Floating-Point Training
Module 1
The IEEE 754 Standard

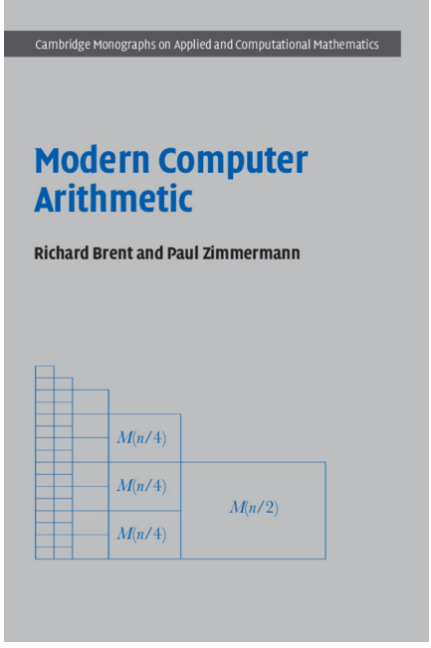
Paul Zimmermann, Inria Nancy

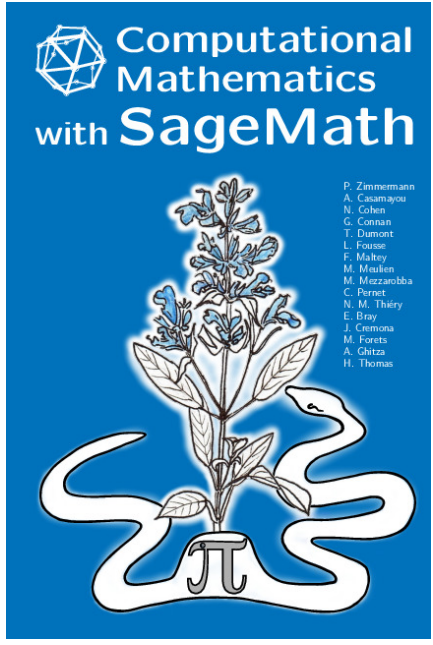
Who I am?

Paul Zimmermann, Senior Researcher at Inria, Nancy, France

Author and main developer (with Vincent Lefèvre) of GNU MPFR, a library for arbitrary precision floating-point computations with correct rounding

Author and main developer (with Alexei Sibidanov) of the CORE-MATH project, a set of efficient correctly-rounded functions for the IEEE 754 formats (single precision, double precision, ...)


Cambridge Monographs on Applied and Computational Mathematics
Modern Computer Arithmetic
Richard Brent and Paul Zimmermann
M(n/4)
M(n/4)
M(n/4)
M(n/2)


Computational Mathematics with SageMath
P. Zimmermann
A. Casamayou
N. Cohen
G. Connan
T. Dumont
L. Fousse
F. Maltey
M. Meulien
M. Mezzarobba
C. Pernet
N. M. Thiéry
E. Bray
J. Cremona
M. Forets
A. Ghitza
H. Thomas

Plan of the training

- **Module 1: The IEEE 754 Standard**
- Module 2: Math Fundamentals
- Module 3: Core Algorithms
- Module 4: Elementary Function Approximation
- Module 5: Software Tools

Module 1: The IEEE 754 Standard

Floating-point representation and encoding (binary32, binary64, binary128)

Extreme values (NaN, Inf, SNaN) and different kinds of zero

Rounding modes

Basic operations (including FMA) and correct rounding

The case of elementary functions

Support of the standard: extended precision, rounding precision control

The Patriot Missile Failure

1991, Gulf War, 28 soldiers killed.

The clock time 1/10 was approximated
in 24-bit fixed point by
0.00011001100110011001100 instead of
0.00011001100110011001100**11001100**
This caused an error after 100 hours since boot of about 0.34 second. Enough to miss the Scud missile, which travels about half a kilometer in that time.

If they had rounded up to
0.00011001100110011001101
the error would have been reduced by 4.
Enough to destroy the Scud missile?

The explosion of Ariane 501

1996, the first Ariane 5 rocket
explodes 40 seconds after start

The 64-bit floating-point speed was converted
to a 16-bit signed integer

Since the speed was larger than 32768
an overflow occurred

Luckily there was a 2nd calculator
in case of issue with the 1st one

But the 2nd calculator produced the same overflow...

History of IEEE 754

Most computers could get zero from $X - Y$ although X and Y were different.

Bizarre tricks like $X = (X + X) - X$...

Intel recruited William Kahan so that all its chips get the same results.

To know more about the pre-754 period and the first days of IEEE-754, read An Interview with the Old Man of Floating-Point.

William Kahan got the ACM's Turing award in 1989.

Revisions of IEEE 754

1985 Initial version (18 pages)

2008 First revision: fused multiply-add (FMA), decimal arithmetic (DPD and BID encodings), recommended correctly rounded functions (70 pages)

2019 Second revision: reproducible floating-point results (84 pages)

2023-2029 Third revision (in progress)

IEEE-754 formats (2019)

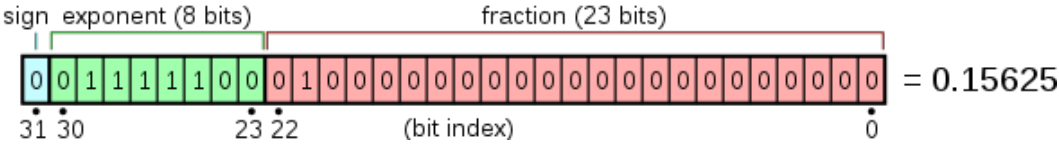
	binary formats			decimal formats	
parameter	binary32	binary64	binary128	decimal64	decimal128
p, digits	24	53	113	16	34
$e_{\max}$	127	1023	16383	384	6144

Binary formats:

Largest positive value $\underbrace{1.111...111}_{p} \cdot 2^{e_{\max}}$

Smallest positive value $2^{-e_{\max}-p+2}$

The binary32 format (single precision)



$$(-1)^s \cdot 2^{e-127} \cdot 1.b_1 b_2 ... b_{22} b_{23}$$

Here $s = 0$, $e = 124$ (unbiased) and $b_i = 0$, except $b_2 = 1$.

We get $2^{-3} \cdot (1 + 2^{-2}) = 0.15625$.

Full correspondance between binary32 and 32-bit integers (`uint32_t`).

The binary32 format: NaN and Infinities

The largest unbiased exponent ($e = 255$) is reserved for NaN and Infinities.

$s = 0$, $e = 255$, $b_1 = \cdots = b_{23} = 0$: $+\infty$

$s = 1$, $e = 255$, $b_1 = \cdots = b_{23} = 0$: $-\infty$

$e = 255$, $b_1 ... b_{23} \neq 0...0$: Not-a-Number (NaN)

Two kinds of NaN: quiet or signalling (not detailed here).

How to recognize NaN: only number for which `x == x` is false.

The binary32 format: signed zeros

There are two zeros: $+0$ and -0.

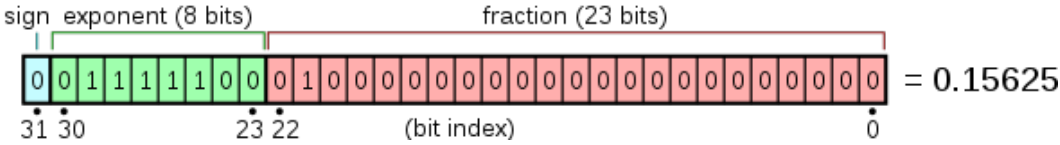
$s = 0$, $e = 0$, $b_1 = \cdots = b_{23} = 0$: $+0$

$s = 1$, $e = 0$, $b_1 = \cdots = b_{23} = 0$: -0

`+0 == -0` is true. By default `x = 0` gives $+0$.

`(+0) + (-0)` yields `+0`, except when rounding towards $-\infty$ where it yields `-0`.

Subnormal numbers



$$(-1)^s \cdot 2^{e-127} \cdot 1.b_1 b_2 ... b_{22} b_{23}$$

The smallest positive normal number is obtained for $e = 1$ and $b_1 ... b_{23} = 0$: it is $x = 2^{-126}$.

The next number is $y = (1 + 2^{-23}) \cdot 2^{-126} = 2^{-126} + 2^{-149}$.

If we compute $y - x$, we get a value that is not representable, should we flush it to zero?

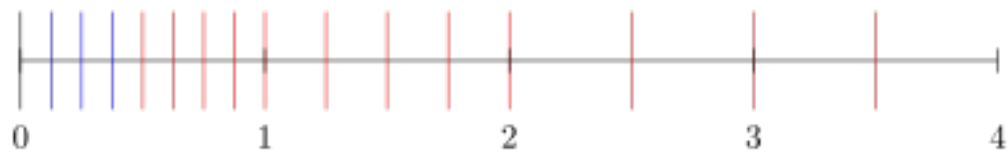
Subnormal numbers

Normal decoding:

$$(-1)^s \cdot 2^{e-127} \cdot 1.b_1 b_2 ... b_{22} b_{23}$$

For $e = 0$, we decode differently:

$$(-1)^s \cdot 2^{-126} \cdot 0.b_1 b_2 ... b_{22} b_{23}$$



The smallest positive number is 2^{-149}.

The binary32 format: extremal values

Smallest positive normal number: 2^{-126} (encoding `0x800000`)

Smallest positive subnormal number: 2^{-149} (encoding `0x1`)

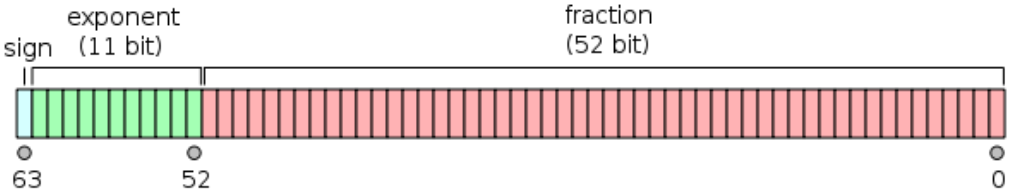
Largest regular number: $2^{128}(1 - 2^{-24})$ (encoding `0x7f7fffff`)

This is the `float` type in the C language.

Hauser Theorem: since every binary32 number x is an integer multiple of 2^{-149}, so is the difference $x - y$ of two binary32 numbers. If $|x - y| \leq 2^{-126}$, then $x - y$ is exact.

Consequence: if $x \neq y$, $y - x$ is an integer multiple of 2^{-149}, and whatever the rounding, is rounded to a non-zero number.

The binary64 format (double precision)



1 bit for sign, 11 bits for exponent, 52 bits for significand (precision 53 with implicit leading bit).

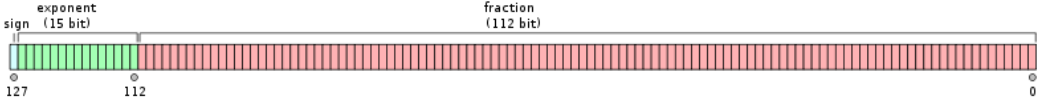
Smallest positive normal number: 2^{-1022}

Smallest positive subnormal number: 2^{-1074}

Largest regular number: $2^{1024}(1 - 2^{-53})$

This is the `double` type in the C language.

The binary128 format (quadruple precision)



1 bit for sign, 15 bits for exponent, 112 bits for significand (precision 113 with implicit leading bit).

Smallest positive normal number: 2^{-16382}

Smallest positive subnormal number: 2^{-16494}

Largest regular number: $2^{16384}(1 - 2^{-113})$

This is the `_Float128` type in the C language.

Hexadecimal format (C99 style)

Decimal numbers like 0.1 are not exactly representable in binary formats.

To avoid ambiguities, it is better to use the hexadecimal format for floating-point constants in source code (or to exchange values).

Three possible conventions (for normal numbers). Example for the binary64 number closest to π:

1. first digit is 1: `0x1.921fb54442d18p+1` (GNU libc)
2. exponent is multiple of 4: `0x3.243f6a8885a3p+0` (GNU MPFR)
3. first digit is largest possible: `0xc.90fdaa22168cp-2`

Hexadecimal format: input/output with SageMath

```
sage: x = RR(pi)
sage: x.hex()
'0x3.243f6a8885a3p+0'
sage: y = RR('0x1.921fb54442d18p+1',16)
sage: y == x
True
```

Decimal formats

IEEE 754 also contains decimal32, decimal64 and decimal128.

These formats are mainly useful for financial applications. Not further considered here.

Rounding modes

IEEE-754 requires four rounding modes (called rounding attributes) for binary formats:

- to nearest, ties to even (RN) [754-2019 also defines ties to away]
- towards zero (RZ)
- towards $+\infty$ (RU)
- towards $-\infty$ (RD)

The next binary32 numbers to $x = 1$ are $y = 1 + 2^{-23}$ and $z = 1 + 2 \cdot 2^{-23}$.

Let $t = 2^{-24}$.

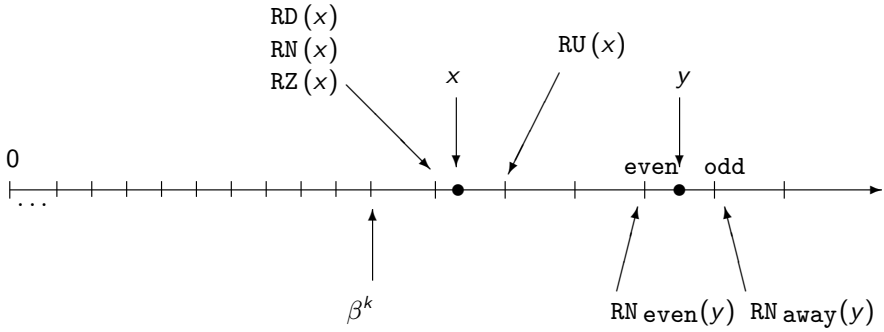
$x + t$ lies exactly in the middle of x and y.

If rounding to nearest even, $x + t$ is rounded to x.

Similarly $y + t$ lies exactly in the middle of y and z.

If rounding to nearest even, $y + t$ is rounded to z.

The standard rounding functions



Here we assume that the real numbers x and y are positive.

Correct rounding

- IEEE 754 specifies the rounding functions RD, RU, RZ, and RN with ties-to-even and ties-to-away (ties-to-zero allowed for some special, not-yet-implemented operations, tries-to-away used only for accounting applications in radix 10);
- the user can choose the rounding function (not always simple);
- the default function is RN ties to even.

Correctly rounded operation: returns what we would get by exact operation followed by rounding.

- correctly rounded $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$ are required;
- correctly rounded sin, cos, exp, ln, etc. are only recommended (not mandatory).

$\rightarrow$ in practice, when the operation `c = a + b` appears in a program, we obtain $c = \mathrm{RN}(a + b)$.

Correct rounding

IEEE-754 (since 1985): Correct rounding for $+$, $-$, $\times$, $\div$, $\sqrt{}$ and some conversions. Advantages:

- if the result of an operation is exactly representable, we get it;
- if we just use the 4 arith. operations and $\sqrt{}$, deterministic arithmetic: one can elaborate algorithms and proofs that use the specifications;
- accuracy and portability are improved;
- playing with rounding towards $+\infty$ and $-\infty \rightarrow$ guaranteed lower and upper bounds: interval arithmetic.

FP arithmetic becomes a structure in itself, that can be studied.

With the math functions work still needs to be done

library	GNU libc	IML	AMD	Newlib	OpenLibm	Musl	Apple	LLVM	MSVC	FreeBSD	ArmPL	CUDA	ROCm
version	2.40	2024.0.2	4.2	4.4.0	0.8.3	1.2.5	14.5	18.1.8	2022	14.1	24.04	12.2.1	5.7.0
acos	**0.523**	0.531	1.36	0.930	0.930	0.930	1.06		0.934	0.930	1.52	1.53	0.772
acosh	2.25	**0.509**	1.32	2.25	2.25	2.25	2.25		3.22	2.25	2.66	2.52	0.661
asin	**0.516**	0.531	1.06	0.981	0.981	0.981	0.709		1.05	0.981	2.69	1.99	0.710
asinh	1.92	**0.507**	1.65	1.92	1.92	1.92	1.58		2.05	1.92	2.04	2.57	0.661
atan	**0.523**	0.528	0.863	0.861	0.861	0.861	0.876		0.863	0.861	2.24	1.77	1.73
atanh	1.78	**0.507**	1.04	1.81	1.81	1.80	2.01		2.50	1.81	3.00	2.50	0.664
cbrt	3.67	0.523	1.53e22	0.670	0.668	0.668	0.729		1.86	0.668	1.79	**0.501**	**0.501**
cos	**0.516**	0.518	0.919	0.887	0.834	0.834	0.948	Inf	0.897	0.834		1.52	0.797
cosh	1.93	**0.516**	1.85	2.67	1.47	1.04	0.523		1.91	1.47	1.93	1.40	0.563
erf	1.43	**0.773**	1.00	1.02	1.02	1.02	6.41		4.62	1.02	2.29	1.50	1.12
erfc	5.19	**0.826**		4.08	4.08	3.72	10.7		8.46	4.08	1.71	4.51	4.08
exp	0.511	0.530	1.01	0.949	0.949	0.511	0.521	**0.500**	1.50	0.949	0.511	0.928	0.929

Largest errors in ulps for double-precision calculation of some math functions. ulp (x) is the distance between two FP numbers in the neighborhood of x (so the largest values should be 0.5 – which is the case with $+$, $-$, $\times$, $\div$, and $\sqrt{\cdot}$).

(Extracted from Gladman, Innocente, Mather, and Zimmermann, *Accuracy of Mathematical Functions...*, Aug. 2024)

Correct rounding bugs

Famous Pentium bug (1994): the binary32 division 4195835/3145727 was wrong by about 700 ulps.

Bug in Newlib 4.1.0: `sqrtf` was not correctly rounded for directed rounding modes.

ulp ("unit in the last place") and ufp ("unit in the first place")

Definition 2 (ulp function)

If $|x| \in [\beta^e, \beta^{e+1})$, then $\mathrm{ulp}(x) = \beta^{\max\{e, e_{\min}\}-p+1}$.

It is the distance between consecutive FP numbers in the neighborhood of x.
Properties:

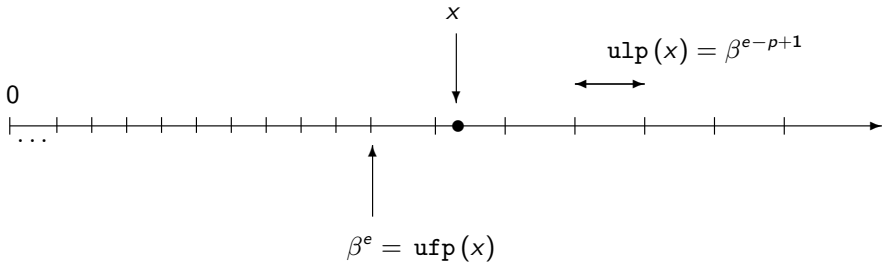
- $|x - \mathrm{RN}(x)| \leq \frac{1}{2}\,\mathrm{ulp}(x)$;
- $|x - \mathrm{RU}(x)|$, $|x - \mathrm{RD}(x)|$, and $|x - \mathrm{RZ}(x)|$ are $< 1\,\mathrm{ulp}(x)$;
- if x is a FP number then it is an integer multiple of $\mathrm{ulp}(x)$.

Function ulp is frequently used for expressing errors.

Definition 3 (ufp function)

If $|x| \in [\beta^e, \beta^{e+1})$, then $\mathrm{ufp}(x) = \beta^e$.

Some useful notions



- this figure: We assume that x is in the normal range: $\beta^{e_{\min}} \leq x \leq \Omega$;
- distance between consecutive FPNs of absolute value $< \beta^{e_{\min}}$: $\beta^{e_{\min}-p+1}$.

Fused multiply-add (FMA)

When computing `a*b+c`, two roundings occur: one for `a*b`, say $r = \circ(ab)$, and one for the addition: $x = \circ(r + c)$.

With the fused multiply-add, only one rounding is performed: $y = \circ(a \cdot b + c)$.

Example: `a=-0x1.7a4cf9d1d29dep-1`, `b=-0x1.95d288e67312p-3`, `c=0x1.0890b6f596598p-3`.

With rounding to nearest, we get `x=0x1.1a3514a59f3c2p-2`, `y=0x1.1a3514a59f3c3p-2`

x is 0.514 ulp far from $ab + c$, y is only 0.486 ulp far.

FMA (2/2)

In the C language: `fma(x,y,z)` stands for $\circ(xy + z)$.

Allows faster (and frequently more accurate) complex multiplication, division, evaluation of polynomials or dot-products.

Facilitates the implementation of correctly-rounded division and square-root using slightly modified Newton-Raphson iterations.

How to control FMA contraction?

- pragma `FP_CONTRACT`
- command-line option `-ffp-contract`
- compiler macro `__FP_FAST_FMA`

Pragma FP_CONTRACT

```
#pragma STDC FP_CONTRACT ON

double a = -0x1.7a4cf9d1d29dep-1;
double b = b=-0x1.95d288e67312p-3;
double c = 0x1.0890b6f596598p-3;
printf ("a*b+c=%la\n", a*b+c);
```

With clang and `-O0`, gives `0x1.1a3514a59f3c2p-2`; with `-O1` or higher, gives `0x1.1a3514a59f3c3p-2`.

This pragma is ignored by gcc (15.2.0), see `BZ #37845`.

Command-line option -ffp-contract

```
double a = -0x1.7a4cf9d1d29dep-1;
double b = b=-0x1.95d288e67312p-3;
double c = 0x1.0890b6f596598p-3;
printf ("a*b+c=%la\n", a*b+c);
```

With `clang -O3` and `-ffp-contract=off`, gives `0x1.1a3514a59f3c2p-2`; with `-ffp-contract=on`, gives `0x1.1a3514a59f3c3p-2`.

For gcc, you need to provide `-march=native`.

Macro __FP_FAST_FMA

Defined if FMA is implemented in hardware.

On x86-64 with gcc, defined up from `-march=x86-64-v3`.

```
#ifdef __FP_FAST_FMA
   d = fma (a, b, c);
#else
   d = a * b + c;
#endif
```

Pragma FENV_ACCESS

Informs the compiler that you might change the rounding mode.

```
#pragma STDC FENV_ACCESS ON

fesetround (FE_TOWARDZERO);
```

Similar to `-frounding-math` on the command line.

Not supported by gcc (see `BZ #34678`).

You can check your floating-point environment with
`http://www.vinc17.org/software/tst-ieee754.c`.

Exceptions

IEEE 754 defines five kinds of exceptions:

- **invalid**: if there is no usefully definable result, for example $\sqrt{-1.0}$ or $(+\infty) + (-\infty)$;
- **division by zero**, for example $1.0/0.0$ or $\log 0.0$;
- **overflow**: when a result exceeds the largest representable number, for example $x + x$ for $x = 2^{1023}$ in double precision;
- **underflow**: when a result is smaller (in absolute value) than the smallest normal number, for example $x \cdot x$ for $x = 2^{-600}$ in double precision;
- **inexact**: when a result is inexact, for example $1.0/3.0$.

When an exception arises, the corresponding flag is set, and it remains set until it is cleared.

Exception handling: the show must go on...

- when an exception occurs: the computation must continue (default behaviour);
- two infinities and two zeros, with intuitive rules: $1/(+0) = +\infty$, $5 + (-\infty) = -\infty \ldots$;
- and yet, something a little odd: $\sqrt{-0} = -0$;
- Not a Number (NaN): result of $\sqrt{-5}$, $(\pm 0)/(\pm 0)$, $(\pm\infty)/(\pm\infty)$, $(\pm 0) \times (\pm\infty)$, NaN $+3$, etc.

$$f(x) = 3 + \frac{1}{x^5}$$

will give the very accurate answer 3 for huge x, even if x^5 overflows.
One should be cautious: behavior of

$$\frac{x^2}{\sqrt{x^3 + 1}}$$

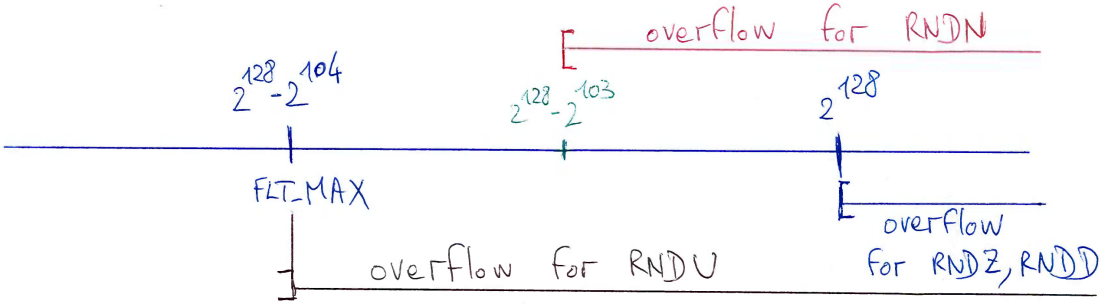
for large x.

Overflow

When the rounded value $\circ(f(x))$, with the target precision and an unbounded exponent range, is beyond the largest positive finite number.

Overflow threshold depends on the rounding mode.

Example for binary32:



Underflow/tininess (1/2)

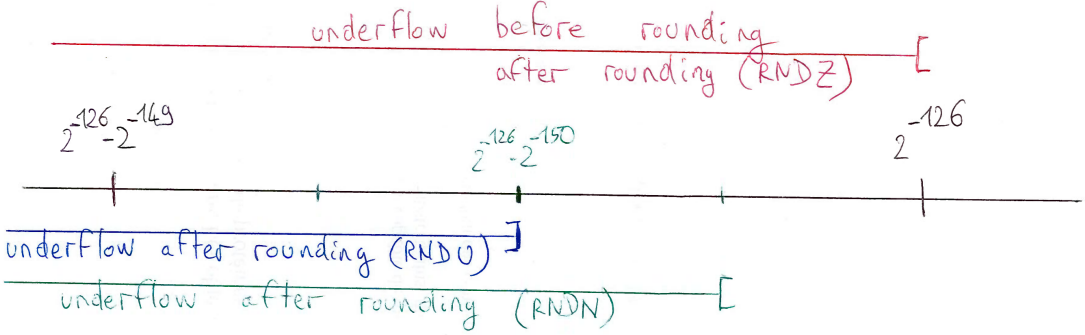
There are two definitions of underflow/tininess, assuming $f(x) > 0$.

Underflow after rounding (example x86_64). When the rounded value $\circ(f(x))$, with full precision and unbounded exponent range, is below the smallest positive normal number.

Underflow before rounding (example aarch64): when $f(x)$ is smaller than the smallest positive normal number.

In both cases (before/after rounding), no underflow if $f(x)$ is exactly representable.

Underflow/tininess (2/2)



The case of elementary functions

Up from the 2008 revision, IEEE 754 has a section on recommended correctly rounded functions.

This says that for example, it is recommended to have a correctly rounded function to compute $\log x$.

It also gives the domain of definition of the function, and the potential exceptions.

But these functions are only recommended...

See more in Module 4.

Language bindings

Since IEEE 754 recommends to have a function to compute $\log x$, and if available it should be correctly rounded, why is the `log` function from `libm.a` not correctly rounded?

This is due to the language bindings.

Each language determines its own bindings of IEEE 754.

The bindings say in particular:

- which type corresponds to each IEEE 754 format
- which name corresponds to each IEEE 754 operations
- how to modify rounding modes

C bindings

For the C language, this is ISO/IEC 60559: Annex F of the C standard.

Annex F contains two tables of operations:

- one for operations with bindings to IEEE 754 operations, which should be correctly rounded (for example $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$, FMA);
- one for other operations, with no binding to IEEE 754 operations or functions (for example log).

Thus the `log` function from the C language has nothing to do with the log function from IEEE 754, and doesn't need to be correctly rounded.

The cr_xxx functions

The next revision of the C standard (C23) will include reserved names `cr_xxx` for correctly rounded functions.

For example, `cr_log` should be a correctly rounded logarithm function, matching the log function from IEEE 754.

The user will have a choice between a fast but maybe inaccurate function `log` and a correctly rounded but maybe slower function `cr_log`.

Support of the standard (in the C language)

The `fenv.h` floating-point environment.

Controlling the rounding modes and exceptions.

The issue of extended precision.

Rounding precision control.

Controlling the rounding modes

The `fenv.h` header file enables to get/set the rounding modes.

```
#include <fenv.h>

fesetround (FE_TONEAREST);
fesetround (FE_TOWARDZERO);
fesetround (FE_UPWARD);
fesetround (FE_DOWNWARD);
```

Warning: the C standard does not fix the values of `FE_TONEAREST`, `FE_TOWARDZERO`, `FE_UPWARD` and `FE_DOWNWARD`! When using the same program with different libraries, you should recompile.

fenv.h (1/2)

```
#include <fenv.h>

fesetround (FE_DOWNWARD);
printf ("1.0/3.0 = %la\n", 1.0 / 3.0);
fesetround (FE_UPWARD);
printf ("1.0/3.0 = %la\n", 1.0 / 3.0);
```

Compile with `-frounding-math`:

```
1.0/3.0 = 0x1.5555555555555p-2
1.0/3.0 = 0x1.5555555555556p-2
```

Other rounding modes are `FE_TONEAREST` and `FE_TOWARDZERO`.

fenv.h (2/2)

To test and clear exceptions: `fetestexcept` and `feclearexcept`.

Clear the underflow and overflow exceptions:

```
feclearexcept (FE_UNDERFLOW | FE_OVERFLOW);
```

Query the inexact exception:

```
fetestexcept (FE_INEXACT);
```

Other exceptions are `FE_DIVBYZERO` and `FE_INVALID`.

The issue of extended precision

Intel implemented an extended double-precision format in its x87 coprocessor.

This format has 1 bit for sign, 15 bits for exponent, and 64 bits for the significand, thus 80 bits in total.

Usually stored on 96 bits on 32-bit processors: ldbl96.

Contrary to the other formats, there is no implicit bit.

This is usually available as `long double` in C.

On some processors, double computations are done using extended double registers, and converted back to double only when required.

This induces a double rounding problem (cf Module 2).

We might get different results compared to all computations done in double precision.

Double rounding issue

Example (credit Vincent Lefèvre): $x = 2^{53} + 2$, $y = 1 - 2^{-16}$.

Rounding in double precision: $x + y = 2^{53} + 3 - 2^{-16}$ is rounded to $2^{53} + 2$.

Rounding in extended precision: $x + y$ is first rounded to $2^{53} + 3$, then to $2^{53} + 4$.

Double rounding example

```
sage: x = RR(2^53+2)
sage: y = RR(1-2^-16)
sage: z.hex()
'0x2.0000000000002p+52'
sage: z.exact_rational() == 2^53+2
True

sage: R64 = RealField(64)
sage: t = RR(R64(x)+R64(y))
sage: t.hex()
'0x2.0000000000004p+52'
sage: t.exact_rational() == 2^53+4
True
```

Type punning

```
#include <stdint.h>

typedef union {double f; uint64_t u;} b64u64_u;

b64u64_u v = {.f = 1.1};

printf ("v.u = 0x%lx\n", v.u);
0x3ff199999999999a
```

Printing and reading floating-point numbers

Printing in hexadecimal:

```
double x = 0.1;
printf ("%la\n", x);
0x1.999999999999ap-4
```

Reading in hexadecimal:

```
char s[] = "0x1p-1074";
sscanf (s, "%la", &x);
printf ("%la\n", x);
0x0.0000000000001p-1022
```

Strange facts with IEEE 754

There are two zeros $+0$ and -0. They have a different encoding but they are equal:

```
float pos_zero = +0.0f, neg_zero = -0.0f;
printf ("pos_zero==neg_zero: %d\n", pos_zero==neg_zero);
pos_zero==neg_zero: 1
```

Not-a-Number (NaN) differs from itself

```
float inf = 1.0f / 0.0f;
float nan = inf - inf;
printf ("inf=%a nan=%a\n", inf, nan);
inf=inf nan=-nan
printf ("inf==inf: %d\n", inf==inf);
inf==inf: 1
printf ("nan==nan: %d\n", nan==nan);
nan==nan: 0
```

Current revision of IEEE 754

Working-group started on November 20, 2023, due for 2029.

Videoconference every two weeks (about 30-40 participants).

Mailing list `STDS-754@LISTSERV.IEEE.ORG` (very active).
`https://listserv.ieee.org/cgi-bin/wa?A0=STDS-754`

Minutes are here: `https://docs.google.com/document/d/121wxt_j_WPqzLzgx7XRs10qBuMw9iHQri5E5zrGRIzY`

Worklist: `https://docs.google.com/spreadsheets/d/1brwilcj_U8ixAHtiap4snj30hgsr_9-OJ97BcjWFLVg`

Beyond the standard

- binary16 and bfloat16
- mini-floats
- unums and posits

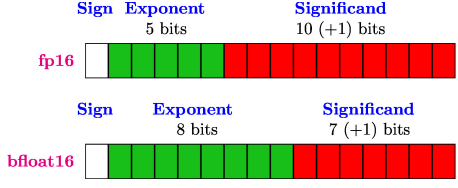
binary16 and bfloat16

IEEE 754-2019 defines only three binary basic formats: binary32, binary64, and binary128.

It also defines binary16 (half precision) as interchange format (but not binary8).

Binary16 has a precision of 11 bits, 5 bits of exponent (largest value $2^{16}(1 - 2^{-11})$, smallest positive value 2^{-24}).

There is also bfloat16, with a precision of 8 bits, and an exponent of 8 bits (brain floating-point). It has the same exponent range as binary32. Heavily used in machine learning.



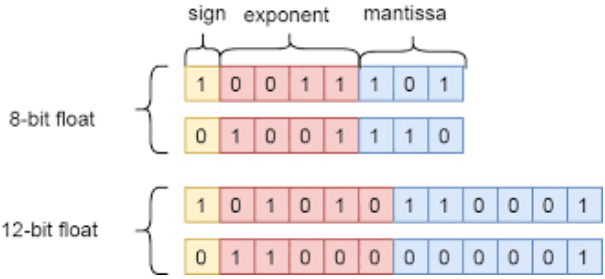
Mini-floats

We can define 8-bit floating-point numbers.

For example the 1.4.3 format:

- 1 bit of sign
- 4 bits of exponent
- 3 bits of significand (+1 implicit)

With an exponent bias of 6, this format can represent numbers from 2^{-9} to about 2^{8}.



Unums and posits

Unums were invented by John Gustafson.

Binary32: 1 sign bit, 8 exponent bits, 23 significant bits. Fixed exponent range.

Some applications have all numbers in a small exponent range and could benefit from more significant bits.

Some applications (combinatorics) require a larger exponent range.

Unum's idea: reserve a few bits to store the size of the exponent.

Posit: hardware-friendly version of unums.

Posit = sign + regime + exponent + significand.

Example: 4 bits of exponent size (k), 1 bit of sign, k bits of exponent, 27-k bits of significand.

$k = 1$: 1 bit of exponent, 26 significant bits

$k = 15$: 15 bits of exponent, 12 significant bits.

See `https://posithub.org/docs/posit_standard.pdf`

Posit32

Max significant bits: 28

Smallest positive number: 2^{-120}

Largest positive number: 2^{120}

S (sign)	R (regime)	E (exponent)	F (fraction)

1. if $S = 0$ and all other fields are 0, then $v = 0$
2. if $S = 1$ and all other fields are 0, then $v = \mathrm{NaR}$
3. otherwise $v = (1 - 3S + F) \times 2^{(-1)^S(4R+E+S)}$

The regime has r identical bits followed by a different bit. It represents $R = -r$ if the first bit is 0, $R = r - 1$ if the first bit is 1.

Example for $v = 2^{-120}$: $S = 0$, $R = -30$, $E = 0$, $F = 0$.

Exercises

Exercise 1: what is the `uint32_t` encoding of $-x$, where x is the binary32 number closest to π?

Exercise 2: what is the binary32 number having 3^{20} as `uint32_t` encoding?

Exercise 3: in binary64, find a, b, c such that $\circ(\circ(ab) + c)$ differs by more than 100 ulps from $\circ(ab + c)$.

Exercise 4: in binary32, is it possible to have as result 2^{-126} while underflow is raised? If yes, find a concrete example. Is it possible to have as result an absolute value smaller than 2^{-126} with no underflow?

Exercise 5: find an operation that raises underflow before rounding but not underflow after rounding. Check with `fetestexcept` on aarch64 and x86_64. Is the converse possible?

Exercise 6: using the rounding modes toward $\pm\infty$, compute using interval arithmetic a rigorous enclosure of Rump's "polynomial" for $a = 77617$ and $b = 33096$ using binary32, binary64 and MPFR with larger precision:

$$p(a, b) = 21b^2 - 2a^2 + 55b^4 - 10a^2b^2 + \frac{a}{2b}$$

Hint: you can use interval arithmetic in SageMath.

```
sage: R = RealIntervalField(100)
sage: a=R(77617); b=R(33096)
sage: r=a/(2*b)
sage: r
1.1726039400531786318588349045?
sage: r.lower().hex()
'0x1.2c2fc595b06beb74a518f018cp+0'
sage: r.upper().hex()
'0x1.2c2fc595b06beb74a518f018ep+0'
```

Exercise 7: implement a function that given a non-zero binary64 number, returns the adjacent binary64 number away from zero.

Exercise 8: implement a function that given a 32-bit integer n, constructs the binary32 number whose encoding is n, without using type punning. Then write the same function with type punning, and compare by exhaustive search the two functions.

Takeaway message

IEEE-754 arithmetic is well-defined (except for elementary functions, see module 4).

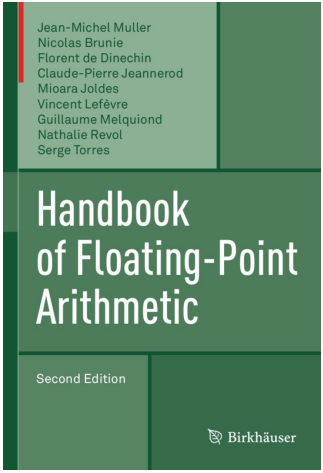
As a consequence, there is only one possible output for each basic operation ($+, -, \times, \div, \sqrt{\cdot}$, FMA)

It is now well supported in the C language (`fenv.h`).

Now even possible to make proofs of floating-point algorithms (more in modules 2-3).

There are nice software tools to play with floating-point numbers (see module 5).

References



The IEEE 754 standard
2019 (latest revision)

What Every Computer Scientist
Should Know About Floating-Point
Arithmetic, David Goldberg, 1991

An Interview with the Old Man
of Floating-Point
by Charles Severance 1998

Some disasters caused by numerical errors
web page maintained by Kees Vuik

Questions, comments ?

Paul.Zimmermann@inria.fr