IEEE floating-point standard

From Wikipedia, the free encyclopedia
(Redirected from IEEE 754)

The **IEEE Standard for Binary Floating-Point Arithmetic** (**IEEE 754**) is the most widely-used standard for floating-point computation, and is followed by many CPU and FPU implementations. The standard defines formats for representing floating-point numbers (including negative zero and denormal numbers) and special values (infinities and NaNs) together with a set of *floating-point operations* that operate on these values. It also specifies four rounding modes and five exceptions (including when the exceptions occur, and what happens when they do occur).

IEEE 754 specifies four formats for representing floating-point values: single-precision (32-bit), double-precision (64-bit), single-extended precision (≥ 43-bit, not commonly used) and double-extended precision (≥ 79-bit, usually implemented with 80 bits). Only 32-bit values are required by the standard; the others are optional. Many languages specify that IEEE formats and arithmetic be implemented, although sometimes it is optional. For example, the C programming language, which pre-dated IEEE 754, now allows but does not require IEEE arithmetic (the C `float` typically is used for IEEE single-precision and `double` uses IEEE double-precision).

The full title of the standard is **IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)**, and it is also known as **IEC 60559:1989, Binary floating-point arithmetic for microprocessor systems** (originally the reference number was IEC 559:1989).[1] Later there was an **IEEE 854-1987** for "radix independent floating point" as long as the radix is 2 or 10.

Contents

Anatomy of a floating-point number

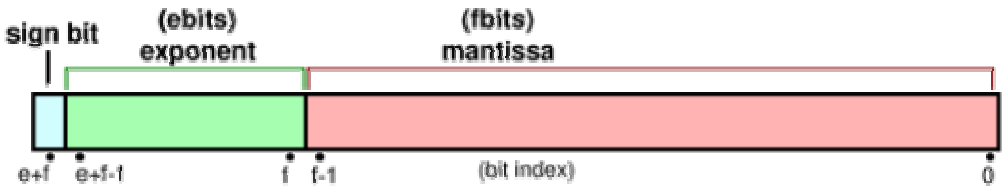
Following is a description of the standards' format for floating-point numbers.

Bit conventions used in this article

Bits within a word of width W are indexed by integers in the range 0 to W−1 inclusive. The bit with index 0 is drawn on the right. The lowest indexed bit is usually the least significant.

General layout

Binary floating-point numbers are stored in a sign-magnitude form as follows:



where the most significant bit is the sign bit, *exponent* is the biased exponent, and *mantissa* is the significand minus the *most significant bit*.

Exponent biasing

The exponent is biased by $2^{e-1} - 1$. Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder. To solve this the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison.

For example, to represent a number which has exponent of 17, *exponent* is $17+2^{e-1} - 1$.

Cases

The most significant bit of the mantissa is determined by the value of *exponent*. If $0 < exponent < 2^e - 1$, the most significant bit of the *mantissa* is 1 , and the number is said to be *normalized*. If *exponent* is 0, the most significant bit of the *mantissa* is 0 and the number is said to be *de-normalized*. Three special cases arise:

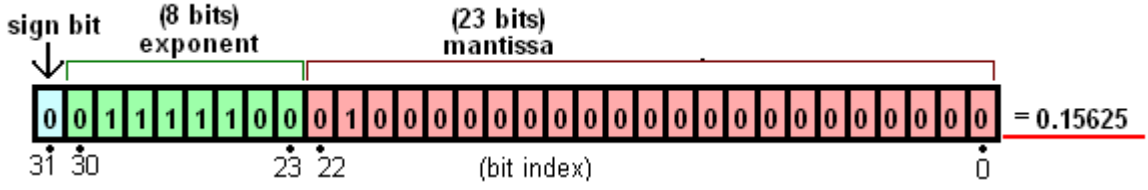
1. if *exponent* is 0 and *mantissa* is 0, the number is ±0 (depending on the sign bit)
2. if *exponent* = $2^e - 1$ and *mantissa* is 0, the number is ±infinity (again depending on the sign bit), and
3. if *exponent* = $2^e - 1$ and *mantissa* is not 0, the number being represented is not a number (NaN).

This can be summarized as:

Type	Exponent	Mantissa
Zeroes	0	0
Denormalized numbers	0	non zero
Normalized numbers	1 to $2^e - 2$	any
Infinities	$2^e - 1$	0
NaNs	$2^e - 1$	non zero

Single-precision 32 bit

A single-precision binary floating-point number is stored in a 32-bit word:



The exponent is biased by $2^{8-1} - 1 = 127$ in this case, so that exponents in the range −126 to +127 are representable. An exponent of −127 would be biased to the value 0 but this is reserved to encode that the value is a denormalized number or zero. An exponent of 128 would be biased to the value 255 but this is reserved to encode an infinity or not a number.

For normalised numbers, the most common, Exp is the biased exponent and Fraction is the fractional part of the significand. The number has value v:

$v = s \times 2^e \times m$

Where

s = +1 (positive numbers) when the sign bit is 0

s = −1 (negative numbers) when the sign bit is 1

e = Exp − 127 (in other words the exponent is stored with 127 added to it, also called "biased with 127")

m = 1.Fraction in binary (that is, the significand is the binary number 1 followed by the radix point followed by the binary bits of Fraction). Therefore, $1 \leq m < 2$.

In the example shown above, the sign is zero, the exponent is −3, and the significand is 1.01 (in binary, which is 1.25 in decimal). The represented number is therefore $+1.25 \times 2^{-3}$, which is +0.15625.

Notes:

1. Denormalized numbers are the same except that e = −126 and m is 0.Fraction. (e is NOT −127 : The significand has to be shifted to the right by one more bit, in order to include the leading bit, which is not always 1 in this case. This is balanced by incrementing the exponent to −126 for the calculation.)
2. −126 is the smallest exponent for a normalized number
3. There are two Zeroes, +0 (S is 0) and −0 (S is 1)
4. There are two Infinities +∞ (S is 0) and −∞ (S is 1)
5. NaNs may have a sign and a significand, but these have no meaning other than for diagnostics; the first bit of the significand is often used to distinguish *signaling NaNs* from *quiet NaNs*
6. NaNs and Infinities have all 1s in the Exp field.
7. The smallest non-zero positive and largest non-zero negative numbers (represented by the denormalized value with all 0s in the Exp field and the binary value 1 in the Fraction field) are
   $\pm 2^{-149} \approx \pm 1.4012985 \times 10^{-45}$
8. The smallest non-zero positive and largest non-zero negative normalized numbers (represented with the binary value 1 in the Exp field and 0 in the Fraction field) are
   $\pm 2^{-126} \approx \pm 1.175494351 \times 10^{-38}$
9. The largest finite positive and smallest finite negative numbers (represented by the value with 254 in the Exp field and all 1s in the Fraction field) are
   $\pm(2^{128} - 2^{104}) \approx \pm 3.4028235 \times 10^{38}$

Here is the summary table from the previous section with some example 32-bit single-precision examples:

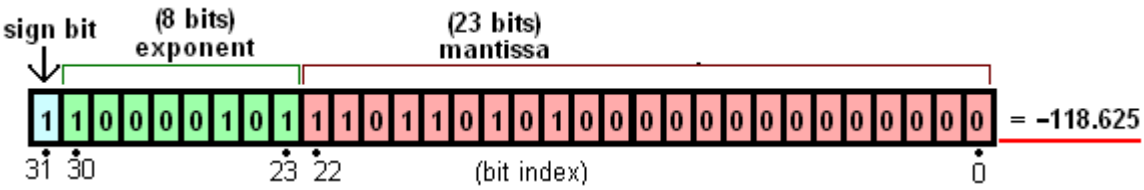
Type	Exponent	Mantissa	Value
Zero	0000 0000	000 0000 0000 0000 0000 0000	0.0
One	0111 1111	000 0000 0000 0000 0000 0000	1.0
Denormalized number	0000 0000	100 0000 0000 0000 0000 0000	5.9×10^{-39}
Large normalized number	1111 1110	111 1111 1111 1111 1111 1111	3.4×10^{38}
Small normalized number	0000 0001	000 0000 0000 0000 0000 0000	1.18×10^{-38}
Infinity	1111 1111	000 0000 0000 0000 0000 0000	Infinity
NaN	1111 1111	010 0000 0000 0000 0000 0000	NaN

A more complex example

Let us encode the decimal number −118.625 using the IEEE 754 system.

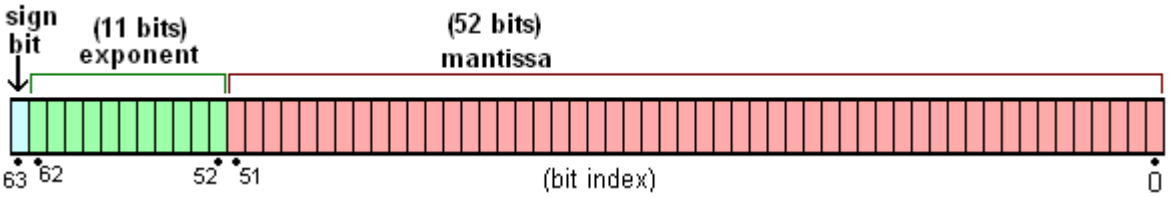
1. First we need to get the sign, the exponent and the fraction. Because it is a negative number, the sign is "1".
2. Now, we write the number (without the sign) using binary notation. The result is 1110110.101.
3. Next, let's move the radix point left, leaving only a 1 at its left: $1110110.101 = 1.110110101 \times 2^6$. This is a normalized floating point number. The mantissa is the part at the right of the radix point, filled with 0 on the right until we get all 23 bits. That is 11011010100000000000000.
4. The exponent is 6, but we need to convert it to binary and bias it (so the most negative exponent is 0, and all exponents are non-negative binary numbers). For the 32-bit IEEE 754 format, the bias is 127 and so 6 + 127 = 133. In binary, this is written as 10000101.

Putting them all together:



Double-precision 64 bit

Double precision is essentially the same except that the fields are wider:



The mantissa is much larger, while the exponent is only slightly larger. This is because precision is more valued than range, according to the creators of the standard.

NaNs and Infinities are represented with Exp being all 1s (2047).

For Normalized numbers the exponent bias is +1023 (so e is Exp − 1023). For Denormalized numbers the exponent is −1022 (the minimum exponent for a normalized number—it is not −1023 because normalised numbers have a leading 1 digit before the binary point and denormalized numbers do not). As before, both infinity and zero are signed.

Notes:

1. The smallest non-zero positive and largest non-zero negative numbers (represented by the denormalized value with all 0s in the Exp field and the binary value 1 in the Fraction field) are
   $\pm 2^{-1074} \approx \pm 5 \times 10^{-324}$
2. The smallest non-zero positive and largest non-zero negative normalized numbers (represented by the value with the binary value 1 in the Exp and 0 in the Fraction field) are
   $\pm 2^{-1022} \approx \pm 2.2250738585072020 \times 10^{-308}$
3. The largest finite positive and smallest finite negative numbers (represented by the value with 2046 in the Exp field and all 1s in the Fraction field) are
   $\pm (2^{1024} - 2^{971}) \approx \pm 1.7976931348623157 \times 10^{308}$

Comparing floating-point numbers

Comparing floating-point numbers is usually best done using floating-point instructions. However, this representation makes comparisons of some subsets of numbers possible on a byte-by-byte basis, if they share the same byte order and the same sign, and NaNs are excluded.

For example, for two positive floating-point numbers a and b, a comparison between a and b (>, <, or ==) gives identical results as the comparison of two signed (or unsigned) binary integers with the same bit patterns and same byte order as a and b. In other words, two positive floating-point numbers (known not to be NaNs) can be compared with a signed (or unsigned) binary integer comparison using the same bits, providing the floating-point numbers use the same byte order. Because the byte order matters, this type of comparison cannot be used in portable code through a union in the C programming language. This is an example of lexicographic ordering.

Rounding floating-point numbers

The IEEE standard has four different rounding modes.

- **Unbiased** which rounds to the nearest value, if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit. This mode is required to be default.

- **Towards zero**
- **Towards positive infinity**
- **Towards negative infinity**

Extending the real numbers

The IEEE standard employs (and extends) the affinely extended real number system, with separate positive and negative infinities. During drafting, there was a proposal for the standard to incorporate the projectively extended real number system, with a single unsigned infinity, by providing programmers with a mode selection option. In the interest of reducing the complexity of the final standard, the projective mode was dropped, however. The Intel 8087 and Intel 80287 floating point co-processors both support this projective mode.[1][2][3]

Recommended functions and predicates

- Under some C compilers, copysign(x,y) returns x with the sign of y, so abs(x) = copysign(x,1.0). Note that this is one of the few operations which operates on a NaN in a way resembling arithmetic. Note that copysign is a new function under the C99 standard.
- −x returns x with the sign reversed. Note that this is different than 0−x in some cases, notably when x is 0. So −(0) is −0, but the sign of 0−0 depends on the rounding mode.
- scalb (y, N)
- logb (x)
- finite (x) a predicate for "x is a finite value", equivalent to −Inf < x < Inf
- isnan (x) a predicate for "x is a nan", equivalent to "x ≠ x"
- x <> y which turns out to have different exception behavior than NOT(x = y).
- unordered (x, y) is true when "x is unordered with y", i.e., either x or y is a NaN.
- class (x)
- nextafter(x,y) returns the next representable value from x in the direction towards y

References

1. ^ John R. Hauser (March 1996). "Handling Floating-Point Exceptions in Numeric Programs" (PDF). *ACM Transactions on Programming Languages and Systems* **18** (2).
2. ^ David Stevenson (March 1981). "IEEE Task P754: A proposed standard for binary floating-point arithmetic". *Computer* **14** (3): 51–62.
3. ^ Kahan, W. and Palmer, J. (1979). "On a proposed floating-point standard". *SIGNUM Newsletter* **14** (Special): 13–21.

- Floating Point Unit by Jidan Al-Eryani

Revision of the standard

Note that the IEEE 754 standard is currently under revision. See: IEEE 754r

See also

- −0 (negative zero)
- IEEE 754r working group to revise IEEE 754-1985.
- NaN (Not a Number)
- minifloat for simple examples of properties of IEEE 754 floating point numbers
- Intel 8087 (early implementation effort)

External links

- IEEE 754 references
- Let's Get To The (Floating) Point by Chris Hecker
- What Every Computer Scientist Should Know About Floating-Point Arithmetic by David Goldberg - a good introduction and explanation.
- IEEE 854-1987 History and minutes
- Converter
- Another Converter

- Converter as MS-Windows program

Retrieved from "http://en.wikipedia.org/wiki/IEEE_floating-point_standard"

Categories: Computer arithmetic | IEEE standards