De calculer juste
à calculer au plus juste

Introduction à l'école PRCN

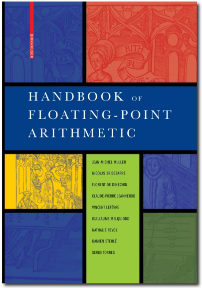
Florent de Dinechin

AriC project

My research group

The AriC project @ École Normale Supérieure de Lyon:
Arithmetic and Computing at large

- Hardware and software
- From addition to linear algebra
- Fixed point, floating-point, multiple-precision, finite fields, ...
- Pervasive concern of performance, numerical quality and validation
- Interactions with computing at large



Outline

Floating-point in your machine

Accuracy versus reproductibility

Performance versus accuracy

Conclusion: It's the Hardware, Stupid

Space-filling advertising: hardware computing just right

Floating-point in your machine

We have a nice floating-point standard

It is called IEEE-754, and you will hear a lot about it.
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 mathematical result.

We have a nice floating-point standard

It is called IEEE-754, and you will hear a lot about it.
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 mathematical result.

(In case of tie, round to the number with an even significand $\Longrightarrow$ no bias.)

We have a nice floating-point standard

It is called IEEE-754, and you will hear a lot about it.
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 mathematical result.

(In case of tie, round to the number with an even significand $\Longrightarrow$ no bias.)

No compromise: this is the best that the format allows

We have a nice floating-point standard

It is called IEEE-754, and you will hear a lot about it.
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 mathematical result.

(In case of tie, round to the number with an even significand $\Longrightarrow$ no bias.)

No compromise: this is the best that the format allows

Nice properties :

- If $a + b$ is a FP number, then $a \oplus b$ returns it
- Rounding is monotonic
- Rounding does not introduce any statistical bias

However and nevertheless,

Let us compile the following C 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);
```

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!

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.*

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.

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
    - 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

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

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:

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.

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.

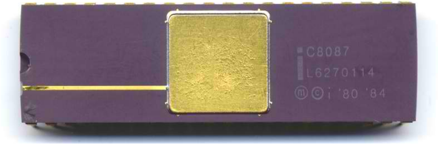
The common denominator of modern processors

- 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

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

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? Because 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,
  - 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 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

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

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.

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` (...) $\sqrt{b^2 - 4ac}$

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)

Reproductibility begins with predictability

When you write

```
sqrt(b*b-4*a*c)
```

do you know how it is going to be compiled?

In general: 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?

In general: 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 $((a+b)+c)+d$

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.

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 $2^{100} + 1 - 2^{100}$)

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?

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`

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

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*

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.

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.

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?

The Patriot bug

In 1991, a Patriot anti-missile failed to intercept a Scud missile.
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

Test: which of the following increments should you use?

10 5 3 1 0.5 0.25 0.2 0.125 0.1

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

C in the gory details

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

- Commutativities are OK

C in the gory details

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

- Commutativities are OK
- `x/2` may be replaced with `0.5*x`,
  because both operations are always exact in IEEE-754.

C in the gory details

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

- Commutativities are OK
- `x/2` 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)

C in the gory details

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

- Commutativities are OK
- `x/2` 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`

C in the gory details

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

- Commutativities are OK
- `x/2` 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

C in the gory details

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

- Commutativities are OK
- `x/2` 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)

C in the gory details

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

- Commutativities are OK
- `x/2` 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.

C in the gory details

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

- Commutativities are OK
- `x/2` 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...

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$\times$ faster)
  - Or, on a newer PC, the same computation became much less accurate!

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$\times$ 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)

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$\times$ 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)

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$\times$ 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.

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$\times$ 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

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$\times$ 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...

Quickly, Java

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

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.

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 of this part: A historical perspective

- Before 1985, floating-point was an ugly mess

Conclusion of this part: 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

Conclusion of this part: 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

Conclusion of this part: 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

Conclusion of this part: 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

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

Accuracy versus reproductibility

Floating-point in your machine

Accuracy versus reproductibility

Performance versus accuracy

Conclusion: It's the Hardware, Stupid

Space-filling advertising: hardware computing just right

Accuracy is important

Is reproducibility important?

- Let us review a few use cases where people wanted numerical reproducibility.
- For each of these use cases, consider these two questions:

The question people ask

What is the cost of reproducibility?

Is reproducibility important?

- Let us review a few use cases where people wanted numerical reproducibility.
- For each of these use cases, consider these two questions:

The question people ask

What is the cost of reproducibility?

The question they should ask

Will the focus on reproducibility lead to **good**, or to **evil**?

A toy use case

- Blender is a 3D authoring tool
- It includes `blenderplayer`: render Blender animations/games in real time
- Competition of animations using this tool
- I am going to show one of the winning entries

The blenderplayer case

What is the cost of reproducibility?

I don't know, I didn't try. More on this on next slide.

The blenderplayer case

What is the cost of reproducibility?

I don't know, I didn't try. More on this on next slide.

Is the focus on reproducibility good or evil?

- Would you design the launch system of a satellite this way?
- What about a game that is playable on XStation but unplayable on PlayBox?

The blenderplayer case

What is the cost of reproducibility?

I don't know, I didn't try. More on this on next slide.

Is the focus on reproducibility good or evil?

- Would you design the launch system of a satellite this way?
- What about a game that is playable on XStation but unplayable on PlayBox?

Conclusion: in such a case,

- A programmer that would insist on reproducibility would be an idiot
- What we need here is tools that make computing *even less reproducible*:

  let me advertise stochastic arithmetic.

By the way, what do we call reproducibility?

- Some kind of predictability, because we have read the standards?

By the way, what do we call reproducibility?

- Some kind of predictability, because we have read the standards?
- Two runs on the same computer with the same OS? But
  - two occurences of the same code may be compiled differently
  - the execution may be serialized differently in a multithreaded environment

By the way, what do we call reproducibility?

- Some kind of predictability, because we have read the standards?
- Two runs on the same computer with the same OS? But
  - two occurences of the same code may be compiled differently
  - the execution may be serialized differently in a multithreaded environment
- Two runs on different computers with the same OS? But
  - different processors have different arithmetic units

By the way, what do we call reproducibility?

- Some kind of predictability, because we have read the standards?
- Two runs on the same computer with the same OS? But
  - two occurences of the same code may be compiled differently
  - the execution may be serialized differently in a multithreaded environment
- Two runs on different computers with the same OS? But
  - different processors have different arithmetic units
- Two runs on the same computer with different OS's? But
  - different mathematical libraries, policies WRT exceptions, default behaviours...

The serious version of the Blender use case

Algorithmic geometry problems:

- Example: compute the determinant of two vectors to decide their relative orientation

The serious version of the Blender use case

Algorithmic geometry problems:

- Example: compute the determinant of two vectors to decide their relative orientation

Here we have a mathematical reference

We know what the code is supposed to compute.
Solution: write a test that detects if rounding may lead to wrong result, and recompute with higher accuracy in this (hopefully rare) case.

The serious version of the Blender use case

Algorithmic geometry problems:

- Example: compute the determinant of two vectors to decide their relative orientation

Here we have a mathematical reference

We know what the code is supposed to compute.
Solution: write a test that detects if rounding may lead to wrong result, and recompute with higher accuracy in this (hopefully rare) case.

What is the cost of reproducibility?

Minor in execution time, high in coffee consumption.
Let me advertise Gappa, a tool that will reduce coffee consumption.

The serious version of the Blender use case

Algorithmic geometry problems:

- Example: compute the determinant of two vectors to decide their relative orientation

Here we have a mathematical reference

We know what the code is supposed to compute.
Solution: write a test that detects if rounding may lead to wrong result, and recompute with higher accuracy in this (hopefully rare) case.

What is the cost of reproducibility?

Minor in execution time, high in coffee consumption.
Let me advertise Gappa, a tool that will reduce coffee consumption.

Is the focus on reproducibility good or evil?

In CAD tools, I guess it is good.
In games, performance (WCET) is more important.

If she moves fast enough you won't notice the bugs

Use case: CERN's LHC@home

Objective:
simulate various configurations of the superconducting magnets (before building them)

- The simulated phenomenon is known chaotic

Use case: CERN's LHC@home

Objective:
simulate various configurations of the superconducting magnets (before building them)

- The simulated phenomenon is known chaotic
- Computation distributed on a large number of untrusted PCs.

Use case: CERN's LHC@home

Objective:
simulate various configurations of the superconducting magnets (before building them)

- The simulated phenomenon is known chaotic
- Computation distributed on a large number of untrusted PCs.
- Confidence by redundancy:
  if two PCs return the exact same result, it is trusted.
  - that is, the computation on each PC is trusted,
  - not its physical significance: the computation is chaotic.

Maybe I am biased on this one.

Here we don't have a mathematical reference

… not even a physical one: we are trying to frame it.

Maybe I am biased on this one.

Here we don't have a mathematical reference

... not even a physical one: we are trying to frame it.

What is the cost of reproducibility?

- Performance *benefit* (more PCs can be exploited)
- Coffee consumption: several engineer-months.
- Recipe:
  - chose a portable compiler,
  - add parentheses to Fortran
  - replace elementary functions with correctly-rounded ones

Maybe I am biased on this one.

Here we don't have a mathematical reference

... not even a physical one: we are trying to frame it.

What is the cost of reproducibility?

- Performance *benefit* (more PCs can be exploited)
- Coffee consumption: several engineer-months.
- Recipe:
  - chose a portable compiler,
  - add parentheses to Fortran
  - replace elementary functions with correctly-rounded ones

Is the focus on reproducibility good or evil?

- Mostly good
- ... but cost/benefit disputable

Use case: the Intel Math Kernel Libraries

The MKL include elementary functions, BLAS, etc.

- Since the transition to multicore, Intel gets bug reports:
  the BLAS are no longer deterministic!
- Solution: a compiler switch that basically imposes a deterministic serialization

Use case: the Intel Math Kernel Libraries

The MKL include elementary functions, BLAS, etc.

- Since the transition to multicore, Intel gets bug reports: the BLAS are no longer deterministic!
- Solution: a compiler switch that basically imposes a deterministic serialization

What is the cost of reproducibility?

Catastrophic in execution time (but may improve in the future)

Use case: the Intel Math Kernel Libraries

The MKL include elementary functions, BLAS, etc.

- Since the transition to multicore, Intel gets bug reports: the BLAS are no longer deterministic!
- Solution: a compiler switch that basically imposes a deterministic serialization

What is the cost of reproducibility?

Catastrophic in execution time (but may improve in the future)

Is the focus on reproducibility good or evil?

- good for debugging
- otherwise bad: *serialization chosen for reproducibility, not for accuracy*
- When different runs give different results, programmers question them.

Use case: the Intel Math Kernel Libraries

The MKL include elementary functions, BLAS, etc.

- Since the transition to multicore, Intel gets bug reports: the BLAS are no longer deterministic!
- Solution: a compiler switch that basically imposes a deterministic serialization

What is the cost of reproducibility?

Catastrophic in execution time (but may improve in the future)

Is the focus on reproducibility good or evil?

- good for debugging
- otherwise bad: *serialization chosen for reproducibility, not for accuracy*
- When different runs give different results, programmers question them.

Here also we have a mathematical reference! Why not use it?

Conclusion on this part

We shouldn't care about reproducibility. What matters is accuracy.

- Perfectly accurate results are reproducible (correct rounding)
  - Reproducibility by specification is good
  - Reproducibility of poorly understood code is dangerous.

Conclusion on this part

We shouldn't care about reproducibility. What matters is accuracy.

- Perfectly accurate results are reproducible (correct rounding)
  - Reproducibility by specification is good
  - Reproducibility of poorly understood code is dangerous.

- I'd rather have
  3 guaranteed digits everywhere
  than
  the exact same (totally wrong) result everywhere

Conclusion on this part

We shouldn't care about reproducibility. What matters is accuracy.

- Perfectly accurate results are reproducible (correct rounding)
  - Reproducibility by specification is good
  - Reproducibility of poorly understood code is dangerous.
- I'd rather have
  3 guaranteed digits everywhere
  than
  the exact same (totally wrong) result everywhere

In other words: **reproducible accuracy**

What is an error? What is accuracy?

The most important sentence of this talk

An error is a difference (absolute or relative) between two values, one being a reference for the other.

Examples:

- error of the FP addition is with reference of the real sum (easy)
- error of the polynomial is with reference to the function (easy)

What is an error? What is accuracy?

The most important sentence of this talk

An error is a difference (absolute or relative) between two values, one being a reference for the other.

Examples:

- error of the FP addition is with reference of the real sum (easy)
- error of the polynomial is with reference to the function (easy)
- error of one FP addition within the polynomial evaluation? (difficult because we have no direct reference in the function)

What is an error? What is accuracy?

The most important sentence of this talk

An error is a difference (absolute or relative) between two values, one being a reference for the other.

Examples:

- error of the FP addition is with reference of the real sum (easy)
- error of the polynomial is with reference to the function (easy)
- error of one FP addition within the polynomial evaluation? (difficult because we have no direct reference in the function)
- accuracy of the BLAS?

What is an error? What is accuracy?

The most important sentence of this talk

An error is a difference (absolute or relative) between two values, one being a reference for the other.

Examples:

- error of the FP addition is with reference of the real sum (easy)
- error of the polynomial is with reference to the function (easy)
- error of one FP addition within the polynomial evaluation? (difficult because we have no direct reference in the function)
- accuracy of the BLAS?

Never say "the error of this term is ...":

it doesn't mean anything without the reference.

If you are not able to define the reference value,

you will not be able to know how accurate you compute

Performance versus accuracy

Bottom line of this part

Common wisdom

The more accurate you compute, the more expensive it gets

Bottom line of this part

Common wisdom

The more accurate you compute, the more expensive it gets

In practice

- We (hopefully) notice it when our computation is not accurate enough.
- But do we notice it when it is too accurate for our needs?

Bottom line of this part

Common wisdom

The more accurate you compute, the more expensive it gets

In practice

- We (hopefully) notice it when our computation is not accurate enough.
- But do we notice it when it is too accurate for our needs?

Reconciling performance and accuracy?

Or, regain performance by computing just right?

Double precision spoils us

The standard binary64 format (formerly known as double-precision) provides roughly 16 decimal digits.

Why should anybody need such 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

Parenthesis: then why binary64?

Parenthesis: then why binary64?

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

Parenthesis: then why binary64?

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

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)

Parenthesis: then why binary64?

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

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.

Parenthesis: then why binary64?

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

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?

One example of performance by computing just right

Correctly rounded elementary functions

- IEEE-754 floating-point single or double-precision
- Elementary functions: sin, cos, exp, log, implemented in the "standard mathematical library" (`libm`)

One example of performance by computing just right

Correctly rounded elementary functions

- IEEE-754 floating-point single or double-precision
- Elementary functions: sin, cos, exp, log, implemented in the "standard mathematical library" (`libm`)
- Correctly rounded: As perfect as can be, considering the finite nature of floating-point arithmetic
  - same standard of quality as $+, \times, /, \sqrt{\ }$

One example of performance by computing just right

Correctly rounded elementary functions

- IEEE-754 floating-point single or double-precision
- Elementary functions: sin, cos, exp, log, implemented in the "standard mathematical library" (`libm`)
- Correctly rounded: As perfect as can be, considering the finite nature of floating-point arithmetic
  - same standard of quality as $+, \times, /, \sqrt{\ }$
- Now recommended by the IEEE754-2008 standard, but long considered too expensive

  because of the Table Maker's Dilemma

The Table Maker's Dilemma

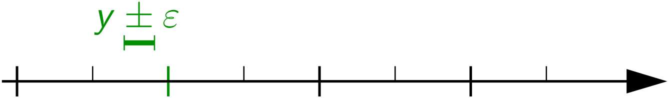
- Finite-precision algorithm for evaluating $f(x)$

The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation $+$ rounding errors $\longrightarrow$ overall error bound $\overline{\varepsilon}$.

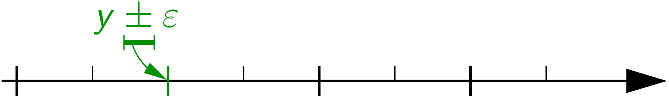
The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound $\overline{\varepsilon}$.
- What we compute: y such that $f(x) \in [y - \overline{\varepsilon}, y + \overline{\varepsilon}]$



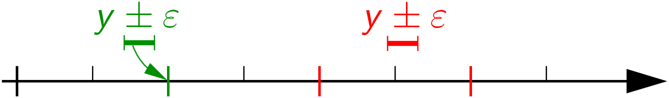
The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound $\overline{\varepsilon}$.
- What we compute: y such that $f(x) \in [y - \overline{\varepsilon}, y + \overline{\varepsilon}]$



The Table Maker's Dilemma

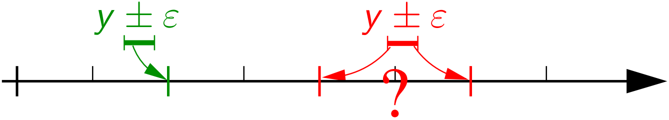
- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound $\overline{\varepsilon}$.
- What we compute: y such that $f(x) \in [y - \overline{\varepsilon}, y + \overline{\varepsilon}]$



Dilemma if this interval contains a midpoint between two FP numbers

The Table Maker's Dilemma

- Finite-precision algorithm for evaluating $f(x)$
- Approximation + rounding errors $\longrightarrow$ overall error bound $\overline{\varepsilon}$.
- What we compute: y such that $f(x) \in [y - \overline{\varepsilon}, y + \overline{\varepsilon}]$



Dilemma if this interval contains a midpoint between two FP numbers

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		100000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00203,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- I want 12 significant digits

The first digital signature algorithm

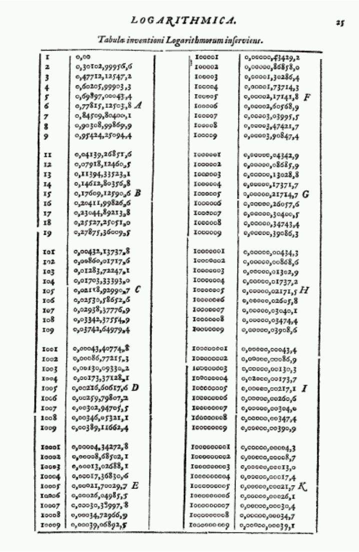
LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- I want 12 significant digits
- I have an approximation scheme that provides 14 digits

The first digital signature algorithm



LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- I want 12 significant digits
- I have an approximation scheme that provides 14 digits
- or,

$$y = \log(x) \pm 10^{-14}$$

The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- I want 12 significant digits
- I have an approximation scheme that provides 14 digits
- or,

$$y = \log(x) \pm 10^{-14}$$

- "Usually" that's enough to round

$$y = x,xxxxxxxxxxx17 \pm 10^{-14}$$

$$y = x,xxxxxxxxxxx83 \pm 10^{-14}$$

The first digital signature algorithm

LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,38997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- I want 12 significant digits
- I have an approximation scheme that provides 14 digits
- or,

$$y = \log(x) \pm 10^{-14}$$

- "Usually" that's enough to round

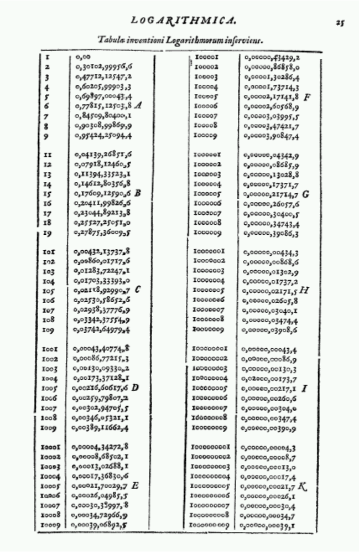
$$y = x, xxxxxxxxxx17 \pm 10^{-14}$$

$$y = x, xxxxxxxxxx83 \pm 10^{-14}$$

- Dilemma when

$$y = x, xxxxxxxxxx50 \pm 10^{-14}$$

The first digital signature algorithm



LOGARITHMICA. 25

Tabulæ inventioni Logarithmorum inserviens.

1	0,00		100001	0,00000,43429,2	
2	0,30102,99956,6		100002	0,00000,86858,0	
3	0,47712,12547,2		100003	0,00001,30286,4	
4	0,60205,99903,3		100004	0,00001,73714,3	
5	0,69897,00043,4		100005	0,00002,17141,8	F
6	0,77815,12503,8	A	100006	0,00002,60568,9	
7	0,84509,80400,1		100007	0,00003,03995,5	
8	0,90308,99869,9		100008	0,00003,47421,7	
9	0,95424,25094,4		100009	0,00003,90847,4	
11	0,04139,26851,6		1000001	0,00000,04342,9	
12	0,07918,12460,5		1000002	0,00000,08685,9	
13	0,11394,33523,1		1000003	0,00000,13028,8	
14	0,14612,80356,8		1000004	0,00000,17371,7	
15	0,17609,12590,6	B	1000005	0,00000,21714,7	G
16	0,20411,99826,6		1000006	0,00000,26057,6	
17	0,23044,89213,8		1000007	0,00000,30400,5	
18	0,25527,25051,0		1000008	0,00000,34743,4	
19	0,27875,36009,5		1000009	0,00000,39086,3	
101	0,00432,13737,8		10000001	0,00000,00434,3	
102	0,00860,01717,6		10000002	0,00000,00868,6	
103	0,01283,72247,1		10000003	0,00000,01302,9	
104	0,01703,33393,0		10000004	0,00000,01737,2	
105	0,02118,92990,7	C	10000005	0,00000,02171,5	H
106	0,02530,58652,6		10000006	0,00000,02605,8	
107	0,02938,37776,9		10000007	0,00000,03040,1	
108	0,03342,37554,9		10000008	0,00000,03474,4	
109	0,03742,64979,4		10000009	0,00000,03908,6	
1001	0,00043,40774,8		100000001	0,00000,00043,4	
1002	0,00086,77215,3		100000002	0,00000,00086,9	
1003	0,00130,09330,2		100000003	0,00000,00130,3	
1004	0,00173,37128,1		100000004	0,02000,00173,7	
1005	0,00216,60617,6	D	100000005	0,00000,00217,1	I
1006	0,00259,79807,2		100000006	0,00000,00260,6	
1007	0,00302,94705,5		100000007	0,00000,00304,0	
1008	0,00346,05321,1		160000008	0,00000,00347,4	
1009	0,00389,11662,4		100000009	0,00000,00390,9	
10001	0,00004,34272,8		1000000001	0,00000,00004,3	
10002	0,00008,68502,1		1000000002	0,00000,00008,7	
10003	0,00013,02688,1		1000000003	0,00000,00013,0	
10004	0,00017,36830,6		1000000004	0,00000,00017,4	
10005	0,00021,70029,7	E	1000000005	0,00000,00021,7	K
10006	0,00026,04985,5		1000000006	0,00000,00026,1	
10007	0,00030,39997,8		1000000007	0,00000,00030,4	
10008	0,00034,72966,9		1000000008	0,00000,00034,7	
10009	0,00039,06892,5		1000000009	0,00000,00039,1	

- I want 12 significant digits
- I have an approximation scheme that provides 14 digits
- or,

$$y = \log(x) \pm 10^{-14}$$

- "Usually" that's enough to round

$$y = x, xxxxxxxxxxx17 \pm 10^{-14}$$

$$y = x, xxxxxxxxxxx83 \pm 10^{-14}$$

- Dilemma when

$$y = x, xxxxxxxxxxx50 \pm 10^{-14}$$

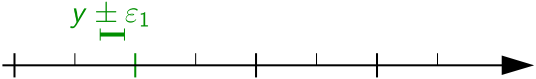
The first table-makers rounded these cases randomly, and recorded them to confound copiers.

Solving the table maker's dilemma

Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$

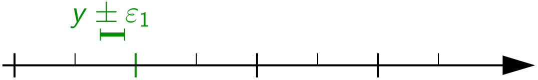
Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$

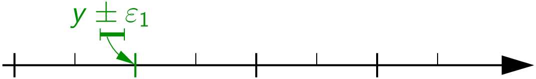
Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$
3. Does $y \pm \varepsilon$ contain the middle point between two FP numbers?

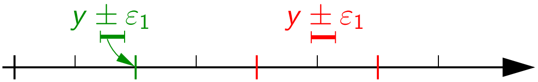
Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$
3. Does $y \pm \varepsilon$ contain the middle point between two FP numbers?
    - If no, return RN(y)

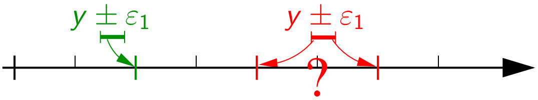
Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$
3. Does $y \pm \varepsilon$ contain the middle point between two FP numbers?
   - If no, return RN(y)
   - If yes,

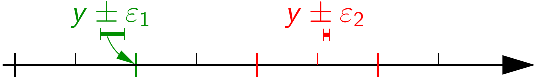
Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$
3. Does $y \pm \varepsilon$ contain the middle point between two FP numbers?
   - If no, return RN(y)
   - If yes,dilemma!

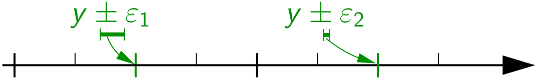
Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$
3. Does $y \pm \varepsilon$ contain the middle point between two FP numbers?
   - If no, return RN(y)
   - If yes,dilemma! Reduce ε, and go back to 2

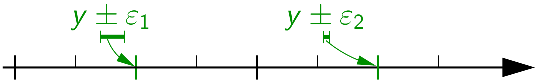
Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$
3. Does $y \pm \varepsilon$ contain the middle point between two FP numbers?
   - If no, return RN(y)
   - If yes,dilemma! Reduce ε, and go back to 2

Solving the table maker's dilemma



Ziv's onion peeling algorithm

1. Initialisation: $\varepsilon = \varepsilon_1$
2. Compute y such that $f(x) = y \pm \varepsilon$
3. Does $y \pm \varepsilon$ contain the middle point between two FP numbers?
   - If no, return RN(y)
   - If yes,dilemma! Reduce ε, and go back to 2

It is a *while* loop...

- Lefèvre and Muller: compute just right the precision at which it terminates.

Accuracy versus performance

When we know that the loop terminates...

CRLibm: 2-step approximation process

- first step fast but accurate to $\overline{\varepsilon}_1$

  sometimes not accurate enough
- (rarely) second step slower but always accurate enough

Accuracy versus performance

When we know that the loop terminates...

CRLibm: 2-step approximation process

- first step fast but accurate to $\overline{\varepsilon}_1$

  sometimes not accurate enough
- (rarely) second step slower but always accurate enough

$$T_{\mathrm{avg}} = T_1 + p_2 T_2$$

Accuracy versus performance

When we know that the loop terminates...

CRLibm: 2-step approximation process

- first step fast but accurate to $\overline{\varepsilon}_1$

  sometimes not accurate enough
- (rarely) second step slower but always accurate enough

$$T_{\text{avg}} = T_1 + p_2 T_2$$

For each step, we want to prove a tight bound $\overline{\varepsilon}$ such that

$$\left|\frac{F(x) - f(x)}{f(x)}\right| \leq \overline{\varepsilon}$$

Accuracy versus performance

When we know that the loop terminates...

CRLibm: 2-step approximation process

- first step fast but accurate to $\overline{\varepsilon}_1$

  sometimes not accurate enough
- (rarely) second step slower but always accurate enough

$$T_{\mathrm{avg}} = T_1 + p_2 T_2$$

For each step, we want to prove a tight bound $\overline{\varepsilon}$ such that

$$\left|\frac{F(x) - f(x)}{f(x)}\right| \leq \overline{\varepsilon}$$

- Overestimating $\overline{\varepsilon}_2$ degrades T_2! (common wisdom)

Accuracy versus performance

When we know that the loop terminates...

CRLibm: 2-step approximation process

- first step fast but accurate to $\overline{\varepsilon}_1$

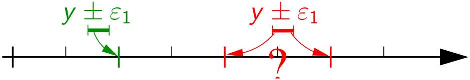
  sometimes not accurate enough
- (rarely) second step slower but always accurate enough

$$T_{\text{avg}} = T_1 + p_2 T_2$$

For each step, we want to prove a tight bound $\overline{\varepsilon}$ such that

$$\left|\frac{F(x) - f(x)}{f(x)}\right| \leq \overline{\varepsilon}$$

- Overestimating $\overline{\varepsilon}_2$ degrades T_2! (common wisdom)
- Overestimating $\overline{\varepsilon}_1$ degrades p_2 !



First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof

First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof
- duration: a Ph.D. thesis (2002)

First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof
- duration: a Ph.D. thesis (2002)

Conclusion was:

- performance and memory consumption of CR elem function is OK

First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof
- duration: a Ph.D. thesis (2002)

Conclusion was:

- performance and memory consumption of CR elem function is OK
- problem now is: performance and coffee consumption of the programmer

Latest function developments in Arénaire

C. Lauter at the end of his PhD,

- development time for sinpi, cospi, tanpi:

Latest function developments in Arénaire

C. Lauter at the end of his PhD,

- development time for sinpi, cospi, tanpi: 2 days
- worst-case time $T_2 \approx 1{,}000$ cycles

Latest function developments in Arénaire

C. Lauter at the end of his PhD,

- development time for sinpi, cospi, tanpi: 2 days
- worst-case time $T_2 \approx 1{,}000$ cycles

(but as a result of three more PhDs)

Summary of the progress made

$$T_{\text{avg}} = T_1 + p_2 T_2$$

- Reduction of T_1 by learning from Intel
- Reduction of p_2 by automating the computation of tight $\overline{\varepsilon}_1$
  (p_2 is proportional to $\overline{\varepsilon}_1$)
- Reduction of T_2 by computing just right
- Reduction of coffee consumption by automating the whole thing

Summary of the progress made

$$T_{\text{avg}} = T_1 + p_2 T_2$$

- Reduction of T_1 by learning from Intel
- Reduction of p_2 by automating the computation of tight $\overline{\varepsilon}_1$
  (p_2 is proportional to $\overline{\varepsilon}_1$)
- Reduction of T_2 by computing just right
- Reduction of coffee consumption by automating the whole thing

The MetaLibm vision

Automate libm expertise so that a new, correct libm can be written for a new processor/context in minutes instead of months.

Conclusion:
It's the Hardware, Stupid

Let us end this talk with the introduction of another one

Doug Burger (Microsoft research) keynote at HiPEAC 2013.

Let us end this talk with the introduction of another one

Doug Burger (Microsoft research) keynote at HiPEAC 2013.

- until 2004: each technology generation gives smaller transistors that are faster and consume less

Let us end this talk with the introduction of another one

Doug Burger (Microsoft research) keynote at HiPEAC 2013.

- until 2004: each technology generation gives smaller transistors that are faster and consume less
- between 2004 and now: we still get smaller transistor, but we cannot clock them faster (power wall)

Let us end this talk with the introduction of another one

Doug Burger (Microsoft research) keynote at HiPEAC 2013.

- until 2004: each technology generation gives smaller transistors that are faster and consume less
- between 2004 and now: we still get smaller transistor, but we cannot clock them faster (power wall)
- tomorrow, transistors still get smaller, but we can't even use them all together (dark silicon)

Let us end this talk with the introduction of another one

Doug Burger (Microsoft research) keynote at HiPEAC 2013.

- until 2004: each technology generation gives smaller transistors that are faster and consume less
- between 2004 and now: we still get smaller transistor, but we cannot clock them faster (power wall)
- tomorrow, transistors still get smaller, but we can't even use them all together (dark silicon)
- *Nothing in our careers has been as fundamental as this transition*

Let us end this talk with the introduction of another one

Doug Burger (Microsoft research) keynote at HiPEAC 2013.

- until 2004: each technology generation gives smaller transistors that are faster and consume less
- between 2004 and now: we still get smaller transistor, but we cannot clock them faster (power wall)
- tomorrow, transistors still get smaller, but we can't even use them all together (dark silicon)
- *Nothing in our careers has been as fundamental as this transition*

The way out according to Doug Burger

We could still "get more" by *specializing* the hardware.

Meanwhile, at Intel

Jeff Arnold (Intel) says:

Meanwhile, at Intel

Jeff Arnold (Intel) says:

Single precision gives you 7 decimal digits. Do you really need this accuracy to compute Angry birds trajectories entered with your fat fingers?

Meanwhile, at Intel

Jeff Arnold (Intel) says:

Single precision gives you 7 decimal digits. Do you really need this accuracy to compute Angry birds trajectories entered with your fat fingers?

... and shows the following slide from his colleagues at ISSCC 2012.

The ISSCC 2012 paper

- notion of “uncertainty”, a power of two attached to inputs and outputs
- technically, computing a center-radius interval
- if uncertainty allows, compute center on 6 or 12 bits only.
- this saves a lot of power.

The ISSCC 2012 paper

- notion of “uncertainty”, a power of two attached to inputs and outputs
- technically, computing a center-radius interval
- if uncertainty allows, compute center on 6 or 12 bits only.
- this saves a lot of power.

Absolutely no use case here... Is this chip usable for real?

The ISSCC 2012 paper

- notion of "uncertainty", a power of two attached to inputs and outputs
- technically, computing a center-radius interval
- if uncertainty allows, compute center on 6 or 12 bits only.
- this saves a lot of power.

Absolutely no use case here... Is this chip usable for real?

What software environment will it need?

From computing right to computing just right

You (probably) came here to learn how to compute right.

From computing right to computing just right

You (probably) came here to learn how to compute right.

This is half the work to compute just right.

From computing right to computing just right

You (probably) came here to learn how to compute right.

This is half the work to compute just right.

What you will learn here might help you address the hardware industry's grand challenge.

Space-filling advertising: hardware computing just right

Computing just right

To sum up,

- Doug Burger says "we should specialize our hardware"
- Kaul et al say "we should design hardware that computes just right"

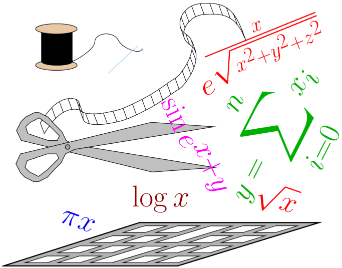
Computing just right

To sum up,

- Doug Burger says "we should specialize our hardware"
- Kaul et al say "we should design hardware that computes just right"

We've been doing both since 2003.

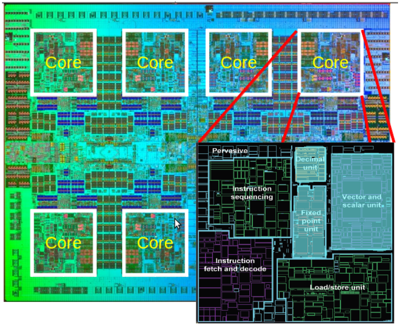
The FloPoCo project

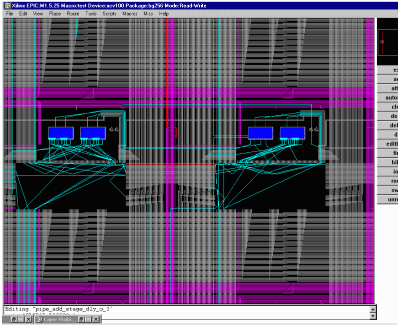


http://flopoco.gforge.inria.fr/

Two different ways of wasting silicon

Here are two universally programmable chips.





Who's best for (insert your computation here) ?

Are FPGAs any good at floating-point?

Long ago (1995), people ported the basic operations: $+, -, \times$

- Versus the highly optimized FPU in the processor,
- each operator **10x slower** in an FPGA

This is the inavoidable overhead of programmability.

Are FPGAs any good at floating-point?

Long ago (1995), people ported the basic operations: $+, -, \times$

- Versus the highly optimized FPU in the processor,
- each operator **10x slower** in an FPGA

This is the inavoidable overhead of programmability.

If you lose according to a metric, change the metric.

Peak figures for double-precision floating-point exponential

- Pentium core: 20 cycles / DPExp @ 4GHz: **200 MDPExp/s**
- FPExp in FPGA: 1 DPExp/cycle @ 400MHz: **400 MDPExp/s**
- Chip vs chip: 6 Pentium cores vs 150 FPExp/FPGA
- Power consumption also better
- Single precision data better

(Intel MKL vector libm, vs FPExp in FloPoCo version 2.0.0)

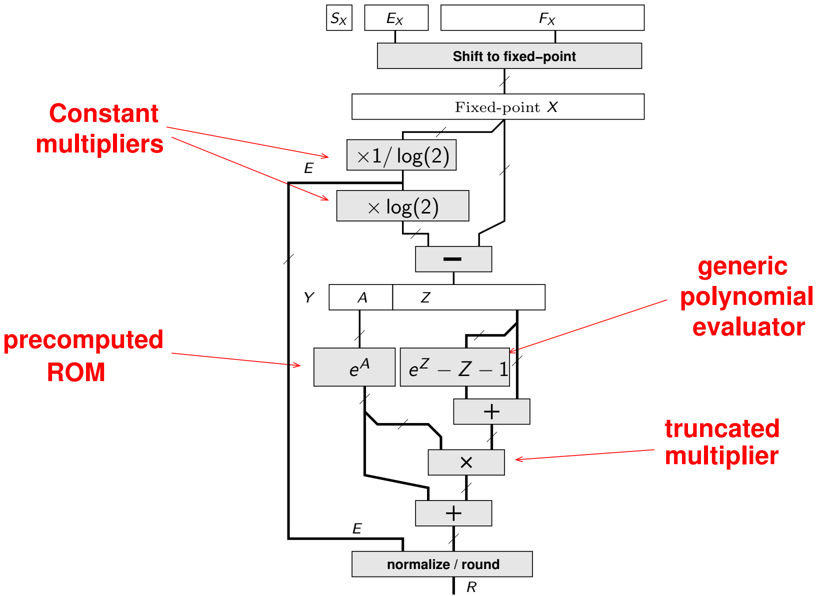
Dura Amdahl lex, sed lex

SPICE Model-Evaluation, cut from Kapre and DeHon (FPL 2009)

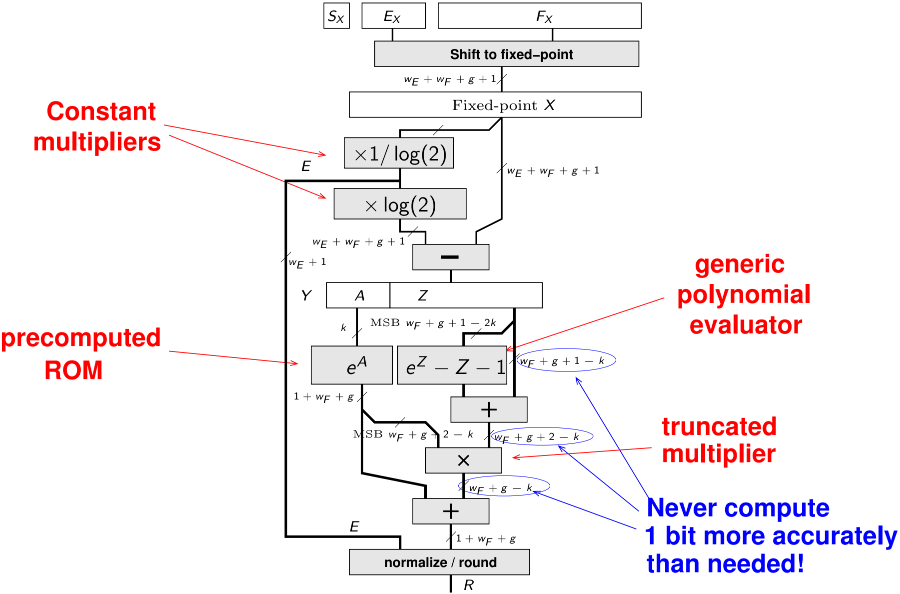
Table 2. Verilog-AMS Compiler Output

Models	Instruction Distribution					
	Add	**Mult.**	**Div.**	**Sqrt.**	**Exp.**	**Log**
`bjt`	22	30	17	0	2	0
`diode`	7	5	4	0	1	2
`hbt`	112	57	51	0	23	18
`jfet`	13	31	2	0	2	0
`mos1`	24	36	7	1	0	0
`vbic`	36	43	18	1	10	4

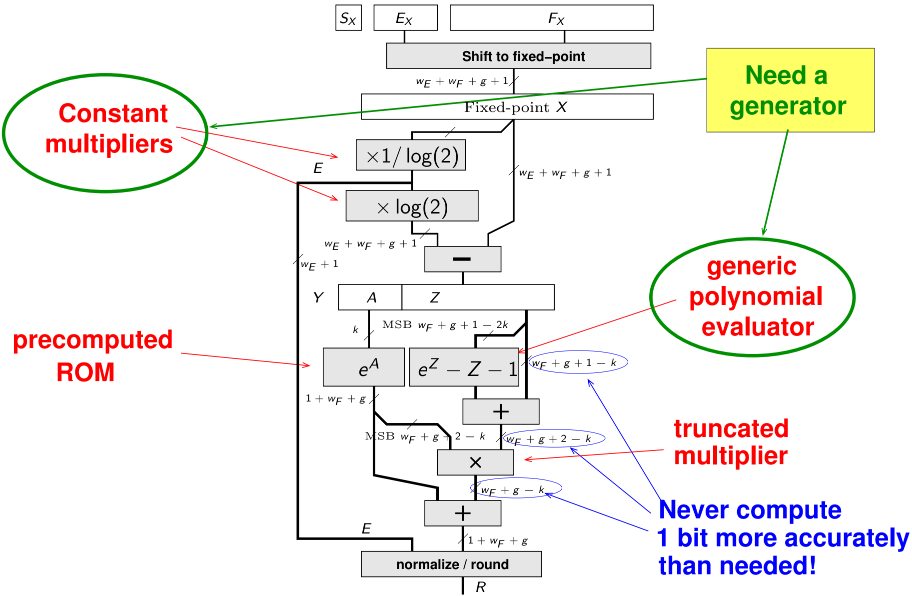
Custom arithmetic (not your Pentium's)



Custom arithmetic (not your Pentium's)



Custom arithmetic (not your Pentium's)



Useful operators that make sense in a processor

- Should a processor include elementary functions ?
  Yes (Paul&Wilson, 1976), No since the transition to RISC
- Should a processor include a divider and square root?
  Yes (Oberman et al, Arith, 1997), No since the transition to FMA (IBM then HP then Intel)
- Should a processor include decimal hardware?
  Yes say IBM, No say Intel
- Should a processor include a multiplier by log(2)?
  No of course.

Useful operators that make sense in an FPGA or ASIC

- Elementary functions ?
  Yes iff your application needs it
- Divider or square root?
  Yes iff your application needs it
- Decimal hardware?
  Yes iff your application needs it
- A multiplier by log(2)?
  Yes iff your application needs it

In FPGAs, useful means: useful to one application.

Enough work to keep me busy to retirement

Arithmetic operators useful to at least one application:

- Elementary functions (sine, exponential, logarithm...)
- Algebraic functions ($\dfrac{x}{\sqrt{x^2+y^2}}$, polynomials, ...)
- Compound functions ($\log_2(1 \pm 2^x)$, e^{-Kt^2}, ...)
- Floating-point sums, dot products, sums of squares
- Specialized operators: constant multipliers, squarers, ...
- Complex arithmetic
- LNS arithmetic
- Decimal arithmetic
- Interval arithmetic
- ...

Enough work to keep me busy to retirement

Arithmetic operators useful to at least one application:

- Elementary functions (sine, exponential, logarithm...)
- Algebraic functions ($\dfrac{x}{\sqrt{x^2+y^2}}$, polynomials, ...)
- Compound functions ($\log_2(1 \pm 2^x)$, e^{-Kt^2}, ...)
- Floating-point sums, dot products, sums of squares
- Specialized operators: constant multipliers, squarers, ...
- Complex arithmetic
- LNS arithmetic
- Decimal arithmetic
- Interval arithmetic
- ...
- Oh yes, basic operations, too.

What do we call arithmetic operators?

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)

What do we call arithmetic operators?

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x) = rounding(operation(x))

$\rightarrow$ Clean mathematical definition (even for floating-point arithmetic)

What do we call arithmetic operators?

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x) = rounding(operation(x))

$\rightarrow$ Clean mathematical definition (even for floating-point arithmetic)

The operator as a *circuit*...

... is a direct acyclic graph (DAG):

- easy to build and pipeline
- easy to test against its mathematical specification

The benefits of custom computing

Example: a floating-point sum of squares

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

The benefits of custom computing

Example: a floating-point sum of squares

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required

The benefits of custom computing

Example: a floating-point sum of squares

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless

The benefits of custom computing

Example: a floating-point sum of squares

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless
- Accuracy can be improved:
  - 5 rounding errors in the floating-point version
  - `(x*x+y*y)+z*z` : asymmetrical

The benefits of custom computing

Example: a floating-point sum of squares

$$x^2 + y^2 + z^2$$

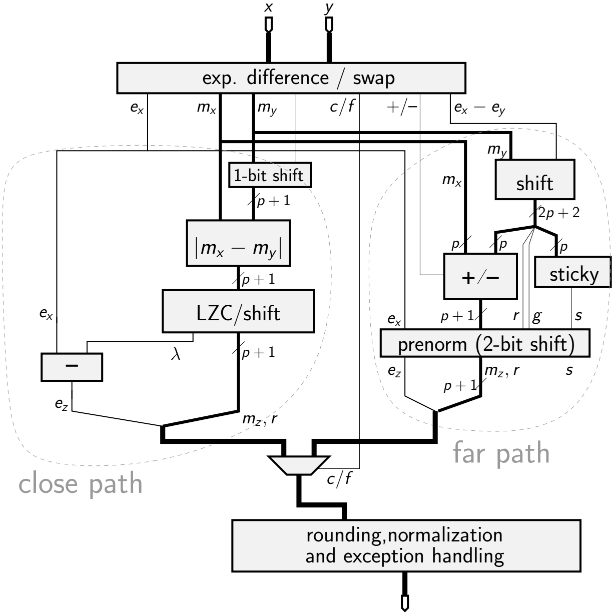
(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless
- Accuracy can be improved:
  - 5 rounding errors in the floating-point version
  - `(x*x+y*y)+z*z` : asymmetrical

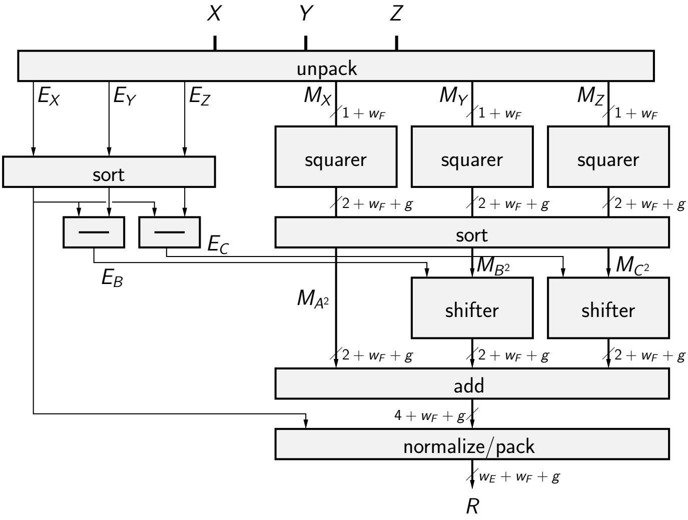
The FloPoCo Recipe

- Floating-point interface for convenience
- Clear accuracy specification for computing just right
- Fixed-point internal architecture for efficiency

A floating-point adder



A fixed-point architecture



The benefits of custom computing

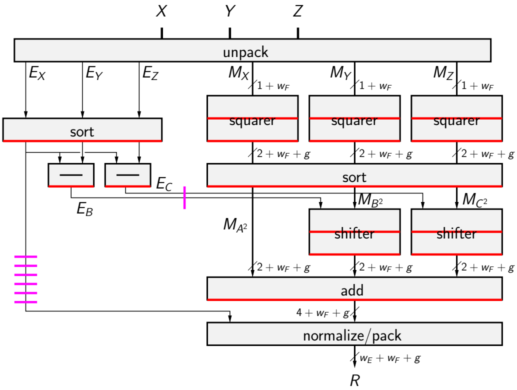
A few results for floating-point sum-of-squares on Virtex4:

Simple Precision	area	performance
LogiCore classic	1282 slices, 20 DSP	43 cycles @ 353 MHz
FloPoCo classic	1188 slices, 12 DSP	29 cycles @ 289 MHz
FloPoCo custom	453 slices, 9 DSP	11 cycles @ 368 MHz

Double Precision	area	performance
FloPoCo classic	4480 slices, 27 DSP	46 cycles @ 276 MHz
FloPoCo custom	1845 slices, 18 DSP	16 cycles @ 362 MHz

- all performance metrics improved, FLOP/s/area more than doubled
- Plus: custom operator more accurate, and symmetrical

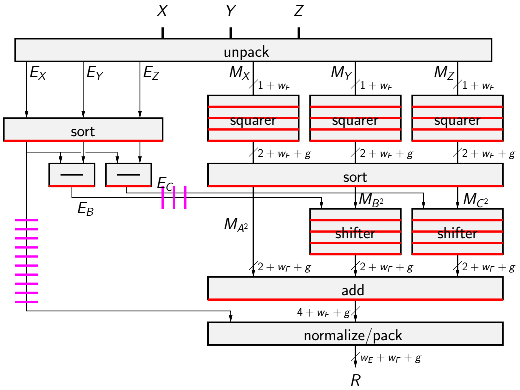
Custom also means: custom pipeline



One operator does not fit all

- Low frequency, low resource consumption

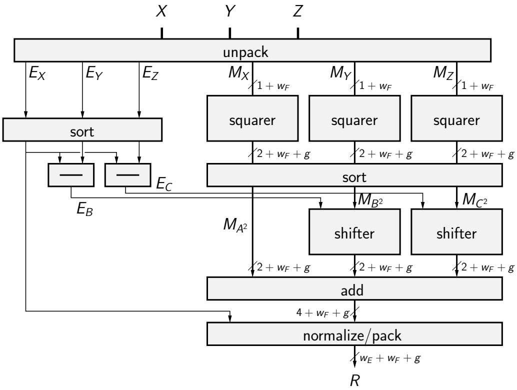
Custom also means: custom pipeline



One operator does not fit all

- Low frequency, low resource consumption
- Faster but larger (more registers)

Custom also means: custom pipeline



One operator does not fit all

- Low frequency, low resource consumption
- Faster but larger (more registers)
- Combinatorial

More slides

- All you ever wanted to know about division by 3
- Application-specific floating-point accumulation
- Architectures computing the floating-point exponential
- ...