# Computing with Floating Point

Florent de Dinechin, AriC Project, ENS-Lyon
Florent.de.Dinechin@ens-lyon.fr

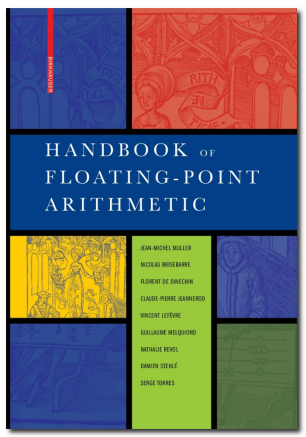
UCBL, 21/02/2012.99999

# First some advertising

To probe further:

- 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.



# Introduction

# Scientific notation

From $9.10938215 \times 10^{-31}$ kg to $6.0221415 \times 10^{23}$ $\text{mol}^{-1}$

- Multiplication algorithm is trivial
  - (but typically involves some rounding)
- Addition algorithm is slightly more complex
  - align the two numbers to the same exponent
  - perform the addition/subtraction
  - optionally, round

## Golden rules (according to my physics teachers)

- The number of digits we write is the number of digits we trust
- Each number has a unit attached to it

# Floating-point in your computer is just that

... with two main differences:

## Binary instead of decimal

Since the Zuse Z1 (1938)

$$1.11111110000110000110000011000 \times 2^{78}$$

(how the exponent is coded is irrelevant)


## The computer doesn't manage the golden rules

- No unit attached (Mars Climate Orbiter crash in 1999)
- The numbers of bits we manipulate is the number of bits we have (correct or wrong)

# Let's be formal

A floating-point number is a rational:

$$x = (-1)^s \times m \times \beta^e$$

- $\beta$ is the radix
  - 10 in your calculator, your bank's computer, and (usually) your head
  - 2 in most computers (binary arithmetic)
- $s \in \{0, 1\}$ is a sign bit
- $m$ is the *mantissa*, a fixed-point number of $p$ digits in radix $\beta$:

$$m = d_0, d_1 d_2 ... d_{p-1}$$

- $e$ is the exponent, a signed integer between $e_{\min}$ and $e_{\max}$

$p$ specifies the precision of the format, $[e_{\min} ... e_{\max}]$ specifies its dynamic.

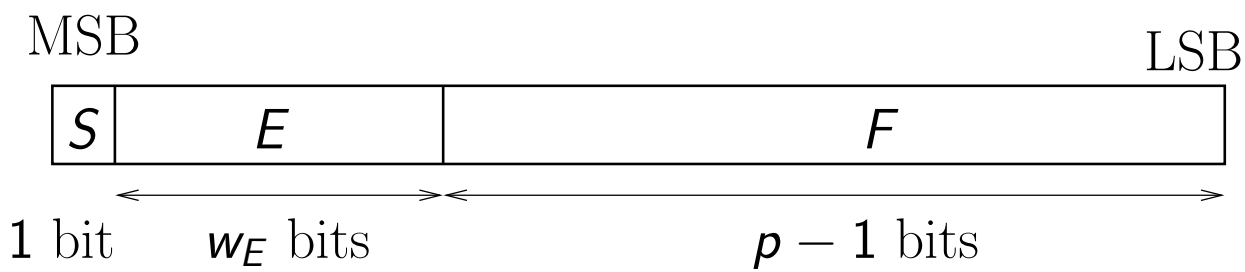
# Normalized representation

- An infinity of equivalent representations:
  - $6.0221415 \times 10^{23}$
  - $60221415 \times 10^{16}$
  - $602214150000000000000000 \times 10^{0}$
  - $0.00000060221415 \times 10^{30}$
- Imposing a unique representation will simplify comparisons
- Which one is best?
  - Leading and trailing zeroes are useless (to the computation)
- The first representation is preferred
  - one and only one non-zero digit before the point
  - then the exponent gives the order of magnitude
- In radix 2, if the first digit is not a zero, it is a one

  no need to store it.

# Mainstream formats of the IEEE-754 standard

| Name (old name) | binary32 (single precision) | binary64 (double precision) |
|---|---|---|
| total size | 32 bits | 64 bits |
| $p$ | 24 | 53 |
| $2^{-p}$ | $\approx 6 \cdot 10^{-8}$ | $\approx 10^{-16}$ |
| $w_E$ | 8 | 12 |
| $e_{\min}, e_{\max}$ | $-126, +127$ | $-1022, +1023$ |
| smallest | $\approx 1.401 \times 10^{-45}$ | $\approx 4.941 \times 10^{-324}$ |
| largest | $\approx 3.403 \times 10^{38}$ | $\approx 1.798 \times 10^{308}$ |



# Non mainstream formats in IEEE754-2008

- binary16 (an exchange format, don't compute with it)
- binary128 (currently unsupported by hardware)
- possibly extended formats
- decimal formats
  - decimal32, decimal64

# The decimal fiasco

Much debated in the early 2000 as the IEEE-754 standard was revised

- intended to support financial calculations
  (interest rates are given in decimal)
- supported in software on intel, in hardware in some IBM mainframes
  - first mess: two different encodings
- money is fixed-point, not floating-point
  - second mess: non-unicity of representation
- My advice:
  - stay clear of decimal numbers,
  - and count your money in a 64-bit integer, it should fit.

# First important message

## Floating point is something well defined and well understood

- The set of floating-point numbers (on 32 or 64 bits) is well defined
- For any real $x$, we may define a function $\circ(x)$ that returns the FP number that is the nearest to $x$
- The operations are also defined to be as good as possible for instance, FP addition of $a$ and $b$ is defined as $\circ(a + b)$

*We can build serious math and serious proofs on top of this*

# Floating-point formats in programming languages

- sometimes `real`, `real*8`,
- sometimes `float`,
- sometimes silly names like `double` or even `long double`

# Parenthesis: good language design

The numeric types in C:

- `char` (the 8-bit integer) is an abbreviated noun (character) from typography
  - `unsigned char` ???
  - you can add two `char` 
- `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

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

*Sorry for that.* *float=binary32*, *double=binary64*

# Common misconceptions

# À tout seigneur, tout honneur

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.
5. Items 1 to 4 do not constitute carte blanche to build floating point any way you like.

# Common misconception 0

## Floating-point numbers are real numbers

- ⊕ Of course they are, since they are rationals.
- ⊖ However, many properties on the reals are no longer true on the floating-point numbers.
  To start with: Floating-point addition is not associative

## 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)
```

# Magnitude graphs

To reason about this kind of programs,

- draw an $x$ axis with the exponents
- position the significands as rectangles of fixed size along this axis
- reason about the position of the result mantissa
- draw the exact results, and the rounded results

## Exercise

Illustrate that floating-point addition is not associative

# Common misconception 0.5

All rational numbers can be represented as floating-point numbers
1/3 cannot. Worst, 1/10, 1/100 etc cannot either.
Remember that FP numbers are binary.
Many bugs in Excel are due to its attempts to hide this fact.

## Exercise

What is the error of representing $\pi$ as a binary32 number?

- define "error"
- compute a tight bound.

# The Patriot bug

In 1991, a Patriot failed to intercept a Scud (28 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

(similar problems have been discovered in civilian air traffic control systems, after near-miss incidents)

**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 not be expected to be deterministic.

- ⊕ 1985: IEEE 754 standard for floating-point arithmetic.
- ⊕ All basic operations must be as accurate as possible.
- ⊕ Supported by all processors and even GPUs
- ⊖ ... 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.
- ⊖ Anyway, parallel computers (multicores) are not deterministic anymore

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

## Common misconception 1.5

A FP program that behaves deterministically probably returns the correct result.

... probably...

# 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)
- ⊖ The computer will not manage the golden rules for you!
- ⊖ 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, and if each operation is as accurate as possible.

If you are able to define accurately the "real value" corresponding to every single variable in your 100,000 lines of code, you definitely know more than the computer.

# Common misconception 3

All floating-point operations involve a (somehow fuzzy) 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

# Examples of exact operations

Decimal, 4 digits of mantissa

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

# 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 ?

Usually good, but bad if the following computation depends on these meaningless digits

# Time for an exercise

## Write a program that solves the quadratic equation

Formulas I learnt in school:

$$\delta = b^2 - 4ac$$

$$\text{if } \delta \geq 0, \quad r = \frac{-b \pm \sqrt{\delta}}{2a}$$

- There are two subtractions here. Can one of them lead to problematic cancellation? In which cases?
- If yes, try and change the formula.

# Solution

- The $-$ in the formula of $\delta$ is harmless
  - we may have a catastrophic cancellation in $\delta$ when $b^2 \approx 4ac$
  - then $\delta << b^2$ so $\sqrt{\delta} << b$
  - the operation following the cancellation is an addition
  - this additions shifts the meaningless digits out of the result
  - so $r \approx \frac{-b}{2a}$ is computed accurately
- There may be a cancellation in $-b \pm \sqrt{\delta}$
  - this happens if $\sqrt{\delta} \approx b$, i.e. $4ac << b^2$
  - the operation following the cancellation is a division
  - this division transfers these meaningless digits to the result
  - (same for a multiplication)
- Solution:
  - test if the $\pm$ is a effective subtraction
  - for instance if $b > 0$ and $\pm = +$,
  - multiply numerator and denominator of $\frac{-b+\sqrt{\delta}}{2a}$ with $b + \sqrt{\delta}$
  - accurate formula in this case:

$$r = \frac{-2c}{b + \sqrt{\delta}}$$

# Misconception 4: 16 digits should be enough for anybody

Double precision (binary64) provides roughly 16 decimal digits.

## 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

# Variants of misconceptions 4

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

- ⊖ Cancellation may destroy 15 digits of information in one subtraction
- ⊖ It will happen to you if you do not expect it
- ⊕ It is relatively easy to avoid if you expect it

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

- ⊕ sometimes it's enough
- ⊖ Consider $\sin(2\pi Ft)$ as time passes...
- ⊖ Your sine implementation needs to store 1440 bits (420 decimal digits) of $1/\pi$...

(I'll have one slide on decimal/binary conversion, don't worry)

# Why then double-precision?

Vendors would sell us hardware that we don't need?

- This PC computes $10^9$ operations per second (1 gigaflops)
- This is a lot. 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) machine builds in one second a pile of paper to the moon.
  - Current top 500 computers reach the petaflop ($10^{16}$ op/s)
- Relationship to precision?

# Where does precision go?

- each operation may involve an error of the weight of the last digit (relative error of $10^{-16}$)
- If you are computing a big sum, these errors add up.
- In a Gflops machine, after one second you have lost 9 digits of your result (remains 6).
- In a petaflops machine, you may have lost all your digits in 0.1s.

Managing this is a big challenge of current HPC

# 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

# Of overflows and infinity arithmetic

## Exercise

You need to compute

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

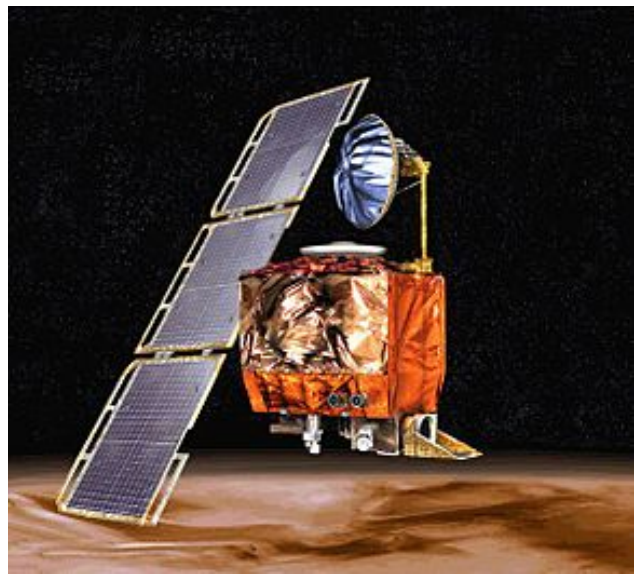
What happens for large values of $x$ ?

- Instead of (large) $\sqrt{x}$ you get 0
  - $x^3$ overflows (to $+\infty$) before $x^2$
  - $\sqrt{+\infty} = +\infty$
  - $\frac{\text{finite}}{+\infty} = 0$
- Here again, the solution is
  - to expect the problem before it hurts you
  - and to protect the computation with a test which returns $\sqrt{x}$ for large values
  - (a more accurate result, obtained faster...)

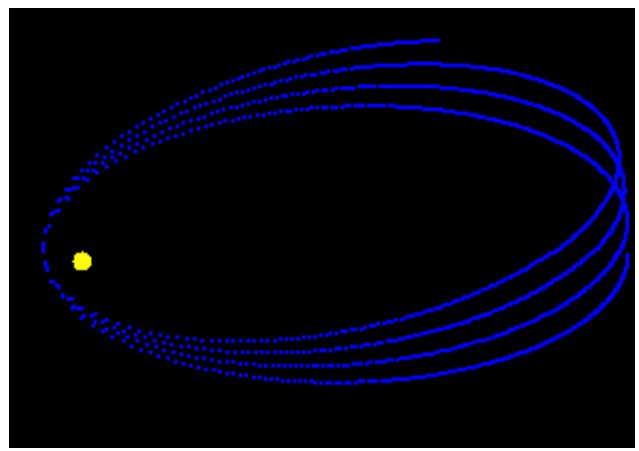
# Common misconceptions 6

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

- Mars Climate Orbiter crash



- Naive two-body simulation

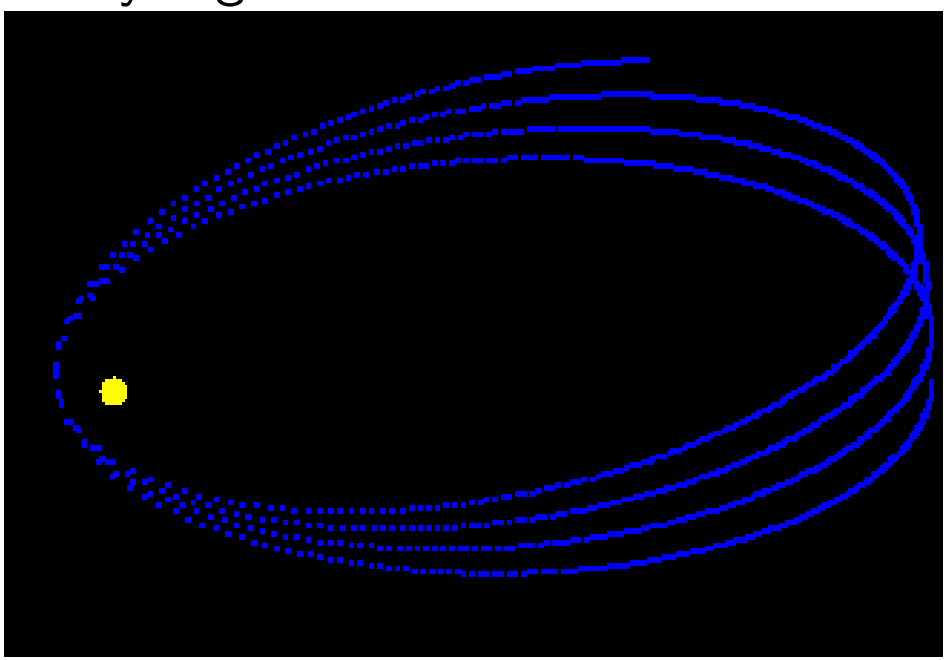


# Arithmetic is not always the culprit

- Ask first-year students to write a simulation of one planet around a sun

$$\begin{cases} \mathbf{x}(t) & := & \mathbf{v}(t)\delta t \\ \mathbf{v}(t) & := & \mathbf{a}(t)\delta t \\ \mathbf{a}(t) & := & \dfrac{K}{||\mathbf{x}(t)||^2} \end{cases}$$

- You always get rotating ellipses
- Analysing the simulation shows that it creates energy.



# Floating-point as it should be: The IEEE-754 standard

# The dark ages of anarchy

In the ancient times (before 1985), there were as many implementations of floating-point as there were machines

- no hope of portability
- little hope of proving results e.g. on the numerical stability of a program
- horror stories : $\arcsin\left(\frac{x}{\sqrt{x^2+y^2}}\right)$ could segfault on a Cray
- therefore, little trust in FP-heavy programs

# Rationale behind the IEEE-754-85 standard

- Enable data exchange
- Ensure portability
- Ensure provability
- Ensure that some important mathematical properties hold
  - People will assume that $x + y == y + x$
  - People will assume that $x + 0 == x$
  - People will assume that $x == y \Leftrightarrow x - y == 0$
  - People will assume that $\frac{x}{\sqrt{x^2+y^2}} \leq 1$
  - ...
- These benefits should not come at a significant performance cost

Obviously, need to specify not only the number formats
but also the operations on these numbers.

# Normal numbers

Desirable properties :

- an FP number has a unique representation
- every FP number has an opposite

## Normal numbers

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

For unicity of representation, we impose $d_0 \neq 0$.
(In binary, $d_0 \neq 0 \Longrightarrow d_0 = 1$: It needn't be stored.)

# Exceptional numbers

Desirable properties :

- representation of 0
- representations of $\pm\infty$ (and therefore $\pm 0$)
- standardized behaviour in case of overflow or underflow.
  - return $\infty$ or 0, and raise some flag/exception
- representations of *NaN*: Not a Number
  (result of $0^0$, $\sqrt{-1}$, ...)
  - Quiet NaN
  - Signalling NaN

# Choice of binary representation

Desirable property: the order of FP numbers is the lexicographical order of their binary representation

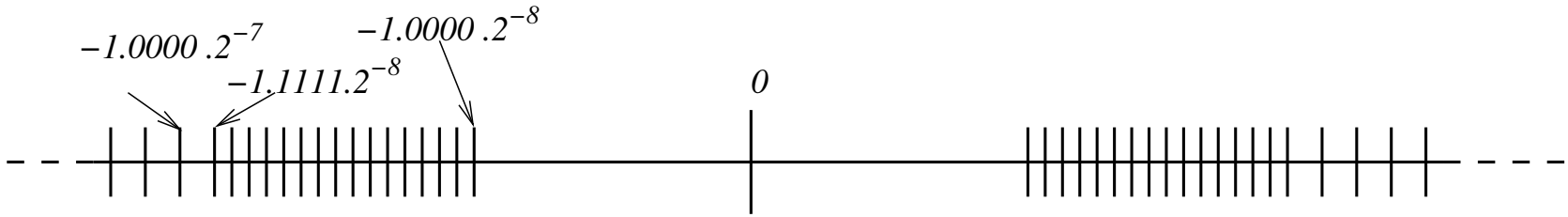
## Binary encoding of positive numbers

- place exponent at the MSB (left of significand)
- infinity is larger than any normal number:
  code it with the largest exponent $111...1_2$
- zero is smaller than any normal number:
  code it with the smallest exponent $000...0_2$
- for normal exponents: biased representation
  - assume $w_E$ bits of exponent
  - exponent field $E \in \{0...2^{w_E} - 1\}$ codes for exponent $e = E - \text{bias}$
  - In IEEE-754, bias for significand in $[1, 2)$ is $\text{bias} = 2^{w_E - 1} - 1 \; = \; 0111...1_2$

How to code NaNs? Significand of infinity? Significand of 0? ...

# Subnormal numbers

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



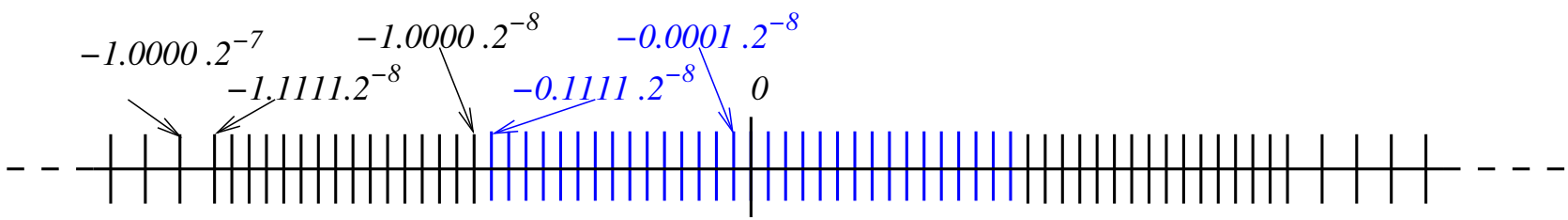

Desirable properties :

- $x == y \Leftrightarrow x - y == 0$
- Graceful degradation of precision around zero

## Subnormal numbers

if $E = 00...0_2$, the implicit $d_0$ is equal to 0:

$$x = (-1)^s \times 2^{e_{\min}} \times 0.m$$



# Complete binary representation (positive numbers)

3 bits of exponent, 4 bits of fraction (4+1 bits of significand)

| exp fraction | value | comment |
|---|---|---|
| 000 0000 | 0 | Zero |
| 000 0001 | $0.0001 \cdot 2^{e_{\min}}$ | smallest positive (subnormal) |
| ... | ... | |
| 000 1111 | $0.1111 \cdot 2^{e_{\min}}$ | largest subnormal |
| 001 0000 | $1.0000 \cdot 2^{e_{\min}}$ | smallest normal |
| ... | ... | |
| 110 1111 | $1.1111 \cdot 2^{e_{\max}}$ | largest normal |
| 111 0000 | $+\infty$ | |
| 111 0001 | NaN | |
| ... | ... | |
| 111 1111 | NaN | |

NextAfter obtained by adding 1 to the binary representation

from 0 to $+\infty$

# Operations

Desirable properties :

- If $a + b$ is a FP number, then $a \oplus b$ should return it
- Rounding should be monotonic
- Rounding should not introduce any statistical bias
- Sensible handling of infinities and NaNs

## 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 mathematical result.

In case of tie, round to the number with an even significand

$\Longrightarrow$ no bias.

An unambiguous choice: this is the best that the format allows

Three other rounding modes: to $+\infty$, to $-\infty$, to 0, with similar correct rounding requirement (and no tie problem).

# Oh, and by the way the standard should be implementable

(back in 1985 this was a bit controversial)

- The exact sum of two FP numbers of precision $p$ can be stored on $\approx 2p$ bits only
- Same for the exact product
- Same for division – even for $1/3 = 0.0101010101(01)^\infty$
  - to compute $x/y$, first compute $(q, r)$ such that $x = yq + r$
  - then use $r$ to decide rounding of $q$
- Same for square root
  - to compute $\sqrt{x}$, first compute $(s, r)$ such that $x = s^2 + r$
  - then use $r$ to decide rounding of $s$

## Most controversial point:

Subnormal handling is indeed complex/expensive, and has long been trapped to software/microcode

Correctly rounded elementary functions were considered not implementable then

# A few theorems (useful or not)

Let $x$ and $y$ be FP numbers.

- Sterbenz Lemma: if $x/2 < y < 2x$ then $x \ominus y = x - y$
- The rounding error when adding $x$ and $y$:
  $r = (x + y) - (x \oplus y)$ is an FP number, and if $x \geq y$ it may be computed as

$$r := y \ominus ((x \oplus y) \ominus x);$$

- The rounding error when multiplying $x$ and $y$:
  $r = xy - (x \otimes y)$ is an FP number and may be computed by a (slightly more complex) sequence of $\otimes$, $\oplus$ and $\ominus$ operations.
- $\sqrt{x \otimes x + y \otimes y} \geq x$
- ...

# Here I should try to prove Sterbenz lemma

Floating-point format in radix $\beta$ with $p$ digits of significand
Suppose $x$ and $y$ are positive.
Notation using integral significands:

$$x = M_x \times \beta^{e_x - p + 1},$$

$$y = M_y \times \beta^{e_y - p + 1},$$

with

$$\begin{cases} e_{\min} \leq e_x \leq e_{\max} \\ e_{\min} \leq e_y \leq e_{\max} \\ 0 \leq M_x \leq \beta^p - 1 \\ 0 \leq M_y \leq \beta^p - 1. \end{cases}$$

Suppose $y \leq x$ therefore $e_y \leq e_x$: define $\delta = e_x - e_y$

$$x - y = \left( M_x \beta^\delta - M_y \right) \times \beta^{e_y - p + 1}.$$

Define $M = M_x \beta^\delta - M_y$

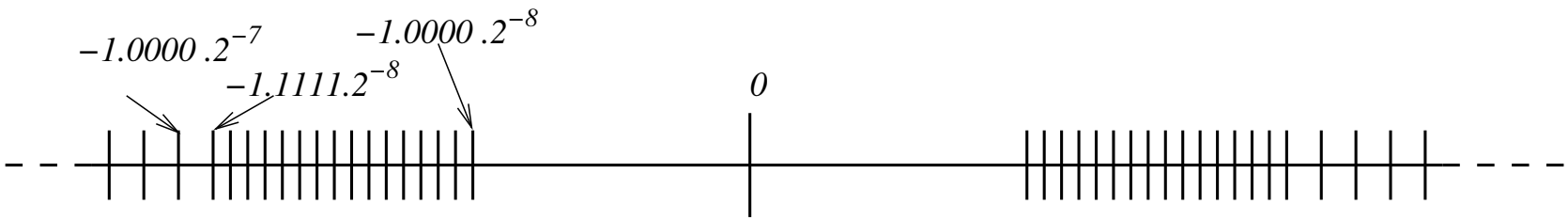
- $x \geq y$ implies $M \geq 0$;
- $x \leq 2y$ implies $x - y \leq y$, hence $M \beta^{e_y - p + 1} \leq M_y \beta^{e_y - p + 1}$; therefore,

$$M \leq M_y \leq \beta^p - 1.$$

So $x - y$ is equal to $M \times \beta^{e - p + 1}$ with $e_{\min} \leq e \leq e_{\max}$ and $|M| \leq \beta^p - 1$. This shows that $x - y$ is a floating-point number, which implies that it is exactly computed.

# Remarks on this proof

- We haven't used the rounding mode ?!?
  - We just proved that the mathematical result is representable
  - Any rounding mode $\circ$ verifies: if $Z$ is representable, then $\circ(Z) = Z$
  - Sterbenz lemma is true for any rounding mode.
- We need subnormals, of course.




(Normal numbers have an integral significand such that $\beta^{p-1} \leq M \leq \beta^p - 1$ and we couldn't prove the left inequality)

- We don't care about the binary encoding (only that there is an $e_{\min}$)

# Representing your constants

Writing a constant in decimal can be safe enough if you are aware of the following.

- Any binary FP number can be written in decimal (given enough digits)
  - first rewrite $m.2^e = (5^{-e}m).10^e$
  - then find some $k$ such that $10^k.m.2^e$ is an integer $n$
  - then $m.2^e = n.10^{e-k}$
- The reciprocal is not true (e.g. 0.1)
- Modern compilers are well behaved:
  - They will consider all the decimal digits you give them
  - They will round the decimal constant you provide to the nearest FP number

# Error-free write-read cycle

**Theorem**

Writing a binary32 (resp. binary64 number) to file on 10 (resp. 20) decimal digits guarantees that the exact same number will be read back.

(Actually the minimal decimal sizes are 9 and 17 digits)

# The conclusion so far

- We have a standard for FP, and it seems well thought out.
- (all we have seen was already in the 1985 version – more on the 2008 revision later)

Let us try to use it.

# Floating-point as it is: processors, OS, languages and compilers

# A frightening introductory example

Let us compile the following C program:

```
1   float ref, index;
2
3   ref = 169.0 / 170.0;
4
5   for (i = 0; i < 250; i++) {
6     index = i;
7     if (ref == (index / (index + 1.0))  )  break;
8   }
9
10  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 ensuring the standard?

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

# Who is in charge of ensuring the standard?

- 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
    - 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 ensuring the standard?

- 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 ensuring the standard?

- 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 ensuring the standard?

- 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.

# 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:
  - 3-7 cycles for addition and multiplication, pipelined (1 op/cycle)
  - 15-50 cycles for division and square root, not pipelined (hard or soft).
  - 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

# A funny horror story

(real story, told by somebody at CERN)

- 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...
- Why?
  - the sort algorithm works under the naive assumption that if $A \not< B$, then $A \geq B$
  - (difficult to write a sort algorithm without this assumption)
  - `x*x+y*y` inlined and compiled differently at two points of the programme,
  - computation on 64 or 80 bits, depending on register allocation
  - enough to break the assumption (horribly rarely).

We will see there was no programming mistake.
And it is very difficult to fix.

# 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 double-precision FP operations in parallel, or
  - 4 identical single-precision 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 2
  - 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 2
  - enable efficient implementation of division and square root
- All the modern FPUs are built around the FMA:
  ARM, Power, IA64, all GPGPUs, and even intel AVX.
- 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)

# Quickly, IA64 (aka Itanium)

They don't sell that many of them, but the best available FP architecture

- Two double-extended FMA (best of IA32 – without the bane– and best of Power)
- instead of one FP status register, 4 of them, selectable on an instruction-basis
  - you can mix round up and round down, double and double-extended
  - on all other architecture, changing the FP status requires flushing the pipeline (10-100 cycles)
- A register format with two more exponent bits (17).

# Evaluation of an expression

Consider the following program, whatever the language

```
float a,b,c,x;
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 and Java have completely different answers.

# Evaluation of an expression

```
float a,b,c,x;
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?
  - Similar issue: should multiply-additions be fused in FMA?

# Evaluation of an expression

```
float a,b,c,x;
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`. Why?
- This line is valid if you replace 5 by 4, but not by 3. Why?

# 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.*

That was the 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 "C99" standard:
*International Standard ISO/IEC 9899:1999(E).*
*Programming languages – C*

- 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

# C in the gory details

*Morceaux choisis* from appendix F.8.2 of the C99 standard:

- Commutativities are OK
- `x/2` may be replaced with `0.5*x`,
  because both operations are always exact in IEEE-754.
- `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 $\infty$ 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
- Strangely, 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` 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
  - Storing a `float` variable in 64 or 80 bits of memory instead of 32 is usually slower, therefore in the C philosophy it should be avoided.
  - 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)
  - 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.
  - (sort bug explained when register file is 80 bits and memory storage is 64 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 sells 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 and perspective

# A historical perspective

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

# 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 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

# Tackling the HPC accuracy challenge

## Floating point operations are not associative

... but optimisations tend to assume they are (or, that the order is not important):

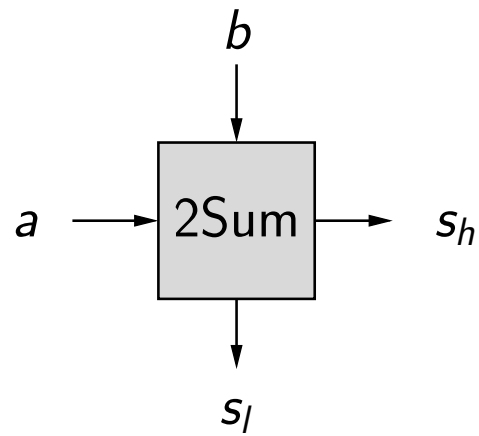
- blocking for optimal cache usage (ATLAS)
- parallelisation
  - The concept of reduction is valid only for associative operations
- ...

Rationale: there is no reason the new computation order should be worse than the sequential one...
Actually there is: the optimizations enable larger problem sizes!

# Example: large sums and sums of products

- Cooking recipes: If you have to add terms of known different magnitude, it may be a good idea to sort them
  - see the Handbook for variations on this theme
- Better: bring associativity back by using error-free transformations

# Basic EFT blocks




- $s_h + s_l = a + b$ exactly, and $s_h = \circ(a + b)$
- Also 2Mul block: $p_h + p_l = a \times b$ exactly, and $p_h = \circ(a \times b)$

# Exact sum of two FP numbers

## Theorem (Fast2Sum algorithm)

*Assuming*

- *floating-point in radix* $\beta \leq 3$, *with subnormal numbers*
- $\circ$ *correct rounding to nearest*
- *a and b floating-point numbers*
- *exponent of* $a \geq$ *exponent of* $b$

*The following algorithm computes two floating-point numbers s and t satisfying:*

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

$$
\begin{aligned}
s &\leftarrow \circ(a+b)\\
z &\leftarrow \circ(s-a)\\
t &\leftarrow \circ(b-z)
\end{aligned}
$$

# If you don't now if $a > b$

- Either sort them
  - used to required a branch, which is Very Bad
  - now we have min and max instructions, much better
- or use the following

## TwoSum

$$
\begin{aligned}
&s \leftarrow \circ(a+b)\\
&a' \leftarrow \circ(s-b), \quad && b' \leftarrow \circ(s-a)\\
&\delta_a \leftarrow \circ(a-a'), \quad && \delta_b \leftarrow \circ(b-b')\\
&t \leftarrow \circ(\delta_a+\delta_b)
\end{aligned}
$$

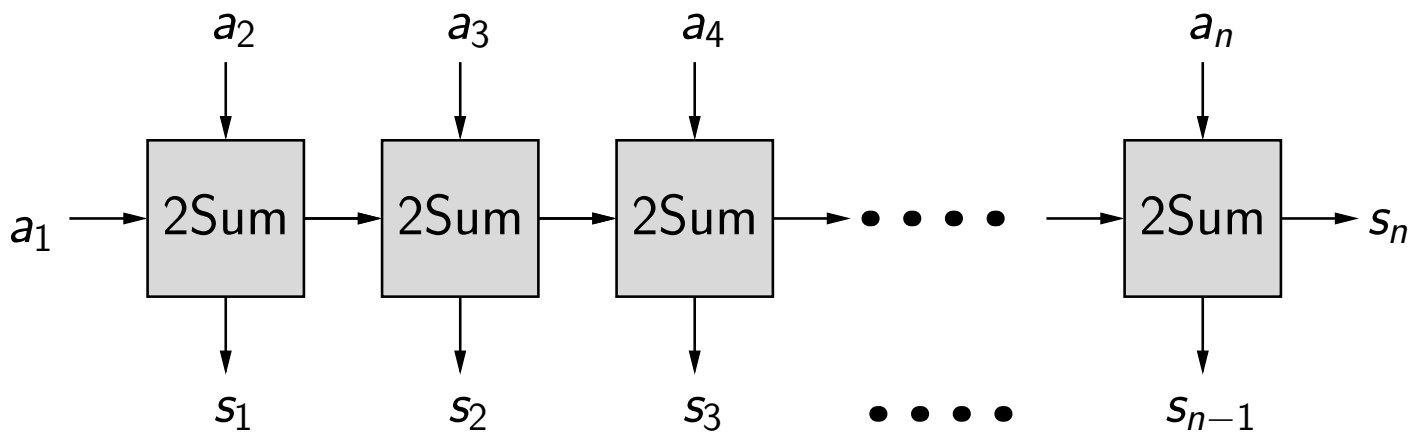
- proven in Coq
  - also works for radix 10
  - even in the presence of underflow
- proven minimal branchless algorithm (by enumeration)

# Exact product of two FP numbers, with an FMA

## TwoMulFMA

$$r_h \leftarrow \circ(a \times b)$$
$$r_l \leftarrow \circ(h - a \times b)$$

# EFT sum




- $\sum_{i=1}^{n} s_i = \sum_{i=1}^{n} a_i$ exactly
- $s_n$ is the iterative floating-point sum.

No information lost: EFT brings associativity back

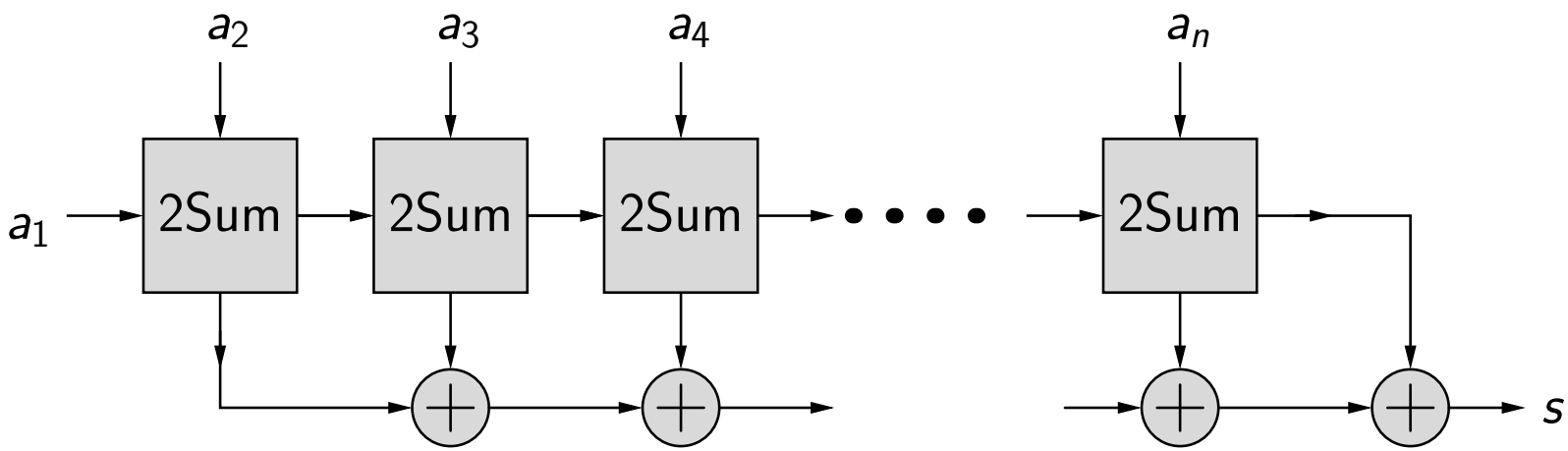
# A better rule of the game

## No information lost: EFT brings associativity back

- Now we can safely play optimization games
- ... with a well-specified rule of the game
- for instance: return correct rounding of the exact sum
- Implementation challenge: compute just right
  (use EFTs only in the degenerate cases that need it)

(about 1 good paper/year on the subject in the last decade)

# Example: Compensated sum




- correct the iterative sum with the sum of the "error terms"
- (the latter being computed naively)

## Theorem (Rump, Ogita, and Oishi)

*If $nu < 1$, then, even in the presence of underflow,*

$$\left| s - \sum_{i=1}^{n} x_i \right| \leq u \left| \sum_{i=1}^{n} x_i \right| + \gamma_{n-1}^2 \sum_{i=1}^{n} |x_i|.$$

# How accurate is a computation?

**error = computed value - reference value**

The reference value should **not** be the one computed by the sequential code.

- It is the value defined by the maths (or the physics)

## Example: the exact sum of $n$ floating-point numbers

(the reference to which sum algorithms should compare)

## In "real" code, the reference is usually very difficult to define

- approximation
- discretisation
- rounding

# Error analysis

- Proving the absence of over/underflow may be relatively easy
  - when you compute energies, not when you compute areas
- Error analysis techniques: how are your equations sensitive to roundoff errors ?
  - Forward error analysis: what errors did you make ?
  - Backward error analysis: which problem did you solve exactly ?
- Notion of conditioning:

$$Cond = \frac{|\text{relative change in output}|}{|\text{relative change in input}|} = \lim_{\widehat{x}\to x} \frac{|(f(\widehat{x}) - f(x))/f(x)|}{|(\widehat{x} - x)/x|}$$

  - $Cond \geq 1$ problem is ill-conditionned / sensitive to rounding
  - $Cond \ll 1$ problem is well-conditionned / resistant to rounding
  - $Cond$ may depend on $x$: again, make cases...

# "Mindless" schemes to improve confidence

- Repeat the computation in arithmetics of increasing precision, until digits of the result agree.
  - Maple, Mathematica, GMP/MPFR
- Repeat the computation with same precision but different (IEEE-754) rounding modes, and compare the results.
  - all you need is change the processor status in the beginning
- Repeat the computation a few times with same precision, rounding each operation randomly, and compare the results.
  - stochastic arithmetic, CESTAC
- Repeat the computation a few times with same precision but slightly different inputs, and compare the results.
  - easy to do yourself

None of these schemes provide any guarantee. They may increase confidence, though.

See "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation ?" on Kahan's web page

# Interval arithmetic

- Instead of computing $f(x)$, compute an interval $[f_l, f_u]$ which is guaranteed to contain $f(x)$
  - operation by operation
  - use directed rounding modes
  - several libraries exist
- This scheme does provide a guarantee
- ... which is often overly pessimistic
  (" Your result is in $[-\infty, +\infty]$, guaranteed")
- Limit interval bloat by being clever (changing your formula)
- ... and/or using bits of arbitrary precision when needed (MPFI library).
- Therefore not a mindless scheme

# The last word

- We have a standard for FP, it is a good one, and eventually your PC will comply
- The standard doesn't guarantee that the result of your program is close at all to the mathematical result it is supposed to compute.
- But at least it enables serious mathematics with floating-point

# So, do you trust your computer now ?

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

A. Householder

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

Feel nervous, but feel in control.
It's not dark magic, it's science.

# Backup slides

# The legacy FPU of IA32 instruction set

Implemented in processors by Intel, AMD, Via/Cyrix, Transmeta... since the Intel 8087 coprocessor in 1985

- internal double-extended format on 80 bits: significand on 64 bits, exponent on 15 bits.
- (almost) perfect IEEE compliance on this double-extended format
- one status register which holds (among other things)
  - the current rounding mode
  - the precision to which operations round the significand: 24, 53 or 64 bits.
  - but the exponent is always 15 bits
- For single and double, IEEE-754-compliant rounding and overflow handling (including exponent) performed when writing back to memory

There probably is a rationale for all this, but... ask Intel people.

# What it means

Assume you want a portable programme, *i.e* use double-precision.

- Fully IEEE-754 compliant possible, but slow:
  - set the status flags to "round significand to 53 bits"
  - then write the result of every single operation to memory
  - (not every single but almost)
- Next best: compliant except for over/underflow handling:
  - set the status flags to "round significand to 53 bits"
  - but computations will use 15-bit exponents instead of 12
  - OK if if you may prove that your program doesn't generate huge nor tiny values
- If you compute in registers: register allocation decides if you're computing on 53 or 64 bits
  - random, unpredictible, unreproducible
  - the bane of floating-point between 1985 and 2005

# Avoiding cancellations in practice

## Computing the area of a triangle

Heron of Alexandria:
$A := \sqrt{(s(s-x)(s-y)(s-z))}$ with $s = (x+y+z)/2$
Kahan's algorithm:
Sort $x$, $y$, $z$ so that $x \geq y \geq z$ ;
If $z < x - y$ then no such triangle exists ;
else $A :=$
$$\sqrt{((x+(y+z)) \times (z-(x-y)) \times (z+(x-y)) \times (x+(y-z)))/4}$$

Exercise: solving the quadratic equation by $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$

# Trust your math

Classical example: Muller's recurrence

$$\begin{cases} x_0 & = & 4 \\ x_1 & = & 4.25 \\ x_{n+1} & = & 108 - (815 - 1500/x_{n-1})/x_n \end{cases}$$

- Any half-competent mathematician will find that it converges to 5
- On any calculator or computer system using non-exact arithmetic, it will converge very convincingly to 100

$$x_n = \frac{\alpha 3^{n+1} + \beta 5^{n+1} + \gamma 100^{n+1}}{\alpha 3^n + \beta 5^n + \gamma 100^n}$$