IEE 745 Fkoat

Contents

Articles

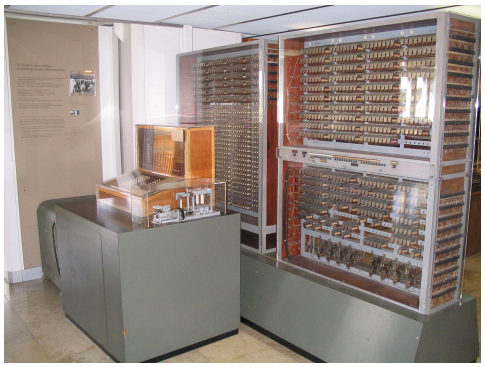
References

Article Licenses

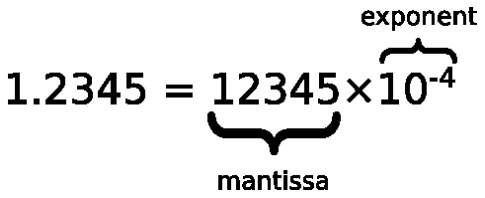
Floating point

In computing, **floating point** describes a method of representing an approximation to real numbers in a way that can support a wide range of values. The numbers are, in general, represented approximately to a fixed number of significant digits (the mantissa) and scaled using an exponent. The base for the scaling is normally 2, 10 or 16. The typical number that can be represented exactly is of the form:

$$\textit{Significant digits} \times \textit{base}^{\textit{exponent}}$$



The first programmable computer, the Z3, included floating point arithmetic (replica on display at Deutsches Museum in Munich).



A diagram showing a representation of a floating point number using a mantissa and an exponent.

The idea of floating-point representation over intrinsically integer fixed-point numbers, which consist purely of significand, is that expanding it with the exponent component achieves greater range. For instance, to represent large values, e.g. distances between galaxies, there is no need to keep all 39 decimal places down to femtometre-resolution, employed in particle physics. Assuming that the best resolution is in light years, only 9 most significant decimal digits matter whereas 30 others bear pure noise and, thus, can be safely dropped. This is 100-bit saving in storage. Instead of these 100 bits, much fewer are used to represent the scale (the exponent), e.g. 8 bits or 2 decimal digits. Now, one number can encode the astronomic and subatomic distances with the same 9 digits of accuracy. But, because 9 digits is 100 times less accurate than 9+2 digits reserved for scale, this is considered as precision-for-range trade-off. The example also explains that using scaling to extend the dynamic range results in another contrast with usual fixed-point numbers: their values are not uniformly spaced. Small values, the ones close to zero, can be represented with much higher resolution (1 femtometre) than distant ones because greater scale (light years) must be selected for encoding significantly larger values.[1] That is, floating-point cannot represent point coordinates with atomic accuracy in the other galaxy, only close to the origin.

The term *floating point* refers to the fact that their radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated as the exponent component in the internal representation, and floating-point can thus be thought of as a computer realization of scientific notation. Over the years, a variety of floating-point representations have been used in computers. However, since the 1990s, the most commonly encountered representation is that defined by the IEEE 754 Standard.

The speed of floating-point operations, commonly referred to in performance measurements as FLOPS, is an important machine characteristic, especially in software that performs large-scale mathematical calculations.

Overview

A number representation (called a numeral system in mathematics) specifies some way of storing a number that may be encoded as a string of digits. The arithmetic is defined as a set of actions on the representation that simulate classical arithmetic operations.

There are several mechanisms by which strings of digits can represent numbers. In common mathematical notation, the digit string can be of any length, and the location of the radix point is indicated by placing an explicit "point" character (dot or comma) there. If the radix point is not specified then it is implicitly assumed to lie at the right (least significant) end of the string (that is, the number is an integer). In fixed-point systems, some specific assumption is made about where the radix point is located in the string. For example, the convention could be that the string consists of 8 decimal digits with the decimal point in the middle, so that "00012345" has a value of 1.2345.

In scientific notation, the given number is scaled by a power of 10 so that it lies within a certain range—typically between 1 and 10, with the radix point appearing immediately after the first digit. The scaling factor, as a power of ten, is then indicated separately at the end of the number. For example, the revolution period of Jupiter's moon Io is 152853.5047 seconds, a value that would be represented in standard-form scientific notation as 1.528535047×10^5 seconds.

Floating-point representation is similar in concept to scientific notation. Logically, a floating-point number consists of:

- A signed digit string of a given length in a given base (or radix). This digit string is referred to as the significand, coefficient or, less often, the mantissa (see below). The length of the significand determines the *precision* to which numbers can be represented. The radix point position is assumed to always be somewhere within the significand—often just after or just before the most significant digit, or to the right of the rightmost (least significant) digit. This article will generally follow the convention that the radix point is just after the most significant (leftmost) digit.
- A signed integer exponent, also referred to as the characteristic or scale, which modifies the magnitude of the number.

To derive the value of the floating point number, one must multiply the *significand* by the *base* raised to the power of the *exponent*, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent—to the right if the exponent is positive or to the left if the exponent is negative.

Using base-10 (the familiar decimal notation) as an example, the number 152853.5047, which has ten decimal digits of precision, is represented as the significand 1.528535047 together with an exponent of 5 (if the implied position of the radix point is after the first most significant digit, here 1). To determine the actual value, a decimal point is placed after the first digit of the significand and the result is multiplied by 10^5 to give 1.528535047×10^5, or 152853.5047. In storing such a number, the base (10) need not be stored, since it will be the same for the entire range of supported numbers, and can thus be inferred.

Symbolically, this final value is

$$s \times b^e$$

where s is the value of the significand (after taking into account the implied radix point), b is the base, and e is the exponent.

Equivalently:

$$\frac{s}{b^{p-1}} \times b^e$$

where s here means the integer value of the entire significand, ignoring any implied decimal point, and p is the precision—the number of digits in the significand.

Historically, several number bases have been used for representing floating-point numbers, with base 2 (binary) being the most common, followed by base 10 (decimal), and other less common varieties, such as base 16

(hexadecimal notation), as well as some exotic ones like 3 (see Setun).

Floating point numbers are rational numbers because they can be represented as one integer divided by another. For example 1.45×10^3 is (145/100)*1000 or 145000/100. The base however determines the fractions that can be represented. For instance, 1/5 cannot be represented exactly as a floating point number using a binary base but can be represented exactly using a decimal base (0.2, or 2×10^{-1}). However 1/3 cannot be represented exactly by either binary (0.010101...) nor decimal (0.333....), but in base 3 it is trivial (0.1 or 1×3^{-1}) . The occasions on which infinite expansions occur depend on the base and its prime factors, as described in the article on Positional Notation.

The way in which the significand, exponent and sign bits are internally stored on a computer is implementation-dependent. The common IEEE formats are described in detail later and elsewhere, but as an example, in the binary single-precision (32-bit) floating-point representation p=24 and so the significand is a string of 24 bits. For instance, the number π's first 33 bits are 11001001 00001111 11011010 10100010 0. Rounding to 24 bits in binary mode means attributing the 24th bit the value of the 25th which yields 11001001 00001111 11011011. When this is stored using the IEEE 754 encoding, this becomes the significand s with e = 1 (where s is assumed to have a binary point to the right of the first bit) after a left-adjustment (or *normalization*) during which leading or trailing zeros are truncated should there be any. Note that they do not matter anyway. Then since the first bit of a non-zero binary significand is always 1 it need not be stored, giving an extra bit of precision. To calculate π the formula is

$$\begin{aligned}
&\left(1+\sum_{n=1}^{p-1}\mathrm{bit}_n\times 2^{-n}\right)\times 2^e \\
&=\left(1+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-4}+1\times 2^{-7}+\cdots+1\times 2^{-23}\right)\times 2^1 \\
&=1.5707964\times 2
\end{aligned}$$

where n is the normalized significand's n-th bit from the left. Normalization, which is reversed when 1 is being added above, can be thought of as a form of compression; it allows a binary significand to be compressed into a field one bit shorter than the maximum precision, at the expense of extra processing.

The word "mantissa" is often used as a synonym for significand. Use of mantissa in place of significand or coefficient is discouraged, as the mantissa is traditionally defined as the fractional part of a logarithm, while the *characteristic* is the integer part. This terminology comes from the manner in which logarithm tables were used before computers became commonplace. Log tables were actually tables of mantissas.

Some other computer representations for non-integral numbers

Floating-point representation, in particular the standard IEEE format, is by far the most common way of representing an approximation to real numbers in computers because it is efficiently handled in most large computer processors. However, there are alternatives:

- Fixed-point representation uses integer hardware operations controlled by a software implementation of a specific convention about the location of the binary or decimal point, for example, 6 bits or digits from the right. The hardware to manipulate these representations is less costly than floating-point and is also commonly used to perform integer operations. Binary fixed point is usually used in special-purpose applications on embedded processors that can only do integer arithmetic, but decimal fixed point is common in commercial applications.
- Binary-coded decimal (BCD) is an encoding for decimal numbers in which each digit is represented by its own binary sequence. It is possible to implement a floating point system with BCD encoding.
- Logarithmic number systems represent a real number by the logarithm of its absolute value and a sign bit. The value distribution is similar to floating-point, but the value-to-representation curve, i. e. the graph of the logarithm function, is smooth (except at 0). Contrary to floating-point arithmetic, in a logarithmic number system multiplication, division and exponentiation are easy to implement but addition and subtraction are difficult. The level index arithmetic of Clenshaw, Olver, and Turner is a scheme based on a generalised logarithm

representation.

- Where greater precision is desired, floating-point arithmetic can be implemented (typically in software) with variable-length significands (and sometimes exponents) that are sized depending on actual need and depending on how the calculation proceeds. This is called arbitrary-precision floating point arithmetic.
- Some numbers (*e.g.,* 1/3 and 1/10) cannot be represented exactly in binary floating-point, no matter what the precision is. Software packages that perform rational arithmetic represent numbers as fractions with integral numerator and denominator, and can therefore represent any rational number exactly. Such packages generally need to use "bignum" arithmetic for the individual integers.
- Computer algebra systems such as Mathematica and Maxima can often handle irrational numbers like π or $\sqrt{3}$ in a completely "formal" way, without dealing with a specific encoding of the significand. Such programs can evaluate expressions like "$\sin 3\pi$" exactly, because they "know" the underlying mathematics.

Range of floating-point numbers

Floating point number consists of two fixed-point components, whose range depends exclusively on the number of bits or digits in their representation. Whereas components linearly depend on their range, the floating point range linearly depends on the significant range and exponentially on the range of exponent component, which attaches outstandingly wider range to the number.

On a typical computer system, a 'double precision' (64-bit) binary floating-point number has a coefficient of 53 bits (one of which is implied), an exponent of 11 bits, and one sign bit. Positive floating-point numbers in this format have an approximate range of 10^{-308} to 10^{308}, because the range of the exponent is [−1022,1023] and 308 is approximately $\log_{10}(2^{1023})$. The complete range of the format is from about -10^{308} through $+10^{308}$ (see IEEE 754).

The number of normalized floating point numbers in a system F (*B*, *P*, *L*, *U*) (where *B* is the base of the system, *P* is the precision of the system to *P* numbers, *L* is the smallest exponent representable in the system, and *U* is the largest exponent used in the system) is: $2(B-1)(B^{P-1})(U-L+1)+1$.

There is a smallest positive normalized floating-point number, Underflow level = UFL = B^L which has a 1 as the leading digit and 0 for the remaining digits of the significand, and the smallest possible value for the exponent.

There is a largest floating point number, Overflow level = OFL = $\left(1-B^{-P}\right)\left(B^{U+1}\right)$ which has $B-1$ as the value for each digit of the significand and the largest possible value for the exponent.

In addition there are representable values strictly between −UFL and UFL. Namely, zero and negative zero, as well as denormalized numbers.

History

Leonardo Torres y Quevedo, in 1914 published an analysis of floating point based on the analytic engine.

Leonardo Torres y Quevedo in 1914 designed an electro-mechanical version of the Analytical Engine of Charles Babbage which included floating-point arithmetic.[2] In 1938, Konrad Zuse of Berlin completed the Z1, the first mechanical binary programmable computer, this was however unreliable in operation.[3] It worked with 22-bit binary floating-point numbers having a 7-bit signed exponent, a 15-bit significand (including one implicit bit), and a sign bit. The memory used sliding metal parts to store 64 words of such numbers. The relay-based Z3, completed in 1941 had representations for plus and minus infinity. It implemented defined operations with infinity such as $1/\infty = 0$ and stopped on undefined operations like $0\times\infty$. It also implemented the square root operation in hardware.

Zuse also proposed, but did not complete, carefully rounded floating–point arithmetic that would have included ±∞ and NaNs, anticipating features of IEEE Standard floating–point by four decades.[4] By contrast, von Neumann recommended against floating point for the 1951 IAS machine, arguing that fixed point arithmetic was preferable.[5]

Konrad Zuse, architect of the first programmable computer, which used 22-bit binary floating point.

The first *commercial* computer with floating point hardware was Zuse's Z4 computer designed in 1942–1945. The Bell Laboratories Mark V computer implemented decimal floating point in 1946.[6]

The Pilot ACE had binary floating point arithmetic which became operational at National Physical Laboratory, UK in 1950. A total of 33 were later sold commercially as the English Electric DEUCE. The arithmetic was actually implemented as subroutines, but with a one megahertz clock rate, the speed of floating point operations and fixed point was initially faster than many competing computers, and since it was only software, all the DEUCE's had it.

The mass-produced vacuum tube-based IBM 704 followed in 1954; it introduced the use of a biased exponent. For many decades after that, floating-point hardware was typically an optional feature, and computers that had it were said to be "scientific computers", or to have "scientific computing" capability. It was not until the launch of the Intel i486 in 1989 that *general-purpose* personal computers had floating point capability in hardware as standard.

The UNIVAC 1100/2200 series, introduced in 1962, supported two floating-point formats. Single precision used 36 bits, organized into a 1-bit sign, an 8-bit exponent, and a 27-bit significand. Double precision used 72 bits organized

as a 1-bit sign, an 11-bit exponent, and a 60-bit significand. The IBM 7094, introduced the same year, also supported single and double precision, with slightly different formats.

Prior to the IEEE-754 standard, computers used many different forms of floating-point. These differed in the word sizes, the format of the representations, and the rounding behavior of operations. These differing systems implemented different parts of the arithmetic in hardware and software, with varying accuracy.

The IEEE-754 standard was created in the early 1980s after word sizes of 32 bits (or 16 or 64) had been generally settled upon. This was based on a proposal from Intel who were designing the i8087 numerical coprocessor. Prof. W. Kahan was the primary architect behind this proposal, along with his student Jerome Coonen at U.C. Berkeley and visiting Prof. Harold Stone, for which he was awarding the 1989 Turing award.[7] Among the innovations are these:

- A precisely specified encoding of the bits, so that all compliant computers would interpret bit patterns the same way. This made it possible to transfer floating-point numbers from one computer to another.
- A precisely specified behavior of the arithmetic operations: arithmetic operations were required to be correctly rounded, i.e. to give the same result as if infinitely precise arithmetic was used and then rounded. This meant that a given program, with given data, would always produce the same result on any compliant computer. This helped reduce the almost mystical reputation that floating-point computation had for seemingly nondeterministic behavior.
- The ability of exceptional conditions (overflow, divide by zero, etc.) to propagate through a computation in a benign manner and be handled by the software in a controlled way.

IEEE 754: floating point in modern computers

The IEEE has standardized the computer representation for binary floating-point numbers in IEEE 754 (aka. IEC 60559). This standard is followed by almost all modern machines. Notable exceptions include IBM mainframes, which support IBM's own format (in addition to the IEEE 754 binary and decimal formats), and Cray vector machines, where the T90 series had an IEEE version, but the SV1 still uses Cray floating-point format.

The standard provides for many closely related formats, differing in only a few details. Five of these formats are called *basic formats* and others are termed *extended formats*, and three of these are especially widely used in computer hardware and languages:

- Single precision, called "float" in the C language family, and "real" or "real*4" in Fortran. This is a binary format that occupies 32 bits (4 bytes) and its significand has a precision of 24 bits (about 7 decimal digits).
- Double precision, called "double" in the C language family, and "double precision" or "real*8" in Fortran. This is a binary format that occupies 64 bits (8 bytes) and its significand has a precision of 53 bits (about 16 decimal digits).
- Double extended format, 80-bit floating point value. This is implemented on most personal computers but not on other devices. Sometimes "long double" is used for this in the C language family (the C99 and C11 standards "IEC 60559 floating-point arithmetic extension- Annex F" recommend the 80-bit extended format to be provided as "long double" when available), though "long double" may be a synonym for "double" or may stand for quadruple precision. Extended precision can help minimise accumulation of round-off error in intermediate calculations.[8]

Less common IEEE formats include:

- Quadruple precision (binary128). This is a binary format that occupies 128 bits (16 bytes) and its significand has a precision of 113 bits (about 34 decimal digits).
- Double precision (decimal64) and quadruple precision (decimal128) decimal floating point formats. These formats, along with the single precision (decimal32) format, are intended for performing decimal rounding correctly.
- Half, also called float16, a 16-bit floating point value.

Any integer with absolute value less than or equal to 2^{24} can be exactly represented in the single precision format, and any integer with absolute value less than or equal to 2^{53} can be exactly represented in the double precision format. Furthermore, a wide range of powers of 2 times such a number can be represented. These properties are sometimes used for purely integer data, to get 53-bit integers on platforms that have double precision floats but only 32-bit integers.

The standard specifies some special values, and their representation: positive infinity (+∞), negative infinity (−∞), a negative zero (−0) distinct from ordinary ("positive") zero, and "not a number" values (NaNs).

Comparison of floating-point numbers, as defined by the IEEE standard, is a bit different from usual integer comparison. Negative and positive zero compare equal, and every NaN compares unequal to every value, including itself. All values except NaN are strictly smaller than +∞ and strictly greater than −∞. Finite floating-point numbers are ordered in the same way as their values (in the set of real numbers).

To a rough approximation, the bit representation of an IEEE binary floating-point number is proportional to its base 2 logarithm, with an average error of about 3%. (This is because the exponent field is in the more significant part of the datum.) This can be exploited in some applications, such as volume ramping in digital sound processing.

A project for revising the IEEE 754 standard was started in 2000 (see IEEE 754 revision); it was completed and approved in June 2008. It includes decimal floating-point formats and a 16 bit floating point format ("binary16"). binary16 has the same structure and rules as the older formats, with 1 sign bit, 5 exponent bits and 10 trailing significand bits. It is being used in the NVIDIA Cg graphics language, and in the openEXR standard.[9]

Internal representation

Floating-point numbers are typically packed into a computer datum as the sign bit, the exponent field, and the significand (mantissa), from left to right. For the IEEE 754 binary formats (basic and extended) which have extant hardware implementations, they are apportioned as follows:

Type	Sign	Exponent	Significand	Total bits		Exponent bias	Bits precision	Number of decimal digits
Half (IEEE 754-2008)	1	5	10	16		15	11	~3.3
Single	1	8	23	32		127	24	~7.2
Double	1	11	52	64		1023	53	~15.9
Double extended (80-bit)	1	15	64	80		16383	64	~19.2
Quad	1	15	112	128		16383	113	~34.0

While the exponent can be positive or negative, in binary formats it is stored as an unsigned number that has a fixed "bias" added to it. Values of all 0s in this field are reserved for the zeros and subnormal numbers, values of all 1s are reserved for the infinities and NaNs. The exponent range for normalized numbers is [−126, 127] for single precision, [−1022, 1023] for double, or [−16382, 16383] for quad. Normalised numbers exclude subnormal values, zeros, infinities, and NaNs.

In the IEEE binary interchange formats the leading 1 bit of a normalized significand is not actually stored in the computer datum. It is called the "hidden" or "implicit" bit. Because of this, single precision format actually has a significand with 24 bits of precision, double precision format has 53, and quad has 113.

For example, it was shown above that π, rounded to 24 bits of precision, has:

- sign = 0 ; e = 1 ; s = 110010010000111111011011 (including the hidden bit)

The sum of the exponent bias (127) and the exponent (1) is 128, so this is represented in single precision format as

- 0 10000000 10010010000111111011011 (excluding the hidden bit) = 40490FDB[10] as a hexadecimal number.

Special values

Signed zero

In the IEEE 754 standard, zero is signed, meaning that there exist both a "positive zero" (+0) and a "negative zero" (−0). In most run-time environments, positive zero is usually printed as "0", while negative zero may be printed as "-0". The two values behave as equal in numerical comparisons, but some operations return different results for +0 and −0. For instance, 1/(−0) returns negative infinity (exactly), while 1/+0 returns positive infinity (exactly) (so that the identity 1/(1/±∞) = ±∞ is maintained). A sign symmetric arccot operation will give different results for +0 and −0 without any exception. The difference between +0 and −0 is mostly noticeable for complex operations at so-called branch cuts.

Subnormal numbers

Subnormal values fill the underflow gap with values where the absolute distance between them are the same as for adjacent values just outside of the underflow gap. This is an improvement over the older practice to just have zero in the underflow gap, and where underflowing results were replaced by zero (flush to zero).

Modern floating point hardware usually handles subnormal values (as well as normal values), and does not require software emulation for subnormals.

Infinities

The infinities of the extended real number line can be represented in IEEE floating point datatypes, just like ordinary floating point values like 1, 1.5 etc. They are not error values in any way, though they are often (but not always, as it depends on the rounding) used as replacement values when there is an overflow. Upon a divide by zero exception, a positive or negative infinity is returned as an exact result. An infinity can also be introduced as a numeral (like C's "INFINITY" macro, or "∞" if the programming language allows that syntax).

IEEE 754 requires infinities to be handled in a reasonable way, such as

- (+∞) + (+7) = (+∞)
- (+∞) × (−2) = (−∞)
- (+∞) × 0 = NaN − there is no meaningful thing to do

NaNs

IEEE 754 specifies a special value called "Not a Number" (NaN) to be returned as the result of certain "invalid" operations, such as 0/0, ∞×0, or sqrt(−1). In general, NaNs will be propagated i.e. most operations involving a NaN will result in a NaN, although functions that would give some defined result for any given floating point value will do so for NaNs as well, e.g. NaN ^ 0 == 1. There are two kinds of NaNs: the default *quiet* NaNs and, optionally, *signaling* NaNs. A signaling NaN in any arithmetic operation (including numerical comparisons) will cause an "invalid" exception to be signalled.

The representation of NaNs specified by the standard has some unspecified bits that could be used to encode the type or source of error; but there is no standard for that encoding. In theory, signaling NaNs could be used by a runtime system to flag uninitialised variables, or extend the floating-point numbers with other special values without slowing down the computations with ordinary values, although such extensions are not common.

IEEE 754 design rationale

William Kahan. A primary architect of the Intel 80x87 floating point coprocessor and IEEE 754 floating point standard.

It is a common misconception that the more esoteric features of the IEEE 754 standard discussed here, such as extended formats, NaN, infinities, subnormals etc., are only of interest to numerical analysts, or for advanced numerical applications; in fact the opposite is true: these features are designed to give safe robust defaults for numerically unsophisticated programmers, in addition to supporting sophisticated numerical libraries by experts. The key designer of IEEE 754, Prof. W. Kahan notes that it is incorrect to "... [deem] features of IEEE Standard 754 for Binary Floating- Point Arithmetic that ...[are] not appreciated to be features usable by none but numerical experts. The facts are quite the opposite. In 1977 those features were designed into the Intel 8087 to serve the widest possible market... . Error-analysis tells us how to design floating-point arithmetic, like IEEE Standard 754, moderately tolerant of well-meaning ignorance among programmers".[11]

- The special values such as infinity and NaN ensure that the floating point arithmetic is algebraically completed, such that every floating point operation produces a well-defined result and will not by default throw a machine interrupt or trap. Moreover, the choices of special values returned in exceptional cases were designed to give the correct answer in many cases, e.g. continued fractions such as R(z) := 7 − 3/(z − 2 − 1/(z − 7 + 10/(z − 2 − 2/(z − 3)))) will give the correct answer in all inputs under IEEE-754 arithmetic as the potential divide by zero in e.g. R(3)=4.6 is correctly handled as +infinity and so can be safely ignored.[12] As noted by Kahan, the unhandled floating point overflow exception that caused the loss of an Ariane 5 rocket would not have happened under IEEE 754 floating point.[11]
- Subnormal numbers ensure that x - y == 0 if and only if x == y, as expected, but which did not hold under earlier floating point representations.[13]
- On the design rationale of the x87 80-bit format, Prof. Kahan notes: "This Extended format is designed to be used, with negligible loss of speed, for all but the simplest arithmetic with float and double operands. For example, it should be used for scratch variables in loops that implement recurrences like polynomial evaluation, scalar products, partial and continued fractions. It often averts premature Over/Underflow or severe local cancellation that can spoil simple algorithms.[14] Computing intermediate results in an extended format with high precision and extended exponent has precedents in the historical practice of scientific calculation and in the design of scientific calculators e.g. Hewlett- Packard's financial calculators performed arithmetic and financial functions to three more significant decimals than they stored or displayed.[14] The implementation of extended precision enabled standard elementary function libraries to be readily developed that normally gave double precision results within one unit in the last place (ULP) at high speed.
- Correct rounding of values to the nearest representable value avoids systematic biases in calculations and slows the growth of errors. Rounding ties to even removes the statistical bias that can occur in adding similar figures.
- Directed rounding was intended as an aid with checking error bounds, for instance in interval arithmetic. It is also used in the implementation of some functions.
- The mathematical basis of the operations enabled high precision multiword arithmetic subroutines to be built relatively easily.
- The single and double precision formats were designed to be easy to sort without using floating point hardware.

Representable numbers, conversion and rounding

By their nature, all numbers expressed in floating-point format are rational numbers with a terminating expansion in the relevant base (for example, a terminating decimal expansion in base-10, or a terminating binary expansion in base-2). Irrational numbers, such as π or √2, or non-terminating rational numbers, must be approximated. The number of digits (or bits) of precision also limits the set of rational numbers that can be represented exactly. For example, the number 123456789 cannot be exactly represented if only eight decimal digits of precision are available.

When a number is represented in some format (such as a character string) which is not a native floating-point representation supported in a computer implementation, then it will require a conversion before it can be used in that implementation. If the number can be represented exactly in the floating-point format then the conversion is exact. If there is not an exact representation then the conversion requires a choice of which floating-point number to use to represent the original value. The representation chosen will have a different value to the original, and the value thus adjusted is called the *rounded value*.

Whether or not a rational number has a terminating expansion depends on the base. For example, in base-10 the number 1/2 has a terminating expansion (0.5) while the number 1/3 does not (0.333...). In base-2 only rationals with denominators that are powers of 2 (such as 1/2 or 3/16) are terminating. Any rational with a denominator that has a prime factor other than 2 will have an infinite binary expansion. This means that numbers which appear to be short and exact when written in decimal format may need to be approximated when converted to binary floating-point. For example, the decimal number 0.1 is not representable in binary floating-point of any finite precision; the exact binary representation would have a "1100" sequence continuing endlessly:

$e = -4$; $s = 1100110011001100110011001100110011...$,

where, as previously, s is the significand and e is the exponent.

When rounded to 24 bits this becomes

$e = -4$; $s = 110011001100110011001101$,

which is actually 0.100000001490116119384765625 in decimal.

As a further example, the real number π, represented in binary as an infinite series of bits is

11.0010010000111111011010101000100010000101101000110000100011010011...

but is

11.00100100001111110110

when approximated by rounding to a precision of 24 bits.

In binary single-precision floating-point, this is represented as $s = 1.00100100001111110110011$ with $e = 1$. This has a decimal value of

3.1415927410125732421875,

whereas a more accurate approximation of the true value of π is

3.14159265358979323846264338327950...

The result of rounding differs from the true value by about 0.03 parts per million, and matches the decimal representation of π in the first 7 digits. The difference is the discretization error and is limited by the machine epsilon.

The arithmetical difference between two consecutive representable floating-point numbers which have the same exponent is called a unit in the last place (ULP). For example, if there is no representable number lying between the representable numbers $1.45a70c22_{hex}$ and $1.45a70c24_{hex}$, the ULP is 2×16^{-8}, or 2^{-31}. For numbers with a base-2 exponent part of 0, i.e. numbers with an absolute value higher than or equal to 1 but lower than 2, an ULP is exactly 2^{-23} or about 10^{-7} in single precision, and exactly 2^{-53} or about 10^{-16} in double precision. The mandated behavior of IEEE-compliant hardware is that the result be within one-half of a ULP.

Rounding modes

Rounding is used when the exact result of a floating-point operation (or a conversion to floating-point format) would need more digits than there are digits in the significand. IEEE 754 requires *correct rounding*: that is, the rounded result is as if infinitely precise arithmetic was used to compute the value and then rounded (although in implementation only three extra bits are needed to ensure this). There are several different rounding schemes (or *rounding modes*). Historically, truncation was the typical approach. Since the introduction of IEEE 754, the default method (*round to nearest, ties to even*, sometimes called Banker's Rounding) is more commonly used. This method rounds the ideal (infinitely precise) result of an arithmetic operation to the nearest representable value, and gives that representation as the result.[15] In the case of a tie, the value that would make the significand end in an even digit is chosen. The IEEE 754 standard requires the same rounding to be applied to all fundamental algebraic operations, including square root and conversions, when there is a numeric (non-NaN) result. It means that the results of IEEE 754 operations are completely determined in all bits of the result, except for the representation of NaNs. ("Library" functions such as cosine and log are not mandated.)

Alternative rounding options are also available. IEEE 754 specifies the following rounding modes:

- round to nearest, where ties round to the nearest even digit in the required position (the default and by far the most common mode)
- round to nearest, where ties round away from zero (optional for binary floating-point and commonly used in decimal)
- round up (toward +∞; negative results thus round toward zero)
- round down (toward −∞; negative results thus round away from zero)
- round toward zero (truncation; it is similar to the common behavior of float-to-integer conversions, which convert −3.9 to −3 and 3.9 to 3)

Alternative modes are useful when the amount of error being introduced must be bounded. Applications that require a bounded error are multi-precision floating-point, and interval arithmetic. The alternative rounding modes are also useful in diagnosing numerical instability: if the results of a subroutine vary substantially between rounding to + and - infinity then it is likely numerically unstable and affected by round-off error.[16] A further use of rounding is when a number is explicitly rounded to a certain number of decimal (or binary) places, as when rounding a result to euros and cents (two decimal places).

Floating-point arithmetic operations

For ease of presentation and understanding, decimal radix with 7 digit precision will be used in the examples, as in the IEEE 754 *decimal32* format. The fundamental principles are the same in any radix or precision, except that normalization is optional (it does not affect the numerical value of the result). Here, *s* denotes the significand and *e* denotes the exponent.

Addition and subtraction

A simple method to add floating-point numbers is to first represent them with the same exponent. In the example below, the second number is shifted right by three digits, and we then proceed with the usual addition method:

```
123456.7 = 1.234567 × 10^5
101.7654 = 1.017654 × 10^2 = 0.001017654 × 10^5
```

```
Hence:
123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
                    = (1.234567 × 10^5) + (0.001017654 × 10^5)
                    = (1.234567 + 0.001017654) × 10^5
                    =  1.235584654 × 10^5
```

In detail:

```
  e=5;  s=1.234567     (123456.7)
+ e=2;  s=1.017654     (101.7654)
```

```
  e=5;  s=1.234567
+ e=5;  s=0.001017654  (after shifting)
--------------------
  e=5;  s=1.235584654  (true sum: 123558.4654)
```

This is the true result, the exact sum of the operands. It will be rounded to seven digits and then normalized if necessary. The final result is

```
  e=5;  s=1.235585    (final sum: 123558.5)
```

Note that the low 3 digits of the second operand (654) are essentially lost. This is round-off error. In extreme cases, the sum of two non-zero numbers may be equal to one of them:

```
  e=5;  s=1.234567
+ e=−3; s=9.876543
```

```
  e=5;  s=1.234567
+ e=5;  s=0.00000009876543 (after shifting)
----------------------
  e=5;  s=1.23456709876543 (true sum)
  e=5;  s=1.234567         (after rounding/normalization)
```

Note that in the above conceptual examples it would appear that a large number of extra digits would need to be provided by the adder to ensure correct rounding: in fact for binary addition or subtraction using careful implementation techniques only two extra *guard* bits and one extra *sticky* bit need to be carried beyond the precision of the operands.[17]

Another problem of loss of significance occurs when two close numbers are subtracted. In the following example $e = 5$; $s = 1.234571$ and $e = 5$; $s = 1.234567$ are representations of the rationals 123457.1467 and 123456.659.

```
  e=5;  s=1.234571
− e=5;  s=1.234567
----------------
  e=5;  s=0.000004
  e=−1; s=4.000000 (after rounding/normalization)
```

The best representation of this difference is $e = -1$; $s = 4.877000$, which differs more than 20% from $e = -1$; $s = 4.000000$. In extreme cases, all significant digits of precision can be lost (although gradual underflow ensures that the result will not be zero unless the two operands were equal). This *cancellation* illustrates the danger in assuming that all of the digits of a computed result are meaningful. Dealing with the consequences of these errors is a topic in numerical analysis; see also Accuracy problems.

Multiplication and division

To multiply, the significands are multiplied while the exponents are added, and the result is rounded and normalized.

```
  e=3;  s=4.734612
× e=5;  s=5.417242
-----------------------
  e=8;  s=25.648538980104 (true product)
  e=8;  s=25.64854        (after rounding)
  e=9;  s=2.564854        (after normalization)
```

Similarly, division is accomplished by subtracting the divisor's exponent from the dividend's exponent, and dividing the dividend's significand by the divisor's significand.

There are no cancellation or absorption problems with multiplication or division, though small errors may accumulate as operations are performed in succession.[18] In practice, the way these operations are carried out in digital logic can be quite complex (see Booth's multiplication algorithm and Division algorithm).[19] For a fast, simple method, see the Horner method.

Dealing with exceptional cases

Floating-point computation in a computer can run into three kinds of problems:

- An operation can be mathematically undefined, such as ∞/∞, or division by zero.
- An operation can be legal in principle, but not supported by the specific format, for example, calculating the square root of −1 or the inverse sine of 2 (both of which result in complex numbers).
- An operation can be legal in principle, but the result can be impossible to represent in the specified format, because the exponent is too large or too small to encode in the exponent field. Such an event is called an overflow (exponent too large), underflow (exponent too small) or denormalization (precision loss).

Prior to the IEEE standard, such conditions usually caused the program to terminate, or triggered some kind of trap that the programmer might be able to catch. How this worked was system-dependent, meaning that floating-point programs were not portable. (Note that the term "exception" as used in IEEE-754 is a general term meaning an exceptional condition, which is not necessarily an error, and is a different usage to that typically defined in programming languages such as a C++ or Java, in which an "exception" is an alternative flow of control, closer to what is termed a "trap" in IEEE-754 terminology).

Here, the required default method of handling exceptions according to IEEE 754 is discussed (the IEEE-754 optional trapping and other "alternate exception handling" modes are not discussed). Arithmetic exceptions are (by default) required to be recorded in "sticky" status flag bits. That they are "sticky" means that they are not reset by the next (arithmetic) operation, but stay set until explicitly reset. The use of "sticky" flags thus allows for testing of exceptional conditions to be delayed until after a full floating point expression or subroutine: without them exceptional conditions that could not be otherwise ignored would require explicit testing immediately after every floating point operation. By default, an operation always returns a result according to specification without interrupting computation. For instance, 1/0 returns +∞, while also setting the divide-by-zero flag bit (this default of ∞ is designed so as to often return a finite result when used in subsequent operations and so be safely ignored).

The original IEEE 754 standard, however, failed to recommend operations to handle such sets of arithmetic exception flag bits. So while these were implemented in hardware, initially programming language implementations typically did not provide a means to access them (apart from assembler). Over time some programming language standards (e.g., C99/C11 and Fortran) have been updated to specify methods to access and change status flag bits. The 2008 version of the IEEE 754 standard now specifies a few operations for accessing and handling the arithmetic flag bits. The programming model is based on a single thread of execution and use of them by multiple threads has to

be handled by a means outside of the standard (e.g. C11 specifies that the flags have thread-local storage).

IEEE 754 specifies five arithmetic exceptions that are to be recorded in the status flags ("sticky bits"):

- **inexact**, set if the rounded (and returned) value is different from the mathematically exact result of the operation.
- **underflow**, set if the rounded value is tiny (as specified in IEEE 754) *and* inexact (or maybe limited to if it has denormalisation loss, as per the 1984 version of IEEE 754), returning a subnormal value including the zeros.
- **overflow**, set if the absolute value of the rounded value is too large to be represented. An infinity or maximal finite value is returned, depending on which rounding is used.
- **divide-by-zero**, set if the result is infinite given finite operands, returning an infinity, either +∞ or −∞.
- **invalid**, set if a real-valued result cannot be returned e.g. sqrt(−1) or 0/0, returning a quiet NaN.

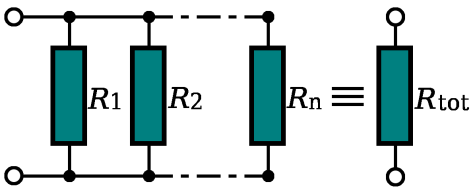


Fig. 1: resistances in parallel, with total resistance R_{tot}

The default return value for each of the exceptions is designed to give the correct result in the majority of cases such that the exceptions can be ignored in the majority of codes. *inexact* returns a correctly rounded result, and *underflow* returns a denormalised small value and so can almost always be ignored.[20] *divide-by-zero* returns infinity exactly, which will typically then divide a finite number and so give zero, or else will give an *invalid* exception subsequently if not, and so can also typically be ignored. For example, the effective resistance of three resistors in parallel (see fig. 1) is given by $R_{tot} = 1/(1/R_1 + 1/R_2 + .. + 1/R_n)$. If a short-circuit develops with R_1 set to 0, $1/R_1$ will return +infinity which will give a final R_{tot} of 0, as expected[21] (see the continued fraction example of IEEE 754 design rationale for another example). *Overflow* and *invalid* exceptions can typically not be ignored, but do not necessarily represent errors: for example, a root-finding routine, as part of its normal operation, may evaluate a passed-in function at values outside of its domain, returning NaN and an *invalid* exception flag to be ignored until finding a useful start point.[22]

Accuracy problems

The fact that floating-point numbers cannot precisely represent all real numbers, and that floating-point operations cannot precisely represent true arithmetic operations, leads to many surprising situations. This is related to the finite precision with which computers generally represent numbers.

For example, the non-representability of 0.1 and 0.01 (in binary) means that the result of attempting to square 0.1 is neither 0.01 nor the representable number closest to it. In 24-bit (single precision) representation, 0.1 (decimal) was given previously as $e = -4$; $s = 11001100110011001100110$, which is

0.100000001490116119384765625 exactly.

Squaring this number gives

0.010000000298023226097399174250313080847263336181640625 exactly.

Squaring it with single-precision floating-point hardware (with rounding) gives

0.010000000707805156707763671875 exactly.

But the representable number closest to 0.01 is

0.009999999776482582092285156250 exactly.

James H. Wilkinson, a pioneer in numerical analysis, demonstrated that floating point algorithms could be rigorously analysed.

Also, the non-representability of π (and π/2) means that an attempted computation of tan(π/2) will not yield a result of infinity, nor will it even overflow. It is simply not possible for standard floating-point hardware to attempt to compute tan(π/2), because π/2 cannot be represented exactly. This computation in C:

```
/* Enough digits to be sure we get the correct approximation. */
double pi = 3.14159265358979323846264338327795;
double z = tan(pi/2.0);
```

will give a result of 16331239353195370.0. In single precision (using the tanf function), the result will be −22877332.0.

By the same token, an attempted computation of sin(π) will not yield zero. The result will be (approximately) 0.1225×10^{-15} in double precision, or -0.8742×10^{-7} in single precision.[23]

While floating-point addition and multiplication are both commutative ($a + b = b + a$ and $a\times b = b\times a$), they are not necessarily associative. That is, $(a + b) + c$ is not necessarily equal to $a + (b + c)$. Using 7-digit mantissa decimal arithmetic:

```
a = 1234.567, b = 45.67834, c = 0.0004
```

```
(a + b) + c:
    1234.567   (a)
  +   45.67834 (b)
  ______________
    1280.24534   rounds to   1280.245
```

```
   1280.245  (a + b)
  +   0.0004 (c)
```

```
  ____________
   1280.2454   rounds to   1280.245  <--- (a + b) + c
```

```
a + (b + c):
  45.67834 (b)
+  0.0004  (c)
____________
  45.67874
```

```
  1234.567   (a)
+   45.67874 (b + c)
____________
  1280.24574   rounds to   1280.246 <--- a + (b + c)
```

They are also not necessarily distributive. That is, $(a + b) \times c$ may not be the same as $a \times c + b \times c$:

```
1234.567 × 3.333333 = 4115.223
1.234567 × 3.333333 = 4.115223
                      4115.223 + 4.115223 = 4119.338
but
1234.567 + 1.234567 = 1235.802
                      1235.802 × 3.333333 = 4119.340
```

In addition to loss of significance, inability to represent numbers such as π and 0.1 exactly, and other slight inaccuracies, the following phenomena may occur:

- Cancellation: subtraction of nearly equal operands may cause extreme loss of accuracy.[24] When we subtract two almost equal numbers we set the most significant digits to zero, leaving ourselves with just the insignificant, and most erroneous, digits. For example, when determining a derivative of a function the following formula is used:

  $$Q(h) = \frac{f(a+h) - f(a)}{h}.$$

  Intuitively one would want an h very close to zero, however when using floating point operations, the smallest number won't give the best approximation of a derivative. As h grows smaller the difference between f (a + h) and f(a) grows smaller, cancelling out the most significant and least erroneous digits and making the most erroneous digits more important. As a result the smallest number of h possible will give a more erroneous approximation of a derivative than a somewhat larger number. This is perhaps the most common and serious accuracy problem.
- Conversions to integer are not intuitive: converting (63.0/9.0) to integer yields 7, but converting (0.63/0.09) may yield 6. This is because conversions generally truncate rather than round. Floor and ceiling functions may produce answers which are off by one from the intuitively expected value.
- Limited exponent range: results might overflow yielding infinity, or underflow yielding a subnormal number or zero. In these cases precision will be lost.
- Testing for safe division is problematic: Checking that the divisor is not zero does not guarantee that a division will not overflow.
- Testing for equality is problematic. Two computational sequences that are mathematically equal may well produce different floating-point values.

Machine precision and backward error analysis

Machine precision is a quantity that characterizes the accuracy of a floating point system, and is used in backward error analysis of floating point algorithms. It is also known as unit roundoff or *machine epsilon*. Usually denoted $\mathrm{E}_{\mathrm{mach}}$, its value depends on the particular rounding being used.

With rounding to zero,

$$\mathrm{E}_{\mathrm{mach}} = B^{1-P},$$

whereas rounding to nearest,

$$\mathrm{E}_{\mathrm{mach}} = \tfrac{1}{2}B^{1-P}.$$

This is important since it bounds the *relative error* in representing any non-zero real number x within the normalised range of a floating point system:

$$\left|\frac{fl(x) - x}{x}\right| \leq \mathrm{E}_{\mathrm{mach}}.$$

Backward error analysis, popularized by James H. Wilkinson, can be used to establish that an algorithm implementing a numerical function is numerically stable. The basic approach is to show that although the calculated result, due to roundoff errors, will not be exactly correct, it is the exact solution to a nearby problem with slightly perturbed input data. If the perturbation required is small, on the order of the uncertainty in the input data, then the results are in some sense as accurate as the data "deserves". The algorithm is then defined as *backward stable*.

As a trivial example, consider a simple expression giving the inner product of (length two) vectors x and y, then

$$fl(x \cdot y) = fl(fl(x_1 * y_1) + fl(x_2 * y_2)) \text{ where } fl() \text{ indicates correctly rounded floating point arithmetic}$$
$$= fl((x_1 * y_1)(1+\delta_1) + (x_2 * y_2)(1+\delta_2)) \text{ where } \delta_n \leq \mathrm{E}_{\mathrm{mach}}, \text{ from above}$$
$$= ((x_1 * y_1)(1+\delta_1) + (x_2 * y_2)(1+\delta_2))(1+\delta_3)$$
$$= (x_1 * y_1)(1+\delta_1)(1+\delta_3) + (x_2 * y_2)(1+\delta_2)(1+\delta_3)$$

and so

$$fl(x \cdot y) = \hat{x} \cdot \hat{y} \text{ where}$$
$$\hat{x}_1 = x_1(1+\delta_1);\ \hat{x}_2 = x_2(1+\delta_2);$$
$$\hat{y}_1 = y_1(1+\delta_3);\ \hat{y}_2 = y_2(1+\delta_3)$$
$$\text{where } \delta_n \leq \mathrm{E}_{\mathrm{mach}}, \text{ by definition}$$

which is the sum of two slightly perturbed (on the order of $\mathrm{E}_{\mathrm{mach}}$) input data, and so is backward stable. More realistic examples require estimating the condition number of the function (see Higham 2002 and other references below).

Minimizing the effect of accuracy problems

Although, as noted previously, individual arithmetic operations of IEEE 754 are guaranteed accurate to within half a ULP, more complicated formulae can suffer from larger errors due to round-off. The loss of accuracy can be substantial if a problem or its data are ill-conditioned, meaning that the correct result is hypersensitive to tiny perturbations in its data. However, even functions that are well-conditioned can suffer from large loss of accuracy if an algorithm numerically unstable for that data is used: apparently equivalent formulations of expressions in a programming language can differ markedly in their numerical stability. One approach to remove the risk of such loss of accuracy is the design and analysis of numerically stable algorithms, which is an aim of the branch of mathematics known as numerical analysis. Another approach that can protect against the risk of numerical instabilities is the computation of intermediate (scratch) values in an algorithm at a higher precision than the final result requires, which can remove, or reduce by orders of magnitude, such risk: IEEE 754 quadruple precision and

extended precision are designed for this purpose when computing at double precision.[25][26]

For example, the following algorithm is a direct implementation to compute the function A(x) = (x−1)/(exp(x−1) − 1) which is well-conditioned at 1.0,[27] however it can be shown to be numerically unstable and lose up to half the significant digits carried by the arithmetic when computed near 1.0.[11]

```
double A(double X)
{
        double  Y, Z;  // [1]
        Y = X - 1.0;
        Z = exp(Y);
        if (Z != 1.0) Z = Y/(Z - 1.0); // [2]
        return(Z);
}
```

If, however, intermediate computations are all performed in extended precision (e.g. by setting line [1] to C99 long double), then up to full precision in the final double result can be maintained.[28] Alternatively, a numerical analysis of the algorithm reveals that if the following non-obvious change to line [2] is made:

```
 if (Z != 1.0) Z = log(Z)/(Z - 1.0);
```

then the algorithm becomes numerically stable and can compute to full double precision.

To maintain the properties of such carefully constructed numerically stable programs, careful handling by the compiler is required. Certain "optimizations" that compilers might make (for example, reordering operations) can work against the goals of well-behaved software. There is some controversy about the failings of compilers and language designs in this area: C99 is an example of a language where such optimisations are carefully specified so as to maintain numerical precision. See the external references at the bottom of this article.

A detailed treatment of the techniques for writing high-quality floating-point software is beyond the scope of this article, and the reader is referred to,[29][30] and the other references at the bottom of this article. Kahan suggests several rules of thumb that can substantially decrease by orders of magnitude[30] the risk of numerical anomalies, in addition to, or in lieu of, a more careful numerical analysis. These include: as noted above, computing all expressions and intermediate results in the highest precision supported in hardware (a common rule of thumb is to carry twice the precision of the desired result i.e. compute in double precision for a final single precision result, or in double extended or quad precision for up to double precision results[31]); and rounding input data and results to only the precision required and supported by the input data (carrying excess precision in the final result beyond that required and supported by the input data can be misleading, increases storage cost and decreases speed, and the excess bits can affect convergence of numerical procedures:[32] notably, the first form of the iterative example given below converges correctly when using this rule of thumb). Brief descriptions of several additional issues and techniques follow.

As decimal fractions can often not be exactly represented in binary floating-point, such arithmetic is at its best when it is simply being used to measure real-world quantities over a wide range of scales (such as the orbital period of a moon around Saturn or the mass of a proton), and at its worst when it is expected to model the interactions of quantities expressed as decimal strings that are expected to be exact.[33][34] An example of the latter case is financial calculations. For this reason, financial software tends not to use a binary floating-point number representation.[35] The "decimal" data type of the C# and Python programming languages, and the IEEE 754-2008 decimal floating-point standard, are designed to avoid the problems of binary floating-point representations when applied to human-entered exact decimal values, and make the arithmetic always behave as expected when numbers are printed in decimal.

Expectations from mathematics may not be realised in the field of floating-point computation. For example, it is known that $(x+y)(x-y) = x^2 - y^2$, and that $\sin^2\theta + \cos^2\theta = 1$, however these facts cannot be relied on when the quantities involved are the result of floating-point computation.

The use of the equality test (`if (x==y) ...`) requires care when dealing with floating point numbers. Even simple expressions like `0.6/0.2-3==0` will, on most computers, fail to be true[36] (in IEEE 754 double precision, for example, `0.6/0.2-3` is approximately equal to -4.44089209850063e-16). Consequently, such tests are sometimes replaced with "fuzzy" comparisons (`if (abs(x-y) < epsilon) ...`, where epsilon is sufficiently small and tailored to the application, such as 1.0E−13). The wisdom of doing this varies greatly, and can require numerical analysis to bound epsilon.[37] Values derived from the primary data representation and their comparisons should be performed in a wider, extended, precision to minimise the risk of such inconsistencies due to round-off errors.[30] It is often better to organize the code in such a way that such tests are unnecessary. For example, in computational geometry, exact tests of whether a point lies off or on a line or plane defined by other points can be performed using adaptive precision or exact arithmetic methods.[38]

Small errors in floating-point arithmetic can grow when mathematical algorithms perform operations an enormous number of times. A few examples are matrix inversion, eigenvector computation, and differential equation solving. These algorithms must be very carefully designed, using numerical approaches such as Iterative refinement, if they are to work well.[39]

Summation of a vector of floating point values is a basic algorithm in scientific computing, and so an awareness of when loss of significance can occur is essential. For example, if one is adding a very large number of numbers, the individual addends are very small compared with the sum. This can lead to loss of significance. A typical addition would then be something like

```
3253.671
+  3.141276
--------
3256.812
```

The low 3 digits of the addends are effectively lost. Suppose, for example, that one needs to add many numbers, all approximately equal to 3. After 1000 of them have been added, the running sum is about 3000; the lost digits are not regained. The Kahan summation algorithm may be used to reduce the errors.[40]

Round-off error can affect the convergence and accuracy of iterative numerical procedures. As an example, Archimedes approximated π by calculating the perimeters of polygons inscribing and circumscribing a circle, starting with hexagons, and successively doubling the number of sides. As noted above, computations may be rearranged in a way that is mathematically equivalent but less prone to error (numerical analysis). Two forms of the recurrence formula for the circumscribed polygon are:

$$t_0 = \frac{1}{\sqrt{3}}$$

$$\text{first form}: \quad t_{i+1} = \frac{\sqrt{t_i^2+1}-1}{t_i} \qquad \text{second form}: \quad t_{i+1} = \frac{t_i}{\sqrt{t_i^2+1}+1}$$

$$\pi \sim 6 \times 2^i \times t_i, \qquad \text{converging as i} \to \infty$$

Here is a computation using IEEE "double" (a significand with 53 bits of precision) arithmetic:

i	$6 \times 2^i \times t_i$, first form	$6 \times 2^i \times t_i$, second form
0	3.4641016151377543863	3.4641016151377543863
1	3.2153903091734710173	3.2153903091734723496
2	3.1596599420974940120	3.1596599420975006733

```
 3   3.1460862151314012979      3.1460862151314352708
 4   3.1427145996453136334      3.1427145996453689225
 5   3.1418730499801259536      3.1418730499798241950
 6   3.1416627470548084133      3.1416627470568494473
 7   3.1416101765997805905      3.1416101766046906629
 8   3.1415970343230776862      3.1415970343215275928
 9   3.1415937488171150615      3.1415937487713536668
10   3.1415929278733740748      3.1415929273850979885
11   3.1415927256228504127      3.1415927220386148377
12   3.1415926717412858693      3.1415926707019992125
13   3.1415926189011456060      3.1415926578678454728
14   3.1415926717412858693      3.1415926546593073709
15   3.1415919358822321783      3.1415926538571730119
16   3.1415926717412858693      3.1415926536566394222
17   3.1415810075796233302      3.1415926536065061913
18   3.1415926717412858693      3.1415926535939728836
19   3.1414061547378810956      3.1415926535908393901
20   3.1405434924008406305      3.1415926535900560168
21   3.1400068646912273617      3.1415926535898608396
22   3.1349453756585929919      3.1415926535898122118
23   3.1400068646912273617      3.1415926535897995552
24   3.2245152435345525443      3.1415926535897968907
25                              3.1415926535897962246
26                              3.1415926535897962246
27                              3.1415926535897962246
28                              3.1415926535897962246
              The true value is 3.1415926535897932384626433832 7...
```

While the two forms of the recurrence formula are clearly mathematically equivalent,[41] the first subtracts 1 from a number extremely close to 1, leading to an increasingly problematic loss of significant digits. As the recurrence is applied repeatedly, the accuracy improves at first, but then it deteriorates. It never gets better than about 8 digits, even though 53-bit arithmetic should be capable of about 16 digits of precision. When the second form of the recurrence is used, the value converges to 15 digits of precision.

Notes and references

[1] W.Smith, Steven (1997). "Chapter 28, Fixed versus Floating Point" (http://www.dspguide.com/ch28/4.htm). *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub. p. 514. ISBN 0966017633. . Retrieved December 31, 2012.

[2] B. Randell (1982). *From analytical engine to electronic digital computer: the contributions of Ludgate, Torres, and Bush. IEEE Annals of the History of Computing, 04(4)*. pp. 327–341.

[3] "Konrad Zuse's Legacy: The Architecture of the Z1 and Z3" (http://ed-thelen.org/comp-hist/Zuse_Z1_and_Z3.pdf). *IEEE Annals of the History of Computing* **19** (2): 5–15. 1997. .

[4] William Kahan (15 July 1997). "The Baleful Effect of Computer Languages and Benchmarks upon Applied Mathematics, Physics and Chemistry" (http://www.cs.berkeley.edu/~wkahan/SIAMjvnl.pdf). .

[5] "The Baleful Effect of Computer Languages and Benchmarks upon Applied Mathematics, Physics and Chemistry. John von Neumann Lecture" (http://www.cs.berkeley.edu/~wkahan/SIAMjvnl.pdf). 16 July 1997. p. 3. .

[6] Randell, Brian, ed. (1982) [1973]. *The Origins of Digital Computers: Selected Papers* (3rd ed.). Berlin; New York: Springer-Verlag. p. 244. ISBN 3-540-11319-3.

[7] Severance, Charles (20 February 1998). "An Interview with the Old Man of Floating-Point" (http://www.eecs.berkeley.edu/~wkahan/ieee754status/754story.html). .

[8] "W. Kahn. "On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic"" (http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf) (PDF). 20 November 2004. . Retrieved 19 February 2012.
[9] "openEXR" (http://www.openexr.com/about.html). openEXR. . Retrieved 25 April 2012.
[10] http://babbage.cs.qc.edu/IEEE-754/32bit.html
[11] William Kahan (1 March 1998). "How JAVA's Floating-Point Hurts Everyone Everywhere" (http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf). .
[12] William Kahan (12 February 1981). "Why do we need a floating-point arithmetic standard?" (http://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf). .
[13] Charles Severance (20 February 1998). "An Interview with the Old Man of Floating-Point" (http://www.eecs.berkeley.edu/~wkahan/ieee754status/754story.html). .
[14] William Kahan (11 June 1996). "The Baleful Effect of Computer Benchmarks upon Applied Mathematics, Physics and Chemistry" (http://www.cs.berkeley.edu/~wkahan/ieee754status/baleful.pdf). .
[15] Computer hardware doesn't necessarily compute the exact value; it simply has to produce the equivalent rounded result as though it had computed the infinitely precise result.
[16] William Kahan (11 January 2006). "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?" (http://www.cs.berkeley.edu/~wkahan/Mindless.pdf). .
[17] David Goldberg (March 1991). "What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, volume 23, issue 1" (http://www.validlab.com/goldberg/paper.pdf). p. 195. .
[18] Goldberg, David (1991). "What Every Computer Scientist Should Know About Floating-Point Arithmetic" (http://docs.sun.com/source/806-3568/ncg_goldberg.html). *ACM Computing Surveys* **23**: 5–48. doi:10.1145/103162.103163. . Retrieved 2 September 2010.
[19] The enormous complexity of modern division algorithms once led to a famous error. An early version of the Intel Pentium chip was shipped with a division instruction that, on rare occasions, gave slightly incorrect results. Many computers had been shipped before the error was discovered. Until the defective computers were replaced, patched versions of compilers were developed that could avoid the failing cases. See *Pentium FDIV bug*.
[20] William Kahan (1 October 1997). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic" (http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF). .
[21] "Intel® 64 and IA-32 Architectures Software Developers' Manuals. Volume 1. section D.3.2.1" (http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html). .
[22] William Kahan (1 October 1997). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic (page 9)" (http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF). .
[23] But an attempted computation of cos(π) yields −1 exactly. Since the derivative is nearly zero near π, the effect of the inaccuracy in the argument is far smaller than the spacing of the floating-point numbers around −1, and the rounded result is exact.
[24] Richard Harris (October 2010). "You're Going To Have To Think!" (http://accu.org/index.php/journals/1702). *Overload* (99 (http://accu.org/var/uploads/journals/overload99.pdf)): 5–10. ISSN 1354-3172. . Retrieved 24 September 2011. "Far more worrying is cancellation error which can yield catastrophic loss of precision."
[25] William Kahan (3 August 2011). "Desperately Needed Remedies for the Undebuggability of Large Floating-Point Computations in Science and Engineering" (http://www.eecs.berkeley.edu/~wkahan/Boulder.pdf). .
[26] Kahan notes: "Except in extremely uncommon situations, extra-precise arithmetic generally attenuates risks due to roundoff at far less cost than the price of a competent error-analyst."
[27] note: the Taylor expansion of this function demonstrates that it is well-conditioned near 1: A(x) = 1 − (x−1)/2 + (x−1)^2/12 − (x−1)^4/720 + (x−1)^6/30240 − (x−1)^8/1209600 + ... for |x−1| < π
[28] if long double is IEEE quad precision then full double precision is retained; if long double is IEEE double extended precision then additional, but not full, precision is retained
[29] Higham, Nicholas (2002). *"Designing stable algorithms" in Accuracy and Stability of Numerical Algorithms (2 ed)*. SIAM. pp. 27–28.
[30] William Kahan. ""Four Rules of Thumb for Best Use of Modern Floating-point Hardware" in Marketing versus Mathematics" (http://www.cs.berkeley.edu/~wkahan/MktgMath.pdf). p. 47. .
[31] William Kahan (12 February 1981). "Why do we need a floating-point arithmetic standard? (page 26)" (http://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf). .
[32] William Kahan (transcribed by David Bindel) (4 June 2001). "Lecture notes of System Support for Scientific Computation" (http://www.cims.nyu.edu/~dbindel/class/cs279/notes-06-04.pdf). .
[33] Prof. W. Kahan (27 August 2000). "Marketing versus Mathematics (p 15)" (http://www.cs.berkeley.edu/~wkahan/MktgMath.pdf). .
[34] Prof. W. Kahan (5 July 2005). "Floating-Point Arithmetic Besieged by "Business Decisions": Keynote Address for the IEEE-Sponsored ARITH 17 Symposium on Computer Arithmetic" (http://www.cs.berkeley.edu/~wkahan/ARITH_17.pdf). p. 6. .
[35] "General Decimal Arithmetic" (http://speleotrove.com/decimal/). Speleotrove.com. . Retrieved 25 April 2012.
[36] Tom Christiansen, Nathan Torkington, and others (2006). "perlfaq4 / Why is int() broken?" (http://perldoc.perl.org/5.8.8/perlfaq4.html#Why-is-int()-broken?). perldoc.perl.org. . Retrieved 11 January 2011.
[37] Higham, Nicholas (2002). *"Subtleties of floating point arithmetic" in Accuracy and Stability of Numerical Algorithms (2 ed)*. SIAM. p. 493.
[38] Jonathan Richard Shewchuk (1997). *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, Discrete & Computational Geometry 18:305-363* (http://www.cs.cmu.edu/~quake/robust.html). .

[39] Prof. W. Kahan and Ms. Melody Y. Ivory (3 July 1997). "Roundoff Degrades an Idealized Cantilever" (http://www.cs.berkeley.edu/~wkahan/Cantilever.pdf). .

[40] Higham, Nicholas (2002). *Summation in "Subtleties of floating point arithmetic" in Accuracy and Stability of Numerical Algorithms (2 ed)*. SIAM. pp. 110–123.

[41] The equivalence of the two forms can be verified algebraically by noting that the denominator of the fraction in the second form is the conjugate of the numerator of the first. By multiplying the top and bottom of the first expression by this conjugate, one obtains the second expression.

Further reading

- *What Every Computer Scientist Should Know About Floating-Point Arithmetic* (http://download.oracle.com/docs/cd/E19422-01/819-3693/ncg_goldberg.html), by David Goldberg, published in the March, 1991 issue of Computing Surveys.
- Nicholas Higham. *Accuracy and Stability of Numerical Algorithms*, Second Edition. SIAM, 2002. ISBN 0-89871-355-2.
- Gene F. Golub and Charles F. van Loan. *Matrix Computations*, Third Edition. Johns Hopkins University Press, 1986. ISBN 0-8018-5413.
- Donald Knuth. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89684-2. Section 4.2: Floating Point Arithmetic, pp. 214–264.
- Press et al. *Numerical Recipes in C++. The Art of Scientific Computing*, ISBN 0-521-75033-4.
- James H. Wilkinson. *Rounding errors in algebraic processes*. 1963. -- Classic influential treatises on floating point arithmetic.
- James H. Wilkinson. *The Algebraic Eigenvalue Problem*, Clarendon Press, 1965.
- P.H. Sterbenz. *Floating point computation*. 1974. -- Another classic book on floating point and error analysis.

External links

- Kahan, William and Darcy, Joseph (2001). How Java's floating-point hurts everyone everywhere (http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf). Retrieved 5 September 2003.
- Survey of Floating-Point Formats (http://www.mrob.com/pub/math/floatformats.html) This page gives a very brief summary of floating-point formats that have been used over the years.
- *The pitfalls of verifying floating-point computations* (http://hal.archives-ouvertes.fr/hal-00128124/en/), by David Monniaux, also printed in *ACM Transactions on programming languages and systems (TOPLAS)*, May 2008: a compendium of non-intuitive behaviours of floating-point on popular architectures, with implications for program verification and testing
- http://www.opencores.org The OpenCores website contains open source floating point IP cores for the implementation of floating point operators in FPGA or ASIC devices. The project, double_fpu, contains verilog source code of a double precision floating point unit. The project, fpuvhdl, contains vhdl source code of a single precision floating point unit.
- http://msdn.microsoft.com/en-us/library/aa289157(v=vs.71).aspx "Microsoft Visual C++ Floating-Point Optimization", by Eric Fleegal, MSDN, 2004

Single-precision floating-point format

Single-precision floating-point format is a computer number format that occupies 4 bytes (32 bits) in computer memory and represents a wide dynamic range of values by using a floating point.

In IEEE 754-2008 the 32-bit base 2 format is officially referred to as **binary32**. It was called **single** in IEEE 754-1985. In older computers, other floating-point formats of 4 bytes were used.

One of the first programming languages to provide single- and double-precision floating-point data types was Fortran. Before the widespread adoption of IEEE 754-1985, the representation and properties of the double float data type depended on the computer manufacturer and computer model.

Single-precision binary floating-point is used due to its wider range over fixed point (of the same bit-width), even if at the cost of precision.

Single precision is known as **float** in C, C++, C#, Java,[1] and Haskell, and as **single** in Delphi (Pascal), Visual Basic, and MATLAB. However, **float** in Python, Ruby, PHP, and OCaml and **single** in versions of Octave prior to 3.2 refer to double-precision numbers.

IEEE 754 single-precision binary floating-point format: binary32

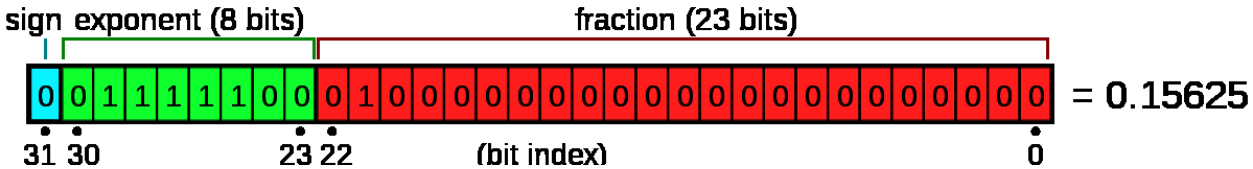
The IEEE 754 standard specifies a **binary32** as having:

- Sign bit: 1 bit
- Exponent width: 8 bits
- Significand precision: 24 (23 explicitly stored)

This gives from 6 to 9 significant decimal digits precision (if a decimal string with at most 6 significant decimal is converted to IEEE 754 single precision and then converted back to the same number of significant decimal, then the final string should match the original; and if an IEEE 754 single precision is converted to a decimal string with at least 9 significant decimal and then converted back to single, then the final number must match the original [2]).

Sign bit determines the sign of the number, which is the sign of the significand as well. Exponent is either an 8 bit signed integer from −128 to 127 (2's Complement) or an 8 bit unsigned integer from 0 to 255 which is the accepted biased form in IEEE 754 binary32 definition. For this case an exponent value of 127 represents the actual zero.

The true significand includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1 unless the exponent is stored with all zeros. Thus only 23 fraction bits of the significand appear in the memory format but the total precision is 24 bits (equivalent to $\log_{10}(2^{24}) \approx 7.225$ decimal digits). The bits are laid out as follows:



The real value assumed by a given 32 bit **binary32** data with a given biased exponent **e** and a **23 bit fraction** is $= (-1)^{\text{sign}}(1.b_{-1}b_{-2}...b_{-23})_2 \times 2^{e-127}$ where more precisely we have:

$$value = (-1)^{\text{sign}}(1 + \sum_{i=1}^{23} b_{23-i}2^{-i}) \times 2^{(e-127)}$$

In this example:

- $sign = 0$
- $1 + \sum_{i=1}^{23} b_{23-i}2^{-i} = 1 + 2^{-2} = 1.25$

- $2^{(e-127)} = 2^{124-127} = 2^{-3}$

thus:

- $value = 1.25 \times 2^{-3} = 0.15625$

Exponent encoding

The single-precision binary floating-point exponent is encoded using an offset-binary representation, with the zero offset being 127; also known as exponent bias in the IEEE 754 standard.

- $E_{min} = 01_H - 7F_H = -126$
- $E_{max} = FE_H - 7F_H = 127$
- Exponent bias = $7F_H = 127$

Thus, in order to get the true exponent as defined by the offset binary representation, the offset of 127 has to be subtracted from the stored exponent.

The stored exponents 00_H and FF_H are interpreted specially.

Exponent	Significand zero	Significand non-zero	Equation
00_H	zero, −0	subnormal numbers	$(-1)^{signbits} \times 2^{-126} \times 0.significandbits$
01_H, ..., FE_H	normalized value		$(-1)^{signbits} \times 2^{exponentbits-127} \times 1.significandbits$
FF_H	±infinity	NaN (quiet, signalling)	

The minimum positive (subnormal) value is $2^{-149} \approx 1.4 \times 10^{-45}$. The minimum positive normal value is $2^{-126} \approx 1.18 \times 10^{-38}$. The maximum representable value is $(2-2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$.

Converting from decimal representation to binary32 format

In general refer to the IEEE 754 standard itself for the strict conversion (including the rounding behaviour) of a real number into its equivalent binary32 format.

Here we can show how to convert a base 10 real number into an IEEE 754 binary32 format using the following outline:

- consider a real number with an integer and a fraction part such as 12.375
- convert and normalize the integer part into binary
- convert the fraction part using the following technique as shown here
- add the two results and adjust them to produce a proper final conversion

Conversion of the fractional part:

consider 0.375, the fractional part of 12.375. To convert it into a binary fraction, multiply the fraction by 2, take the integer part and re-multiply new fraction by 2 until a fraction of zero is found or until the precision limit is reached which is 23 fraction digits for IEEE 754 binary32 format.

0.375 x 2 = 0.750 = 0 + 0.750 => $b_{-1} = 0$, the integer part represents the binary fraction digit. Re-multiply 0.750 by 2 to proceed

0.750 x 2 = 1.500 = 1 + 0.500 => $b_{-2} = 1$

0.500 x 2 = 1.000 = 1 + 0.000 => $b_{-3} = 1$, fraction = 0.000, terminate

We see that $(0.375)_{10}$ can be exactly represented in binary as $(0.011)_2$. Not all decimal fractions can be represented in a finite digit binary fraction. For example decimal 0.1 cannot be represented in binary exactly. So it is only approximated.

Therefore $(12.375)_{10} = (12)_{10} + (0.375)_{10} = (1100)_2 + (0.011)_2 = (1100.011)_2$

Also IEEE 754 binary32 format requires that you represent real values in $(1.x_1x_2...x_{23})_2 \times 2^e$ format, (see Normalized number, Denormalized number) so that 1100.011 is shifted to the right by 3 digits to become $(1.100011)_2 \times 2^3$

Finally we can see that: $(12.375)_{10} = (1.100011)_2 \times 2^3$

From which we deduce:

- The exponent is 3 (and in the biased form it is therefore 130 = 1000 0010)
- The fraction is 100011 (looking to the right of the binary point)

From these we can form the resulting 32 bit IEEE 754 binary32 format representation of 12.375 as: 0-10000010-10001100000000000000000 = 41460000_H

Note: consider converting 68.123 into IEEE 754 binary32 format: Using the above procedure you expect to get $42883EF9_H$ with the last 4 bits being 1001 However due to the default rounding behaviour of IEEE 754 format what you get is $42883EFA_H$ whose last 4 bits are 1010 .

Ex 1: Consider decimal 1 We can see that: $(1)_{10} = (1.0)_2 \times 2^0$

From which we deduce:

- The exponent is 0 (and in the biased form it is therefore 127 = 0111 1111)
- The fraction is 0 (looking to the right of the binary point in 1.0 is all 0 = 000...0)

From these we can form the resulting 32 bit IEEE 754 binary32 format representation of real number 1 as: 0-01111111-00000000000000000000000 = $3f800000_H$

Ex 2: Consider a value 0.25 . We can see that: $(0.25)_{10} = (1.0)_2 \times 2^{-2}$

From which we deduce:

- The exponent is −2 (and in the biased form it is 127+(−2)= 125 = 0111 1101)
- The fraction is 0 (looking to the right of binary point in 1.0 is all zeros)

From these we can form the resulting 32 bit IEEE 754 binary32 format representation of real number 0.25 as: 0-01111101-00000000000000000000000 = $3e800000_H$

Ex 3: Consider a value of 0.375 . We saw that $0.375 = (1.1)_2 \times 2^{-2}$

Hence after determining a representation of 0.375 as $(1.1)_2 \times 2^{-2}$ we can proceed as above:

- The exponent is −2 (and in the biased form it is 127+(−2)= 125 = 0111 1101)
- The fraction is 1 (looking to the right of binary point in 1.1 is a single 1 = x_1)

From these we can form the resulting 32 bit IEEE 754 binary32 format representation of real number 0.375 as: 0-01111101-10000000000000000000000 = $3ec00000_H$

Single-precision examples

These examples are given in bit *representation*, in hexadecimal, of the floating-point value. This includes the sign, (biased) exponent, and significand.

```
3f80 0000   = 1
c000 0000   = −2

7f7f ffff   ≈ 3.4028234 × 10^38  (max single precision)

0000 0000   = 0
8000 0000   = −0

7f80 0000   = infinity
```

```
ff80 0000   = −infinity

3eaa aaab   ≈ 1/3
```

By default, 1/3 rounds up instead of down like double precision, because of the even number of bits in the significand. So the bits beyond the rounding point are `1010...` which is more than 1/2 of a unit in the last place.

Converting from single-precision binary to decimal

We start with the hexadecimal representation of the value, 41c80000, in this example, and convert it to binary

$$41c8\ 0000_{16} = 0100\ 0001\ 1100\ 1000\ 0000\ 0000\ 0000\ 0000_2$$

then we break it down into three parts; sign bit, exponent and significand.

$$\text{Sign bit: } 0$$
$$\text{Exponent: } 1000\ 0011_2 = 83_{16} = 131$$
$$\text{Significand: } 100\ 1000\ 0000\ 0000\ 0000\ 0000_2 = 480000_{16}$$

We then add the implicit 24th bit to the significand

$$\text{Significand: } \mathbf{1}100\ 1000\ 0000\ 0000\ 0000\ 0000_2 = C80000_{16}$$

and decode the exponent value by subtracting 127

$$\text{Raw exponent: } 83_{16} = 131$$
$$\text{Decoded exponent: } 131 - 127 = \mathbf{4}$$

Each of the 24 bits of the significand (including the implicit 24th bit), bit 23 to bit 0, represents a value, starting at 1 and halves for each bit, as follows

```
bit 23 = 1
bit 22 = 0.5
bit 21 = 0.25
bit 20 = 0.125
bit 19 = 0.0625
.
.
bit  0 = 0.000000119209289550781 25
```

The significand in this example has three bits set, bit 23, bit 22 and bit 19. We can now decode the significand by adding the values represented by these bits.

$$\text{Decoded significand: } 1 + 0.5 + 0.0625 = 1.5625 = C80000/2^{23}$$

Then we need to multiply with the base, 2, to the power of the exponent to get the final result

$$1.5625 \times 2^4 = \mathbf{25}$$

Thus

```
41c8 0000   = 25
```

This is equivalent to:

where s is the sign bit, x is the exponent, and m is the significand.

External links

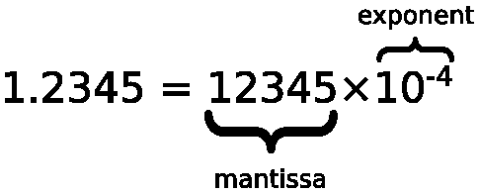
- Online calculator [3]
- Online converter for IEEE 754 numbers with single precision [4]
- C source code to convert between IEEE double, single, and half precision can be found here [5]

References

[1] http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html
[2] William Kahan (1 October 1987). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic" (http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF). .
[3] http://www.h-schmidt.net/FloatApplet/IEEE754.html
[4] http://www.binaryconvert.com/convert_float.html
[5] http://www.mathworks.com/matlabcentral/fileexchange/23173

Significand

The **significand** (also *coefficient* or *mantissa*) is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Depending on the interpretation of the exponent, the significand may represent an integer or a fraction.



A diagram showing a representation of a floating point number using a significand and an exponent.

Examples

The number 123.45 can be represented as a decimal floating-point number with an integer significand of 12345 and an exponent of −2. Its value is given by the following arithmetic:

$$12345 \times 10^{-2}$$

This same value can also be represented in normalized form with a fractional coefficient of 1.2345 and an exponent of +2:

$$1.2345 \times 10^{+2}$$

Finally, this value can be represented in the format given by the Language Independent Arithmetic standard and several programming language standards, including Ada, C, Fortran and Modula-2, as:

$$0.12345 \times 10^{+3}$$

Significands and the hidden bit

When working in binary, the significand is characterized by its width in binary digits (bits). Because the most significant bit is always 1 for a normalized number, this bit is not typically stored and is called the "hidden bit". Depending on the context, the hidden bit may or may not be counted towards the width of the significand. For example, the same IEEE 754 double precision format is commonly described as having either a 53-bit significand, including the hidden bit, or a 52-bit significand, not including the hidden bit. The notion of a hidden bit only applies to binary representations. IEEE 754 defines the precision, p, to be the number of digits in the significand, including any implicit leading bit (e.g. precision, p, of double precision format is 53).

Use of "mantissa"

In American English, the original word for this seems to have been *mantissa* (Burks *et al.*), and as of 2005 this usage remains common in computing and among computer scientists. However, this use of *mantissa* is discouraged by the IEEE floating-point standard committee and by some professionals such as William Kahan and Donald Knuth, because it conflicts with the pre-existing use of *mantissa* for the fractional part of a logarithm (see also common logarithm).

The confusion is because scientific notation and floating point are log-linear representations, not logarithmic. To multiply two numbers, given their logarithms, one just adds them – adds the characteristic (integer part) and adds the mantissa (fractional part). By contrast, to multiply two floating point numbers, one adds the exponent (which is logarithmic) and *multiplies* the significand (which is linear). Using "mantissa" for both terms obscures this distinction and creates a risk of confusion.

References

- Burks, Arthur W.; Goldstine, Herman H.; Von Neumann, John (1946). *Preliminary discussion of the logical design of an electronic computing instrument*. Technical Report, Institute for Advanced Study, Princeton, NJ. In Von Neumann, *Collected Works*, Vol. 5, A. H. Taub, ed., MacMillan, New York, 1963, p. 42:

  > 5.3. 'Several of the digital computers being built or planned in this country and England are to contain a so-called "floating decimal point". This is a mechanism for expressing each word as a characteristic and a mantissa—e.g. 123.45 would be carried in the machine as (0.12345,03), where the 3 is the exponent of 10 associated with the number.'

Article Sources and Contributors

Floating point *Source*: http://en.wikipedia.org/w/index.php?oldid=536019646 *Contributors*: 1exec1, 208.222.150.xxx, 47.83.107.xxx, 63.192.137.xxx, A. Pichler, AXRL, Abjad, Abovechief, Acdman1, Ahoerstemeier, Alexius08, Altenmann, Amanaplanacanalpanama, Ambulnick, Amoss, AnAj, Anand.arumug, AndrewKepert, AndyKali, Andyroo316, AnnaFrance, Apantomimehorse, Arnero, Arthena, Ashley Y, Ataleh, Attilios, Aykayel, Azraell, Bazzaah, Bdawson, Beland, Big Brother 1984, Bigdumbdinosaur, Bluebusy, Bluemoose, Bmearns, Bongwarrior, Booyabazooka, Borgx, BradBeattie, Brf, Brianbjparker, CIreland, CRGreathouse, Cabyd, CambridgeBayWeather, Canwolf, Cburnett, Cdion, CesarB, Charles Matthews, Chary pr23, Chris the speller, CitizenB, Cmdrjameson, Colonies Chris, Conversion script, Copyeditor42, Craig t moore, Cybercobra, Cyfal, Cyhawk, Damian Yerrick, Daniel.cussen, Davewho2, David-Sarah Hopwood, David.Monniaux, Davidhorman, Dcljr, Dcoetzee, Decora, Delirium, Dendodge, Derek Parnell, Derek farn, Devine9, Dmcq, Donfbreed2, Dooywopwopbanjio345, Dulciana, Długosz, Eclipsed, EdJohnston, Ednn, Efa, Ehudshapira, Electricmuffin11, Epbr123, Etu, Evaluist, Everyking, Evil saltine, Fang Aili, Ferritecore, Finell, Focomoso, Foobaz, Fredrik, Fresheneesz, Furrykef, Gaius Cornelius, Garde, GermanX, Gesslein, Giftlite, Godden46, Goudzovski, Graham87, Grim23, Grr82, Gunter, Guy Macon, Hairy Dude, HappyVR, Hefiz, Hex, Highpriority, Hires an editor, Iamsreehari, Ikanreed, Illusionz, InverseHypercube, Iseeaboar, Isilanes, Isomorphic, JNighthawk, JaGa, Jaan Vajakas, JakeVortex, Javalenok, Javier Carro, JeepdaySock, Jehan60188, Jennavecia, JimJJewett, Jimp, Jitse Niesen, Jmath666, Joe Decker, Jonathan de Boyne Pollard, Jorge Stolfi, Jotomicron, JulesH, Justanothervisitor, KSmrq, Kbdank71, Kbthompson, Keith D, Keka, Kenahoo, Kevin B12, Kjmathew, Kuszi, Kypzto, LaHaine, Lambiam, Lightmouse, Liviu trifoi, Lotje, Lovely idiot, Luckstev, Magioladitis, Malcolmxl5, Marioxcc, Maros, Mathiastck, Mav, Mcoupal, Meaningful Username, Meiskam, Merope, Mfc, Mgrant79, Michael Hardy, Michael.Pohoreski, Mikiemike, Miko3k, Mild Bill Hiccup, MishBaker, Misterblues, Mitch Ames, Miterdale, Mjb, Mr1278, Mrdvt92, Mshonle, MuhannadAjjan, N8mills, Nanshu, Nd, NickyMcLean, Nixdorf, Nutrimentia, Object01, Octahedron80, Oleg Alexandrov, OlivierM, Patrick, Paul Foxworthy, Pbroks13, Perl87, Pete142, Philip Trueman, Photographerguy, Physicistjedi, Poorsod, Premil, Puffin, R. S. Shaw, RTC, Reedy, Repelsteeltje, Ricklethickets, Rjwilmsi, RobertG, Ross Smith NZ, Ruud Koot, Ryk, S.Chepurin, Sanchom, Sgeo, Shanes, ShelfSkewed, Shuroo, Simetrical, SimonTrew, Simoneau, Slo-mo, SmileToday, Sns, Softtest123, Sonett72, Soyweiser, Spiel496, Sss41, Stevenj, Stux, Subversive, Suruena, Swat671, Tabletop, Taemyr, TakuyaMurata, Tbhotch, Tbleher, Teles, That Guy, From That Show!, The Anome, Thebestofall007, Thecheesykid, Tim1988, Tobias Bergemann, Tofergregg, Tomchiukc, Toolnut, Tsuji, Unixplumber, Unyoyega, Uriyan, Vanished user fweflklkaskwi4r592uofmoaihr, Verdy p, Vincent Lefèvre, Wanker jam, Wavelength, Wbrameld, Wernher, WikiDao, Wikipelli, Wikomidia, William Ackerman, Wilt, Wmmorrow, Wolfrock, Wordsoup, Wrs1864, Yonidebest, Yrkoon, Zelytic, ZeroOne, Zippanova, 535 anonymous edits

Single-precision floating-point format *Source*: http://en.wikipedia.org/w/index.php?oldid=532529476 *Contributors*: 198.207.223.xxx, Aklassyguy, Anthony Appleyard, B4hand, Brianbjparker, Cabyd, Conversion script, Coolant123, Cybercobra, Danilozf, Dicklyon, Dryguy, Dvaselaar, EncMstr, Etoombs, Foobaz, Goblin, Goodrone, Graham87, Happyuk, Harutsedo2, JLaTondre, JakeVortex, Jshadias, KeegY, Keka, KlappCK, Kuttipapu, Lamegeek8, Lone boatman, Mandarax, MattGiuca, Maury Markowitz, Mfc, Michael Angelkovich, Michael Hardy, Mortense, MrOllie, Qwerty112233, Radagast83, Rjstott, Sawak, ShashClp, Spacepotato, Stannered, StefanNL, Suruena, Theshadow27, UNV, Vadmium, Ylai, 103 anonymous edits

Significand *Source*: http://en.wikipedia.org/w/index.php?oldid=528593710 *Contributors*: Abdull, Aleenf1, Brianbjparker, Charles Matthews, Darkwind, Decrease789, Dexter Nextnumber, Fresheneesz, Gah4, Gene Nygaard, GermanX, Greenstruck, Herbee, Iosif, Jamelan, Josh Parris, KSmrq, Kmoksy, Macrakis, Mfc, Michael Hardy, NYKevin, Nbarth, PigFlu Oink, Polluks, Sandrobt, Sss41, Stevenj, Stevertigo, Thumperward, Wernher, 18 anonymous edits

Image Sources, Licenses and Contributors

File:Z3 Deutsches Museum.JPG *Source*: http://en.wikipedia.org/w/index.php?title=File:Z3_Deutsches_Museum.JPG *License*: GNU Free Documentation License *Contributors*: Original uploader was Venusianer at de.wikipedia (Original text : Venusianer 14:13, 3. Jan. 2007 (CET))

File:Float mantissa exponent.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Float_mantissa_exponent.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Sss41

File:Leonardo Torres Quevedo.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Leonardo_Torres_Quevedo.jpg *License*: Public Domain *Contributors*: desconocido/unknown

File:Konrad Zuse (1992).jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Konrad_Zuse_(1992).jpg *License*: GNU Free Documentation License *Contributors*: A.Savin, GeorgHH, Gildemax, Mentifisto, RHunscher, Siebrand, Vuk, 7 anonymous edits

File:William Kahan.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:William_Kahan.jpg *License*: GNU Free Documentation License *Contributors*: Baroc, YMS

File:Resistors in Parallel.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Resistors_in_Parallel.svg *License*: Public Domain *Contributors*: Inductiveload

File:Wilkinson.jpeg *Source*: http://en.wikipedia.org/w/index.php?title=File:Wilkinson.jpeg *License*: Public Domain *Contributors*: -

Image:Float example.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Float_example.svg *License*: GNU Free Documentation License *Contributors*: en:User:Fresheneesz, traced by User:Stannered

File:Float_mantissa_exponent.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Float_mantissa_exponent.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Sss41

License