A Brief Review of Floating-Point Arithmetic

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2014

History

- Floating point - system for representing numerals with a string of digits (bits), in which the decimal (radix) point can “float” with respect to the significant digits
- Floating point achieves greater range than fixed point since only the significant digits are stored (as well as the position of the radix point)
- Many different flavors of floating-point existed in computing prior to the adoption of the IEEE-754 standard, made for great difficulty in writing/maintaining portable code

IEEE 754-1985

- Turing award (1989) to W. Kahan (U.C. Berkeley) for design of IEEE Standard for binary floating-point arithmetic (ANSI/IEEE 754-1985, also IEC 60559:1989) and IEEE 854-1987 (decimal).
- Almost universally implemented in general purpose machines (note that the Cell multiprocessor and GPUs prior to nVidia’s Fermi architecture do not).
- Standardize representation, rounding behavior, exception handling.

Floating-point Structure

$$a = (-1)^s \times 2^e \times 1.m$$

- The $1.m$ is *normalized*, while $0.m$ is *denormalized*, m is the mantissa (or significand)
- s is the sign bit
- $e = \text{exponent} - \text{bias}$ is a *biased* exponent, where the bias is $2^{\text{exponent bits}} - 1$ (127 for 32-bit single precision, and 1023 for normalized double precision values, 1022 for denormalized).

Simple Examples

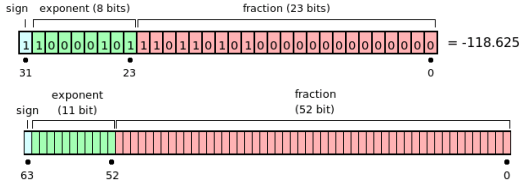


Figure : bit values for IEEE 754 single and double precision.

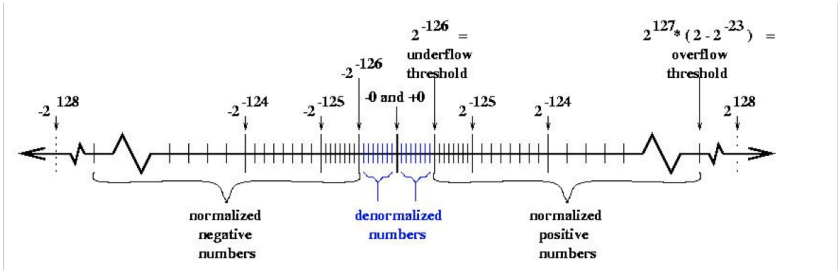
(images courtesy Wikipedia)

Representable Numbers

Form	bits	mantissa bits	macheps	exponent bits	exponent range
Single	32	23	$2^{-24} \sim 10^{-7}$	8	$[2^{-126}, 2^{127}] \sim 10^{\pm 38}$
Double	64	52	$2^{-53} \sim 10^{-16}$	11	$[2^{-1022}, 2^{1023}] \sim 10^{\pm 308}$
Extended*	80	>=64	$2^{-64} \sim 10^{-19}$	>=15	$[2^{-16382}, 2^{16383}] \sim 10^{\pm 4932}$

- *Optional - Intel and compatible processors have extended precision support.
- macheps = "machine epsilon" = $2^{-(\text{mantissa bits} +1)}$, relative error in each operation, also known as the **unit in the last place (ulp)**.

Schematic Representation



Schematic for IEEE single precision (figure courtesy J. Demmel, U.C. Berkeley)

Rounding

Four different rounding modes:

Nearest (default) rounds to nearest value, ties go to nearest value with zero (even) least significant bit

Zero rounds towards zero

Up rounds towards positive infinity

Down rounds towards negative infinity

Exact rounding is specified by IEEE 754, meaning that the result of a floating-point operation is the same as if the real numbers were used, then rounded according to the above rules

Floating-Point Exceptions (FPEs)

Five "exceptions," namely result is not a real number, or out of range:

Overflow	exact result is too large to represent
Underflow	exact result nonzero and too small to represent (usually harmless, but can be problematic for error bounds)
Division by Zero	nonzero divided by zero
Invalid	0/0, SQRT(-1), ...
Inexact	rounding error encountered (quite common)

Response to FPE varies - can stop with an error message (trap), or keep going (default).

FPE Handling

What happens when you encounter a FPE?

- A flag gets set when the exception occurs
- Flag can be read/reset by the user
- Exception flag is also set, indicates whether a **trap** should be initiated:
  - Not trapping is (typically) the default - continues to execute, returning NaN, ($\pm\infty$), or a denorm (between zero and smallest representable number)
  - trap provides user-writable exception handler
  - traps are pretty expensive - but quite useful for debugging

More on FPE Handling

- In a typical scientific application, Overflow, Division by Zero, and Invalid are the worrisome FPEs (usually catastrophic) - basically you should never see a NaN, or $\pm\infty$.
- You can use FPE handling to terminate (or operate under special circumstances) when one of the "big three" exceptions are encountered.
- Operating systems have a tendency to get in the way when handling exceptions - witness kernel traps(!) produced by denormals in IA64 systems.

Floating-Point Operations

- FP operations behave a lot like their real counterparts - but do not be deceived ...
- **Exact** rounding is specified by IEEE 754, meaning that the result of a floating-point operation is the same as if the real numbers were used, then rounded according to the above rules
- One (important) example - FP operations are **not** associative! For example, in floating-point double precision, consider:

$$(10^{20} + 1) - 10^{20} = 0 \neq 1 = (10^{20} - 10^{20}) + 1$$

- Now think about the above example when doing large-scale concurrent (parallel) computation ...
- Impact: compilers (and compiler-writers) have to be very careful about making assumptions that would otherwise greatly simplify the mathematics (ever crank up the optimization level when compiling and have it break your code?) ...

Relaxations

- Most (all?) compilers have flags that will enable you to relax (or turn-off) some IEEE 754 features in order to gain computational efficiency.
- One example, *flush-to-zero* (**-ftz** in the Intel compilers) is an SSE (SSE = Streaming SIMD Extensions, available on all modern Intel and compatible processors) feature that subnormals are not generated and simply replaced by zero.
- One implication of this example is that $x - y = 0$ is no longer a guarantee that $x = y$.
- **Beware** compiler behavior in which the *default* is to neglect language standards (and thereby allow inherently unsafe optimizations).

Intel Compiler

Some illustrative options in the modern versions of Intel's compilers (this is from **ifort** 10.1):

-mp Maintains floating-point precision (while disabling some optimizations). The **-mp** option restricts optimization to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI* and IEEE standards. This is the same as specifying **-fltconsistency** or **-mieee-fp**.
For most programs, specifying this option adversely affects performance. If you are not sure whether your application needs this option, try compiling and running your program both with and without it to evaluate the effects on both performance and precision.

-mp1 Improves floating-point precision and consistency. This option disables fewer optimizations and has less impact on performance than -fltconsistency or **-mp**.

-mieee-fp Enables improved floating-point consistency. Floating-point operations are not reordered and the result of each floating-point operation is stored in the target variable rather than being kept in the floating-point processor for use in a subsequent calculation. This is the same as specifying -fltconsistency or **-mp**. The default, **-mno-ieee-fp**, provides better accuracy and run-time performance at the expense of less consistent floating-point results.

(cont'd):

-fpe<n> Specifies floating-point exception handling for the main program at run-time. You can specify one of the following values for <n>:

0 Floating-point invalid, divide-by-zero, and overflow exceptions are enabled. If any such exceptions occur, execution is aborted. Underflow results will be set to zero unless you explicitly specify **-no-ftz**. On systems using IA-32 architecture or Intel(R) 64 architecture, underflow results from SSE instructions, as well as x87 instructions, will be set to zero. By contrast, option **-ftz** only sets SSE underflow results to zero.
To get more detailed location information about where the error occurred, use **-traceback**.

1 All floating-point exceptions are disabled. On systems using IA-64 architecture, underflow behavior is equivalent to specifying option **-ftz**. On systems using IA-32 architecture and systems using Intel(R) 64 architecture, underflow results from SSE instructions, as well as x87 instructions, will be set to zero.

3 All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero. This is the default; it provides full IEEE support. (Also see **-ftz**.)

Byte Storage

One thing the IEEE floating-point specification does not dictate is the order in which the bytes are stored (note that this is more general than just storing floating-point numbers, it affects any multi-byte entity including binary data):

- **little-endian**, smallest (least significant) byte is stored first. Examples include `x86` and `x86_64` under Linux, Windows, or Mac OS.
- **big-endian**, the largest(most significant) byte is stored first. Examples include POWER under AIX or Linux, PowerPC under Linux or Mac OS, Sparc under Solaris.
- Some architectures can switch between the two (e.g., ARM) in a hardware-dependent fashion.

Note that the etymology of the term **endian** is an amusing literary reference (feel free to look it up)... Also note that the Intel Fortran compiler supports a non-standard option for converting binary formatted files from one endian representation to another.

Simple Arithmetic?

Consider the following results when you simply sum (left-to-right) five double-precision values together (using IEEE standard for representation)[1]:

a_1	a_2	a_3	a_4	a_5	
1.0E21	-1.0E21	17	-10	130	137

That is ok, but the order of operation is significant ...

a_1	a_2	a_3	a_4	a_5	
1.0E21	17	-10	130	-1.0E21	0
1.0E21	-10	130	-1.0E21	17	17
1.0E21	17	-1.0E21	130	-10	120
1.0E21	-10	-1.0E21	130	17	147
1.0E21	17	130	-1.0E21	-10	-10

Now do this in parallel, and with a lot more terms ...

[1] Source: U. Kulisch, *Computer Arithmetic and Validity*, de Gruyter Studies in Mathematics **33**, 250 (2008).

Further Reading

- www.cs.berkeley.edu/~wkahan
- en.wikipedia.org/wiki/Floating_point
- en.wikipedia.org/wiki/IEEE_754-1985