Floating Point Representation
and the IEEE Standard

Michael L. Overton

copyright ©1997

1 Introduction

Numerical computing means *computing with numbers*, and the subject is almost as old as civilization itself. Ancient peoples knew quite sophisticated techniques to solve many numerical tasks, of which perhaps the most impressive was the prediction of astronomical events such as eclipses. The abacus was used for centuries as an aid to calculation; this originated as a table with counting stones in the middle east and evolved into the wooden devices with beads on wires which are still common in the far east. Modern numerical computing began with Isaac Newton in England in the seventeenth century; his invention of calculus was in large part driven by its usefulness in solving numerical problems. In Newton's footsteps followed Euler, Lagrange, Gauss and many other great mathematicians of the 18th and 19th centuries. Up until that time, calculation was primarily done with pencil and paper in the west and the abacus in the east. In the first half of the 20th century the slide rule became supreme, and several generations of engineers used this clever invention which relied on a simple idea: addition is easy; multiplication is time-consuming; but multiplication can be done by taking logarithms, adding these together, and exponentiating the result. The slide rule makes this easy, though only to about three accurate digits, as the numbers are represented on the slide rule explicitly in a logarithmic scale. During World War II, scientific laboratories had rooms full of people doing different parts of a complicated calculation, using slide rules and mechanical calculators. Indeed, these may be thought of as the early days of parallel computing before the powerful electronic digital computer became available for scientific research in the 1950's.

Electronic computers were first invented in the 1940's and 1950's for exactly one purpose: solving hard scientific and engineering problems which

required a great deal of numerical computing. This was the motivation for the first operating computers build by Zuse in Germany in the early 1940's; these machines used electromechanical switching devices. (The machines were developed too late to have much impact on the war effort, though they were used to model rocket trajectories.) The first electronic computer built using vacuum tube technology was the ENIAC (Electronic Numerical Integrator and Calculator) at the University of Pennsylvania, shortly after World War II. During the 1950's, the primary usage of computers was for numerical computing in scientific applications. In the 1960's, computers became widely used by large businesses, but their purpose was not primarily numerical; rather, the principal use of computers became the processing of all kinds of information. Some of this information, such as accounting, involved numbers, but the numbers involved were all integers: numbers of dollars and cents. Other information, such as character strings, could be regarded as equivalent to numbers and indeed was represented in the computer as binary numbers. But the primary business applications were not numerical in nature. During the next two decades, computers became ever more widespread, to medium-sized businesses in the 1970's and to many millions of small businesses and individuals during the PC revolution of the 1980's. The vast majority of these computer users do not see computing with numbers as their primary interest; rather, they are typically interested in the processing of information: up until the mid-1980's, usually textual, but increasingly also aural and visual.

However, in most scientific disciplines, computing with numbers remains by far the most important use of computers. Physicists use computers to solve complicated equations modeling everything from the expansion of the universe to the microstructure of the atom, and to test their theories against huge quantities of experimental data. Chemists and biologists use computers to determine the molecular structure of proteins, and a major project now underway is the modeling of the human genome. Medical researchers use computers to design new and better medical techniques, for example modeling the flow of blood in the heart to understand how to design better artificial heart valves. Atmospheric scientists use numerical computing to process huge quantities of data and solve appropriate equations to predict the weather, typically producing five day forecasts which are quite reliable. Aeronautical engineers use computers to design better airplanes: the Boeing 757, 767 and 777 were designed relying more heavily on computer modeling than on the older method of wind tunnel tests. The design of the space shuttle was heavily dependent on numerical computational testing. Ironically, the tragic 1986 Challenger accident was more due to political errors than scientific ones. From
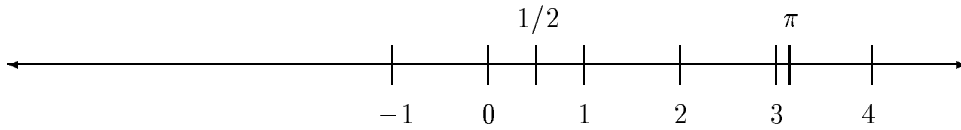
Figure 1: The Real Line

a scientific point of view, re-entry of the space shuttle into the atmosphere was a far more delicate and difficult procedure than lift-off, and many nervous scientists were elated and relieved to see that their calculations had worked so well when the space shuttle first re-entered the atmosphere and landed. A similar story is often told about the engineer who designed Carnegie Hall a hundred years earlier and nervously rechecked his numerical calculations during the first performance. Virtually all branches of engineering rely heavily on numerical computing.

2   The Real Numbers

Since we will mainly be computing with real numbers, it is important to understand the different types of real number. The real numbers can conveniently be represented by a line. Every point on the line corresponds to a real number, but only a few are marked in Figure 1.

The line stretches infinitely far in both directions, towards ∞ and $-\infty$. The real numbers may be divided into two classes: the rational numbers and the irrational numbers. The rational numbers are further divided into two subclasses: the integers and the non-integral fractions. The *integers* may be listed as follows: $0, 1, -1, 2, -2, 3, -3, \cdots$. Thus we say that there are an *infinite* but *countable* number of integers; by this we mean that every integer will eventually appear in the list if we count for long enough, even though we can never count all of them. The *rational* numbers are all those which consist of a ratio of two integers, e.g. $1/2$, $2/3$, $6/3$, etc.; some of these, e.g. $6/3$, are in fact integers. Those which are not integers are called non-integral fractions, or fractions for short. To see that the number of rational numbers is countable, imagine them all listed in an infinite two-dimensional array as follows:

	1	2	3	4	...
1	1/1	1/2	1/3	1/4	...
2	2/1	2/2	2/3	2/4	...
3	3/1	3/2	3/3	3/4	...
4	4/1	4/2	4/3	4/4	...
...	...	...	...	...	...

Listing the first line and then the second, etc., does not work, since the first line never terminates. Instead, we generate a list of all rational numbers diagonal by diagonal: first $1/1$; then $2/1, 1/2$; then $3/1, 2/2, 1/3$; then $4/1, 3/2, 2/3, 1/4$; etc. In this way, every rational number (including every integer) is eventually generated. In fact, every number is generated many times (e.g. $1/2$ and $2/4$ are the same number). However, every rational number does have a unique representation in lowest terms, that is canceling any common factor in the numerator and denominator (thus $2/4$ reduces to $1/2$).

Most real numbers are not rational, i.e. there is no way of writing them as the ratio of two integers. These numbers are called *irrational*. Familiar examples of irrational numbers are $\sqrt{2}$, π and e. The first two of these numbers were known to ancient peoples such as the Babylonians and Greeks. The square root of a positive number a is defined to be the positive number x satisfying

$$x^2 = a$$

while π is defined as the ratio of a circle's circumference to its diameter. The third number mentioned, e, or Euler's number, is defined to be the limit of

$$(1 + \frac{1}{n})^n$$

as $n \to \infty$. Irrational numbers can always be defined as limits of sequences of rational numbers, but there is no way of listing all the irrational numbers: the set of irrational numbers is said to be *uncountable*.

What is the most convenient way to represent numbers? It is interesting to note that the positional system we use today was not standard in ancient times; for example, the Romans used a system where each power of 10 required a different symbol: X for 10, C for 100, M for 1000, etc. Large numbers cannot conveniently be represented by such a system. The positional notation we use today requires a key idea: the representation of zero by a symbol. Such a system was developed in India and used widely in the middle east before being passed on to Europe by the Arabs about a thousand years

ago, giving rise to the name Arabic numerals. This decimal, or base 10, system requires 10 symbols, representing the numbers zero through nine. The system is called *positional* because the meaning of the number is understood from the position of the symbols. Zero is needed, for example, to distinguish the number 601 from the number 61.[1] The reason for the decimal choice is the simple biological fact that humans have ten fingers and thumbs. Other positional systems developed by ancient peoples were a base 20 system developed by the Mayans, which was used for very sophisticated astronomical calculations, and a base 60 system used by the Babylonians, whose vestiges are still seen today in our division of the hour into 60 minutes and the minute into 60 seconds. Although decimal representation is convenient for people, it is not particularly convenient for computer purposes. The binary, or base 2, representation system is much more convenient: in this system, every number is represented as a string of 0's and 1's.

Every real number has a decimal representation and a binary representation (and, indeed, a representation in a base equal to any other integer greater than one). The representation of integers is simple, requiring an expansion in nonnegative powers of the base; for example here is a decimal number:

$$(71)_{10} = 7 \times 10 + 1$$

and its binary equivalent:

$$(1000111)_2 = 1 \times 64 + 0 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1.$$

Non-integral fractions have entries to the right of the decimal (and binary) point, and these entries may be either finite or infinite. For example, the fraction 11/2 has the representations

$$\frac{11}{2} = (5.5)_{10} = 5 \times 1 + 5 \times \frac{1}{10}$$

and

$$\frac{11}{2} = (101.1)_2 = 1 \times 4 + 0 \times 2 + 1 \times 1 + 1 \times \frac{1}{2}.$$

[1] Although the need for a zero might seem obvious, it was not present in the ancient cultures. It was not used by the Babylonians until about the 3rd century B.C., and it seems likely that the Indian zero was derived later from the Babylonian one. The Chinese did not use a zero until much later still, following the Indian influence. The only well documented use of zero known to be independent of the Babylonians is the Mayan system.

Both of these are finite. However, the fraction 1/10, while obviously having the finite decimal expansion $(0.1)_{10}$, has the binary representation

$$\frac{1}{10} = (0.0001100110011\ldots)_2 = \frac{1}{16} + \frac{1}{32} + \frac{0}{64} + \frac{0}{128} + \frac{1}{256} + \frac{1}{512} + \frac{0}{1024} + \cdots.$$

Note that this representation, while infinite, is *repeating*. The fraction 1/3 has infinite representations in both binary and decimal:

$$1/3 = (0.333\ldots)_{10} = (0.010101\ldots)_2.$$

If the representation of a rational number is infinite, it must be repeating. For example,

$$1/7 = (0.142857142857\ldots)_{10}$$

Irrational numbers always have infinite, non-repeating expansions. For example:

$$\sqrt{2} = (1.414213\ldots)_{10}, \ \ \pi = (3.141592\ldots)_{10}, \ \ e = (2.71828182845\ldots)_{10}.$$

Looking at just the first 8 digits of e, you might think its representation is repeating, but in fact it is not.

Exercise 1 *Determine the binary representations of some integers and fractions, and then convert these back to decimal again as a check. Since $(1001.11)_2$ is just the decimal number*

$$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2},$$

we see it is easy to convert from binary to decimal. For decimal to binary we can convert the integer and fractional parts separately. For example if x is an integer written in decimal, we wish to find coefficients $a_0, a_1, \ldots, a_n$, all 0 or 1, so that

$$a_n \times 2^n + a_{n-1} \times 2^{n-1} + \cdots + a_0 \times 2^0 = x,$$

giving the two representations $(a_n a_{n-1} \cdots a_0)_2 = (x)_{10}$. Clearly dividing x by 2 gives a_0 as remainder, leaving as quotient

$$a_n \times 2^{n-1} + a_{n-1} \times 2^{n-2} + \cdots + a_1 \times 2^0,$$

and so we can continue to find a_1 then a_2 etc. Work out a similar approach for decimal fractions. When working with binary numbers, it is often convenient to write them in octal notation, that is base 8, using the symbols 0 through 7 to write the binary strings 000 through 111. Alternatively, use the hexadecimal system, i.e. base 16, using the symbols 0,...,9,A,...,F to represent the binary strings 0000 through 1111.

3   Computer Representation of Numbers

What is the best way to represent a real number in the computer? In the case of the nonnegative integers, the answer is easy: as a bitstring giving the binary representation of the integer. Thus, the integer 71 would be represented as the bitstring

$$\boxed{000\ldots 01000111}.$$

Suppose that the computer word size is 32 bits. The number 2^{32} is too big to be stored in such a word using the straightforward bitstring representation, since its binary representation consists of a one followed by 32 zeros. The next smallest integer, i.e. $2^{32} - 1$, whose binary representation consists of 32 ones, is the largest integer which will fit.

How should we represent negative integers? The most obvious suggestion is *sign-and-modulus*: store the magnitude of the integer as just discussed and use an extra bit to represent the sign. Since 32 bits is the standard computer word size, the use of one bit for the sign restricts the allowable magnitude to $2^{31} - 1$ instead of $2^{32} - 1$. However, there is a more convenient representation called *2's complement*, used by most machines. The nonnegative integers between 0 and $2^{31} - 1$ inclusive are stored in the same way as before, but a negative integer $-x$, where x has a value between 1 and 2^{31} inclusive, is stored instead as the positive integer $2^{32} - x$, which must then have a value between 2^{31} and $2^{32} - 1$. For example, the integer -71 would be stored as the bitstring

$$\boxed{111\ldots 10111001}.$$

That this is correct can be verified by adding the bitstring for 71 to it and seeing that the sum is 2^{32}. In general since $2^{32} - x = (2^{32} - 1 - x) + 1$, and $2^{32} - 1$ is all 1s, converting the binary representation of x to the 2's complement representation $2^{32} - x$ of $-x$ requires simply changing all zero bits to ones, one bits to zero and adding one.

To see one advantage of 2's complement representation, suppose we want to do the operation $y + (-x)$, where x and y are both nonnegative numbers between 0 and $2^{31} - 1$. Note that

$$y - x = -(x - y),$$

and the 2's complement representations of y and $-x$ are the nonnegative numbers y and $2^{32} - x$. If we *add* these representations, we obtain

$$2^{32} + y - x = 2^{32} - (x - y).$$

7

If $y \geq x$, the left hand side will not fit in a 32-bit word, and the leading bit can be dropped over the left end of the word, giving the correct result, $y - x$. If $y < x$, the right hand side shows we already have the correct result, since it represents the negative value $-(x - y)$ in the correct way. Thus, *no special hardware is needed for integer subtraction.* The addition hardware can be used once the negative number $-x$ has been represented using 2's complement.

There is a third system for the representation of negative integers called 1's complement. This is similar to 2's complement, except that a negative integer $-x$ is stored as $2^{32} - x - 1$. This system was used on the CDC computers which dominated supercomputing in the 1960's and 1970's but is now obsolete.

Exercise 2 *Try some examples until you are convinced that this discussion is correct. Use a 4 bit word instead of a 32 bit word to keep things simple.*

Exercise 3 *For a given wordlength,*

 (a) *Which of these 3 systems can represent the most integers?*

 (b) *For which of these 3 systems is there only one possible representation for zero?*

 (c) *What is the quickest way of deciding if a number is negative or non-negative using 2's complement representation?*

How should non-integral real numbers be represented? Rational numbers could be represented by a list of two integers, the numerator and denominator. This has the advantage of accuracy, but the disadvantage of being very inconvenient for arithmetic. Systems which represent rational numbers in this way are said to be *symbolic* rather that numeric. For most numerical computing purposes however, real numbers, whether rational or irrational, are approximately stored using the binary representation of the number. There are two possible methods, called fixed point and floating point.

In *fixed point* representation, the computer word is divided into three fields, one one-bit field for the sign of the number, one field of bits for the binary representation of the number before the binary point, and one field for the binary representation after the binary point. For example, in a 32-bit word with field widths of 1,15 and 16, the number 11/2 would be stored as:

0	000000000000101	1000000000000000

while the number 1/10 would be stored as

| 0 | 000000000000000 | 0001100110011001 |

The fixed point system has a severe limitation on the size of the numbers to be stored, however. In the example just given, only numbers ranging in size from (exactly) 2^{-16} to (slightly less than) 2^{15} could be stored. This is not adequate for general-purpose scientific applications, where one may need to use very large or very small numbers, e.g. representing the age of the earth or the mass of the electron. Therefore, fixed point representation is rarely used for *general* numerical computing and will not be discussed further.

Floating point representation solves this problem by using a system closely related to well known *exponential notation* (also known as *scientific notation*). In (normalized) exponential notation, a nonzero real number is written as

$$\pm m \times 10^E, \quad \text{where } 1 \le m < 10,$$

and E is an integer. The numbers m and E are called the *significand* and the *exponent* respectively. For example, the number 365.25 is represented as 3.625×10^2, and the number 0.00036525 is represented as 3.6525×10^{-4}. For representation on the computer, we prefer a base which is binary, not decimal, so we write a nonzero number x as

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

This is always possible, as m can be obtained from x by repeatedly multiplying or dividing by 2 until the result is less than 2 but greater than or equal to one, changing the exponent E accordingly. Consequently, the binary expansion for m is

$$m = (b_0.b_1b_2b_3\ldots)_2, \quad \text{with } b_0 = 1.$$

To *store* such numbers, consider dividing the computer word into 3 fields, to represent 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 (if 2's complement representation is used). The significand field can store the first 23 bits of the binary representation of m, namely

$$b_0.b_1\ldots b_{22}, \quad \text{(if we store } b_0).$$

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 *(computer) 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	$E = 2$	1.01100000000000000000000

,

and the number

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

would be represented by

0	$E = 6$	1.00011100000000000000000

.

To avoid confusion, the exponent E, which is actually stored in a binary representation, is shown in decimal for the moment. The binary point between b_0 and b_1 is shown for convenience, but is not stored. The representation for negative numbers is obtained by changing the sign bit in the first field from 0 to 1. This sign-and-modulus convention is standard for floating point significands, in contrast to integer representation. Now consider the much larger number

$$2^{71} = (1.000)_2 \times 2^{71}.$$

This number, although an integer, is much too large to store in a 32-bit word using the standard integer format. However, there is no problem representing it in floating point, using the representation

0	$E = 71$	1.00000000000000000000000

.

Exercise 4 *What is the largest floating point number in this system, i.e. where the significand field can store only the bits $b_0.b_1 \ldots b_{22}$ and the exponent is limited by $-128 \leq E \leq 127$?*

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

| 0 | $E = 4$ | 0.0101100000000000000000000 |

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 some-where. (An alternative, namely *rounding*, is discussed later.) The simplest way to truncate the expansion to 23 bits would give the representation

| 0 | $E = 0$ | 0.00011001100110011001100110 |

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

| 0 | $E = 4$ | 0.00000001100110011001100110 |

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

| 0 | $E = -4$ | 1.10011001100110011001100110 |

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. $2 > m \geq 1$. 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

0	$E = 0$	0.0000000000000000000000

.

The gap between the number 1 and the next largest floating point number is called the *precision* of the floating point system, [2] or, often, the *machine precision*, and we shall denote this by ϵ. In the system just described, the next floating point number bigger than 1 is

$$b_0.b_1 \ldots b_{22} = 1.0000000000000000000001,$$

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

Exercise 5 *What is the smallest possible positive normalized floating point number using the system just described?*

Exercise 6 *Could nonzero numbers instead be normalized so that $\frac{1}{2} \leq m < 1$? 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 number system*. The set of (normalized) toy floating point numbers is shown in Figure 2.

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 magnitudes of the numbers themselves get smaller, and *bigger* as the numbers get bigger. Specifically, consider the positive floating point numbers with $E = 0$: these are just the numbers 1, 1.25, 1.5 and 1.75. For each of these numbers, say x, the gap between x and the next floating point number larger than x is $\epsilon = 0.25$. Then

---

[2] 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.
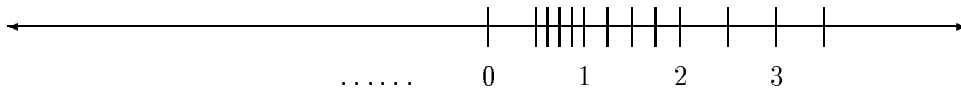
Figure 2: The Toy Floating Point Numbers

look at the positive floating point numbers x with $E = 1$: we see that the gap is twice as big, i.e. 2ϵ. Then look at the positive floating point numbers with $E = -1$: we see that the gap is $\frac{1}{2}\epsilon$. In general, we see that the gap between a floating point number $(b_0.b_1b_2)_2 \times 2^E$ and the next bigger floating point number is

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

Another important observation to make about Figure 2 is 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".

4 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 roughly similar to the one described in the last section, 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

13

and Electronics Engineers. [3] (There is also a decimal version of the standard but we shall not discuss this.)

The IEEE standard has three very important requirements:

- 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.

$$m = \pm (b_0.b_1 b_2 b_3 \ldots)_2,$$

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 before the IEEE standard was developed.

Exercise 7 *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.

---

[3] ANSI/IEEE Std 754-1985

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 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	00000000000000000000000

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*. [4] Let us postpone the discussion of subnormal numbers for the moment and go on to the other lines of the table.

---

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

Table 1: IEEE Single Precision

| $\pm$ | $a_1 a_2 a_3 \ldots a_8$ | $b_1 b_2 b_3 \ldots b_{23}$ |

If exponent bitstring $a_1 \ldots a_8$ is	Then numerical value represented is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{-126}$
$(00000001)_2 = (1)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = (2)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{-125}$
$(00000011)_2 = (3)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{-124}$
$\downarrow$	$\downarrow$
$(01111111)_2 = (127)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^0$
$(10000000)_2 = (128)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^1$
$\downarrow$	$\downarrow$
$(11111100)_2 = (252)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{125}$
$(11111101)_2 = (253)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{126}$
$(11111110)_2 = (254)_{10}$	$\pm(1.b_1 b_2 b_3 \ldots b_{23})_2 \times 2^{127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1 = \ldots = b_{23} = 0$, NaN otherwise

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 $a_1 a_2 a_3 \ldots a_8$ 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 $1 = (1.000 \ldots 0)_2 \times 2^0$ 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 | 01100000000000000000000 |

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 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 2, and see how the addition of subnormal numbers changes it. We
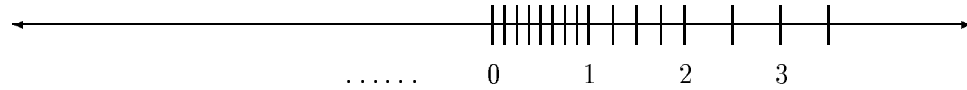
Figure 3: The Toy System including Subnormal Numbers

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 3. 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 8 *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 9 *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 10 *What is the smallest power of 2 which is not a single precision floating point number?*

Exercise 11 *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*

Table 2: IEEE Double Precision

$\pm$	$a_1 a_2 a_3 \dots a_{11}$	$b_1 b_2 b_3 \dots b_{52}$

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

number is

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

(It may be helpful to recall the discussion following Figure 2.)

Exercise 12 *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.*

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 $b_1, \dots, b_{52}$ can be stored instead of just $b_1, \dots, b_{23}$.

There is a third IEEE floating point format called extended precision. Although the standard does not require a particular format for this, the stan-

Table 3: What is that Precision?

IEEE Single	$\epsilon = 2^{-23} \approx 1.2 \times 10^{-7}$
IEEE Double	$\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$
IEEE Extended	$\epsilon = 2^{-63} \approx 1.1 \times 10^{-19}$

dard 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 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 field of a single precision normalized number has *exactly* 23 *bits of precision*, i.e. the significand has 24 bits counting the hidden bit. This corresponds to *approximately* 7 accurate decimal digits, since

$$2^{-24} \approx 10^{-7}.$$

In double precision, the fraction has *exactly* 52 *bits of precision*, i.e. the significand has 53 bits counting the hidden bit. This corresponds to *approximately* 16 *accurate decimal digits*, since

$$2^{-53} \approx 10^{-16}.$$

In extended precision, the significand has *exactly* 64 *bits of precision*, and this corresponds to *approximately* 19 *accurate decimal digits*. So, for example, the single precision representation for the number π is *approximately* 3.14159, with about 7 accurate digits. The double precision representation of π is *approximately* 3.14159265358979, with about 16 accurate digits. (Since a 24 bit binary number cannot be converted exactly to a 7 digit number, the actual decimal value for the binary representation of π has more than 7 nonzero digits, but only the first 7 digits are accurate approximations to π.)
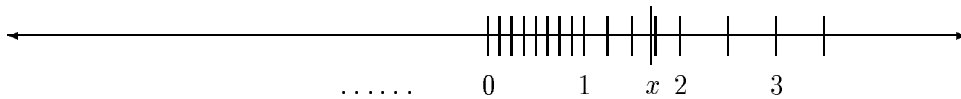
Figure 4: Rounding in the Toy System

5 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. The set of floating point numbers is a finite subset of the real numbers. 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 2 and 3. If $x = 1.7$, for example, then we have $x_- = 1.5$ and $x_+ = 1.75$, as shown in Figure 4.

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

21

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.* (Sometimes called round towards $-\infty$).
  round$(x) = x_-$.

- *Round up.* (Sometimes called round towards ∞).
  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 13 *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.*

Exercise 14 *Suppose x and y are single precision floating point numbers. Using* round to nearest, *is it true that* 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.*

Exercise 15 *Make up an example where x_- and x_+ are the same distance from x, and use the tie breaking rule to decide which is used for round-to-nearest.*

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 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

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

23

for any rounding mode and

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

for *round to nearest.*

Exercise 16 *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 17 *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

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

(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

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

Exercise 18 *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 *cor-*

rectly 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:

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

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

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

and

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

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

$$|\delta| \leq \epsilon$$

for all rounding modes and

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

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

Exercise 19 *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 20 *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*:* $1 + 10^{-5}$, $1 + 10^{-10}$, $1 + 10^{-15}$*?*

Exercise 21 *What is the smallest number a for which the correctly rounded value of $1 + a$ is exactly 1, using single precision with round to nearest. What about double precision?*

We see that the IEEE rule is that the result of a single floating point operation must always be the correctly rounded value of the true result. *However, this does not apply to a sequence of operations!* For example, consider the computation $a + b - c$, where $a = 1$, $b = 2^{-25}$, and $c = 1$, using IEEE single precision with *round to nearest*. These are all floating point numbers, with $a = c = 1.0 \times 2^0$ and $b = 1.0 \times 2^{-25}$. The value of $a + b$ is 1.00000000000000000000000001, which is not a single precision floating point number, so it is correctly rounded to the value 1. The final result is computed to be $1 - c$ which is 0. However, the exact value of $a + b - c$ is 2^{-25}, which is a floating point number.

Exercise 22 *Using the same example, what is the result of $a + b - c$ if the rounding mode is* round up*?*

Exercise 23 *In exact arithmetic, the addition operation is commutative, i.e.*

$$a + b = b + a$$

for any two numbers a, b, and also associative, i.e.

$$a + (b + c) = (a + b) + c.$$

Are these also true of the floating point addition operator $\oplus$? Give a careful answer.

The Intel Pentium chip received a lot of bad publicity in the Fall of 1994 when the fact that it had a floating point hardware bug was exposed. For example, on the original Pentium, the division operation

$$\frac{4195835}{3145727}$$

gave an inaccurate answer. The error took place only in rare cases, and could easily have remained undiscovered much longer than it did (it was found by a mathematics professor doing experiments in number theory). Nonetheless, it created a sensation, mainly because it turned out that Intel knew about the bug but had not released the information. The public outcry against incorrect floating point arithmetic depressed Intel's stock value significantly until the company finally agreed to replace defective processors for everyone, not just those that Intel thought really needed correct arithmetic! See The New York Times, Nov. 22, 1994, p. D1, and also *The Mathematics of the*

Pentium Division Bug, by Alan Edelman, SIAM Review, March 1997 (URL
`ftp://theory.lcs.mit.edu/pub/people/edelman/pentium/pentium.ps`)

Now we ask the question: how is correctly rounded arithmetic implemented? This is surprisingly complicated. 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:

$$
\begin{array}{rll}
& (\ 1.10000000000000000000000 &)_2 \times 2^1 \\
+ & (\ 1.00000000000000000000000 &)_2 \times 2^1 \\
= & (\ 10.1000000000000000000000 &)_2 \times 2^1 \\
\text{Normalize}: & (\ 1.01000000000000000000000 &)_2 \times 2^2.
\end{array}
$$

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:

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

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

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

$$
\begin{array}{rll}
& (\ 1.10000000000000000000000 &)_2 \times 2^1 \\
+ & (\ 0.00000000000000000000001|1 &)_2 \times 2^1 \\
= & (\ 1.10000000000000000000001|1 &)_2 \times 2^1 \\
\text{Round Down}: & (\ 1.10000000000000000000001 &)_2 \times 2^1 \\
\text{or Round Up}: & (\ 1.10000000000000000000010 &)_2 \times 2^1.
\end{array}
$$

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*. In the case of rounding to nearest, there is a tie, so the result with the even final bit is used (round up in this case).

The situation is still more complicated by the fact that rounding must not take place until the result is normalized. Consider the example of subtracting

the floating point number $1 + 2^{-22} + 2^{-23}$ from 3 (or equivalently adding 3 and $-(1 + 2^{-22} + 2^{-23})$). We get

$$
\begin{array}{rll}
 & (\ 1.10000000000000000000000 &)_2 \times 2^1 \\
- & (\ 0.10000000000000000000001|1 &)_2 \times 2^1 \\
= & (\ 0.11111111111111111111110|1 &)_2 \times 2^1 \\
\text{Normalize}: & (\ 1.11111111111111111111101 &)_2 \times 2^0.
\end{array}
$$

Thus, rounding is not needed in this example.

Exercise 24 *Work out what happens for the examples $1 + 2^{-24}$ and $1 - 2^{-24}$. Give some additional examples of your own.*

The following example shows that implementation of correct addition and subtraction is not trivial even in the case that 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:

$$
\begin{array}{rll}
 & (\ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ 0.11111111111111111111111|1 &)_2 \times 2^0 \\
= & (\ 0.00000000000000000000000|1 &)_2 \times 2^0 \\
\text{Normalize}: & (\ 1.00000000000000000000000|0 &)_2 \times 2^{-24}.
\end{array}
$$

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 did 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, we have

$$ x \ominus y = 2(x - y) \quad \text{instead of } x \ominus y = (x - y)(1 + \delta), \quad \text{where } \delta \leq \epsilon. \quad (3) $$

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 the final result $x \ominus y = 0$, an even worse answer than the XMP gives. Evidently, Cray supercomputers do not use correctly rounded arithmetic.

Machines supporting the IEEE standard do, however, have correctly rounded arithmetic. Exactly how this is implemented depends on the machine. On PC's, 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:

$$
\begin{array}{rl}
& (\ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ 0.00000000000000000000000|01000000000000000000000001 &)_2 \times 2^0 \\
= & (\ 0.11111111111111111111111|10111111111111111111111111 &)_2 \times 2^0 \\
\text{Normalize}: & (\ 1.11111111111111111111111|01111111111111111111111110 &)_2 \times 2^{-1} \\
\text{Round to Nearest}: & (\ 1.11111111111111111111111 &)_2 \times 2^{-1}
\end{array}
$$

This 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:

$$
\begin{array}{rl}
& (\ 1.00000000000000000000000| &)_2 \times 2^0 \\
- & (\ 0.00000000000000000000000|01 &)_2 \times 2^0 \\
= & (\ 0.11111111111111111111111|11 &)_2 \times 2^0 \\
\text{Normalize}: & (\ 1.11111111111111111111111|1 &)_2 \times 2^{-1} \\
\text{Round to Nearest}: & (\ 10.00000000000000000000000 &)_2 \times 2^{-1} \\
\text{Renormalize}: & (\ 1.00000000000000000000000 &)_2 \times 2^0
\end{array}
$$

In this case, normalizing and rounding 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.

Exercise 25 *Assume that x and y are normalized numbers, i.e. $1 \leq |m| < 2$, $1 \leq |p| < 2$. How many bits may it be necessary to shift the significand product $m \times p$ left or right to normalize the result?*

6 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,
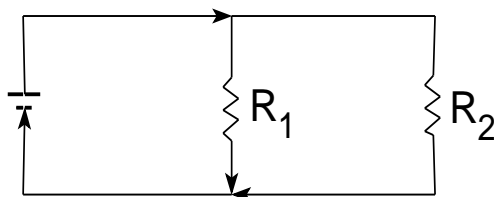
Figure 5: The Parallel Resistance Circuit

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 5.

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 arithmetic computation with an expression which has a NaN value also results in 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 x_k or y_k diverges faster to ∞.)

Exercise 26 *What are the possible values for*

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

33

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

Exercise 27 *What are sensible values for the expressions $\infty/0$, $0/\infty$ and ∞/∞?*

Exercise 28 *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 $\langle 0 = -0 \rangle$ has the value **true** while $\langle \infty = -\infty \rangle$ has 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 29 *What are the values of the expressions $0/(-0)$, $\infty/(-\infty)$ and $-\infty/(-0)$?*

Exercise 30 *What is the result of the parallel resistance formula if an input value is -0, or NaN?*

Another perhaps unexpected consequence of these conventions concerns arithmetic *comparisons*. When a and b are finite 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. In both cases we also have for *finite a*: $-\infty < a < \infty$. 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 are defined to have *different* values (the first **false**, the second **true**) if either a or b is a NaN.

Exercise 31 *Suppose a and b both have the value ∞. Which of the following are true: $a = b$, $a \geq b$ and $a > b$?*

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 32 *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 33 *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 normalized numbers? [5]

---

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

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, or Inexact	Set result to correctly rounded value

Exercise 34 *Consider the computation of the previous exercise again. Is it possible that c and d 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. For example, the Borland C++ compiler does pass on this feature to the user. 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.

7   The Intel Floating Point Processor

The IEEE floating point standard was developed in the early 1980's, during the design phase of the early Personal Computers (PC's). The two largest

manufacturers of chips for PC's incorporated most of the standard in the
early stages of their design process: these were Intel (used by IBM PC's
and clones) and Motorola (used by Apple Macintoshes and the early Sun
workstations). We shall confine our attention to the Intel 8087/8088 and its
successors, since these chips are the most widely used.

The original IBM PC used the Intel 8088 chip. This chip included the
central processing unit (CPU) and the arithmetic-logical unit (ALU) but did
not support floating point operations. The floating point unit (FPU) was
contained in the 8087 numeric processor extension (NPX), also known as
the floating point coprocessor chip. This design choice was made because
many users did not need floating point and, because floating point logic is
complicated, overall costs could be reduced by eliminating floating point from
the primary chip. The separate floating point coprocessor design continued
with the successors of the 8088/8087, namely the 80286/80287 (used by the
IBM AT) and the 80386/80387 DX(used by the IBM PS/2), or just 286/287
and 386/387 for short. However, the successor to the 386/387 DX, namely
the 486 DX, included the floating point unit on the main chip. (The slower
486 SX does not include floating point.) As the processors have became very
much faster with each new chip design, keeping floating point on a separate
chip became a liability rather than an advantage. Floating point instructions
can be executed on 8088, 80286 and 80386 machines without floating point
coprocessors, using *emulation*, but they take a great deal more processor
time.

Although the various versions of the Intel floating point unit differ in
many details, the basic organization is the same. Floating point instructions
operate primarily on data stored in eight 80-bit floating point registers, each
of which can accommodate an extended precision floating point number. The
eight registers are numbered 0 to 7 and are organized in a stack. At any
given time, the top register in the stack is denoted ST(0), the second-to-top
register ST(1), etc. The actual physical register corresponding to the top
register ST(0) is determined by a top-of-stack pointer stored in a 3-bit field
of the 16-bit *status word*, which is stored in a dedicated 16-bit register. If this
bitstring is 011, for example, ST(0) is equivalent to physical register 3, ST(1)
is equivalent to physical register 4, and so on, i.e. ST(7) is physical register
2. When the register stack is *pushed*, the top-of-stack pointer is *decremented*,
e.g. the top register is changed from physical register 3 to physical register 2,
ST(1) becomes physical register 3, and ST(7) becomes physical register 1.

The register stack is very convenient for the evaluation of arithmetic ex-

pressions. For example, consider the task of computing the expression

$$(a + b) * c$$

assuming the floating point numbers a, b and c are available in memory locations A,B and C respectively, and that the result is to be stored in memory location X. A sequence of assembly language instructions which will carry out this task is

```
FLD    A
FLD    B
FADD
FLD    C
FMUL
FSTP   X
```

Here the first FLD instruction *pushes* the value in memory location A onto the stack; in other words, it first decrements the top-of-stack pointer and then copies the value in A, namely a, to the new stack register $ST(0)$. The second FLD instruction then pushes the value in the memory location B onto the stack; this requires decrementing the stack pointer and copying the value in B, namely b, to the new $ST(0)$. At this point $ST(0)$ contains b and $ST(1)$ contains a. The FADD instruction then *adds* the value in $ST(0)$ to the value in $ST(1)$ and *pops* the stack, i.e. increments the top-of-stack pointer. The third FLD instruction pushes the value in location C, namely c, onto the stack. Then the FMUL instruction multiplies the new value in $ST(0)$, namely c, onto the value in $ST(1)$, namely $a + b$, and pops the stack, leaving the final value of the expression in the top register $ST(0)$. Finally, the FSTP instruction stores the final result in memory location X, popping the stack one more time. The register stack now has the same configuration it did before the expression evaluation began (either empty, or containing some other results still to be processed). The whole computation is summarized in Table 5.

Notice that the computation is always organized so that the latest result is at the top of the stack. The values in the registers are floating point numbers, not formulas, of course. The expression $a \oplus b$ is used rather than $a + b$, because this is the actual computed value, the rounded value of $a + b$.

Suppose $ST(0)$ is initially equivalent to physical register 3. Then the contents of the physical registers during the expression evaluation are shown in Table 6. The symbol $\rightarrow$ indicates the top register in the stack at each point in time.

Table 5: Logical Register Stack Contents, at Successive Times

Register	Time 0	Time 1	Time 2	Time 3	Time 4	Time 5
ST(0)		a	b	$a \oplus b$	c	$(a \oplus b) \otimes c$
ST(1)			a		$a \oplus b$	
ST(2)						
ST(3)						
ST(4)						
ST(5)						
ST(6)						
ST(7)						

Table 6: Physical Register Stack Contents, at Successive Times

Register	Time 0	Time 1	Time 2	Time 3	Time 4	Time 5
P.R. 0						
P.R. 1			$\rightarrow \ b$	b	$\rightarrow \ c$	
P.R. 2		$\rightarrow \ a$	a	$\rightarrow \ a \oplus b$	$a \oplus b$	$\rightarrow \ (a \oplus b) \otimes c$
P.R. 3						
P.R. 4						
P.R. 5						
P.R. 6						
P.R. 7						

Each time the register stack is pushed, the top register, i.e. $ST(0)$, moves one position in terms of the physical registers. Also, when the register stack is popped, and $ST(0)$ moves back to its previous position, the numerical value in the physical register remains unchanged until it is overwritten, e.g. by a subsequent push instruction.

It is clear that the register stack can conveniently handle arithmetic expressions nested up to seven levels. However, it is possible to overflow the stack by pushing it too many times. When this happens, an invalid operation is signaled and, if the invalid operation exception is masked, a NaN is generated. Clearly, compiler writers should bear this in mind so that stack overflow does not occur when complex expressions are parsed.

There are other versions of the floating point arithmetic instructions which require only one operand to be in a register and the other operand to be in memory. Though these result in slightly shorter assembly language programs, they make the functionality of the stack somewhat less clear.

In addition to the top-of-stack pointer, the *status word* also contains the five exception flags which are required by the IEEE standard. The flag is set to one when the corresponding exception takes place. These flags can be cleared by the programmer using special assembly language instructions such as FCLEX.

In addition to the status word, the floating point unit has another special word called the *control word* which is stored in another dedicated 16-bit register. The control word is used to control the *rounding mode*, the *precision mode* and the *exception masks*. There are two bits associated with control of the rounding mode, set to 00 for round to nearest, for example. There are also two bits used to control the precision mode, which can have one of three values: single, double and extended. Five bits are used for exception masks. When an exception occurs, and the corresponding flag in the status word is set, the corresponding mask in the control word is examined. If the exception mask is zero, the exception is trapped and control is passed to the user-provided trapping routine. If the exception mask is one, i.e. the exception is masked, the processor takes the default action described in Table 4 and execution continues normally. The user can load any desired value for the control word using the assembly instruction FLDCW. However, this is not a good idea! The best way to reset the floating point environment to a default initial state is by using the instruction FNINIT: this clears all status flags in the status word, and sets the control word as follows: rounding control is round to nearest, precision mode is extended, and all exceptions masked.

8   Higher-Level Languages

Although it is interesting to be able to access the features of the IEEE standard (such as rounding control) by using Assembly language, in practice almost all users of floating point computing use higher-level languages. The first of these was Fortran, which dates to 1958. Its successors Fortran 77 and Fortran 90 are still in wide use in the scientific computing community. In the 1980's, the C programming language became a popular choice for numerical computing, as has its derivatives C++ and Java more recently.. A great deal of numerical code is available in Fortran and C, including many important public-domain software packages: go to the URL `http://www.netlib.org` for information.

A very popular alternative to more conventional compiled languages is the interactive system Matlab. Although not as fast as compiled code, Matlab is very popular because of its ease of use, its extensive software toolboxes and its graphics capabilities. An excellent introduction to numerical computing using Matlab is Charles Van Loan's *Introduction to Scientific Computing: A Matrix-Vector Approach using Matlab* (Prentice-Hall, 1997).

Unfortunately, many of the features of the IEEE standard remain inaccessible to the user of high-level languages. For example, when writing code in a high-level language, one cannot be sure what precision is in use: for example, when two double-precision numbers are added together, this operation may take place in an extended precision register, and one may not know whether or not the result is rounded to double-precision following the operation. Making the features and intents of the standard fully available to users of higher-level languages remains a challenge to the compiler and interpreter writers of the next generation.