Floating Point Representation

Michael L. Overton

1 Computer Representation of Numbers

Computers which work with real arithmetic use a system called *floating point*. Suppose a real number x has the binary expansion

$$x = \pm m \times 2^E, \quad \text{where } 1 \leq m < 2$$

and

$$m = (b_0.b_1b_2b_3\ldots)_2.$$

To store a number in floating point representation, a computer word is divided into 3 fields, representing the sign, the exponent E, and the significand m respectively. A 32-bit word could be divided into fields as follows: 1 bit for the sign, 8 bits for the exponent and 23 bits for the significand. Since the exponent field is 8 bits, it can be used to represent exponents between -128 and 127. The significand field can store the first 23 bits of the binary representation of m, namely

$$b_0.b_1\ldots b_{22}.$$

If $b_{23}, b_{24}, \ldots$ are not all zero, this floating point representation of x is not exact but approximate. A number is called a *floating point number* if it can be stored *exactly* on the computer using the given floating point representation scheme, i.e. in this case, $b_{23}, b_{24}, \ldots$ are all zero. For example, the number

$$11/2 = (1.011)_2 \times 2^2$$

would be represented by

0	$E = 2$	1.0110000000000000000000

,

and the number

$$71 = (1.000111)_2 \times 2^6$$

would be represented by

$$\begin{array}{|c|c|c|}\hline 0 & E = 6 & 1.00011100000000000000000 \\ \hline \end{array}.$$

To avoid confusion, the exponent E, which is actually stored in a binary representation, is shown in decimal for the moment.

The floating point representation of a nonzero number is unique as long as we require that $1 \leq m < 2$. If it were not for this requirement, the number $11/2$ could also be written

$$(0.01011)_2 \times 2^4$$

and could therefore be represented by

$$\begin{array}{|c|c|c|}\hline 0 & E = 4 & 0.01011000000000000000000 \\ \hline \end{array}.$$

However, this is not allowed since $b_0 = 0$ and so $m < 1$. A more interesting example is

$$1/10 = (0.0001100110011\ldots)_2.$$

Since this binary expansion is infinite, we must *truncate* the expansion somewhere. (An alternative, namely *rounding*, is discussed later.) The simplest way to truncate the expansion to 23 bits would give the representation

$$\begin{array}{|c|c|c|}\hline 0 & E = 0 & 0.0001100110011001100110 \\ \hline \end{array},$$

but this means $m < 1$ since $b_0 = 0$. An even worse choice of representation would be the following: since

$$1/10 = (0.00000001100110011\ldots)_2 \times 2^4,$$

the number could be represented by

$$\begin{array}{|c|c|c|}\hline 0 & E = 4 & 0.0000000110011001100110 \\ \hline \end{array}.$$

This is clearly a bad choice since less of the binary expansion of $1/10$ is stored, due to the space wasted by the leading zeros in the significand field. *This is the reason why $m < 1$, i.e. $b_0 = 0$, is not allowed.* The only allowable representation for $1/10$ uses the fact that

$$1/10 = (1.100110011\ldots)_2 \times 2^{-4},$$

giving the representation

$$\begin{array}{|c|c|c|}\hline 0 & E=-4 & 1.1001100110011001100110 \\ \hline\end{array}\,.$$

This representation includes *more of the binary expansion of* $1/10$ than the others, and is said to be *normalized*, since $b_0 = 1$, i.e. $m > 1$. Thus none of the available bits is wasted by storing leading zeros.

We can see from this example why the name *floating point* is used: the binary *point* of the number $1/10$ can be *floated* to any position in the bitstring we like by choosing the appropriate exponent: the normalized representation, with $b_0 = 1$, is the one which should be always be used when possible. It is clear that an irrational number such as π is also represented most accurately by a normalized representation: significand bits should not be wasted by storing leading zeros. However, the number *zero* is special. It cannot be normalized, since all the bits in its representation are zero. The exponent E is irrelevant and can be set to zero. Thus, zero could be represented as

$$\begin{array}{|c|c|c|}\hline 0 & E=0 & 0.0000000000000000000000 \\ \hline\end{array}\,.$$

The gap between the number 1 and the next largest floating point number is called the *precision* of the floating point system, [1] or, often, the *machine precision*, and we shall denote this by ϵ. In the system just described, the next floating point bigger than 1 is

$$1.0000000000000000000001,$$

with the last bit $b_{22} = 1$. Therefore, the precision is $\epsilon = 2^{-22}$.

Exercise 1 *What is the smallest possible positive normalized floating point number using the system just described?*

Exercise 2 *Could nonzero numbers instead be normalized so that* $\frac{1}{2} \le m < 1$*? Would this be just as good?*

It is quite instructive to suppose that the computer word size is much smaller than 32 bits and work out in detail what all the possible floating numbers are in such a case. Suppose that the significand field has room only to store $b_0.b_1b_2$, and that the only possible values for the exponent E are -1, 0 and 1. We shall call this system our *toy floating point system*. The set of toy floating point numbers is shown in Figure 1

[1] Actually, the usual definition of precision is one half of this quantity, for reasons that will become apparent in the next section. We prefer to omit the factor of one half in the definition.

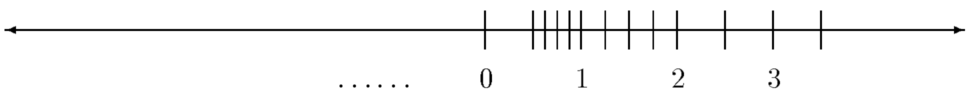


Figure 1: The Toy Floating Point Numbers

The largest number is $(1.11)_2 \times 2^1 = (3.5)_{10}$, and the smallest positive normalized number is $(1.00)_2 \times 2^{-1} = (0.5)_{10}$. All of the numbers shown are normalized except zero. Since the next floating point number bigger than 1 is 1.25, the precision of the toy system is $\epsilon = 0.25$. Note that the gap between floating point numbers becomes *smaller* as the magnitude of the numbers themselves get smaller, and *bigger* as the numbers get bigger. Note also that the gap between between zero and the smallest positive number is *much bigger* than the gap between the smallest positive number and the next positive number. We shall show in the next section how this gap can be "filled in" with the introduction of "subnormal numbers".

2 IEEE Floating Point Representation

In the 1960's and 1970's, each computer manufacturer developed its own floating point system, leading to a lot of inconsistency as to how the same program behaved on different machines. For example, although most machines used binary floating point systems, the IBM 360/370 series, which dominated computing during this period, used a hexadecimal base, i.e. numbers were represented as $\pm m \times 16^E$. Other machines, such as HP calculators, used a decimal floating point system. Through the efforts of several computer scientists, particularly W. Kahan, a binary floating point standard was developed in the early 1980's and, most importantly, followed very carefully by the principal manufacturers of floating point chips for personal computers, namely Intel and Motorola. This standard has become known as the IEEE floating point standard since it was developed and endorsed by a working committee of the Institute for Electrical and Electronics Engineers.[2] (There is also a decimal version of the standard but we shall not discuss this.)

The IEEE standard has three very important requirements:

[2] ANSI/IEEE Std 754-1985. Thanks to Jim Demmel for introducing the author to the standard.

- consistent representation of floating point numbers across all machines adopting the standard
- correctly rounded arithmetic (to be explained in the next section)
- consistent and sensible treatment of exceptional situations such as division by zero (to be discussed in the following section).

We will not describe the standard in detail, but we will cover the main points.

We start with the following observation. In the last section, we chose to normalize a nonzero number x so that $x = m \times 2^E$, where $1 \leq m < 2$, i.e.

$$m = (b_0.b_1b_2b_3\ldots)_2,$$

with $b_0 = 1$. In the simple floating point model discussed in the previous section, we stored the leading nonzero bit b_0 in the first position of the field provided for m. Note, however, that since we know this bit has the value one, *it is not necessary to store it.* Consequently, we can use the 23 bits of the significand field to store $b_1, b_2, \ldots, b_{23}$ instead of $b_0, b_1, \ldots, b_22$, changing the machine precision from $\epsilon = 2^{-22}$ to $\epsilon = 2^{-23}$. Since the bitstring stored in the significand field is now actually the *fractional part* of the significand, we shall refer henceforth to the field as the *fraction field.* Given a string of bits in the fraction field, it is necessary to imagine that the symbols "1." appear in front of the string, even though these symbols are not stored. This technique is called *hidden bit normalization* and was used by Digital for the Vax machine in the late 1970's.

Exercise 3 *Show that the hidden bit technique does not result in a more accurate representation of* 1/10. *Would this still be true if we had started with a field width of 24 bits before applying the hidden bit technique?*

Note an important point: since zero cannot be normalized to have a leading nonzero bit, hidden bit representation *requires a special technique for storing zero.* We shall see what this is shortly. A pattern of all zeros in the fraction field of a normalized number represents the significand 1.0, not 0.0.

Zero is not the only special number for which the IEEE standard has a special representation. Another special number, not used on older machines but very useful, is the number ∞. This allows the possibility of dividing a nonzero number by 0 and storing a sensible mathematical result, namely ∞, instead of terminating with an overflow message. This turns out to be

Table 1: IEEE Single Precision

$\pm$	$a_1a_2a_3\ldots a_8$	$b_1b_2b_3\ldots b_{23}$

If exponent bitstring $a_1\ldots a_8$ is	Then numerical value represented is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\ldots b_{23})_2 \times 2^{-126}$
$(00000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-125}$
$(00000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{-124}$
$\downarrow$	$\downarrow$
$(01111111)_2 = (127)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{0}$
$(10000000)_2 = (128)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111100)_2 = (252)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{125}$
$(11111101)_2 = (253)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{126}$
$(11111110)_2 = (254)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \times 2^{127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1 = \ldots = b_{23} = 0$, NaN otherwise

very useful, as we shall see later, although one must be careful about what is meant by such a result. One question which then arises is: what about $-\infty$? It turns out to be convenient to have representations for $-\infty$ as well as ∞ and -0 as well as 0. We will give more details later, but note for now that -0 and 0 are *two different representations for the same value zero*, while $-\infty$ and ∞ represent *two very different numbers*. Another special number is NaN, which stands for "Not a Number" and is consequently not really a number at all, but an error pattern. This too will be discussed further later. All of these special numbers, as well as some other special numbers called subnormal numbers, are represented through the use of a special bit pattern in the exponent field. This slightly reduces the exponent range, but this is quite acceptable since the range is so large.

There are three standard types in IEEE floating point arithmetic: single precision, double precision and extended precision. Single precision numbers require a 32-bit word and their representations are summarized in Table 1.

Let us discuss Table 1 in some detail. The $\pm$ refers to the sign of the number, a zero bit being used to represent a positive sign. The first line shows that the representation for zero requires a special zero bitstring for the exponent field *as well as* a zero bitstring for the fraction, i.e.

0	00000000	0000000000000000000000

.

No other line in the table can be used to represent the number zero, for all lines except the first and the last represent normalized numbers, with an initial bit equal to one; this is the one that is not stored. In the case of the first line of the table, the initial unstored bit is zero, not one. The 2^{-126} in the first line is confusing at first sight, but let us ignore that for the moment since $(0.000\ldots0)_2 \times 2^{-126}$ is certainly one way to write the number zero. In the case when the exponent field has a zero bitstring but the fraction field has a nonzero bitstring, the number represented is said to be *subnormal.* [3] Let us postpone the discussion of subnormal numbers for the moment and go on to the other lines of the table.

All the lines of Table 1 except the first and the last refer to the normalized numbers, i.e. all the floating point numbers which are not special in some way. Note especially the relationship between the exponent bitstring $a_1a_2a_3\ldots a_8$ and the actual exponent E, i.e. the power of 2 which the bitstring is intended to represent. We see that the exponent representation does not use any of sign-and-modulus, 2's complement or 1's complement, but rather something called *biased representation*: the bitstring which is stored is simply the binary representation of $E+127$. In this case, the number 127 which is added to the desired exponent E is called the *exponent bias.* For example, the number $1 = (1.000\ldots0)_2 \times 2^0$ is stored as

0	01111111	00000000000000000000000

.

Here the exponent bitstring is the binary representation for $0+127$ and the fraction bitstring is the binary representation for 0 (the fractional part of 1.0). The number $11/2 = (1.011)_2 \times 2^2$ is stored as

0	10000001	0110000000000000000000

,

and the number $1/10 = (1.100110011\ldots)_2 \times 2^{-4}$ is stored as

0	01111011	10011001100110011001100

.

We see that the range of exponent field bitstrings for normalized numbers is 00000001 to 11111110 (the decimal numbers 1 through 254), representing

[3] These numbers were called *denormalized* in early versions of the standard.

actual exponents from $E_{min} = -126$ to $E_{max} = 127$. The smallest normalized number which can be stored is represented by

0	00000001	00000000000000000000000

meaning $(1.000\ldots0)_2 \times 2^{-126}$, i.e. 2^{-126}, which is approximately 1.2×10^{-38}, while the largest normalized number is represented by

0	11111110	11111111111111111111111

meaning $(1.111\ldots1)_2 \times 2^{127}$, i.e. $(2 - 2^{-23}) \times 2^{127}$, which is approximately 3.4×10^{38}.

The last line of Table 1 shows that an exponent bitstring consisting of all ones is a special pattern used for representing $\pm\infty$ and NaN, depending on the value of the fraction bitstring. We will discuss the meaning of these later.

Finally, let us return to the first line of the table. The idea here is as follows: although 2^{-126} is the smallest normalized number which can be represented, we can use the combination of the special zero exponent bitstring and a nonzero fraction bitstring to represent smaller numbers called *subnormal* numbers. For example, 2^{-127}, which is the same as $(0.1)_2 \times 2^{-126}$, is represented as

0	00000000	10000000000000000000000

,

while $2^{-149} = (0.0000\ldots01)_2 \times 2^{-126}$ (with 22 zero bits after the binary point) is stored as

0	00000000	00000000000000000000001

.

This last number is the smallest nonzero number which can be stored. Now we see the reason for the 2^{-126} in the first line. It allows us to represent numbers in the range immediately below the smallest normalized number. Subnormal numbers cannot be normalized, since that would result in an exponent which does not fit in the field.

Let us return to our example of a machine with a tiny word size, illustrated in Figure 1, and see how the addition of subnormal numbers changes it. We get three extra numbers: $(0.11)_2 \times 2^{-1} = 3/8$, $(0.10)_2 \times 2^{-1} = 1/4$ and $(0.01)_2 \times 2^{-1} = 1/8$: these are shown in Figure 2.

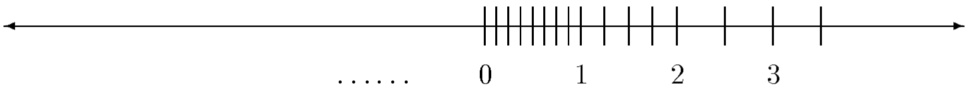


Figure 2: The Toy System including Subnormal Numbers

Note that *the gap between zero and the smallest positive normalized number is nicely filled in by the subnormal numbers*, using the same spacing as that between the normalized numbers with exponent -1.

Subnormal numbers are *less accurate*, i.e. they have less room for nonzero bits in the fraction field, than normalized numbers. Indeed, the accuracy drops as the size of the subnormal number decreases. Thus $(1/10) \times 2^{-123} = (0.11001100\ldots)_2 \times 2^{-126}$ is stored as

0	00000000	11001100110011001100110

,

while $(1/10) \times 2^{-133} = (0.11001100\ldots)_2 \times 2^{-136}$ is stored as

0	00000000	00000000001100110011001

.

Exercise 4 *Determine the IEEE single precision floating point representation of the following numbers: 2, 1000, 23/4,* $(23/4) \times 2^{100}$, $(23/4) \times 2^{-100}$, $(23/4) \times 2^{-135}$, $1/5$, $1024/5$, $(1/10) \times 2^{-140}$.

Exercise 5 *Write down an algorithm that tests whether a floating point number x is less than, equal to or greater than another floating point number y, by simply comparing their floating point representations bitwise from left to right, stopping as soon as the first differing bit is encountered. The fact that this can be done easily is the main motivation for biased exponent notation.*

Exercise 6 *Suppose x and y are single precision floating point numbers. Is it true*[4] *that* $\mathrm{round}(x - y) = 0$ *only when $x = y$? Illustrate your answer with some examples. Do you get the same answer if subnormal numbers are not allowed, i.e. subnormal results are rounded to zero? Again, illustrate with an example.*

[4]The expression $\mathrm{round}(x)$ is defined in the next section.

Table 2: IEEE Double Precision

$\pm$	$a_1a_2a_3\ldots a_{11}$	$b_1b_2b_3\ldots b_{52}$

If exponent bitstring is $a_1\ldots a_{11}$	Then numerical value represented is
$(00000000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1022}$
$(00000000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1022}$
$(00000000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1021}$
$(00000000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1020}$
$\downarrow$	$\downarrow$
$(01111111111)_2 = (1023)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{0}$
$(10000000000)_2 = (1024)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111111100)_2 = (2044)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1021}$
$(11111111101)_2 = (2045)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1022}$
$(11111111110)_2 = (2046)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1023}$
$(11111111111)_2 = (2047)_{10}$	$\pm\infty$ if $b_1 = \ldots = b_{52} = 0$, NaN otherwise

For many applications, single precision numbers are quite adequate. However, double precision is a commonly used alternative. In this case each floating point number is stored in a 64-bit double word. Details are shown in Table 2. The ideas are all the same; only the field widths and exponent bias are different. Clearly, a number like 1/10 with an infinite binary expansion is stored more accurately in double precision than in single, since $b_1, \ldots, b_{52}$ can be stored instead of just $b_1, \ldots, b_{23}$.

There is a third IEEE floating point format called extended precision. Although the standard does not require a particular format for this, the standard implementation used on PC's is an 80-bit word, with 1 bit used for the sign, 15 bits for the exponent and 64 bits for the significand. The leading bit of a normalized number is not generally hidden as it is in single and double precision, but is explicitly stored. Otherwise, the format is much the same as single and double precision.

We see that the first *single precision* number larger than 1 is $1 + 2^{-23}$, while the first *double precision* number larger than 1 is $1+2^{-52}$. The extended precision case is a little more tricky: since there is no hidden bit, $1 + 2^{-64}$ cannot be stored exactly, so the first number larger than 1 is $1+2^{-63}$. Thus the

Table 3: What is that Precision?

IEEE Single	$\epsilon = 2^{-23} \approx 1.2 \times 10^{-7}$
IEEE Double	$\epsilon = 2^{-52} \approx 1.1 \times 10^{-16}$
IEEE Extended	$\epsilon = 2^{-63} \approx 1.1 \times 10^{-19}$

exact machine precision, together with its approximate decimal equivalent, is shown in Table 3 for each of the IEEE single, double and extended formats.

The fraction of a single precision normalized number has *exactly* 23 *bits of accuracy*, i.e. the significand has 24 bits of accuracy counting the hidden bit. This corresponds to *approximately* 7 decimal digits of accuracy. In double precision, the fraction has *exactly* 52 *bits of accuracy*, i.e. the significand has 53 bits of accuracy counting the hidden bit. This corresponds to *approximately* 16 *decimal digits of accuracy.* In extended precision, the significand has *exactly* 64 *bits of accuracy*, and this corresponds to *approximately* 19 *decimal digits of accuracy.* So, for example, the single precision representation for the number π is *approximately* 3.141592, while the double precision representation is *approximately* 3.141592653589793. To see *exactly* what the single and double precision representations of π are would require writing out the binary representations and converting these to decimal.

Exercise 7 *What is the gap between 2 and the first IEEE single precision number larger than 2? What is the gap between 1024 and the first IEEE single precision number larger than 1024? What is the gap between 2 and the first IEEE double precision number larger than 2?*

Exercise 8 *Let $x = m \times 2^E$ be a normalized single precision number, with $1 \leq m < 2$. Show that the gap between x and the next largest single precision number is*

$$\epsilon \times 2^E.$$

(It may be helpful to recall the discussion following Figure 1.)

3 Rounding and Correctly Rounded Arithmetic

We use the terminology "floating point numbers" to mean all acceptable numbers in a given IEEE floating point arithmetic format. This set consists of ± 0, subnormal and normalized numbers, and $\pm\infty$, but not NaN values, and is a finite subset of the reals. We have seen that most real numbers, such as $1/10$ and π, cannot be represented exactly as floating point numbers. For ease of expression we will say a general real number is "normalized" if its modulus lies between the smallest and largest positive normalized floating point numbers, with a corresponding use of the word "subnormal". In both cases the representations we give for these numbers will parallel the floating point number representations in that $b_0 = 1$ for normalized numbers, and $b_0 = 0$ with $E = -126$ for subnormal numbers.

For any number x which is not a floating point number, there are two obvious choices for the floating point approximation to x: the closest floating point number *less* than x, and the closest floating point number *greater* than x. Let us denote these x_- and x_+ respectively. For example, consider the toy floating point number system illustrated in Figures 1 and 2. If $x = 1.7$, for example, then we have $x_- = 1.5$ and $x_+ = 1.75$, as shown in Figure 3.

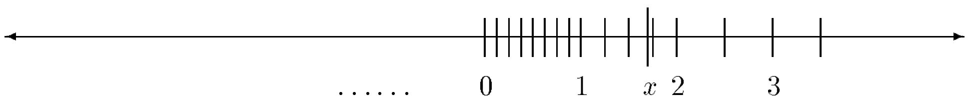


Figure 3: Rounding in the Toy System

Now let us assume that the floating point system we are using is IEEE single precision. Then if our general real number x is *positive*, (and normalized or subnormal), with

$$x = (b_0.b_1b_2 \ldots b_{23}b_{24}b_{25} \ldots)_2 \times 2^E,$$

we have

$$x_- = (b_0.b_1b_2 \ldots b_{23})_2 \times 2^E.$$

Thus, x_- is obtained simply by truncating the binary expansion of m at the 23rd bit and discarding b_{24}, b_{25}, etc. This is clearly the closest floating point number which is less than x. Writing a formula for x_+ is more complicated since, if $b_{23} = 1$, finding the closest floating point number bigger than x will involve some bit "carries" and possibly, in rare cases, a change in E. If x is negative, the situation is reversed: it is x_+ which is obtained by dropping bits b_{24}, b_{25}, etc., since discarding bits of a negative number makes the number closer to zero, and therefore larger (further to the right on the real line).

The IEEE standard defines the *correctly rounded value of x*, which we shall denote round(x), as follows. If x happens to be a floating point number, then round$(x) = x$. Otherwise, the correctly rounded value depends on which of the following four *rounding modes* is in effect:

- *Round down.*
  round$(x) = x_-$.
- *Round up.*
  round$(x) = x_+$.
- *Round towards zero.*
  round(x) is either x_- or x_+, whichever is between zero and x.
- *Round to nearest.*
  round(x) is either x_- or x_+, whichever is nearer to x. In the case of a tie, the one with its *least significant bit equal to zero* is chosen.

If x is positive, then x_- is between zero and x, so *round down* and *round towards zero* have the same effect. If x is negative, then x_+ is between zero and x, so it is *round up* and *round towards zero* which have the same effect. In either case, *round towards zero* simply requires truncating the binary expansion, i.e. discarding bits.

The most useful rounding mode, and the one which is *almost always* used, is *round to nearest*, since this produces the floating point number which is closest to x. In the case of "toy" precision, with $x = 1.7$, it is clear that *round to nearest* gives a rounded value of x equal to 1.75. When the word "round" is used without any qualification, it almost always means "round to nearest". In the more familiar decimal context, if we "round" the number $\pi = 3.14159\ldots$ to four decimal digits, we obtain the result 3.142, which is closer to π than the truncated result 3.141.

Exercise 9 *What is the rounded value of* $1/10$ *for each of the four rounding modes? Give the answer in terms of the binary representation of the number, not the decimal equivalent.*

The (absolute value of the) difference between round(x) and x is called the *absolute rounding error* associated with x, and its value depends on the rounding mode in effect. In toy precision, when *round down* or *round towards zero* is in effect, the absolute rounding error for $x = 1.7$ is 0.2 (since round$(x) = 1.5$), but if *round up* or *round to nearest* is in effect, the absolute rounding error for $x = 1.7$ is 0.05 (since $\text{round}(x) = 1.75$). For all rounding modes, it is clear that the *absolute rounding error associated with x is less than the gap between x_- and x_+*, while in the case of *round to nearest*, the absolute rounding error can be *no more than half the gap between x_- and x_+*.

Now let x be a normalized IEEE single precision number, and suppose that $x > 0$, so that

$$x = (b_0.b_1b_2 \ldots b_{23}b_{24}b_{25} \ldots)_2 \times 2^E,$$

with $b_0 = 1$. Clearly,

$$x_- = (b_0.b_1b_2 \ldots b_{23})_2 \times 2^E.$$

Thus we have, for any rounding mode, that

$$|\text{round}(x) - x| < 2^{-23} \times 2^E,$$

while for *round to nearest*

$$|\text{round}(x) - x| \leq 2^{-24} \times 2^E.$$

Similar results hold for double and extended precision, replacing 2^{-23} by 2^{-52} and 2^{-63} respectively, so that in general we have

$$|\text{round}(x) - x| < \epsilon \times 2^E, \tag{1}$$

for any rounding mode and

$$|\text{round}(x) - x| \leq \frac{1}{2}\epsilon \times 2^E,$$

for *round to nearest.*

Exercise 10 *For* round towards zero, *could the absolute rounding error be exactly equal to $\epsilon \times 2^E$? For* round to nearest, *could the absolute rounding error be exactly equal to $\frac{1}{2}\epsilon \times 2^E$?*

Exercise 11 *Does (1) hold if x is subnormal, i.e. $E = -126$ and $b_0 = 0$?*

The presence of the factor 2^E is inconvenient, so let us consider the *relative rounding error* associated with x, defined to be

$$\delta = \frac{\text{round}(x)}{x} - 1 = \frac{\text{round}(x) - x}{x}.$$

Since for normalized numbers

$$x = \pm m \times 2^E, \qquad \text{where } m \geq 1$$

(because $b_0 = 1$) we have, for all rounding modes,

$$|\delta| < \frac{\epsilon \times 2^E}{2^E} = \epsilon. \tag{2}$$

In the case of *round to nearest*, we have

$$|\delta| \leq \frac{\frac{1}{2}\epsilon \times 2^E}{2^E} = \frac{1}{2}\epsilon.$$

Exercise 12 *Does (2) hold if x is subnormal, i.e. $E = -126$ and $b_0 = 0$? If not, how big could δ be?*

Now another way to write the definition of δ is

$$\text{round}(x) = x(1 + \delta),$$

so we have the following result: the rounded value of a normalized number x is, when not exactly equal to x, equal to $x(1 + \delta)$, where, regardless of the rounding mode,

$$|\delta| < \epsilon.$$

Here, as before, ϵ is the machine precision. In the case of *round to nearest*, we have

$$|\delta| \leq \frac{1}{2}\epsilon.$$

This result is very important, because it shows that, no matter how x is displayed, for example either in binary format or in a converted decimal format, you can think of the value shown as *not exact*, but as *exact within a factor* of $1 + \epsilon$. Using Table 3 we see, for example, that IEEE single precision numbers are good to a factor of about $1 + 10^{-7}$, which means that they have about 7 accurate decimal digits.

Numbers are normally input to the computer using some kind of high-level programming language, to be processed by a compiler or an interpreter. There are two different ways that a number such as $1/10$ might be input. One way would be to input the decimal string 0.1 directly, either in the program itself or in the input to the program. The compiler or interpreter then calls a standard input-handling procedure which generates machine instructions to convert the decimal string to its binary representation and store the correctly rounded result in memory or a register. Alternatively, the integers 1 and 10 might be input to the program and the ratio 1/10 generated by a division operation. In this case too, the input-handling program must be called to read the integer strings 1 and 10 and convert them to binary representation. Either integer or floating point format might be used for storing these values in memory, depending on the type of the variables used in the program, but these values must be converted to floating point format before the division operation computes the ratio 1/10 and stores the final floating point result.

From the point of view of the underlying hardware, there are relatively few operations which can be done on floating point numbers. These include the standard arithmetic operations (add, subtract, multiply, divide) as well as a few others such as square root. When the computer performs such a floating point operation, the operands must be available in the processor registers or in memory. The operands are therefore, by definition, floating point numbers, even if they are only approximations to the original program data. However, the result of a standard operation on two floating point numbers may well *not* be a floating point number. For example, 1 and 10 are both floating point numbers but we have already seen that 1/10 is not. In fact, multiplication of two arbitrary 24-bit significands generally gives a 48-bit significand which cannot be represented exactly in single precision.

When the result of a floating point operation is not a floating point number, the IEEE standard requires that the computed result must be the *correctly rounded value of the exact result*, using the rounding mode and precision currently in effect. It is worth stating this requirement carefully. Let x and y be floating point numbers, let $+,-,*,/$ denote the four standard arithmetic operations, and let $\oplus,\ominus,\otimes,\oslash$ denote the corresponding operations as they are actually implemented on the computer. Thus, $x+y$ may not be a floating point number, but $x \oplus y$ is the floating point number which the computer computes as its approximation of $x+y$. The IEEE rule is then precisely:

$$x \oplus y = \mathrm{round}(x+y),$$

$$x \ominus y = \mathrm{round}(x-y),$$

$$x \otimes y = \mathrm{round}(x*y),$$

and

$$x \oslash y = \mathrm{round}(x/y).$$

From the discussion of relative rounding errors given above, we see then that the computed value $x \oplus y$ satisfies

$$x \oplus y = (x + y)(1 + \delta)$$

where

$$|\delta| \le \epsilon$$

for all rounding modes and

$$\delta \le \frac{1}{2}\epsilon$$

in the case of *round to nearest*. The same result also holds for the other operations $\ominus$, $\otimes$ and $\oslash$.

Exercise 13

- *Show that it follows from the IEEE rule for correctly rounded arithmetic that floating point addition is commutative, i.e.*

$$a \oplus b = b \oplus a$$

  for any two floating point numbers a and b.

- *Show with a simple example that floating point addition is* not *associative, i.e. it may not be true that*

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c$$

  for some floating point numbers a, b and c.

Now we ask the question: how is correctly rounded arithmetic implemented? Let us consider the addition of two floating point numbers $x = m \times 2^E$ and $y = p \times 2^F$, using IEEE single precision. If the two exponents E and F are the same, it is necessary only to add the significands m and p. The final result is $(m+p) \times 2^E$, which then needs further normalization if $m+p$ is 2 or larger, or less than 1. For example, the result of adding $3 = (1.100)_2 \times 2^1$ to $2 = (1.000)_2 \times 2^1$ is:

$$\begin{array}{cl} & (\ 1.10000000000000000000000\ )_2 \times 2^1 \\ + & (\ 1.00000000000000000000000\ )_2 \times 2^1 \\ = & (10.10000000000000000000000\ )_2 \times 2^1 \end{array}$$

and the significand of the sum is shifted right 1 bit to obtain the normalized format $(1.0100\ldots0)_2 \times 2^2$. However, if the two exponents E and F are different, say with $E > F$, the first step in adding the two numbers is to *align the significands*, shifting p right $E - F$ positions so that the second number is no longer normalized and both numbers have the same exponent E. The significands are then added as before. For example, adding $3 = (1.100)_2 \times 2^1$ to $3/4 = (1.100)_2 \times 2^{-1}$ gives:

$$\begin{array}{rl} & (\ 1.10000000000000000000000\ \ )_2 \times 2^1 \\ + & (\ 0.01100000000000000000000\ \ )_2 \times 2^1 \\ = & (\ 1.11100000000000000000000\ \ )_2 \times 2^1. \end{array}$$

In this case, the result does not need further normalization.

Now consider adding 3 to 3×2^{-23}. We get

$$\begin{array}{rl} & (\ 1.10000000000000000000000\ \ \ \ \ )_2 \times 2^1 \\ + & (\ 0.00000000000000000000001|1\ \ )_2 \times 2^1 \\ = & (\ 1.10000000000000000000001|1\ \ )_2 \times 2^1. \end{array}$$

This time, the result is not an IEEE single precision floating point number, since its significand has 24 bits after the binary point: the 24th is shown beyond the vertical bar. Therefore, the result must be *correctly rounded.* Rounding down gives the result $(1.10000000000000000000001)_2 \times 2^1$, while rounding up gives $(1.10000000000000000000010)_2 \times 2^1$. In the case of rounding to nearest, there is a tie, so the latter result, with the even final bit, is obtained.

For another example, consider the numbers 1, 2^{-15} and 2^{15}, which are all floating point numbers. The result of adding the first to the second is $(1.000000000000001)_2$, which is a floating point number; the result of adding the first to the third is $(1.000000000000001)_2 \times 2^{15}$, which is also a floating point number. However, the result of adding the second number to the third is

$$(1.000000000000000000000000000001)_2 \times 2^{15},$$

which is not a floating point number in the IEEE single precision format, since the fraction field would need 30 bits to represent this number exactly. In this example, using any of round towards zero, round to nearest, or round down as the rounding mode, the correctly rounded result is 2^{15}. If round up is used, the correctly rounded result is $(1.00000000000000000000001)_2 \times 2^{15}$.

Exercise 14 *In IEEE single precision, using* round to nearest*, what are the correctly rounded values for:* $64 + 2^{20}$*,* $64 + 2^{-20}$*,* $32 + 2^{-20}$*,* $16 + 2^{-20}$*,* $8 + 2^{-20}$*. Give the binary representations, not the decimal equivalent. What are the results if the rounding mode is changed to* round up*?*

Exercise 15 *Recalling how many decimal digits correspond to the 23 bit fraction in an IEEE single precision number, which of the following numbers do you think round exactly to the number 1, using* round to nearest*:* $1+10^{-5}$*,* $1+10^{-10}$*,* $1+10^{-15}$*?*

Although the operations of addition and subtraction are conceptually much simpler than multiplication and division and are much easier to carry out by hand, the implementation of correctly rounded floating point addition and subtraction is not trivial, even for the case when the result is a floating point number and therefore does not require rounding. For example, consider computing $x - y$ with $x = (1.0)_2 \times 2^0$ and $y = (1.1111\ldots1)_2 \times 2^{-1}$, where the fraction field for y contains 23 ones after the binary point. (Notice that y is only slightly smaller than x; in fact it is the next floating point number smaller than x.) Aligning the significands, we obtain:

$$\begin{array}{rl} & (\ 1.00000000000000000000000|\quad\ )_2 \times 2^0 \\ - & (\ 0.11111111111111111111111|1\ \ )_2 \times 2^0 \\ = & (\ 0.00000000000000000000000|1\ \ )_2 \times 2^0. \end{array}$$

This is an example of *cancellation*, since almost all the bits in the two numbers cancel each other. The result is $(1.0)_2 \times 2^{-24}$, which is a floating point number, but in order to obtain this correct result we must be sure to *carry out the subtraction using an extra bit*, called a *guard bit*, which is shown after the vertical line following the b_{23} position. When the IBM 360 was first released, it did not have a guard bit, and it was only after the strenuous objections of certain computer scientists that later versions of the machine incorporated a guard bit. Twenty-five years later, the Cray supercomputer still does not have a guard bit. When the operation just illustrated, modified to reflect the Cray's longer wordlength, is performed on a Cray XMP, the result generated is wrong by a factor of two since a one is shifted past the end of the second operand's significand and discarded. In this example, instead of having

$$x \ominus y = (x - y)(1 + \delta), \qquad \text{where } \delta \le \epsilon, \tag{3}$$

we have $x \ominus y = 2(x - y)$. On a Cray YMP, on the other hand, the second operand is rounded before the operation takes place. This converts the second operand to the value 1.0 and causes a final result of 0.0 to be computed, i.e. $x \ominus y = 0$. Evidently, Cray supercomputers do not use correctly rounded arithmetic.

Machines supporting the IEEE standard do, however, have correctly rounded arithmetic, so that (3), for example, always holds. Exactly how this is implemented depends on the machine, but typically floating point operations are carried out using *extended precision registers*, e.g. 80-bit registers, even if the values loaded from and stored to memory are only single or double precision. This effectively provides many guard bits for single and double precision operations, but if an extended precision operation on extended precision operands is desired, at least one additional guard bit is needed. In fact, the following example (given in single precision for convenience) shows that one, two or even 24 guard bits are not enough to guarantee correctly rounded addition with 24-bit significands when the rounding mode is round to nearest. Consider computing $x - y$ where $x = 1.0$ and $y = (1.000\ldots01)_2 \times 2^{-25}$, where y has 22 zero bits between the binary point and the final one bit. In exact arithmetic, which requires 25 guard bits in this case, we get:

$$\begin{array}{rll} & (1.00000000000000000000000| &)_2 \times 2^0 \\ - & (0.00000000000000000000000|0100000000000000000000001 &)_2 \times 2^0 \\ = & (0.11111111111111111111111|1011111111111111111111111 &)_2 \times 2^0 \end{array}$$

Normalizing the result, and then rounding this to the nearest floating point number, we get $(1.111\ldots1)_2 \times 2^{-1}$, which is the correctly rounded value of the exact sum of the numbers. However, if we were to use only two guard bits (or indeed any number from 2 to 24), we would get the result:

$$\begin{array}{rll} & (1.00000000000000000000000| &)_2 \times 2^0 \\ - & (0.00000000000000000000000|01 &)_2 \times 2^0 \\ = & (0.11111111111111111111111|11 &)_2 \times 2^0 \end{array}$$

Normalizing and rounding then results in rounding up instead of down, giving the final result 1.0, which is *not* the correctly rounded value of the exact sum. Machines that implement correctly rounded arithmetic take such possibilities into account, and it turns out that correctly rounded results can be achieved in all cases using only two guard bits together with an extra bit, called a sticky bit, which is used to flag a rounding problem of this kind.

Floating point multiplication, unlike addition and subtraction, does not require significands to be aligned. If $x = m \times 2^E$ and $y = p \times 2^F$, then

$$x \times y = (m \times p) \times 2^{E+F}$$

so there are three steps to floating point multiplication: multiply the significands, add the exponents, and normalize and correctly round the result.

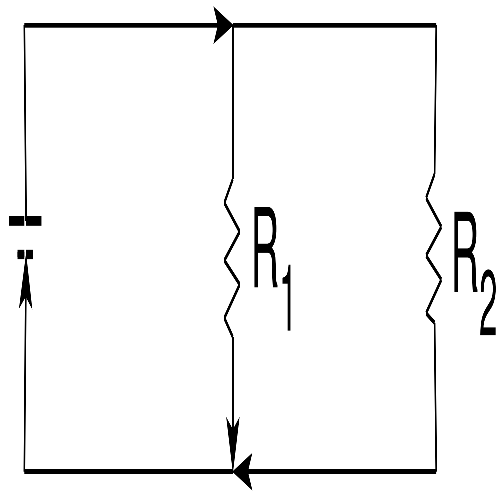
Single precision significands are easily multiplied in an extended precision register, since the product of two 24-bit significand bitstrings is a 48-bit bitstring which is then correctly rounded to 24 bits after normalization. Multiplication of double precision or extended precision significands is not so straightforward, however, since dropping bits may, as in addition, lead to incorrectly rounded results. The actual multiplication and division implementation may or may not use the standard "long multiplication" and "long division" algorithms which we learned in school. One alternative for division will be mentioned in a later section. Multiplication and division are much more complicated operations than addition and subtraction, and consequently they may require more execution time on the computer; this is the case for PC's. However, this is not necessarily true on faster supercomputers since it is possible, by using enough space on the chip, to design the microcode for the instructions so that they are all equally fast.

Exercise 16 *Assume that x and y are normalized numbers, i.e. $1 \le m < 2$, $1 \le p < 2$. How many bits may it be necessary to shift the significand product $m \times p$ left or right to normalize the result?*

4 Exceptional Situations

One of the most difficult things about programming is the need to anticipate exceptional situations. In as much as it is possible to do so, a program should handle exceptional data in a manner consistent with the handling of standard data. For example, a program which reads integers from an input file and echos them to an output file until the end of the input file is reached should not fail just because the input file is empty. On the other hand, if it is further required to compute the average value of the input data, no reasonable solution is available if the input file is empty. So it is with floating point arithmetic. When a reasonable response to exceptional data is possible, it should be used.

The simplest example is *division by zero*. Before the IEEE standard was devised, there were two standard responses to division of a positive number by zero. One often used in the 1950's was to generate the largest floating point number as the result. The rationale offered by the manufacturers was that the user would notice the large number in the output and draw the conclusion that something had gone wrong. However, this often led to total disaster: for example the expression $1/0 - 1/0$ would then have a result of 0, which is completely meaningless; furthermore, as the value is not large, the user might *not* notice that any error had taken place. Consequently, it was emphasized in the 1960's that division by zero should lead to the generation of a program interrupt, giving the user an informative message such as "fatal error — division by zero". In order to avoid this abnormal termination, the burden was on the programmer to make sure that division by zero would never occur. Suppose, for example, it is desired to compute the total resistance in an electrical circuit with two resistors connected in parallel, with resistances respectively R_1 and R_2 ohms, as shown in Figure 4.



The standard formula for the total resistance of the circuit is

$$T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}.$$

This formula makes intuitive sense: if both resistances R_1 and R_2 are the same value R, then the resistance of the whole circuit is $T = R/2$, since the current divides equally, with equal amounts flowing through each resistor. On the other hand, if one of the resistances, say R_1, is very much smaller than the other, the resistance of the whole circuit is almost equal to that very small value, since most of the current will flow through that resistor and avoid the other one. What if R_1 is zero? The answer is intuitively clear: if one resistor offers no resistance to the current, *all* the current will flow through that resistor and avoid the other one; therefore, the total resistance in the circuit is zero. The formula for T also makes perfect sense mathematically; we get

$$T = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0.$$

Why, then, should a programmer writing code for the evaluation of parallel resistance formulas have to worry about treating division by zero as an exceptional situation? In IEEE floating point arithmetic, the programmer is relieved from that burden. As long as the initial floating point environment is set properly (as explained below), division by zero does not generate an interrupt but gives an infinite result, program execution continuing normally. In the case of the parallel resistance formula this leads to a final correct result of $1/\infty = 0$.

It is, of course, true that $a * 0$ has the value 0 for any *finite* value of a. Similarly, we can adopt the convention that $a/0 = \infty$ for any *positive* value of a. Multiplication with ∞ also makes sense: $a \times \infty$ has the value ∞ for any *positive* value of a. But the expressions $\infty * 0$ and $0/0$ make no mathematical sense. An attempt to compute either of these quantities is called an *invalid operation*, and the IEEE standard calls for the result of such an operation to be set to NaN (Not a Number). Any arithmetic operation on a NaN gives a NaN result, so any subsequent computation with expressions which have a NaN value are themselves assigned a NaN value. When a NaN is discovered in the output of a program, the programmer knows something has gone wrong and can invoke debugging tools to determine what the problem is. This may be assisted by the fact that the bitstring in the fraction field can be used to code the origin of the NaN. Consequently, we do not speak of a unique NaN value but of many possible NaN values. Note that an ∞ in the output of a program may or may not indicate a programming error, depending on the context.

Addition and subtraction with ∞ also make mathematical sense. In the parallel resistance example, we see that $\infty + \frac{1}{R_2} = \infty$. This is true even if R_2 also happens to be zero, because $\infty + \infty = \infty$. We also have $a + \infty = \infty$ and $a - \infty = -\infty$ for any *finite* value of a. But there is no way to make sense of the expression $\infty - \infty$, which must therefore have a NaN value. (These observations can be justified mathematically by considering addition of limits. Suppose there are two sequences x_k and y_k both diverging to ∞, e.g. $x_k = 2^k$, $y_k = 2k$, for $k = 1, 2, 3, \ldots$. Clearly, the sequence $x_k + y_k$ must also diverge to ∞. This justifies the expression $\infty + \infty = \infty$. But it is impossible to make a statement about the limit of $x_k - y_k$, without knowing more than the fact that they both diverge to ∞, since the result depends on which of a_k or b_k diverges faster to ∞.)

Exercise 17 *What are the possible values for*

$$\frac{1}{a} - \frac{1}{b}$$

where a and b are both nonnegative (positive or zero)?

Exercise 18 *What are sensible values for the expressions $\infty/0$, $0/\infty$ and ∞/∞?*

Exercise 19 *Using the 1950's convention for treatment of division by zero mentioned above, the expression* $(1/0)/10000000)$ *results in a number very much smaller than the largest floating point number. What is the result in IEEE arithmetic?*

The reader may very reasonably ask the following question: why should $1/0$ have the value ∞ rather than $-\infty$? This is the main reason for the existence of the value -0, so that the conventions $a/0 = \infty$ and $a/(-0) = -\infty$ may be followed, where a is a positive number. (The reverse holds if a is negative.) It is essential, however, that the logical expression $\langle 0 = -0 \rangle$ have the value **true** while $\langle \infty = -\infty \rangle$ have the value **false**. Thus we see that it is possible that the logical expressions $\langle a = b \rangle$ and $\langle 1/a = 1/b \rangle$ have different values, namely in the case $a = 0$, $b = -0$ (or $a = \infty$, $b = -\infty$). This phenomenon is a direct consequence of the convention for handling infinity.

Exercise 20 *What are the values of the expressions $0/(-0)$, $\infty/(-\infty)$ and $-\infty/(-0)$?*

Exercise 21 *What is the result of the parallel resistance formula if an input value is negative, -0, or* NaN*?*

Another perhaps unexpected consequence of these conventions concerns arithmetic comparisons. When a and b are real numbers, one of three conditions holds: $a = b$, $a < b$ or $a > b$. The same is true if a and b are floating point numbers in the conventional sense, even if the values $\pm\infty$ are permitted. However, if either a or b has a NaN value, none of the three conditions can be said to hold (even if both a and b have NaN values). Instead, a and b are said to be *unordered*. Consequently, although the logical expressions $\langle a \leq b \rangle$

and $\langle \text{not}(a > b) \rangle$ usually have the same value, they have different values (the first **false**, the second **true**) if either a or b is a NaN.

Let us now turn our attention to overflow and underflow. *Overflow* is said to occur when the true result of an arithmetic operation is finite but larger in magnitude than the largest floating point number which can be stored using the given precision. As with division by zero, there were two standard treatments before IEEE arithmetic: either set the result to (plus or minus) the largest floating point number, or interrupt the program with an error message. In IEEE arithmetic, the standard response depends on the rounding mode. Suppose that the overflowed value is positive. Then *round up* gives the result ∞, while *round down* and *round towards zero* set the result to the largest floating point number. In the case of *round to nearest*, the result is ∞. From a strictly mathematical point of view, this is not consistent with the definition for non-overflowed values, since a finite overflow value cannot be said to be closer to ∞ than to some other finite number. From a practical point of view, however, the choice ∞ is important, since *round to nearest* is the default rounding mode and any other choice may lead to very misleading final computational results.

Underflow is said to occur when the true result of an arithmetic operation is smaller in magnitude than the smallest normalized floating point number which can be stored. Historically, the response to this was almost always the same: replace the result by zero. In IEEE arithmetic, the result may be a subnormal positive number instead of zero. This allows results much smaller than the smallest normalized number to be stored, closing the gap between the normalized numbers and zero as illustrated earlier. However, it also allows the possibility of *loss of accuracy*, as subnormal numbers have fewer bits of precision than normalized numbers.

Exercise 22 *Work out the sensible rounding conventions for underflow. For example, using* round to nearest*, what values are rounded down to zero and what values are rounded up to the smallest subnormal number?*

Exercise 23 *More often than not the result of ∞ following division by zero indicates a programming problem. Given two numbers a and b, consider setting*

$$c = \frac{a}{\sqrt{a^2+b^2}}, \quad d = \frac{b}{\sqrt{a^2+b^2}}$$

Is it possible that c or d (or both) is set to the value ∞, even if a and b are

Table 4: IEEE Standard Response to Exceptions

Invalid Operation	Set result to NaN
Division by Zero	Set result to $\pm\infty$
Overflow	Set result to $\pm\infty$ or largest f.p. number
Underflow	Set result to zero or subnormal number
Precision	Set result to correctly rounded value

normalized numbers? [5]*.*

Exercise 24 *Consider the computation of the previous exercise again. Is it possible that d and e could have* many *less digits of accuracy than a and b, even though a and b are normalized numbers?*

Altogether, the IEEE standard defines five kinds of exceptions: invalid operation, division by zero, overflow, underflow and precision, together with a *standard response* for each of these. All of these have just been described except the last. The last exception is, in fact, not exceptional at all because it occurs every time the result of an arithmetic operation is not a floating point number and therefore requires rounding. Table 4 summarizes the standard response for the five exceptions.

The IEEE standard specifies that when an exception occurs it must be *signaled* by setting an associated *status flag*, and that the programmer should have the option of either *trapping the exception*, providing special code to be executed when the exception occurs, or *masking the exception*, in which case the program continues executing with the standard response shown in the table. If the user is not programming in assembly language, but in a higher-level language being processed by an interpreter or a compiler, the ability to trap exceptions may or may not be passed on to the programmer. However, users rarely needs to trap exceptions in this manner. It is usually better to mask all exceptions and rely on the standard responses described in the table. Again, in the case of higher-level languages the interpreter or compiler in use may or may not mask the exceptions as its default action.

[5] A better way to make this computation may be found in the LAPACK routine `slartg`, which can be obtained by sending the message "slartg from lapack" to the Internet address netlib@ornl.gov