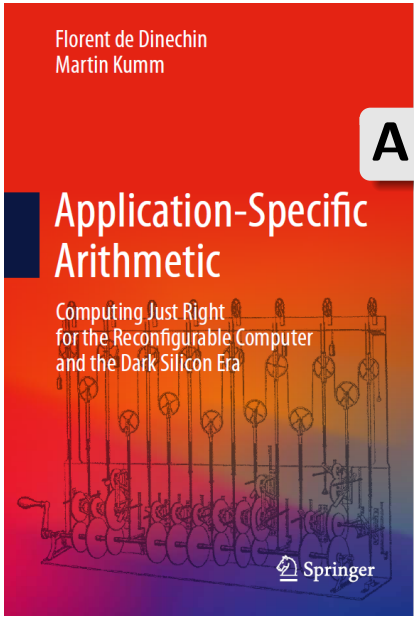


ARITH 2025

Application-Specific Arithmetic

Florent de Dinechin
Martin Kumm

Anti-introduction: traditional arithmetic
Opportunities of application-specific arithmetic
Conclusion

citi lab — INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON — Inria informatique mathématique — UNIVERSITÉ DE LYON

Hochschule Fulda
University of Applied Sciences

Anti-introduction: traditional arithmetic

General-Purpose Processors and GPUs

The good arithmetic in a general-purpose processor is the most generally useful: **additions**, **multiplications**, and then?

- *Should a processor include a* **divider***? A* **square root***?*

- *Should a processor include* **elementary functions** *(exp, log, sine/cosine)? Which?*
- *Should a processor include decimal hardware?*
- *Should a processor include a multiplier modulo 3329?*
- *Should a processor include an 8-bit tensor multiplier?*
- ...

Example: Should a processor include a divider?

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

... and it should be **fast**:

Dura Amdahl lex, sed lex

Although division is not frequent, a high-latency divider can ruin your average performance

Example: Should a processor include a divider?

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

... and it should be **fast**:

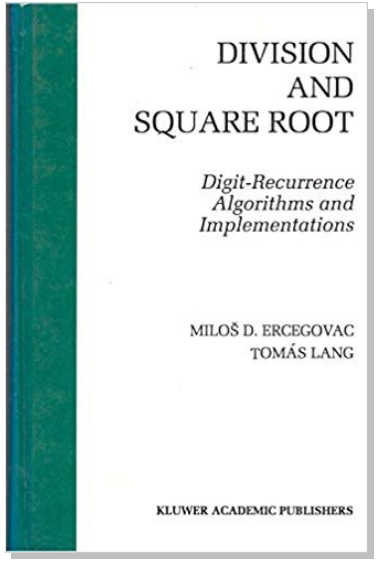
Dura Amdahl lex, sed lex

Although division is not frequent, a high-latency divider can ruin your average performance

A lot of ARITH research on division

- digit recurrence algorithms

(worth one full book)



Example: Should a processor include a divider?

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

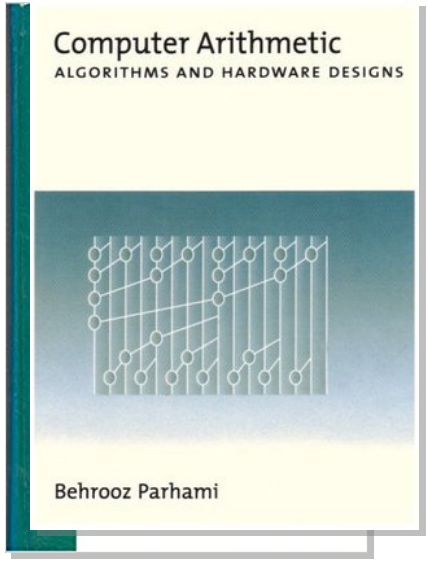
... and it should be **fast**:

Dura Amdahl lex, sed lex

Although division is not frequent, a high-latency divider can ruin your average performance

A lot of ARITH research on division

- digit recurrence algorithms (worth one full book)
- multiplicative algorithms (a chapter in each of the standard textbooks)



Example: Should a processor include a divider?

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

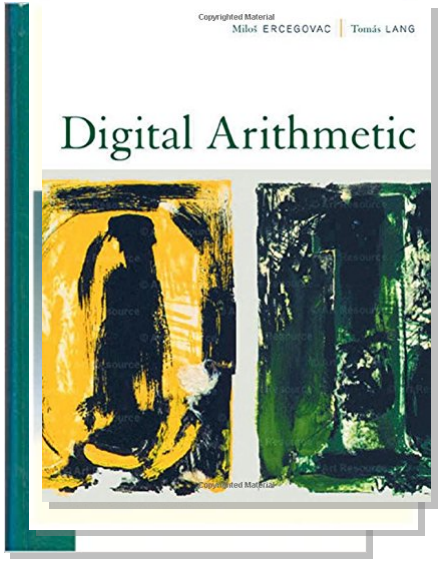
... and it should be **fast**:

Dura Amdahl lex, sed lex

Although division is not frequent, a high-latency divider can ruin your average performance

A lot of ARITH research on division

- digit recurrence algorithms
  (worth one full book)
- multiplicative algorithms
  (a chapter in each of the standard textbooks)



Example: Should a processor include a divider?

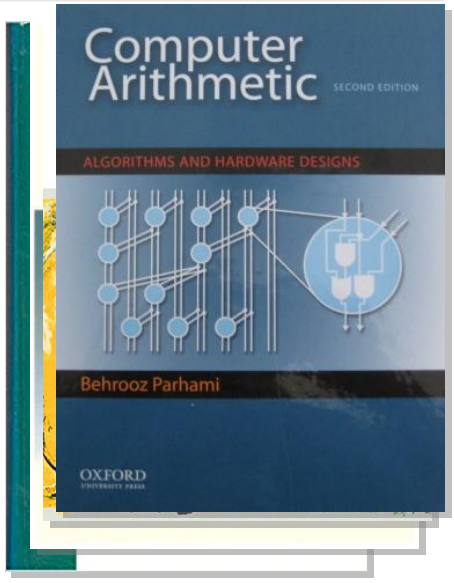
Answer in 1993 is : **YES** (Oberman & Flynn, 1993)
... and it should be **fast**:

Dura Amdahl lex, sed lex

Although division is not frequent, a high-latency divider can ruin your average performance

A lot of ARITH research on division

- digit recurrence algorithms
  (worth one full book)
- multiplicative algorithms
  (a chapter in each of the standard textbooks)



Should a processor include a divider? (2)

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

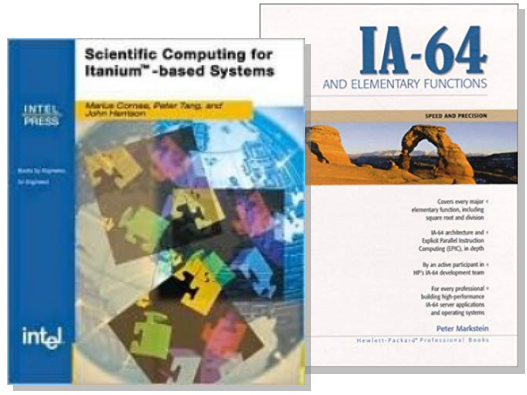
Instead of a hardware divider,

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

BLAS, FFTs, etc. 2x faster...

Two FMAs enable efficient divisions in software

- several algorithms to choose from
  - Newton-Raphson
  - Goldschimdt
  - Quadratic series expansion
  - ...
- the freedom of software:
  - quick and dirty, or accurate but slow
  - high throughput or short latency
  - ...
- and two more books



Should a processor include a divider? (3)

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

Should a processor include a divider? (3)

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

- a double-precision divider in 11 cycles for ARM processors
- thanks to a **totally wasteful** implementation
  - hardware: **20** fast 58-bit adders, **12** 58-bit muxes, tables, and more ...
  - hardware speculation all over the place:
    compute many options in parallel, then discard them all except one
- in a processor that is supposed to go in your smartphone?!?

Should a processor include a divider? (3)

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

- a double-precision divider in 11 cycles for ARM processors
- thanks to a **totally wasteful** implementation
  - hardware: **20** fast 58-bit adders, **12** 58-bit muxes, tables, and more ...
  - hardware speculation all over the place:
    compute many options in parallel, then discard them all except one
- in a processor that is supposed to go in your smartphone?!?

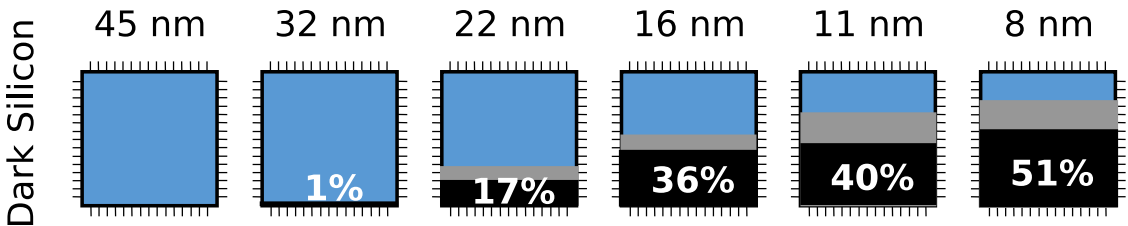
We do this to reduce overal energy consumption!
There is this huge superscalar ARM core that consumes a lot,
we save energy if we can switch it off a few cycles earlier

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.

We just can't dissipate the heat, and it gets worse with Moore's Law.
"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 (i.e. most of the time), serve as radiator for the parts in use

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 (i.e. most of the time), serve as radiator for the parts in use

Should a processor include elementary functions? (1)

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

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

Should a processor include elementary functions? (1)

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

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

Dura Amdahl lex, sed lex

- add and mult: 2 to 5 cycles
- exp or log: 10 to 100 cycles

Here the processor spends most of its time computing elementary functions

Should a processor include elementary functions? (2)

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

- implemented in microcode
- with some hardware assistance, in particular the 80-bit floating-point 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 2025 is... **sometimes**?

Should a processor include elementary functions? (4)

Answer in 2025 is... **sometimes**?

- 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)
- These days, countless machine learning accelerators include ad-hoc exponential hardware for SoftMax

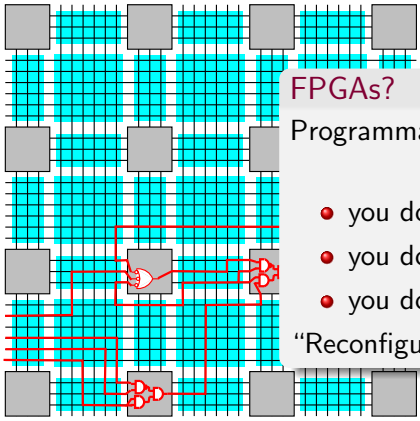
I won't answer the other questions here

- ✓ *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 multiplier modulo 3329?*
- *Should a processor include an 8-bit tensor multiplier?*
- ...

I won't answer the other questions here

- ✓ *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 multiplier modulo 3329?*
- *Should a processor include an 8-bit tensor multiplier?*
- ...
- *Should a processor include an instruction to divide a floating-point number by 3?*
  at least this one is clear: **no**, of course.

Enters the Field-Programmable Gate Arrays



FPGAs?

Programmable chips,
but the programming model is the **digital circuit**

- you don't develop programs, you design circuits;
- you don't compile, you synthesize;
- you don't load a program, you configure an FPGA.

"Reconfigurable computing" means "computing with FPGAs"

One nice things with FPGAs

There is a simpler answer to all these questions:

Should an application running on an FPGA include a circuit for

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

One nice things with FPGAs

There is a simpler answer to all these questions:

Should an application running on an FPGA include a circuit for

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

...

In reconfigurable computing, useful means: useful to one application.

Application-specific arithmetic

All sorts of useful arithmetic operators that just **wouldn't make sense** in a processor...

... and therefore didn't yet have a book dedicated to them.

There is a simpler answer to

Should an application runni
- ✓ *division? square root?*
- ✓ *elementary functions?*
- ✓ *FFT operator?*
- ✓ *multiplier by* log(2)*? B*

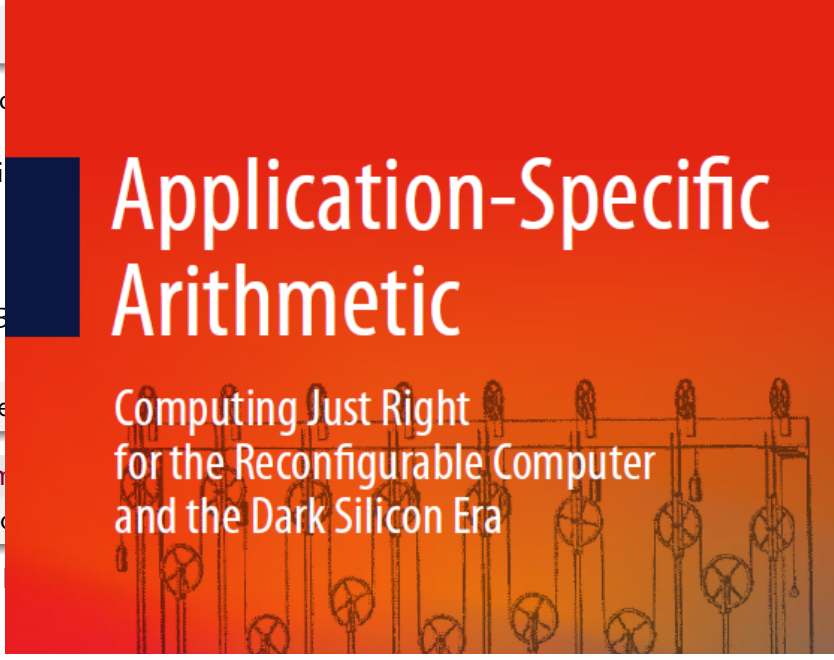
...

In reconfigurable

Application-specific arithm

All sorts of useful arithmeti

... and therefore didn't yet



Conclusion so far

Application-specific arithmetic $\supseteq$ arithmetic for CPUs or GPGPUs

Conclusion so far

Application-specific arithmetic $\supseteq$ arithmetic for CPUs or GPGPUs

This is a **qualitative** question, but there is a related **quantitative** question:

How many bits?

In a processor, data is 8, 16, 32 or 64 bits (at best).
In an FPGA, data formats may be tightly fitted to the requirements of the application:

if you need 17 bits, compute only 17 bits

Compute as few bits as possible, but compute them correctly

- If the lower bits carry useless noise, you don't want to compute them...
- ... and you want even less to store them, transmit them, compute on them.

Conclusion so far

Application-specific arithmetic $\supseteq$ arithmetic for CPUs or GPGPUs

This is a **qualitative** question, but there is a related **quantitative** question:

How many bits?

In a processor, data is 8, 16, 32 or 64 bits (at best).
In an FPGA, data formats may be tightly fitted to the requirements of the application:

if you need 17 bits, compute only 17 bits

Compute as few bits as possible, but compute them correctly

- If the lower bits carry useless noise, you don't want to compute them...
- ... and you want even less to store them, transmit them, compute on them.

Computing just right

Applicat

This is a **qualitative**

How many bits?

In a processor, data i
In an FPGA, data for

Compute as few b

- If the lower bits
- ... and you want

Enough advertising for the book

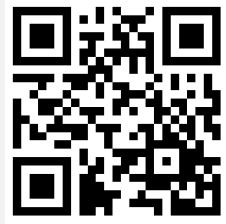
Application-specific arithmetic is also the subject of the **FloPoCo** software project

http://flopoco.org/

- Open source C++
- Input operator specifications, outputs synthesizable VHDL
- Generates an infinity of operators
  - ... (and their test bench, because we couldn't test them all)
- Always with clean (IEEE754-inspired) specifications:
  - An arithmetic operation is a *function* (in the mathematical sense)
  - An operator is the *implementation* of such a function:

$$\boxed{\text{operator}(x) = \text{rounding}(\text{operation}(x))}$$

  - so the *precision* of the output format defines the *accuracy* of the operator



Any mathematical function is of interest! We are busy until retirement.

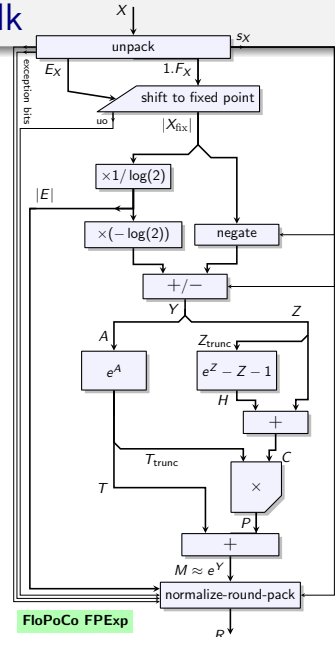
Let us introduce the running example of this talk

What do P.T.P. Tang, Ch. Lauter, and G. Melquiond have in common?

Let us introduce the running example of this talk

What do P.T.P. Tang, Ch. Lauter, and G. Melquiond have in common?

They have worked on the floating-point exponential (here a *hardware* variant of Tang's algorithm) $\longrightarrow$



FloPoCo FPExp

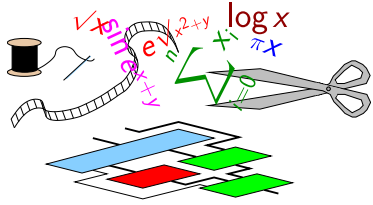
Let us introduce the running example of this talk

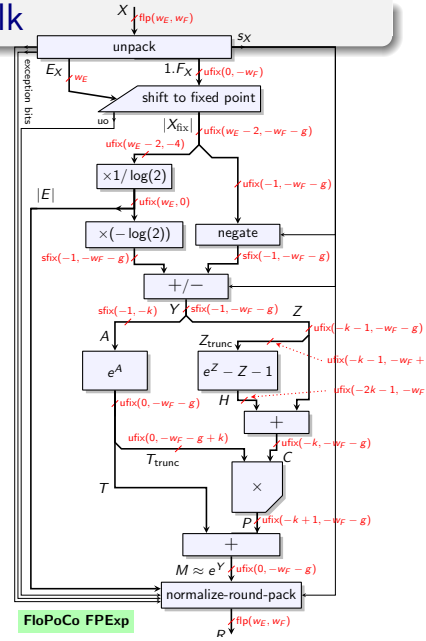
What do P.T.P. Tang, Ch. Lauter, and G. Melquiond have in common?

They have worked on the floating-point exponential (here a *hardware* variant of Tang's algorithm) ⟶

Computing just right

Each wire, each component is **tailored to its context** with love and care.





Conclusion of the introduction

Arithmetic in software versus hardware

- In a processor, **constraint:** data is 8, 16, 32 or 64 bits (at best).
- In a circuit, **freedom:** we may choose, for each variable, how many bits are computed/stored/transmitted! $\longrightarrow$ **the opportunities**

Overwhelming freedom! Help! $\longrightarrow$ **the challenges**

Opportunities of application-specific arithmetic

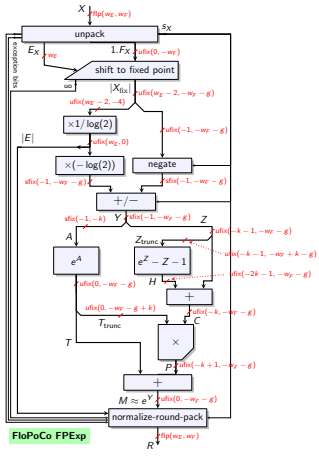
Opportunities of application-specific arithmetic

1. Operator parameterization
2. Operator specialization
3. Resource sharing
4. Operator fusion
5. Target-specific optimizations
6. Function evaluation

Opportunities of application-specific arithmetic

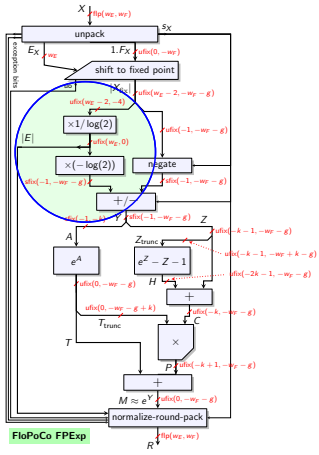
1. **Operator parameterization**
2. Operator specialization
3. Resource sharing
4. Operator fusion
5. Target-specific optimizations
6. Function evaluation

Opportunity #1: Operator (over-)parameterization



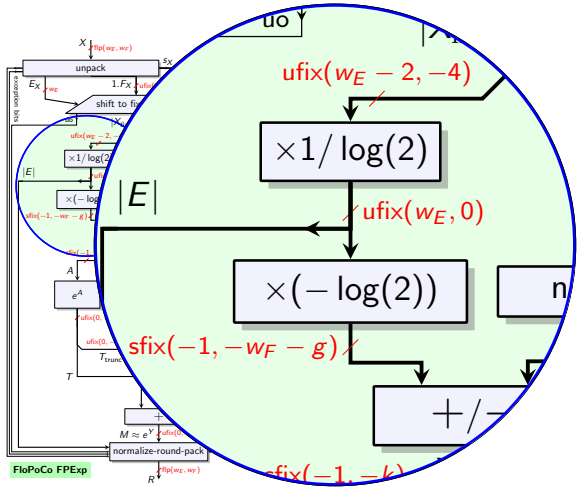
Example:

Opportunity #1: Operator (over-)parameterization



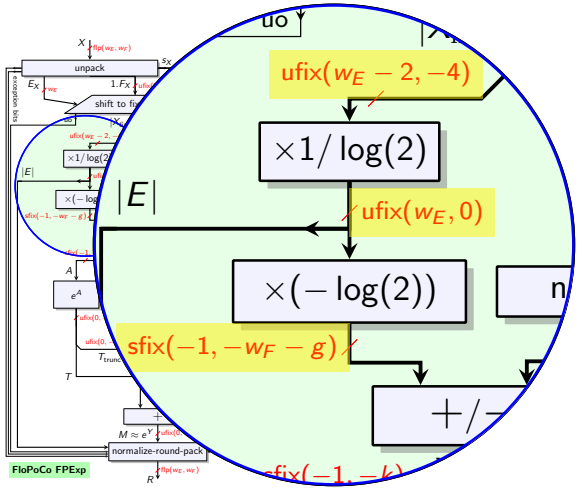
Example:

Opportunity #1: Operator (over-)parameterization



Example:

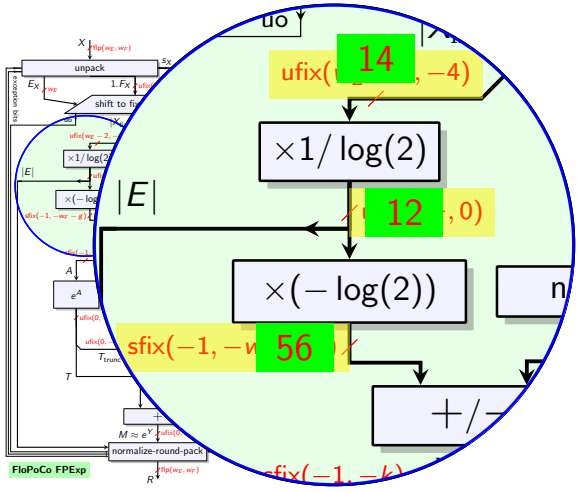
Opportunity #1: Operator (over-)parameterization



Example:

Multipliers of all shapes and sizes

Opportunity #1: Operator (over-)parameterization



Example:

Multipliers of all shapes and sizes

In a double-precision exponential,

- $w_E = 11$, $w_F = 52$,
- first multiplier 14-bits in, 12 bits out
- second multiplier 12-bits in, 56 bits out

... and truncated left and right

Over-parameterization is a Good Thing

- OK, there is a bit more work involved in designing a parametric operator
  - To start with, it must be a hardware-generating program

Over-parameterization is a Good Thing

- ⊖ OK, there is a bit more work involved in designing a parametric operator
  - To start with, it must be a hardware-generating program
- ⊕ Direct benefit to end-users: freedom of choice

Over-parameterization is a Good Thing

- ⊖ OK, there is a bit more work involved in designing a parametric operator
  - To start with, it must be a hardware-generating program
- ⊕ Direct benefit to end-users: freedom of choice
- ⊕ Easy to retarget, future-proof, etc.

Over-parameterization is a Good Thing

⊖ OK, there is a bit more work involved in designing a parametric operator
- To start with, it must be a hardware-generating program

⊕ Direct benefit to end-users: freedom of choice

⊕ Easy to retarget, future-proof, etc.

⊕ It actually simplifies design of composite operators (e.g. the exponential)
- No need to take any dramatic decision in the design phase:
  You don't know how many bits on this wire make sense? Keep it open as a parameter.
- Then estimate cost and accuracy as a function of the parameters
- Then find the optimal values of the parameters,
  e.g. using common sense or ILP (whichever gives the best results)

Opportunities of application-specific arithmetic

1. Operator parameterization
2. **Operator specialization**
3. Resource sharing
4. Operator fusion
5. Target-specific optimizations
6. Function evaluation

Opportunity #2: Operator specialization

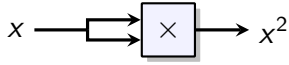
- Division by 3 (for various values of 3)
  - correctly rounded floating-point division by 3 and 9 (Jacobi, etc)
  - round-robin addressing with 3 banks of memory (need quotient and remainder)
  - ...

Opportunity #2: Operator specialization

- Division by 3 (for various values of 3)
  - correctly rounded floating-point division by 3 and 9 (Jacobi, etc)
  - round-robin addressing with 3 banks of memory (need quotient and remainder)
  - ...
- Multiplications by constants
  - Integer constants, or reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - Relevant in digital filters, linear transforms (like FFTs), etc.

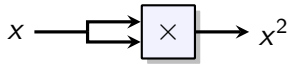
Opportunity #2: Operator specialization

- Division by 3 (for various values of 3)
  - correctly rounded floating-point division by 3 and 9 (Jacobi, etc)
  - round-robin addressing with 3 banks of memory (need quotient and remainder)
  - ...
- Multiplications by constants
  - Integer constants, or reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - Relevant in digital filters, linear transforms (like FFTs), etc.
- A squarer is a multiplier specialization



Opportunity #2: Operator specialization

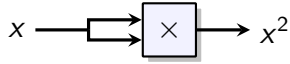
- Division by 3 (for various values of 3)
  - correctly rounded floating-point division by 3 and 9 (Jacobi, etc)
  - round-robin addressing with 3 banks of memory (need quotient and remainder)
  - ...
- Multiplications by constants
  - Integer constants, or reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - Relevant in digital filters, linear transforms (like FFTs), etc.
- A squarer is a multiplier specialization



- FP adder for positive numbers only (it saves cancellation management)

Opportunity #2: Operator specialization

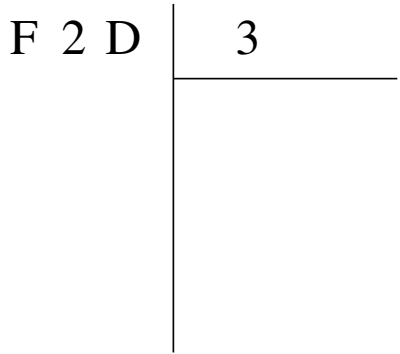
- Division by 3 (for various values of 3)
  - correctly rounded floating-point division by 3 and 9 (Jacobi, etc)
  - round-robin addressing with 3 banks of memory (need quotient and remainder)
  - ...
- Multiplications by constants
  - Integer constants, or reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - Relevant in digital filters, linear transforms (like FFTs), etc.
- A squarer is a multiplier specialization



- FP adder for positive numbers only (it saves cancellation management)
- Specialization of elementary functions to specific domains
  - Stable SoftMax needs e^x for $x < 0$ only
  - ...
- ...

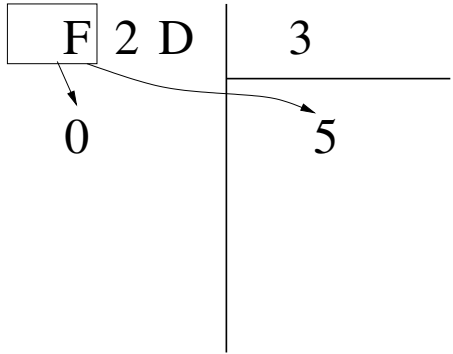
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



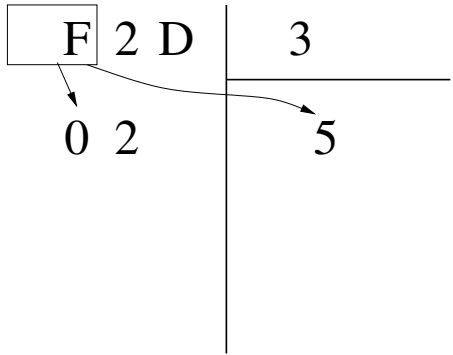
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



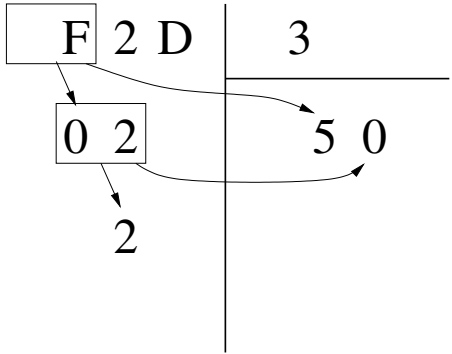
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



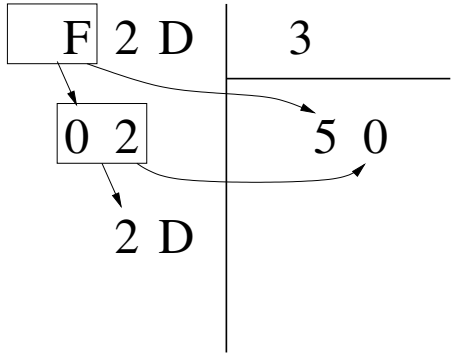
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



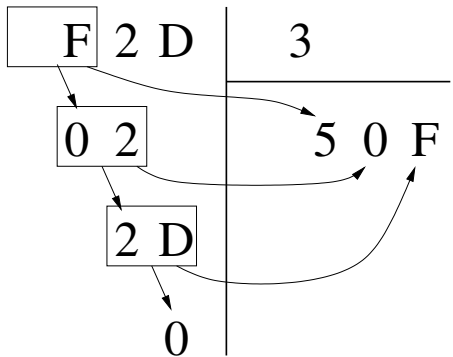
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3

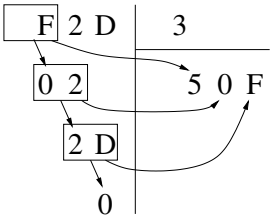


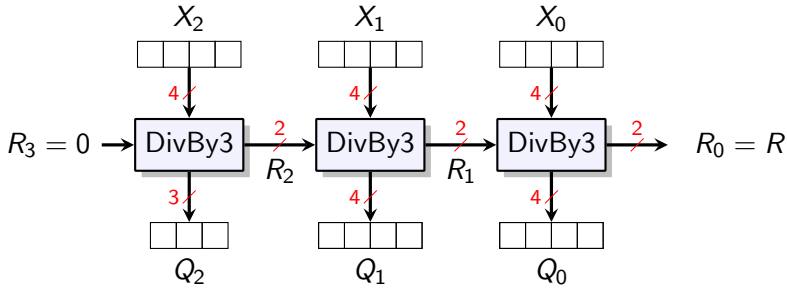
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3

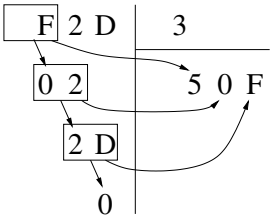


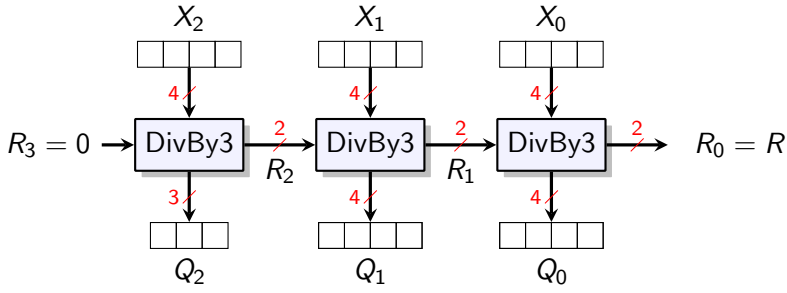
Division by 3 should not be more complex than multiplication by 3





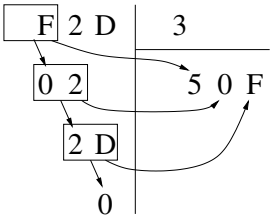
Division by 3 should not be more complex than multiplication by 3

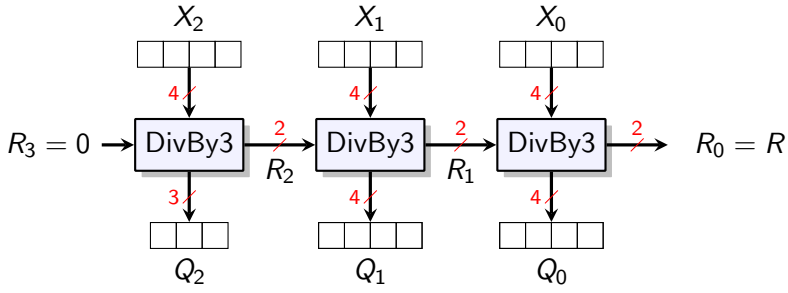




OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

Division by 3 should not be more complex than multiplication by 3



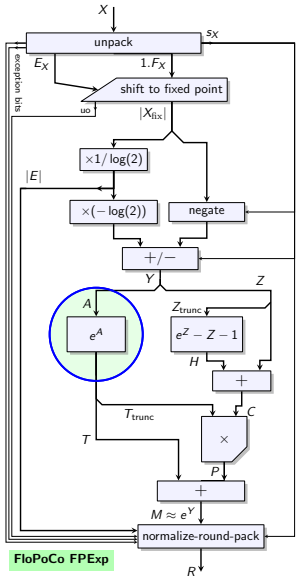


OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

If you don't know how to compute it, then tabulate it

... here a table of 2^6 entries of 6 bits each.
(small enough to be called a truth table and submitted to synthesis tools)

Tabulation, the Swiss army knife of the lazy and the stupid

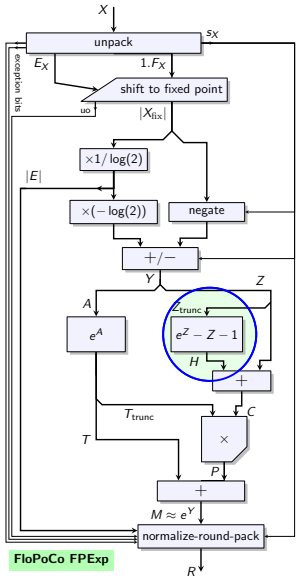


FloPoCo FPExp

Being unable to trust my reasoning, I learnt by heart
the results of all the possible multiplications
(E. Ionesco)

- ... and all the possible exponentials

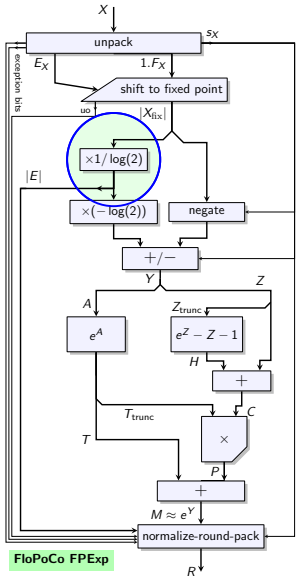
Tabulation, the Swiss army knife of the lazy and the stupid



Being unable to trust my reasoning, I learnt by heart
the results of all the possible multiplications
(E. Ionesco)

- ... and all the possible exponentials
- ... and all the possible values of $e^Z - Z - 1$

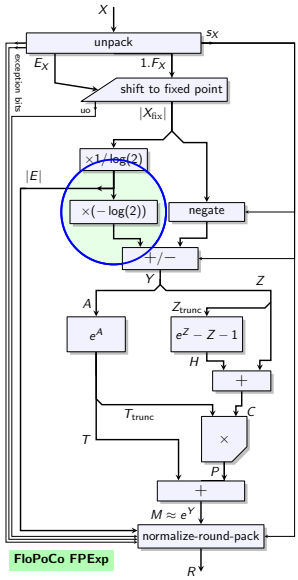
Tabulation, the Swiss army knife of the lazy and the stupid



Being unable to trust my reasoning, I learnt by heart
the results of all the possible multiplications
(E. Ionesco)

- ... and all the possible exponentials
- ... and all the possible values of $e^Z - Z - 1$
- ... and indeed, all the possible multiplications

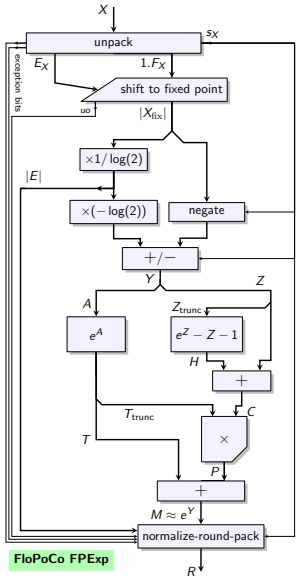
Tabulation, the Swiss army knife of the lazy and the stupid



Being unable to trust my reasoning, I learnt by heart
the results of all the possible multiplications
(E. Ionesco)

- ... and all the possible exponentials
- ... and all the possible values of $e^Z - Z - 1$
- ... and indeed, all the possible multiplications

Tabulation, the Swiss army knife of the lazy and the stupid



Being unable to trust my reasoning, I learnt by heart
the results of all the possible multiplications
(E. Ionesco)

- ... and all the possible exponentials
- ... and all the possible values of $e^Z - Z - 1$
- ... and indeed, all the possible multiplications

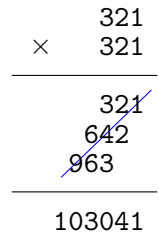
Reading a tabulated value is very efficient
when the table is close to the consumer.

Opportunities of application-specific arithmetic

1. Operator parameterization
2. Operator specialization
3. **Resource sharing**
4. Operator fusion
5. Target-specific optimizations
6. Function evaluation

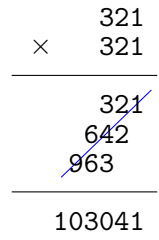
Opportunity #3: Resource sharing

- A squarer is smaller than a multipliers
  because it shares and reuses partial products.



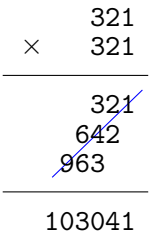
Opportunity #3: Resource sharing

- A squarer is smaller than a multipliers
  because it shares and reuses partial products.
- Karatsuba shares and re-uses intermediate results in large multipliers.



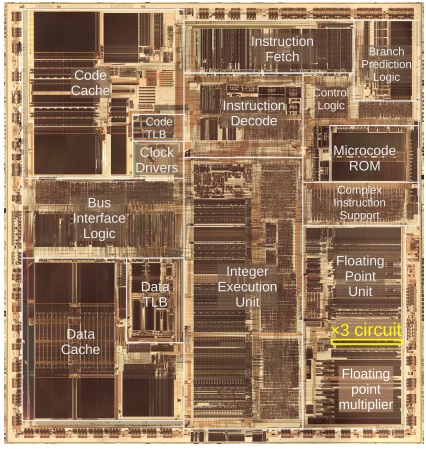
Opportunity #3: Resource sharing

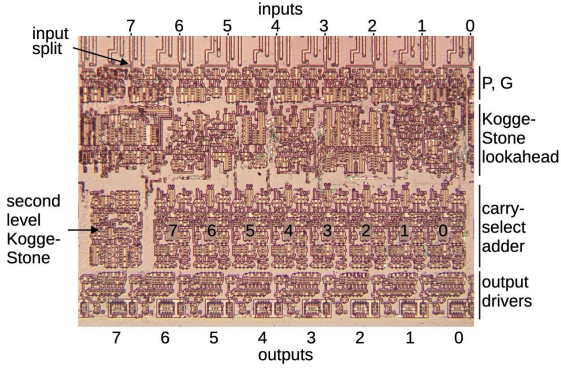
- A squarer is smaller than a multipliers
  because it shares and reuses partial products.
- Karatsuba shares and re-uses intermediate results in large multipliers.
- Multiplication by constant(s) (yes, again)



Single Constant Multiplication

Even the Pentium required a $\times 3$ constant multiplication (for higher radix float mult.)

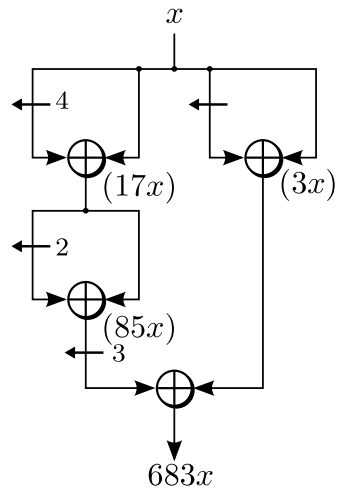




source: `https://www.righto.com/2025/03/pentium-multiplier-adder-reverse-engineered.html`

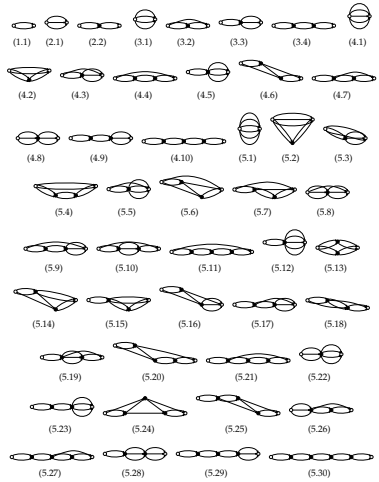
Single Constant Multiplication

- Realizing constant multiplications by additions, subtractions and bit-shifts
- Goal is to find circuit with minimum adders (NP-hard optimization problem)
- Commonly denoted as Single Constant Multiplication (SCM)



Single Constant Multiplication

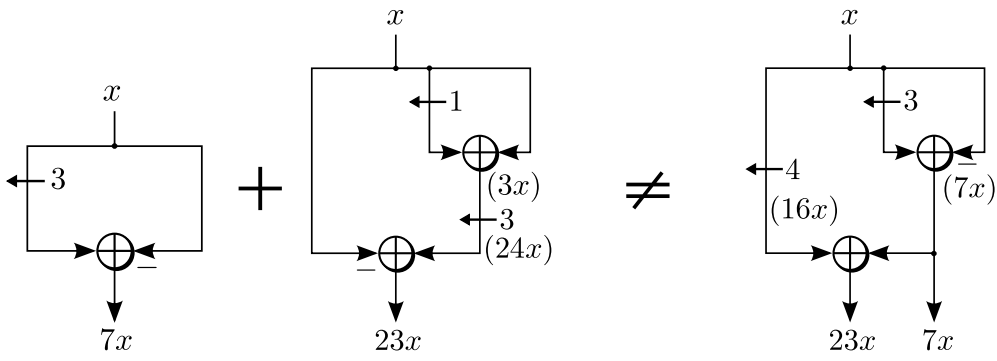
- Gustafsson (2002): Graphs with up to 5 adders cover all constants up to 19 bit
  - by enumeration of all the graphs
- Today: Integer Linear Programming (ILP) and (very recent) a satisfiability (SAT) method that scales well to larger constants (∼32 bits)



All graph topologies up to 5 adders

Multiple Constant Multiplication

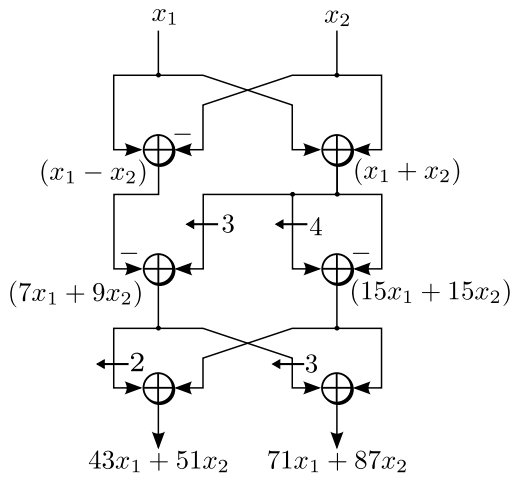
More opportunities when multiplying with several constants (constant vector $\times$ scalar):



Constant Matrix Multiplication

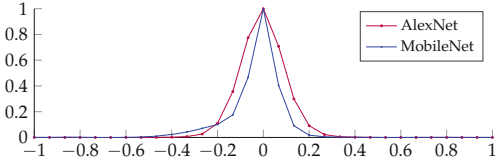
And even more opportunities when multiplying a constant matrix with a vector:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 43 & 51 \\ 71 & 87 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$



Application-specific Multipliers

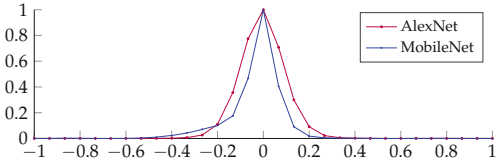
Tune the multiplicands to your application!



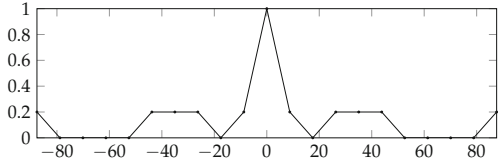
Weight distribution of AlexNet and MobileNet

Application-specific Multipliers

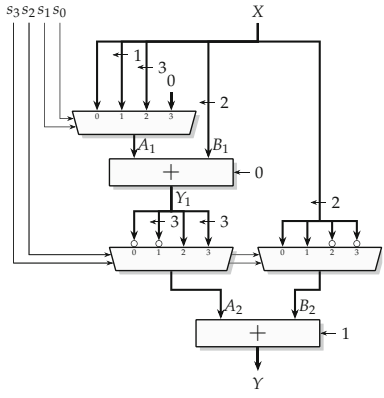
Tune the multiplicands to your application!



Weight distribution of AlexNet and MobileNet



Weight distribution of reconf. multiplier



Reconf. multiplier topology for the coefficients $\pm\{0, 1, 2, 8, 28, 36, 44, 92\}$

Opportunities of application-specific arithmetic

1. Operator parameterization
2. Operator specialization
3. Resource sharing
4. **Operator fusion**
5. Target-specific optimizations
6. Function evaluation

Opportunity #4: Operator fusion

3 examples among many others:

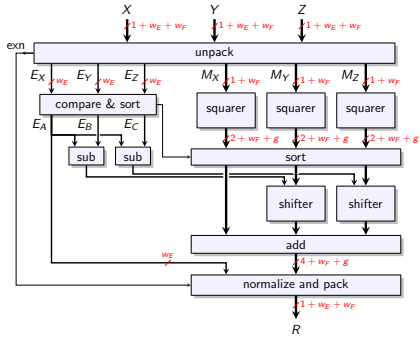
- We just saw multiple constant multiplication.

Opportunity #4: Operator fusion

3 examples among many others:

- We just saw multiple constant multiplication.
- A squared norm in floating point: $X^2 + Y^2 + Z^2$
  - FP unpack / round / pack only once
  - parallel execution
  - no need to handle cancellations
  - symmetry respected thanks to internal fixed-point
  - altogether smaller, faster, more accurate

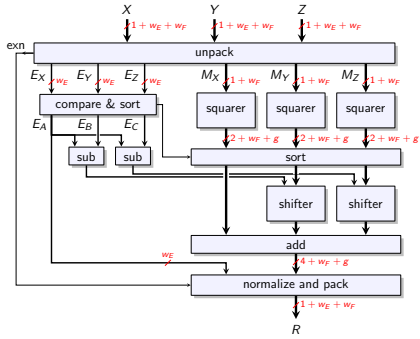
than 3 FP$\times$ and 2 FP $+$



Opportunity #4: Operator fusion

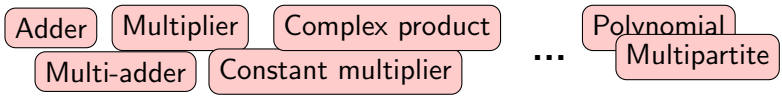
3 examples among many others:

- We just saw multiple constant multiplication.
- A squared norm in floating point: $X^2 + Y^2 + Z^2$
  - FP unpack / round / pack only once
  - parallel execution
  - no need to handle cancellations
  - symmetry respected thanks to internal fixed-point
  - altogether smaller, faster, more accurate than 3 FP$\times$ and 2 FP $+$
- A very generic idea: *bit-level merged arithmetic*
  - many-term sums, products, sums of products, ... multi-variate polynomials
  - a generic framework: the bit heap



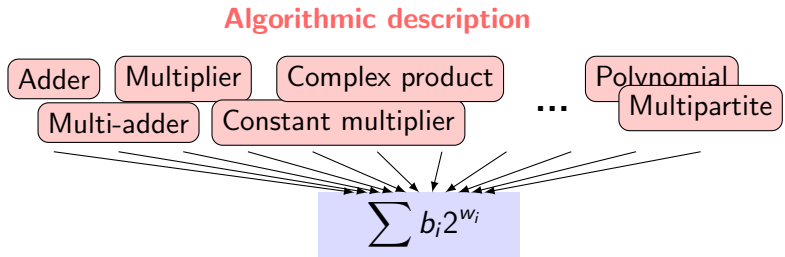
Merged arithmetic in bit heaps

Algorithmic description



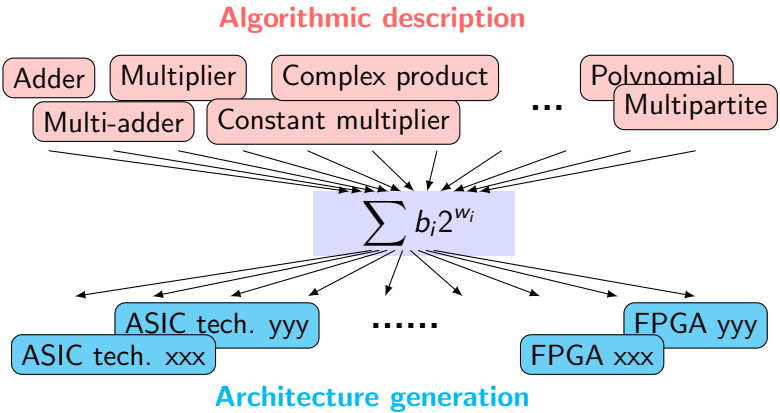
Merged arithmetic in bit heaps

One data-structure to rule them all...



Merged arithmetic in bit heaps

One data-structure to rule them all... and in the hardware to bind them

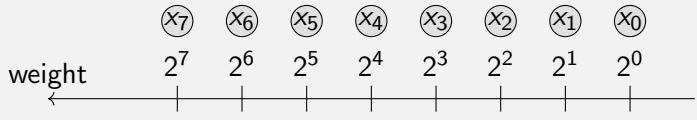


Weighted bits

Integers or real numbers represented in binary fixed-point

$$X = \sum_{i=i_{\min}}^{i_{\max}} 2^i x_i$$

Representation as a **dot diagram**

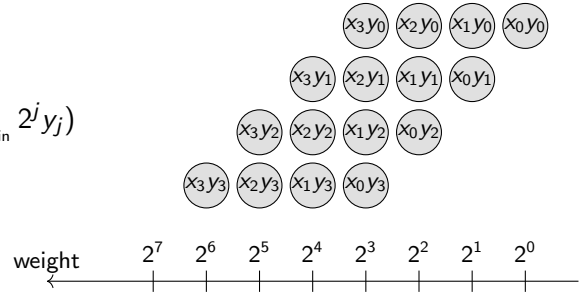


The historical bit heap

$$
\begin{aligned}
X \times Y &= \left(\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i\right) \times \left(\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j\right) \\
&= \sum_{i,j} 2^{i+j} x_i y_j
\end{aligned}
$$

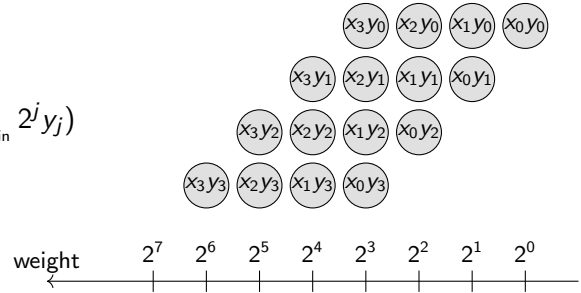
The historical bit heap

$$
\begin{aligned}
X \times Y &= \left(\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i\right) \times \left(\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j\right) \\
&= \sum_{i,j} 2^{i+j} x_i y_j
\end{aligned}
$$



The historical bit heap

$$X \times Y = (\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i) \times (\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j)$$
$$= \sum_{i,j} 2^{i+j} x_i y_j$$



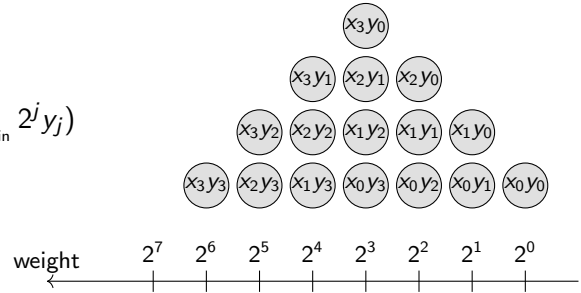
A multiplier is an architecture that computes this sum.

Historical motivation for bit heaps

$\sum_{i,j} 2^{i+j} x_i y_j$ **expresses the bit-level parallelism of the problem**

The historical bit heap

$$X \times Y = (\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i) \times (\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j)$$
$$= \sum_{i,j} 2^{i+j} x_i y_j$$



A multiplier is an architecture that computes this sum.

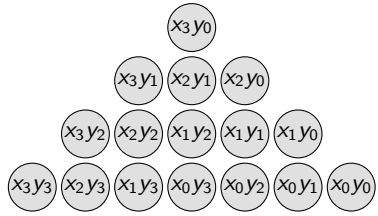
Historical motivation for bit heaps

$\sum_{i,j} 2^{i+j} x_i y_j$ **expresses the bit-level parallelism of the problem**

(freedom thanks to associativity and commutativity of addition)

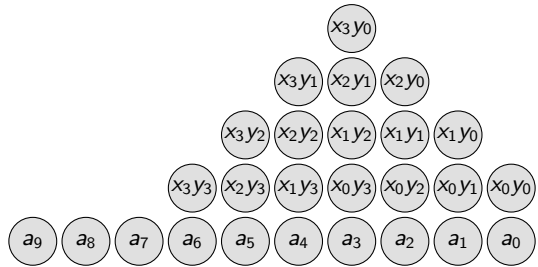
Beyond product

$$XY = \sum_{i,j} 2^{i+j} x_i y_j$$



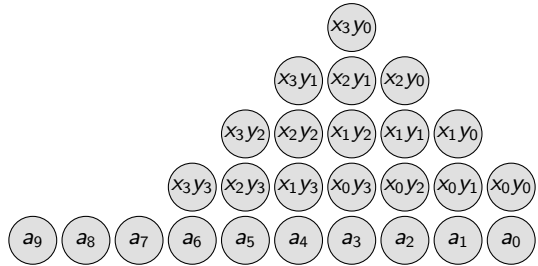
Beyond product

$$A + XY = \sum_i 2^i a_i + \sum_{i,j} 2^{i+j} x_i y_j$$



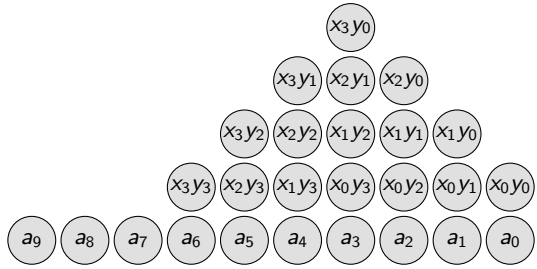
Beyond product

$$A + XY = \sum_{w,h} 2^w b_{w,h}$$



Beyond product

$$A + XY = \sum_{w,h} 2^w b_{w,h}$$

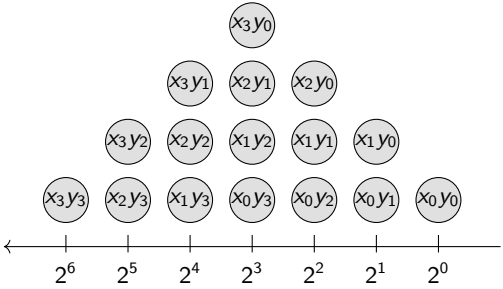


When generating an architecture

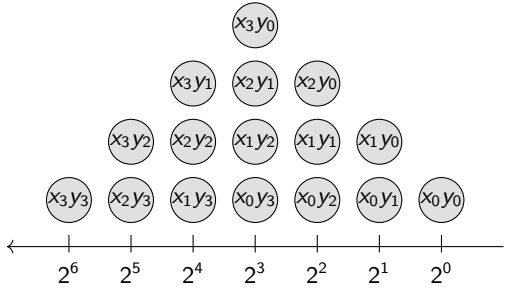
consider **only one big sum of weighted bits**

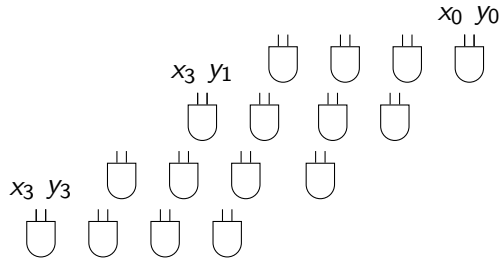
- get rid of artificial sequentiality (inside operators, and between operators)
- a global optimization instead of several local ones (and solved by ILP)

Computing the sum: bit heap compression

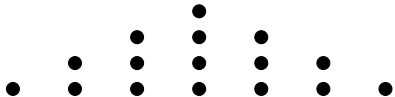


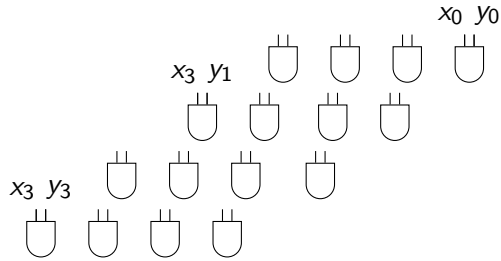
Computing the sum: bit heap compression



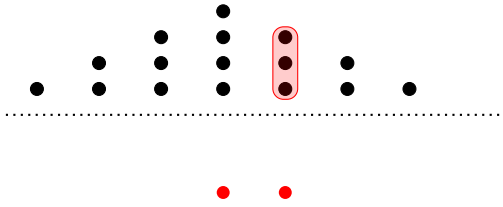


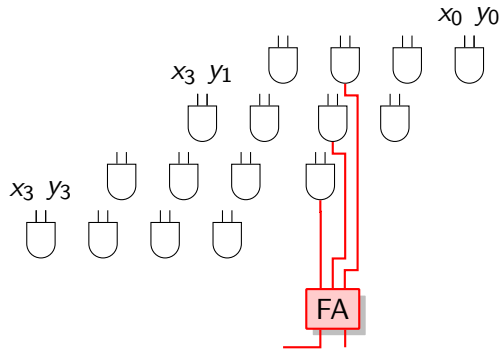
Computing the sum: bit heap compression



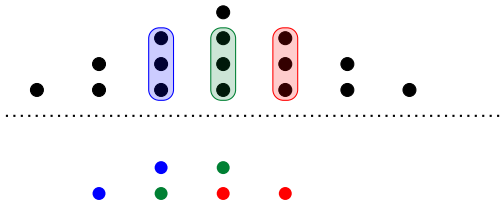


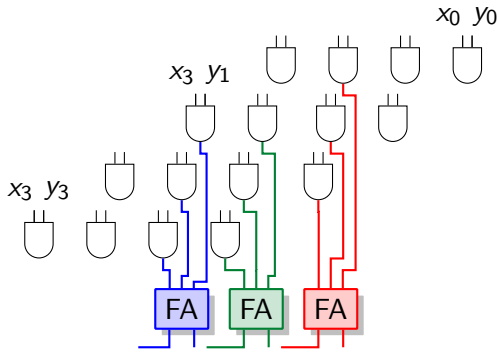
Computing the sum: bit heap compression



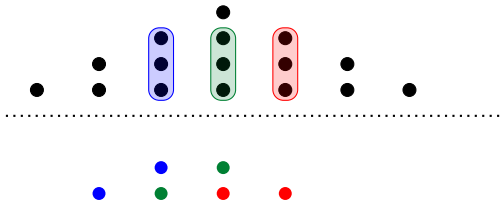


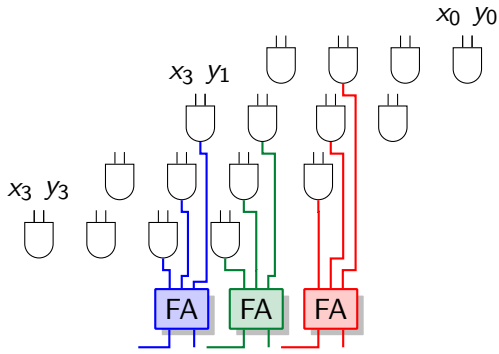
Computing the sum: bit heap compression



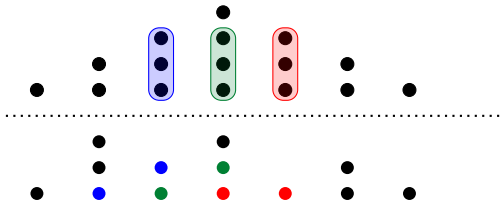


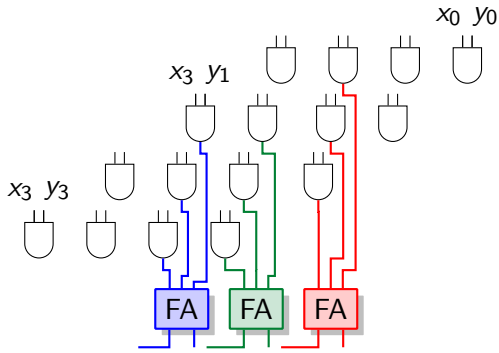
Computing the sum: bit heap compression



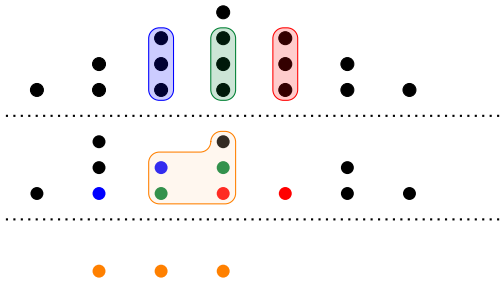


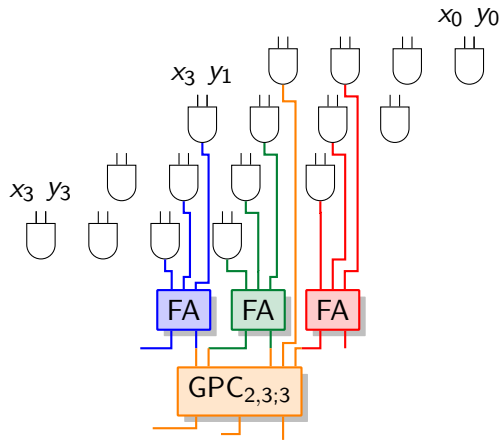
Computing the sum: bit heap compression



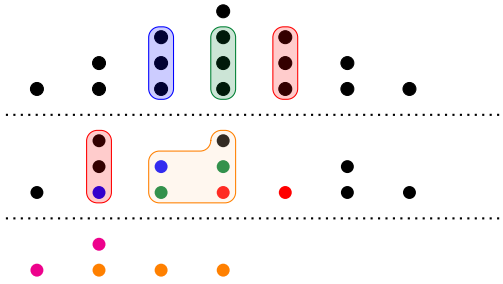


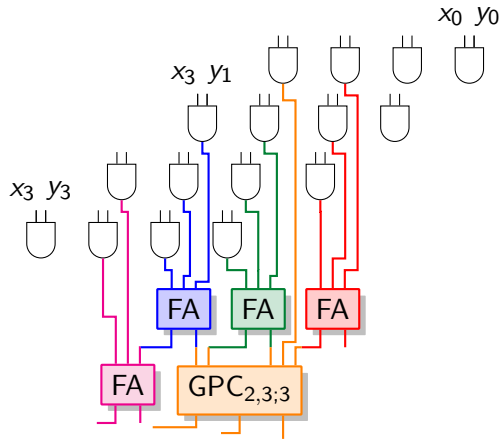
Computing the sum: bit heap compression



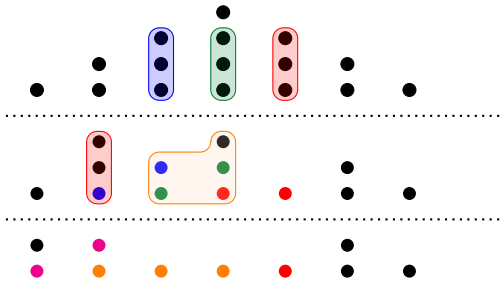


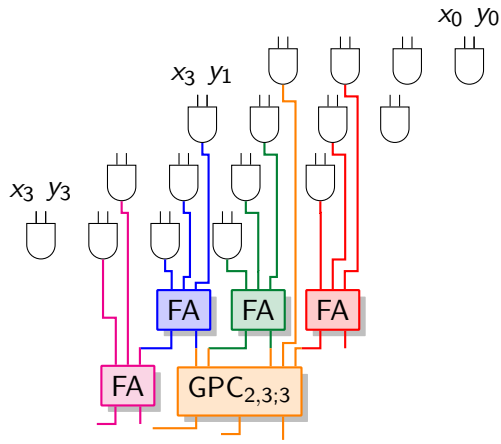
Computing the sum: bit heap compression



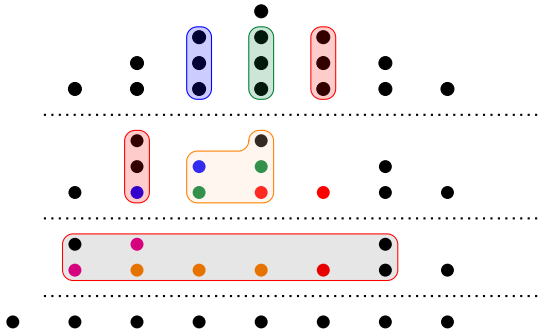


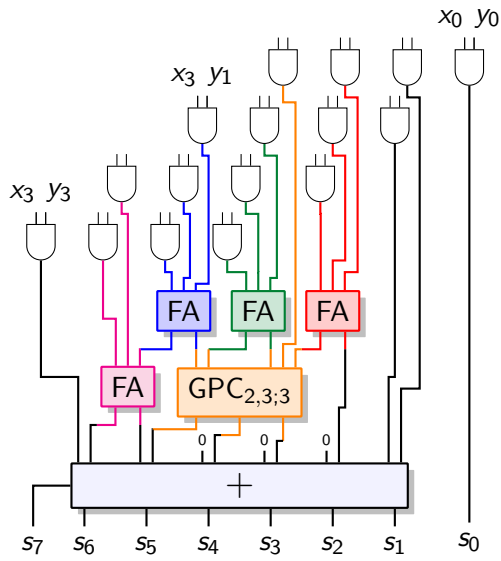
Computing the sum: bit heap compression





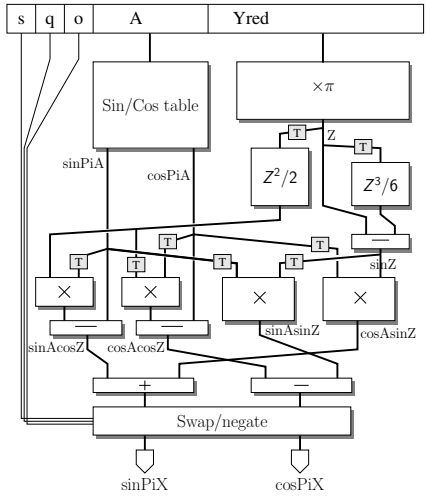
Computing the sum: bit heap compression





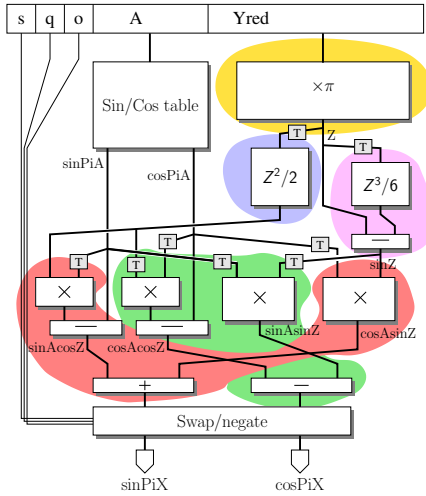
When you have a good hammer, you see nails everywhere

A sine/cosine architecture (Iştoan, HEART 2013):

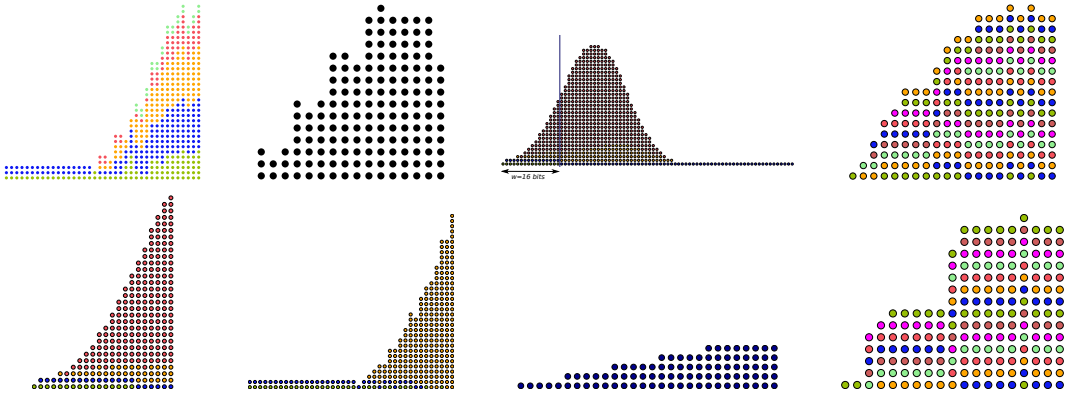


When you have a good hammer, you see nails everywhere

A sine/cosine architecture (Iştoan, HEART 2013): 5 bit heaps



Bit heaps for some operators and filters



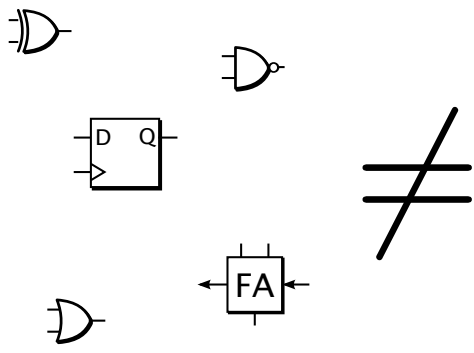
Why are some people still insisting I should call these “bit arrays”?

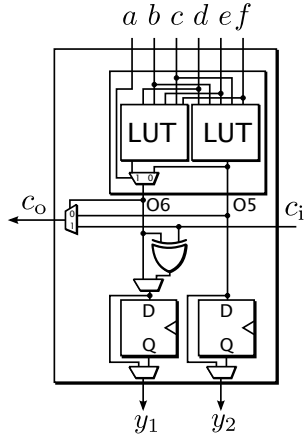
Opportunities of application-specific arithmetic

1. Operator parameterization
2. Operator specialization
3. Resource sharing
4. Operator fusion
5. **Target-specific optimizations**
6. Function evaluation

Opportunity #5: Target-specific optimizations

Optimizing gates does not mean that you optimize for the target technology (here: FPGAs)

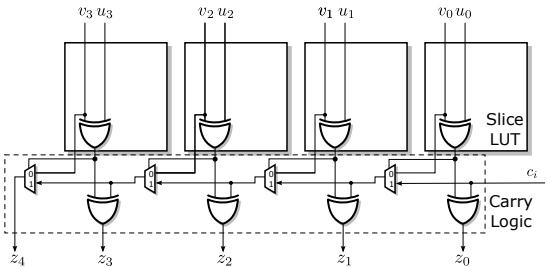


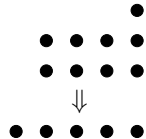


We should look the other way around!

Addition on FPGAs

There is so much free space in FPGA adders:

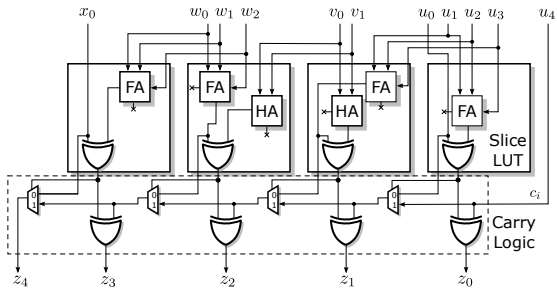


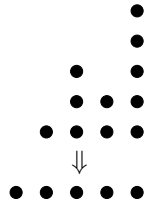


So, let's use this space to compress more bits!

Addition on FPGAs

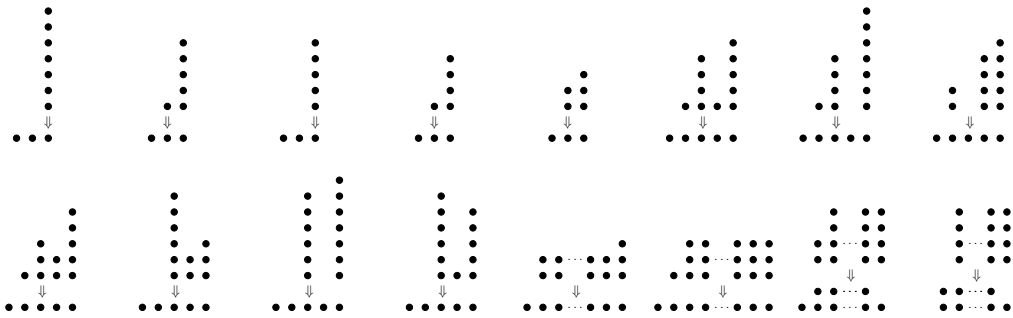
There is so much free space in FPGA adders:



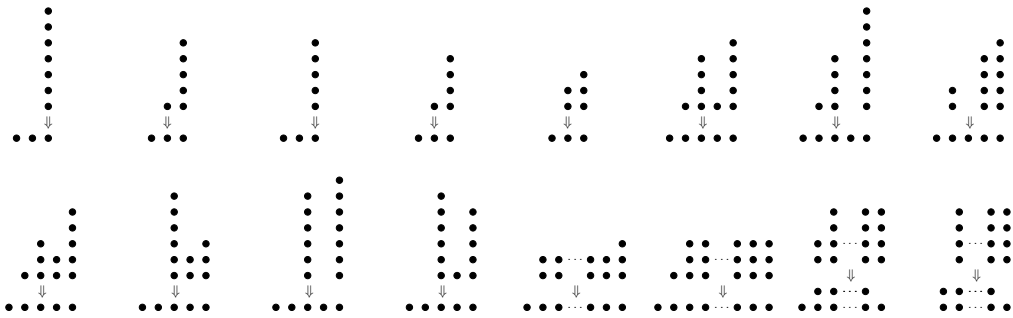


So, let's use this space to compress more bits!

A bestiary of compressors



A bestiary of compressors



Designing a compressor tree now becomes a challenge!

A bestiary of compressors

Heuristic and ILP-based optimal methods are there!

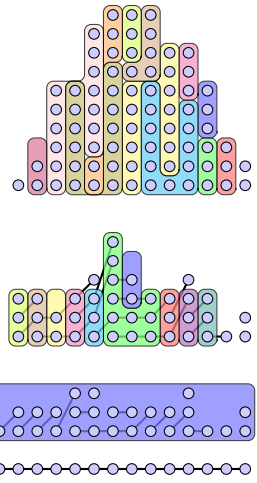
$$\text{minimize} \sum_{s=0}^{S-1} \sum_{c=0}^{C-1} \sum_{e=0}^{E-1} c_e k_{s,e,c}$$

subject to

$$\mathcal{C}1: \quad N_{s-1,c} \leq \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} M_{e,c+c'}\, k_{s-1,e,c+c'} \quad \text{for } s = 1 \ldots S,\ c = 0 \ldots C-1$$

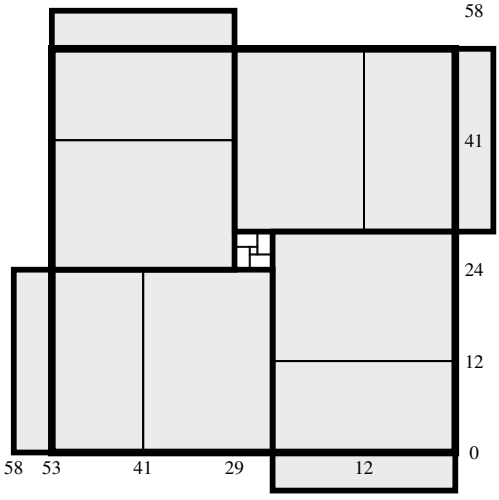
$$\mathcal{C}2: \quad N_{s,c} = \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} K_{e,c+c'}\, k_{s-1,e,c+c'} \quad \text{for } s = 1 \ldots S,\ c = 0 \ldots C-1$$

$$\mathcal{C}3: \quad N_{S,c} \leq I$$



Multiplier Tiling

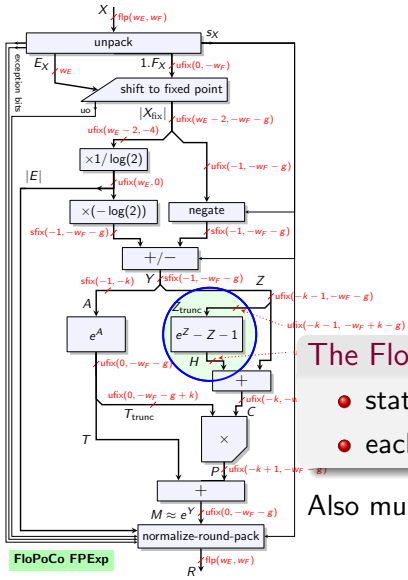
Instead of fixed radix-2, -4, etc. we use what the target provides (Logic and DSP)!

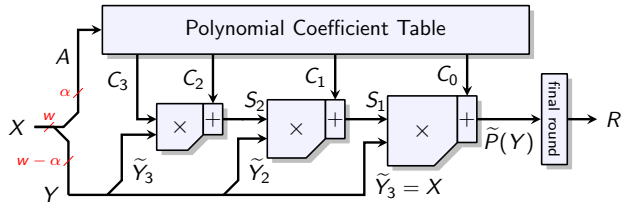


Opportunities of application-specific arithmetic

1. Operator parameterization
2. Operator specialization
3. Resource sharing
4. Operator fusion
5. Target-specific optimizations
6. **Function evaluation**

Opportunity #6: Function evaluation with generic approximators





The FloPoCo FixFunctionByPiecewisePoly operator

- state-of-the-art polynomial approximation
- each multiplier tailored with love and care

Also multipartite tables, filter approximators, and more to come.

Conclusion

Anti-introduction: traditional arithmetic

Opportunities of application-specific arithmetic

Conclusion

Lessons learnt over the past 15 years

- There is arithmetic beyond the ARITH logo
- For good floating-point, you also need good fixed-point
- (For good posit, you also need good floating-point)
- Implementing a hardware generator is even more fun than designing hardware
- Martin and his disciples are quietly replacing all the heuristic tinkered by Florent with mathematical models that capture the optimal operator

FloPoCo only solves the easy problem

DONE Good, flexible, versatile application-specific operators

TODO Now how many bits do I need for this variable in my application?

Regrettably, there still exist people who have not read all my papers

Example of bug report by a highly valued FloPoCo user

```
./flopoco FPConstMult wE=8 wF=23 constant=0.3333
```

Can you see what is wrong here?

Regrettably, there still exist people who have not read all my papers

Example of bug report by a highly valued FloPoCo user

```
./flopoco FPConstMult wE=8 wF=23 constant=0.3333
```

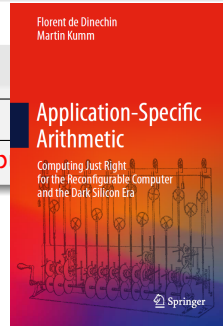
Can you see what is wrong here? $\approx 1/3 \pm 2^{-14}$ Argh! Not Computing Just Right!

Regrettably, there still exist people who have not read all my papers

Example of bug report by a highly valued FloPoCo user

```
./flopoco FPConstMult wE=8 wF=23 constant=0.3333
```

Can you see what is wrong here? $\approx 1/3 \pm 2^{-14}$ Argh! Not Comp



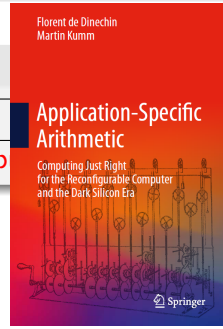
Solution 1: Did I mention that we published this book?

Regrettably, there still exist people who have not read all my papers

Example of bug report by a highly valued FloPoCo user

```
./flopoco FPConstMult wE=8 wF=23 constant=0.3333
```

Can you see what is wrong here? $\approx 1/3 \pm 2^{-14}$ Argh! Not Comp



Solution 1: Did I mention that we published this book?

Solution2

Integrate the FloPoCo spirit in a *High-Level Synthesis* compiler

(this means a C to hardware compiler, haha)

Current effort with MLIR, the Multi-Level Intermediate Representation.

Why move useless bits around?

Some of the successive advertising phrases for the FloPoCo project

- When FPGAs are better at floating point than microprocessors
- Not your neighbor's FPU
- All the operators you will never see in a microprocessor
- FPGA arithmetic the way it should be
- Circuits computing just right
- Fantastic arithmetic beasts (and how to build them)

Backup slides

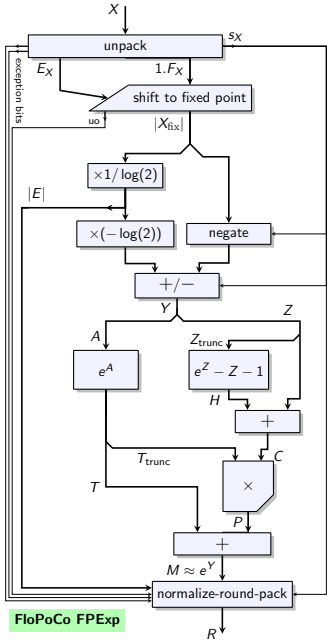
First, a math proficiency test

Three identities to remember from our happy school days

$$2^X = e^{X \log(2)} \tag{1}$$

$$e^{A+B} = e^A \times e^B \tag{2}$$

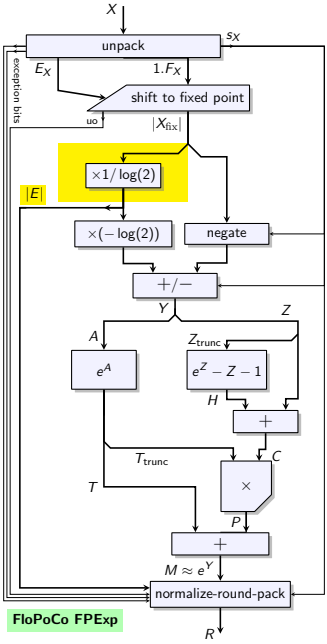
$$e^Z \approx 1 + Z + \frac{Z^2}{2} \quad \text{if } Z \text{ is small} \tag{3}$$



FloPoCo FPExp

We want to obtain e^X as

$$e^X = 2^E \cdot 1.F$$

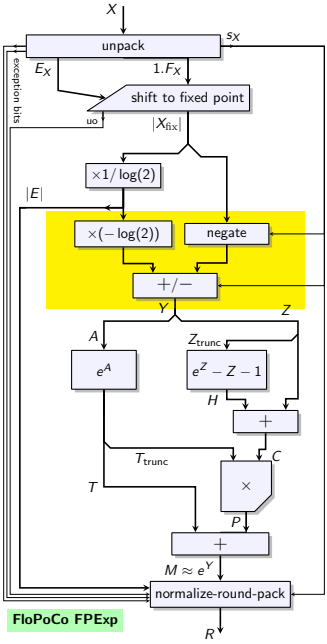


We want to obtain e^X as

$$e^X = 2^E \cdot 1.F$$

Compute

$$E \approx \left\lfloor \frac{X}{\log 2} \right\rceil$$



FloPoCo FPExp

We want to obtain e^X as

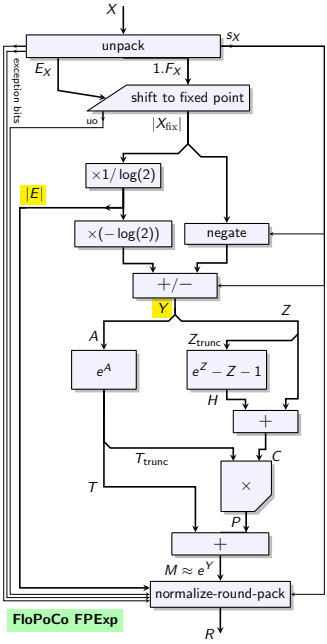
$$e^X = 2^E \cdot 1.F$$

Compute

$$E \approx \left\lfloor \frac{X}{\log 2} \right\rceil$$

then

$$Y \approx X - E \times \log 2.$$



We want to obtain e^X as

$$e^X = 2^E \cdot 1.F$$

Compute

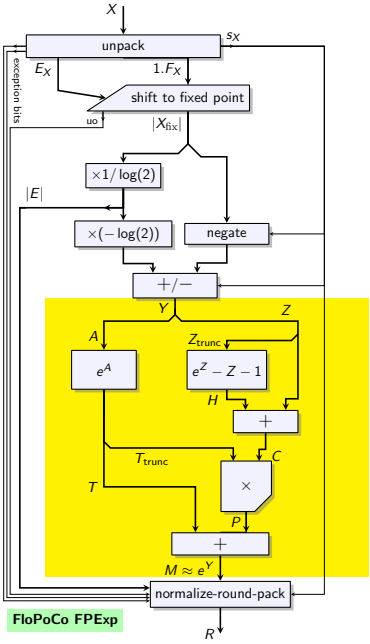
$$E \approx \left\lfloor \frac{X}{\log 2} \right\rceil$$

then

$$Y \approx X - E \times \log 2.$$

Now

$$\begin{aligned} e^X &= e^{E \log 2 + Y} \\ &= e^{E \log 2} \cdot e^Y \\ &= 2^E \cdot e^Y \end{aligned}$$

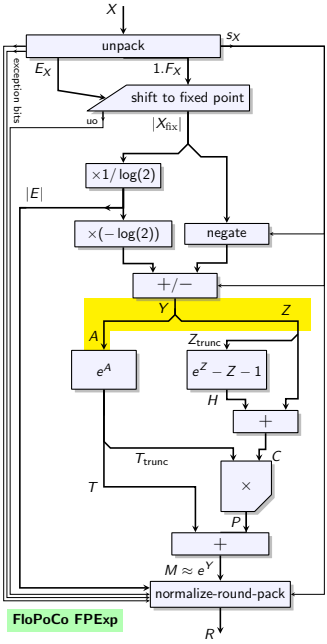


FloPoCo FPExp

We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

Now we have to compute e^Y
with $Y \in (-1/2, 1/2)$.



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

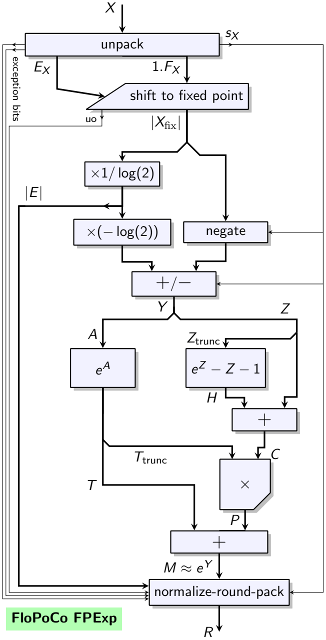
Now we have to compute e^Y
with $Y \in (-1/2, 1/2)$.

Split Y:

$$Y = \underbrace{\quad A \quad}_{-1 \;\ldots\; -k} \; \underbrace{\quad Z \quad}_{-k \;\ldots\; -w_F - g}$$

i.e. write

$$Y = A + Z \qquad \text{with} \quad Z < 2^{-k}$$

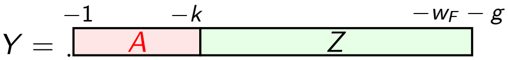


We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

Now we have to compute e^Y with $Y \in (-1/2, 1/2)$.

Split Y:

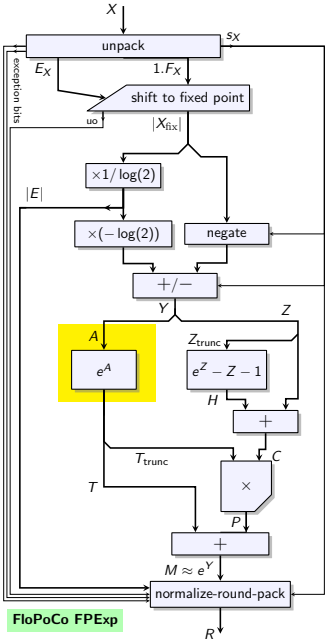


i.e. write

$$Y = A + Z \qquad \text{with} \quad Z < 2^{-k}$$

so

$$e^Y = e^A \times e^Z$$



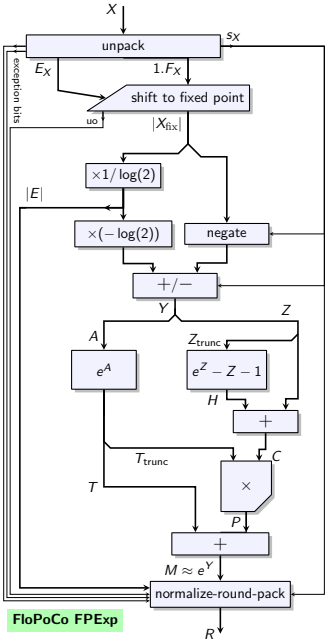
FloPoCo FPExp

We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

Tabulate e^A in a ROM



FloPoCo FPExp

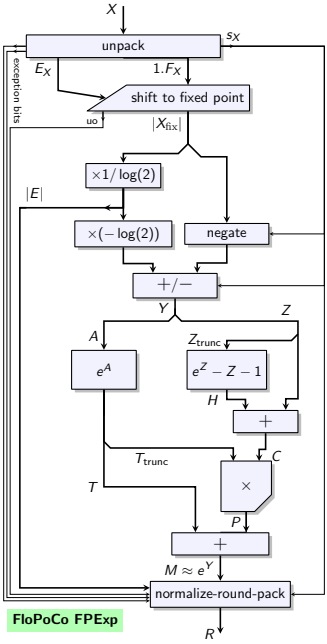
We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

Evaluation of e^Z: $Z < 2^{-k}$, so

$$e^Z \approx 1 + Z + Z^2/2$$



We want to obtain e^X as

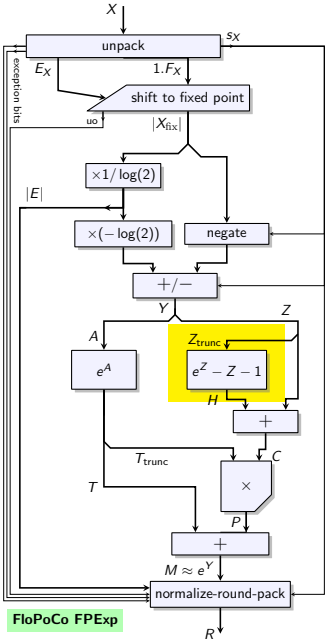
$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

Evaluation of e^Z: $Z < 2^{-k}$, so

$$e^Z \approx 1 + Z + Z^2/2$$

Notice that $e^Z - 1 - Z \approx Z^2/2 < 2^{-2k}$



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

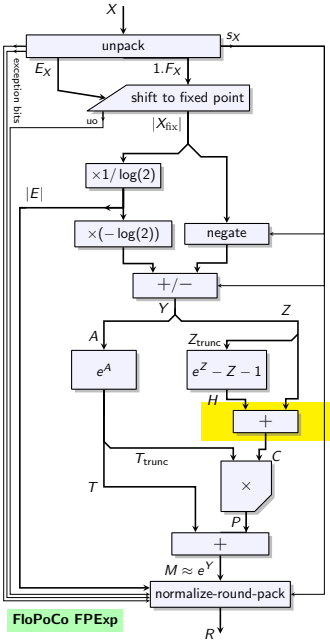
$$e^Y = e^A \times e^Z$$

Evaluation of e^Z: $Z < 2^{-k}$, so

$$e^Z \approx 1 + Z + Z^2/2$$

Notice that $e^Z - 1 - Z \approx Z^2/2 < 2^{-2k}$

Evaluate $e^Z - Z - 1$ somewhow
(out of Z truncated to its higher bits only)



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

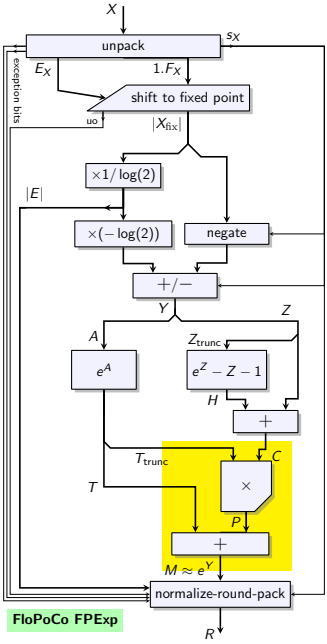
$$e^Y = e^A \times e^Z$$

Evaluation of e^Z: $Z < 2^{-k}$, so

$$e^Z \approx 1 + Z + Z^2/2$$

Notice that $e^Z - 1 - Z \approx Z^2/2 < 2^{-2k}$

Evaluate $e^Z - Z - 1$ somewhow
(out of Z truncated to its higher bits only)
then add Z to obtain $e^Z - 1$



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

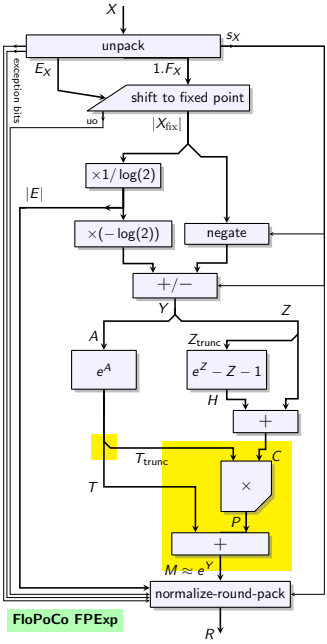
$$e^Y = e^A \times e^Z$$

Also notice that

$$e^Z = 1.\overbrace{000...000}^{k-1 \text{ zeroes}} zzzz$$

Evaluate $e^A \times e^Z$ as

$$e^A + e^A \times (e^Z - 1)$$



We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

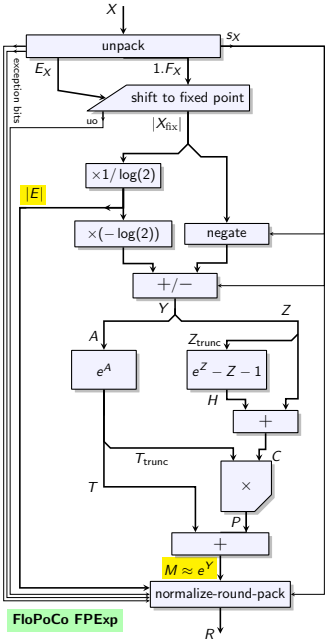
Also notice that

$$e^Z = 1.\overbrace{000...000}^{k-1 \text{ zeroes}} zzzz$$

Evaluate $e^A \times e^Z$ as

$$e^A + e^A \times (e^Z - 1)$$

(before the product, truncate e^A to precision of $e^Z - 1$)

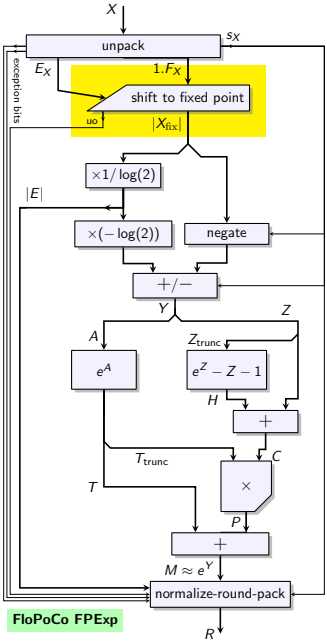


We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

And that's it, we have E and e^Y

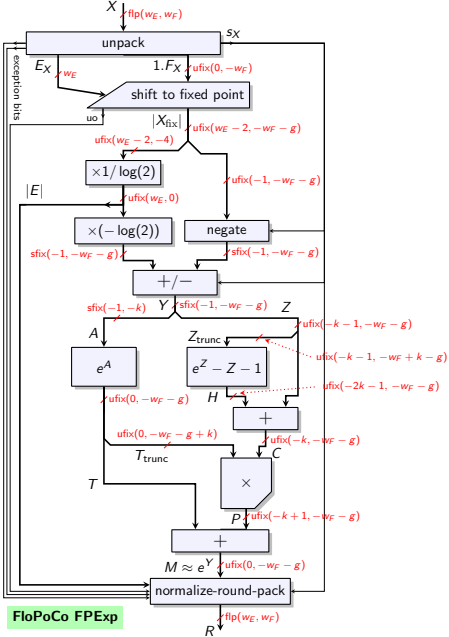


We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

And that's it, we have E and e^Y
(using only *fixed-point* computations)



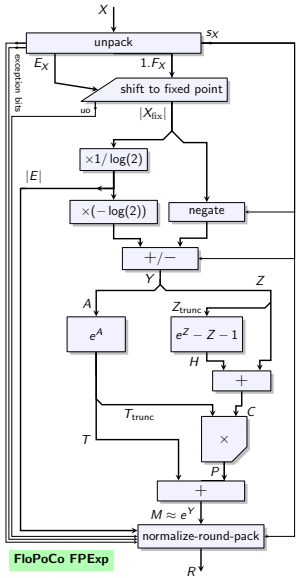
We want to obtain e^X as

$$e^X = 2^E \cdot e^Y$$

$$e^Y = e^A \times e^Z$$

And that's it, we have E and e^Y
(using only *fixed-point* computations)

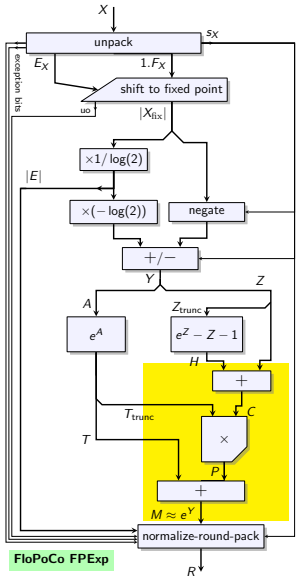
Single-precision magic



FloPoCo FPExp

Modern FPGAs also have

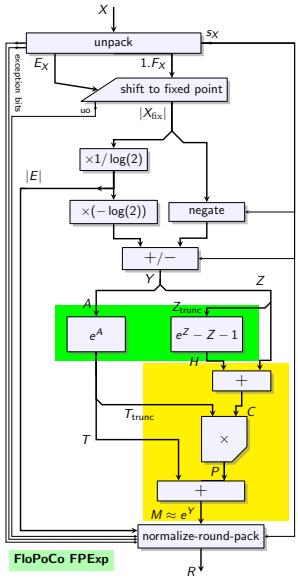
Single-precision magic



Modern FPGAs also have

- small multipliers with pre-adders and post-adders

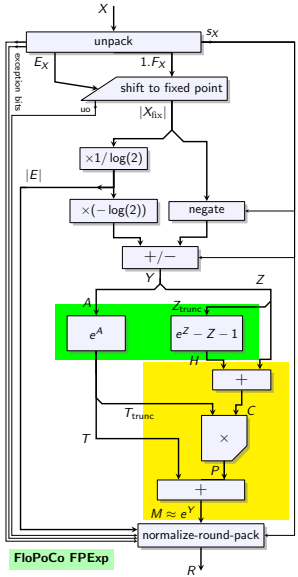
Single-precision magic



Modern FPGAs also have

- small multipliers with pre-adders and post-adders
- ... and dual-ported small memories

Single-precision magic



Modern FPGAs also have

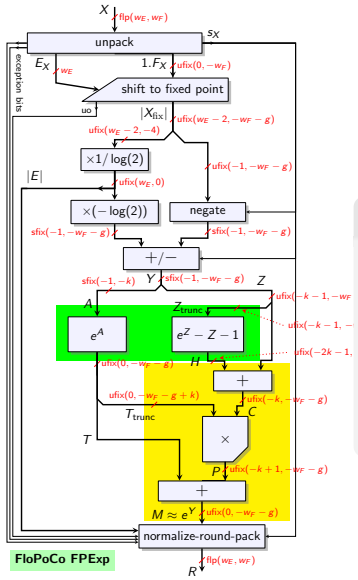
- small multipliers with pre-adders and post-adders
- ... and dual-ported small memories

Single-precision accurate exponential on Xilinx

- one block RAM (0.1% of the chip)
- one DSP block (0.1%)
- $<$ 400 LUTs (0.1%, $\approx$ one FP adder)

to compute one exponential per cycle at 500MHz
($\sim$ one AVX512 core trashing on its 16 FP32 lanes)

Single-precision magic



Modern FPGAs also have

- small multipliers with pre-adders and post-adders
- ... and dual-ported small memories

Single-precision accurate exponential on Xilinx

- one block RAM (0.1% of the chip)
- one DSP block (0.1%)
- $<$ 400 LUTs (0.1%, $\approx$ one FP adder)

to compute one exponential per cycle at 500MHz
($\sim$ one AVX512 core trashing on its 16 FP32 lanes)

For one specific value only of the architectural parameter k!
(over-parameterization is cool)