

Double-precision floating-point format

Double-precision floating-point format (sometimes called **FP64**) is a computer number format, usually occupying 64 bits in computer memory; it represents a wide dynamic range of numeric values by using a floating radix point.

Floating point is used to represent fractional values, or when a wider range is needed than is provided by fixed point (of the same bit width), even if at the cost of precision. Double precision may be chosen when the range or precision of single precision would be insufficient.

In the IEEE 754-2008 standard, the 64-bit base-2 format is officially referred to as **binary64**; it was called **double** in IEEE 754-1985. IEEE 754 specifies additional floating-point formats, including 32-bit base-2 *single precision* and, more recently, base-10 representations.

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 floating-point data types depended on the computer manufacturer and computer model, and upon decisions made by programming-language implementers. E.g., GW-BASIC's double-precision data type was the 64-bit MBF floating-point format.

Contents

IEEE 754 double-precision binary floating-point format: binary64

Exponent encoding

Endianness

Double-precision examples

Execution speed with double-precision arithmetic

Implementations

C and C++

Fortran

Common Lisp

Java

JavaScript

See also

Notes and references

IEEE 754 double-precision binary floating-point format: binary64

Double-precision binary floating-point is a commonly used format on PCs, due to its wider range over single-precision floating point, in spite of its performance and bandwidth cost. It is commonly known simply as *double*. The IEEE 754 standard specifies a **binary64** as having:

- Sign bit: 1 bit
- Exponent: 11 bits

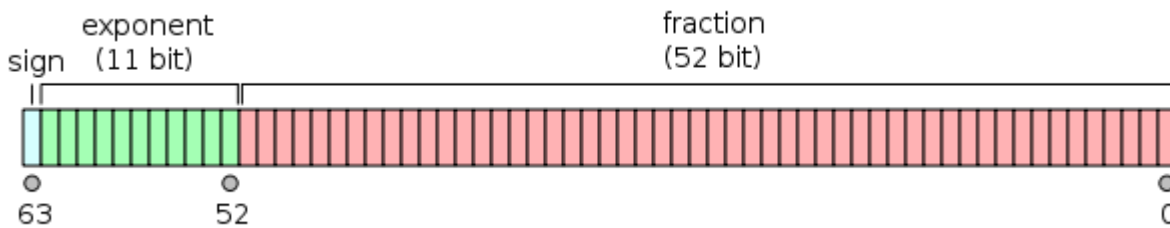
- Significand precision: 53 bits (52 explicitly stored)

The sign bit determines the sign of the number (including when this number is zero, which is signed).

The exponent field is an 11-bit unsigned integer from 0 to 2047, in biased form: an exponent value of 1023 represents the actual zero. Exponents range from -1022 to $+1023$ because exponents of -1023 (all 0s) and $+1024$ (all 1s) are reserved for special numbers.

The 53-bit significand precision gives from 15 to 17 significant decimal digits precision ($2^{-53} \approx 1.11 \times 10^{-16}$). If a decimal string with at most 15 significant digits is converted to IEEE 754 double-precision representation, and then converted back to a decimal string with the same number of digits, the final result should match the original string. If an IEEE 754 double-precision number is converted to a decimal string with at least 17 significant digits, and then converted back to double-precision representation, the final result must match the original number.^[1]

The format is written with the significand having an implicit integer bit of value 1 (except for special data, see the exponent encoding below). With the 52 bits of the fraction (F) significand appearing in the memory format, the total precision is therefore 53 bits (approximately 16 decimal digits, $53 \log_{10}(2) \approx 15.955$). The bits are laid out as follows:



The real value assumed by a given 64-bit double-precision datum with a given biased exponent e and a 52-bit fraction is

$$(-1)^{\text{sign}} (1.b_{51}b_{50}\dots b_0)_2 \times 2^{e-1023}$$

or

$$(-1)^{\text{sign}} \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i} \right) \times 2^{e-1023}$$

Between $2^{52}=4,503,599,627,370,496$ and $2^{53}=9,007,199,254,740,992$ the representable numbers are exactly the integers. For the next range, from 2^{53} to 2^{54} , everything is multiplied by 2, so the representable numbers are the even ones, etc. Conversely, for the previous range from 2^{51} to 2^{52} , the spacing is 0.5, etc.

The spacing as a fraction of the numbers in the range from 2^n to 2^{n+1} is 2^{n-52} . The maximum relative rounding error when rounding a number to the nearest representable one (the machine epsilon) is therefore 2^{-53} .

The 11 bit width of the exponent allows the representation of numbers between 10^{-308} and 10^{308} , with full 15–17 decimal digits precision. By compromising precision, the subnormal representation allows even smaller values up to about 5×10^{-324} .

Exponent encoding

The double-precision binary floating-point exponent is encoded using an offset-binary representation, with the zero offset being 1023; also known as exponent bias in the IEEE 754 standard. Examples of such representations would be:

$$e=00000000001_2=001_{16}=1: \quad 2^{1-1023} = 2^{-1022} \quad (\text{smallest exponent for normal numbers})$$

$$e = 011111111111_2 = 3ff_{16} = 1023: \quad 2^{1023-1023} = 2^0 \quad (\text{zero offset})$$

$$e = 10000000101_2 = 405_{16} = 1029: \quad 2^{1029-1023} = 2^6$$

$$e = 111111111110_2 = 7fe_{16} = 2046: \quad 2^{2046-1023} = 2^{1023} \text{ (highest exponent)}$$

The exponents 000_{16} and $7ff_{16}$ have a special meaning:

- $000000000000_2 = 000_{16}$ is used to represent a signed zero (if $F = 0$) and subnormals (if $F \neq 0$); and
- $111111111111_2 = 7ff_{16}$ is used to represent ∞ (if $F = 0$) and NaNs (if $F \neq 0$),

where F is the fractional part of the significand. All bit patterns are valid encoding.

Except for the above exceptions, the entire double-precision number is described by:

$$(-1)^{\text{sign}} \times 2^{e-1023} \times 1.\text{fraction}$$

In the case of subnormals ($e = 0$) the double-precision number is described by:

$$(-1)^{\text{sign}} \times 2^{1-1023} \times 0.\text{fraction} = (-1)^{\text{sign}} \times 2^{-1022} \times 0.\text{fraction}$$

Endianness

Although the ubiquitous x86 processors of today use little-endian storage for all types of data (integer, floating point), there are a number of hardware architectures where floating-point numbers are represented in big-endian form while integers are represented in little-endian form.^[2] There are ARM processors that have half little-endian, half big-endian floating-point representation for double-precision numbers: both 32-bit words are stored in little-endian like integer registers, but the most significant one first. Because there have been many floating-point formats with no "network" standard representation for them, the XDR standard uses big-endian IEEE 754 as its representation. It may therefore appear strange that the widespread IEEE 754 floating-point standard does not specify endianness.^[3] Theoretically, this means that even standard IEEE floating-point data written by one machine might not be readable by another. However, on modern standard computers (i.e., implementing IEEE 754), one may in practice safely assume that the endianness is the same for floating-point numbers as for integers, making the conversion straightforward regardless of data type. (Small embedded systems using special floating-point formats may be another matter however.)

Double-precision examples

$$0 \text{ } 0111111111 \text{ } 00_2 \triangleq 3\text{FF}0 \text{ } 0000 \text{ } 0000$$

$$0000_{16} \triangleq +2^0 \times 1 = 1$$

[illegible]

[illegible]

FFFF₁₆ \triangleq NaN (an alternative encoding of NaN)

$$\begin{aligned} &0 \ 011111111101 \ 01_2 \\ &= 3fd5 \ 5555 \ 5555 \ 5555_{16} \triangleq +2^{-2} \times (1 + 2^{-2} + 2^{-4} + \dots + 2^{-52}) \\ &\approx 1/3 \end{aligned}$$

0 100000000000 100100100001111110110101010000100001000001011010000110000₂
= 4009 21fb 5444 2d18₁₆ ≈ pi

Encodings of qNaN and sNaN are not completely specified in IEEE 754 and depend on the processor. Most processors, such as the x86 family and the ARM family processors, use the most significant bit of the significand field to indicate a quiet NaN; this is what is recommended by IEEE 754. The PA-RISC processors use the bit to indicate a signaling NaN.

By default, $1/3$ rounds down, instead of up like single precision, because of the odd number of bits in the significand.

In more detail:

Given the hexadecimal representation 3FD5 5555 5555 5555₁₆,

Sign = 0

Exponent = 3FD₁₆ = 1021

Exponent Bias = 1023 (constant value; see above)

Fraction = 5 5555 5555 5555₁₆

Value = $2^{(\text{Exponent} - \text{Exponent Bias})} \times 1.\text{Fraction}$ - Note that Fraction must not be converted to decimal here

$= 2^{-2} \times (15\ 5555\ 5555\ 5555_{16} \times 2^{-52})$

$= 2^{-54} \times 15\ 5555\ 5555\ 5555_{16}$

$= 0.333333333333333314829616256247390992939472198486328125$

$\approx 1/3$

Execution speed with double-precision arithmetic

Using double-precision floating-point variables and mathematical functions (e.g., sin, cos, atan2, log, exp and sqrt) are slower than working with their single precision counterparts. One area of computing where this is a particular issue is parallel code running on GPUs. For example, when using NVIDIA's CUDA platform, calculations with double precision take, depending on a hardware, approximately 2 to 32 times as long to complete compared to those done using single precision.^[4]

Implementations

Doubles are implemented in many programming languages in different ways such as the following. On processors with only dynamic precision, such as x86 without SSE2 (or when SSE2 is not used, for compatibility purpose) and with extended precision used by default, software may have difficulties to fulfill some requirements.

C and C++

C and C++ offer a wide variety of arithmetic types. Double precision is not required by the standards (except by the optional annex F of C99, covering IEEE 754 arithmetic), but on most systems, the `double` type corresponds to double precision. However, on 32-bit x86 with extended precision by default, some compilers

may not conform to the C standard and/or the arithmetic may suffer from double rounding.^[5]

Fortran

Fortran provides several integer and real types, and the 64-bit type `real64`, accessible via Fortran's intrinsic module `iso_fortran_env`, corresponds to double precision.

Common Lisp

Common Lisp provides the types `SHORT-FLOAT`, `SINGLE-FLOAT`, `DOUBLE-FLOAT` and `LONG-FLOAT`. Most implementations provide `SINGLE-FLOAT`s and `DOUBLE-FLOAT`s with the other types appropriate synonyms. Common Lisp provides exceptions for catching floating-point underflows and overflows, and the inexact floating-point exception, as per IEEE 754. No infinities and NaNs are described in the ANSI standard, however, several implementations do provide these as extensions.

Java

On Java before version 1.2, every implementation had to be IEEE 754 compliant. Version 1.2 allowed implementations to bring extra precision in intermediate computations for platforms like x87. Thus a modifier `strictfp` was introduced to enforce strict IEEE 754 computations.

JavaScript

As specified by the ECMAScript standard, all arithmetic in JavaScript shall be done using double-precision floating-point arithmetic.^[6]

See also

- IEEE 754, IEEE standard for floating-point arithmetic

Notes and references

1. 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>) (PDF). Archived (<https://web.archive.org/web/20120208075518/http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>) (PDF) from the original on 8 February 2012.
2. Savard, John J. G. (2018) [2005], "Floating-Point Formats" (<http://www.quadibloc.com/comp/cp0201.htm>), *quadibloc*, archived (<https://web.archive.org/web/20180703001709/http://www.quadibloc.com/comp/cp0201.htm>) from the original on 2018-07-03, retrieved 2018-07-16
3. "pack – convert a list into a binary representation" (<http://www.perl.com/doc/manual/html/pod/perlfunc/pack.html>).
4. "Nvidia's New Titan V Pushes 110 Teraflops From A Single Chip" (<https://www.tomshardware.com/news/nvidia-titan-v-110-teraflops,36085.html>). *Tom's Hardware*. 2017-12-08. Retrieved 2018-11-05.
5. "Bug 323 – optimized code gives strange floating point results" (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=323). *gcc.gnu.org*. Archived (https://web.archive.org/web/20180430012629/http://gcc.gnu.org/bugzilla/show_bug.cgi?id=323) from the original on 30 April 2018. Retrieved 30 April 2018.

6. *ECMA-262 ECMAScript Language Specification* (<http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205th%20edition%20December%202009.pdf>) (PDF) (5th ed.). Ecma International. p. 29, §8.5 *The Number Type*. Archived (<https://web.archive.org/web/20120313145717/http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205th%20edition%20December%202009.pdf>) (PDF) from the original on 2012-03-13.
-

Retrieved from "https://en.wikipedia.org/w/index.php?title=Double-precision_floating-point_format&oldid=981522661"

This page was last edited on 2 October 2020, at 20:40 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.