Que calcule vraiment un ordinateur?

Florent de Dinechin
(ex-) projet AriC

Introduction

Facts and figures

Some common misconceptions about floating point

The floating-point software/hardware stack

Conclusion

Introduction

Introduction

Facts and figures

Some common misconceptions about floating point

The floating-point software/hardware stack

Conclusion

This talk is mostly about floating-point

Do you know there is a standard for floating-point?
It is called IEEE-754, and it is quite nice. For instance,

Correct rounding to the nearest

The basic operations (noted $\oplus$, $\ominus$, $\otimes$, $\oslash$), and the square root should return the FP number closest to the exact mathematical result.

If you think of it, this is the best that the format allows

Nice properties :

- Rounding is monotonic
- ...

However and nevertheless,

Let us compile the following C program on my Debian standard-respecting computer:

```
float ref, index;

ref = 169.0 / 170.0;

for (i = 0; i < 250; i++) {
  index = i;
  if (ref == (index / (index + 1.0))  )  break;
}

printf("i=%d\n",i);
```

First conclusion

Equality test between FP variables is dangerous.

Or,

If you can replace `a==b` with `(a-b)<epsilon` in your code, do it!

A physical point of view

Given two coordinates (x, y) on a snooker table,
the probability that the ball stops at position (x, y) is always zero.

Still, on this expensive laptop, FP computing is not straightforward, even within such a small program.

Go fetch me the person in charge

Who is in charge of floating-point?

- The processor
  - has internal FP registers,
  - performs basic FP operations,
  - raises exceptions,
  - writes results to memory.

Who is in charge of floating-point?

- The processor
- The operating system
  - handles exceptions
  - computes functions/operations not handled directly in hardware
    - most elementary functions (sine/cosine, exp, log, ...),
    - divisions and square roots on recent processors
    - some situations with subnormal numbers
  - handles floating-point status: precision, rounding mode, ...
    - older processors: global status register
    - more recent FPUs: rounding mode may be encoded in the instruction

Who is in charge of floating-point?

- The processor
- The operating system
- The programming language
    - should have a well-defined semantic,
    - ... (detailed in some arcane 1000-pages document)

Who is in charge of floating-point?

- The processor
- The operating system
- The programming language
- The compiler
  - has hundreds of options
  - some of which to preserve the well-defined semantic of the language
  - but probably not by default:
  - Marketing says: default should be *optimize for speed*!

Who is in charge of floating-point?

- The processor
- The operating system
- The programming language
- The compiler
- The programmer
  - ... is in charge in the end.

Of course, eventually, the programmer will get the blame.

Enough toy examples, give us real horror stories

Told to me by a CERN programmer:

- Use the (robust and tested) standard sort function of the STL C++ library
- to sort objects by their radius: according to `x*x+y*y`.
- Sometimes (rarely) segfault, infinite loop...

After long and painful debugging

The sort algorithm works under the following naive assumption:
if $A \not< B$, then, later, $A \geq B$

- `x*x+y*y` inlined and compiled differently at two points of the program,
- sometimes (depending on register allocation) comparison returns $=$, sometimes returns $<$
- enough to break the assumption (horribly rarely).

This talk is about avoiding such problems.

Facts and figures

Introduction

Facts and figures

Some common misconceptions about floating point

The floating-point software/hardware stack

Conclusion

IEEE 754 in one slide

This standard, defined in 1985, revised in 2008 (new stuff *in italic* below), specifies

- floating-point formats:
  - on *16-bits,* 32-bits, 64-bits, *128-bits*
  - in binary $(1.0110100101101010110101 1 \cdot 2^{13})$ and *decimal* $(3.543672 \cdot 10^{-2})$
  - exceptional cases: ± 0, $\pm\infty$, NaN (Not a Number), subnormals
- floating-point rounding modes
  - to the nearest, up, down, towards zero
- operations
  - $+$, $-$, $\times$, $\div$, $\sqrt{\ }$, *fused multiply-and-add*
  - format conversions
  - *recommended elementary functions*
  - *recommended sums, sums of products, sums of square*
- *control on the reproducibility/performance trade-off*

Arithmetic hardware in the typical processor

	desktop processor (Pentium etc)	*embedded processor (ARM)*
integer	64 bits, 32 bits, 16 bits, 8 bits $+, -, \times, /$	32 bits $+, -, \times$
floating-point	64 bits, 32 bits $+, -, \times, /, \sqrt{\ }$	32 bits $+, -, \times$

Floating point formats

- 64 bits means 53-bit mantissa ("double precision")
- 32 bits means 24-bit mantissa ("single precision")

A parenthesis on good language design

Consider the names chosen for the numeric types in C:

- `char` (the 8-bit integer) is an abbreviated noun (character) from typography
  - `unsigned char` ???
  - you can add two `char` AB
- `int` is an abbreviated noun (integer) from mathematics
  - although 2147483647 $+1 =$ -2147483648
- `short` and `long` are adjectives
- `float` is a verb, at least it is a computer term
- `double` means double what?
- `long double` is not even syntactically correct in english
  (and, in case you ask, not the same number of bits as a `long int`)

After so much nonsense, if you're lost, it is not your fault

Single precision spoils us

Single precision provides 7 decimal digits of accuracy

Does Angry Bird really need to compute to an accuracy of 10^{-7} the trajectory input using your fat fingers?

Double precision spoils us

Double-precision provides 15 digits of accuracy

Count the digits in the following

- Definition of the second: *the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium 133 atom.*
- Definition of the metre: *the distance travelled by light in vacuum in 1/299,792,458 of a second.*
- Most accurate measurement ever (another atomic frequency) to 14 decimal places
- Most accurate measurement of the Planck constant to date: to 7 decimal places
- The gravitation constant G is known to 3 decimal places only
  and Voyager made it to Neptune nevertheless

Parenthesis: integer precision spoils us, too

In `float`, $10^9 + 1.0 = 10^9$ (recall definition of IEEE-754 rounding).

Money is and will always be fixed-point

Even when you are very very very rich,
when you pay a small croissant with your credit card,
your bank will make sure this euro is not lost to rounding...

Count the digits in the following

The richest man in the world owns	7,500,000,000,000 cents
A 64-bit integer holds	18,446,744,073,709,551,615 cents

Why then binary64 (double precision)?

A ploy by IBM/Intel/AMD to sell us expensive over-powered chips?

- This PC computes 10^9 operations per second (1 GFlops)

An allegory due to Kulisch

- print the numbers in 100 lines of 5 columns double-sided:
  1000 numbers/sheet
- 1000 sheets $\approx$ a heap of 10 cm
- 10^9 flops $\approx$ heap height speed of 100m/s, or 360km/h
- A teraflops (10^{12} op/s) prints to the moon in one second
- Current top 500 computers reach the petaflop (10^{15} op/s)

- each operation may involve a relative error of 10^{-16},
  and they accumulate.

Doesn't this sound wrong?

We would use these 16 digits just to accumulate garbage in them?

But don't worry

You don't need a petaflops computer to compute completely wrong.

Example: floating-point addition is not associative

$(10^9 + 1.0) - 10^9$ versus $((10^9 - 10^9) + 1.0)$

Some common misconceptions about floating point

Introduction

Facts and figures

Some common misconceptions about floating point

The floating-point software/hardware stack

Conclusion

Many reasons to be afraid

From Kahan's lecture notes (on the web):

1. What you see is often not what you have.
2. What you have is sometimes not what you wanted.
3. If what you have hurt you, you will probably never know how or why.
4. Things go wrong too rarely to be properly appreciated, but not rarely enough to be ignored.

Common misconception 0

Floating-point numbers are real numbers

- ⊕ Of course they are, since they are rationals ($\pm M \cdot 2^e$).
- ⊖ However, many properties on the reals are no longer true on the floating-point numbers

A perfectly sensible floating-point program (Malcolm-Gentleman)

```
A := 1.0;
B := 1.0;
while ((A+1.0)-A)-1.0 = 0.0
   A := 2 * A;
while ((A+B)-A)-B <> 0.0
   B := B + 1.0;
return(B)
```

Common misconception 0.5

All rational numbers can be represented as floating-point numbers

$1/3$ cannot. Worst, 0.1 or 0.01 cannot either.

A killer bug

In 1991, a Patriot missile failed to intercept a Scud missile, and 28 people were killed.

- The code worked with time increments of 0.1 s.
- But 0.1 is not representable in binary.
- In the 24-bit format used, the number stored was 0.099999904632568359375
- The error was 0.0000000953.
- After 100 hours = 360,000 seconds, time is wrong by 0.34s.
- In 0.34s, a Scud moves 500m

Of course in this case the main bug, still unfixed, is the human nature's propension to war.

Reboot your computer every day: If you don't know why, he does

"It makes me nervous to fly on airplanes since I know they are designed using floating-point arithmetic."

Alston Householder

(... well, now they are *piloted* using floating-point arithmetic...)

Patriot-like problems have been discovered in civilian air traffic control systems, after near-miss incidents.

We should all be nervous that elementary knowledge of floating-point arithmetic is not taught to all programmers.

Test: which of the following increments should you use?

10 5 3 1 0.5 0.25 0.2 0.125 0.1

Common misconception 1

Floating-point arithmetic is fuzzily defined, programs involving floating-point should ne be expected to be deterministic.

- ⊕ Since 1985, there has been an IEEE standard for floating-point arithmetic.
- ⊕ Everybody agrees it is a good thing and will do his best to comply
- ⊖ ... but full compliance requires more cooperation between processor, OS, languages, and compilers than the world is able to provide.
- ⊖ Besides full compliance has a cost in terms of performance.

Floating-point programs may be deterministic and portable... but not without work.

Common misconception 2

A floating-point number somehow represents an interval of values around the “real value”.

- ⊕ An FP number only represents itself (a rational), and that is difficult enough
- ⊖ If there is an epsilon or an incertainty somewhere in your data, it is your job (as a programmer) to model and handle it.
- ⊕ This is much easier if an FP number only represents itself.

The computer has no clue about the “real value”

There is no way to express it in C or Fortran or Python!

I suspect, too often, the programmer has no clue either

… probably the first issue to address.

Common misconception 3

All floating-point operations involve a (somehow random) rounding error.

- ⊕ Many are exact, we know who they are, and we may even force them into our programs
- ⊕ Since the IEEE-754 standard, rounding is well defined, and you can do maths about it

Correct rounding

Operations return a result as if it had been computed to infinite precision, then rounded to the nearest available floating-point number.

Rounding is not random: it is the best the format allows.

Examples of exact operations

Decimal, 4 digits of mantissa

- $4.200 \cdot 10^1 \times 1.000 \cdot 10^1 \quad = \quad 4.200 \cdot 10^2$
- $4.200 \cdot 10^1 \times 1.700 \cdot 10^6 \quad = \quad 7.140 \cdot 10^7$
- $1.234 + 5.678 \quad = \quad 6.912$
- $1.234 - 1.233 \quad = \quad 0.001 \quad = \quad 1.000 \cdot 10^{-3}$

My first cancellation

$$1.234 - 1.233 \quad = \quad 0.001 \quad = \quad 1.000 \cdot 10^{-3}$$

- On one hand, this operation is exact
  - if I consider that a floating-point number represents only itself
- On the other hand, the 0s in the mantissa of the result are probably meaningless
  - if I consider that, in the "real world", my two input numbers would have had digits beyond these 4.

So, is this situation good or bad ?
It really depends if the following code depends on these meaningless zeroes...

Example of property based on correct rounding

Theorem (Fast2Sum algorithm)

Let a and b be floating-point numbers such that $|a| \geq |b|$, and consider the following algorithm, where $\oplus$ and $\ominus$ are IEEE-754 floating-point operations:

```
$s \leftarrow a \oplus b$
$e \leftarrow b \ominus (s \ominus a)$
```

This algorithm computes two floating-point numbers s and e that satisfy the following:

- $s + e = a + b$ *exactly;*
- *s is the floating-point number that is closest to* $a + b$.

In other words:

- The rounding error in a floating-point addition is always a floating-point number itself
- It can be computed easily

Misconception 4:
16 digits should be enough for anybody

See slide “double precision spoils us”

Variants of misconceptions 4

If I need 3 significant digits in the end,
I shouldn't worry about accuracy.

- We've seen two ways to destroy 15 digits in two operations
  - It will hurt you if you do not expect it
  - It is easy to avoid being hurt if you expect it
- In large computations, errors add up

Yet another variant: `PI=3.1416` at the beginning of you program

- ⊖ to compute sine correctly, I need to store 1440 bits (420 decimal digits) of $1/\pi$...
- ⊖ Consider $\sin(2\pi Ft)$ as time passes (Patriot bug again)...

Common misconception 5

$$\frac{\text{Estimated diameter of the Universe}}{\text{Planck length}} \approx 10^{62} \quad ;$$

A double-precision FP number holds numbers up to 10^{308};
No need to worry about over/underflow

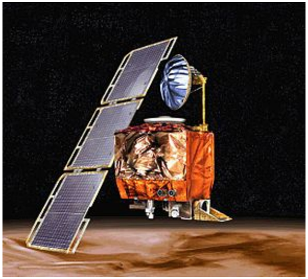
- ⊖ Over/underflows do happen in real code:
  - geometry (very flat triangles, etc)
  - statistics/probabilities
  - intermediate values, approximation formulae
  - ...
- ⊖ it will happen to you if you do not expect it
- ⊕ It is relatively easy to avoid if you expect it

Example: behaviour of $\dfrac{\sqrt{x}}{1+x^2}$

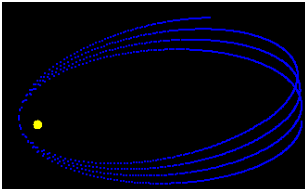
Common misconceptions 6

My good program gives wrong results, it's because of approximate floating-point arithmetic.

- Mars Climate Orbiter crash



- Bad time discretization in a naive two-body simulation



The floating-point software/hardware stack

Introduction

Facts and figures

Some common misconceptions about floating point

The floating-point software/hardware stack

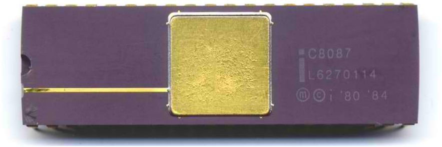
Conclusion

If I'm late I should skip directly to slide 45

The common denominator of modern processors

- Hardware support for
  - addition/subtraction and multiplication
  - in single-precision (binary32) and double-precision (binary64)
  - SIMD versions: two binary32 operations for one binary64
  - various conversions and memory accesses
- Typical performance (for one SIMD way):
  - 3-7 cycles for addition and multiplication, pipelined (1 op/cycle)
  - 15-50 cycles for division and square root,
    hard or soft, not pipelined (1 op / n cycles).
  - 50-500 cycles for elementary functions (soft)

Keep clear from the legacy IA32/x87 FPU



- It is slower than the (more recent) SSE2 FPU
- It is more accurate ("double-extended" 80 bit format), but at the cost of entailing horrible bugs in well-written programs
- the bane of floating-point between 1985 and 2005

The SSE2 unit of current IA32 processors

Available for all recent x86 processors (AMD and Intel):

- An additional set of 128-bit registers
- An additional FP unit able of
  - 2 identical binary64 FP operations in parallel, or
  - 4 identical binary32 FP operations in parallel.
- clean and standard implementation
  - subnormals trapped to software, or flushed to zero
  - depending on a compiler switch (gcc has the safe default)

And soon AVX: multiply all these numbers by 2

(256-bit registers, etc)

Quickly, the Power family

Power and PowerPC processors, also in IBM mainframes and supercomputers

- No floating-point adders or multipliers
- Instead, one or two FMA: Fused Multiply-and-Add
- Compute $\circ(a \times b + c)$:
  - faster: roughly in the time of a FP multiplication
  - more accurate: only one rounding instead of two
  - enable efficient implementation of division and square root
- Standardized in IEEE-754-2008
  - but not yet in your favorite language

FMA: the good

- Compute $\circ(a \times b + c)$:
  - faster: roughly in the time of a FP multiplication
  - more accurate: only one rounding instead of two
  - enable efficient implementation of division and square root
- All the modern FPUs are built around the FMA:
  ARM, Power, IA64, all GPGPUs, and even latest Intel and AMD processors.
- enables classical operations, too...
  - Addition: $\circ(a \times 1 + c)$
  - Multiplication: $\circ(a \times b + 0)$

FMA: ...the bad and the ugly

$$\circ(a \times b + c)$$

Using it breaks some expected mathematical propertie

- Loss of symmetry in $\sqrt{a^2 + b^2}$
- Worse: $a^2 - b^2$, when $a = b$:
  $\circ(\ \circ(a \times a) - a \times a\ )$
- Worse: `if` $b^2 \geq 4ac$ `then` (...) $\sqrt{b^2 - 4ac}$

Do you see the sort bug lurking?

By default, gcc disables the use of FMA altogether
(except as $+$ and $\times$)

(compiler switches to turn it on)

Evaluation of an expression

Consider the following program, whatever the language

```
double x; float a,b,c,d;
x = a+b+c+d;
```

Two questions:

- In which order will the three addition be executed?
- What precision will be used for the intermediate results?

Fortran, C, Java and Python have completely different answers.

Evaluation of an expression

```
double x; float a,b,c,d;
x = a+b+c+d;
```

- In which order will the three addition be executed?
  - With two FPUs (dual FMA, or SSE2, ...), $(a+b)+(c+d)$ faster than $((a+b)+c)+d$
  - If a, c, d are constants, $(a+c+d)+b$ faster.
  - (here we should remind that FP addition is not associative Consider $2^{100}+1-2^{100}$)
  - Is the order fixed by the language, or is the compiler free to choose?
  - Should additions and multiplications be merged in FMA, and how?

Evaluation of an expression

```
double x; float a,b,c,d;
x = a+b+c+d;
```

- In which order will the three addition be executed?
- What precision will be used for the intermediate results?
  - *Bottom up* precision: (here all `float`)
    - elegant (context-independent)
    - portable
    - sometimes dangerous: compare `C=(F-32)*(5/9)` and `C=(F-32)*5/9`
  - Use the maximum precision available which is no slower
    - in C, variable types refer to memory locations
    - more accurate result
  - Is the precision fixed by the language, or is the compiler free to choose?

Fortran's philosophy (1)

Citations are from the Fortran 2000 language standard: *International Standard ISO/IEC1539-1:2004. Programming languages – Fortran – Part 1: Base language*

The FORmula TRANslator translates mathematical formula into computations.

Any difference between the values of the expressions `(1./3.)*3.` and `1.` is a computational difference, not a mathematical difference. The difference between the values of the expressions `5/2` and `5./2.` is a mathematical difference, not a computational difference.

Fortran's philosophy (2)

Fortran respects mathematics, and only mathematics.

(...) the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated. Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type may produce different computational results.

Remark: This philosophy applies to both order and precision.

Fortran in details

X,Y,Z of any numerical type, A,B,C of type real or complex, I, J of integer type.

Expression	Allowable alternative form
X+Y	Y+X
X*Y	Y*X
-X + Y	Y-X
X+Y+Z	X + (Y + Z)
X-Y+Z	X - (Y - Z)
X*A/Z	X * (A / Z)
X*Y-X*Z	X * (Y - Z)
A/B/C	A / (B * C)
A / 5.0	0.2 * A

Consider the last line :

- `A/5.0` is actually more accurate `0.2*A`. Do you see why?
- This line is valid if you replace 5.0 by 4.0, and then accuracy is the same
- ... but not if you replace 5.0 by 3.0, do you see why? Does it make sense?

Fortran in details (2)

Fortunately, Fortran respects your parentheses.

In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression.

(this was the solution to the last FP bug of LHC@Home at CERN)

Fortran in details (3)

X,Y,Z of any numerical type, A,B,C of type real or complex, I, J of integer type.

Expression	Forbidden alternative form
I/2	0.5 * I
X*I/J	X * (I / J)
I/J/A	I / (J * A)
(X + Y) + Z	X + (Y + Z)
(X * Y) - (X * Z)	X * (Y - Z)
X * (Y - Z)	X*Y-X*Z

Fortran in details (4)

You have been warned.

The inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions `A*I/J` and `A*(I/J)` may have different mathematical values if I and J are of type integer.

Difference between `C=(F-32)*(5/9)` and `C=(F-32)*5/9`.

Enough standard, the rest is in the manual

(yes, you should read the manual of your favorite language
and also that of your favorite compiler)

The C philosophy

The "C11" standard:
International Standard ISO/IEC ISO/IEC 9899:2011.

- Contrary to Fortran, the standard imposes an order of evaluation
  - Parentheses are always respected,
  - Otherwise, left to right order with usual priorities
  - If you write $x = a/b/c/d$ (all FP), you get 3 (slow) divisions.
- Consequence: little expressions rewriting
  - Only if the compiler is able to prove that the two expressions always return the same FP number, including in exceptional cases

Finally we can fix this program

```
    float ref, index;

    ref = 169.0 / 170.0;

    for (i = 0; i < 250; i++) {
      index = i;
      if (ref == (index / (index + 1.0))  )  break;
    }

    printf("i=%d\n",i);
```

(Note to self: fix in one character because no double rounding in 169.0/170.0.)

C in the gory details

Morceaux choisis from appendix F.8.2 of the C11 standard:

- Commutativities are OK
- `x/2.0` may be replaced with `0.5*x`,
  because both operations are always exact in IEEE-754.
- but `x/5.0` may not be replaced with `0.2*x`
  (C won't introduce the Patriot bug)
- `x*1` and `x/1` may be replaced with `x`
- `x-x` may not be replaced with `0`
  unless the compiler is able to prove that `x` will never be ∞ nor NaN
- Worse: `x+0` may not be replaced with `x`
  unless the compiler is able to prove that `x` will never be -0
  because $(-0) + (+0) = (+0)$ and not (-0)
- On the other hand `x-0` may be replaced with `x`
  if the compiler is sure that rounding mode will be to nearest.
- `x == x` may not be replaced with `true`
  unless the compiler is able to prove that `x` will never be NaN.

Obvious impact on *performance*

Therefore, default behaviour of commercial compiler tend to ignore this part of the standard...
But there is always an option to enable it.

The C philosophy (2)

- So, perfect determinism wrt **order of evaluation**
- Strangely, **intermediate precision** is not determined by the standard: it defines a bottom-up minimum precision, but invites the compiler to take the largest precision which is larger than this minimum, and no slower
- Idea:
  - If you wrote `float` somewhere, you probably did so because you thought it would be faster than `double`.
  - If the compiler gives you `long double` for the same price, you won't complain.

Drawbacks of C philosophy

- Small drawback
  - Before SSE, `float` was almost always double or double-extended
  - With SSE, `float` should be single precision (2-4× faster)
  - Or, on a newer PC, the same computation became much less accurate!
- Big drawbacks
  - The compiler is free to choose which variables stay in registers, and which go to memory (register allocation/spilling)
  - It does so almost randomly (it totally depends on the context)
  - But... storing a `float` variable in 64 or 80 bits of memory instead of 32 is usually slower, therefore (C philosophy) it should be avoided.
  - Thus, sometimes a value is rounded twice, which may be even less accurate than the target precision
  - And sometimes, the same computation may give different results at different points of the program.

The sort bug explained (because `double` promoted to 80 bits)

Quickly, Java

- Integrist approach to determinism: *compile once, run everywhere*
  - `float` and `double` only.
  - Evaluation semantics with fixed order and precision.
  - ⊕ No sort bug.
  - ⊖ Performance impact, but... only on PCs (Sun also sold SPARCs)
  - ⊖ You've paid for double-extended processor, and you can't use it (because it doesn't *run anywhere*)

The great Kahan doesn't like it.

- Many numerical unstabilities are solved by using a larger precision
- Look up *Why Java hurts everybody everywhere* on the Internet

I tend to disagree with him here. We can't allow the sort bug.

Quickly, Python

Floating point numbers
These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow.

You have been warned.

Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

Conclusion

Introduction

Facts and figures

Some common misconceptions about floating point

The floating-point software/hardware stack

Conclusion

A historical perspective

- Before 1985, floating-point was an ugly mess
- From 1985 to 2000, the IEEE-754 standard becomes pervasive, but the party is spoiled by x87 messy implementation WRT extended precision
- Newer instruction sets solve this, but introduce the FMA mess
- 2008 IEEE 754-2008 cleans all this, but adds the decimal mess
- and then arrives the multicore mess (determinism lost forever)

It shouldn't be so messy, should it?

Don't worry, things are improving

- SSE2 has cleant up IA32 floating-point
- Soon (AVX/SSE5) we have an FMA in virtually any processor and we may use the `fma()` to exploit it safely and portably
- The 2008 revision of IEEE-754 addresses the issues of
  - reproducibility versus performance
  - precision of intermediate computations
  - etc
- but it will take a while to percolate to your programming environment

To finish on a positive note

Feel in control

- Floating-point computing is well thought out and well documented.
- Understand the articulation between hardware and software,
- … and then, go RTFM (all of them).

… and ask us questions! Now, or later when floating-point bites you.

To probe further

- D. Goldberg: *What Every Computer Scientist Should Know About Floating-Point Arithmetic* (Google will find you several copies)
- The web page of William Kahan at Berkeley.
- The web page of the AriC group.
- *Handbook of Floating-Point Arithmetic*, by Muller et al.