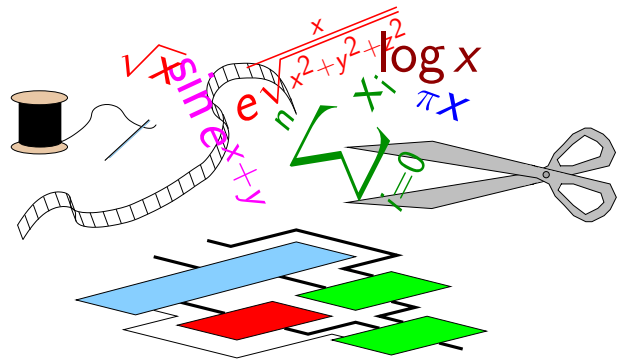
# Computing just right: Application-specific arithmetic

**Florent de Dinechin**




citi lab

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Inria informatiques mathématiques

UNIVERSITÉ DE LYON

# Outline

Anti-introduction: the arithmetic you want in a processor

Operator parameterization

Operator specialization

Operator fusion

Tabulation of pre-computed values

Conclusion: the FloPoCo project

# Anti-introduction: the arithmetic you want in a processor

# Processors are general-purpose

(more or less – Intel and ARM more, GPUs less)
The good arithmetic in a processor is the most generally useful: additions, multiplications, and then?

- *Should a processor include a divider and square root?*
- *Should a processor include elementary functions (exp, log sine/cosine)*
- *Should a processor include decimal hardware?*
- ...

# Should a processor include a divider? (1)

How do you divide $X$ by $D$?

# Should a processor include a divider? (1)

How do you divide $X$ by $D$? As in decimal, but simpler:

## The iteration of the paper-and-pencil algorithm

- find the next quotient digit — *binary: it can be 0 or 1, so try 1*
- multiply this digit by the dividend — *this one is easy*
- subtract from the divisor — *one subtraction here*
- if the result is negative,
  - the quotient digit should have been zero,
  - therefore we should have subtracted 0,
  - it will be easy to fix.
- start again, one digit to the right

Very light iteration (one subtraction and one test),
but each iteration provides only one bit of the quotient:
(more than) **53 cycles** for double-precision floating-point.

# Should a processor include a divider? (2)

Answer in 1993 is : **YES** (Oberman & Flynn, 1993)

# Should a processor include a divider? (2)

Answer in 1993 is : **YES** (Oberman & Flynn, 1993)
And this divider should be a fast one, because of **Amdahl law**.
Although division is not frequent, *(...) a high latency divider can contribute an additional 0.50 CPI to a system executing SPECfp92*

# Should a processor include a divider? (2)

Answer in 1993 is : **YES** (Oberman & Flynn, 1993)
And this divider should be a fast one, because of **Amdahl law**.
Although division is not frequent, *(...) a high latency divider can contribute an additional 0.50 CPI to a system executing SPECfp92*

## Digit recurrence algorithms

Generalizations of the paper-and-pencil algorithm

- large radix: from $2^3$ to $2^6$
- fancy internal number systems to speedup
  - digit-by-number product
  - subtraction
  - finding the next quotient digit

Heavier iterations,
giving one digit (2 to 5 bits) per iteration.

# Should a processor include a divider? (2)

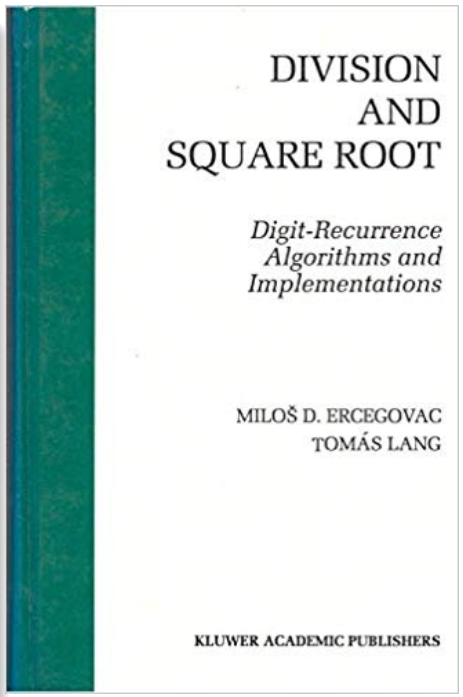
Answer in 1993 is : **YES** (Oberman & Flynn, 1993)
And this divider should be a fast one, because of **Amdahl law**.
Although division is not frequent, *(...) a high latency divider can contribute an additional 0.50 CPI to a system executing SPECfp92*

## Digit recurrence algorithms

Generalizations of the paper-and-pencil algorithm

- large radix: from $2^3$ to $2^6$
- fancy internal number systems to speedup
  - digit-by-number product
  - subtraction
  - finding the next quotient digit

Heavier iterations,
giving one digit (2 to 5 bits) per iteration.




A lot of research, worth one full book (Ercegovac and Lang, 1994)

# Should a processor include a divider? (3)

Answer in 2000 is : **NO** (Markstein)

# Should a processor include a divider? (3)

Answer in 2000 is : **NO** (Markstein)

The Itanium: a brand new processor without a divide instruction.

**Instead of a hardware divider,**

**a second FMA (fused multiply and add) is more generally useful**

# Should a processor include a divider? (3)

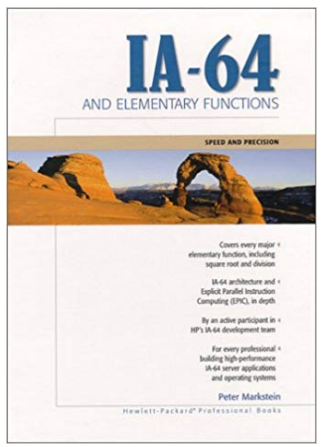
Answer in 2000 is : **NO** (Markstein)
The Itanium: a brand new processor without a divide instruction.
**Instead of a hardware divider,**
**a second FMA (fused multiply and add) is more generally useful**
and can even be used to compute divisions.

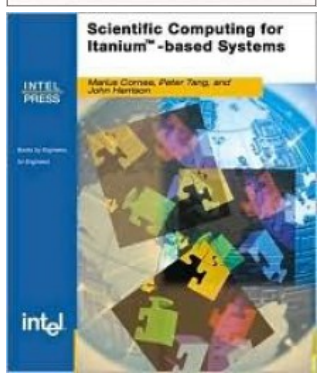
## Multiplicative division algorithms

Executive summary: approximate $1/D$

- Various iterations involving 2 multiplications
  - Newton-Raphson, Goldschmidt, ...
  - Polynomial approximation (Taylor-like), ...
- Each iteration *doubles* the number of correct quotient digits

Heavy iterations, but few of them,
and all the freedom of software.




... and two more books.

# Should a processor include a divider? (4)

Answer in 2018 is : **YES** again (Bruguera, Arith 2018)

# Should a processor include a divider? (4)

Answer in 2018 is : **YES** again (Bruguera, Arith 2018)
Bruguera designs floating-point units for ARM (low-power processors)

# Should a processor include a divider? (4)

Answer in 2018 is : **YES** again (Bruguera, Arith 2018)
Bruguera designs floating-point units for ARM (low-power processors)
Their current divisor is the most expensive you could think of

- Digit-recurrence, but 6 quotients bits per iteration
- 11 cycles for double precision (better than intel, IBM, ...)

# Should a processor include a divider? (4)

Answer in 2018 is : **YES** again (Bruguera, Arith 2018)
Bruguera designs floating-point units for ARM (low-power processors)
Their current divisor is the most expensive you could think of

- Digit-recurrence, but 6 quotients bits per iteration
- 11 cycles for double precision (better than intel, IBM, ...)

Achieved thanks to a totally redneck implementation

- speculation all over the place
- prescaling and other tricks
- iteration hardware: **20** fast 58-bit adders, **12** 58-bit muxes, and more...

# Should a processor include a divider? (4)

Answer in 2018 is : **YES** again (Bruguera, Arith 2018)
Bruguera designs floating-point units for ARM (low-power processors)
Their current divisor is the most expensive you could think of

- Digit-recurrence, but 6 quotients bits per iteration
- 11 cycles for double precision (better than intel, IBM, ...)

Achieved thanks to a totally redneck implementation

- speculation all over the place
- prescaling and other tricks
- iteration hardware: **20** fast 58-bit adders, **12** 58-bit muxes, and more...

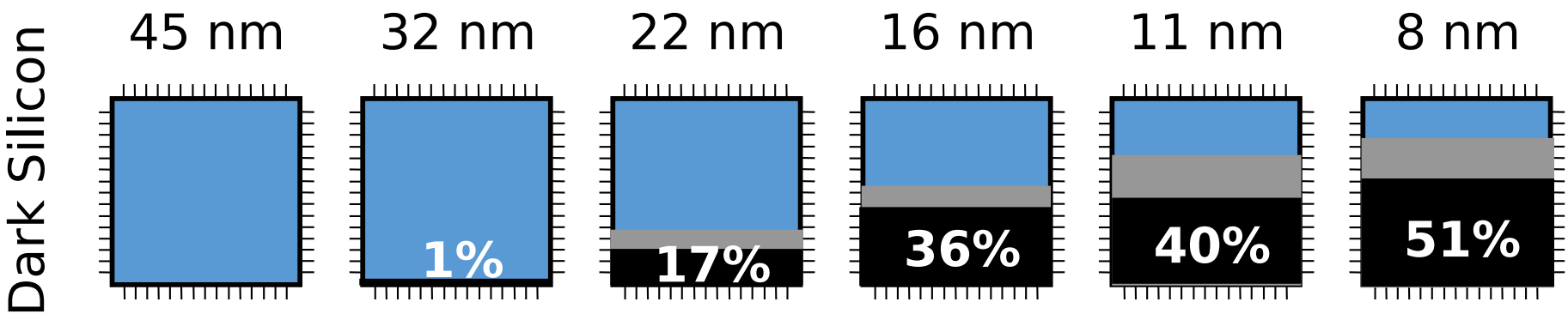
*We do this to reduce overal energy consumption!*
*There is this huge superscalar ARM core that consumes a lot,*
*we save energy if we can switch it off a few cycles earlier*

# A good example of dark silicon made useful

## Dark silicon?

In current tech, you can no longer
use 100% of the transistors 100% of the time
without destroying your chip.

"Dark silicon" is the percentage that must be off at a given time




(picture from a 2013 HiPEAC keynote by Doug Burger)

# Pleasant times to be an architect

## One way out the dark silicon apocalypse (M.B. Taylor, 2012)

Hardware implementations of rare (but useful) operations:

- when used, dramatically reduce the energy per operation (compared to a software implementation that would take many more cycles)
- when unused, serve as radiator for the used parts

# Should a processor include elementary functions? (1)

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 |

Current performance of exp or log is 10 to 100 cycles, to compare with 1 to 5 cycles for add and mult.

# Should a processor include elementary functions? (2)

Answer in 1976 is **YES** (Paul&Wilson)

# Should a processor include elementary functions? (2)

Answer in 1976 is **YES** (Paul&Wilson)
... and the initial x87 floating-point coprocessor was designed with a basic set of elementary functions

- implemented in microcode
- with some hardware assistance, in particular the 80-bit extended format.

# Should a processor include elementary functions? (3)

Answer in 1991 is **NO** (Tang)

# Should a processor include elementary functions? (3)

Answer in 1991 is **NO** (Tang)

## Table-based algorithms

- Moore's Law means cheap memory
- Fast algorithms thanks to huge (tens of Kbytes!)
  **tables of pre-computed values**
- Software beats micro-code, which cannot afford such tables

# Should a processor include elementary functions? (3)

Answer in 1991 is **NO** (Tang)

## Table-based algorithms

- Moore's Law means cheap memory
- Fast algorithms thanks to huge (tens of Kbytes!)
  **tables of pre-computed values**
- Software beats micro-code, which cannot afford such tables

None of the RISC processors designed in this period
even considers elementary functions support

# Should a processor include elementary functions? (4)

Answer in 2018 is... **maybe**?

# Should a processor include elementary functions? (4)

Answer in 2018 is... **maybe**?

- A few low-precision hardware functions in NVidia GPUs (Oberman & Siu 2005)
- The SpiNNaker-2 chip includes hardware exp and log (Mikaitis et al. 2018)
- Intel AVX-512 includes all sort of fancy floating-point instructions to speed up elementary function evaluation (Anderson et al. 2018)

# I won't answer the other questions here

... because we are working on them

- ✓ *Should a processor include a divider and square root?*
- ✓ *Should a processor include elementary functions (exp, log sine/cosine)*
- *Should a processor include decimal hardware?*
- ...

# At this point of the talk...

... everybody is wondering when I start talking about FPGAs.

# One nice thing with FPGAs

... is that there is an easy answer to all these questions

| | |
|---|---|
| ✓ *divider? square root?* | Yes iff your application needs it |
| ✓ *elementary functions?* | Yes iff your application needs it |
| ✓ *decimal hardware?* | Yes iff your application needs it |

# One nice thing with FPGAs

... is that there is an easy answer to all these questions

- ✓ *divider? square root?* Yes iff your application needs it
- ✓ *elementary functions?* Yes iff your application needs it
- ✓ *decimal hardware?* Yes iff your application needs it
- ✓ *multiplier by* $\log(2)$*? By* $\sin\frac{17\pi}{256}$*?* Yes iff your application needs it

# One nice thing with FPGAs

... is that there is an easy answer to all these questions

- ✓ *divider? square root?* Yes iff your application needs it
- ✓ *elementary functions?* Yes iff your application needs it
- ✓ *decimal hardware?* Yes iff your application needs it
- ✓ *multiplier by* $\log(2)$*? By* $\sin\frac{17\pi}{256}$? Yes iff your application needs it
  - there probably never will be an instruction "multiply by $log(2)$" in a general purpose processor.
- ...

In FPGAs, useful means: useful to one application.

# In an FPGA, you pay only for what you need

If your application is to simulate `jfet`,

**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 |

... you want to build a floating-point unit with 13 adds, 31 mults, 2 divs, 2 exps, **and nothing more**.

# Conclusion so far: FPGA arithmetic is ...

... all sorts of operators that just wouldn't make sense in a processor.

## 4 recipes to exploit the flexibility of FPGAs

- operator parameterization
- operator specialization
- operator fusion
- tabulation of precomputed values

# Conclusion so far: FPGA arithmetic is ...

... all sorts of operators that just wouldn't make sense in a processor.

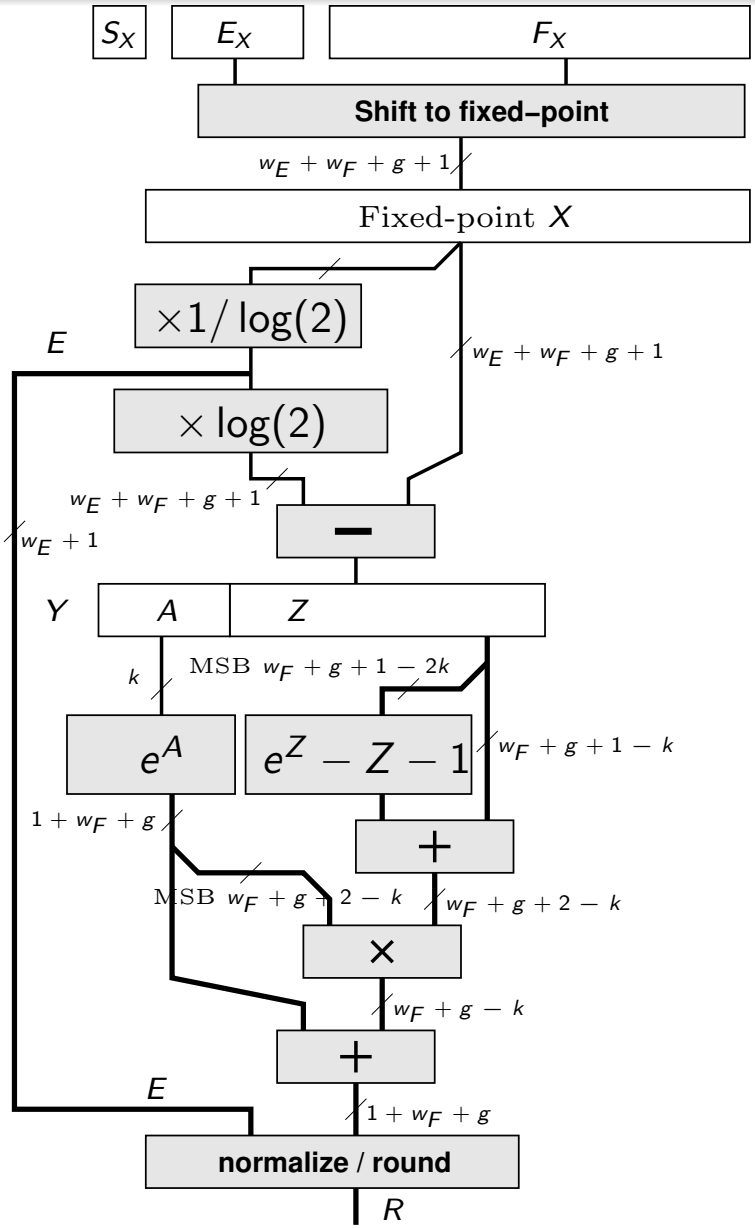
## 4 recipes to exploit the flexibility of FPGAs

- operator parameterization
- operator specialization
- operator fusion
- tabulation of precomputed values

(I hesitated to add a fifth: fancy number systems)

# Operator parameterization

# Example: an architecture for floating-point exponential



# Don't move useless bits around!

- In software, you have to make dramatic choices between a few integer formats and a few floating-point ones.

# Don't move useless bits around!

- In software, you have to make dramatic choices between a few integer formats and a few floating-point ones.
- When designing for FPGAs, bit-level freedom!
  - in this exponential, some signals are 12 bits, some 69 bits.

# Don't move useless bits around!

- In software, you have to make dramatic choices between a few integer formats and a few floating-point ones.
- When designing for FPGAs, bit-level freedom!
  - in this exponential, some signals are 12 bits, some 69 bits.

## Overwhelming freedom! Too many parameters!

Fortunately, we have constraints:

- Computing just right:
  a high-level constraint of **overal accuracy** (to be defined).

# Don't move useless bits around!

- In software, you have to make dramatic choices between a few integer formats and a few floating-point ones.
- When designing for FPGAs, bit-level freedom!
  - in this exponential, some signals are 12 bits, some 69 bits.

## Overwhelming freedom! Too many parameters!

Fortunately, we have constraints:

- Computing just right:
  a high-level constraint of **overal accuracy** (to be defined).
- A few resource/performance constraints:
  - dimensions of DSP and RAM blocks
  - LUT cluster size,
  - ...

# Don't move useless bits around!

- In software, you have to make dramatic choices between a few integer formats and a few floating-point ones.
- When designing for FPGAs, bit-level freedom!
  - in this exponential, some signals are 12 bits, some 69 bits.

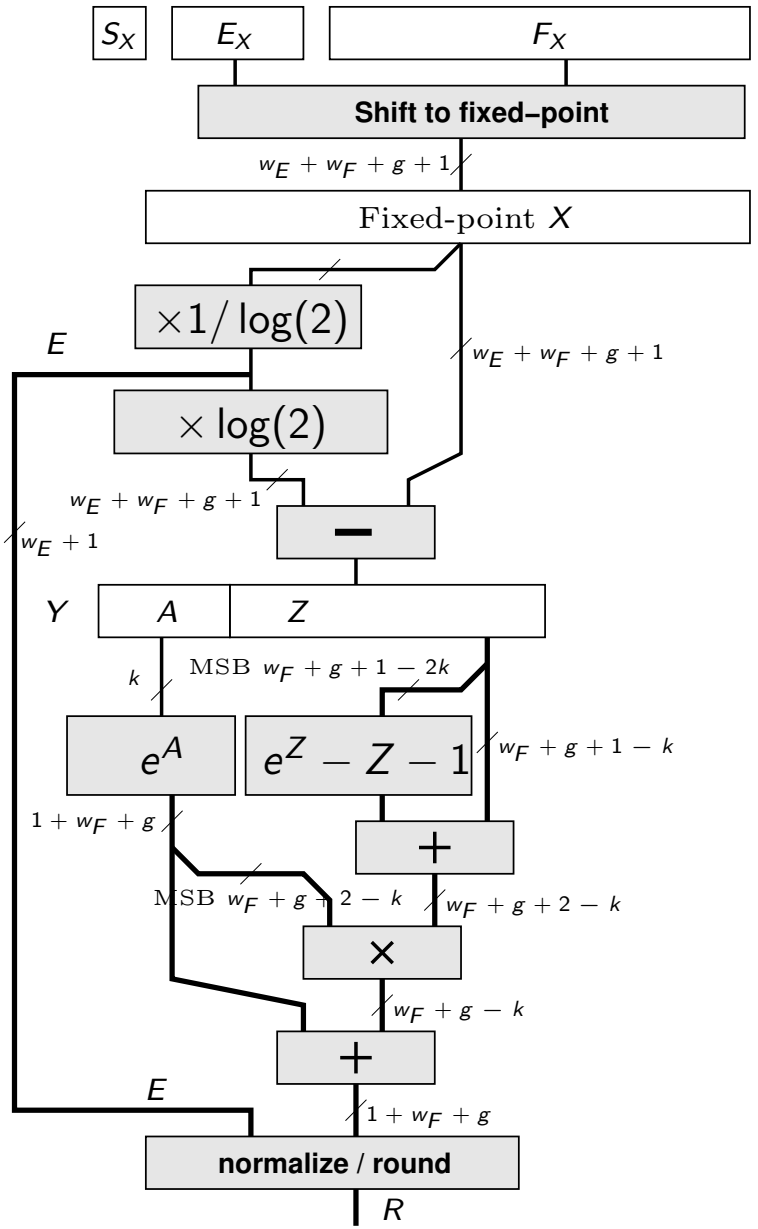
## Overwhelming freedom! Too many parameters!

Fortunately, we have constraints:

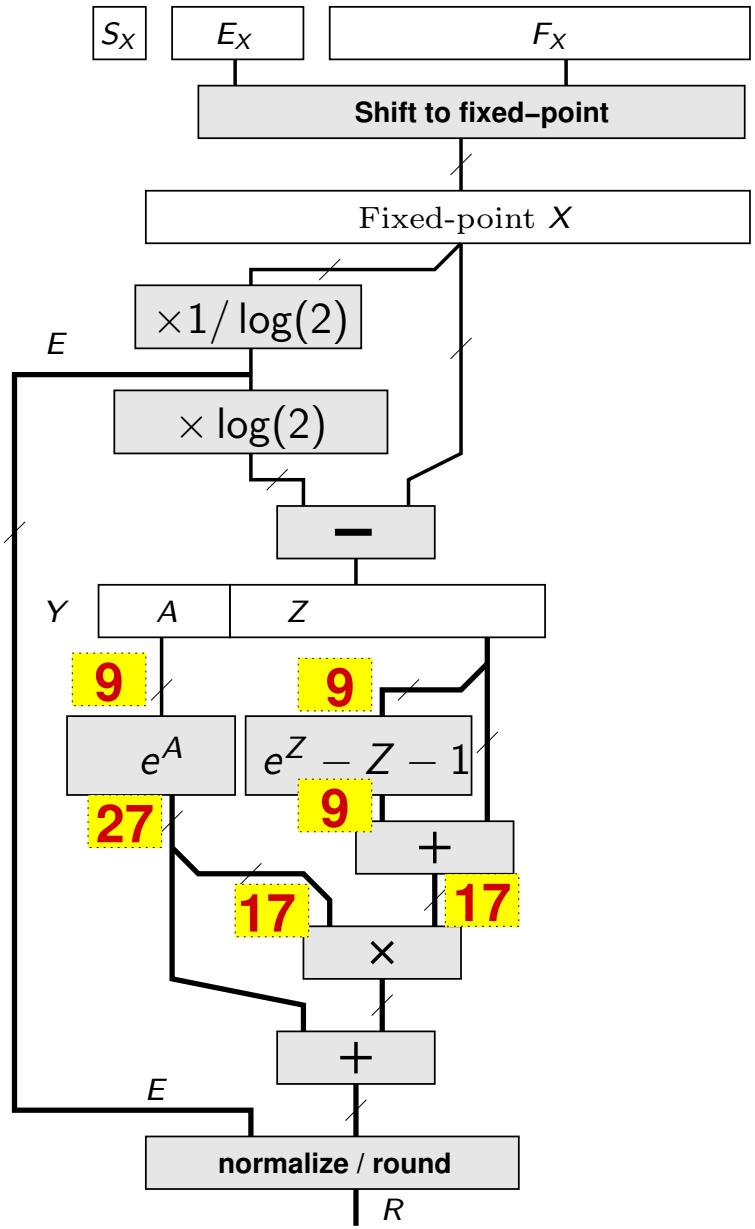
- Computing just right:
  a high-level constraint of **overal accuracy** (to be defined).
- A few resource/performance constraints:
  - dimensions of DSP and RAM blocks
  - LUT cluster size,
  - ...

... to guide you when navigating the implementation space

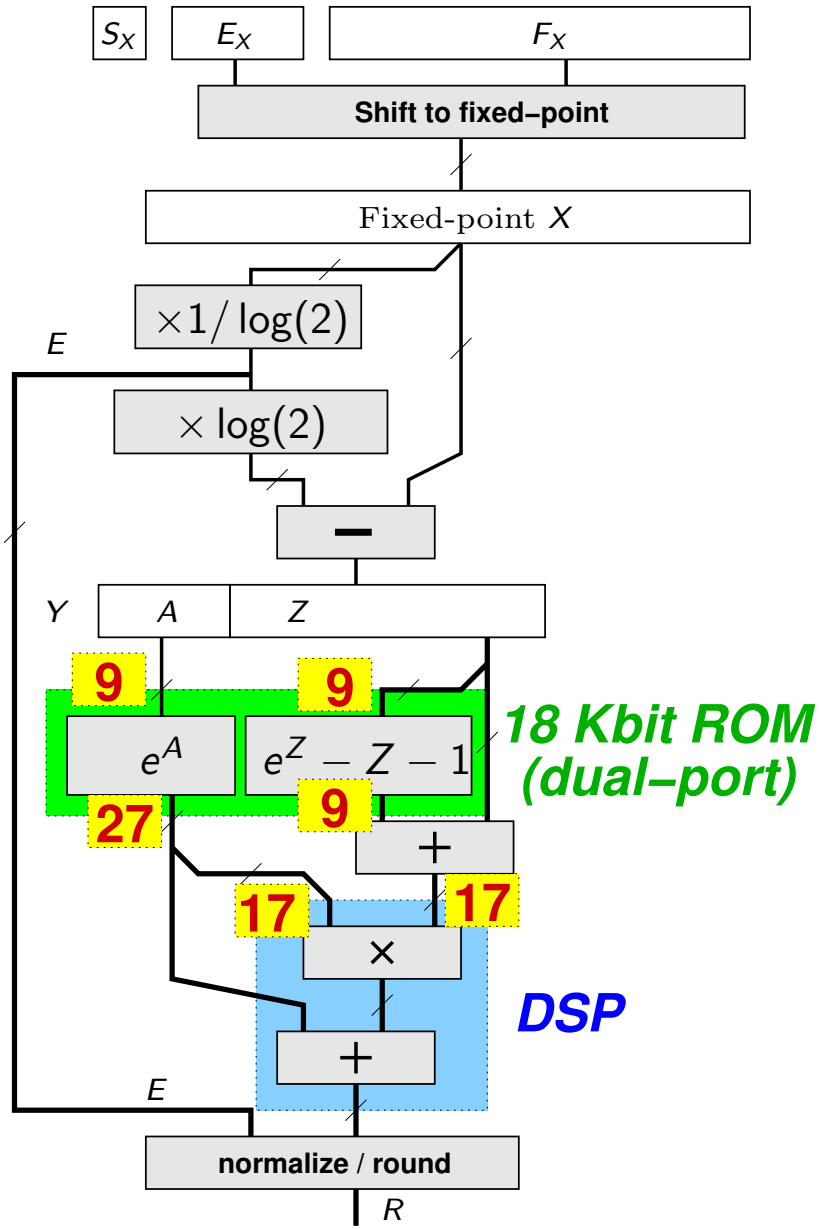
# Example: single precision exponential



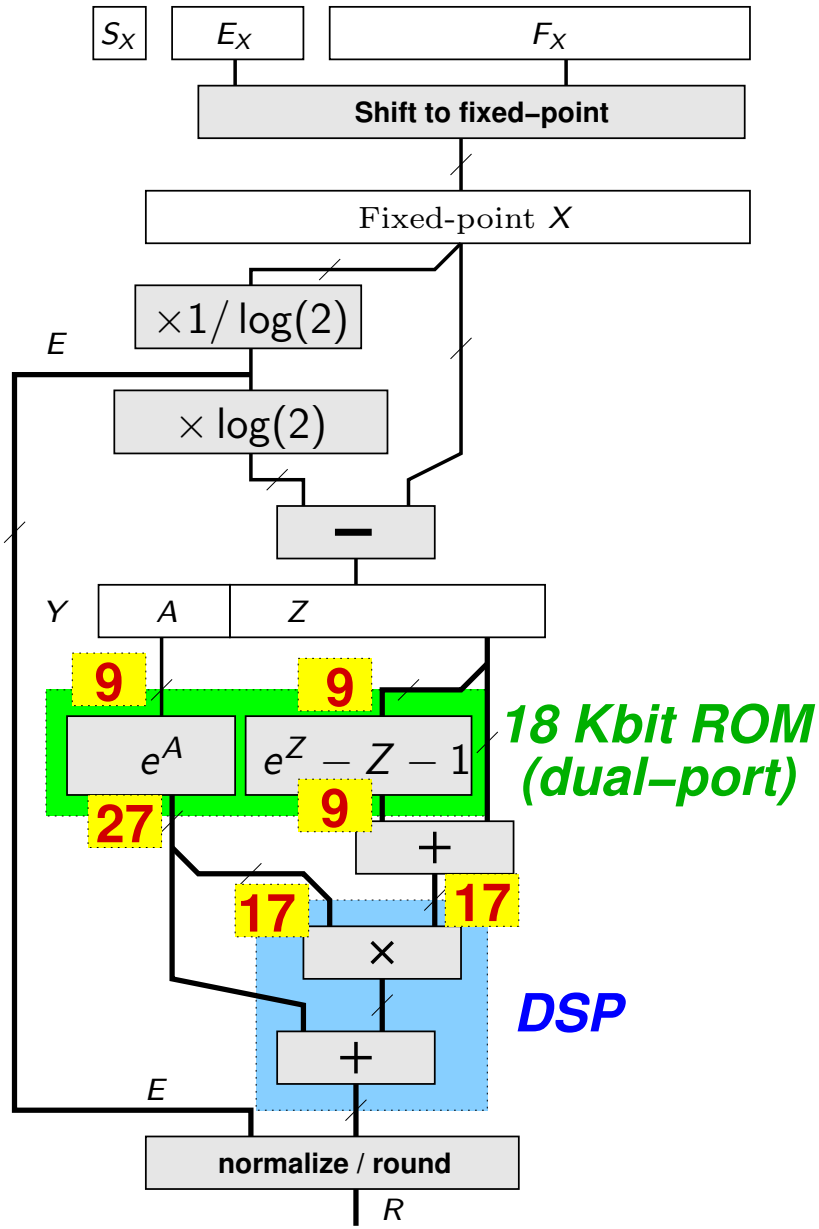
# Example: single precision exponential



# Example: single precision exponential



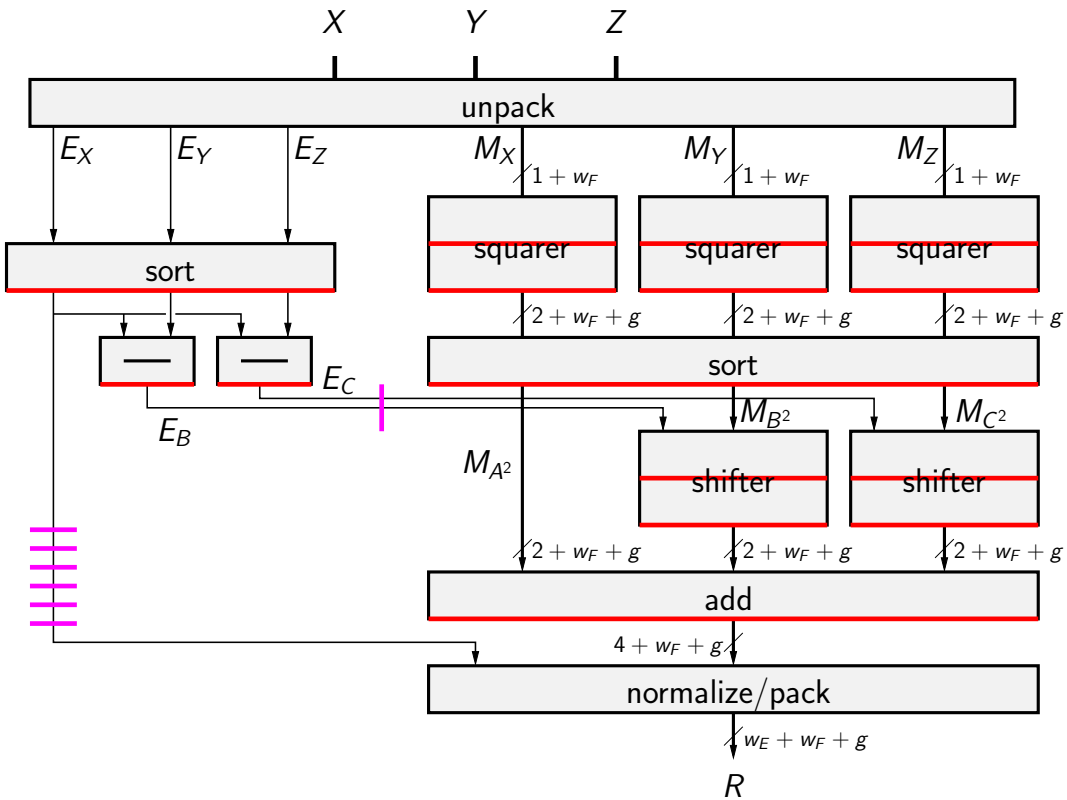
# Example: single precision exponential




## Virtex-4 consumption

- 1 BlockRAM,
- 1 DSP,
- and <400 slices

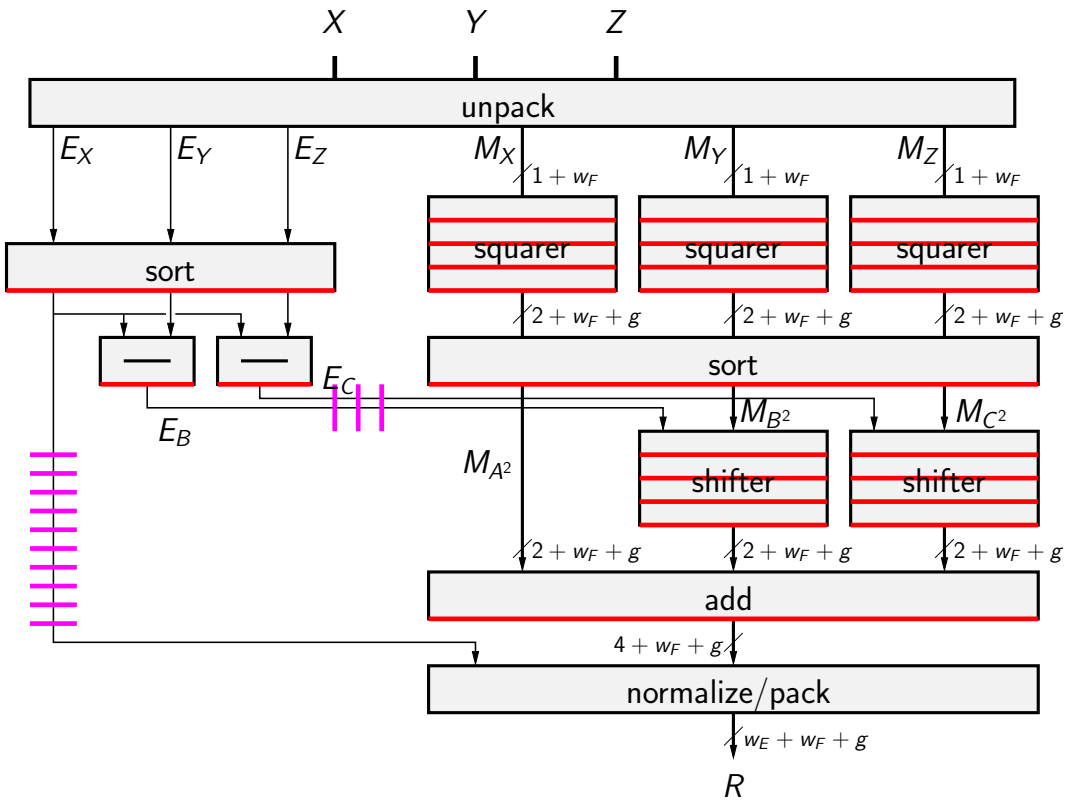
# Adapting to the performance context




## One operator does not fit all

- Low frequency, low resource consumption

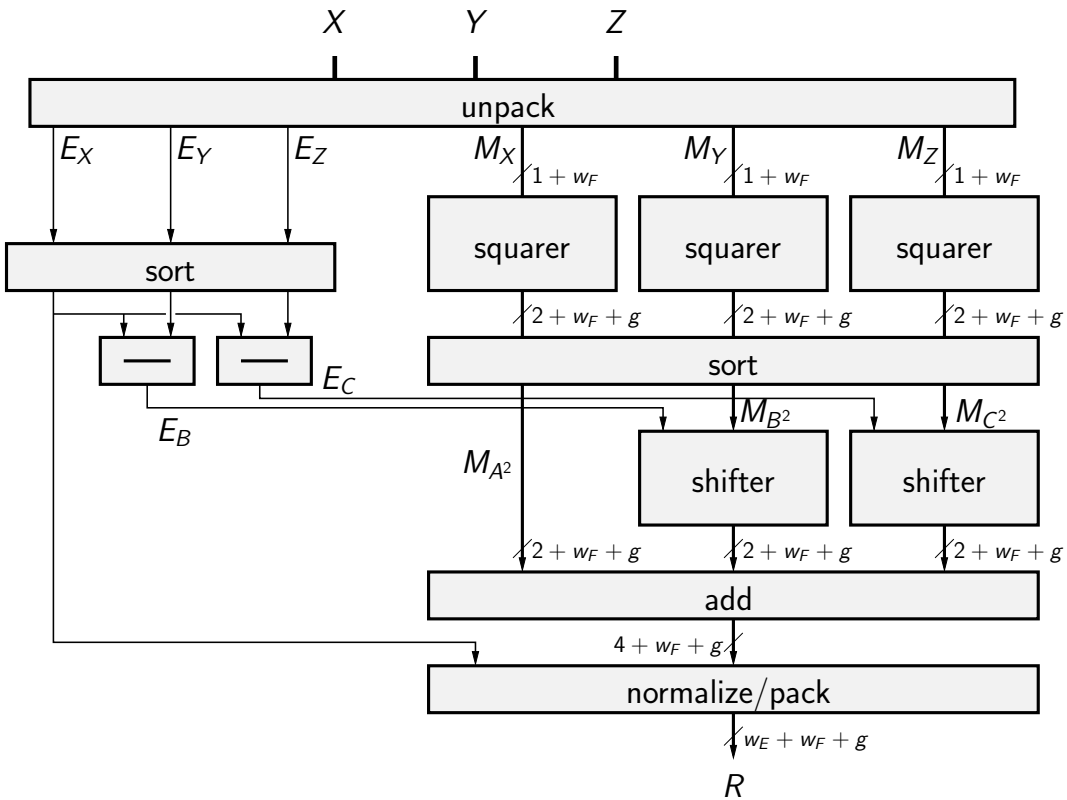
# Adapting to the performance context




## One operator does not fit all

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

# Adapting to the performance context




## One operator does not fit all

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

# Frequency-directed pipelining

## The good interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

# Frequency-directed pipelining

## The good interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by the early core generators of FPGA vendors ...

# Frequency-directed pipelining

## The good interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by the early core generators of FPGA vendors ...

## Better because *compositional*

When you assemble components working at frequency $f$, you obtain a component working at frequency $f$.

# Conclusion about operator parameterization

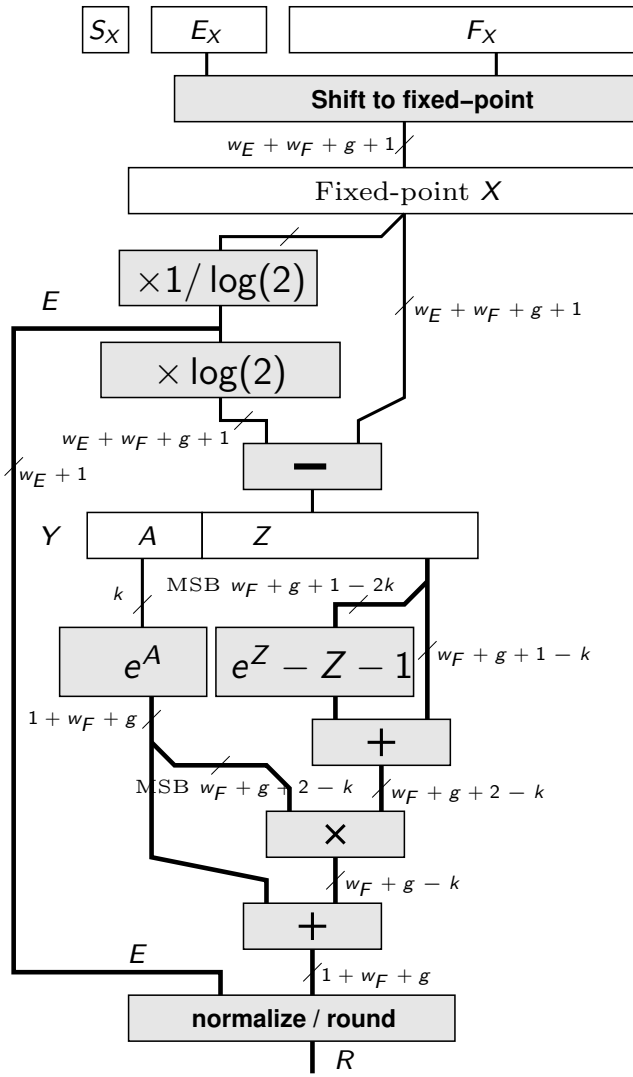
- Designing heavily parameterized operators
  is a lot more work,

# Conclusion about operator parameterization

- Designing heavily parameterized operators is a lot more work,
- but it is the **easy** part
- Chosing the value of the parameters is the **difficult** part
  - Error analysis needed
  - ... context-specific implicit knowledge

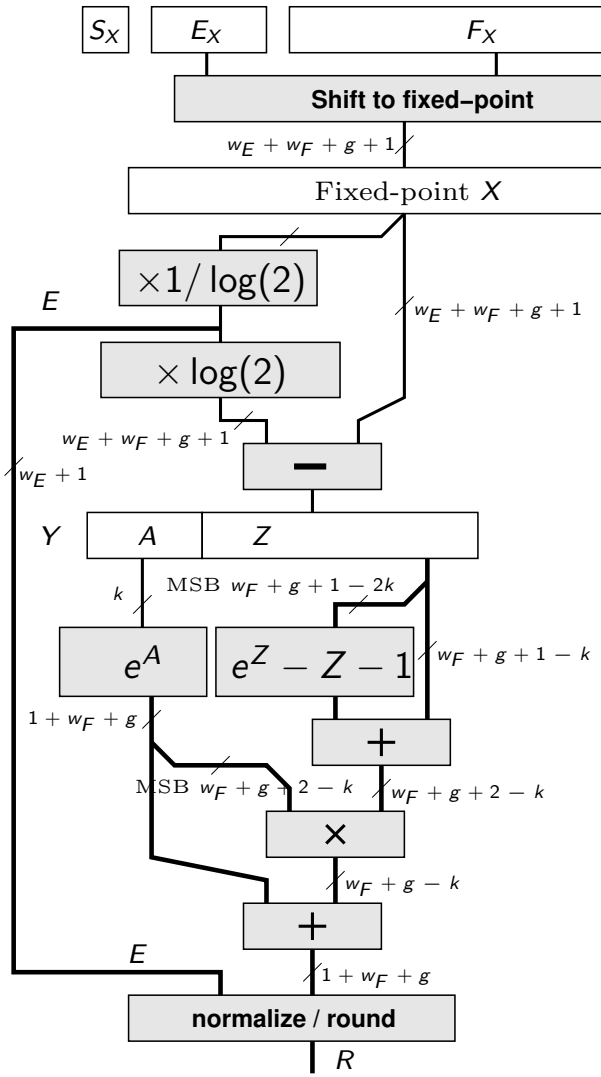
# Conclusion about operator parameterization

- Designing heavily parameterized operators is a lot more work,
- but it is the **easy** part
- Chosing the value of the parameters is the **difficult** part
  - Error analysis needed
  - ... context-specific implicit knowledge
- Parameterization is useful
  - at the application level,
  - but also when designing compound components.



# Conclusion about operator parameterization

- Designing heavily parameterized operators is a lot more work,
- but it is the **easy** part
- Chosing the value of the parameters is the **difficult** part
  - Error analysis needed
  - ... context-specific implicit knowledge
- Parameterization is useful
  - at the application level,
  - but also when designing compound components.
- Fancy situations will occur
  - example: the multiplier by log(2):
    - small input (12 bits for FP64)
    - large output (69 bits for FP64)



# Operator specialization

Anti-introduction: the arithmetic you want in a processor

Operator parameterization

Operator specialization

Operator fusion

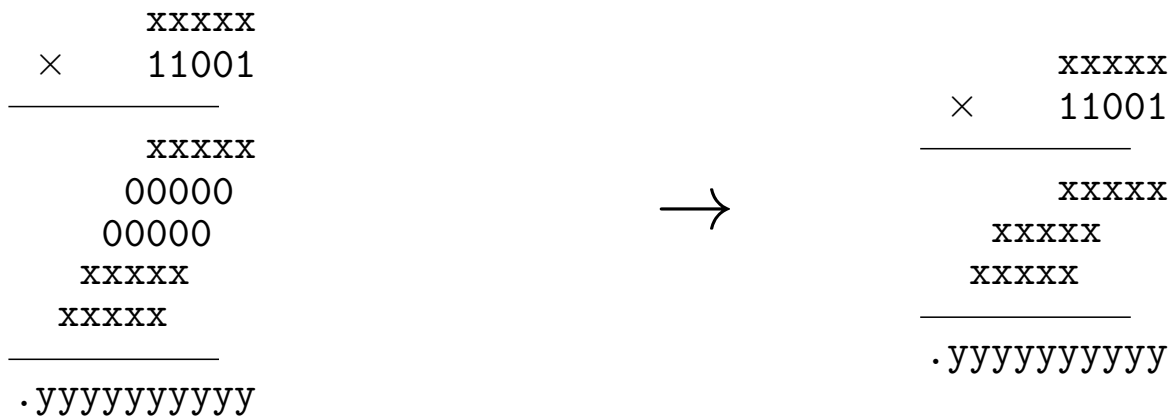
Tabulation of pre-computed values

Conclusion: the FloPoCo project

# Specializing an operator to its context

First idea: design a specific architecture when one input is constant

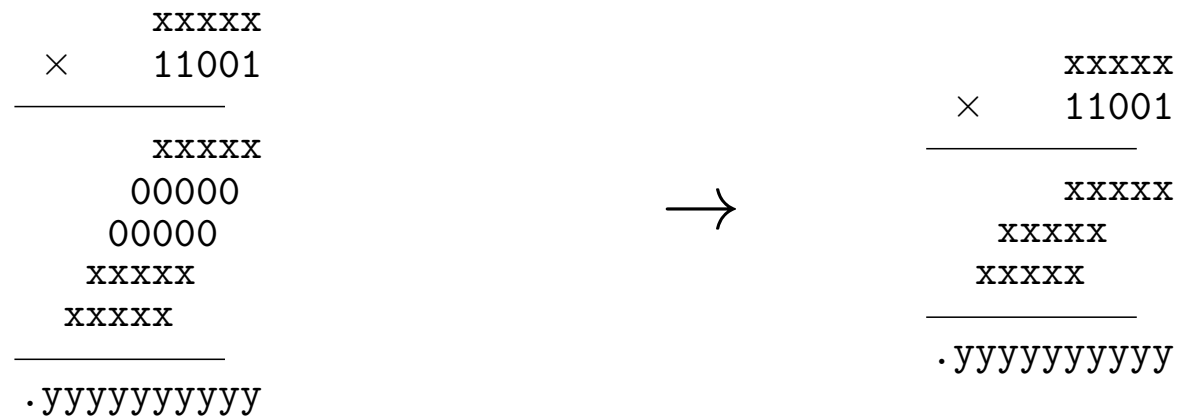
- **multiplier by a constant**
  more efficient than inputting the constant to a standard multiplier



# Specializing an operator to its context

First idea: design a specific architecture when one input is constant

- **multiplier by a constant**
  more efficient than inputting the constant to a standard multiplier

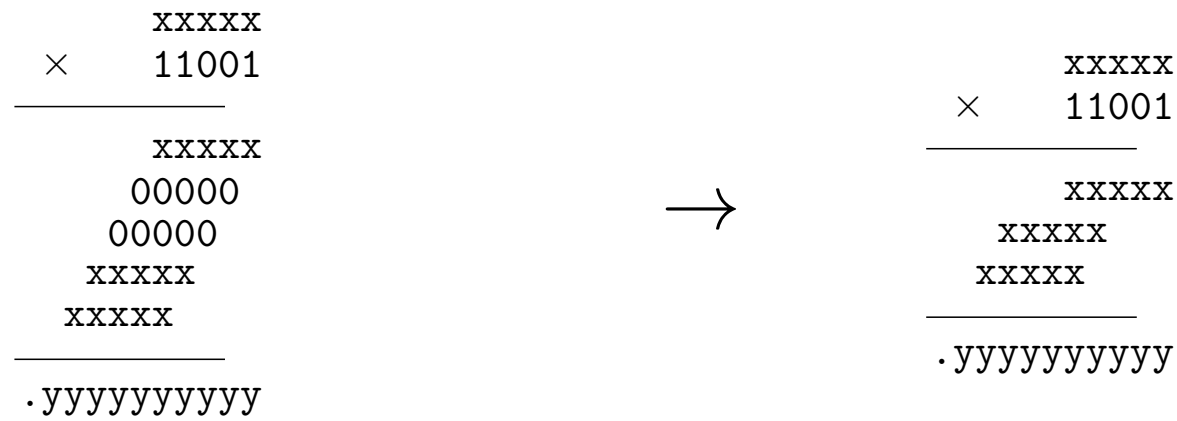



  - two competitive well-researched techniques, tens of publications
  - (well beyond what synthesis tools would optimize out – details later)

# Specializing an operator to its context

First idea: design a specific architecture when one input is constant

- **multiplier by a constant**
  more efficient than inputting the constant to a standard multiplier



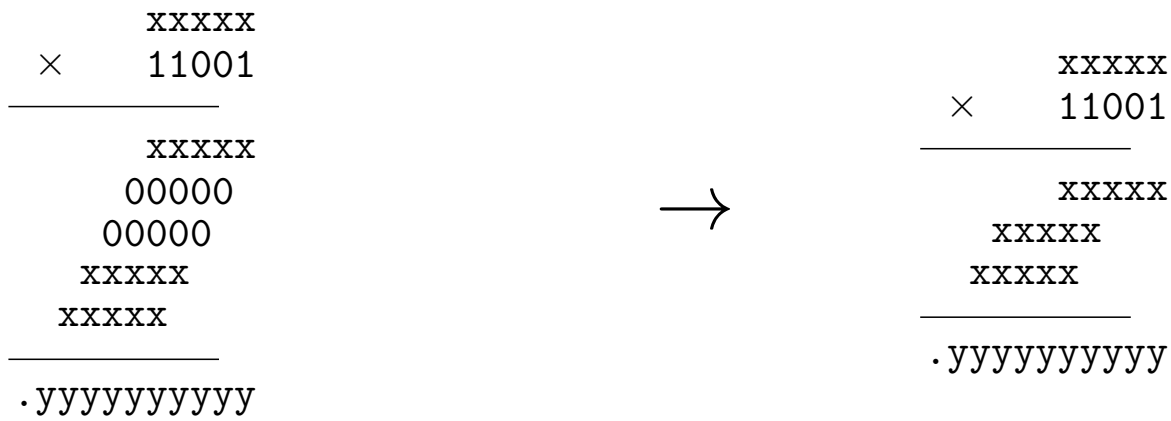

  - two competitive well-researched techniques, tens of publications
  - (well beyond what synthesis tools would optimize out – details later)

- **divider by 3**
  much more efficient than inputting 3 to a standard divider
  - and even more efficient than multiplying by 1/3
  - (technique shown later)
  - Here, we use a completely different algorithm

# Specializing an operator to its context

First idea: design a specific architecture when one input is constant

- **multiplier by a constant**
  more efficient than inputting the constant to a standard multiplier

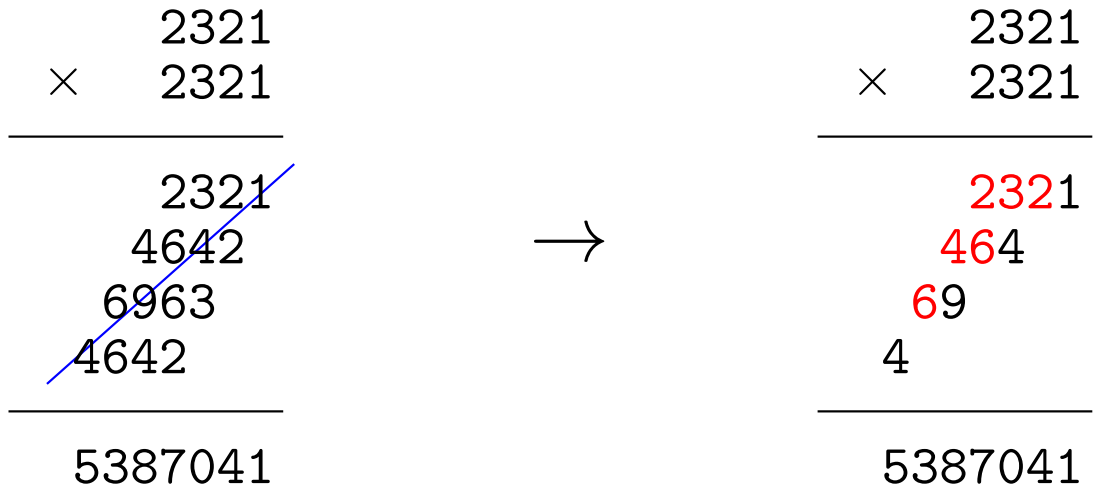



  - two competitive well-researched techniques, tens of publications
  - (well beyond what synthesis tools would optimize out – details later)
- **divider by 3**
  much more efficient than inputting 3 to a standard divider
  - and even more efficient than multiplying by 1/3
  - (technique shown later)
  - Here, we use a completely different algorithm
- (addition of a constant doesn't save much on an FPGA in general)

# Specializing an operator to its context

Second idea: shared inputs

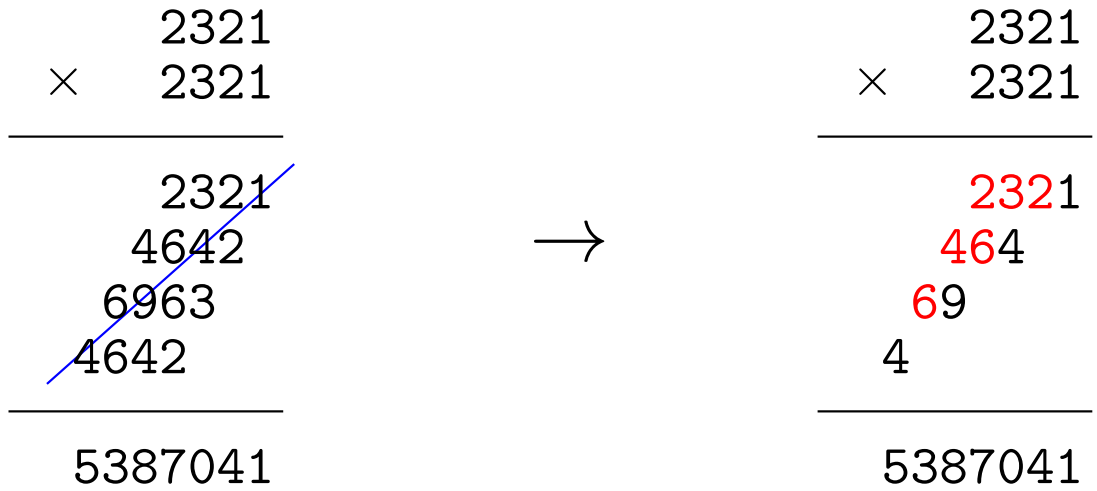
- **squarer** more efficient than multiplier
  - each digit-by digit product is computed twice in a squarer



# Specializing an operator to its context

Second idea: shared inputs

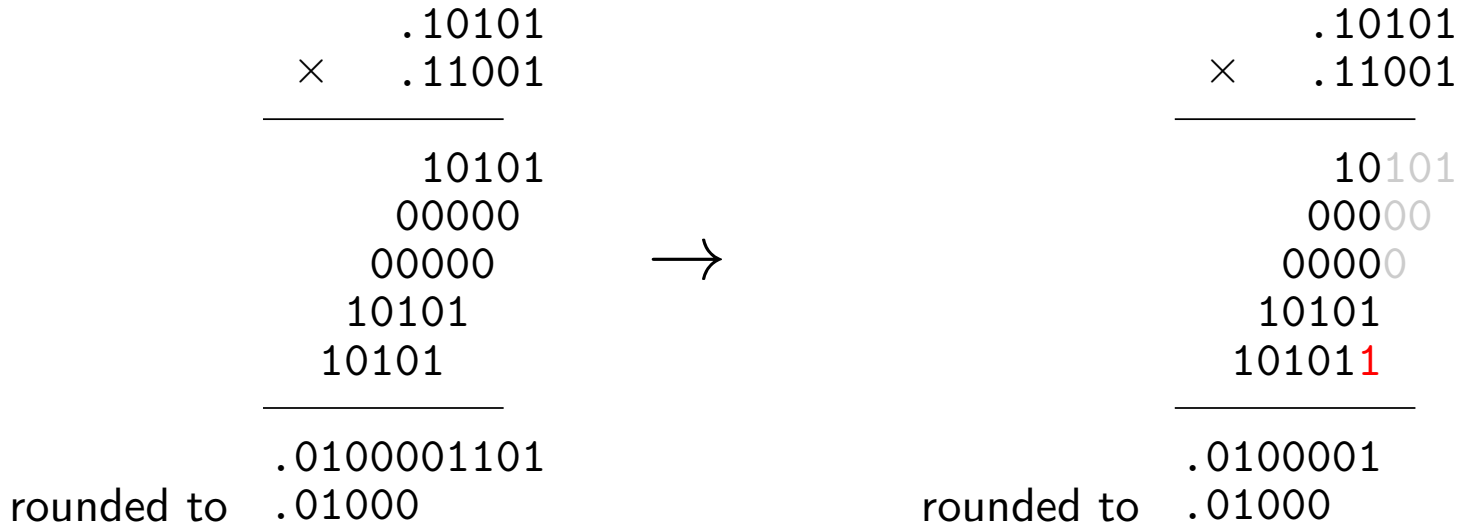
- **squarer** more efficient than multiplier
  - each digit-by digit product is computed twice in a squarer




- Same idea works for $x^3$, etc
- ...

# More subtle operator specialization (1)

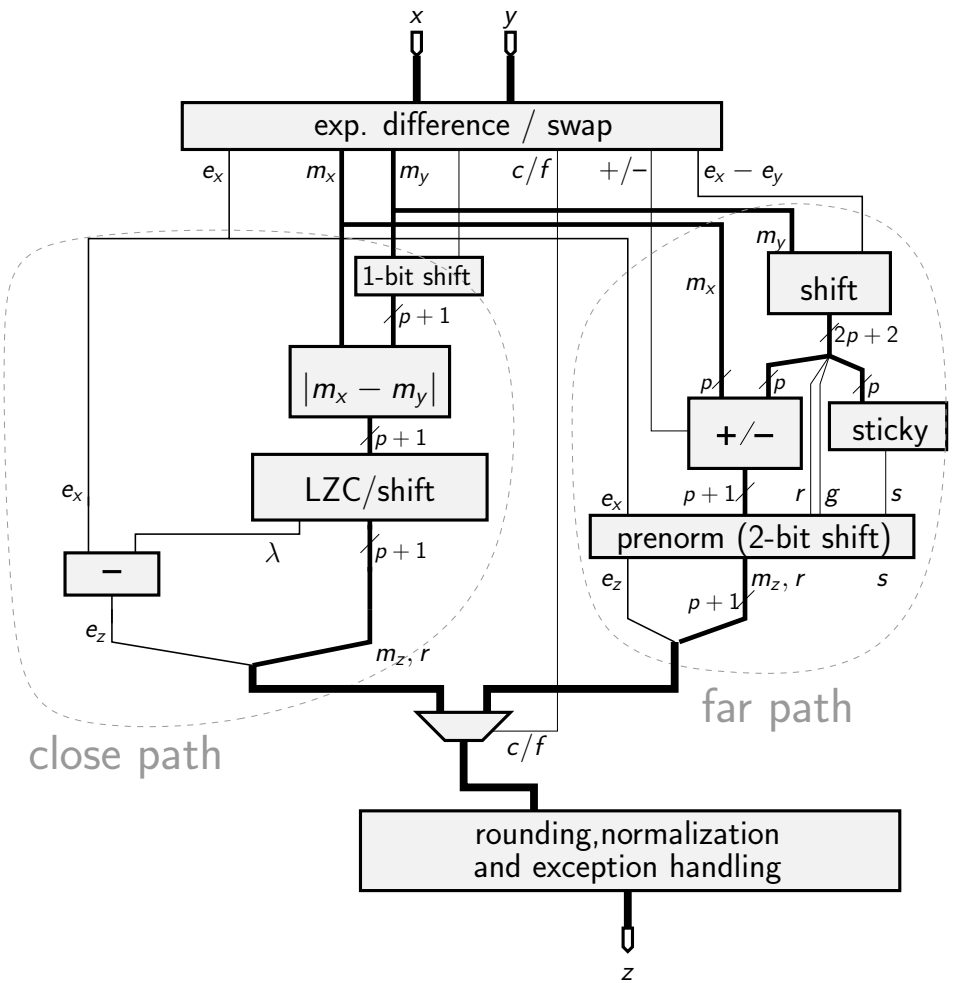
- **truncated multiplier** in fixed point




  - same accuracy with truncated(n+1) as with standard(n)
  - almost half the cost

# More subtle operator specialization (2)

- **Floating-point addition of two numbers of the same sign**
  - This happens in sum of squares, etc – or when physics tells you!
  - one leading-zero counter and one shifter can be saved:



# More subtle operator specialization (3)

- **Fixed-point large accumulator** of floating-point values
  - ... when the physics tells you so
  - (to be detailed later)

# More subtle operator specialization (3)

- **Fixed-point large accumulator** of floating-point values
  - ... when the physics tells you so
  - (to be detailed later)
- **Elementary functions that work only on a smaller range**
  - ... when the physics tells you so

# More subtle operator specialization (3)

- **Fixed-point large accumulator** of floating-point values
  - ... when the physics tells you so
  - (to be detailed later)
- **Elementary functions that work only on a smaller range**
  - ... when the physics tells you so
- ...

# Conclusion on operator specialization

Look at your equations,
they are full of operations waiting to be specialized

# Operator fusion

$\frac{x}{\sqrt{x^2+y^2}}$ really more complex than $x/y$ ?

- From the hardware point of view: same black box
- From the mathematical point of view: both are algebraic functions

# A simpler example: floating-point sum of squares

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

(not a toy example but a useful building block)

# A simpler example: 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

# A simpler example: 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

# A simpler example: 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
  - $(x^2 + y^2) + z^2$ : asymmetrical

# A simpler example: floating-point sum of squares

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

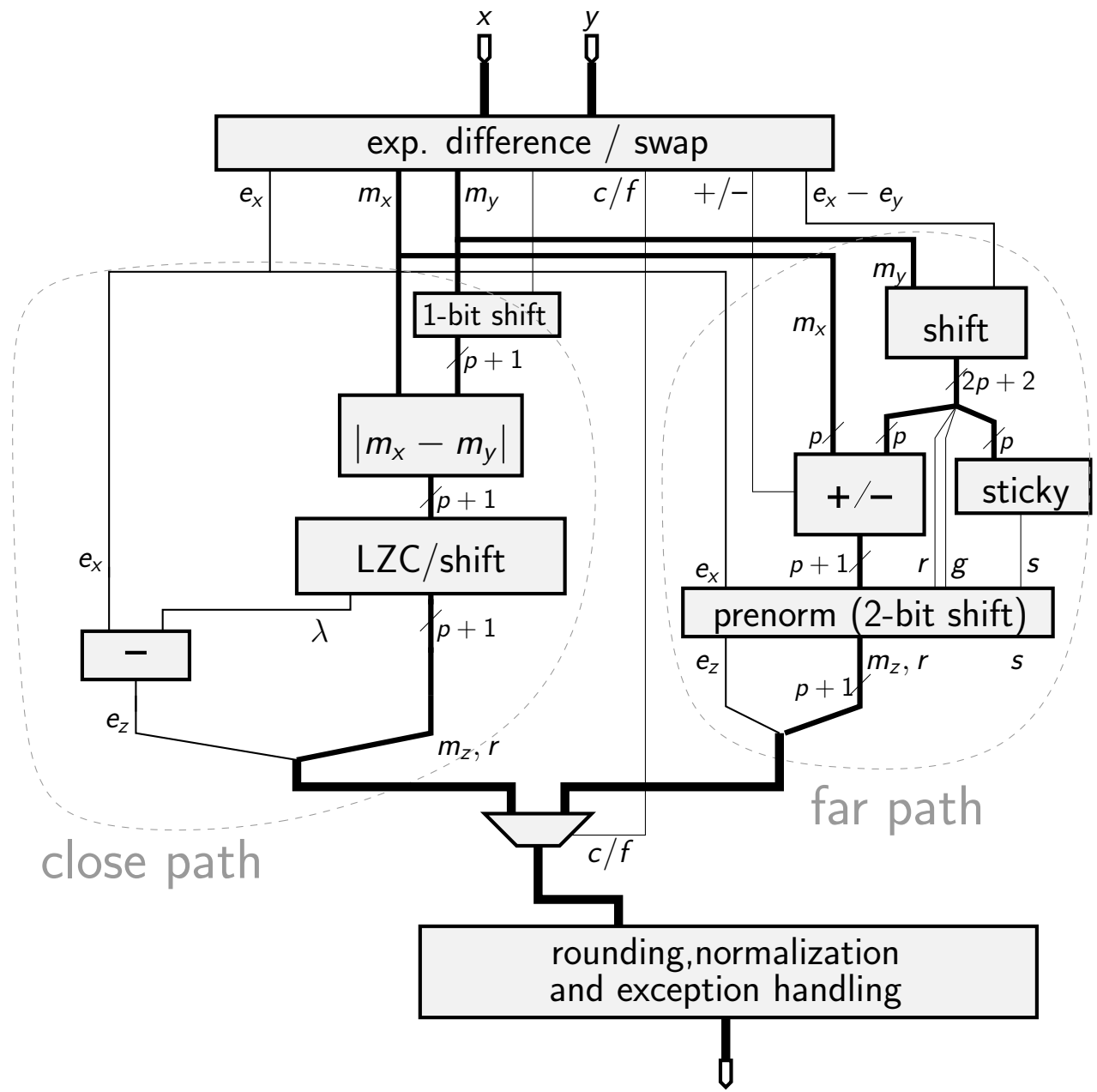
(not a toy example but a useful building block)

- 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
  - $(x^2 + y^2) + z^2$ : asymmetrical

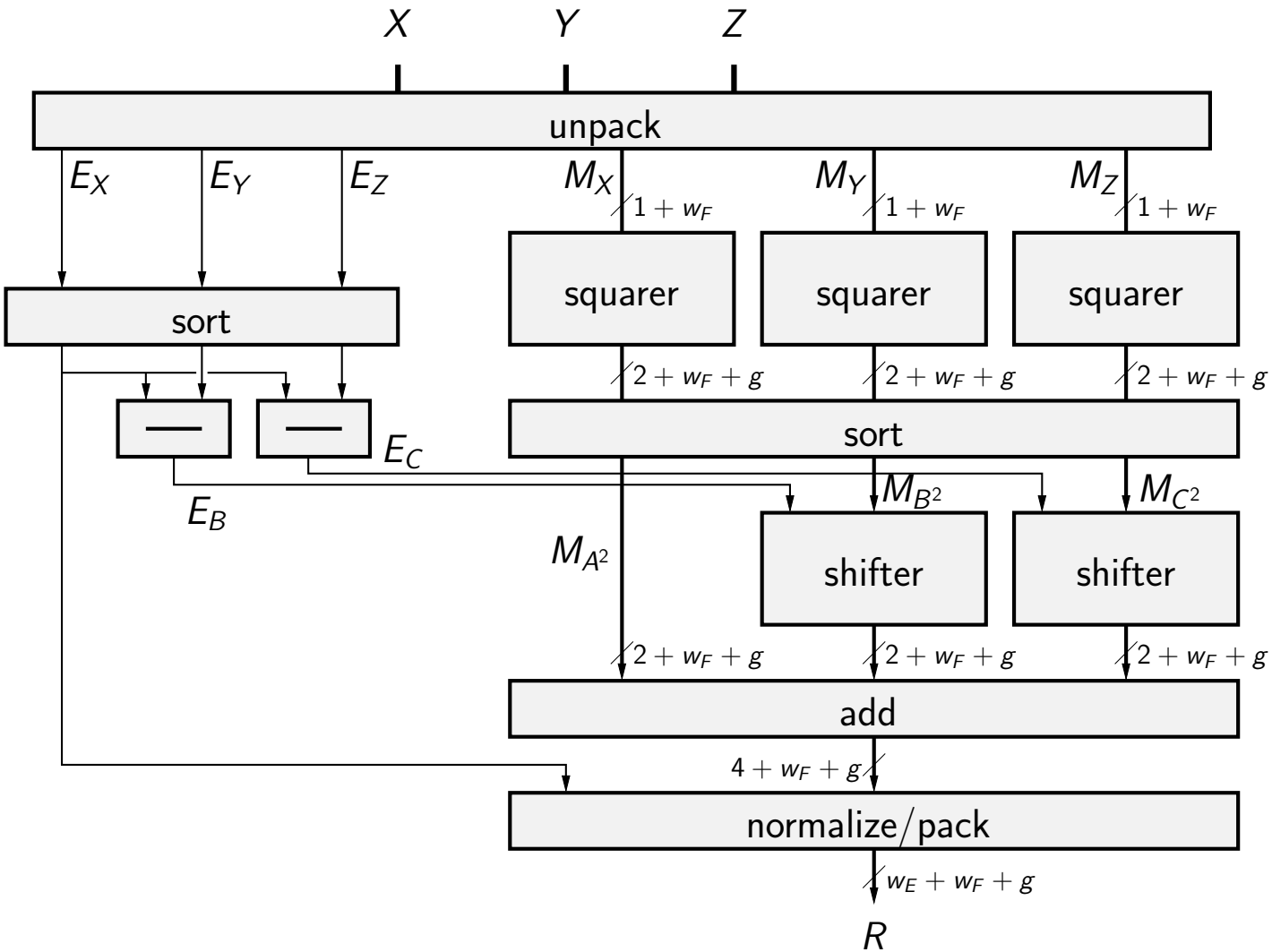
## Operator fusion

- provide the floating-point interface
- optimize a fixed-point architecture
- ensure a clear accuracy specification

# A floating-point adder



# A floating-point sum-of-product architecture



# Savings

A few (old) results for floating-point sum-of-squares on Virtex4:
(*classic:* assembly of classical FP adders and multipliers,
*custom:* the architecture on previous slide)

| 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

# Second fusion example: the floating-point exponential

## Everybody knows FPGAs are bad at floating-point

- Versus the highly optimized FPU in a processor,
- basic operations $(+, -, \times)$ are **10x slower** in an FPGA

**This is the inavoidable overhead of programmability.**

# Second fusion example: the floating-point exponential

## Everybody knows FPGAs are bad at floating-point

- Versus the highly optimized FPU in a processor,
- basic operations $(+, -, \times)$ are **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

- Software in a PC: 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 even better

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

# Not all FLOPS are equal

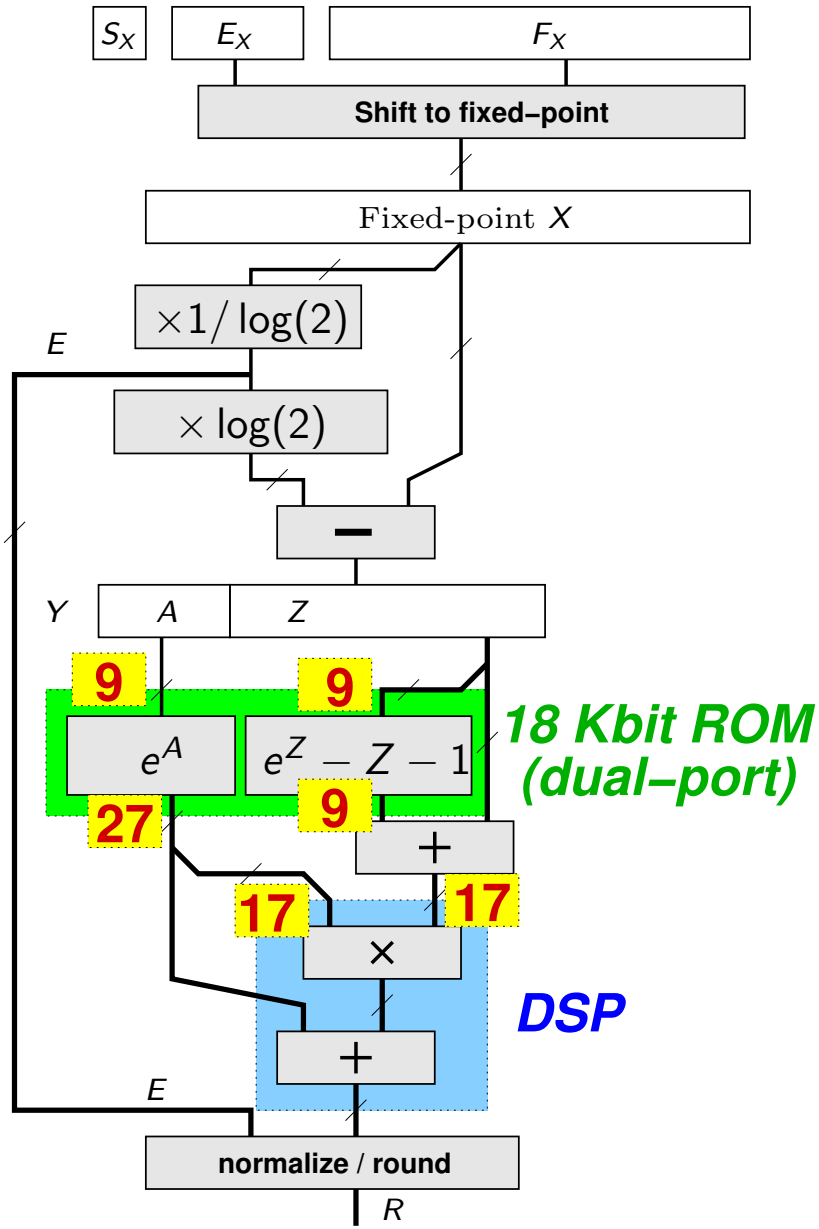
## SPICE Model-Evaluation, cut from Kapre and DeHon (FPL 2009)

**Table 2**. Verilog-AMS Compiler Output

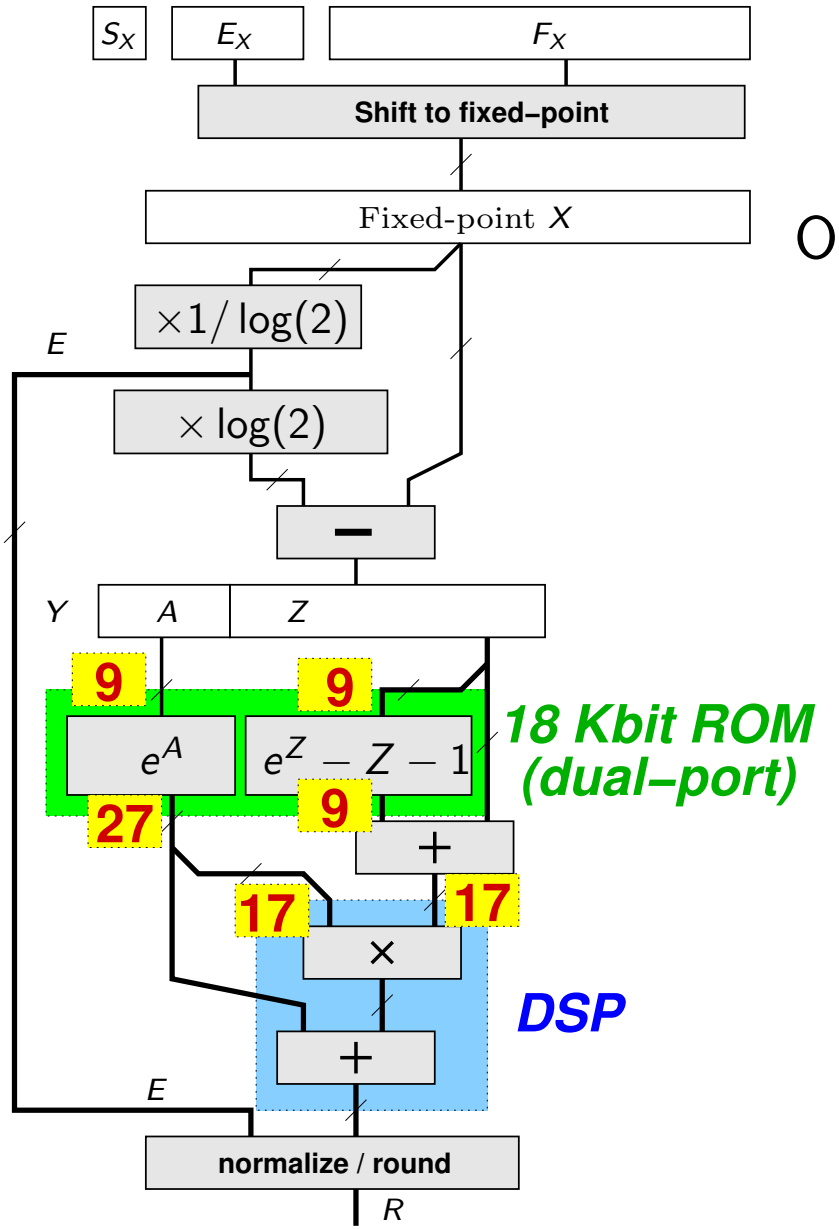
| Models | Instruction Distribution | | | | | |
|---|---|---|---|---|---|---|
| | **Add** | **Mult.** | **Div.** | **Sqrt.** | **Exp.** | **Log** |
| `bjt` | 22 | 30 | 17 | 0 | 2 | 0 |
| `diode` | 7 | 5 | 4 | 0 | 1 | 2 |
| `hbt` | 112 | 57 | 51 | 0 | 23 | 18 |
| `jfet` | 13 | 31 | 2 | 0 | 2 | 0 |
| `mos1`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

# Tabulation of pre-computed values

# We have seen it already



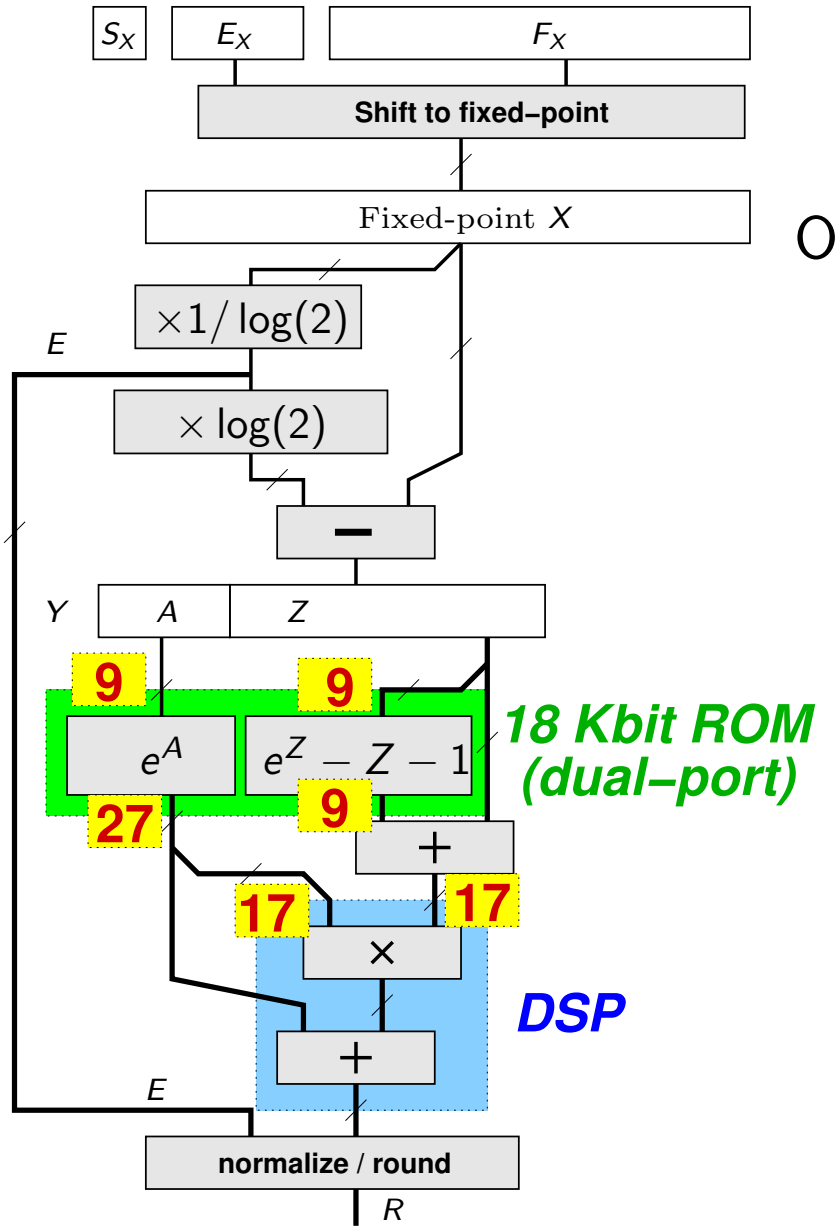
# We have seen it already




Other examples:

- The KCM constant multiplication technique

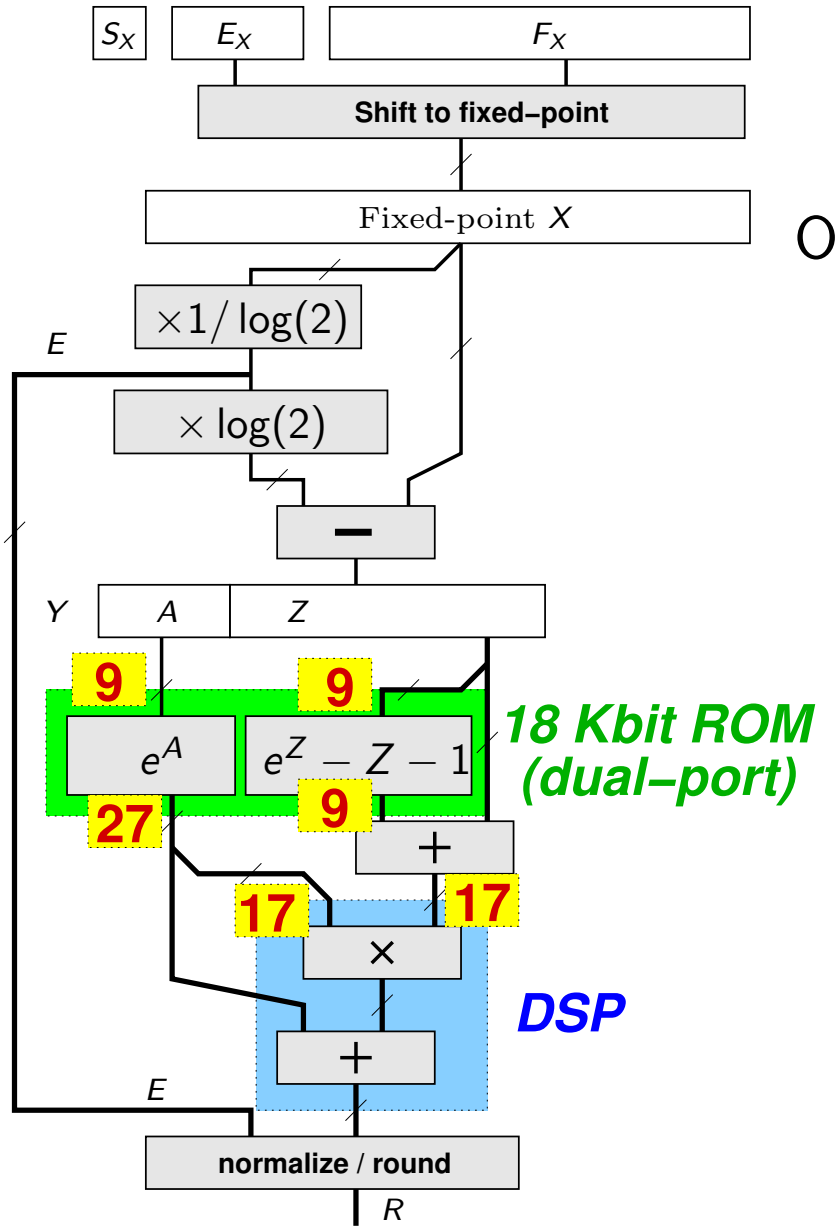
# We have seen it already




Other examples:

- The KCM constant multiplication technique
- The state of the art division by 3

# We have seen it already




Other examples:

- The KCM constant multiplication technique
- The state of the art division by 3
- Computing $A \times B \mod N$ as

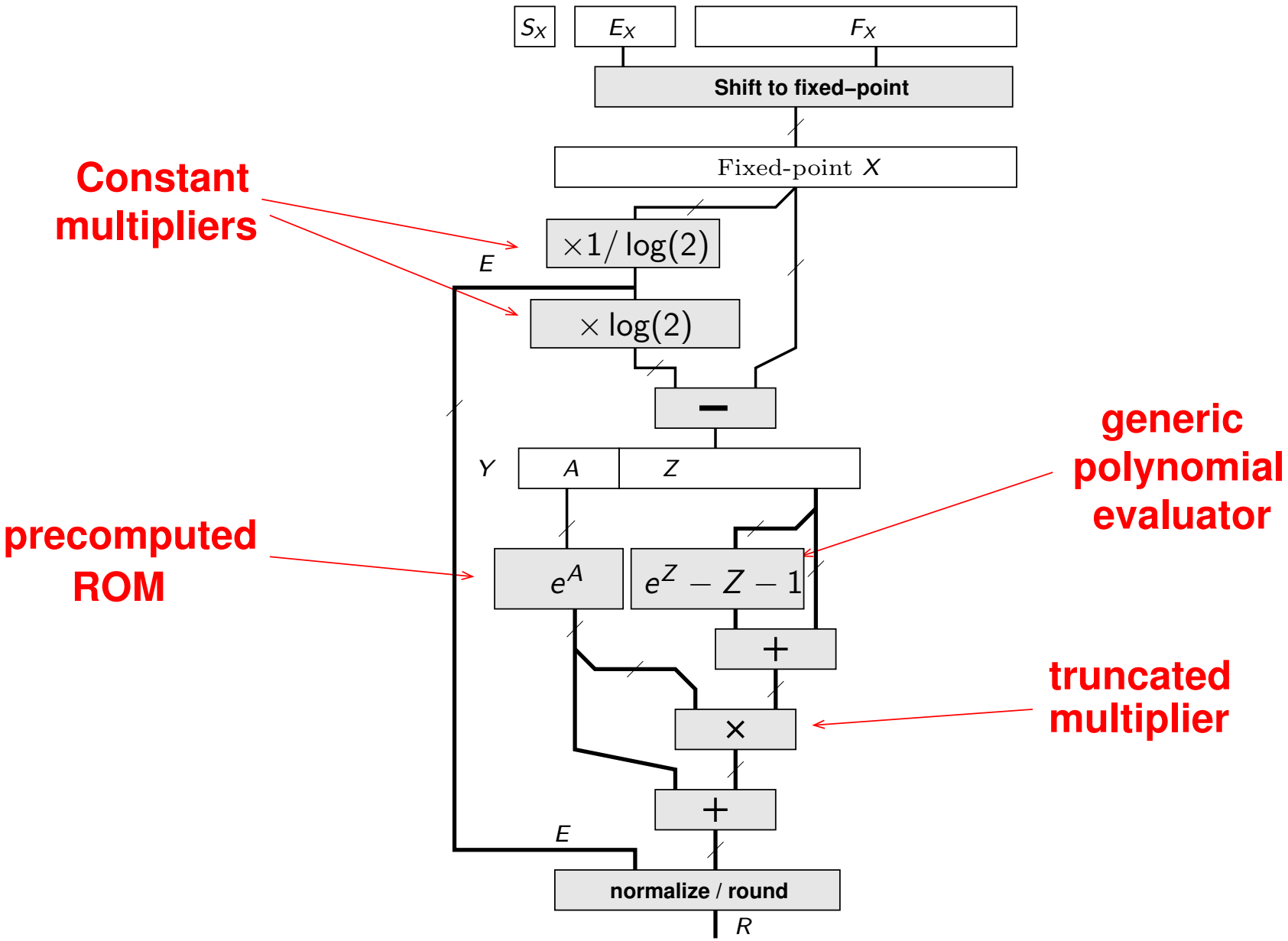
$$\frac{1}{4}((A+B)^2 - (A-B)^2 \mod N$$

where $X^2 \mod N$ is tabulated

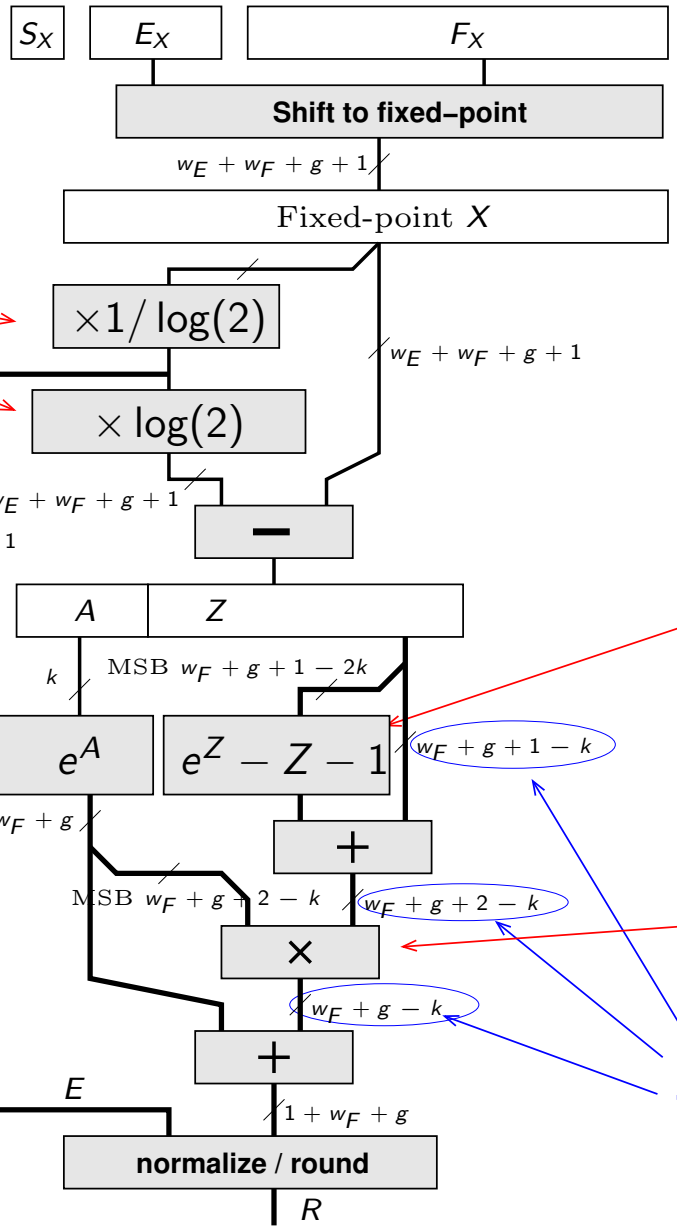
- ...

# Conclusion: the FloPoCo project

# Summing up: not your PC's exponential



# Summing up: not your PC's exponential




**Constant multipliers**

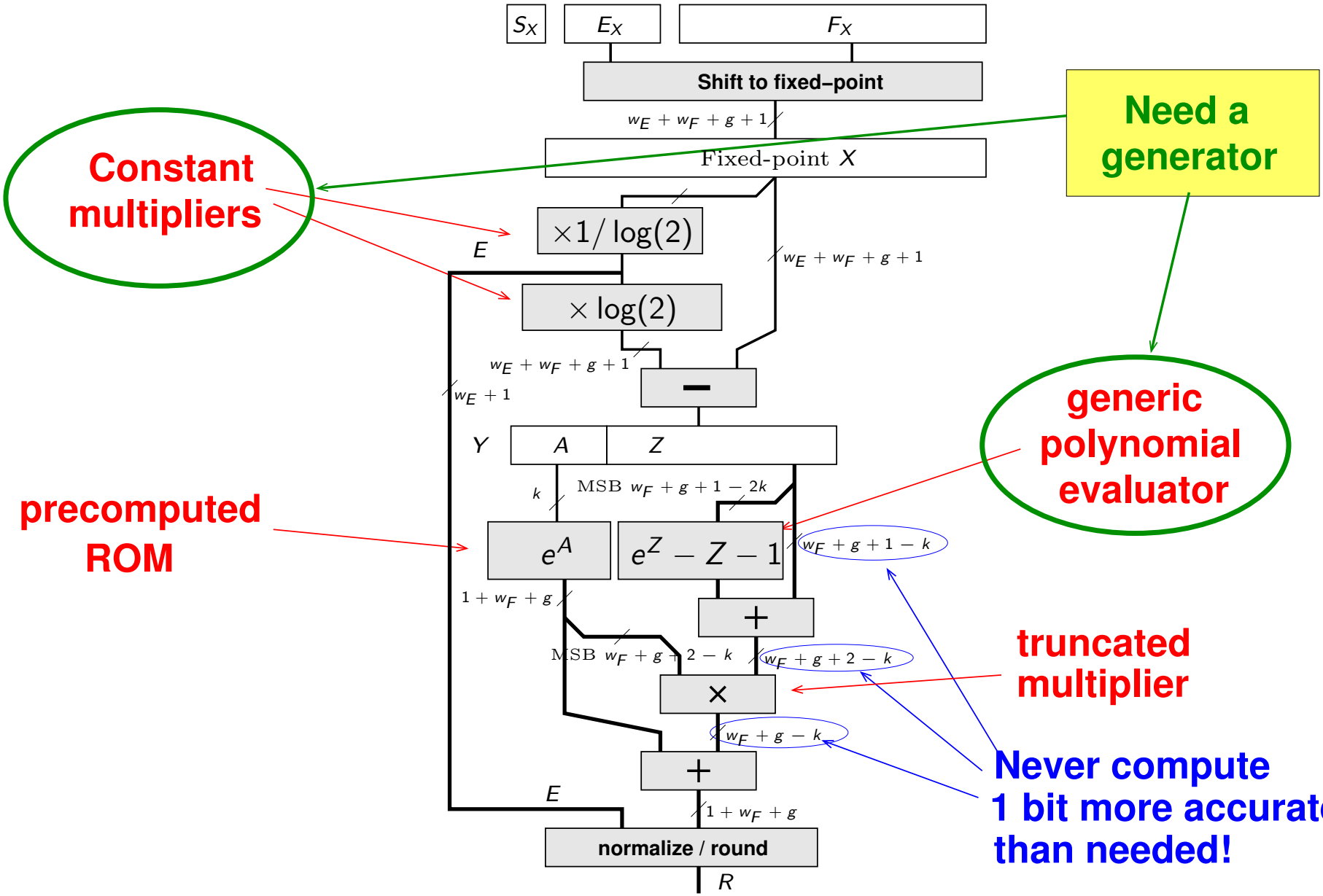
**precomputed ROM**

**generic polynomial evaluator**

**truncated multiplier**

**Never compute 1 bit more accurately than needed!**

# Summing up: not your PC's exponential



# Hey, but I am a physicist !

... I don't want to design all these fancy operators !

# Hey, but I am a physicist !

... I don't want to design all these fancy operators !

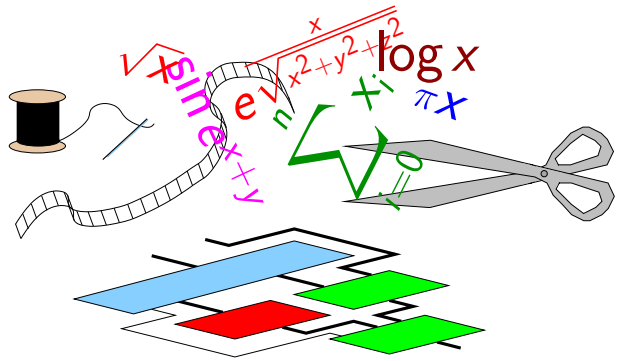
**You don't have to, it is my job**

And it is a very comfortable niche

- An infinite list of operators to keep me busy until retirement
- small arithmetic objects, relatively technology-independent

# The FloPoCo project

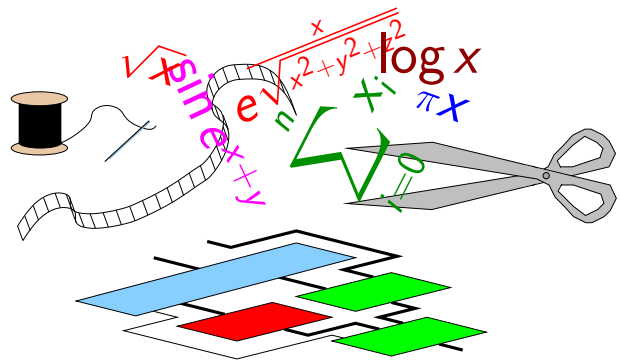
`http://flopoco.gforge.inria.fr/`




- A generator framework
  - written in C++, outputting VHDL
  - open and extensible

# The FloPoCo project

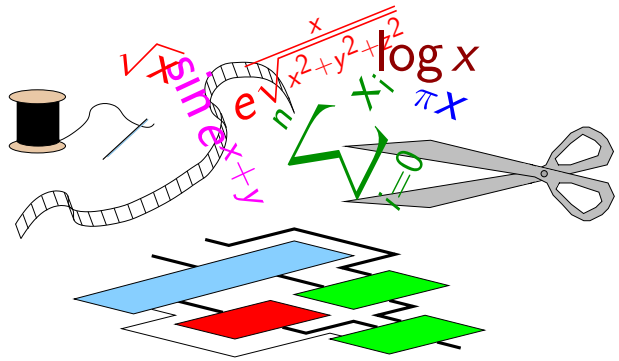
`http://flopoco.gforge.inria.fr/`




- A generator framework
  - written in C++, outputting VHDL
  - open and extensible
- Goal: provide all the application-specific arithmetic operators you want (even if you don't know yet that you want them)
  - open-ended list, about 50 in the stable version, and a few others in "obscure branches"
  - integer, fixed-point, floating-point, logarithm number system
  - all operators fully parameterized
  - flexible pipeline for all operators

# The FloPoCo project

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




- A generator framework
  - written in C++, outputting VHDL
  - open and extensible
- Goal: provide all the application-specific arithmetic operators you want (even if you don't know yet that you want them)
  - open-ended list, about 50 in the stable version, and a few others in "obscure branches"
  - integer, fixed-point, floating-point, logarithm number system
  - all operators fully parameterized
  - flexible pipeline for all operators
- Approach: **computing just right**
  - Interface: never output bits that are not numerically meaningful
  - Inside: never compute bits that are not useful to the final result

# Where do we stop?

## My own personal definition of an arithmetic operator

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect
    - (even *filters* are defined by a transfer function)

# Where do we stop?

## My own personal definition of an arithmetic operator

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect
    - (even *filters* are defined by a transfer function)
- An operator is the *implementation* of such a function
  - ... mathematically specified in terms of a rounding function
  - e.g. IEEE-754 FP standard: operator(x) = rounding(operation(x))

$\rightarrow$ Clean mathematic definition, even for floating-point arithmetic

# Where do we stop?

## My own personal definition of an arithmetic operator

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect
    - (even *filters* are defined by a transfer function)
- An operator is the *implementation* of such a function
  - ... mathematically specified in terms of a rounding function
  - e.g. IEEE-754 FP standard: operator(x) = rounding(operation(x))

→ Clean mathematic definition, even for floating-point arithmetic

## An operator, as a *circuit*...

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

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

# One small problem

FloPoCo can generate an infinite number of operators, I don't want to test them all...

# One small problem

FloPoCo can generate an infinite number of operators, I don't want to test them all...

## Solution

Each operator comes with its testbench generator

- expected outputs built from the mathematical specification,
- **not** by emulating the operator architecture!

# Here should come a demo

- Command line syntax: a sequence of operator specifications
- Options: target frequency, target hardware, ...
- Output: synthesizable VHDL.

FloPoCo is open-source and freely available from

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