0.5 \leq |m_1 * m_2| < 1$, in which case no shifting is required.

Case 2. $0.25 \leq |m_1 * m_2| < .5$ requires one place left shift and reducing the resultant exponent by one. (Cases 1 and 2 are justified, since $0.5 \leq |m_1| < 1$, $0.5 \leq |m_2| < 1$, then $0.25 \leq |m_1 * m_2| < 1$.)

Case 3. If either operand is zero, then the postnormalization produces a normalized zero.

Case 4. If either Case 1 or Case 2 (after exponent reduction) generates an exponent spill, then the postnormalization forces the largest absolute value in the case of overflow (smallest, for underflow) with the sign bit set to the proper sign (the EXCLUSIVE-OR of the two operand signs).

1.6.3 Division

To perform floating point division F_1/F_2, the mantissas are divided (m_1/m_2) and the exponent of the divisor is subtracted from the exponent of the dividend. Since $0.5 \leq |m_1| < 1$ and $0.5 \leq |m_2| < 1$, then $0.5 \leq |m_1/m_2| < 2$.

Case 1. If $m_1 < m_2$, then $0.5 < |m_1/m_2| < 1$, and no postnormalization is required.

Case 2. If $m_1 \geq m_2$, then $1 \leq |m_1/m_2| < 2$, and postnormalization is done by shifting the mantissa one place to the right and increasing the exponent by one.

Case 3. If the dividend is zero, postnormalization produces normalized zero.

Case 4. If the divisor is zero, the result overflows and the postnormalization forces the largest representable value.

Case 5. If both dividend and divisor are zero, then the result is undefined. (Later on we will call such a result NAN = Not A Number.)

Case 6. If the postnormalized exponent is out of bounds, the result is overflow or underflow.

1.7 Problems in Floating Point Computations

1.7.1 Loss of Significance

The following example [21] illustrates the loss of significance problem. Assume that two numbers are different by less than 2^{-24}. (The representation is the IBM System 370 short format.)

$$A = 0.1\,0\,0\,0\,0\,0 \times 16^1$$
$$B = 0.FFFFFF \times 16^0.$$

When one is subtracted from the other, the smaller must be shifted right to align the radix points. (Note that the least significant digit of B is now lost.)

$$\begin{aligned} A &= 0.10\,0\,0\,0\,0 \times 16^1 \\ B &= 0.0FFFFF \times 16^1 \\ \hline A - B &= 0.00\,0\,0\,0\,1 \times 16^1 = .1 \times 16^{-4}. \end{aligned}$$

Now let us calculate the error generated due to loss of digits in the smaller number. The result is (assuming infinite precision):

$$\begin{aligned} A &= 0.10\,0\,0\,0\,0\,0 \times 16^1 \\ B &= 0.0FFFFFF \times 16^1 \\ \hline A - B &= 0.00\,0\,0\,0\,0\,1 \times 16^1 = .1 \times 16^{-5}. \end{aligned}$$

$$\text{ERROR} \quad = 0.1 \times 16^{-4} - 0.1 \times 16^{-5} = 0.F \times 16^{-5}.$$

Thus, the loss of significance (error) is $.F \times 16^{-5}$. An obvious solution to this problem is a guard digit, that is, additional bits are used to the right of the mantissa to hold intermediate results. In the IBM format, an additional 4 bits (one hexadecimal digit) are appended to the 24 bits of the mantissa. Thus, with a guard digit the above example will produce no error. On first thought, one might think that in order to obtain maximum accuracy it is necessary to equate the number of guard bits to the number of bits in the mantissa. However, Yohe [48] has proven that two guard digits are always sufficient to preserve maximal accuracy. Regardless of operation (subtraction and multiplication are the operations of concern), only one nonzero bit can be left-postshifted into the result mantissa. Thus, no more than one guard digit will enter the final significant result. However, to insure an unbiased rounding, a third digit (sticky digit) can be added beyond the two guard digits. Rounding and sticky digits will be described in detail in the next section.

The following example illustrates another loss of significance inherent in the floating point number system (S/360 short format).

$$\left.\begin{array}{rcl} A & = & 0.100000 \times 16^{1} \\ B & = & \underline{0.100000 \times 16^{-10}} \end{array}\right\} \text{ Original Operands}$$

$$\left.\begin{array}{rcl} A & = & 0.100000000000 \times 16^{1} \\ B & = & \underline{0.000000000001 \times 16^{1}} \end{array}\right\} \text{ Alignment}$$

$$\begin{array}{rcll} A + B & = & 0.100000000001 \times 16^{1} & \text{Addition} \\ A + B & = & 0.100000 \times 16^{1} & \text{Postnormalization} \end{array}$$

Thus, $A + B = A$, while $B \neq 0$. This violation of a basic law of algebra is characteristic of the approximation used in the floating point system.

1.7.2 Rounding: Mapping the Reals into the Floating Point Numbers

Rounding in floating point arithmetic and associated error analysis are probably the most discussed subjects in floating point literature. References include [48, 24]; for a more complete list see Garner [17].

The following formal definition of rounding is taken from Yohe [48]:

> *Let R be the real number system and let M be the set of machine representable numbers. A mapping Round: $R \rightarrow M$ is said to be* rounding *if, for all $a \in R$, $b \in R$, Round$(a) \in M$, and Round$(b) \in M$ we have:*
>
> $$\text{Round}(a) \leq \text{Round}(b) \text{ whenever } a \leq b.$$
>
> *Further: A rounding is called optimal if for all $a \in M$, Round $(a) = a$.*

"Optimal" implies that if $a \in R$ and m_1, m_2 are consecutive members of M with $m_1 < a < m_2$, then Round$(a) = m_1$ or Round$(a) = m_2$. Rounding is symmetric if Round $(a) = -$Round $(-a)$. For example, Round $(39.2) = -$Round $(-39.2) = 39$.

Yohe defines a total of five rounding modes for all $a \in R$, including three optimal symmetric roundings: T, A, and RND.

1. Downward directed rounding: $\nabla a \leq a$.
2. Upward directed rounding: $\Delta a \geq a$.
3. Truncation (T), rounding toward zero.
4. Augmentation (A), rounding away from zero.
5. Rounding (RND), which selects the closest machine number, and in the case of a tie selects the number whose magnitude is larger. This rounding is the most frequently used, since it produces maximum accuracy. Since round to larger (RN) produces a consistent bias in the result, a variation in using round to even has been adopted by the IEEE Standard (discussed later). Accuracy is unaffected.

The directed rounding, while not provided outside of the IEEE format (discussed later), is very important in interval arithmetic. Interval arithmetic is a procedure for computing an upper and lower bound on the true value of a computation. These bounds define an interval which contains the true result. Yohe claims that hardware designed to produce these roundings would enable interval operations to be executed in one tenth to one fifth of the time normally required to execute them with simulated floating point arithmetic.

The preceding five rounding methods are illustrated in the following example for decimal arithmetic.

	∇	Δ	T	A	RND
+39.7	+39	+40	+39	+40	+40
+39.5	+39	+40	+39	+40	+40
+39.2	+39	+40	+39	+40	+39
+39.0	+39	+39	+39	+39	+39
−39.0	−39	−39	−39	−39	−39
−39.2	−40	−39	−39	−40	−39
−39.5	−40	−39	−39	−40	−40
−39.7	−40	−39	−39	−40	−40

In order to handle overflow and underflow, Yohe describes hardware architecture that includes an indicator digit representing one of four conditions:

1. o = overflow indicator.
2. u = underflow indicator.
3. i = infinity indicator.

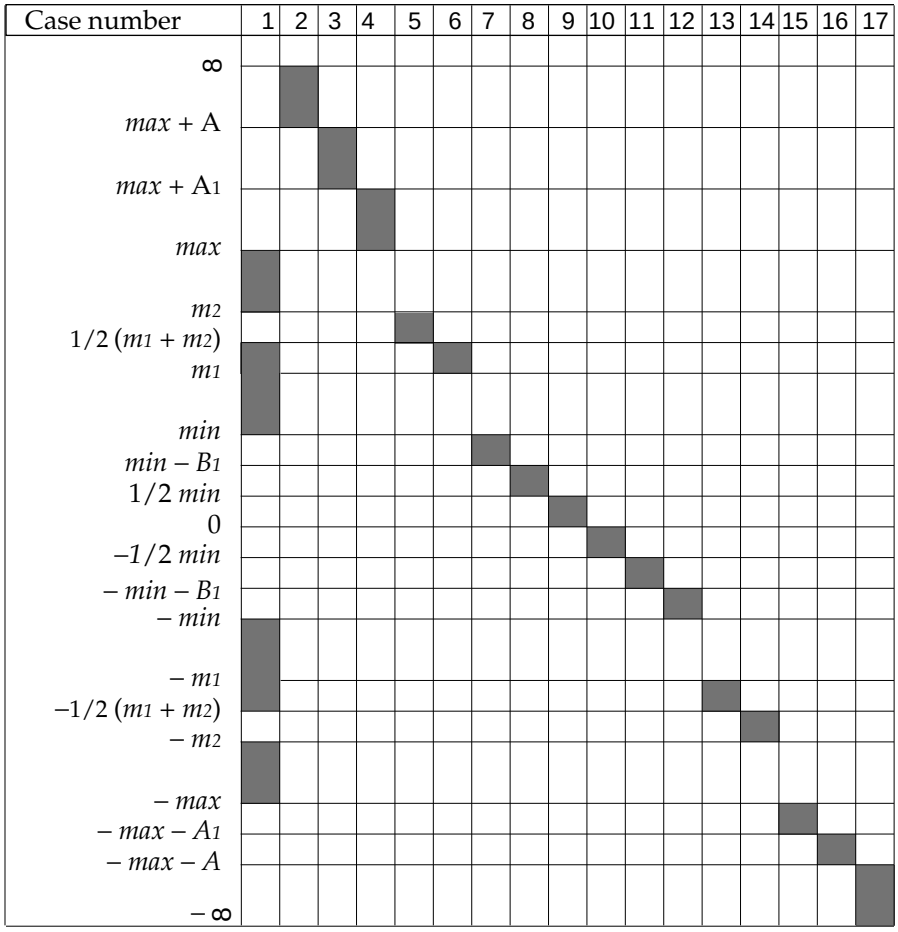


Figure 1.1: Graphical illustration of the 17 different cases of the range of the reals.

4. Representable number (none of the above).

The largest positive floating point number is denoted by *max*, and the smallest normalized positive floating point number by *min*. The effects of the five rounding methods are summarized in Table 1.1, while Figure 1.1 graphically partitions the infinite space of the reals into several ranges. All together, seventeen cases are tabulated. Case 1 gives the same result for all five rounding methods, since it is an exact normalized machine number. In this table, m_1 and m_2 represent consecutive positive normalized floating point numbers with $m_1 < m_2$. Cases 5, 6, 13, and 14 show the rounding effect for $m_1 \leq R \leq m_2$. All the remaining cases overflow or involve underflow. For example, in the overflow cases, the rounding algorithm will set the exponent overflow indicator. If the rounding option implies rounding toward zero, the result will be $\pm$ *max*. If the rounding option implies rounding away from zero, the infinity indicator is set and the result is replaced by the particular bit configuration used to represent infinity.

Case	Range of Values	R	Δ Value	Δ Ind	∇ Value	∇ Ind	Rounding Option T Value	T Ind	A Value	A Ind	RND Value	RND Ind	Case
1		$R = m$	m		m		m		m		m		1
2	$max + \beta^{e_{max}-p}$	$\leq R$	∞	o,i	max	o	max	o	∞	o,i	∞	o,i	2
3	$max + \frac{1}{2}\beta^{e_{max}-(p+1)}$	$\leq R < max + \beta^{e_{max}-p}$	∞	o,i	max		max		∞	o,i	∞	o,i	3
4	max	$< R < max + \frac{1}{2}\beta^{e_{max}-(p+1)}$	∞	o,i	max		max		∞	o,i	max		4
5	$\frac{1}{2}(m_1 + m_2)$	$\leq R < m_2$	m_2		m_1		m_1		m_2		m_2		5
6	m_1	$< R < \frac{1}{2}(m + m_2)$	m_2		m_1		m_1		m_2		m_1		6
7	$min - \frac{1}{2}\beta^{e_{min}-(p+1)}$	$\leq R < min$	min		0	u	0	u	min		min		7
8	$\frac{1}{2}\ min$	$\leq R < min - \frac{1}{2}\beta^{e_{min}-(p+1)}$	min	u	0	u	0	u	min	u	min	u	8
9	0	$< R < \frac{1}{2}\ min$	min	u	0	u	0	u	min	u	0	u	9
10	$-\frac{1}{2}\ min$	$< R < 0$	0	u	$-min$	u	0	u	$-min$	u	0	u	10
11	$-\ min + \frac{1}{2}\beta^{e_{min}-(p+1)}$	$< R \leq -\frac{1}{2}min$	0	u	$-min$	u	0	u	$-min$	u	$-min$	u	11
12	$-min$	$< R \leq -min + \frac{1}{2}\beta^{e_{min}-(p+1)}$	0	u	$-min$		0	u	$-min$		$-min$		12
13	$-\frac{1}{2}(m_1 + m_2)$	$< R < -m_1$	$-m_1$		$-m_2$		$-m_1$		$-m_2$		$-m_1$		13
14	$-m_2$	$< R \leq -\frac{1}{2}(m_1 + m_2)$	$-m_1$		$-m_2$		$-m_1$		$-m_2$		$-m_2$		14
15	$-max - \frac{1}{2}\beta^{e_{max}-(p+1)}$	$< R < -\ max$	$-\ max$		$-\infty$	o,i	$-max$		$-\infty$	o,i	$-max$		15
16	$-\ max - \beta^{e_{max}-p}$	$< R \leq -max - \frac{1}{2}\beta^{e_{max}-(p+1)}$	$-\ max$		$-\infty$	o,i	$-max$		$-\infty$	o,i	$-\infty$	o,i	16
17		$R \leq -max - \beta^{e_{max}-p}$	$-\ max$	o	$-\infty$	o,i	$-max$	o	$-\infty$	o,i	$-\infty$	o,i	17

Table 1.1: The effects of rounding, from Yohe [48]. i represents the infinity indicator, o the overflow indicator, and u the underflow indicator. The left column, which describes the range of the real numbers, is illustrated graphically in Figure 1.1 .

The following shorthand notation is used in Figure 1.1.

$$\begin{aligned} A &= \beta^{e_{\max}-p} \\ A_1 &= \frac{1}{2}\beta^{e_{\max}-(p)} \\ B_1 &= \frac{1}{2}\beta^{e_{\min}-p} \end{aligned}$$

where:

$$\begin{aligned} \beta &= \text{radix}, \\ e_{\max} &= \text{maximum possible exponent}, \\ e_{\min} &= \text{minimum possible exponent}, \\ p &= \text{number of } \beta\text{-base mantissa digits}. \end{aligned}$$

Note that A corresponds to $\beta^{e_{\max}} \cdot \beta^{-p}$, a number with maximum exponent and least possible (unnormalized) mantissa. A_1 and B_1 represent quantities with mantissas out of range of the indicated exponent.

1.7.3 Radix Tradeoffs and Error Analysis

So far, the range and significance were discussed independently, but for a given number of bits there is a tradeoff between them. Recall the previously mentioned 32-bit format with 24 bits of unsigned mantissa, 7 bits of exponent, and one sign bit. The tradeoffs between range and significance are illustrated by comparing a system with a radix of 16 (hexadecimal) against a system with a radix of 2 (binary system).

	Largest Number	Smallest Number	Precision
Hexadecimal system	7.2×10^{75}	5.4×10^{-79}	16^{-6}
Binary system	9.2×10^{18}	2.7×10^{-20}	2^{-24}

While the hexadecimal system has the same precision as binary, hex-normalization may result in three leading zeros, whereas nonzero binary normalization never has leading zeros. Accuracy is the guaranteed or minimum number of significant mantissa bits (excluding leading zero). This table indicates that for a given word length, there is a tradeoff between range and accuracy; more accuracy (base 2) is associated with less range, and vice versa (base 16). There is quite a bit of sacrifice in range for a little accuracy. In base 2 systems there is also a property that can be used to increase the precision, a normalized number must start by 1. In such a case, there is no need to store this 1 with the rest of the bits. Rather, the number is stored starting from the following bit location and that 1 is assumed to be there by the hardware or software manipulating the numbers. This 1 is called the hidden one. Cody [6] tabulates the error as a function of the radix for three 32-bit floating point formats having essentially identical range.

Base	Exponent Bits	Range	Mantissa Bits	Maximum Relative Error	Average Relative Error
2	9	$2^{255} \simeq 6 \times 10^{76}$	22	0.5×2^{-21}	0.18×2^{-21}
4	8	$4^{127} \simeq 3 \times 10^{76}$	23	0.5×2^{-21}	0.14×2^{-21}
16	7	$16^{63} \simeq 0.7 \times 10^{76}$	24	2^{-21}	0.17×2^{-21}

The base 2 entries in the table are without use of the hidden one.

The relative error of $X \in R$ in the right columns is defined as the magnitude of the representable error divided by X (with exponents x). The maximum relative representation error (MRRE) over all normalized fractions is computed following (obviously, the maximum representable error is $\frac{1}{2}$ of the gap):

$$\text{MRRE} = \frac{\text{maximum representable error}}{\text{smallest normalized fraction}} = \frac{\frac{1}{2} * \beta^{-p} * \beta^{x}}{\frac{1}{\beta} * \beta^{x}} = \frac{1}{2}\beta^{-p}\beta.$$

For example, in the hexadecimal system

$$\text{MRRE} = \frac{1}{2} \times 2^{-24} \times 16 = 2^{-25} \times 2^{4} = 2^{-21}.$$

The average relative representation error (ARRE) assumes a logarithmic distribution for floating point number and a uniformly distributed minimum representation error.

From the preceding table, the binary system seems better in range and accuracy than the hexadecimal system. So why use hexadecimal radix at all? The answer is in the higher computational speed associated with larger base value, as illustrated by the following example.

Example 1.11

Assume a 24-bit mantissa with all bits zero except the least significant bit. Now, compare the maximum number of shifts required for each case of postnormalization.

Case 1: Binary system: radix = 2. In this case, 23 shifts are required.

Case 2: Hexadecimal system: radix = 16. In this case we shift four bits at a time (since each hexadecimal digit is made of 4 bits). Therefore, the maximum number of shifts is five.

Thus, the hexadecimal shifting is several times faster than the binary. ♢

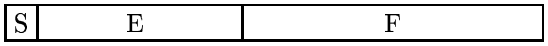
Garner [17] summarizes the tradeoffs:

> "...the designer of a floating number system must make decisions affecting both computational speed and accuracy. Better accuracy is obtained with small base values and sophisticated round-off algorithms, while computational speed is associated with larger base values and crude round-off procedures such as truncation."

1.8 A Standard Floating Point Representation

1.8.1 Background

Presently there are more than 20 different floating point formats in use by various computer manufacturers. We illustrate below the formats of three computers that were popular for scientific computing, using the following general terminology: *Exponent* will represent all forms of biased and unbiased exponents, while *significand* represents the mantissa independent of the location of the radix point.



S = Sign bit (indicates the sign of the significand).
E = Biased Exponent.
F = Significand.

Strictly speaking, only mantissas of the form 0.xxx ... are fractions. When discussing both fraction and other mantissa forms (e.g. 1.xxx ...), we use the more general term *significand*.

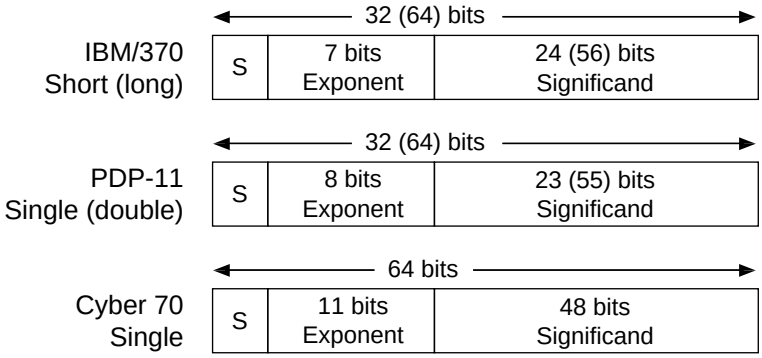


Table 1.2 shows further details of the three formats; from it we can see that there is hardly any similarity between the various formats. This situation, which prohibits data portability produced by numerical software, was the main motivation in 1978 for setting up an IEEE (Computer Society) committee to standardize floating point arithmetic. The main goal of the standardization efforts is to establish a standard which will allow communication between systems at the data level without the need for conversion.

In addition to the respectable goal of "the same format for all computers," the committee wanted to ensure that it would be the best possible standard for a given number of bits. Specifically, the concern was to ensure correct results, that is, the same as those given by the corresponding infinite precision with an error of 1/2 of the LSB. Furthermore, to ensure portability of all numerical data, the committee specified exceptional conditions and what to do in each case (overflow, underflow, etc.). Finally, it was desirable to make possible future extensions of the standard such as interval arithmetic.

	IBM S/370	DEC PDP-11	CDC Cyber 70
	S = Short L = Long	S = Short L =Long	
Word length	S: 32 bits L: 64 bits	S: 32 bits L: 64 bits	60 bits
Exponent	7 bits	8 bits	11 bits
Significand	S: 6 digits L: 14 digits	S: (1)+23 bits L: (1)+55 bits	48 bits
Bias of exponent	64	128	1024
Radix	16	2	2
Hidden '1'	No	Yes	No
Radix point	Left of Fraction	Left of hidden '1'	Right of MSB of Fraction
Range of Fraction (F)	$(1/16) \leq F < 1$	$0.5 \leq F < 1$	$1 \leq F < 2$
F representation	Signed magnitude	Signed magnitude	One's complement
Approximate max. positive number*	$16^{63} \simeq 10^{76}$	$2^{126} \simeq 10^{38}$	$2^{1023} \simeq 10^{307}$
Precision	S: $16^{-6} \simeq 10^{-7}$ L: $16^{-14} \simeq 10^{-17}$	S: $2^{-24} \simeq 10^{-7}$ L: $2^{-56} \simeq 10^{-17}$	$2^{-48} \simeq 10^{-14}$

*Approximate maximum positive number for the DEC PDP-11 is 2^{126}, as 127 is a reserved exponent.

Table 1.2: Comparison of floating point specification for three popular computers.

The motivation of this section is quite independent of the adoption of this standard. Rather, one can view the standard as an "ideal" numeric representation largely derived without implementation or compatibility constraints. It then serves (at the least) as the basis for comparison with actual floating point formats.

Before we describe the details of the standard, let us analyze the good and bad points of each of the above three popular formats.

Representation Error: According to Ginsberg [19], the combination of base 16, short mantissa size, and truncated arithmetic should definitely be avoided. This criticism is of the IBM short format where, due to the hexadecimal base, the MRRE is 2^{-21}. By contrast, the 23 bits combined with the hidden '1' (as on PDP-11) seems to be a more sensible tradeoff between range and precision in a 32-bit word, with MRRE of 2^{-24}.

Range: While the PDP-11 scores well on its precision on the single format, it loses on its double format. In a 64-bit word, the tradeoff between range and precision is unbalanced and more exponent range should be given at the expense of precision. The exponent range of the CYBER 70 seems to be more appropriate for the majority of scientific applications.

Rounding: None of the three formats uses an *unbiased* rounding to the nearest machine number in case of ties [19].

Implementation: The advantage of a radix 16 format (as in IBM S/370) over a radix 2 format is the relative ease of implementation of addition and subtraction. Radix 16 simplifies the shifter hardware, since shifts can only be made on 4-bit boundaries, while radix 2 formats must accommodate shifts to any bit position.

	Single	Double
Word length	32 bits	64 bits
Sign	1-bit	1-bit
Biased exponent	8 bits	11 bits
Significand	(1) + 23 bits	(1) + 52 bits
Bias of Exponent	127	1023
Ranges:		
(a) Approximate max. positive number*	$2^{128} \simeq 3.4 * 10^{38}$	$2^{1024} \simeq 1.8 * 10^{308}$
(b) Minimum positive normalized number	$2^{-126} \simeq 1.2 * 10^{-38}$	$2^{-1022} \simeq 2.2 * 10^{-308}$
Precision	$2^{-23} \simeq 10^{-7}$	$2^{-52} \simeq 10^{-16}$

*Actual maximum positive number is slightly smaller. For example, in the single precision it is: $2^{127}(2 - 2^{-23})$

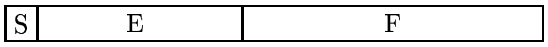
Table 1.3: The two levels of precision of the floating point data format.

1.8.2 The IEEE Standard (754) Floating Point Representation

The IEEE Computer Society received several proposals for standard representations for consideration; however, the most complete was the one prepared by Kahan, Coonen, Stone, and Palmer [8]. This proposal became the basis for the IEEE standard (754) floating point representation.

We first describe the standard, and then point out some controversial issues, with the pros and cons for each such issue.

Basic Formats: The floating point data format is made up of three parts (from left to right): sign bit, biased exponent (characteristic), and significand (mantissa):



where

$$\begin{aligned} & S &=& \text{ Sign bit (indicates the sign of the significand),} \\ & E &=& \text{ Biased exponent,} \\ & F &=& \text{ Significand;} \\ \text{then} \quad & e &=& \text{ true exponent } = \text{ E } - \text{bias,} \\ & f &=& \text{ true significand } = \text{ 1.F.} \end{aligned}$$

A normalized nonzero number, X, has the following interpretation.

$$\boxed{X = (-1)^{\mathrm{S}} * 2^{\mathrm{E-bias}} * (1.F)}$$

This format has two levels of precision as outlined in Table 1.3: SINGLE and DOUBLE.

Note: The first version of the proposal was drafted in April 1978 by Harold Stone and the final version was published in the March 1981 *Computer*. The original draft included a QUAD format, while the final one omitted it.

For SINGLE precision of the 32 bits, 1 is used for the sign, 8 for the biased exponent, and 23 for the significand. The significand (F) is represented by a sign-magnitude notation, with an

Table 1.4: Exponent indicators.

Biased Exponent	True Exponent	
	IEEE	DEC PDP-11
E	Bias = 127	Bias = 128
0	Reserved Operand	Reserved Operand
1	-126	-127
2	-125	-126
$\vdots$	$\vdots$	$\vdots$
127	0	-1
128	1	0
129	2	1
$\vdots$	$\vdots$	$\vdots$
254	127	126
255	Reserved Operand	Reserved Operand

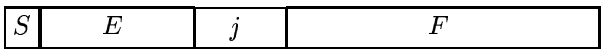
implied leading one (hidden '1') and an implied binary point to the right of the hidden '1'. By now, it should be clear that the format (at least on SINGLE precision) is very similar to that of the PDP-11 and the VAX from Digital Equipment Corporation (DEC), but it is not identical. For example, the IEEE true significand is in the range $1 \leq f < 2$, whereas the DEC significand is $0.5 \leq f < 1$. Other differences will be pointed out later on. The biased exponent is an 8-bit number of the range $0 < E < 255$ (the values of 0 and 255 are for reserved operands), and it is biased by the value 127. Thus, the true exponent has the representations $-127 < e < 128$, but -127 and 128 are reserved. Thus, the range is $-126 < e < 127$, as shown in Table 1.4.

The different decoding of the reserved operands are compared in Table 1.5.

1.8.3 Extended Formats

The standard does not require, but recommends, the use of extended formats for temporary results. This mechanism reduces the chance of an intermediate overflow of a computation whose result would have been representable in a basic format. It also reduces the round off error introduced during a long chain of operations.

Each of the two basic formats has associated with it an extended format; thus, we have single-extended and double-extended formats as follows (note that the hidden bit is not used here, since it has sufficient significand bits):



where j is the leading bit of the significand that was hidden in the basic format.

Table 1.5: IEEE and DEC decoding of the reserved operands (illustrated with the SINGLE format, i.e., $E_{\max} = 255$).

S	Biased Exp	Significand	Interpretation	
0	0	0	+Zero	
1	0	0	−Zero	
0/1	0	Not 0	±Denormalized Numbers	IEEE
0	255	0	+Infinity	
1	255	0	−Infinity	
X	255	Not 0	NAN (Not a Number)	
S	Biased Exp	Significand	Interpretation	
0	0	Don't care	Unsigned zero	DEC
1	0	Don't care	General purpose reserved operands	

	Single-extended	Double-extended
Word length ≥	44 bits	80 bits
S = Sign	1-bit	1-bit
E = Exponent ≥	11 bits	15 bits
F = Significand ≥	31 bits	63 bits

Arithmetic Operations. The IEEE standard specifies the following operations:

1. Numerical operations
   - Add
   - Subtract
   - Multiply
   - Divide
   - Square Root
   - Remainder

2. Conversion operations
   - Floating point ↔ Integer.
   - Binary (integer) ↔ Decimal (integer).
   - Binary (floating) ↔ Decimal (floating).

3. Comparisons

   Four operations allowed (<, =, >, unordered). Unordered occurs when
   - One operand is a NAN.

4. Miscellaneous operations
   - Move from one format width to another.
   - Compare and set condition code (option).
   - Find integer part.

Rounding. There are four rounding modes:

1. RN = Unbiased rounding to nearest (in case of a tie round to even).
2. RZ = Round toward zero (chop, truncate).
3. RM = Round toward minus infinity.
4. RP = Round toward plus infinity.

Unbiased rounding is very similar to the conventional round to nearest which is implemented by adding 1/2 of the digit to be discarded and then truncate to the desired precision. For round to integer, we might have:

Example 1.12

$$\begin{array}{r} 39.2 \\ 0.5 \\ \hline 39.7 \rightarrow 39 \end{array} \qquad \begin{array}{r} 39.7 \\ 0.5 \\ \hline 40.2 \rightarrow 40 \end{array}$$

But suppose the number to be rounded is exactly halfway between two numbers: which one is the nearest? To answer the question, let us add the same 0.5 to the two following numbers:

$$\begin{array}{r} 38.5 \\ 0.5 \\ \hline 39.0 \rightarrow 39 \end{array} \qquad \begin{array}{r} 39.5 \\ 0.5 \\ \hline 40.0 \rightarrow 40 \end{array}$$

$\Diamond$

Notice that we rounded up in both cases, even though each number was exactly halfway between smaller and larger numbers. Therefore, by simply adding 0.5 and truncating, a biased rounding is generated. In order to have unbiased rounding, we round to even whenever there is a tie between two numbers. Now, using the previous numbers we get:

$$\begin{aligned} 38.5 &\rightarrow 38 \\ 39.5 &\rightarrow 40 \end{aligned}$$

In the first case the number is rounded down, and in the second case the number is rounded up. Therefore, we have statistically unbiased rounding. Of course, the same unbiased rounding could be obtained by rounding to odd (instead of even) in the tie case. This time, the rounding looks like this:

$$\begin{aligned} 38.5 &\rightarrow 39 \\ 39.5 &\rightarrow 39 \end{aligned}$$

However, rounding to even is preferred because it may result in "nice" integer numbers, as in the following examples when rounding to the first fractional position:

$$1.95 \rightarrow 2$$
$$2.05 \rightarrow 2,$$

whereas rounding to odd results in the more frequent occurrence of noninteger numbers:

$$1.95 \rightarrow 1.9$$
$$2.05 \rightarrow 2.1.$$

Now we illustrate the implementation of the unbiased rounding to even, and introduce the so-called "sticky bit".

The conventional system for rounding is to add 1/2 of the LSD position of the desired precision to the MSD of the portion to be discarded. But this scheme has a problem, as is illustrated below (the XXXX are additional bits). Thus,

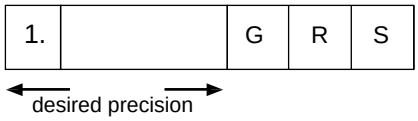
38.5 X X X X X X X	← Number to be rounded
0.5 0 0 0 0 0 0 0	← Add 0.5
39.0 X X X X X X X	← Result
39	← Truncate

Two cases have to be distinguished:

Case 1: $X\,X\,X\,X\,X\,X \neq 0$ (at least one $X = 1$), and the rounding is correct since 39 is nearest $38.5 + \Delta$, where $0 < \Delta < 0.5$.

Case 2: $X\,X\,X\,X\,X\,X = 0$ (all bits are $X = 0$). Now the rounding is incorrect because we have a tie case that requires the result to be rounded to even (38).

It is obvious that, regardless of the number of X bits, all possible permutations can be mapped into one of the two preceding cases. Therefore, one bit can be used to distinguish between Case 1 and Case 2. This bit is called the "sticky bit", and it has the value one for Case 1 and the value zero for Case 2. The logic implementation of the sticky bit is simply ORing of the bits to the right of the second guard bit (or *round* bit), as illustrated following for the addition/subtraction operation.



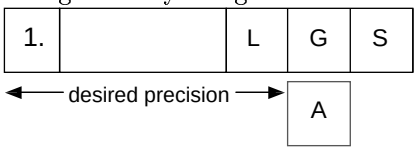
where

L = LSB of the significand before rounding,
G = first guard bit,
R = second guard bit,
S = sticky bit.

In the case of a left shift (normalization after subtraction), S does not participate in the left shift, but instead zeros are shifted into R. In the case of a right shift due to a significand overflow

(during magnitude addition or no shift), the S and R guard bits are ORed into S (i.e., $L \rightarrow G$ and $G + R + S \rightarrow S$).

The preceding format (with two guard bits, G and R) is necessary during the postnormalization; the final result just before rounding has only one guard bit and the sticky bit.



where

L = LSB of the significand before rounding,
G = guard bit,
S = sticky bit,
A = bit to be added to G for proper rounding.

The proper action to obtain unbiased rounding-to-even (RN) is determined from the following table:

L	G	S	Action	A
X	0	0	Exact result. No rounding is necessary.	0
X	0	1	Inexact result, but significand is rounded properly.	0
0	1	0	The tie case with even significand. No rounding needed.	0
1	1	0	The tie case with odd significand. Round to nearest even.	1
X	1	1	Round to nearest by adding 1 to the L-bit.	1

Example 1.13

Number line for Round 4-bit significands

		G	$S \rightarrow$ *Action*
a)	1.000X	0	0 $\rightarrow$ machine number
b)	1.000X	0	1 $\rightarrow$ closer to .000X
c)	1.0000	1	0 $\rightarrow$ tie with LSB even
d)	1.0001	1	0 $\rightarrow$ tie with LSB odd; becomes 1.0010
e)	1.000X	1	1 $\rightarrow$ round up

$\Diamond$

So far, we have addressed only the unbiased rounding; but there are three more optional modes. The round to zero (RZ) is simply a truncation that is used in certain integer related operations (actually, most present-day computers provide truncation as the only rounding option). The remaining two rounding modes are rounding toward $+\infty$ and rounding toward $-\infty$. These two directed roundings are used in interval arithmetic where one computes the upper and lower bounds of an interval by executing the same sequence of instructions twice, rounding up during one pass and down during the next. The sticky bit, introduced previously, is also essential for the correct direct rounding as illustrated following for rounding upward.

Example 1.14 Directed Upward Rounding

Case 1: No sticky bit is used; e.g., for round to integer we would have:

$38.00001 \rightarrow 38$

$38.00000 \rightarrow 38.$

Case 2: Sticky bit is used:

$38.00001 \rightarrow 39$ (sticky bit = 1)

$38.00000 \rightarrow 38$ (sticky bit = 0, exact number).

$\Diamond$

1.8.4 Exceptions and What to Do in Each Case

The IEEE standard specifies five exceptional conditions that may arise during an arithmetic operation:

1. INVALID OPERATION and NANs,
2. OVERFLOW,
3. DIVISION BY ZERO,
4. UNDERFLOW,
5. INEXACT RESULT.

Exceptions are handled in one of two ways:

1. TRAP and supply the necessary information to correct the fault. For example:

   What instruction caused the TRAP?
   What were the values of the input operands?
   Etc.

2. DISABLE TRAP, but deliver a specified result. For example on UNDERFLOW: "Set result to zero and continue".

In either case, there is a corresponding exception—the flag is set and it remains set until cleared by the user.

Exception 1: INVALID OPERATION and NANs

The INVALID OPERATION exception occurs during a variety of arithmetic operations that do not produce valid numerical results.

Example 1.15

$\sqrt{-5}$

$(+\infty) + (-\infty)$

$0 * \infty$

$0/0$

∞/∞

$\infty \bmod X$

Conversion errors

Compare of unordered numbers $\Diamond$

Operations like the above generate an entity called NAN (Not a Number), which is one class of the reserved operands (Table 1.5a). This class is characterized by a biased exponent of $111\ldots111$ and a nonzero significand. Therefore, there are (in the SINGLE precision format) $2^{23} \simeq 8$ million members in the NAN class. These NANs can be used (if the trap is enabled) to communicate information to the user program. For example, the result of any of the preceding operations may be a pointer to the offending line of code. The standard does not specify the exact information to be contained in the NANs, since this type of information is highly system dependent.

Since the NAN is a valid result of an arithmetic operation, it is necessary to specify exactly what to do if a NAN appears as an input operand. Generally, the NAN simply propagates through the arithmetic operation.

Example 1.16

$$\begin{aligned} 5 + \text{NAN} &\rightarrow \text{NAN} \\ 3 * \text{NAN} &\rightarrow \text{NAN} \\ \sqrt{\text{NAN}} &\rightarrow \text{NAN} \quad \Diamond \end{aligned}$$

In case both of the input operands are NANs, the result is the NAN with the smaller significand. However, the INVALID OPERATION FLAG is set on the creation of any NAN operand.

Exception 2: OVERFLOW and infinities

The OVERFLOW FLAG is set whenever the exponent of the result exceeds the allowed range for the corresponding precision. For example, in SINGLE precision, OVERFLOW occurs if the unbiased exponent exceeds 127.

If the OVERFLOW trap is enabled when an overflow occurs, a value is delivered to the trap handler that allows the handler to determine the correct result. This value is identical to the normal floating point representation of the result, except that the biased exponent is adjusted by subtracting 192 for single and 1536 for double. This bias adjust has the effect of wrapping the exponent back into the middle of the allowable range. For example, suppose we multiply two large numbers to produce a single precision result:

$$2^{127} * 2^{127} = 2^{254} \leftarrow \text{overflow.}$$

The value delivered to the trap handler would have a biased exponent:

$$254 + 127 - 192 = 189.$$

If the OVERFLOW trap is disabled when an overflow occurs, infinity with the sign of the overflowed result is delivered as the final result. The infinities are represented by floating point numbers with the maximum allowable biased exponent and a zero significand.

The infinities can be valid in many situations. For example,

$$+\infty + \text{ Real Number } = +\infty.$$

$$-\infty + \text{ Real Number } = -\infty.$$

$$\sqrt{+\infty} = \infty.$$

$$\text{Positive Real Number} \div -\infty = -0.$$

Except for the invalid operations described below, operations upon $\pm$ infinity are considered to be *exact* and raise no exceptions. However, a number of uses for infinity raise the INVALID OPERATION exception: $+\infty + -\infty$, $0 \times \infty$, and ∞/∞.

Exception 3: DIVISION BY ZERO

The division by zero is a special case of the OVERFLOW. It happens in a division operation when the divisor is zero and the dividend is a nonzero number (including infinity). If the trap is disabled, the delivered result is a signed infinity.

Exception 4: UNDERFLOW and DENORMALIZED numbers

The UNDERFLOW exception occurs whenever the biased exponent becomes zero or negative, that is, $E < 1$. If the UNDERFLOW trap is enabled, then the exponent is wrapped around into the desired range with a bias adjust identical to the technique used in the OVERFLOW case, except that the bias adjust is added instead of subtracted from the bias exponent.

If the UNDERFLOW trap is disabled, then the number is denormalized by right shifting the significand and correspondingly incrementing the exponent until it reaches the minimum allowed exponent ($e = -126$). At this point, the hidden '1' is made explicit and $E = 0$. The following example [8] illustrates the denormalizing process.

Example 1.17

Assume (for simplicity) that we have a SINGLE precision exponent and a significand of 6 bits.

Actual result	$=2^{-130} * 1.01101.$	$\vert..$	$-130 < -126$ so we denormalize
can be represented as	$=2^{-126} * 0.000101$	$\vert 101\ldots$	we round (to nearest)
and rounded	$=2^{-126} * 0.000110$	$\vert$	= the result to be delivered.

$\Diamond$

The denormalization as a result of UNDERFLOW has been called GRADUAL UNDERFLOW or GRACEFUL UNDERFLOW. Of course, this approach merely postpones the fatal underflow

which occurs when all the nonzero bits have been right shifted out of the significand. Note that since denormalized numbers and ± zero have the same exponent ($E = 0$), such a fatal underflow would automatically produce the properly signed zero. Use of a signed zero indicator is an interesting example of taking a potential disadvantage—two representations for the same value—and turning it (carefully!) into an advantage.

When a denormalized number is an input operand, it is treated the same as a normalized number if the operation is ADD/SUBTRACT. If the result can be expressed as a normalized number, then the loss of significance in the denormalized operand did not affect the precision of the operation and computation proceeds normally. Otherwise, the result will also be denormalized.

If an operation uses denormalized input operand and produces a normalized result, usually a *denormalization loss* occurs. As an example, suppose $0.0010\ldots * 2^{-126}$ were multiplied by $1.000\ldots * 2^{9}$. The result, $1.000\ldots * 2^{-120}$, can be expressed as a normalized number, but it has three fewer bits of precision than implied. The standard suggests that "extraordinary" loss of accuracy be detected as an underflow, but leaves details to the implementor.

Finally, note that operations on denormalized operands can produce normalized results with or without exceptions noted to the programmer. Some examples are:

	$2^{-126} \times 0.1000000$	denormalized number
$+$	$2^{-126} \times 0.1000000$	denormalized number
	$2^{-126} \times 1.0000000$	normalized number, no exception
	$2^{-126} \times 0.1110000$	denormalized number
$\times$	$2^{1} \times 1.1110000$	normalized number
	$2^{-125} \times 1.1010010$	normalized number, no exception
	$2^{-126} \times 0.1000000$	denormalized number
	move to double precision result	
	$2^{-126} \times 0.1000000$	denormalized number, EXCEPTION (implementation dependent)

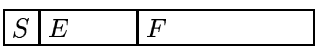
Exception 5: INEXACT RESULT

Exact result is obtained whenever both the guard bit and the sticky bit are each equal to zero. Any other combinations of the guard and sticky bit implies that a round off error has taken place, in which case the INEXACT RESULT FLAG is set. The purpose of this flag is to allow integer calculations with a fast floating point execution unit. The multiplication or addition of integers can be performed with the most significant bits of the floating point result *assumed* to be an integer. The INEXACT RESULT FLAG will cause an interrupt whenever the actual result extends outside the allocated floating point precision.

1.8.5 Analysis of the Standard

There seems to be general agreement that the following features of the proposed standard are best for the given number of bits.

- The format of:



- The two levels of precision (SINGLE and DOUBLE).
- The various rounding modes.
- The specification of arithmetic operations.
- Condition causing exceptions.

However, on a more detailed level, there seem to be many controversial issues, which we outline next.

GRADUAL UNDERFLOW. This is the area where the most controversy exists. The obvious advantage of the GRADUAL UNDERFLOW is the extension of the range for small numbers, and similarly, the compression of the gap between the smallest representable number and zero. For example, in SINGLE precision the gap is $2^{-126} \simeq 1.2 * 10^{-38}$ for normalized numbers, whereas the use of denormalized numbers narrows the gap to $2^{-149} \simeq 1.4 * 10^{-45}$. However, the argument is that GRADUAL UNDERFLOW is needed not so much to extend the exponent range as to allow further computation with some sacrifice of precision in order to defer as long as possible the need to decide whether the UNDERFLOW will have significant consequences.

There are several objections to this GRADUAL UNDERFLOW:

1. Payne [29] maintains that the range is extended only from 10^{-38} to 10^{-45} (coupled with complete loss of precision at 10^{-45}) and it makes sense only if SINGLE precision frequently generates intermediate results in the range 10^{-38} to 10^{-45}. However, for such cases, she believes that the use of SINGLE precision (for intermediate results) is generally inappropriate.

2. Fraley [13] objects to the use of GRADUAL UNDERFLOW for three reasons:

   (a) There are nonuniformities in the treatment of GRADUAL UNDERFLOW;

   (b) There is no sufficient documented need for it;

   (c) There is no mechanism for the confinement of these values.

3. Another objection to the GRADUAL UNDERFLOW is the increased implementation cost in floating point hardware. It is much more economical and faster to simply generate a zero output on underflow, and not have to recognize a denormalized number as a valid input.

An alternative approach to denormalized numbers is the use of a pointer to a heap on occurrence of underflow [29]. In this scheme, a temporary extension of range can be implemented on occurrence of either underflow or overflow without sacrifice of precision. Furthermore, multiplication (and division) work as well as addition and subtraction. While this scheme seems adequate, or even better than GRADUAL UNDERFLOW, it also has the same cost disadvantage outlined in number (3) above.

Significand range and exponent bias. The standard has a significand $1 \leq F < 2$, and the exponent is biased by 127 (in the SINGLE precision). These yield a number system with a magnitude between 2^{-126} and $\approx 2^{128}$, thus, the system is asymmetric in such a way that overflow is presumably less likely to happen than underflow. However, if GRADUAL UNDERFLOW is not used, then the above rationale disappears and one can go back to a PDP-11 format with significand of $0.5 \leq F < 1$ and an exponent biased by 128. The PDP-11 SINGLE precision numbers have a magnitude between 2^{-128} and $\approx 2^{128}$, such that overflows and underflows are symmetric.

Zeros and infinities. The IEEE standard has two zero values ($+0$ and -0) and two infinities ($+\infty$ and $-\infty$), and has been called the *two zero system*. An alternate approach, the *three zero system*, was suggested by Fraley [13]. His system has values $+0$, -0, and 0, $+\infty$, $-\infty$, and ∞.

The basic properties of the two systems are shown below:

2-Zero	3-Zero	Difference
$+0 = -0$	$-0 < 0 < +0$	3 zero system introduces an unsigned zero
$-\infty < +\infty$ or $-\infty = +\infty$	$-\infty < +\infty$ or ∞ not comparable	
$x - x = +0$	$x - x = 0$	
$1/+0 = +\infty$	$1/+0 = +\infty$	
$1/-0 = -\infty$	$1/-0 = -\infty$	
	$1/0 = \infty$	

The main advantage of the three zeros system is the availability of a true zero and a true infinity in the algebraic sense. This is illustrated by the following two examples.

1. Suppose $f(x) = e^{1/x}$. In the two zeros system we have:
$$f(-0) = +0,$$
$$f(+0) = +\infty;$$
thus, $f(-0) \neq f(+0)$, even though $-0 = +0$.
This, of course, is a contradiction of the basic theorem:
$$\text{if } x = y \text{ then } f(x) = f(y).$$
By contrast, in the three zeros system, this theorem holds since:
$$-0 \neq +0.$$

2. The two zeros system fails to distinguish zeros that result from underflow from those which are mathematically zero. The result of $x - x$ is $+0$ in the two zeros system. In the three zeros system, $x - x = 0$, whereas $+0$ is the result of an underflow of a positive number; that is,
$$0 < +0 < \text{smallest representable number}.$$

1.9 Cray Floating Point

The IEEE standard is an attempt to provide functionality and information to the floating point user. All floating point designs are necessarily compromises between user functionality and engineering requirements; between function and performance. A clear illustration of this is the Cray Research Corporation floating point design (as used in the CRAY-1 and CRAY-XMP). The Cray format is primarily organized about performance considerations, providing an interesting contrast to the IEEE standard.

1.9.1 Data Format

As before, the format ($\beta = 2$) consists of sign bit, biased exponent and fraction (mantissa):

S	E	F
1	$\leftarrow$ 15 $\rightarrow$	$\leftarrow$ 48 $\rightarrow$

where S = sign bit of fraction
E = biased exponent
F = fraction
then
e = true exponent = E-bias
f = true mantissa = 0.F

A normalized nonzero number X would be represented as

$$X = (-1)^S * 2^{E-\text{bias}} * (0.\text{F})$$

bias $= 2^{14} = 16384$

1.9.2 Machine Maximum

$$max = 2^{2^{13}-1}(1 - 2^{-48}) = 2^{8191}(1 - 2^{-48}).$$

Note that overflow is strictly defined by the exponent value. Any result with an exponent containing two leading ones is said to have *overflowed.*

1.9.3 Machine Minimum

$$min = 2^{-(2^{13})} \cdot 2^{-1} = 2^{-8193}.$$

Any result with an exponent containing two leading zeros is said to have *underflowed.* There are no underflow interrupt flags on the Cray machines; underflowed results are to be set to zero. Notice the relatively large range of representations that are designated "underflowed" or "overflowed."

To further simplify (and speed up) implementations, the range tests (tests for nonzero numbers which exceed *max* or are under *min*) are largely performed *before postnormalization*! (There is

an exception.) To expedite matters still further, range testing is *not* done on input operands (except zero testing)!

This gives rise to a number of curious results:

1. $(min + s) - min = s$, where s is 2^{-2} through 2^{-48} times *min*, since $min + s > min$ *before postnormalization.*

   The machine will normalize such results producing a number up to 2^{-48} smaller than *min*. This number is not set to zero.

2. A number below *min*, call it s, can participate in computations. Thus,

   - $min + s =$ is produced as the sum of *min* and s.
   - $s + 0 = 0$, since now the invalid exponent of s is detected in the floating point adder result.
   - $s * 1.0 = 0$ if s is less than $2^{-1} * min$, since the sum of exponents is less than *min* (recall 1.0 is represented by exponent = 1, fraction = 1/2).
   - $s * 1.0 = s$ if $min > s > 2^{-1} * min$, since the sum of the exponents *before* postnormalization is equal to *min*.
   - $s * Y = 0$ if the exponent of Y is not positive enough to bring $\exp(s) + \exp(Y)$ into range.
   - $s * Y = s * Y$ if $\exp(s) + \exp(Y) \geq \exp(min)$.

On overflow, the machine may be interrupted (maskable). An uninterrupted overflow is represented by $\exp(max) + 1$ or 11000..0 (bias $+2^{13}$) in the exponent field (actually, 11xx...x indicates an overflow condition). The fraction may be anything.

Overflow checking is performed on multiply. If the upper bits of the exponent are "1", the result is set to "overflow" (exponent = 1100...0) unless the other argument is zero (exponent = 000..0, fraction = xx...x), in which case the result is zero (all zeros exponent and fraction).

Still, it is possible to have the following:

$$max * 1.0 = max \text{ with overflow flag set.}$$

This is because 1.0 has exp = 1, which causes the result exponent to overflow *before* postnormalization.

The input multiplier operands are not checked for underflow, as just illustrated.

1.9.4 Treatment of Zero

Cray represents zero as all 0's (i.e., positive sign) and sets a detected underflowed number to zero. The machine checks input operands for zero by exponent inspection only. Further, the Cray machine uses the floating point multiplier to do integer operations. Integers are detected by having all zeros in the "exponent" field for *both* operands. If only one operand of the

multiplier has a zero exponent, that operand is interpreted as floating point zero and the result of multiplication is zero regardless of the value of the other operand. Thus,

$$(\text{zero}) * (+\text{overflow}) = \text{zero},$$

since zero takes precedence. Zero is (almost) always designated as +00...0. Thus, even in this case:

$$(+\text{zero}) * (-\text{overflow}) = +\text{zero}.$$

However, in the case of

$$(+\text{zero}) * (-\text{zero}) = -\text{zero},$$

since both exponents are zero, the operands are interpreted as valid *integer* operands and the sign is computed as such. However,

$$(-\text{zero}) * (Y) = (+\text{zero})$$

for any nonzero value of Y, since +zero is "always" the result of multiplication with a zero exponent operand.

1.9.5 Operations

The Cray systems have three floating point functional units:

- Floating Point Add/Subtract.
- Floating Point Multiplication.
- Floating Point Reciprocal.

On floating point add/subtract, the fraction result is checked for all zeros. In this case, the sign is set and the exponent is set to all zeros. No such checking is performed as multiplication.

1.9.6 Overflow

As mentioned earlier, overflow is detected on the results of add and multiply, and on the input operands of multiply. In overflow detection, the result exponent is set to exp(max)+1—two leading exponent "1"s followed by "0"s. The fraction for all operations is unaffected by results in an overflow condition.

The exceptions to the test for over/underflow on result (only) before postnormalization are two:

- The input argument to multiply are tested for overflow.
- The result of addition is tested for overflow (also) after postnormalization. This is (in part) a natural consequence of the operation

  $$max + max = \text{overflow}$$

  and the overflow flag is set. Also,

  $$(-max) + (-max) = +\text{overflow}.$$

  The sign of the overflow designation is correctly set.

		test on input	test on output before post-normalization	test on output after post-normalization
underflow	+/−	No	Yes	No
	*	No	Yes	No
overflow	+/−	No	Yes	Yes
	*	Yes	Yes	No

Table 1.6: Underflow/overflow designations.

Thus, the "overflow" designation is somewhat "firmer" than "underflow." Table 1.6 illustrates the difference.

Since fractions are not checked on multiply, some anomalies may result, such as:

$$\text{overflow} * 0.0 \times 2^1 = \text{overflow with 0.0 fraction.}$$

1.10 Additional Readings

Sterbenz [35] is an excellent introduction to the problem of floating point computation—a comprehensive treatment of the earlier approaches to floating point representation and their difficulties.

The January 1980 and March 1981 issues of IEEE *Computer* have several valuable articles on the proposed standard; Stevenson [36] provides a precise description of the proposed IEEE 754 standard with good introductory remarks.

Cody [7] provides a detailed analysis of the three major proposals and shows the similarity between all of them.

Coonen [10] gives an excellent tutorial on underflows and denormalized numbers. He attempts to clear the misconceptions about gradual underflows and shows how it fits naturally into the proposed standard.

Hough [22] describes applications of the standard for computing elementary functions such as trigonometric and exponential functions. This interesting article also explains the need for some of the unique features of the standard: extended formats, unbiased rounding, and infinite operands.

Coonen [9] also published a guide for the implementation of the standard. His guide provides practical algorithms for floating point arithmetic operations and suggests the hardware/software mix for handling exceptions. His guide also includes a narrative description of the standard, including the QUAD format. For actual hardware implementation of the IEEE Standard, see additional readings on page **??**.

1.11 Summary

In arithmetic, the representation of integers is a key problem. Machines, by their nature, have a finite set of symbols or codes upon which they can operate, as contrasted with the infinity that they are supposed to represent. This finitude defines a modular form of arithmetic widely used in computing systems. The familiar modular system, a single binary base, lends itself readily towards complement coding schemes which serve to scale negative numbers into the positive integer domain.

Pairs of *signed* integers can be used to represent approximations to real numbers called floating point numbers. Floating point representations broadly involve tradeoffs between precision, range, and implementation problems. With the relatively decreasing importance of implementation costs, the possibility of defining more suitable floating point representations has led to efforts toward a standard floating point representation.

1.12 Exercises

1. Consider the operation of integer division: $\pm 11 \div \pm 5$; that is, find the quotient and remainder for each of the four sign combinations.

   (a) For signed integers.

   (b) For modulus division.

2. If we denote $\div_m$ as modular division (q_m, r_m quotient and remainder) and $\div_s$ (q_s, r_s) as signed division, find q_s, r_s in terms of q_m, r_m.

3. Another type of division is possible; this is called "floor division." In this, the quotient is the greatest integer that is contained by (is less than or equal to) the numerator divided by the denominator (note that minus 3 is greater than minus 4). Find q_f, r_f in terms of q_m, r_m.

4. Find an algorithm for computing X mod M for known M, using only the addition, subtraction, multiplication, and comparison operations. You should not make any assumptions as to the relative size of X and M in considering this problem.

5. Find an algorithm for multiplication with negative numbers using an unsigned multiplication operation. Show either that nothing need be done, or describe in detail what must be done to the product of the two numbers, one or both of which may be in complement form.

   (a) For radix complement.

   (b) For diminished radix complement.

6. For a variety of reasons, a special purpose machine is built that uses 32-bit representation for floating point numbers. A minimum of 24 bits of precision is required. Compare a 370-like (truncation) system with a simple binary system.

7. Repeat the previous problem, changing the binary system to a modified version of the IEEE standard. Compare the IBM/370 format to the IEEE standard format with respect to the (a) associative, (b) commutative, (c) distributive properties of basic arithmetic operations. In which cases do the properties fail?

8. Find the value of *max* and *min* (largest and smallest representable numbers) for single and double precision.

   (a) IEEE standard.
   (b) System/370.
   (c) PDP-11.

9. For the computation $s = a - b * c$ (a, b, c, s are floating point numbers that must be rounded), find the guaranteed significance interval $[s_{\min}, s_{\max}]$ in terms of a, b, and c, and the Yohe rounding operations ∇, Δ, T, A, and RN.

10. For IEEE single precision, if $A = (1).0100\ldots \times 2^{-126}$, $B = (1).000\ldots \times 2^{-3}$, and $C = (1).000\ldots \times 2^{5}$ (A, B, and C are positive):

    (a) What is the result of $A * B * C$, RM round, if performed $(A * B) * C$?
    (b) Repeat, if performed $A * (B * C)$.
    (c) Find $A + B + C$, RP round.
    (d) If $D = (1).01000\ldots \times 2^{122}$, find $C * D$, RP round.
    (e) Find $(2 * C) * D$, RZ round.

11. All of the floating point representations studied use sign and magnitude to represent the mantissa, and excess code for the exponent. Instead, consider a floating point representation system that uses radix 2 complement coding for both the mantissa and the exponent for a binary based system.

    (a) If the magnitude of a normalized mantissa is in the range $1/2 < m < 1$, where is the implied binary point?
    (b) Can this representation make use of a technique similar to the hidden one technique studied in class? If so, which bit is hidden and what is its value? If not, why not?

12. When performing addition and subtraction on floating point numbers, guard bits are used to maintain precision and for rounding operations. Prove that only one guard bit is needed to maintain precision when performing subtraction. Assume normalized numbers and a binary radix.
    Hint: In each case, consider how many bits the smaller mantissa is shifted during alignment, and how many bits the result is shifted during post-normalization.

13. In each of the following, you are given the ALU output of floating point operations *before* post-normalization and rounding. An IEEE-type format is assumed, but (for problem simplicity) only four bits of fraction are used (i.e., a hidden "1".xxx, plus three bits)—and three fraction bits are stored.

    M is the most significant bit, immediately to the left of the radix point.
    X are intermediate bits.
    L is the least significant bit.
    S is the sign (1 = neg., 0 = pos.)

(1) Show results after post-normalization and rounding—exactly the way the fraction will be stored. (2) Note the effects (the change) in exponent value.

(a) Result after subtraction, round RZ

S	$M.XXLGRS$
1	0 0 0 0 1 0 0

Result after post-normalization and round:

S	significand	change to exponent
_____	_____	_____

(b) Result after multiplication, round RN

S	$M.XXLGRS$
0	1 0 1 0 1 0 1 0

Result after post-normalization and round:

S	significand	change to exponent
_____	_____	_____

(c) Result after multiplication, round RN

S	$M.XXLGRS$
0	1 1 1 1 1 1 0 0

Result after post-normalization and round:

S	significand	change to exponent
_____	_____	_____

(d) Result after addition, RM

S	$M.XXLGRS$
1	1 0 1 1 0 0 0 1

Result after post-normalization and round:

S	significand	change to exponent
_____	_____	_____

14. On page 37, there is an action table for RN. Create a similar table for RP. State all input bits used and all actions on the final round bit, A.