FLOATING-POINT NUMBERS

Andrew S. Tanenbaum

Vrije Universiteit
AMSTERDAM

1. Editor's Note (by David Brailsford)

This article is taken almost in its entirety from Appendix B. of Andy Tanenbaum's book: *Structured Computer Organization (4th Edn.)*. I have performed just a few minor re-wordings to make this into a suitable handout for use in G51CSA

2. INTRODUCTION

In many calculations the range of numbers used is very large. For example. a calculation in astronomy might involve the mass of the electron, 9×10^{-28} grams, and the mass of the sun, 2×10^{33} grams, a range exceeding 10^{60}. These numbers could be represented by

0000000000000000000000000000000000.0000000000000000000000000009
2000000000000000000000000000000000.0000000000000000000000000000

and all calculations could be carried out keeping 34 digits to the left of the decimal point and 28 rlaces to the right of it. Doing so would allow 62 significant digits in the results. On a binary computer, multiple-precision arithmetic could be used to provide enough significance. However. the mass of the sun is not even known accurately to five significant digits, let alone 62. In fact few measurements of any kind can (or need) be made accurately to 62 significant digits. Although it would he possible to keep all intermediate results to 62 significant digits and then throw away 50 or 60 of them before printing the final results, doing this is wasteful of both CPU time and memory.

What is needed is a system for representing numbers in which the range of expressible numbers is independent of the number of significant digits. In this appendix, such a system will he discussed. It is hased on the scientific notation commonly used in physics, chemistry, and engineering.

3. PRINCIPLES OF FLOATING POINT

One way of separating the range from the precision is to express numbers in the familiar scientific notation

$$n = f \times 10^{e}$$

where *f* is called the **fraction**, or **mantissa**, and *e* is a positive or negative integer called the **exponent**. The computer version of this notation is called **floating point**. Some examples of numbers expressed in this form are

$$3.14 = 0.314 \times 10^{1} = 3.14 \times 10^{0}$$

$$0.000001 = 0.1 \times 10^{-5} = 1.0 \times 10^{-6}$$

$$1941 = 0.1941 \times 10^{4} = 1.941 \times 10^{3}$$

The range is effectively determined by the number of digits in the exponent and the precision is determined by the number of digits in the fraction. Because there is more than one way to represent a given number, one form is usually chosen as the standard. In order to investigate the properties of this method of representing numbers, consider a representation, *R*, with a signed three-digit fraction in the range $0.1 \le |f| < 1$ or zero and a signed two-digit exponent. These numbers range in magnitude from $+0.100 \times 10^{99}$ to $+0.999 \times 10^{+99}$, a span of nearly 199 orders of magnitude, yet only five digits and two signs are needed to store a number.

Floating-point numbers can be used to model the real-number system of mathematics. although there are some important differences. Figure 1 gives a grossly exaggerated schematic of the real number line. The real line is divided up into seven regions:

1. Large negative numbers less than -0.999×10^{99}.
2. Negative numbers between -0.999×10^{99} and -0.100×10^{-99}.
3. Small negative numbers with magnitudes less than 0.100×10^{-99}.
4. Zero.
5. Small positive numbers with magnitudes less than 0.100×10^{-99}
6. Positive numbers between 0.100×10^{-99} and 0.999×10^{99}
7. Large positive numbers greater than 0.999×10^{99}

One major difference between the set of numbers representable with three fraction and two exponent digits and the real numbers is that the former cannot he used to express any numbers in region 1, 3, 5, or 7. If the result of an arithmetic operation yields a number in regions 1 or 7 — for example, $10^{60} \times 10^{60} = 10^{120}$ — an **overflow error** will occur and the answer will be incorrect. The reason is due to the finite nature of the representation for numbers and is unavoidable.

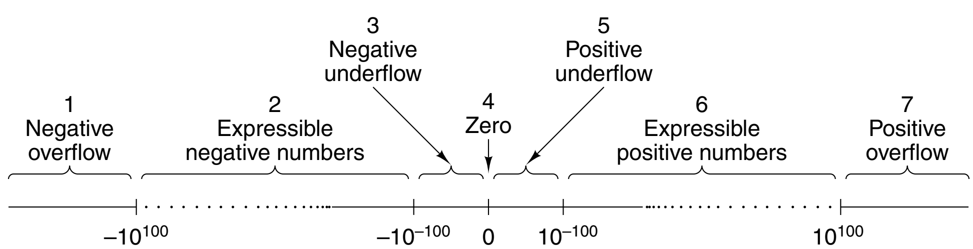

Figure 1. The real number line can be divided into seven regions

Similarly, a result in region 3 or 5 cannot be expressed either. This situation is called an **underflow error**. Underflow error is less serious than an overflow error, because 0 is often a satisfactory approximation to numbers in regions 3 and 5. A bank balance of 10^{-102} dollars is hardly better than a bank balance of 0.

Another important difference between floating-point numbers and the real numbers is their density. Between any two real numbers. x and y, is another real number, no matter how close x is to y. This property comes from the fact that for any distinct real numbers, x and y, $z = (x + y) / 2$ is a real number between them. The real numbers form a continuum.

Floating-point numbers, in contrast, do not form a continuum. Exactly 179,100 positive numbers can be expressed in the five-digit, two-sign system used above, 179,100 negative numbers and 0 (which can be expressed in many ways), for a total of 358,201 numbers. Of the infinite number of real numbers hetween -10^{+100} and $+0.999 \times 10^{99}$, only 358,201 of them can be specified by this notation. They are symbolized by the dots in figure 1.

It is quite possible for the result of a calculation to be one of the other numbers, even though it is in region 2 or 6. For example, $+0.100 \times 10^{3}$ divided by 3 cannot be expressed *exactly* in our system of representation. If the result of a calculation cannot be expressed in the number representation heing used. the obvious thing to do is to use the nearest number that can be expressed. This process is called **rounding**.

The spacing between adjacent expressible numhers is not constant throughout region 2 or 6. The separation between $+0.998 \times 10^{99}$ and $+0.999 \times 10^{99}$ is vastly more than the separation between $+0.998 \times 10^{0}$ and $+0.999x10^{0}$. However, when the separation between a number and its successor is expressed as a percentage of that number. there is no systematic variation throughout region 2 or 6. In other words, the **relative error** introduced by rounding is approximately the same for small numbers as large numhers.

Although the preceding discussion was in terms of a representation system with a three-digit fraction and a two-digit exponent, the conclusions drawn are valid for other representation systems as well. Changing the number of digits in the fraction or exponent merely shifts the boundaries of regions 2 and 6 and changes the number of expressible points in them. Increasing the number of digits in the fraction increases the density of points and therefore improves the accuracy of approximations. Increasing the number of digits in the exponent increases the size of regions 2 and 6 by shrinking regions 1, 3, 5, and 7. Figure 2 shows the approximate boundaries of region 6 for floating-point decimal numbers for various sizes of fraction and exponent.

Digits in fraction	**Digits in exponent**	**Lower bound**	**Upper bound**
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Figure 2. The approximate lower and upper bounds of expressible (unnormalized) floating-point decimal numbers

A variation or this representation is used in computers. For efficiency, exponentiation is to base 2, 4, 8, or 16 rather than 10, in which case the fraction consists or a string of binary, base-4, octal, or hexadecimal digits. If the leftmost of these digits is zero. all the digits can be shifted one place to the left and the exponent decreased by 1, without changing the value of the numher (barring underflow), A fraction with a nonzero leftmost digit is said to be **normalized**.

Normalized numbers are generally preferable to unnormalized numbers. because there is only one normalized form, whereas there are many unnormalized forms. Examples of normalized floating-point numbers are given in Figure 3 for two bases of exponentiation. In these examples a 16-bit fraction (including sign bit) and a 7-bit exponent using excess 64 notation are shown. Thc radix point is to the left of the leftmost fraction bit — that is. to the right of the exponent.

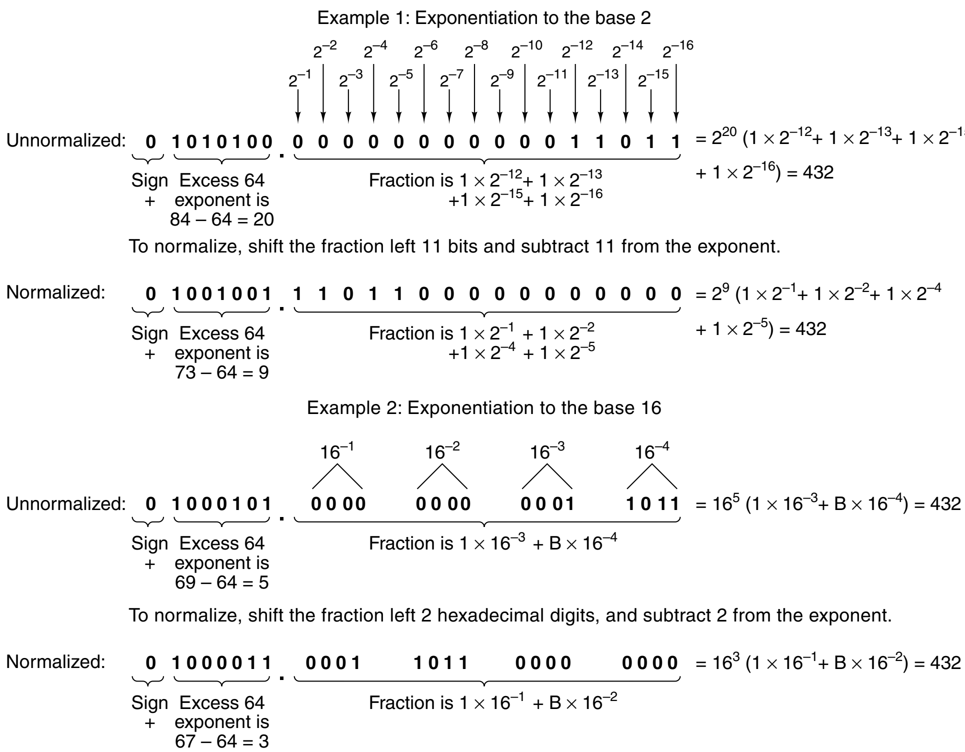

Figure 3. Examples of normalized floating-point numbers

4. IEEE FLOATING-POINT STANDARD 754

Until about 1980, each computer manufacturer had its own floating-point format. Needless to say. all were different. Worse yet. some of them actually did arithmetic incorrectly because floating-point arithmetic has some subtleties not obvious to the average hardware designer.

To rectify this situation, in the late 1970s IEEE set up a committee to standardize floating-point arithmetic. The goal was not only to permit floating-point data to be exchanged among different computers but also to provide hardware designers with a model known to be correct. The resulting work led to IEEE Standard 754 (IEEE, 1985). Most CPUs these days (including Intel, SPARC, MIPS etc.) have floating-point instructions that conform to the IEEE floating-point standard. Unlike many standards, which tend to be wishy-washy compromises that please no one, this one is not bad, in large part because it was primarily the work or one person, Berkeley math professor William Kahan. The standard will he described in the remainder of this section. The standard defines three formats: single precision (32 bits). double precision (64 bits), and extended precision (80 bits). The extended-precision format is intended to reduce roundoff errors. It is used primarily inside floating-point arithmetic units, so we will not discuss it further. Both the single- and double-precision formats use radix 2 for fractions und excess notation for exponents. The formats are shown in Figure 4, Both formats start with a sign bit for the number as a whole, 0 being positive and 1 being negative. Next comes the exponent, using excess 127 for single precision and excess 1023 for double precision. The minimum (0) and maximum (255 and 2047) exponents are not used for normalized numbers; they have special uses described below. Finally, we have the fractions, 23 and 52 bits, respectively.

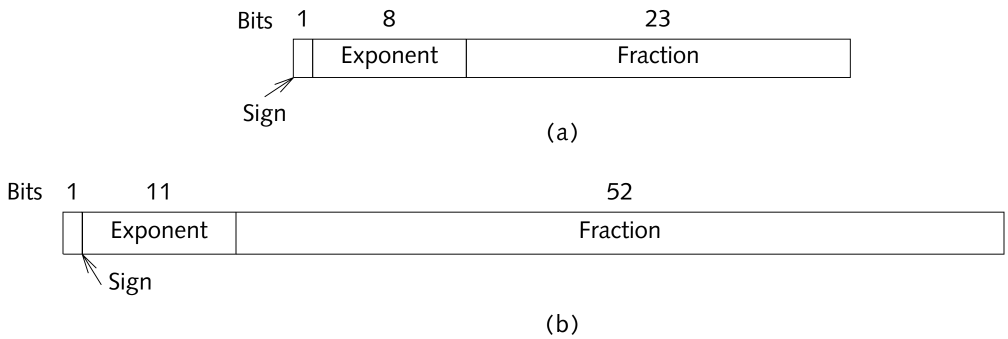

Figure 4. IEEE floating-point formats. (a) Single precision, (b) Double precision.

A normalized fraction begins with a binary point. followed by a 1 bit, and then the rest of the fraction. Following a practice started on the PDP-11, the authors of the standard realized that the leading 1 bit in the fraction does not have to be stored, since it can just be assumed to be present. Consequently, the standard defines the fraction in a slightly different way from usual. It consists of an implied 1 bit, an implied binary point and then either 23 or 52 arbitrary bits. If all 23 or 52 fraction bits are 0s, the fraction has the numerical value 1.0; if all of them are 1s, the fraction is numerically slightly less than 2.0. To avoid confusion with a conventional fraction, the combination of the implied 1, the implied binary point and the 21 or 52 explicit bits is sometimes called a **significand** instead of a fraction or mantissa. All normalized numbers have a significand, s, in the range $1 \leq s < 2$.

The numerical characteristics of the IEEE floating-point numbers are shown in Figure 5. As examples, consider the numbers 0.5, 1, and 1.5 in normalized single-precision format. These are represented in hexadecimal as 3F000000, 3F800000, and 3FC00000, respectively.

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excesss 1023
Exponent range	−126 to +127	−1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

Figure 5. Characteristics of IEEE floating-point numbers.

One of the traditional problems with floating-point numbers is how to deal with underflow, overflow. and uninitialized numhers. The IEEE standard deals with these problems explicitly, borrowing its approach in part from the CDC 6600 computer. In addition to normalized numbers. the standard has four other numerical types, described below and shown in Figure 6.

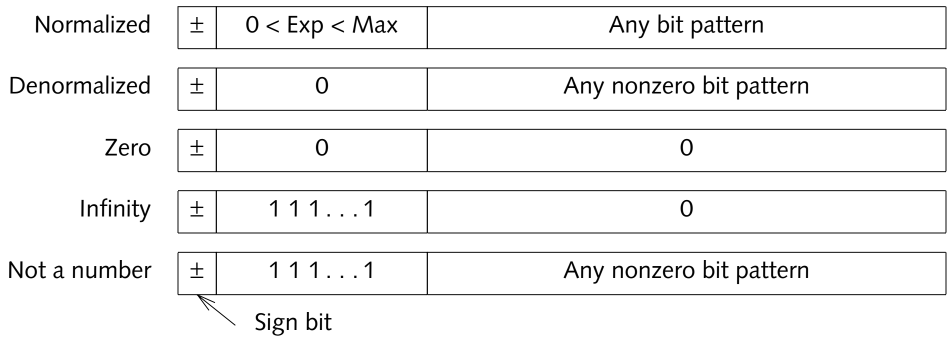

Figure 6. IEEE numerical types.

A problem arises when the rcsult of a calculation has a magnitude smaller than the smallest normalized floating-point number that can he represented in this system. Previously. most hardware took one of two approaches: just set the result to zero and continue, or cause a floating-point underflow trap. Neither of these is really satisfactory, so IEEE invented **denormalizcd numbers**. These numhers have an exponent of 0 and a fraction given by the following 23 or 52 bits. The implicit 1 bit to the left of the binary point now becomes a 0. Denormalized numbers can he distinguished from normalized ones because the latter are not permitted to have an exponent of 0.

The smallest normalized single precision number has a 1 as exponent and 0 as fraction. and represents 1.0×2^{-126}. The largest denormalized number has a 0 as exponent and all 1s in the fraction, and represents about $0.9999999 \times 2^{-127}$, which is almost the same thing. One thing to note however, is that this numher has only 23 bits of significance, versus 24 for all normalized numhers.

As calculations further decrease this result, the exponent stays put at 0, hut the first few bits of the fraction become zeros, reducing both thc value and the number of significant bits in the fraction. The smallest nonzero denormalized number consists of a 1 in the rightmost bit. with the rest being 0. The exponent represents 2^{-127} and the fraction represents 2^{-23} so the value is 2^{-150}. This scheme provides for a graceful underflow by giving up significance instead of jumping to 0 when the result cannot be expressed as a normalized number.

Two zeros are present in this scheme, positive and negative, determined by the sign bit. Both have an exponent of 0 and a fraction of 0. Here too, the bit to the left of the binary point is implicitly 0 rather than 1.

Overflow cannot be handled gracefully. There are no bit combinations left. Instead, a special representation is provided for infinity. consisting of an exponent with all Is (not allowed for normalized numbers), and a fraction of 0. This number can be used as an operand and behaves according tn the usual mathematica] rules for infinity. For example infinity plus anything is infinity, and any finite number divided by infinity is zero. Similarly, any finite number divided by zero yields infinity.

What about infinity divided by infinity? The result is undefined. To handle this case, another special format is provided, called **NaN** (**Not a Number**). It too, can be used as an operand with predictable results.