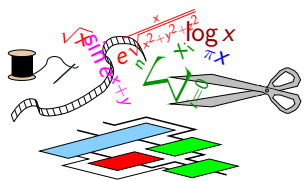
FPGAs Computing Just Right: Application-Specific Arithmetic

Florent de Dinechin



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

Outline

Anti-introduction: the arithmetic you want in a processor

Some opportunities of hardware computing just right

Conclusion: the FloPoCo project

Anti-introduction: the arithmetic you want in a processor

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 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 25 by 5 in binary?

```
011001 | 101
       |----
       | ?
       |
       |
       |
```

Just like decimal, but simpler

- find the next quotient digit

Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
 011001 | 101
 -101   |-----
=0001   | 1
        |
        |
```

Just like decimal, but simpler

- find the next quotient digit — *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*

Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | ----
 00101 | 1
       |
       |
       |
       |
```

Just like decimal, but simpler

- find the next quotient digit — *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 remainder is positive,
  - keep it and proceed to next iteration
- start again, one digit to the right

Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 1?
 00101 |
       |
       |
       |
```

Just like decimal, but simpler

- find the next quotient digit — *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 remainder is positive,
  - keep it and proceed to next iteration
- start again, one digit to the right

Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 11
 00101
 -101
=1101
```

Just like decimal, but simpler

- find the next quotient digit — *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 remainder is positive,
  - keep it and proceed to next iteration
- if the remainder is negative,
- start again, one digit to the right

Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 10?
 00101 |
 -101  |
=1101  |
 00101 |
       |
```

Just like decimal, but simpler

- find the next quotient digit — *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 remainder is positive,
  - keep it and proceed to next iteration
- if the remainder is negative,
  - quotient digit should have been 0, undo the subtraction
- start again, one digit to the right

Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 101
 00101 |
 -101  |
=1101  |
 00101 |
  -101 |
  =0000|
```

Just like decimal, but simpler

- find the next quotient digit — *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 remainder is positive,
  - keep it and proceed to next iteration
- if the remainder is negative,
  - quotient digit should have been 0, undo the subtraction
- start again, one digit to the right

Should a processor include a divider? (1)

How do you divide 25 by 5 in binary?

```
011001 | 101
-101   | 101
 00101 |
 -101  |
=1101  |
 00101 |
  -101 |
   000 |
```

Just like decimal, but simpler

- find the next quotient digit — *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 remainder is positive,
  - keep it and proceed to next iteration
- if the remainder is negative,
  - quotient digit should have been 0, undo the subtraction
- start again, one digit to the right

Light iteration (one subtraction and one test), but **one bit of the quotient per iteration**:
(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.

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

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*

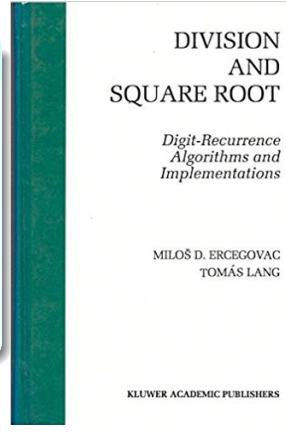
Digit recurrence algorithms

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, expensive processor... without a divide instruction.

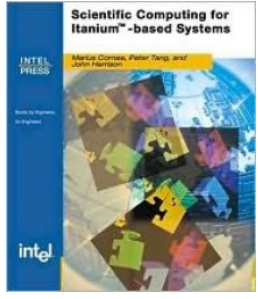
Instead of a hardware divider,

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

... and can even be used to compute divisions:

Multiplicative division algorithms

- several algorithms
  using a handful of multiplications
- the freedom of software:
  - quick and dirty, or accurate but slow
  - high throughput or short latency
  - ...
- and with a second FMA,
  BLAS and FFTs are 2x faster !



... and two more books.

Should a processor include a divider? (4)

Answer in 2022 is : **YES** again (Bruguera, Arith 2018)

Should a processor include a divider? (4)

Answer in 2022 is : **YES** again (Bruguera, Arith 2018)

- a double-precision divider in 11 cycles for ARM processors
- thanks to a **totally redneck** implementation
  - hardware: **20** fast 58-bit adders, **12** 58-bit muxes, tables, and more ...
  - (hardware speculation all over the place, etc)

Should a processor include a divider? (4)

Answer in 2022 is : **YES** again (Bruguera, Arith 2018)

- a double-precision divider in 11 cycles for ARM processors
- thanks to a **totally redneck** implementation
  - hardware: **20** fast 58-bit adders, **12** 58-bit muxes, tables, and more ...
  - (hardware speculation all over the place, etc)

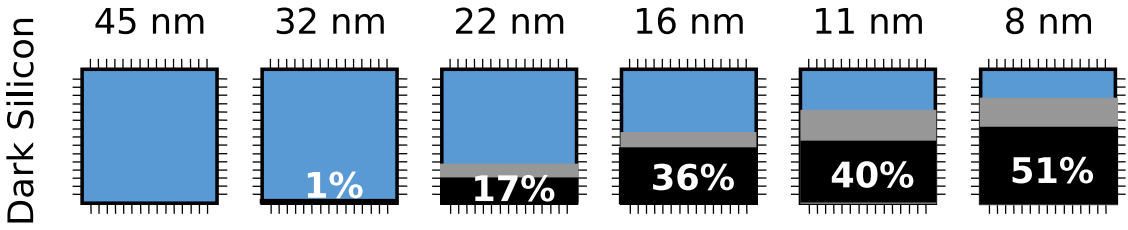
We do this to reduce overal energy consumption!
There is this huge superscalar ARM core that consumes a lot,
we save energy if we can switch it off a few cycles earlier

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

Should a processor include elementary functions? (1)

Dura Amdahl lex, sed lex

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

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 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 2022 is... **maybe**?

Should a processor include elementary functions? (4)

Answer in 2022 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

- ✓ *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 an FFT operator?*
- *Should a processor include an AI accelerator?*
- ...
- *Should a processor include a divider by 3? A multiplier by* $\log(2)$ *?*

no, of course.

At this point of the talk...

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

One nice thing with FPGAs

On FPGAs, there is a simpler 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

On FPGAs, there is a simpler 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

On FPGAs, there is a simpler 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
- ...

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`,

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 $\neq$ arithmetic for CPUs or GPGPUs

Application-specific arithmetic

All sorts of arithmetic operators that just **wouldn't make sense** in a processor can be useful in FPGAs.
This is what the FloPoCo project is about.

Conclusion so far

FPGA arithmetic $\neq$ arithmetic for CPUs or GPGPUs

Application-specific arithmetic

All sorts of arithmetic operators that just **wouldn't make sense** in a processor can be useful in FPGAs.
This is what the FloPoCo project is about.

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

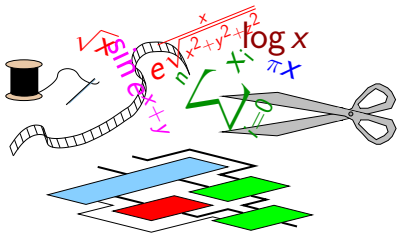
Computing just right

In FPGAs, data formats may be tightly fitted to the requirements of the application (not only 8, 16, 32 or 64 bits)...

Let us discuss this, too.

Computing just right?

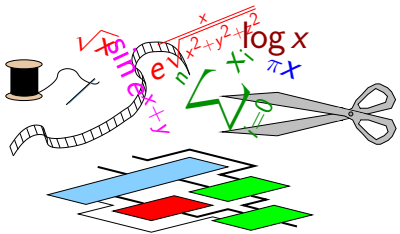
This is the pathetic logo of the FloPoCo project:



(the proper term is probably *allogory*)

Computing just right?

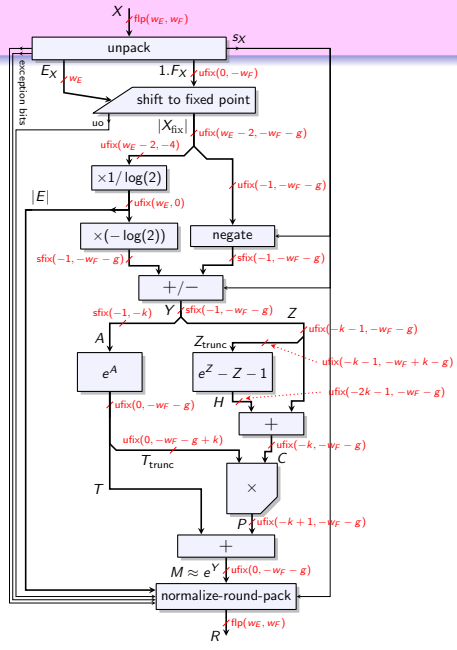
This is the pathetic logo of the FloPoCo project:



(the proper term is probably *allogory*)

This is the kind of thing FloPoCo does ⟶
It is a floating-point exponential operator
where each wire, each component is
tailored to its context with love and care.

(not a very good logo either)



Save power! Don't move useless bits around!

What is true for transatlantic cat videos is also true inside a circuit.

In software, if your result is correct, it is probably wasteful

Did you really need the bits 18 to 31 of this 32-bit word?

- If they carry useless noise, you don't want to compute them...
- ... and you want even less to compute on them.
- But in software, you don't really have the choice (it's 32 bits or 64 bits)

Save power! Don't move useless bits around!

What is true for transatlantic cat videos is also true inside a circuit.

In software, if your result is correct, it is probably wasteful

Did you really need the bits 18 to 31 of this 32-bit word?

- If they carry useless noise, you don't want to compute them...
- ... and you want even less to compute on them.
- But in software, you don't really have the choice (it's 32 bits or 64 bits)

Here we have more freedom when designing hardware

- In a circuit, we may choose, for each variable, how many bits are computed/stored/transmitted! $\longrightarrow$ **the opportunities**
- Overwhelming freedom! Help! $\longrightarrow$ **the challenges**

Save power! Don't move useless bits around!

What is true for transatlantic cat videos is also true inside a circuit.

In software, if your result is correct, it is probably wasteful

Did you really need the bits 18 to 31 of this 32-bit word?

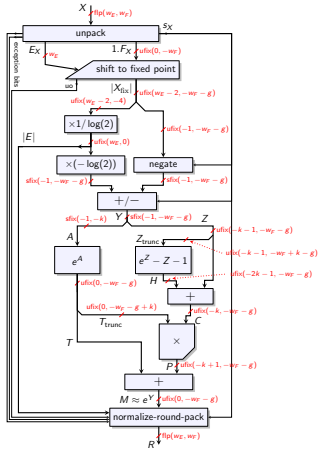
- If they carry useless noise, you don't want to compute them...
- ... and you want even less to compute on them.
- But in software, you don't really have the choice (it's 32 bits or 64 bits)

Here we have more freedom when designing hardware

- In a circuit, we may choose, for each variable, how many bits are computed/stored/transmitted! $\longrightarrow$ **the opportunities**
- Overwhelming freedom! Help! $\longrightarrow$ **the challenges**

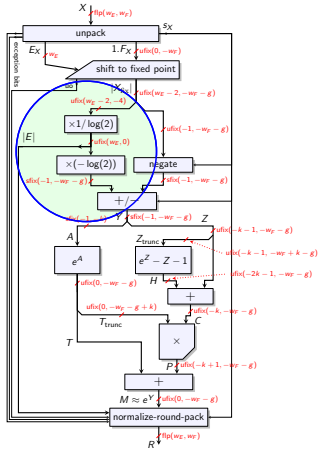
Some opportunities of hardware computing just right

Opportunity #1: Over-parameterization



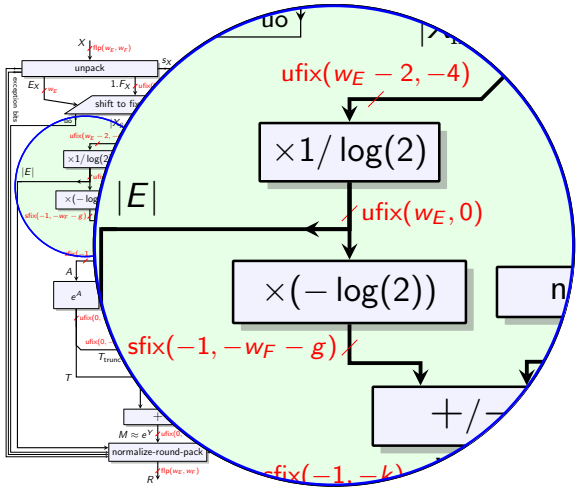
Example:

Opportunity #1: Over-parameterization



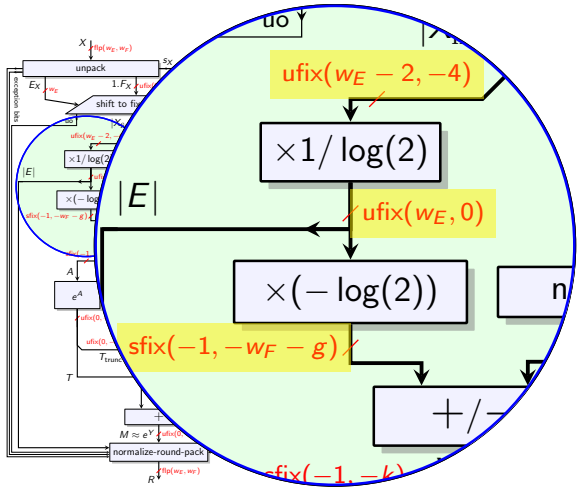
Example:

Opportunity #1: Over-parameterization



Example:

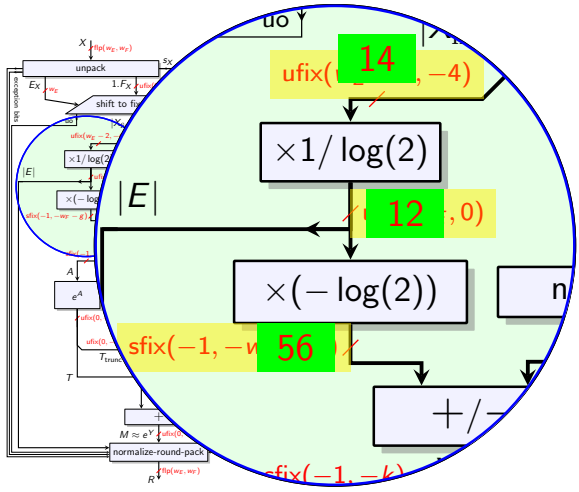
Opportunity #1: Over-parameterization



Example:

Multipliers of all shapes and sizes

Opportunity #1: 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 cool

- 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 cool

- ⊖ 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
  - People used to publish "An exponential architecture for single-precision", standard is now "A family of exponential architectures for each precision"
  - Application-specific optimal, future-proof, etc.

Over-parameterization is cool

⊖ 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
- People used to publish "An exponential architecture for single-precision", standard is now "A family of exponential architectures for each precision"
- Application-specific optimal, 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 ILP or common sense (whichever gives the best results)

Opportunity #2: Operator specialization

Ha, that's something software people don't get!

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$
  - but also by reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - *An FFT mostly consists of constant multiplications*

Opportunity #2: Operator specialization

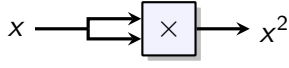
Ha, that's something software people don't get!

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$
  - but also by reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - *An FFT mostly consists of constant multiplications*
- Division by 3 (for various values of 3)
  - in floating point for Jacobi and other stencils
  - integer (quotient and remainder) for addressing in 3 memory banks

Opportunity #2: Operator specialization

Ha, that's something software people don't get!

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$
  - but also by reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - *An FFT mostly consists of constant multiplications*
- Division by 3 (for various values of 3)
  - in floating point for Jacobi and other stencils
  - integer (quotient and remainder) for addressing in 3 memory banks
- A squarer is a multiplier specialization

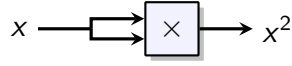


$$\begin{array}{rr} & 321 \\ \times & 321 \\ \hline & 321 \\ & 642 \\ & 963 \\ \hline & 103041 \end{array}$$

Opportunity #2: Operator specialization

Ha, that's something software people don't get!

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$
  - but also by reals such as $\log(2)$ or $\sin(42\pi/256)$
  - Two main techniques, tens of papers
  - *An FFT mostly consists of constant multiplications*
- Division by 3 (for various values of 3)
  - in floating point for Jacobi and other stencils
  - integer (quotient and remainder) for addressing in 3 memory banks
- A squarer is a multiplier specialization

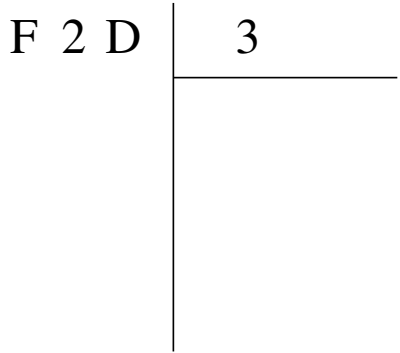


- Specialization of elementary functions to specific domains
- ...

$$\begin{array}{r} 321 \\ \times\ 321 \\ \hline 321 \\ 642 \\ 963 \\ \hline 103041 \end{array}$$

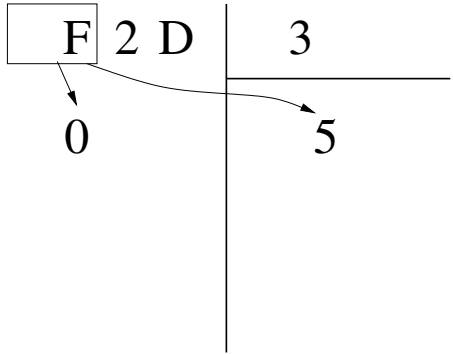
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



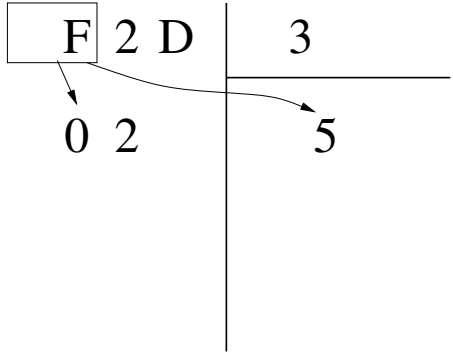
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



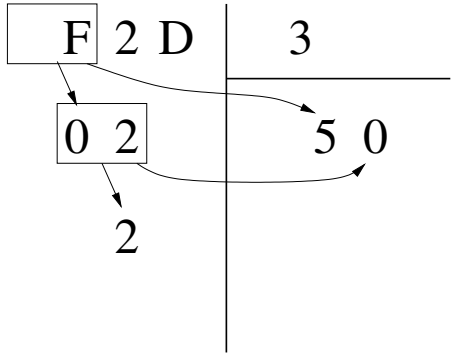
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



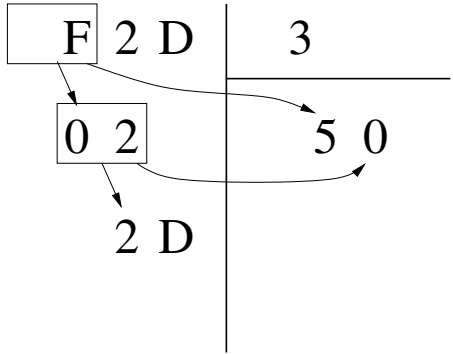
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3



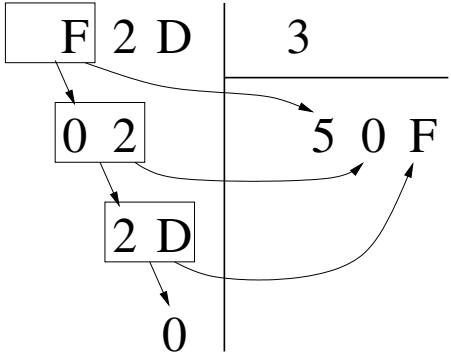
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3

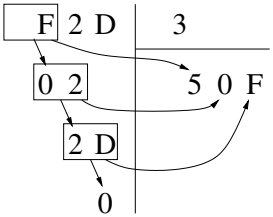


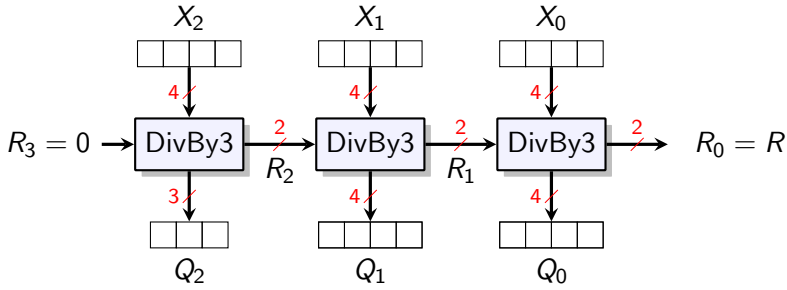
Maybe more people will understand division by 3 than exponential?

Dividing an **hexadecimal** number by 3

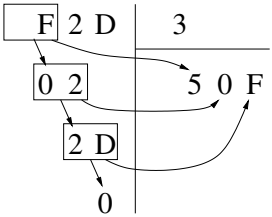


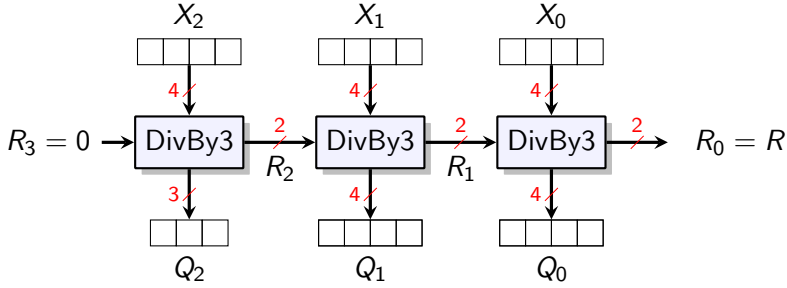
Getting inspiration from the vexations of childhood





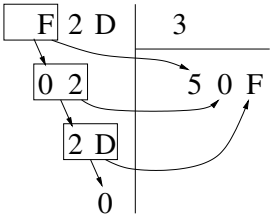
Getting inspiration from the vexations of childhood

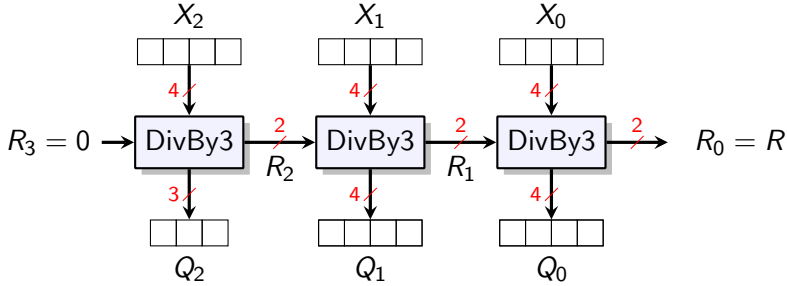




OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

Getting inspiration from the vexations of childhood

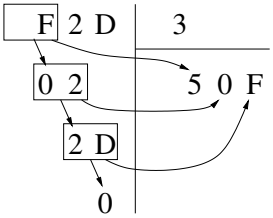


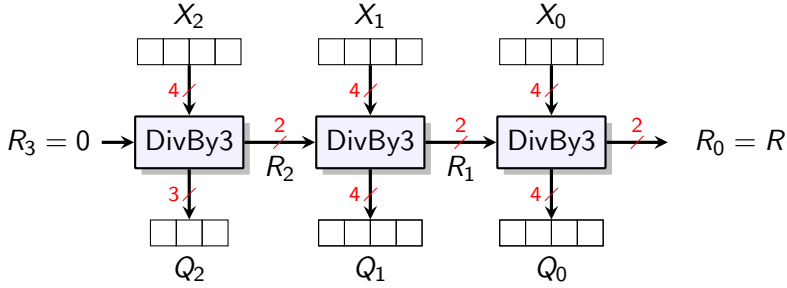


OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

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

Getting inspiration from the vexations of childhood





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

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

If you're too lazy to compute, then tabulate

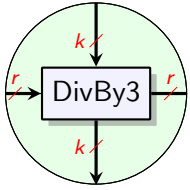
... here a table of 2^6 entries of 6 bits each.

What, my taxpayer money is wasted on studies of division by 3?

We did it for the fun of it, but it turns out to be useful for

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

Opportunity #3: target-specific optimizations

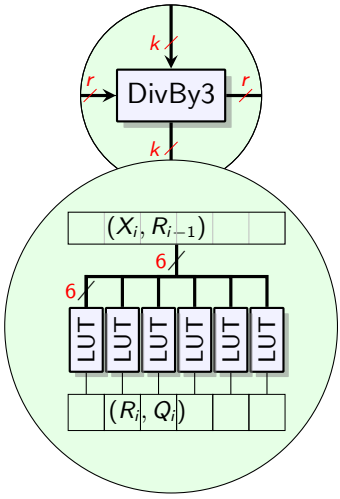


Generalizing hexadecimal to radix 2^k

... or, how over-parameterization allows for adaptation

- to various values of 3, like $D = 5$, or 7, or 9

Opportunity #3: target-specific optimizations



Generalizing hexadecimal to radix 2^k

... or, how over-parameterization allows for adaptation

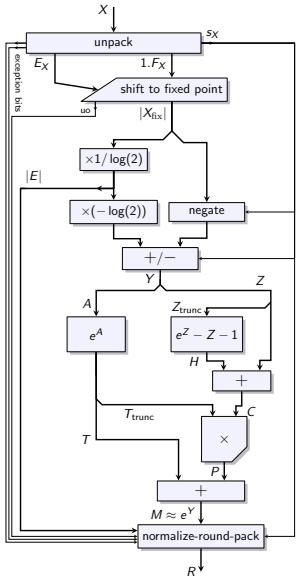
- to various values of 3, like $D = 5$, or 7, or 9
- to a given FPGA

Perfect match to modern FPGAs

Unit of area: the LUT, with α input bits (here $\alpha = 6$)

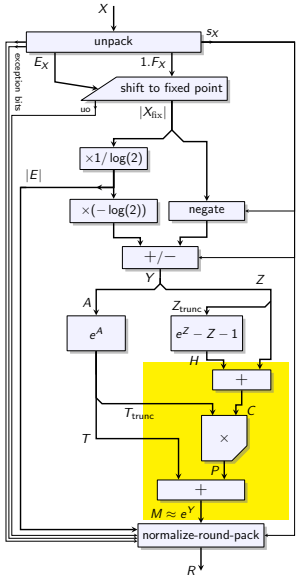
Isn't over-parameterization cool?

Opportunity #3: target-specific optimizations



Modern FPGAs also have

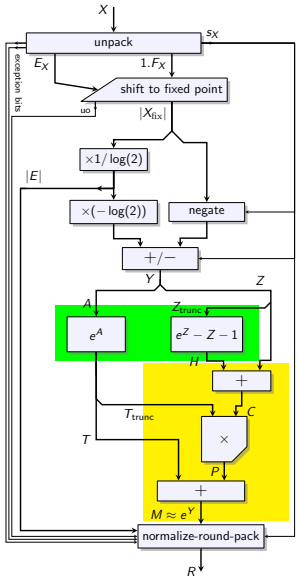
Opportunity #3: target-specific optimizations



Modern FPGAs also have

- small multipliers with pre-adders and post-adders

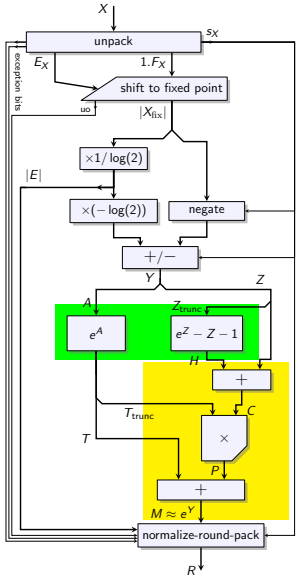
Opportunity #3: target-specific optimizations



Modern FPGAs also have

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

Opportunity #3: target-specific optimizations



Modern FPGAs also have

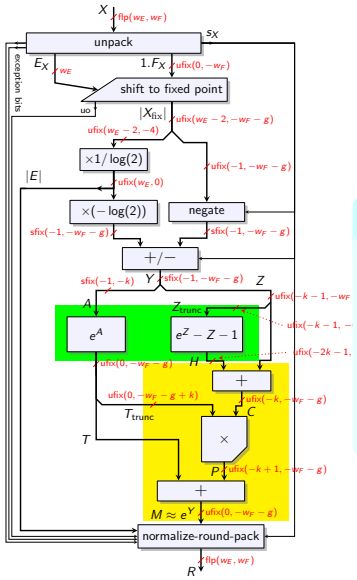
- small multipliers with pre-adders and post-adders
- ... and dual-ported small memories

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)

Opportunity #3: target-specific optimizations



Modern FPGAs also have

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

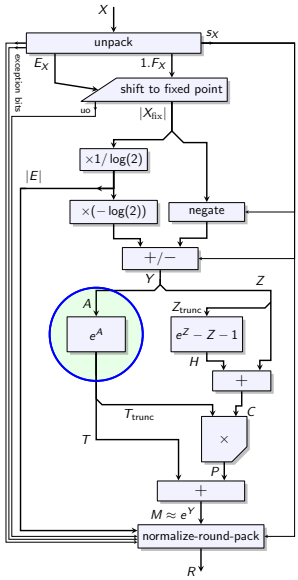
Single-precision accurate exponential on Xilinx

- 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)

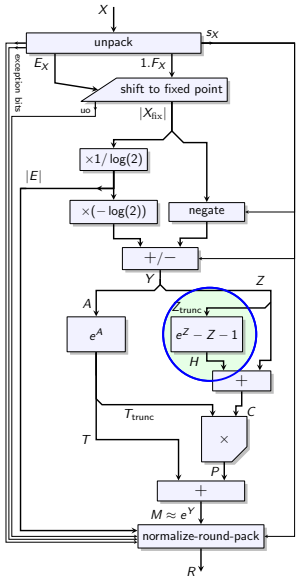
Opportunity #4: Tabulation



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

- ... and all the possible exponentials

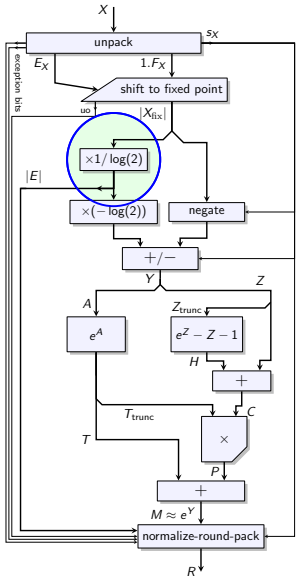
Opportunity #4: Tabulation



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 $e^Z - Z - 1$

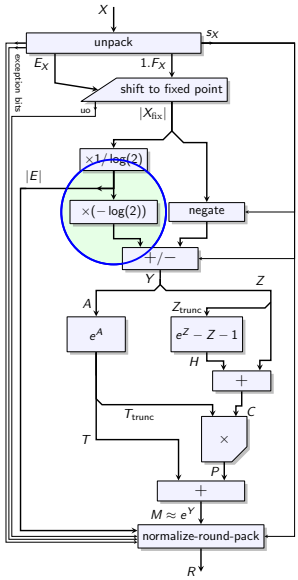
Opportunity #4: Tabulation



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

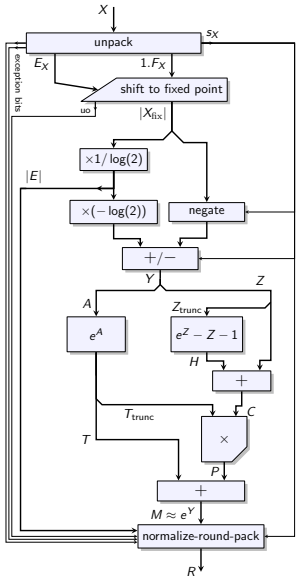
Opportunity #4: Tabulation



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

Opportunity #4: Tabulation

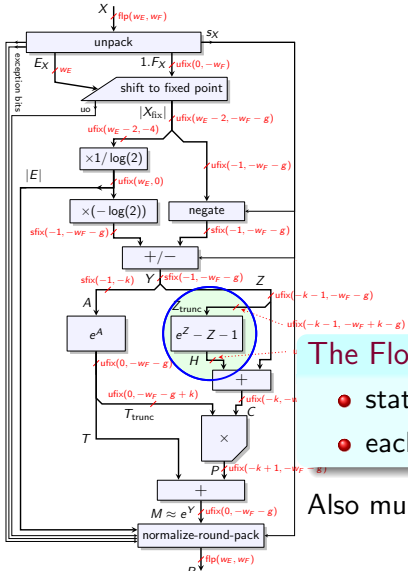


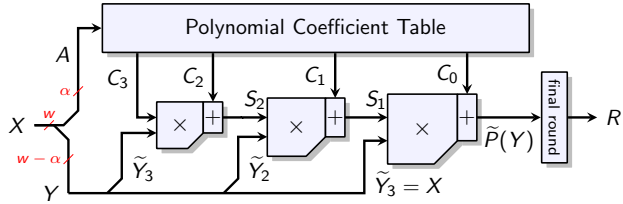
Being unable to trust my reasoning, I learnt by heart
the results of all the possible multiplications
(E. Ionesco)

- ... 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.

Opportunity #5: Generic approximators (when tabulation won't scale)



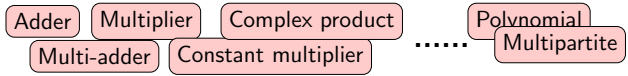


The FloPoCo FixFunctionByPiecewisePoly operator

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

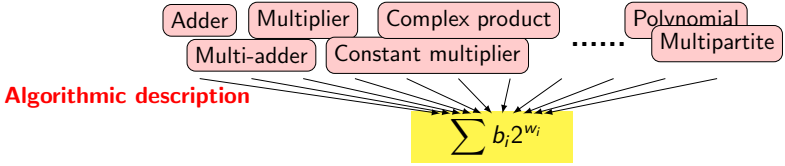
Also multipartite tables, filter approximators, and more to come.

Opportunity #6: merged arithmetic in bit heaps



Opportunity #6: merged arithmetic in bit heaps

One data-structure to rule them all...

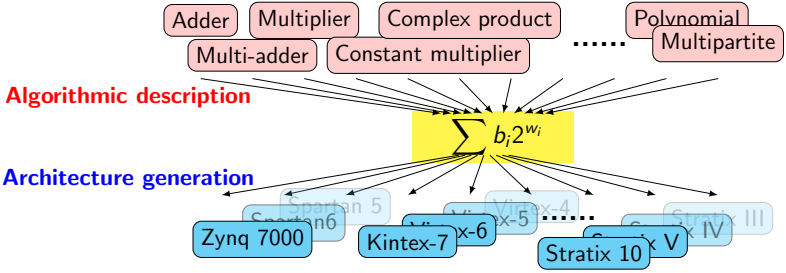


The **sum of weighted bits** as a first-class arithmetic object

- A very wide class of operators: multi-valued polynomials, and more
- Captures the true bit-level parallelism, enables bit-level optimization opportunities

Opportunity #6: merged arithmetic in bit heaps

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

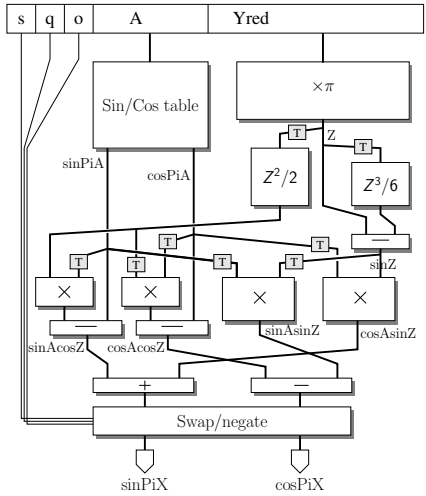


The **sum of weighted bits** as a first-class arithmetic object

- A very wide class of operators: multi-valued polynomials, and more
- Captures the true bit-level parallelism, enables bit-level optimization opportunities
- **Bit-array compressor trees** can be optimized for each target
  ... and optimally so for practical sizes, thanks to M. Kumm

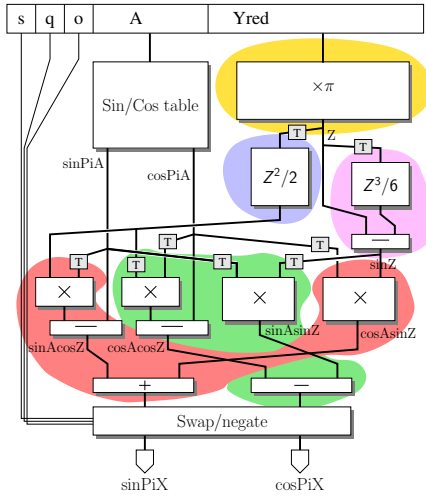
When you have a good hammer, you see nails everywhere

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

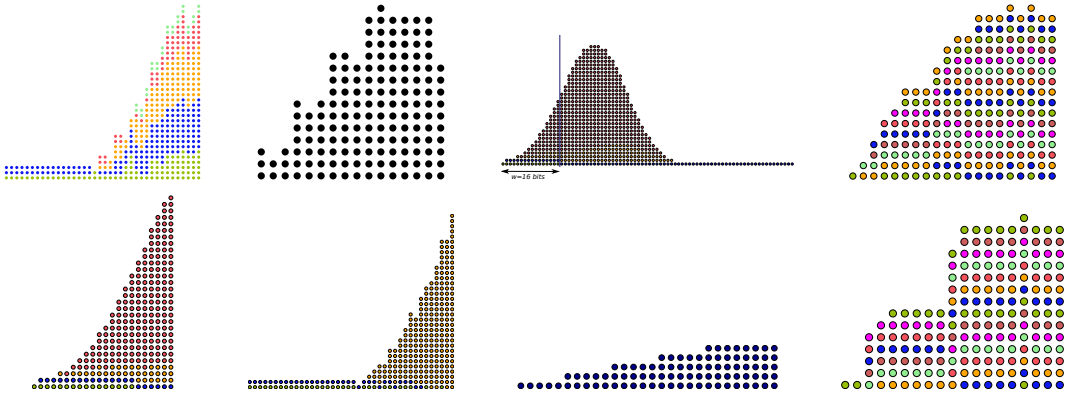


When you have a good hammer, you see nails everywhere

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”?

Conclusion: the FloPoCo project

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 !

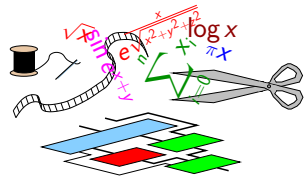
You don't have to, it is my job

And it is a very comfortable niche

- There is an infinite list of operators to keep me busy until retirement.
- They are small arithmetic objects, relatively technology-independent.

The FloPoCo project

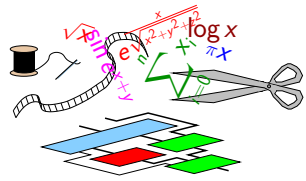
`http://flopoco.org/`



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

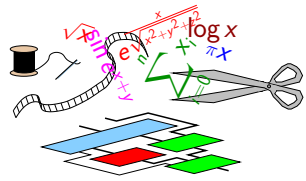
The FloPoCo project

`http://flopoco.org/`



- 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.org/`

- 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 *DSP 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 *DSP 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

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 *DSP 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.org/`