IEEE 754-1985

IEEE 754-1985 was an industry standard for representing floating-point numbers in computers, officially adopted in 1985 and superseded in 2008 by IEEE 754-2008. During its 23 years, it was the most widely used format for floating-point computation. It was implemented in software, in the form of floating-point libraries, and in hardware, in the instructions of many CPUs and FPUs. The first integrated circuit to implement the draft of what was to become IEEE 754-1985 was the Intel 8087.

IEEE 754-1985 represents numbers in binary, providing definitions for four levels of precision, of which the two most commonly used are:

Level	Width	Range at full precision	Precision[a]
Single precision	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	Approximately 7 decimal digits
Double precision	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	Approximately 16 decimal digits

The standard also defines representations for positive and negative infinity, a "negative zero", five exceptions to handle invalid results like division by zero, special values called NaNs for representing those exceptions, denormal numbers to represent numbers smaller than shown above, and four rounding modes.

Contents

Representation of numbers
- Zero
- Denormalized numbers

Representation of non-numbers
- Positive and negative infinity
- NaN

Range and precision
- Single precision
- Double precision
- Extended formats

Examples

Comparing floating-point numbers

Rounding floating-point numbers

Extending the real numbers

Functions and predicates
- Standard operations
- Recommended functions and predicates

History

See also

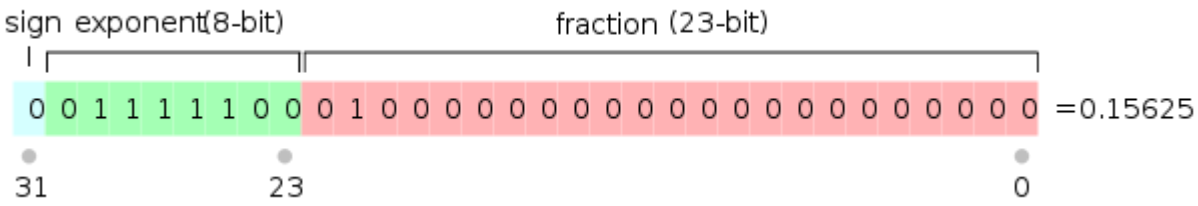
Notes

References

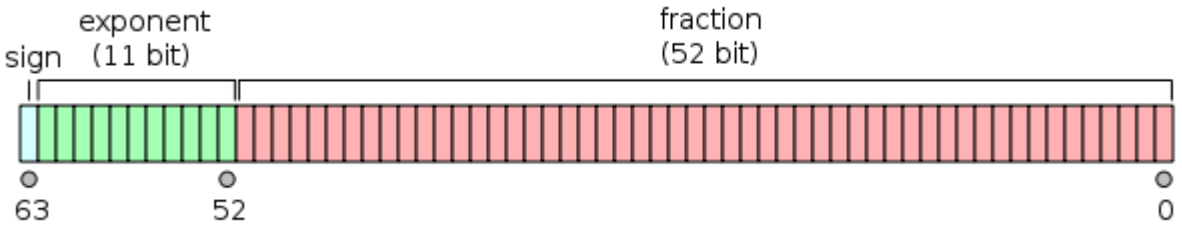
Further reading

External links

Representation of numbers



The number 0.15625 represented as a single-precision IEEE 754-1985 floating-point number. See text for explanation.



The three fields in a 64bit IEEE 754 float

Floating-point numbers in IEEE 754 format consist of three fields: a sign bit, a biased exponent, and a fraction. The following example illustrates the meaning of each.

The decimal number 0.15625_{10} represented in binary is 0.00101_2 (that is, 1/8 + 1/32). (Subscripts indicate the number base.) Analogous to scientific notation, where numbers are written to have a single non-zero digit to the left of the decimal point, we rewrite this number so it has a single 1 bit to the left of the "binary point". We simply multiply by the appropriate power of 2 to compensate for shifting the bits left by three positions:

$$0.00101_2 = 1.01_2 \times 2^{-3}$$

Now we can read off the fraction and the exponent: the fraction is $.01_2$ and the exponent is −3.

As illustrated in the pictures, the three fields in the IEEE 754 representation of this number are:

sign = 0, because the number is positive. (1 indicates negative.)
biased exponent = −3 + the "bias". In **single precision**, the bias is **127**, so in this example the biased exponent is 124; in **double precision**, the bias is **1023**, so the biased exponent in this example is 1020.
fraction = $.01000\ldots_2$.

IEEE 754 adds a bias to the exponent so that numbers can in many cases be compared conveniently by the same hardware that compares signed 2's-complement integers. Using a biased exponent, the lesser of two positive floating-point numbers will come out "less than" the greater following the same ordering as for sign and magnitude integers. If two floating-point numbers have different signs, the sign-and-magnitude comparison also works with biased exponents. However, if both

biased-exponent floating-point numbers are negative, then the ordering must be reversed. If the exponent were represented as, say, a 2's-complement number, comparison to see which of two numbers is greater would not be as convenient.

The leading 1 bit is omitted since all numbers except zero start with a leading 1; the leading 1 is implicit and doesn't actually need to be stored which gives an extra bit of precision for "free."

Zero

The number zero is represented specially:

sign = 0 for positive zero, 1 for negative zero.
biased exponent = 0.
fraction = 0.

Denormalized numbers

The number representations described above are called *normalized,* meaning that the implicit leading binary digit is a 1. To reduce the loss of precision when an underflow occurs, IEEE 754 includes the ability to represent fractions smaller than are possible in the normalized representation, by making the implicit leading digit a 0. Such numbers are called denormal. They don't include as many significant digits as a normalized number, but they enable a gradual loss of precision when the result of an arithmetic operation is not exactly zero but is too close to zero to be represented by a normalized number.

A denormal number is represented with a biased exponent of all 0 bits, which represents an exponent of −126 in single precision (not −127), or −1022 in double precision (not −1023).[1] In contrast, the smallest biased exponent representing a normal number is 1 (see examples below).

Representation of non-numbers

The biased-exponent field is filled with all 1 bits to indicate either infinity or an invalid result of a computation.

Positive and negative infinity

Positive and negative infinity are represented thus:

sign = 0 for positive infinity, 1 for negative infinity.
biased exponent = all 1 bits.
fraction = all 0 bits.

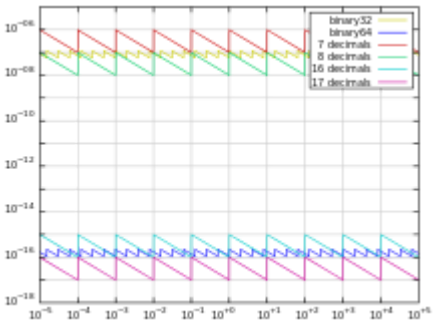
NaN

Some operations of floating-point arithmetic are invalid, such as taking the square root of a negative number. The act of reaching an invalid result is called a floating-point *exception*. An exceptional result is represented by a special code called a NaN, for "Not a Number". All NaNs in IEEE 754-1985 have this format:

sign = either 0 or 1.
biased exponent = all 1 bits.
fraction = anything except all 0 bits (since all 0 bits represents infinity).

Range and precision

Precision is defined as the minimum difference between two successive mantissa representations; thus it is a function only in the mantissa; while the gap is defined as the difference between two successive numbers.[2]



Relative precision of single (binary32) and double precision (binary64) numbers, compared with decimal representations using a fixed number of significant digits. Relative precision is defined here as ulp(x)/x, where ulp(x) is the unit in the last place in the representation of x, i.e. the gap between x and the next representable number.

Single precision

Single-precision numbers occupy 32 bits. In single precision:

- The positive and negative numbers closest to zero (represented by the denormalized value with all 0s in the exponent field and the binary value 1 in the fraction field) are

  $$\pm 2^{-149} \approx \pm 1.40130 \times 10^{-45}$$

- The positive and negative normalized numbers closest to zero (represented with the binary value 1 in the exponent field and 0 in the fraction field) are

  $$\pm 2^{-126} \approx \pm 1.17549 \times 10^{-38}$$

- The finite positive and finite negative numbers furthest from zero (represented by the value with 254 in the exponent field and all 1s in the fraction field) are

  $$\pm(1-2^{-24}) \times 2^{128[3]} \approx \pm 3.40282 \times 10^{38}$$

Some example range and gap values for given exponents in single precision:

Actual Exponent (unbiased)	Exp (biased)	Minimum	Maximum	Gap
0	127	1	≈ 1.999999880791	≈ 1.19209e-7
1	128	2	≈ 3.999999761581	≈ 2.38419e-7
2	129	4	≈ 7.999999523163	≈ 4.76837e-7
10	137	1024	≈ 2047.999877930	≈ 1.22070e-4
11	138	2048	≈ 4095.999755859	≈ 2.44141e-4
23	150	8388608	16777215	1
24	151	16777216	33554430	2
127	254	≈ 1.70141e38	≈ 3.40282e38	≈ 2.02824e31

As an example, 16,777,217 can not be encoded as a 32-bit float as it will be rounded to 16,777,216. This shows why floating point arithmetic is unsuitable for accounting software. However, all integers within the representable range that are a power of 2 can be stored in a 32-bit float without rounding.

Double precision

Double-precision numbers occupy 64 bits. In double precision:

- The positive and negative numbers closest to zero (represented by the denormalized value with all 0s in the Exp field and the binary value 1 in the Fraction field) are

$$\pm 2^{-1074} \approx \pm 4.94066 \times 10^{-324}$$

- The positive and negative normalized numbers closest to zero (represented with the binary value 1 in the Exp field and 0 in the fraction field) are

$$\pm 2^{-1022} \approx \pm 2.22507 \times 10^{-308}$$

- The finite positive and finite negative numbers furthest from zero (represented by the value with 2046 in the Exp field and all 1s in the fraction field) are

$$\pm(1-2^{-53}) \times 2^{1024[3]} \approx \pm 1.79769 \times 10^{308}$$

Some example range and gap values for given exponents in double precision:

Actual Exponent (unbiased)	Exp (biased)	Minimum	Maximum	Gap
0	1023	1	≈ 1.9999999999999777955	≈ 2.22045e-16
1	1024	2	≈ 3.9999999999999555911	≈ 4.44089e-16
2	1025	4	≈ 7.9999999999999111822	≈ 8.88178e-16
10	1033	1024	≈ 2047.999999999999772626	≈ 2.27374e-13
11	1034	2048	≈ 4095.999999999999545253	≈ 4.54747e-13
52	1075	4503599627370496	9007199254740991	1
53	1076	9007199254740992	18014398509481982	2
1023	2046	≈ 8.98847e307	≈ 1.79769e308	≈ 1.99584e292

Extended formats

The standard also recommends extended format(s) to be used to perform internal computations at a higher precision than that required for the final result, to minimise round-off errors: the standard only specifies minimum precision and exponent requirements for such formats. The x87 80-bit extended format is the most commonly implemented extended format that meets these requirements.

Examples

Here are some examples of single-precision IEEE 754 representations:

Type	Sign	Actual Exponent	Exp (biased)	Exponent field	Significand (fraction field)	Value
Zero	0	−127	0	0000 0000	000 0000 0000 0000 0000 0000	0.0
Negative zero	1	−127	0	0000 0000	000 0000 0000 0000 0000 0000	−0.0
One	0	0	127	0111 1111	000 0000 0000 0000 0000 0000	1.0
Minus One	1	0	127	0111 1111	000 0000 0000 0000 0000 0000	−1.0
Smallest denormalized number	*	−126	0	0000 0000	000 0000 0000 0000 0000 0001	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	−126	0	0000 0000	100 0000 0000 0000 0000 0000	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	−126	0	0000 0000	111 1111 1111 1111 1111 1111	$\pm(1-2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	−126	1	0000 0001	000 0000 0000 0000 0000 0000	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Largest normalized number	*	127	254	1111 1110	111 1111 1111 1111 1111 1111	$\pm(2-2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	128	255	1111 1111	000 0000 0000 0000 0000 0000	+∞
Negative infinity	1	128	255	1111 1111	000 0000 0000 0000 0000 0000	−∞
Not a number	*	128	255	1111 1111	non zero	NaN
* Sign bit can be either 0 or 1 .						

Comparing floating-point numbers

Every possible bit combination is either a NaN or a number with a unique value in the affinely extended real number system with its associated order, except for the two bit combinations negative zero and positive zero, which sometimes require special attention (see below). The binary representation has the special property that, excluding NaNs, any two numbers can be compared as sign and magnitude integers (endianness issues apply). When comparing as 2's-complement integers: If the sign bits differ, the negative number precedes the positive number, so 2's complement gives the correct result (except that negative zero and positive zero should be considered equal). If both values are positive, the 2's complement comparison again gives the correct result. Otherwise (two negative numbers), the correct FP ordering is the opposite of the 2's complement ordering.

Rounding errors inherent in floating point calculations often make comparison of results for exact equality not useful. Choosing an acceptable range is a complex topic.

Although negative zero and positive zero are generally considered equal for comparison purposes, some programming language relational operators and similar constructs might or do treat them as distinct. According to the Java Language Specification,[4] comparison and equality operators treat them as equal, but `Math.min()` and `Math.max()` distinguish

them (officially starting with Java version 1.1 but actually with 1.1.1), as do the comparison methods `equals()`, `compareTo()` and even `compare()` of classes `Float` and `Double`.

Rounding floating-point numbers

The IEEE standard has four different rounding modes; the first is the default; the others are called *directed roundings*.

- **Round to Nearest** – rounds to the nearest value; if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50% of the time (in IEEE 754-2008 this mode is called *roundTiesToEven* to distinguish it from another round-to-nearest mode)
- **Round toward 0** – directed rounding towards zero
- **Round toward +∞** – directed rounding towards positive infinity
- **Round toward −∞** – directed rounding 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.[5][6][7]

Functions and predicates

Standard operations

The following functions must be provided:

- Add, subtract, multiply, divide
- Square root
- Floating point remainder. This is not like a normal modulo operation, it can be negative for two positive numbers. It returns the exact value of $x-(\text{round}(x/y)\cdot y)$.
- Round to nearest integer. For undirected rounding when halfway between two integers the even integer is chosen.
- Comparison operations. Besides the more obvious results, IEEE 754 defines that $-\infty = -\infty$, $+\infty = +\infty$ and $x \neq \texttt{NaN}$ for any x (including `NaN`).

Recommended functions and predicates

- `copysign(x,y)` returns x with the sign of y, so `abs(x)` equals `copysign(x,1.0)`. This is one of the few operations which operates on a NaN in a way resembling arithmetic. The function `copysign` is new in the C99 standard.
- −x returns x with the sign reversed. This is different from 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

History

In 1976 Intel began planning to produce a floating point coprocessor. John Palmer, the manager of the effort, persuaded them that they should try to develop a standard for all their floating point operations. William Kahan was hired as a consultant; he had helped improve the accuracy of Hewlett-Packard's calculators. Kahan initially recommended that the floating point base be decimal[8] but the hardware design of the coprocessor was too far along to make that change.

The work within Intel worried other vendors, who set up a standardization effort to ensure a 'level playing field'. Kahan attended the second IEEE 754 standards working group meeting, held in November 1977. Here, he received permission from Intel to put forward a draft proposal based on the standard arithmetic part of their design for a coprocessor. The arguments over gradual underflow lasted until 1981 when an expert hired by DEC to assess it sided against the dissenters.

Even before it was approved, the draft standard had been implemented by a number of manufacturers.[9][10] The Intel 8087, which was announced in 1980, was the first chip to implement the draft standard.

See also

- −0 (negative zero)
- Intel 8087
- minifloat for simple examples of properties of IEEE 754 floating point numbers
- Fixed-point arithmetic

Notes

a. Precision: The number of decimal digits precision is calculated via number_of_mantissa_bits * $\text{Log}_{10}(2)$. Thus ~7.2 and ~15.9 for single and double precision respectively.

References

1. Hennessy. *Computer Organization and Design*. Morgan Kaufmann. p. 270.
2. Hossam A. H. Fahmy; Shlomo Waser; Michael J. Flynn, *Computer Arithmetic* (https://web.archive.org/web/20101008203307/http://arith.stanford.edu/~hfahmy/webpages/arith_class/arith.pdf) (PDF), archived from the original (http://arith.stanford.edu/~hfahmy/webpages/arith_class/arith.pdf) (PDF) on 2010-10-08, retrieved 2011-01-02
3. William Kahan. "Lecture Notes on the Status of IEEE 754" (http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF) (PDF). October 1, 1997 3:36 am. Elect. Eng. & Computer Science University of California. Retrieved 2007-04-12.
4. "Java Language and Virtual Machine Specifications" (http://java.sun.com/docs/books/jls/). *Java Documentation*.
5. John R. Hauser (March 1996). "Handling Floating-Point Exceptions in Numeric Programs" (http://www.jhauser.us/publications/1996_Hauser_FloatingPointExceptions.html) (PDF). *ACM Transactions on Programming Languages and Systems*. **18** (2): 139–174. doi:10.1145/227699.227701 (https://doi.org/10.1145%2F227699.227701).
6. David Stevenson (March 1981). "IEEE Task P754: A proposed standard for binary floating-point arithmetic". *IEEE Computer*. **14** (3): 51–62.
7. William Kahan and John Palmer (1979). "On a proposed floating-point standard". *SIGNUM Newsletter*. **14** (Special): 13–21. doi:10.1145/1057520.1057522 (https://doi.org/10.1145%2F1057520.1057522).
8. W. Kahan 2003, pers. comm. to Mike Cowlishaw and others after an IEEE 754 meeting
9. Charles Severance (20 February 1998). "An Interview with the Old Man of Floating-Point" (http://www.eecs.berkeley.edu/~wkahan/ieee754status/754story.html).
10. Charles Severance. "History of IEEE Floating-Point Format" (http://cnx.org/content/m32770/latest/). Connexions.

Further reading

- Charles Severance (March 1998). "IEEE 754: An Interview with William Kahan" (http://www.freecollab.com/dr-chuck/papers/columns/r3114.pdf) (PDF). *IEEE Computer*. **31** (3): 114–115. doi:10.1109/MC.1998.660194 (https://doi.org/10.1109%2FMC.1998.660194). Retrieved 2008-04-28.
- David Goldberg (March 1991). "What Every Computer Scientist Should Know About Floating-Point Arithmetic" (http://www.validlab.com/goldberg/paper.pdf) (PDF). *ACM Computing Surveys*. **23** (1): 5–48. doi:10.1145/103162.103163 (https://doi.org/10.1145%2F103162.103163). Retrieved 2008-04-28.
- Chris Hecker (February 1996). "Let's Get To The (Floating) Point" (http://www.d6.com/users/checker/pdfs/gdmfp.pdf) (PDF). *Game Developer Magazine*: 19–24. ISSN 1073-922X (https://www.worldcat.org/issn/1073-922X).
- David Monniaux (May 2008). "The pitfalls of verifying floating-point computations" (http://hal.archives-ouvertes.fr/hal-00128124/en/). *ACM Transactions on Programming Languages and Systems*. **30** (3): article #12. doi:10.1145/1353445.1353446 (https://doi.org/10.1145%2F1353445.1353446). ISSN 0164-0925 (https://www.worldcat.org/issn/0164-0925).: A compendium of non-intuitive behaviours of floating-point on popular architectures, with implications for program verification and testing.

External links

- Comparing floats (http://www.cygnus-software.com/papers/comparingfloats/Obsolete%20comparing%20floating%20point%20numbers.htm)
- Coprocessor.info: x87 FPU pictures, development and manufacturer information (https://web.archive.org/web/20070314154031/http://www.coprocessor.info/)
- IEEE 854-1987 (http://speleotrove.com/decimal/854mins.html) — History and minutes
- IEEE754 (Single and Double precision) Online Converter (http://www.binaryconvert.com/convert_float.html)
