FPL 2024

Fantastic arithmetic beasts and where to find them

Florent de Dinechin

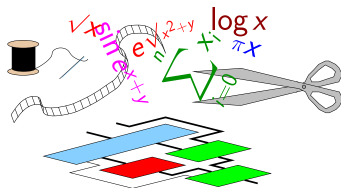
Bogdan Pasca

Where to find them? In your application...

Where to find them? In your application...

... but then you will need some help to bring them to light.

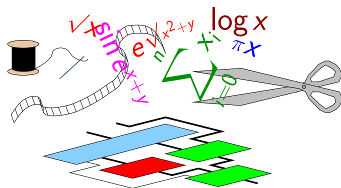
www.flopoco.org



Where to find them? In your application...

... but then you will need some help to bring them to light.

www.flopoco.org



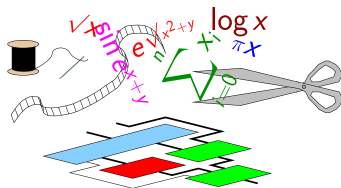
A generator of **application-specific** hardware arithmetic operators

- written in C++, outputting VHDL
- open and extensible

Where to find them? In your application...

... but then you will need some help to bring them to light.

www.flopoco.org



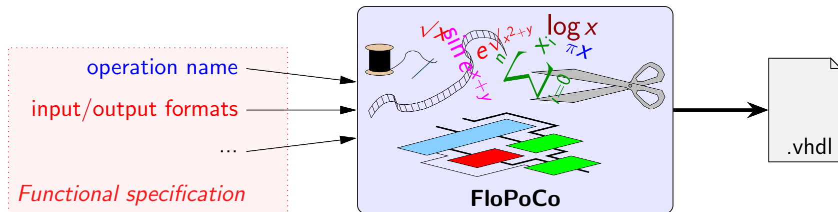
A generator of application-specific hardware arithmetic operators

- written in C++, outputting VHDL
- open and extensible

A philosophy of computing just right

- Interface: You ask for 17 bits, you get 17 *correct* bits.
- Inside: (try to) never compute bits that are not useful to the final result

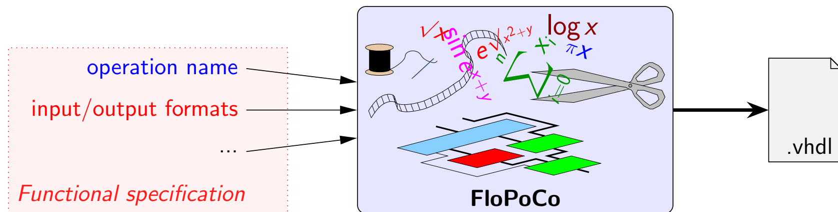
From the boring domestic animals ...



Everybody likes a single precision floating-point adder (here combinatorial)

```
./flopoco IEEEFPAdd wE=8 wF=23
```

From the boring domestic animals ...



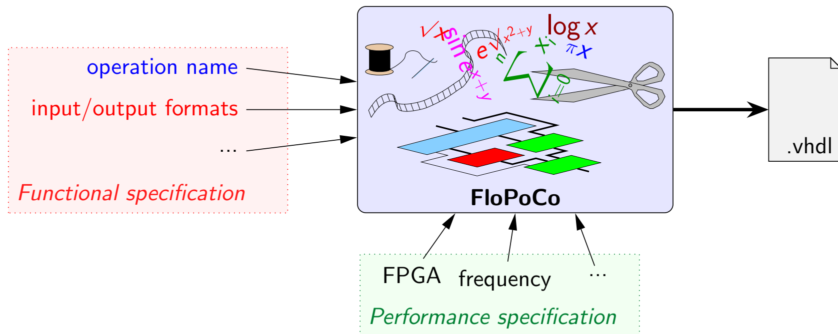
Everybody likes a single precision floating-point adder (here combinatorial)

```
./flopoco IEEEFPAdd wE=8 wF=23
```

... but it has many interesting little brothers...

```
./flopoco FPAdd wE=6 wF=31
```

From the boring domestic animals ...



Everybody likes a single precision floating-point adder (here combinatorial)

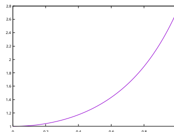
```
./flopoco IEEEFPAdd wE=8 wF=23
```

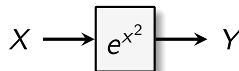
... but it has many interesting little brothers...

```
./flopoco FPAdd wE=6 wF=31 frequency=300 dualpath=true
```

... to fantastic arithmetic beasts

Suppose you need to evaluate some function,
say $e^{(x^2)}$ on $[0, 1)$...



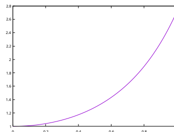


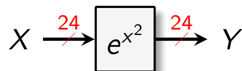
... to fantastic arithmetic beasts

Suppose you need to evaluate some function,

say $e^{(x^2)}$ on $[0, 1)$...

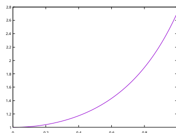
... with inputs and outputs on 24 bits.

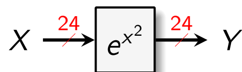




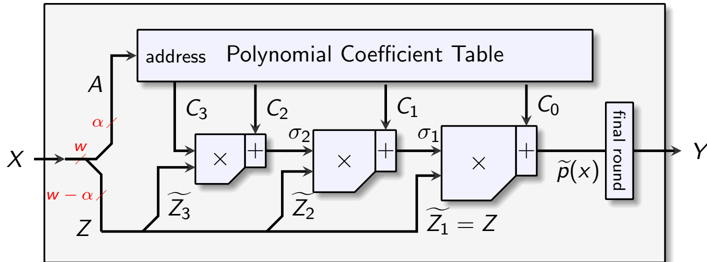
... to fantastic arithmetic beasts

Suppose you need to evaluate some function[1],
say $e^{(x^2)}$ on $[0, 1)$...
... with inputs and outputs on 24 bits.
There are several ways of doing this in FloPoCo.
Here is one of them.






```
./flopoco FixFunctionByPiecewisePoly f="exp(x*x)" lsbIn=-24 lsbOut=-24 d=3
```



[1] It works on the set of functions on which it works (TM)

Another one from one of yesterday's talk

Computing $X \bmod 3329$ for X a 24-bit integer:

```
./flopoco IntConstDiv wIn=24 d=3329 computeQuotient=false
    computeRemainder=true arch=3
```

(Not as good as yesterday's paper, though.)

Danila Gorodecky and Leonel Sousa
Scalable architecture of constant division on FPGA.
ARITH, 2023.

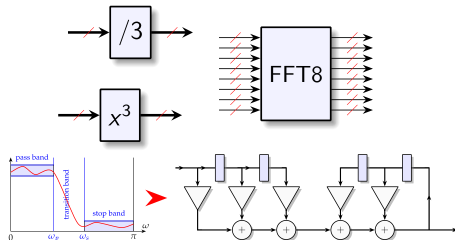
Florent is busy until retirement

We'll see more fantastic beast in this talk.
Not all: there is already an infinite number of them in FloPoCo...
and a larger infinity still to be discovered.

Scope of FloPoCo

Hardware finite-precision implementations
of any computing kernel with a **clear mathematical definition**.

- We have only scratched the surface of function approximation
- We'll see many variants of classical operations
- Coarser kernels such as Fast Fourier Transforms
- From a frequency response to an IIR
- ...



Agenda

Careless PhD students and their pets gone wrong

Fantastic but not evil: circuits computing just right

Fantastic arithmetic beasts escaped to vendor tools

Bit heaps: the mutant biology of arithmetic beasts

Why fantastic arithmetic beasts didn't take over the world (and how to address it)

Backup slides

Careless PhD students and their pets gone wrong

Careless PhD students and their pets gone wrong

Fantastic but not evil: circuits computing just right

Fantastic arithmetic beasts escaped to vendor tools

Bit heaps: the mutant biology of arithmetic beasts

Why fantastic arithmetic beasts didn't take over the world (and how to address it)

Backup slides

All my life, I have been afflicted with very good students

Very good students tend to write kilolines of code...

All my life, I have been afflicted with very good students

Very good students tend to write kilolines of code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$

All my life, I have been afflicted with very good students

Very good students tend to write kilolines of code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...

All my life, I have been afflicted with very good students

Very good students tend to write kilolines of code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- using two novel generic techniques for hardware function approximation
  - multipartite tables
  - HOTBM (higher-order table method)

All my life, I have been afflicted with very good students

Very good students tend to write kilolines of code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{}$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- using two novel generic techniques for hardware function approximation
  - multipartite tables
  - HOTBM (higher-order table method)
- then LNS (Logarithm Number System) operators for good measure

All my life, I have been afflicted with very good students

Very good students tend to write kilolines of code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- using two novel generic techniques for hardware function approximation
  - multipartite tables
  - HOTBM (higher-order table method)
- then LNS (Logarithm Number System) operators for good measure

16 papers, thanks to a solid and well-tested agile development methodology

one paper, one random heap of quick-and-dirty code

All my life, I have been afflicted with very good students

Very good students tend to write kilolines of code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- using two novel generic techniques for hardware function approximation
  - multipartite tables
  - HOTBM (higher-order table method)
- then LNS (Logarithm Number System) operators for good measure

16 papers, thanks to a solid and well-tested agile development methodology

one paper, one random heap of quick-and-dirty code

- All sorts of bits of Java/Python/C++ to generate some of the VHDL

All my life, I have been afflicted with very good students

Very good students tend to write kilolines of code...

Jérémie Detrey's PhD, 2004-2007

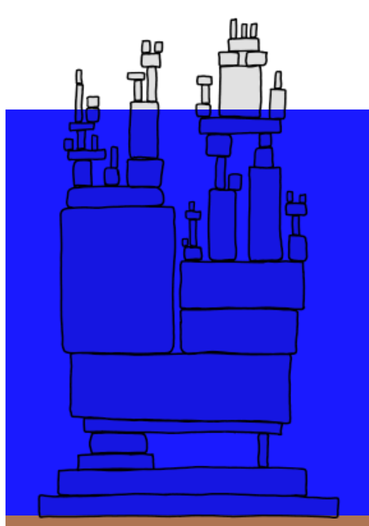
- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- using two novel generic techniques for hardware function approximation
  - multipartite tables
  - HOTBM (higher-order table method)
- then LNS (Logarithm Number System) operators for good measure

16 papers, thanks to a solid and well-tested agile development methodology

one paper, one random heap of quick-and-dirty code

- All sorts of bits of Java/Python/C++ to generate some of the VHDL
- Design-space exploration scripts, test-benches, etc

Our scientific artifacts after Jérémie's PhD

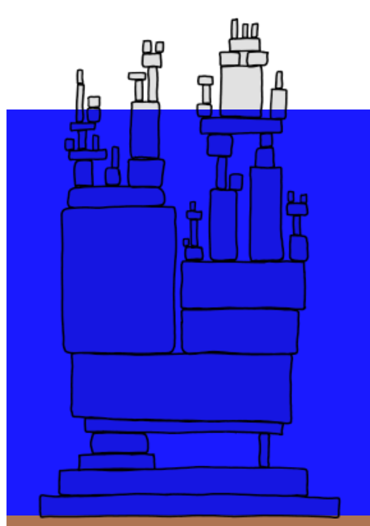


FPLibrary (VHDL available online)

stuff described in Jérémie's PhD

drawing from https://xkcd.com/2347/

Our scientific artifacts after Jérémie's PhD

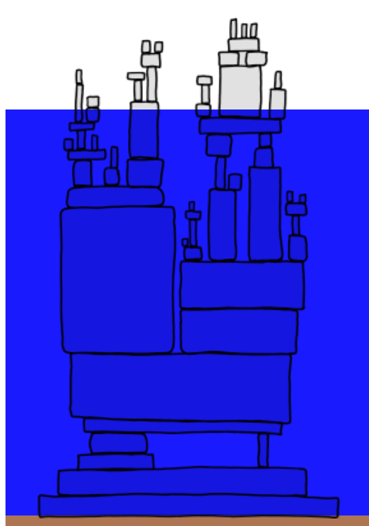


FPLibrary (VHDL available online)

stuff described in Jérémie's PhD
(in French)

drawing from https://xkcd.com/2347/

Our scientific artifacts after Jérémie's PhD



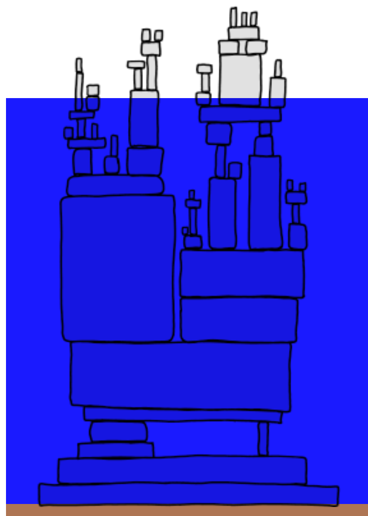
FPLibrary (VHDL available online)

stuff described in Jérémie's PhD
(in French)

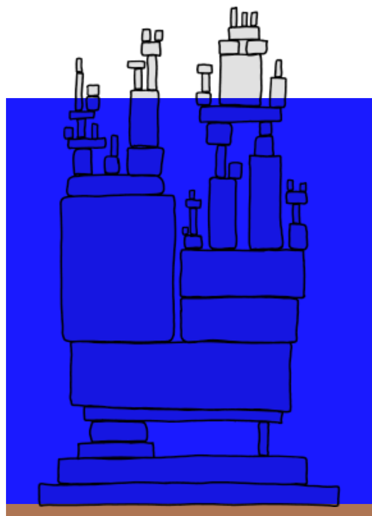
in other words:
work doomed to oblivion when the student leaves
(after his PhD, Jérémie defected to finite-field arithmetic)

drawing from https://xkcd.com/2347/

Hence an Engineering Grand Plan



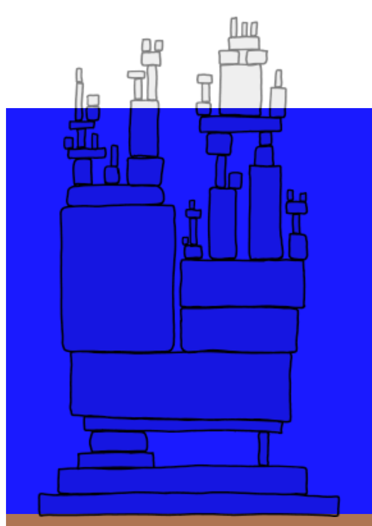
Hence an Engineering Grand Plan



Rewrite this from scratch,
and distribute it

and it shall be called FloPoCo:
Floating-**Po**int **Co**res (but not only)

Hence an Engineering Grand Plan

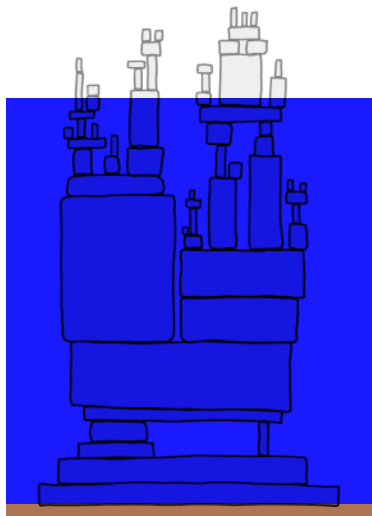


VHDL is generated,
no need to distribute it

Rewrite this from scratch,
and distribute it

and it shall be called FloPoCo:
Floating-**Po**int **Co**res (but not only)

Hence an Engineering Grand Plan



VHDL is generated,
no need to distribute it

Rewrite this from scratch,
and distribute it

and it shall be called FloPoCo:
Floating-**Po**int **Co**res (but not only)

OK, it doesn't really look like a winning move...
but wait a bit.
(and I need to hire a Really Good PhD student)

Historical excuses for all the bad technical choices

- It had to be **C++** because Jérémie had written HOTBM in C++

  (and the thing is, at the time, I didn't understand half of it)

Historical excuses for all the bad technical choices

- It had to be **C++** because Jérémie had written HOTBM in C++

  (and the thing is, at the time, I didn't understand half of it)
- **Generating VHDL** because FPLibrary was written in VHDL

  (and to be frank, quite a lot of it was magical to me)

  First version of FloPoCo was a superset of FPLibrary... by printing out FPLibrary code

Historical excuses for all the bad technical choices

- It had to be **C++** because Jérémie had written HOTBM in C++

  (and the thing is, at the time, I didn't understand half of it)

- **Generating VHDL** because FPLibrary was written in VHDL

  (and to be frank, quite a lot of it was magical to me)

First version of FloPoCo was a superset of FPLibrary... by printing out FPLibrary code

A ~~stupid~~ ~~primitive~~ modest approach to hardware generation, but immediate benefits

- better scaling, easier debugging than parametric VHDL

  when you have many parameters

  - instead of a VHDL `generate if`, a C++ `if` $\Longrightarrow$ only the `true` branch in the VHDL
  - same for `generate for` loops
  - (compared to Jérémie's parametric recursive VHDL for tree-like structures)

Historical excuses for all the bad technical choices

- It had to be **C++** because Jérémie had written HOTBM in C++

  (and the thing is, at the time, I didn't understand half of it)
- **Generating VHDL** because FPLibrary was written in VHDL

  (and to be frank, quite a lot of it was magical to me)

First version of FloPoCo was a superset of FPLibrary... by printing out FPLibrary code

A ~~stupid~~ ~~primitive~~ modest approach to hardware generation, but immediate benefits

- better scaling, easier debugging than parametric VHDL

  when you have many parameters
  - instead of a VHDL `generate if`, a C++ `if` $\Longrightarrow$ only the `true` branch in the VHDL
  - same for `generate for` loops
  - (compared to Jérémie's parametric recursive VHDL for tree-like structures)
- and very soon: **automatic pipelining** – because each submitted paper stated:

  "the design will be pipelined in the final version"

  and this is a perfect waste of good student's time

The engineering foundations to a Scientific Grand Plan

First written in this paper

When FPGAs are better at floating-point than microprocessors

The engineering foundations to a Scientific Grand Plan

First written in this paper

When FPGAs are better at floating-point than microprocessors

- When? As soon as the processor lacks hardware support:

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

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

Dura Amdahl lex, sed lex.

- but also fused operations such as $\sqrt{x^2 + y^2}$, and more...
- In my humble opinion, this was a visionary paper: submitted to ISFPGA 2008

First written in this paper

When FPGAs are better at floating-point than micro

- When? As soon as the processor lacks hardware supp

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

SPICE
cut fro

Dura A

- but also fused operations such as $\sqrt{x^2+y^2}$, and mo
- In my humble opinion, this was a visionary paper: su
- Tepid reviews (“prove it”, “lack of results”)...
- $\Longrightarrow$ poster

When FPGAs are better at floating-point than microprocessors

Florent de Dinechin, Jérémie Detrey, projet Arénaire, Université de Lyon
Octavian Creţ, Radu Tudoran, Universitatea Tehnica Cluj - Napoca

FPGAs as floating-point accelerators?

IEEE-754-compatible $+, -, \times, \div, \sqrt{}$

⊕ Massive parallelism

⊖ Slower by a factor 10

⟶ No match to GPGPU, ClearSpeed, ...

Exotic precision, exotic operators

⊕ Flexibility in precision: Never compute more accurate than needed

⊕ Hardware algorithms for operators implemented in software in processors.

- Demonstrated on elementary functions: Virtex II provides 10× the througphput of Pentium on double-precision exp and log.

FPGAs can compute a much more accurate result result much faster.

A case study

Inductance computing

Computation core:

- in a loop nest of depth 8
- x_i, y_i, z_i: 3D cartesian coordinates read from a RAM
- $t \in [0..1[$ integration variable
- Matlab double-precision reference implementation

Floating-point to fixed-point conversion

- x_i, y_i, z_i: 12-bit fixed-point resolution
- t takes 10 values, replace it with a 3-bit number

→ keep all additions and multiplications exact

→ result 24/30-bit

- square root then identical to floating-point one.
- *more accurate result, obtained faster*

An ad-hoc datapath for v_2

Large accumulator

Accumulation of FP numbers

A typical FP adder

Large accumulator (top) and post-normalisation unit (bottom).

The FloPoCo project

Provide FPGA operators that processors have to implement in software

- Multiplication by a constant
- Elementary functions (sine, exponential, logarithm...)
- Algebraic functions ($1/x$, $\sqrt{x}$, polynomials, ...)
- Compound functions ($\log_2(1 \pm 2^x)$, e^{-Kt^2}, ...)
- Interval arithmetic
- ...

Example: exact FP add and sum

Special cases of floating-point addition

The 2Sum and 2Mul operators

Example: FP sine/cosine architecture

Overview *Range reduction* *Sine eval.*

FPGA improvements needed?

No FPUs, thanks

Competition from the GPUs?

Cascadable combined LZOC/shifters as hard blocks some time ?

Shifters and leading zero counters, please

Floating-point hardware = integer hardware (already there) + shifters (for mantissa alignment) + LZOC (for normalization).

Selected publications

[1] N. Brisebarre and J.-M. Muller. **Correctly rounded multiplication by arbitrary precision constants.** In *Proc. 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, June 2005.
[2] J. Detrey and F. de Dinechin. **Floating-point trigonometric functions for FPGAs.** In *Intl Conference on Field-Programmable Logic and Applications*, pages 29–34. IEEE, Aug. 2007.
[3] J. Detrey and F. de Dinechin. **Parameterized floating-point logarithm and exponential functions for FPGAs.** *Microprocessors and Microsystems*, 31(8):537–545, Jun. 2007. Elsevier.
[4] J. Detrey, F. de Dinechin, and X. Pujol. **Return of the hardware floating-point elementary function.** In *18th Symposium on Computer Arithmetic*, pages 161–168. IEEE, June 2007.
[5] D. Goldberg. **What every computer scientist should know about floating-point arithmetic.** *ACM Computing Surveys*, 23(1):5–47, Mar. 1991.
[6] U. Kulisch. **Circuitry for generating scalar products and sums of floating point numbers with maximum accuracy.** United States Patent 4622650, 1986.
[7] T. Ogita, S. M. Rump, and S. Oishi. **Accurate sum and dot product.** *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.
[8] N. Takagi and S. Kuwahara. **A VLSI algorithm for computing the euclidean norm of a 3D vector.** *IEEE Transactions on Computers*, 49(10):1074–1082, 2000.
[9] I. Trestian, O. Creţ, L. Creţ, L. Văcariu, R. Tudoran, and F. de Dinechin. **FPGA-based Computation of the Inductance of Coils Used for the Magnetic Stimulation of the Nervous System** In *Biomedical Electronics and Devices*, IASTED, 2008.

This work was partly supported by the Egide Brancusi programme of the French government, and the XtremeData university programme.

Lack of results, prove it

As we all know, a reviewer is always right.

Therefore, we stubbornly wrote

When FPGAs are better at floating-point than microprocessors

Florent de Dinechin, Jérémie Detrey, projet Arénaire, Université de Lyon
Octavian Creţ, Radu Tudoran, Universitatea Tehnica Cluj - Napoca

FPGAs as floating-point accelerators?

A case study

The FloPoCo project

FPGA improvements needed?

Selected publications

Lack of results, prove it

As we all know, a reviewer is always right.
Therefore, we stubbornly wrote

- An FPL 2009 paper: *Generating high-performance custom floating-point pipelines.*

When FPGAs are better at floating-point than microprocessors

GENERATING HIGH-PERFORMANCE CUSTOM FLOATING-POINT PIPELINES

Florent de Dinechin, Cristian Klein and Bogdan Pasca

LIP (CNRS/INRIA/ENS-Lyon/UCBL), Université de Lyon
École Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon cedex
email: {Florent.de.Dinechin, Cristian.Klein, Bogdan.Pasca}@ens-lyon.fr

ABSTRACT

Custom operators, working at custom precisions, are a key ingredient to fully exploit the FPGA flexibility advantage for high-performance computing. Unfortunately, such operators are costly to design, and application designers tend to rely on less efficient off-the-shelf operators. To address this issue, an open-source architecture generator framework is introduced. Its salient features are an easy learning curve from VHDL, the ability to embed arbitrary synthesizable VHDL code, portability to mainstream FPGA targets from Xilinx and Altera, automatic management of complex pipelines with support for frequency-directed pipeline, and automatic test-bench generation. This generator is presented around the simple example of a collision detector, which it significantly improves in accuracy, DSP count, logic usage, frequency and latency with respect to an implementation using standard floating-point operators.

1. INTRODUCTION

1.1. Flexible operators for FPGAs

FPGA-based coprocessors are available from a variety of vendors and may be used for accelerating intensive computations, including floating-point (FP) ones. On matrix multiplication, their floating-point performance barely surpasses that of a contemporary processor [1], using tens of operators on the FPGA to compensate their much slower frequency. Besides, FPGAs are in competitions with GPUs here. There is much more acceleration potential in operations for which processors or GPUs have no dedicated hardware, for instance double-precision elementary functions [2] or special functions used in the generation of floating-point random numbers [3]. Besides, operators can also be *specialized* in FPGAs. For example, a squarer requires less logic than a multiplier [4]; A floating-point multiplication by the constant 2.0 boils down to adding one to the exponent (a 12-bit addition for double-precision), and need not use a full-blown FP multiplier as it does in a processor. Indeed it is possible to build an optimized architecture for any multiplication by a constant [5]. Finally, and building upon the previous optimizations, operators can be *fused* on an FPGA [6]. The running example illustrating this article is such a fused operator, a collision detection predicate, testing if the norm of a 3D vector is larger than some threshold.

The recipe for implementing such operators is to respect their floating-point interface, but perform most internal computations in fixed point for which FPGAs are highly optimized. This recipe is also used in block-floating floating-point approaches [7] and in recent experimental C-to-FPGA datapath synthesis tools [6].

There are many more opportunities for floating-point on FPGAs [8]. The goal of the FloPoCo project[1] is to study and develop such FPGA-specific **Flo**ating-**Po**int **Co**res. FloPoCo is both a software framework and a generator of arithmetic cores using this framework. It currently offers about 20 operators, from simple ones such as shifters or integer adders to very complex ones such as floating-point exponential and logarithm. Some of the original operators have been described elsewhere [9, 2, 5, 10], and this article presents the framework itself. FloPoCo is distributed under the LGPL, and interested readers are welcome to try it, use it and improve it.

1.2. The arithmetic context

Our definition of an arithmetic *operator* is much broader than the usual $+$, $-$, $\times$ and $/$. In this work, an arithmetic operator is the implementation of any useful mathematical function, be it $\ln(-\ln(x))$ [3] or $\frac{x}{\sqrt{x^2+y^2+z^2}}$. Although very general, this mathematical definition has several important consequences when designing an operator.

Firstly, as the implementation of a mathematical function, an operator usually involves no feedback loop or state machine[2]. This restriction enables a simple framework for

This work was partly supported by the XtremeData university programme and the ANR EVAFlo and TCHATER projects.

[1] www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

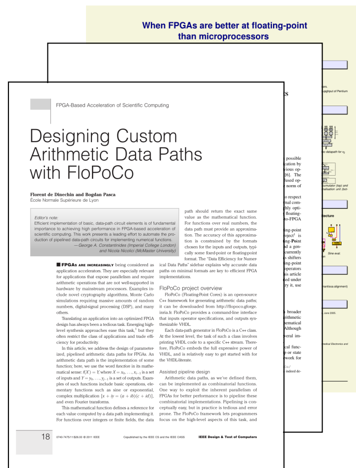
[2] The only current exception is an accumulator [10] which is indeed designed with a minimal, well isolated feedback loop.

978-1-4244-3892-1/09/$25.00 ©2009 IEEE

Lack of results, prove it

As we all know, a reviewer is always right.
Therefore, we stubbornly wrote

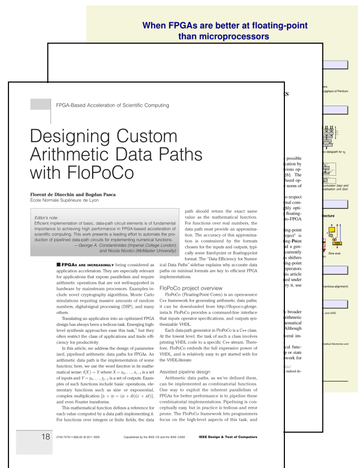
- An FPL 2009 paper:
  Generating high-performance custom floating-point pipelines.
- Its journal version *Designing custom arithmetic data paths with FloPoCo.*
  IEEE Design & Test of Computers, 2011.



Lack of results, prove it

As we all know, a reviewer is always right.
Therefore, we stubbornly wrote

- An FPL 2009 paper:
  Generating high-performance custom floating-point pipelines.
- Its journal version *Designing custom arithmetic data paths with FloPoCo.*
  IEEE Design & Test of Computers, 2011.
  - 400+ GScholar citations,
    TCFPGA Hall of Fame this year


When FPGAs are better at floating-point than microprocessors

FPGA-Based Acceleration of Scientific Computing

Designing Custom Arithmetic Data Paths with FloPoCo

Florent de Dinechin and Bogdan Pasca
École Normale Supérieure de Lyon

Editor's note:
Efficient implementation of basic, data-path circuit elements is of fundamental importance to achieving high performance in FPGA-based acceleration of scientific computing. This work presents a leading effort to automate the production of pipelined data-path circuits for implementing numerical functions.
—George A. Constantinides (Imperial College London) and Nicola Nicolici (McMaster University)

FPGAs are increasingly being considered as application accelerators. They are especially relevant for applications that expose parallelism and require arithmetic operations that are not well-supported in hardware by mainstream processors. Examples include novel cryptography algorithms, Monte Carlo simulations requiring massive amounts of random numbers, digital-signal processing (DSP), and many others.

Translating an application into an optimized FPGA design has always been a tedious task. Emerging high-level synthesis approaches ease this task,[1] but they often restrict the class of applications and trade efficiency for productivity.

In this article, we address the design of parameterized, pipelined arithmetic data paths for FPGAs. An arithmetic data path is the implementation of some function; here, we use the word *function* in its mathematical sense: $f(X) = Y$, where $X = x_0, \ldots, x_{i-1}$ is a set of inputs and $Y = y_0, \ldots, y_{j-1}$ is a set of outputs. Examples of such functions include basic operations, elementary functions such as sine or exponential, complex multiplication $[x + iy = (a + ib)(c + id)]$, and even Fourier transforms.

This mathematical function defines a reference for each value computed by a data path implementing it. For functions over integers or finite fields, the data path should return the exact same value as the mathematical function. For functions over real numbers, the data path must provide an approximation. The accuracy of this approximation is constrained by the formats chosen for the inputs and outputs, typically some fixed-point or floating-point format. The "Data Efficiency for Numerical Data Paths" sidebar explains why accurate data paths on minimal formats are key to efficient FPGA implementations.

FloPoCo project overview

FloPoCo (Floating-Point Cores) is an open-source C++ framework for generating arithmetic data paths; it can be downloaded from http://flopoco.gforge.inria.fr. FloPoCo provides a command-line interface that inputs operator specifications, and outputs synthesizable VHDL.

Each data-path generator in FloPoCo is a C++ class. At the lowest level, the task of such a class involves printing VHDL code to a specific C++ stream. Therefore, FloPoCo embeds the full expressive power of VHDL, and is relatively easy to get started with for the VHDL-literate.

Assisted pipeline design

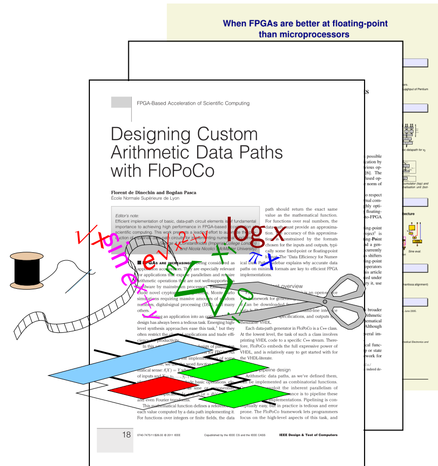
Arithmetic data paths, as we've defined them, can be implemented as combinatorial functions. One way to exploit the inherent parallelism of FPGAs for better performance is to pipeline these combinatorial implementations. Pipelining is conceptually easy, but in practice is tedious and error prone. The FloPoCo framework lets programmers focus on the high-level aspects of this task, and

18 0740-7475/11/$26.00 © 2011 IEEE Copublished by the IEEE CS and the IEEE CASS **IEEE Design & Test of Computers**

Lack of results, prove it

As we all know, a reviewer is always right.
Therefore, we stubbornly wrote

- An FPL 2009 paper:
  Generating high-performance custom floating-point pipelines.
- Its journal version *Designing custom arithmetic data paths with FloPoCo.*
  IEEE Design & Test of Computers, 2011.
  - 400+ GScholar citations,
    TCFPGA Hall of Fame this year
  - How many people actually read it?
    It is the "how to cite" paper for FloPoCo



Lack of results, prove it

As we all know, a reviewer is always right.
Therefore, we stubbornly wrote

- An FPL 2009 paper:
  Generating high-performance custom floating-point pipelines.
- Its journal version *Designing custom arithmetic data paths with FloPoCo.*
  IEEE Design & Test of Computers, 2011.
  - 400+ GScholar citations,
    TCFPGA Hall of Fame this year
  - How many people actually read it?
    It is the "how to cite" paper for FloPoCo
- And finally, an **800-page** book:
  Application-Specific Arithmetic. Springer, 2024.

Refinement of the Grand Plan

If the final title of your PhD is the same as it was when you started,

your research is probably boring.

When FPGAs are better at floating-point than microprocessors

Refinement of the Grand Plan

If the final title of your PhD is the same as it was when you started,

your research is probably boring.

When FPGAs are better at floating-point than microprocessors

Not your neighbour's FPU

Refinement of the Grand Plan

If the final title of your PhD is the same as it was when you started,

your research is probably boring.

When FPGAs are better at floating-point than microprocessors

Not your neighbour's FPU

FPGA-specific arithmetic *(floating-point, but not only)*

Refinement of the Grand Plan

If the final title of your PhD is the same as it was when you started,

your research is probably boring.

When FPGAs are better at floating-point than microprocessors

Not your neighbour's FPU

FPGA-specific arithmetic *(floating-point, but not only)*

All the operators you will never see in a processor (and how to build them)

Refinement of the Grand Plan

If the final title of your PhD is the same as it was when you started,

your research is probably boring.

When FPGAs are better at floating-point than microprocessors

Not your neighbour's FPU

FPGA-specific arithmetic *(floating-point, but not only)*

All the operators you will never see in a processor (and how to build them)

Save routing! Save power! Don't move around useless bits!

Refinement of the Grand Plan

If the final title of your PhD is the same as it was when you started,

your research is probably boring.

When FPGAs are better at floating-point than microprocessors

Not your neighbour's FPU

FPGA-specific arithmetic *(floating-point, but not only)*

All the operators you will never see in a processor (and how to build them)

Save routing! Save power! Don't move around useless bits!

Application-specific arithmetic *(FPGA, but not only)*

Refinement of the Grand Plan

If the final title of your PhD is the same as it was when you started

y

When FPGAs are better at floating-point than micropr

Not your neighbour's FPU

FPGA-specific arithmetic

All the operators you will never see in a processor (and

Save routing! Save power! Don't move around useless

Application-specific arithmetic

Circuits computing just right

Message to my younger self

If you believe in an idea, stick to it, whatever the reviews say.

(bad reviews just mean your good idea was badly explained...)

Fantastic but not evil: circuits computing just right

Careless PhD students and their pets gone wrong

Fantastic but not evil: circuits computing just right

Fantastic arithmetic beasts escaped to vendor tools

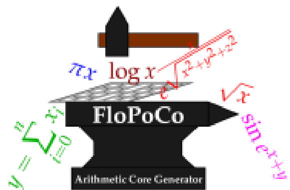
Bit heaps: the mutant biology of arithmetic beasts

Why fantastic arithmetic beasts didn't take over the world (and how to address it)

Backup slides

Computing just right?

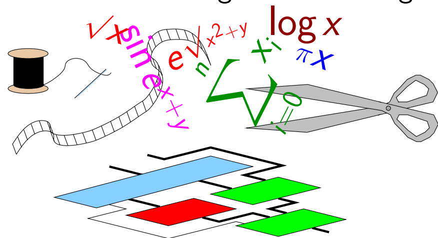
Bogdan said “great project, we need a logo” and designed this:



It was initially OK, but... soon we were using a *delicate chisel* more than a **hammer**.

Computing just right?

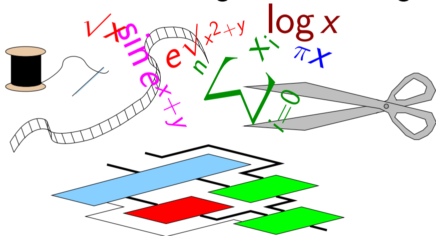
So as soon as Bogdan left I designed this:



(the proper term is probably *allogory*)

Computing just right?

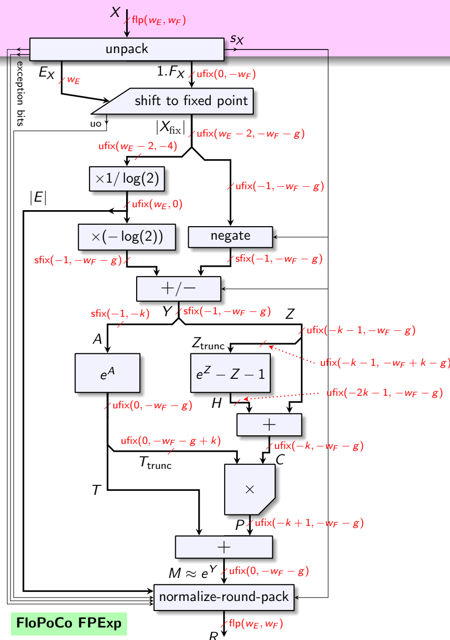
So as soon as Bogdan left I designed this:



(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 resources! Save power! Don't move useless bits around!

In software, as soon as your result is correct, it is probably wasteful

Gustafson: Does *Angry birds* really need single precision (8 decimal digits of accuracy) considering that the trajectory was input using your fat fingers?

Save resources! Save power! Don't move useless bits around!

In software, as soon as your result is correct, it is probably wasteful

Gustafson: Does *Angry birds* really need single precision (8 decimal digits of accuracy) considering that the trajectory was input using your fat fingers?

Plain common sense

- 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.

In FPGAs, we have this freedom.

Save resources! Save power! Don't move useless bits around!

In software, as soon as your result is correct, it is probably wasteful

Gustafson: Does *Angry birds* really need single precision (8 decimal digits of accuracy) considering that the trajectory was input using your fat fingers?

Plain common sense

- 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.

In FPGAs, we have this freedom.

With great freedom come great opportunities

- 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**

Opportunity #1: Computing Just Right makes interfaces simpler

Output format (number of output bits) specifies operator accuracy

The output format defines a quantum of precision (u or ulp for “unit in the last place”)

Opportunity #1: Computing Just Right makes interfaces simpler

Output format (number of output bits) specifies operator accuracy

The output format defines a quantum of precision (u or ulp for "unit in the last place")

- No need to compute more accurately than u: we couldn't output it
- No sense in computing less accurately than u: we don't want to output garbage bits

Opportunity #1: Computing Just Right makes interfaces simpler

Output format (number of output bits) specifies operator accuracy

The output format defines a quantum of precision (u or ulp for "unit in the last place")

- No need to compute more accurately than u: we couldn't output it
- No sense in computing less accurately than u: we don't want to output garbage bits

Inspired by IEEE-754

- define quantization(x) for $x \in \mathbb{R}$.
- define operator(X) = quantization(operation(X))

Example: round to the nearest.

Opportunity #1: Computing Just Right makes interfaces simpler

Output format (number of output bits) specifies operator accuracy

The output format defines a quantum of precision (u or ulp for "unit in the last place")

- No need to compute more accurately than u: we couldn't output it
- No sense in computing less accurately than u: we don't want to output garbage bits

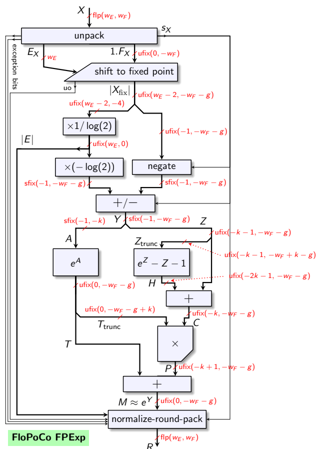
Inspired by IEEE-754

- define quantization(x) for $x \in \mathbb{R}$.
- define operator(X) = quantization(operation(X))

Example: round to the nearest.

If you add one bit to the output, you divide u by 2, hence double the accuracy.

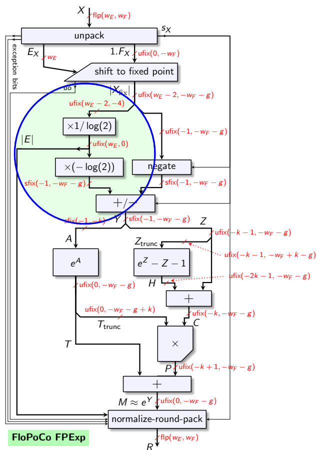
Computing just right $\iff$ over-parameterization



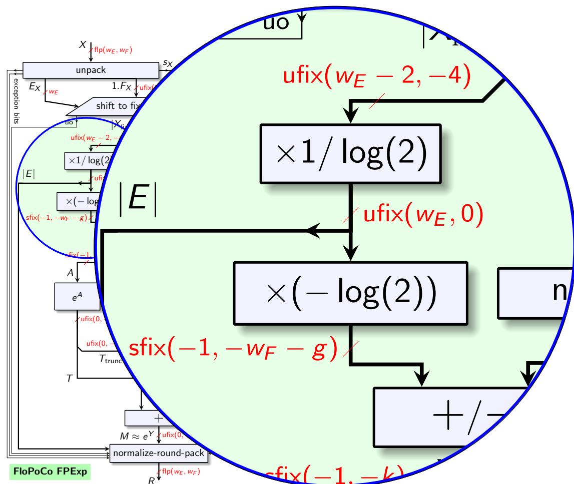
Example:

Computing just right $\iff$ over-parameterization

Example:

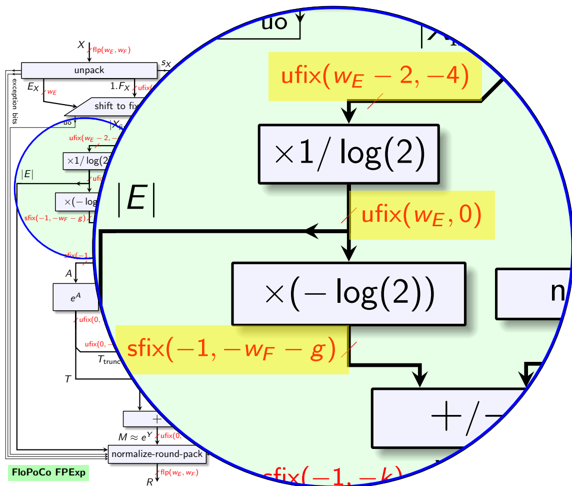


Computing just right $\iff$ over-parameterization



Example:

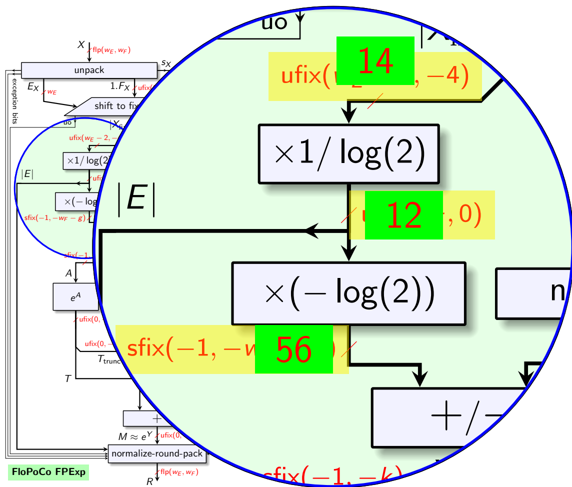
Computing just right $\Longleftrightarrow$ over-parameterization



Example:

Multipliers of all shapes and sizes

Computing just right $\iff$ 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

Not your neighbour's multiplier.
Very strange arithmetic beasts.

Over-parameterization sounds a bit like over-engineering, doesn't it?

- OK, there is a bit more work involved in designing a parametric operator
  - To start with, it must be a hardware-generating program:
    There is an infinite number of multipliers-by-a-constant.
    You cannot chain them all in a library.

Over-parameterization sounds a bit like over-engineering, doesn't it?

- ⊖ OK, there is a bit more work involved in designing a parametric operator
  - To start with, it must be a hardware-generating program:
    There is an infinite number of multipliers-by-a-constant.
    You cannot chain them all in a library.

- ⊕ Direct benefit to end-users: freedom of choice, application-specific, etc.

Over-parameterization sounds a bit like over-engineering, doesn't it?

- ⊖ OK, there is a bit more work involved in designing a parametric operator
  - To start with, it must be a hardware-generating program:
    There is an infinite number of multipliers-by-a-constant.
    You cannot chain them all in a library.

- ⊕ Direct benefit to end-users: freedom of choice, application-specific, etc.

- ⊕ More future-proof when the target hardware changes

Over-parameterization sounds a bit like over-engineering, doesn't it?

⊖ OK, there is a bit more work involved in designing a parametric operator

- To start with, it must be a hardware-generating program:
  There is an infinite number of multipliers-by-a-constant.
  You cannot chain them all in a library.

⊕ Direct benefit to end-users: freedom of choice, application-specific, etc.

⊕ More future-proof when the target hardware changes

⊕ **It actually simplifies the design of composite operators** (e.g. the exponential)!

- You don't know how many bits on this wire make sense? Keep it open as a parameter.
- Then experiment: estimate cost and accuracy as a function of the parameters
- Then program the choice of the best parameter values,
  e.g. using ILP or common sense (whichever gives the best results)

Opportunity #2: Operator specialization

Not really fantastic arithmetic beasts... just the usual ones with special disabilities.

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$; $8721X = ((17X) \ll 9) + 17X$
  - 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

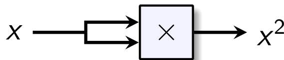
Not really fantastic arithmetic beasts... just the usual ones with special disabilities.

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$; $8721X = ((17X) \ll 9) + 17X$
  - 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

Not really fantastic arithmetic beasts... just the usual ones with special disabilities.

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$; $8721X = ((17X) \ll 9) + 17X$
  - 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

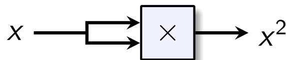



```
       321
 ×     321
 ---------
       321
      642
     963
 ---------
    103041
```

Opportunity #2: Operator specialization

Not really fantastic arithmetic beasts... just the usual ones with special disabilities.

- Multiplication by a constant
  - multiplication by integers: $17X = (X \ll 4) + X$; $8721X = ((17X) \ll 9) + 17X$
  - 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

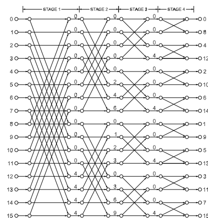


- Specialization of elementary functions to specific domains
- ...

```
      321
×     321
---------
      321
     642
    963
---------
   103041
```

A FloPoCo-enabled success story

Mario Garrido, Konrad Möller, and Martin Kumm.
World's Fastest FFT Architectures: Breaking the Barrier of 100 GS/s.
IEEE Transactions on Circuits and Systems I, 66(4):1507–1516, 2019.



- Fully unrolled FFT (up to 256 points)
  - i.e. inputting 256 complex values per cycle, at 500 MHz
  - well above 10 TOp/s if you count all additions and multiplications
- 16-bit in/out, wider datapath inside
- Look, Ma: no multiplier !
  - each multiplier expanded as an adder graph (and optimally so)
- about 1/5th of LUT + registers of the target device (Virtex UltraScale 190)
  - ... leaving the 1800 DSP blocks free for more interesting things.

A good start, in FPGA design, is not to imitate the processor solution.

Division by 3 is simpler than exponential

TL;DR: multiplying X by 3 is computing $2X + X$;

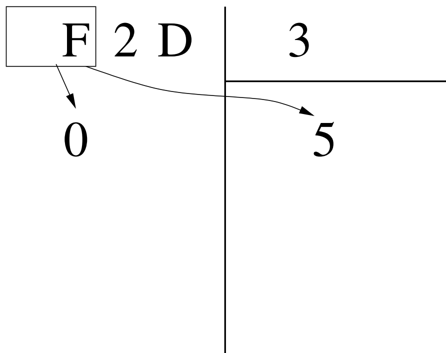
Dividing by 3 should not be much more complex.

Division by 3 is simpler than exponential

TL;DR: multiplying X by 3 is computing $2X + X$;

Dividing by 3 should not be much more complex.

Dividing an **hexadecimal** number by 3

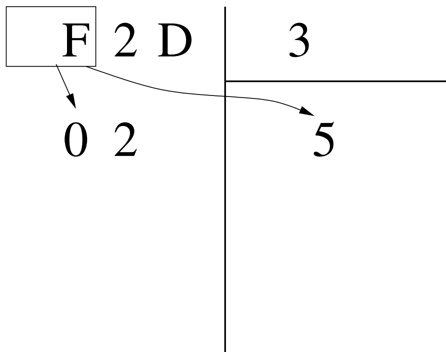


Division by 3 is simpler than exponential

TL;DR: multiplying X by 3 is computing $2X + X$;

Dividing by 3 should not be much more complex.

Dividing an **hexadecimal** number by 3

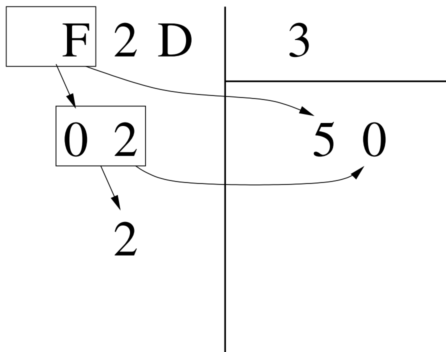


Division by 3 is simpler than exponential

TL;DR: multiplying X by 3 is computing $2X + X$;

Dividing by 3 should not be much more complex.

Dividing an **hexadecimal** number by 3

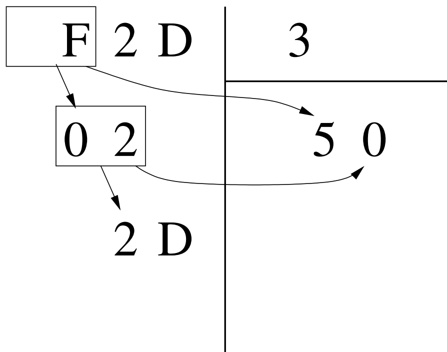


Division by 3 is simpler than exponential

TL;DR: multiplying X by 3 is computing $2X + X$;

Dividing by 3 should not be much more complex.

Dividing an **hexadecimal** number by 3

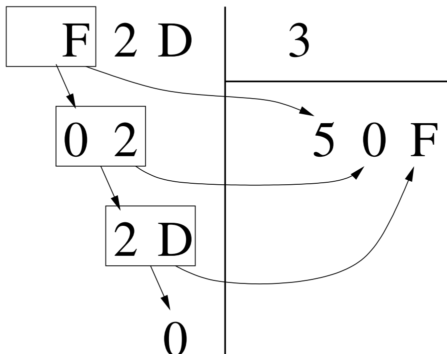


Division by 3 is simpler than exponential

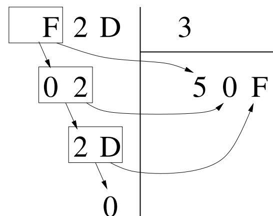
TL;DR: multiplying X by 3 is computing $2X + X$;

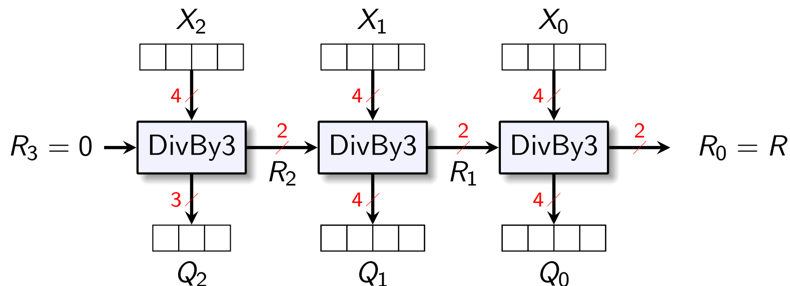
Dividing by 3 should not be much more complex.

Dividing an **hexadecimal** number by 3

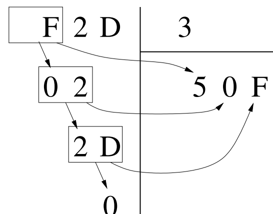


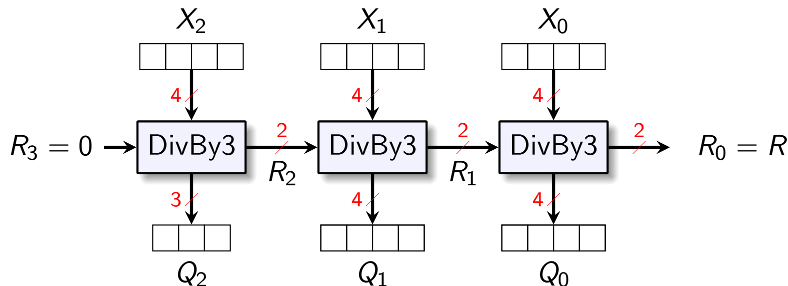
Getting inspiration from the vexations of childhood





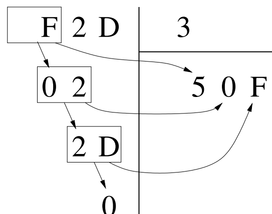
Getting inspiration from the vexations of childhood

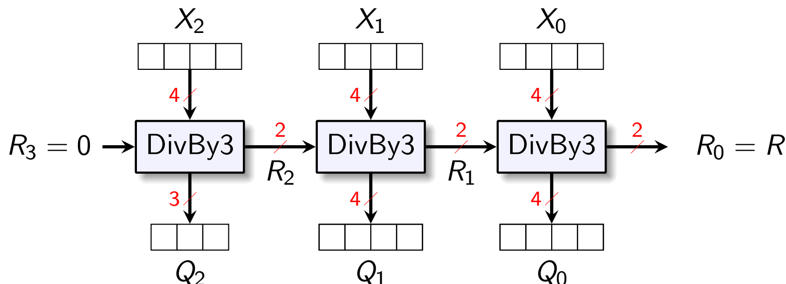




OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

Getting inspiration from the vexations of childhood



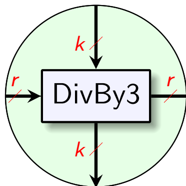


OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

If you're too lazy to compute, then tabulate

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

Opportunity #3: target-specific optimizations

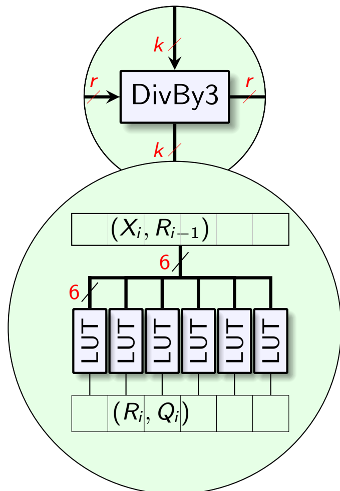


Generalizing hexadecimal to radix 2^k

... 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

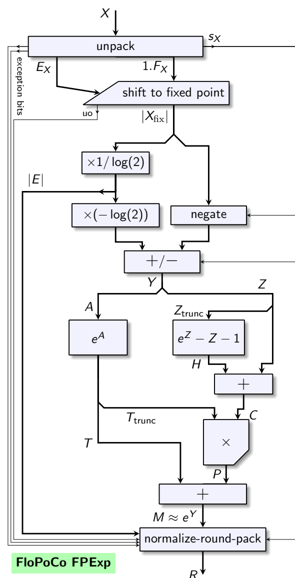
... or, how over-parameterization allows for adaptation

- 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 $\alpha = 6$)

Opportunity #3: target-specific optimizations



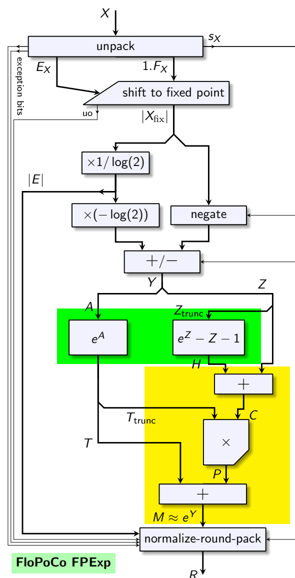
Modern FPGAs also have

Opportunity #3: target-specific optimizations

Modern FPGAs also have

- small multipliers with pre-adders and post-adders

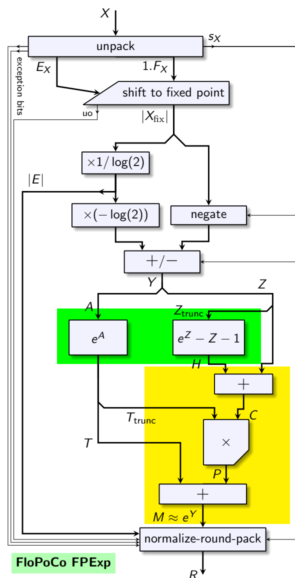
Opportunity #3: target-specific optimizations



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

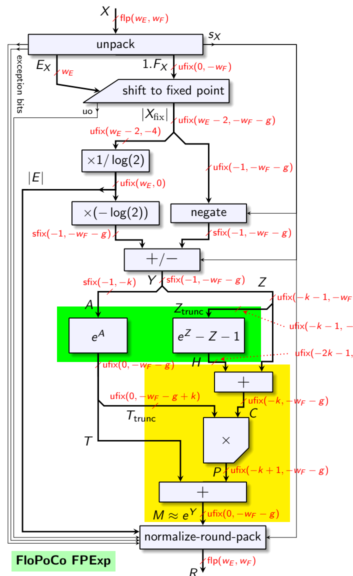
- small multipliers with pre-adders and post-adders
- ... and dual-ported small memories

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

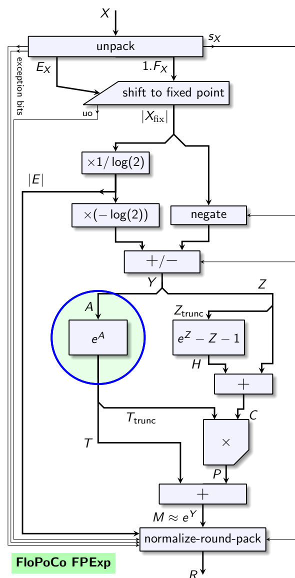
Single-precision accurate exponential on Xilinx

- 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)

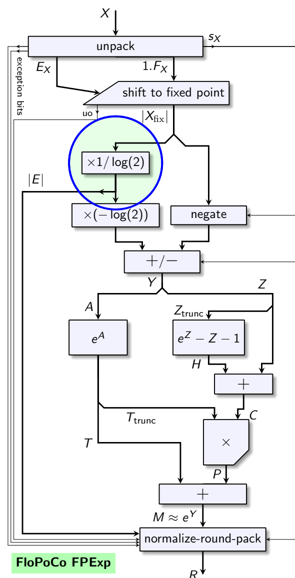
- ... and all the possible exponentials

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

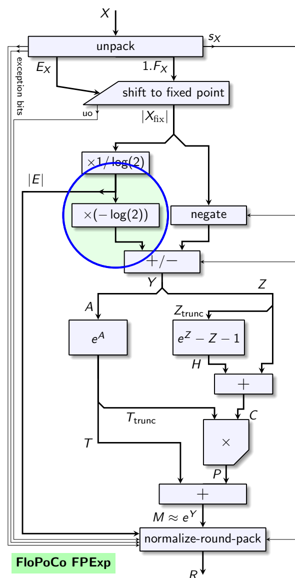


Being unable to trust my reasoning, I learnt by heart the results of all the possible multiplications

(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

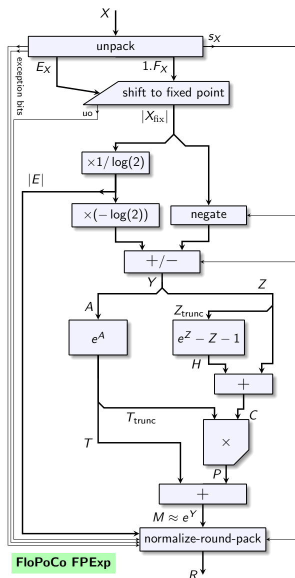


Being unable to trust my reasoning, I learnt by heart the results of all the possible multiplications

(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

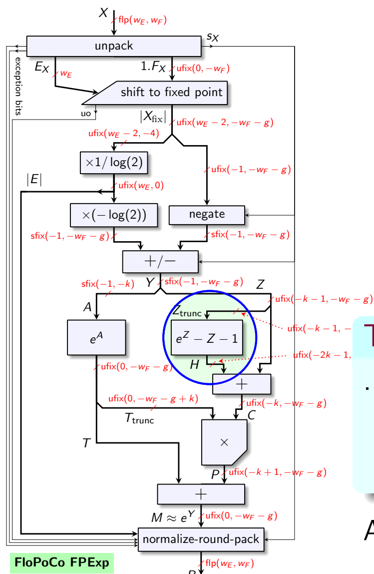


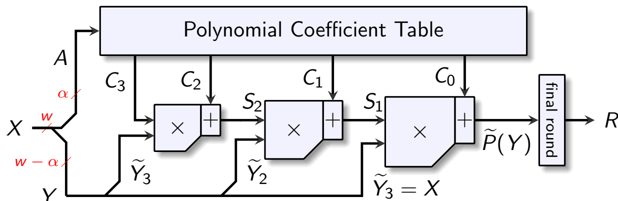
Being unable to trust my reasoning, I learnt by heart
the results of all the possible multiplications
(E. Ionesco)

- ... 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

... has taken us so much time it is well worth a full part.

- 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

… has taken us so much time it is well worth a full part.

Fantastic arithmetic beasts escaped to vendor tools

Careless PhD students and their pets gone wrong

Fantastic but not evil: circuits computing just right

Fantastic arithmetic beasts escaped to vendor tools

Bit heaps: the mutant biology of arithmetic beasts

Why fantastic arithmetic beasts didn't take over the world (and how to address it)

Backup slides

First steps

FloPoCo, PhD and Altera

- 2011 joined Altera European Technology Center
- brought the FloPoCo spirit along
- grafted into the DSP Builder team

First steps

https://agwaycapecod.com/the-art-of-grafting-plants/

FloPoCo, PhD and Altera

- 2011 joined Altera European Technology Center
- brought the FloPoCo spirit along
- grafted into the DSP Builder team
  - model-based design (Matlab Simulink frontend)
  - powerful mapping backend (using WYSIWYG)
  - floating-point support in its infancy

PhD life v.s. Industry

- PhD: highly efficient exponential implementation
- few months: approach analysis, design, implementation, test

PhD life v.s. Industry

- PhD: highly efficient exponential implementation
- few months: approach analysis, design, implementation, test

Floating-point exponential functions for DSP-enabled FPGAs

Florent de Dinechin, Bogdan Pasca

LIP (ENSL-CNRS-Inria-UCBL), École Normale Supérieure de Lyon
46 allée d'Italie, F-69364 Lyon, France
{Florent.de.Dinechin, Bogdan.Pasca}@ens-lyon.fr

***Abstract*—This article presents a generator of floating-point exponential operators targeting recent FPGAs with embedded memories and DSP blocks. A single-precision operator consumes just one DSP block, 18Kbits of dual-port memory, and 392 slices on Virtex-4. For larger precisions, a generic approach based on polynomial approximation is used and proves more resource-efficient than the literature. For instance a double-precision operator consumes 5 BlockRAM and 12 DSP48 blocks on Virtex-5, or 10 M9k and 22 18x18 multipliers on Stratix III. This approach is flexible and is demonstrated to scale up to quadruple-precision, while enabling frequencies close to the FPGA's nominal frequency. All the proposed architectures are last-bit accurate for all the floating-point range. They are available in the open-source FloPoCo framework.**

I. Introduction

The exponential function is, after the basic arithmetic operators, one of the next most useful building block for floating-point applications. On FPGAs, it has been used for scientific or financial Monte-Carlo simulations [1], for SPICE simulation [2], in phylogenetic tree reconstruction, in quantum chemistry simulations, and in the implementation of the power function [3] among others.

A. Previous works

Several publications have described exponential implementations. We list them here, and will discuss in more details the choices they made and their performance impact in Section IV.

Earlier works targetted single precision, first by adapting to FPGAs a software algorithm based on floating-point operations [4], then by using a more efficient fixed-point architecture [5]. This architecture was later improved [1], however the table-based method used there doesn't scale up to double-precision, as the size of the tables grows exponentially with the mantissa size.

As FPGAs are increasingly being used for double-precision floating-point, iterative architectures that scale better [6], [7], [8] were adapted for FPGAs [9]. The architecture in [9] was designed with 5-input LUTs in mind, but is poorly suited to DSP-enabled FPGAs, as IV-B will show. It was parameterized in precision, but to our knowledge was never pipelined. Another pipelined, but double-precision only implementation was proposed in [10], [11].

In [12], a CORDIC-based approach using several parallel CORDIC cores was proposed. It has a complex control including input and output FIFOs. Being radix-2 CORDIC, it computes one digit per iteration and thus has a very long latency. Moreover, it is based on a floating-point adder, whereas CORDIC is inherently a fixed-point computation, so there is probably room for improvement there.

From a user point of view, the current state of the art is probably the floating-point exponential function `ALTFP_EXP` provided with Altera Megawizard since 2008 [13]. This implementation exploits the DSP blocks, is parameterized in exponent and mantissa size, and is fully pipelined. Being included in the standard Quartus releases, it is widely available, although only for Altera targets.

Many other publications have addressed the computation of exponential function in ASIC, e.g. [6], [7], [14], [8]. However, it is difficult to evaluate the relevance of such works on FPGAs.

B. Contributions

In the present article, we propose yet another architecture for the floating-point evaluation of the exponential function, and its implementation in the open-source FloPoCo project [1]. Its main specificities are the following.

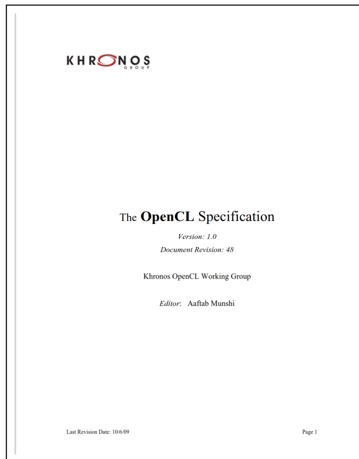
- The algorithm, based on the usual multiplicative range reduction followed by a polynomial approximation, was chosen with DSP blocks and embedded memories in mind, so it makes efficient use of these resources. For instance, the single-precision version now involves just one 17x17-bit multiplier and 18Kbits of dual-port memory, and runs at 375MHz on a Virtex-4, which is a large improvement in all respects over the state of the art [1].
- As we believe that floating-point on FPGA should exploit the flexibility of the target and therefore not be limited to IEEE single and double precision, the algorithm and implementation proposed here are fully parametrized in exponent and mantissa size. They scale to double-precision and beyond.
- The implementation is pipelined to a user-specified frequency. It is last-bit accurate for all supported mantissa sizes.
- The architectures are generated as synthesizable VHDL portable to any FPGA target. In addition, many target-specific optimizations are performed by the FloPoCo framework [15], [16], [17].

978-1-4244-8983-1/10/$26.00 ©2010 IEEE

[1] http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

PhD life v.s. Industry

- OpenCL - first real driver for math.h coverage



KHRONOS GROUP

The **OpenCL** Specification

Version: 1.0

Document Revision: 48

Khronos OpenCL Working Group

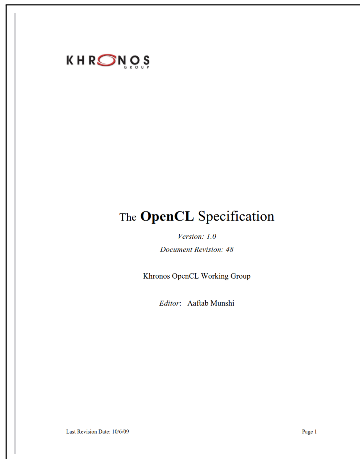
Editor: Aaftab Munshi

Last Revision Date: 10/6/09

Page 1

PhD life v.s. Industry

- OpenCL - first real driver for math.h coverage



KHRONOS GROUP

The **OpenCL** Specification

Version: 1.0

Document Revision: 48

Khronos OpenCL Working Group

Editor: Aaftab Munshi

Last Revision Date: 10/6/09 Page 1

6.5 Address Space Qualifiers ... 150
6.5.1 __global (or global) ... 150
6.5.2 __local (or local) ... 151
6.5.3 __constant (or constant) ... 151
6.5.4 __private (or private) ... 152

6.6 Image Access Qualifiers ... 153

6.7 Function Qualifiers ... 154
6.7.1 __kernel (or kernel) ... 154
6.7.2 Optional Attribute Qualifiers ... 154

6.8 Restrictions ... 157

6.9 Preprocessor Directives and Macros ... 160

6.10 Attribute Qualifiers ... 162
6.10.1 Specifying Attributes of Types ... 163
6.10.2 Specifying Attributes of Functions ... 165
6.10.3 Specifying Attributes of Variables ... 165
6.10.4 Specifying Attributes of Blocks and Control-Flow-Statements ... 167
6.10.5 Extending Attribute Qualifiers ... 167

6.11 Built-in Functions ... 168
6.11.1 Work-Item Functions ... 168
6.11.2 Math Functions ... 170
6.11.2.1 Floating-point macros and pragmas for math.h ... 175
6.11.3 Integer Functions ... 177
6.11.4 Common Functions ... 180
6.11.5 Geometric Functions ... 182
6.11.6 Relational Functions ... 184
6.11.7 Vector Data Load and Store Functions ... 186
6.11.8 Image Read and Write Functions ... 190
6.11.8.1 Samplers ... 190
6.11.8.2 Built-in Image Functions ... 192
6.11.9 Synchronization Functions ... 199
6.11.10 Explicit Memory Fence Functions ... 200
6.11.11 Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch ... 201

7. OPENCL NUMERICAL COMPLIANCE ... 203

7.1 Rounding Modes ... 203

7.2 INF, NaN and Denormalized Numbers ... 203

7.3 Floating-Point Exceptions ... 204

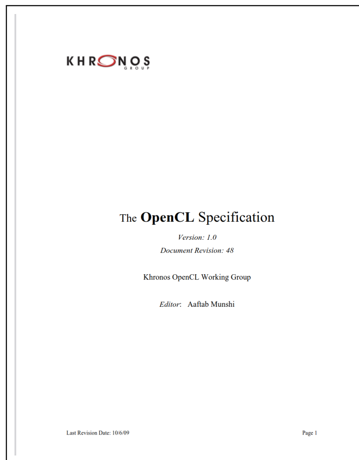
7.4 Relative Error as ULPs ... 204

7.5 Edge Case Behavior ... 207
7.5.1 Additional Requirements Beyond C99 TC2 ... 207
7.5.2 Changes to C99 TC2 Behavior ... 210
7.5.3 Edge Case Behavior in Flush To Zero Mode ... 211

Last Revision Date: 10/6/09 Page 4

PhD life v.s. Industry

- OpenCL - first real driver for math.h coverage



KHRONOS GROUP

The **OpenCL** Specification

Version: 1.0

Document Revision: 48

Khronos OpenCL Working Group

Editor: Aaftab Munshi

Last Revision Date: 10/6/09 Page 1

6.5 Address Space Qualifiers ... **150**
6.5.1 __global (or global) ... 150
6.5.2 __local (or local) ... 151
6.5.3 __constant (or constant) ... 151
6.5.4 __private (or private) ... 152

6.6 Image Access Qualifiers ... **153**

6.7 Function Qualifiers ... **154**
6.7.1 __kernel (or kernel) ... 154
6.7.2 Optional Attribute Qualifiers ... 154

6.8 Restrictions ... **157**

6.9 Preprocessor Directives and Macros ... **160**

6.10 Attribute Qualifiers ... **162**
6.10.1 Specifying Attributes of Types ... 163
6.10.2 Specifying Attributes of Functions ... 165
6.10.3 Specifying Attributes of Variables ... 165
6.10.4 Specifying Attributes of Blocks and Control-Flow-Statements ... 167
6.10.5 Extending Attribute Qualifiers ... 167

6.11 Built-in Functions ... **168**
6.11.1 Work-Item Functions ... 168
6.11.2 Math Functions ... 170
6.11.2.1 Floating-point macros and pragmas for math.h ... 175
6.11.3 Integer Functions ... 177
6.11.4 Common Functions ... 180
6.11.5 Geometric Functions ... 182
6.11.6 Relational Functions ... 184
6.11.7 Vector Data Load and Store Functions ... 186
6.11.8 Image Read and Write Functions ... 190
6.11.8.1 Samplers ... 190
6.11.8.2 Built-in Image Functions ... 192
6.11.9 Synchronization Functions ... 199
6.11.10 Explicit Memory Fence Functions ... 200
6.11.11 Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch ... 201

7. OPENCL NUMERICAL COMPLIANCE ... **203**

7.1 Rounding Modes ... **203**

7.2 INF, NaN and Denormalized Numbers ... **203**

7.3 Floating-Point Exceptions ... **204**

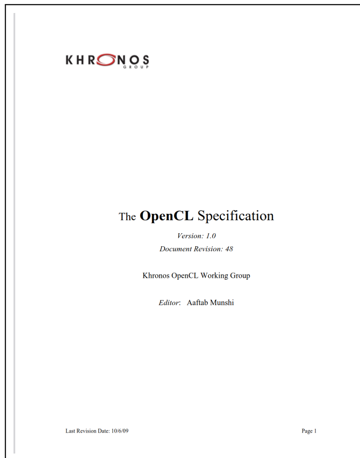
7.4 Relative Error as ULPs ... **204**

7.5 Edge Case Behavior ... **207**
7.5.1 Additional Requirements Beyond C99 TC2 ... 207
7.5.2 Changes to C99 TC2 Behavior ... 210
7.5.3 Edge Case Behavior in Flush To Zero Mode ... 211

Last Revision Date: 10/6/09 Page 4

PhD life v.s. Industry

- OpenCL - first real driver for math.h coverage



KHRONOS GROUP

The **OpenCL** Specification

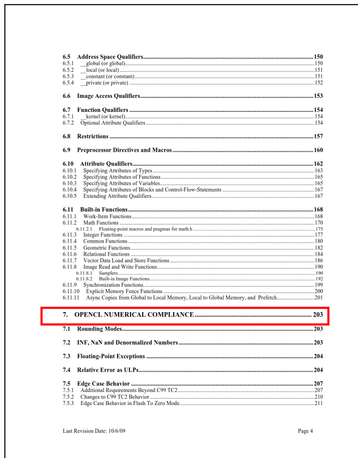
Version: 1.0

Document Revision: 48

Khronos OpenCL Working Group

Editor: Aaftab Munshi

Last Revision Date: 10/6/09 Page 1



6.5 Address Space Qualifiers **150**
6.5.1 __global (or global) 150
6.5.2 __local (or local) 151
6.5.3 __constant (or constant) 151
6.5.4 __private (or private) 152

6.6 Image Access Qualifiers **153**

6.7 Function Qualifiers **154**
6.7.1 __kernel (or kernel) 154
6.7.2 Optional Attribute Qualifiers 154

6.8 Restrictions **157**

6.9 Preprocessor Directives and Macros **160**

6.10 Attribute Qualifiers **162**
6.10.1 Specifying Attributes of Types 163
6.10.2 Specifying Attributes of Functions 165
6.10.3 Specifying Attributes of Variables 165
6.10.4 Specifying Attributes of Blocks and Control-Flow-Statements 167
6.10.5 Extending Attribute Qualifiers 167

6.11 Built-in Functions **168**
6.11.1 Work-Item Functions 168
6.11.2 Math Functions 170
6.11.2.1 Floating-point macros and pragmas for math.h 175
6.11.3 Integer Functions 177
6.11.4 Common Functions 180
6.11.5 Geometric Functions 182
6.11.6 Relational Functions 184
6.11.7 Vector Data Load and Store Functions 186
6.11.8 Image Read and Write Functions 190
6.11.8.1 Samplers 190
6.11.8.2 Built-in Image Functions 192
6.11.9 Synchronization Functions 199
6.11.10 Explicit Memory Fence Functions 200
6.11.11 Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch 201

7. OPENCL NUMERICAL COMPLIANCE **203**

7.1 Rounding Modes **203**

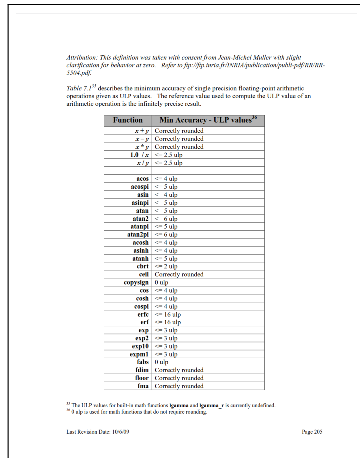
7.2 INF, NaN and Denormalized Numbers **203**

7.3 Floating-Point Exceptions **204**

7.4 Relative Error as ULPs **204**

7.5 Edge Case Behavior **207**
7.5.1 Additional Requirements Beyond C99 TC2 207
7.5.2 Changes to C99 TC2 Behavior 210
7.5.3 Edge Case Behavior in Flush To Zero Mode 211

Last Revision Date: 10/6/09 Page 4



Attribution: This definition was taken with consent from Jean-Michel Muller with slight clarification for behavior at zero. Refer to ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf.

Table 7.1[35] describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

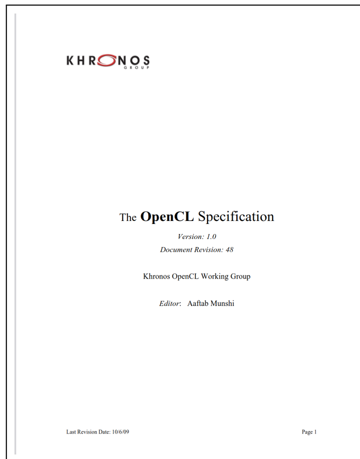
Function	Min Accuracy - ULP values[36]
$x + y$	Correctly rounded
$x - y$	Correctly rounded
$x * y$	Correctly rounded
$1.0 / x$	<= 2.5 ulp
x / y	<= 2.5 ulp
acos	<= 4 ulp
acospi	<= 5 ulp
asin	<= 4 ulp
asinpi	<= 5 ulp
atan	<= 5 ulp
atan2	<= 6 ulp
atanpi	<= 5 ulp
atan2pi	<= 6 ulp
acosh	<= 4 ulp
asinh	<= 4 ulp
atanh	<= 5 ulp
cbrt	<= 2 ulp
ceil	Correctly rounded
copysign	0 ulp
cos	<= 4 ulp
cosh	<= 4 ulp
cospi	<= 4 ulp
erfc	<= 16 ulp
erf	<= 16 ulp
exp	<= 3 ulp
exp2	<= 3 ulp
exp10	<= 3 ulp
expm1	<= 3 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded

[35] The ULP values for built-in math functions **lgamma** and **lgamma_r** is currently undefined.
[36] 0 ulp is used for math functions that do not require rounding.

Last Revision Date: 10/6/09 Page 205

PhD life v.s. Industry

- OpenCL - first real driver for math.h coverage



KHRONOS GROUP

The **OpenCL** Specification

Version: 1.0

Document Revision: 48

Khronos OpenCL Working Group

Editor: Aaftab Munshi

Last Revision Date: 10/6/09 Page 1

6.5 **Address Space Qualifiers** 150
6.5.1 __global (or global) 150
6.5.2 __local (or local) 151
6.5.3 __constant (or constant) 151
6.5.4 __private (or private) 152

6.6 **Image Access Qualifiers** 153

6.7 **Function Qualifiers** 154
6.7.1 __kernel (or kernel) 154
6.7.2 Optional Attribute Qualifiers 154

6.8 **Restrictions** 157

6.9 **Preprocessor Directives and Macros** 160

6.10 **Attribute Qualifiers** 162
6.10.1 Specifying Attributes of Types 163
6.10.2 Specifying Attributes of Functions 165
6.10.3 Specifying Attributes of Variables 165
6.10.4 Specifying Attributes of Blocks and Control-Flow-Statements 167
6.10.5 Extending Attribute Qualifiers 167

6.11 **Built-in Functions** 168
6.11.1 Work-Item Functions 168
6.11.2 Math Functions 170
6.11.2.1 Floating-point macros and pragmas for math.h 175
6.11.3 Integer Functions 177
6.11.4 Common Functions 180
6.11.5 Geometric Functions 182
6.11.6 Relational Functions 184
6.11.7 Vector Data Load and Store Functions 186
6.11.8 Image Read and Write Functions 190
6.11.8.1 Samplers 190
6.11.8.2 Built-in Image Functions 192
6.11.9 Synchronization Functions 199
6.11.10 Explicit Memory Fence Functions 200
6.11.11 Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch 201

7. **OPENCL NUMERICAL COMPLIANCE** 203

7.1 **Rounding Modes** 203

7.2 **INF, NaN and Denormalized Numbers** 203

7.3 **Floating-Point Exceptions** 204

7.4 **Relative Error as ULPs** 204

7.5 **Edge Case Behavior** 207
7.5.1 Additional Requirements Beyond C99 TC2 207
7.5.2 Changes to C99 TC2 Behavior 210
7.5.3 Edge Case Behavior in Flush To Zero Mode 211

Last Revision Date: 10/6/09 Page 4

Hardware Testing
Exhaustive where possible

Pivotal tool for arithmetic function design

Plenty of previous works using hardware polynomial approximation

- most of it hand-tuned for a given function (not generic)
- not accessible (papers, not code)
- heuristics used do not scale to precisions larger than 32 bits

Pivotal tool for arithmetic function design

Plenty of previous works using hardware polynomial approximation

- most of it hand-tuned for a given function (not generic)
- not accessible (papers, not code)
- heuristics used do not scale to precisions larger than 32 bits

Highlights

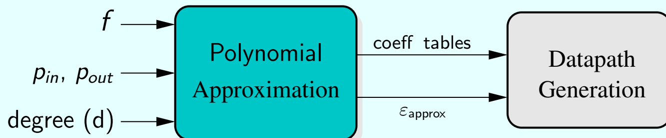
- scales up to **64bits and more**
- state-of-the art polynomial approximations thanks to Sollya
- finer **datapath optimization**
- **pipelined** to a user-specified frequency
- **fully automated** and integrated in open-source **FloPoCo**

How do we do it?

Consider the function $f(x)$, with $x \in [0,1)$ and $IM(f(x)) \in [0,1)$
Approximate it with the polynomial p of degree d **(given)** such that:

$$\varepsilon_{\text{total}} = max|f - p| \leq 2^{-p_{out}} = 1\text{ulp}$$

Two fold process

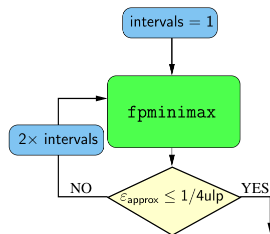


$$\varepsilon_{\text{total}} = \varepsilon_{\text{approx}} + \varepsilon_{\text{eval}} + 1/2\text{ulp} \leq 1\text{ulp}$$

Polynomial approximation

- use modified Remez algorithm from Sollya (`fpminimax`)

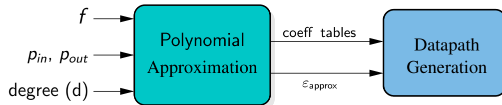
  N. Brisebarre and S. Chevillard, *Efficient polynomial L^∞- approximations*
- *input:*
  - function f, degree d
  - interval I
  - list of coefficient size constraints
- *output:*
  - polynomial with **precision-constrained** coefficients (no need to round them)
  - $max(|f_i(y) - p_i(y)|) \leq \varepsilon_{\text{approx}} \forall i \in \{0..2^k - 1\}$



Advantages

- usually best polynomials given the input specifications
- might reduce by 1 polynomial degree for some intervals

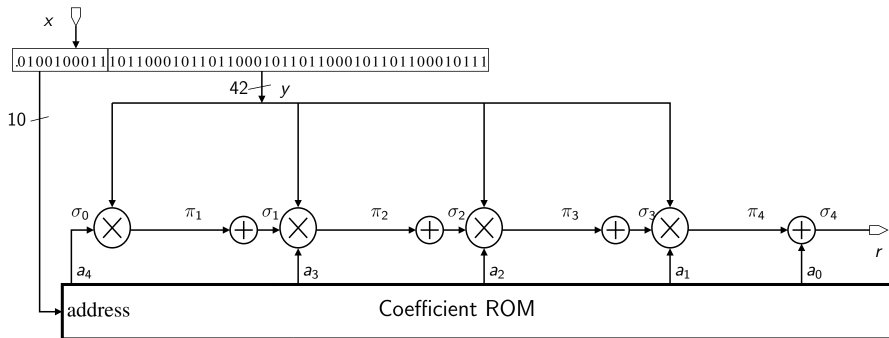
Polynomial Evaluation



Use `Horner` (trade latency for size)

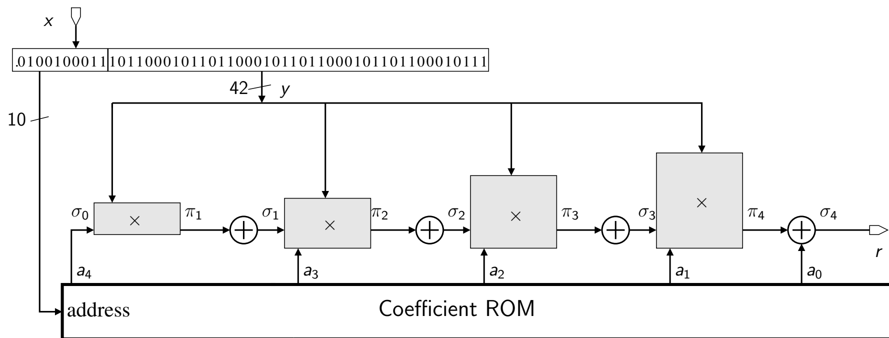
$$p(y) = \underbrace{\underbrace{a_0 + \underbrace{y \times \underbrace{(a_1 + \underbrace{y \times \underbrace{(a_2 + \underbrace{y \times \underbrace{(a_3 + \underbrace{y \times \underbrace{a_4}_{\sigma_0}}_{\pi_1})}_{\sigma_1}}_{\pi_2})}_{\sigma_2}}_{\pi_3})}_{\sigma_3}}_{\pi_4}}_{\sigma_4}}$$

The architecture for $log_2(1+x)$, DP, $d=4$



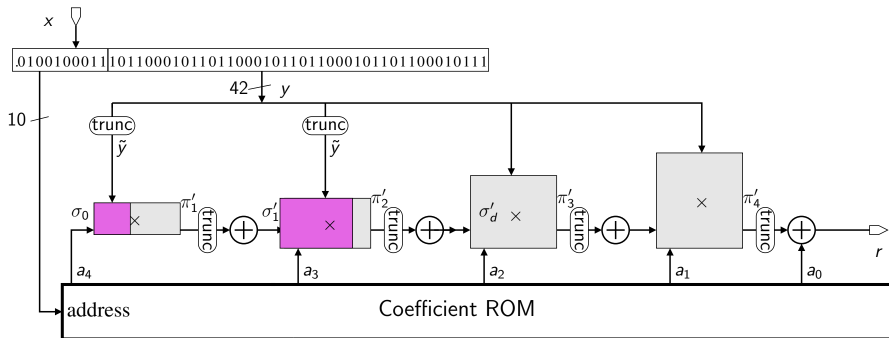
truncation on y and on π_j

The architecture for $log_2(1 + x)$, DP, $d = 4$



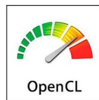
truncation on y and on π_j

The architecture for $log_2(1+x)$, DP, $d=4$



truncation on y and on π_j

Where to find them? In Megawizard, DSP Builder, oneAPI, OpenCL



Same function, different VHDL:

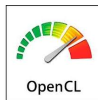
- pipelined to 300 MHz

```
./flopoco frequency=300 FPAdd wE=6 wF=31
```

- A larger but shorter-latency architectural variant:

```
./flopoco FPAdd wE=8 wF=23 dualpath=true
```

Where to find them? In Megawizard, DSP Builder, oneAPI, OpenCL



Same function, different VHDL:

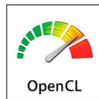
- pipelined to 300 MHz

```
./flopoco frequency=300 FPAdd wE=6 wF=31
```

- A larger but shorter-latency architectural variant:

```
./flopoco FPAdd wE=8 wF=23 dualpath=true
```

Where to find them? In Megawizard, DSP Builder, oneAPI, OpenCL



Same function, different VHDL:

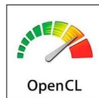
- pipelined to 300 MHz

```
./cmdPolyEval -frequency 300 FPAdd 6 31
```

- A larger but shorter-latency architectural variant:

```
./cmdPolyEval FPAddExpert 8 23 1 1 0
```

Where to find them? In Megawizard, DSP Builder, oneAPI, OpenCL



Same function, different VHDL:

- pipelined to 300 MHz

```
./cmdPolyEval -frequency 300 FPAdd 6 31
```

- A larger but shorter-latency architectural variant:

```
./cmdPolyEval FPAddExpert 8 23 1 1 0
```

- different function, another language (Verilog):

```
./cmdPolyEval -lang VERILOG FPArctan2 10 44
```

Bit heaps: the mutant biology of arithmetic beasts

Careless PhD students and their pets gone wrong

Fantastic but not evil: circuits computing just right

Fantastic arithmetic beasts escaped to vendor tools

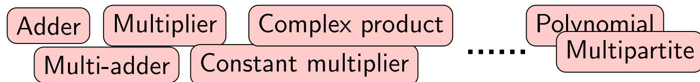
Bit heaps: the mutant biology of arithmetic beasts

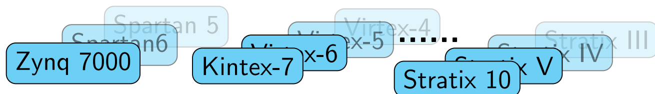
Why fantastic arithmetic beasts didn't take over the world (and how to address it)

Backup slides

So much VHDL to write, so few ~~slaves~~ students to write it !

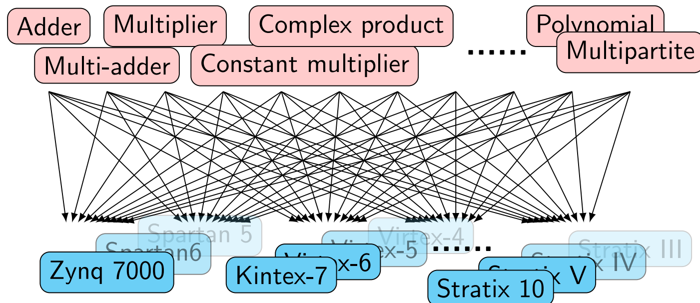
In theory, I know how to optimize by hand each operator for each target...





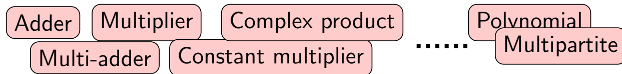
So much VHDL to write, so few ~~slaves~~ students to write it !

In theory, I know how to optimize by hand each operator for each target...

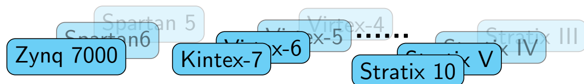


But I don't have the resources.

One data-structure to rule them all...

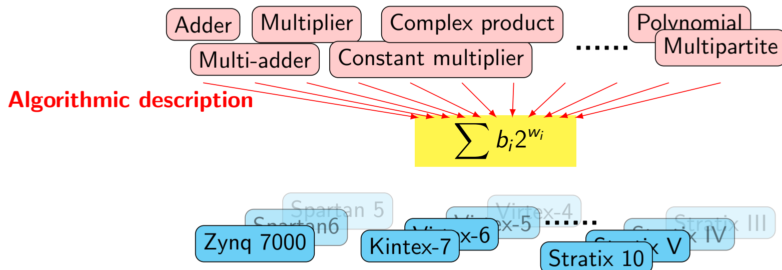


$$\sum b_i 2^{w_i}$$



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

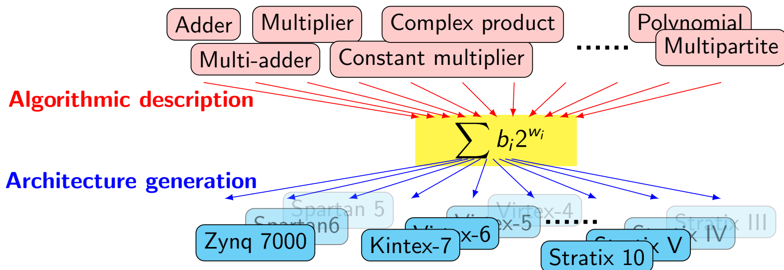
One data-structure to rule them all...



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

- Captures the true binary math of an operation
  (with all sorts of **bit-level optimization** opportunities)

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

- Captures the true binary math of an operation (with all sorts of **bit-level optimization** opportunities)
- The corresponding **compressor trees** can be optimized for each target ... and optimally so for practical sizes, thanks to M. Kumm

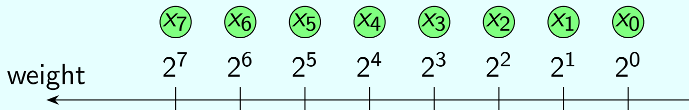
Sums of weighted bits?

- Integers or real numbers represented in binary fixed-point

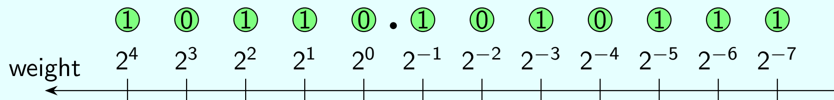
$$X = \sum_{i=i_{\min}}^{i_{\max}} 2^i x_i$$

- 2^i : "weight" $\Longrightarrow$ "X is a sum of weighted bits"

Representation as a **dot diagrams**



Example: 17.42 written in binary



The historical bit heap was invented for building multipliers

$$XY = (\sum_{i=0}^{3} 2^i x_i) \times (\sum_{j=0}^{3} 2^j y_j)$$

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

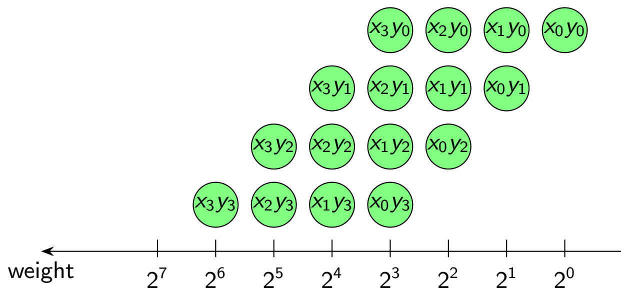
A multiplier is an architecture that computes this sum.

The historical bit heap was invented for building multipliers

$$XY = (\sum_{i=0}^{3} 2^i x_i) \times (\sum_{j=0}^{3} 2^j y_j)$$

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

A multiplier is an architecture that computes this sum.

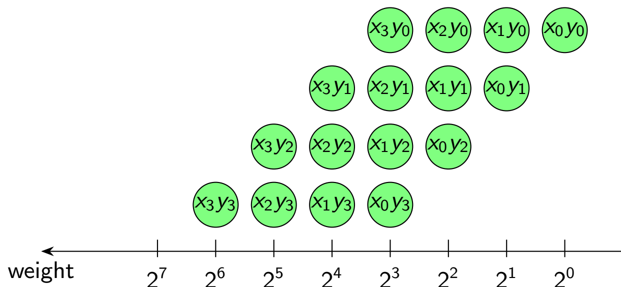


The historical bit heap was invented for building multipliers

$$XY = (\sum_{i=0}^{3} 2^i x_i) \times (\sum_{j=0}^{3} 2^j y_j)$$

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

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**

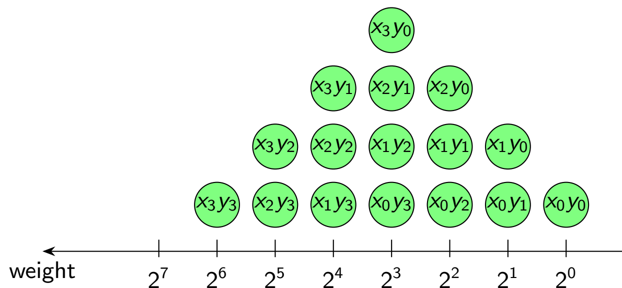
... exposing design freedom thanks to **associativity** and **commutativity** of the $\sum$
(and a few other boolean tricks)

The historical bit heap was invented for building multipliers

$$XY = \left(\sum_{i=0}^{3} 2^i x_i\right) \times \left(\sum_{j=0}^{3} 2^j y_j\right)$$

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

A multiplier is an architecture that computes this sum.



Historical motivation for bit heaps:

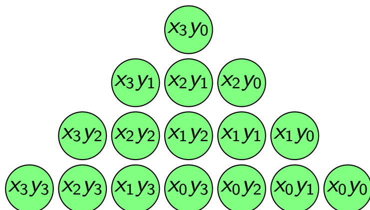
$\sum_{i,j} 2^{i+j} x_i y_j$ **expresses the bit-level parallelism of the problem**

... exposing design freedom thanks to **associativity** and **commutativity** of the $\sum$

(and a few other boolean tricks)

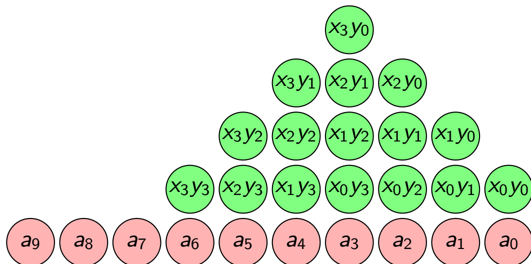
Beyond product

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



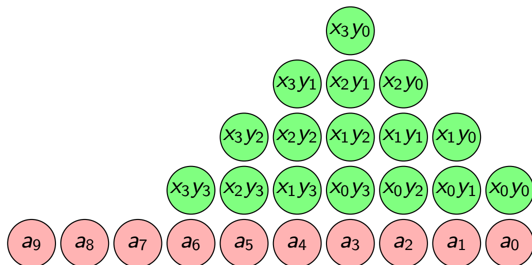
Beyond product

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



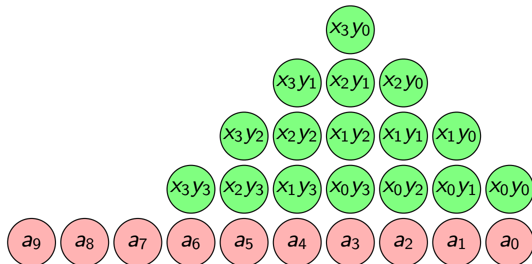
Beyond product

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



Beyond product

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



When generating an architecture

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

- get rid of artificial sequentiality (inside operators, and between operators)
- focus on true timing information (e.g. critical path delay of each weighted bit)
- A global optimization instead of several local ones (and solved by ILP)

Well beyond product

A bit heap is anything that can be developed as $\sum_{w,h} 2^w b_{w,h}$

- the sum of two bit heaps is obviously a bit heap
- the product of two bit heaps is also a bit heap

Well beyond product

A bit heap is anything that can be developed as $\sum_{w,h} 2^w b_{w,h}$

- the sum of two bit heaps is obviously a bit heap
- the product of two bit heaps is also a bit heap

Any polynomial of multiple variables is a bit heap

... where each $b_{w,h}$ is the AND of a few input bits.
This includes sums of squares, FIR filters, etc

Well beyond product

A bit heap is anything that can be developed as $\sum_{w,h} 2^w b_{w,h}$

- the sum of two bit heaps is obviously a bit heap
- the product of two bit heaps is also a bit heap

Any polynomial of multiple variables is a bit heap

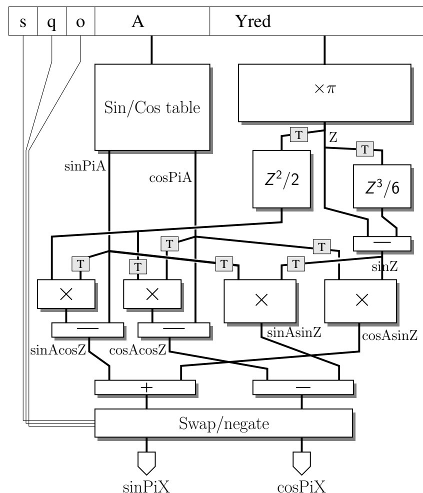
... where each $b_{w,h}$ is the AND of a few input bits.
This includes sums of squares, FIR filters, etc

And then more

- A huge class of function may be *approximated* by polynomials
- The $b_{w,h}$ may be read from arbitrary look-up tables
- An operator may include several bit heaps

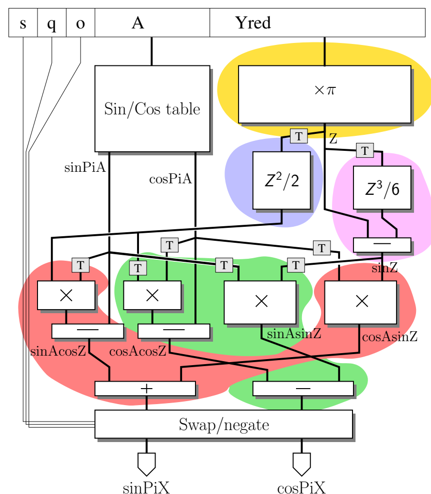
A good hammer transforms every problem into a nail

A sine/cosine architecture (HEART 2013)

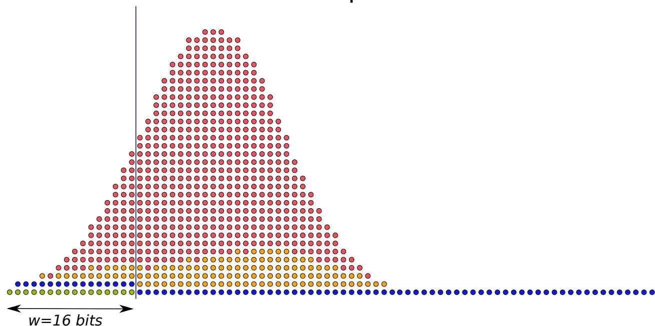


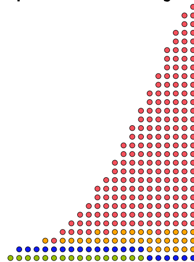
A good hammer transforms every problem into a nail

A sine/cosine architecture (HEART 2013) mostly consists of **5 bit heaps**

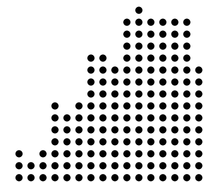
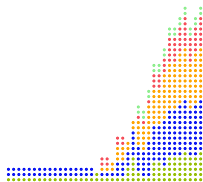
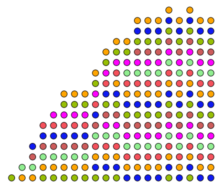
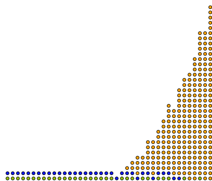
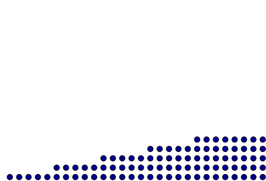
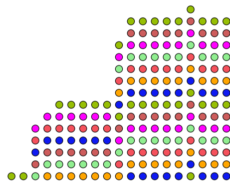


A bit heap for $Z - Z^3/6$ in the previous architecture

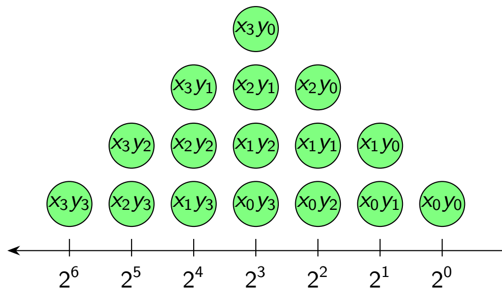




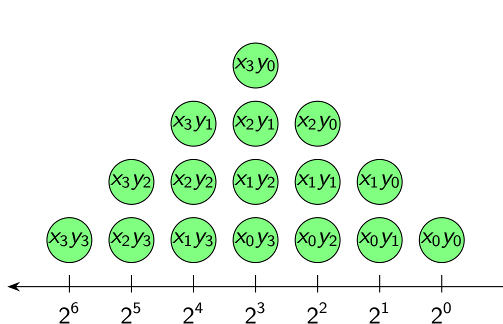
Bit heaps for other operators and filters

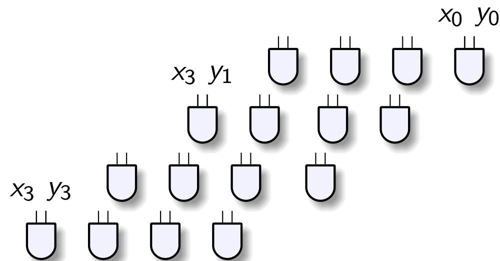







Computing the sum: bit heap compression

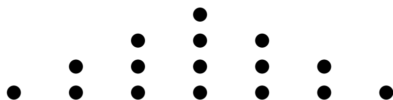


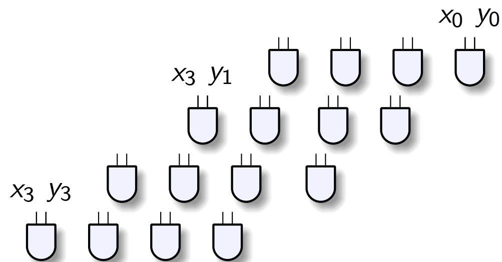
Computing the sum: bit heap compression



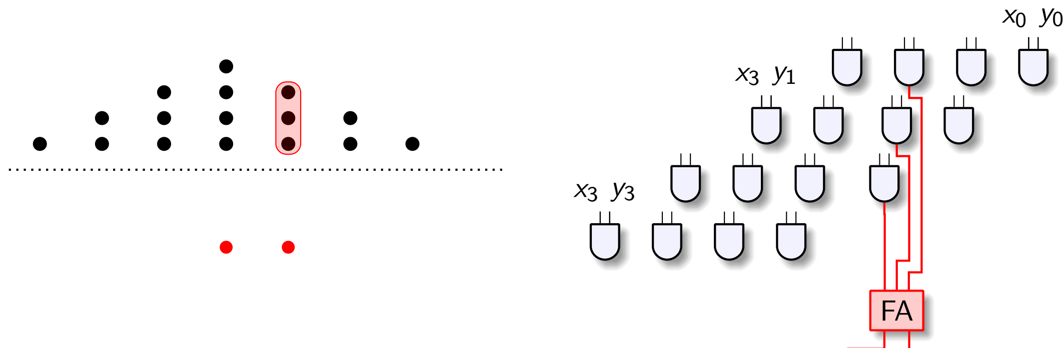


Computing the sum: bit heap compression



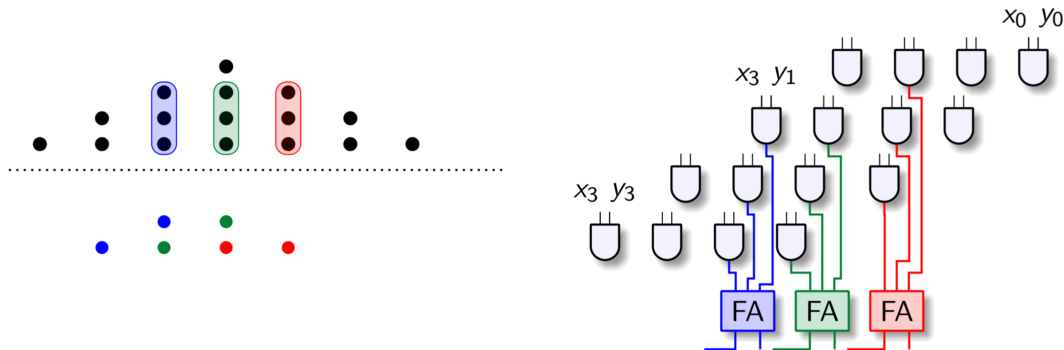


Computing the sum: bit heap compression



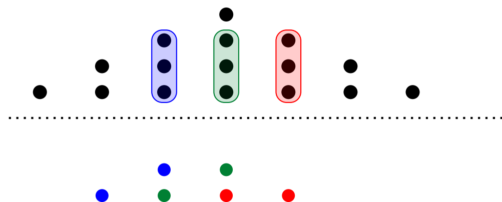
A *full adder* (FA) inputs three bits of the same weight
and outputs their sum, written in binary on two bits.

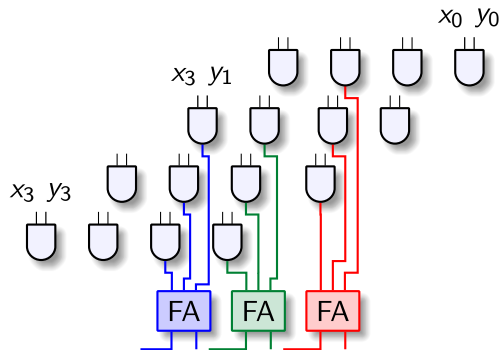
Computing the sum: bit heap compression



Let us pave the bit heap with as many FA as possible. They work in parallel.
We obtain a new bit heap

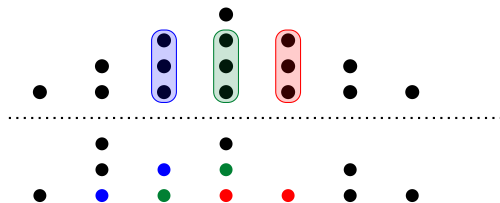
Computing the sum: bit heap compression

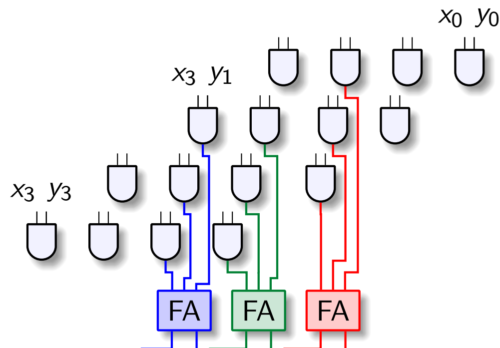




Some of the initial bits remain uncompressed.

Computing the sum: bit heap compression



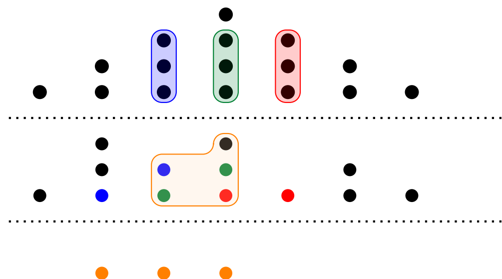


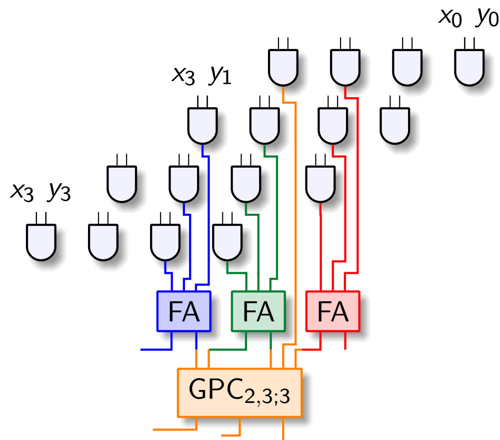
Some of the initial bits remain uncompressed.

They are simply transfered to the new bit heap.

Now let's compress this new bit heap.

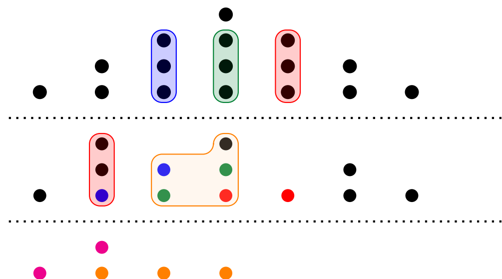
Computing the sum: bit heap compression

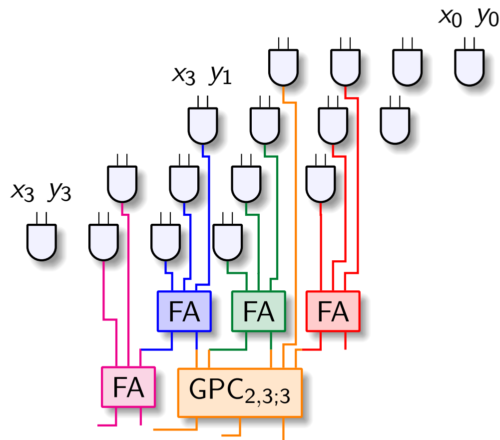




On FPGAs, fancy compressors are possible.
This one costs 3 LUT5 working in parallel to compress 5 bits into 3.
All things considered, it is more efficient than using FAs.

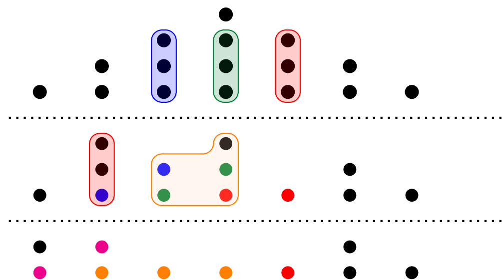
Computing the sum: bit heap compression

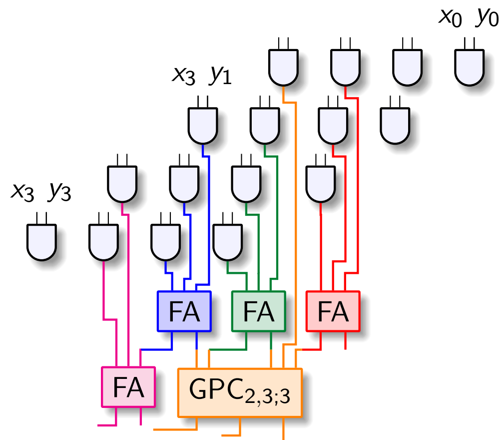




… and another full adder in parallel

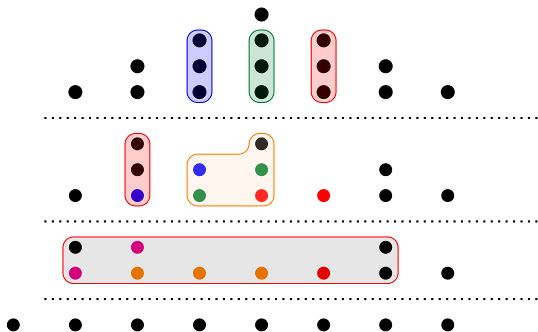
Computing the sum: bit heap compression



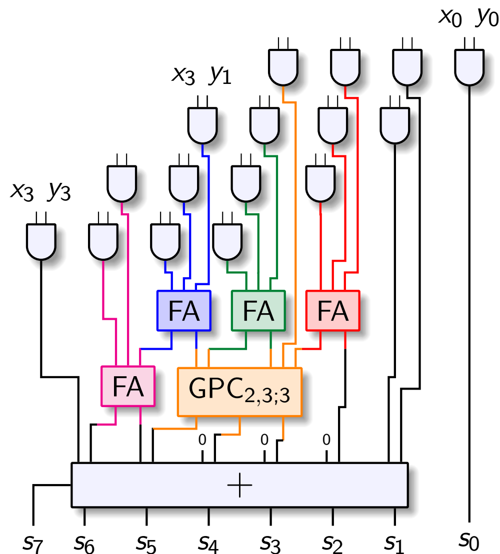


... and some bits untouched.

Computing the sum: bit heap compression



Finally, a bit heap of height ≤ 2
can be compressed by an **adder**
(a *fast adder* in VLSI,
using the *fast carry chain* on FPGAs)



Was it the optimal solution?

Answer is of course: it depends!

- on the target FPGA
- on the cost function to optimize (latency, or area, or ...)

I used to write ad-hoc heuristics for bit heap compression.

The first wave of an invasion of optimization techniques

Martin Kumm and Peter Zipf.
Pipelined Compressor Tree Optimization Using Integer Linear Programming
FPL, 2014.

Martin Kumm and Johannes Kappauf.
Advanced Compressor Tree Synthesis for FPGAs.
IEEE Transactions on Computers, 2018

This 11-year-old operator has improved by magic...

s q o A Yred

Sin/Cos table

×π

sinPiA cosPiA

$Z^2/2$ Z $Z^3/6$

sinZ

sinAcosZ cosAcosZ sinAsinZ cosAsinZ

Swap/negate

sinPiX cosPiX

Adder Multiplier Complex product Polynomial

Multi-adder Constant multiplier ...... Multipartite

Algorithmic description

$$\sum b_i 2^{w_i}$$

Architecture generation

Spartan 5 Spartan6 Zynq 7000 Kintex-7 Virtex-6 Virtex-5 Virtex-4 ...... Stratix III Stratix IV Stratix V Stratix 10

Why fantastic arithmetic beasts didn't take over the world (and how to address it)

Careless PhD students and their pets gone wrong

Fantastic but not evil: circuits computing just right

Fantastic arithmetic beasts escaped to vendor tools

Bit heaps: the mutant biology of arithmetic beasts

Why fantastic arithmetic beasts didn't take over the world (and how to address it)

Backup slides

As a tool for real-world designers, FloPoCo is mostly useless

The project is in a double **technological dead-end**

- on the **input**:
  we cannot expect every designer to have read all our papers
  (e.g. to know that there exists a floating-point divider-by-3.)

Example of bug report by a highly valued user

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

Can you see what is wrong here?

As a tool for real-world designers, FloPoCo is mostly useless

The project is in a double **technological dead-end**

- on the **input**:
  we cannot expect every designer to have read all our papers
  (e.g. to know that there exists a floating-point divider-by-3.)

Example of bug report by a highly valued 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!

As a tool for real-world designers, FloPoCo is mostly useless

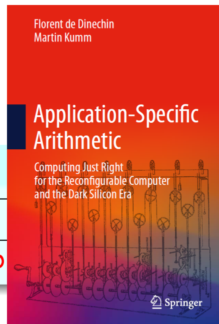
The project is in a double **technological dead-end**

- on the **input**:
  we cannot expect every designer to have read all our papers
  (e.g. to know that there exists a floating-point divider-by-3.)

Example of bug report by a highly valued 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

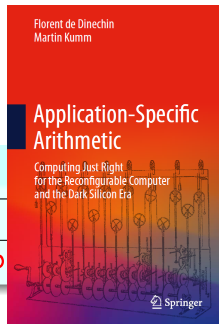


(all real-world designers should buy the book, though)

As a tool for real-world designers, FloPoCo is mostly useless

The project is in a double **technological dead-end**

- on the **input**:
  we cannot expect every designer to have read all our papers
  (e.g. to know that there exists a floating-point divider-by-3.)



Example of bug report by a highly valued 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

(all real-world designers should buy the book, though)

- on the **output**: the future is HLS, but
  flopoco-generated VHDL is incompatible with HLS
  - a-posteriori interfacing is barely possible (and painful)
  - but result will be inefficient anyway
    as long as HLS doesn't control the computation at the core of the loop nest

Why should I care about real-world designers, I am an academic

Should these fantastic arithmetic beasts remain chained in an ivory tower?

Why should I care about real-world designers, I am an academic

Should these fantastic arithmetic beasts remain chained in an ivory tower?

We need an HLS framework

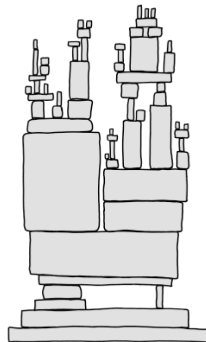
An HLS framework where we can

- detect interesting *operations* (or compound operations)
- to convert them to efficient application-specific *operators*

by invoking a FloPoCo-like tool automatically

It sounds like Another Grand Plan

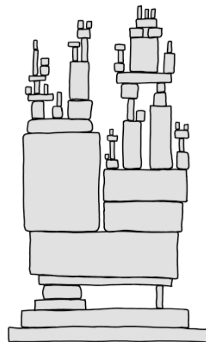
Escaping the FloPoCo ivory tower:



It sounds like Another Grand Plan

Escaping the FloPoCo ivory tower:

- MLIR:
  Multi-Level Intermediate
  Representation

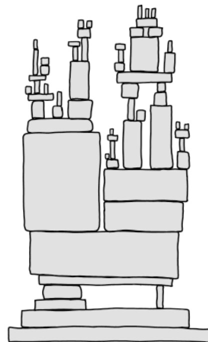


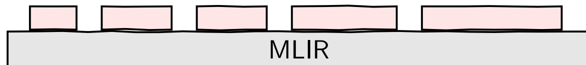
MLIR

It sounds like Another Grand Plan

Escaping the FloPoCo ivory tower:

- MLIR:
  Multi-Level Intermediate Representation
  - helps defining domain-specific Intermediate Representations ("dialects")

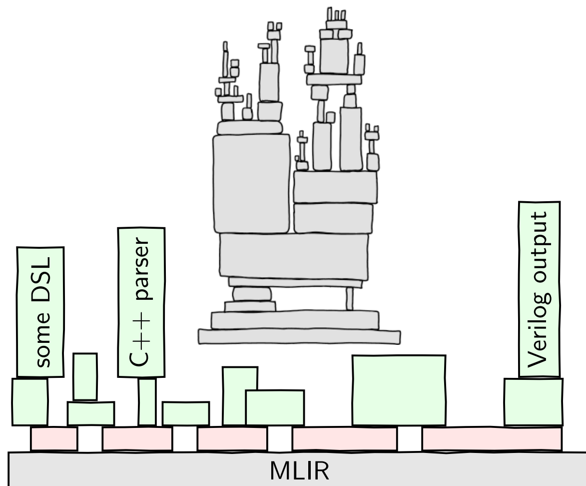




It sounds like Another Grand Plan

Escaping the FloPoCo ivory tower:

- MLIR:
  Multi-Level Intermediate Representation
  - helps defining domain-specific Intermediate Representations (“dialects”)
  - ... and program transformation / optimization passes
  - ... from high-level languages to assembly code, or... Verilog.

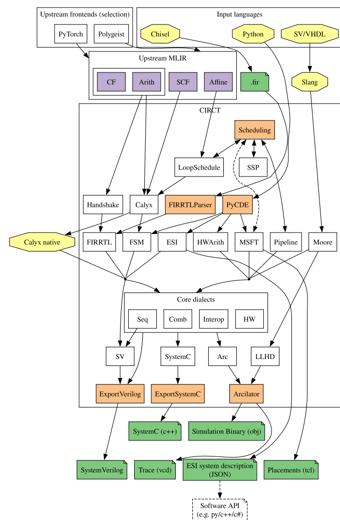


It sounds like Another Grand Plan

Escaping the FloPoCo ivory tower:

- MLIR:
  Multi-Level Intermediate Representation
  - helps defining domain-specific Intermediate Representations ("dialects")
  - ... and program transformation / optimization passes
  - ... from high-level languages to assembly code, or... Verilog.

  (it was a very simplified overview)

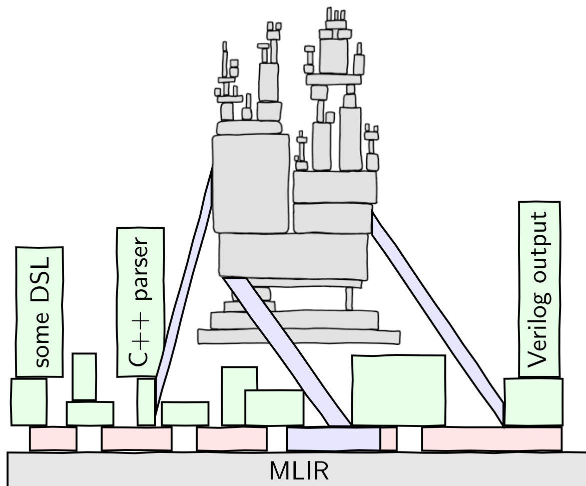


It sounds like Another Grand Plan

Escaping the FloPoCo ivory tower:

- MLIR:
  Multi-Level Intermediate Representation
  - helps defining domain-specific Intermediate Representations ("dialects")
  - ... and program transformation / optimization passes
  - ... from high-level languages to assembly code, or... Verilog.

  (it was a very simplified overview)
- All we need is a few bridges

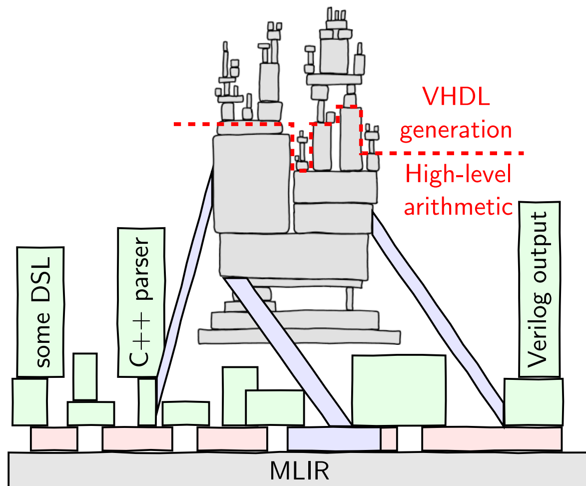


It sounds like Another Grand Plan

Escaping the FloPoCo ivory tower:

- MLIR:
  Multi-Level Intermediate Representation
  - helps defining domain-specific Intermediate Representations ("dialects")
  - ... and program transformation / optimization passes
  - ... from high-level languages to assembly code, or... Verilog.

  (it was a very simplified overview)
- All we need is a few bridges
- ... and quite a lot of janitoring

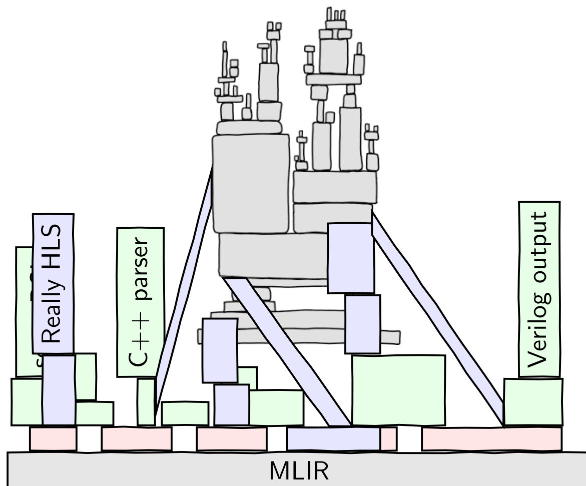


It sounds like Another Grand Plan

Escaping the FloPoCo ivory tower:

- MLIR:
  Multi-Level Intermediate Representation
  - helps defining domain-specific Intermediate Representations ("dialects")
  - ... and program transformation / optimization passes
  - ... from high-level languages to assembly code, or... Verilog.

  (it was a very simplified overview)
- All we need is a few bridges
- ... and quite a lot of janitoring
- ... then we can write endless arithmetic optimization passes

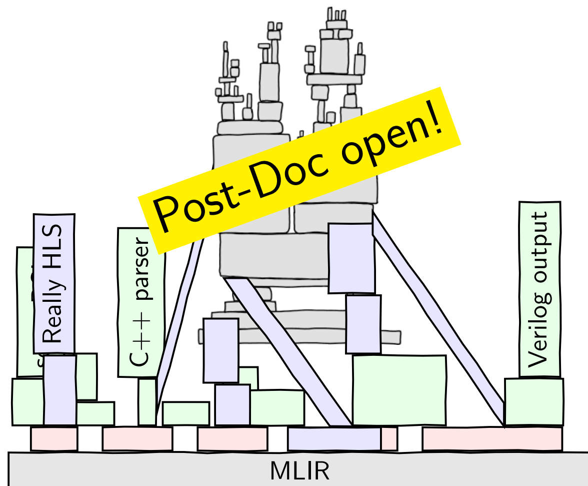


It sounds like Another Grand Plan

Escaping the FloPoCo ivory tower:

- MLIR:
  Multi-Level Intermediate Representation
  - helps defining domain-specific Intermediate Representations ("dialects")
  - ... and program transformation / optimization passes
  - ... from high-level languages to assembly code, or... Verilog.

  (it was a very simplified overview)
- All we need is a few bridges
- ... and quite a lot of janitoring
- ... then we can write endless arithmetic optimization passes



Doesn't it look like a winning move?

Thanks for your attention

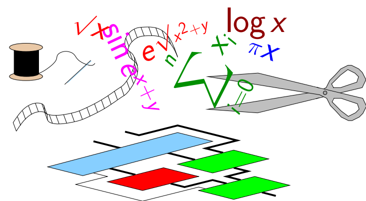
Save petrol! Save the planet! Don't move useless metal! Ride a bicycle!

Thanks for your attention

Save petrol! Save the planet! Don't move useless metal! Ride a bicycle!
Save routing! Save power! Don't move useless bits around!

Thanks to all the FloPoCo contributors:

H.Abdoli, S. Banescu, L. Besème, A. Böttcher, N. Bonfante, N. Brunie, R. Bouarah, V. Capelle, M. Christ, C. Collange, Q. Corradi, O. Desrentes, J. Detrey, A. Dudermel, P. Echeverría, F. Ferrandi, N. Fiege, L. Forget, M. Grad, M. Hardieck, V. Huguet, K. Illyes, M. Istoan, M. Joldes, J. Kappauf, C. Klein, M. Kleinlein, K. Klug, M. Kumm, J. Kühle, K. Kullmann, L. Ledoux, J. Marchal, D. Mastrandrea, K. Möller, R. Murillo, B. Pasca, B. Popa, X. Pujol, G. Sergent, V. Schmidt, D. Thomas, R. Tudoran, A. Vasquez, A. Volkova.



and the authors of GMP, MPFR, Sollya, SCIP, nvc, LaTeX, TikZ, ...

Backup slides

Careless PhD students and their pets gone wrong

Fantastic but not evil: circuits computing just right

Fantastic arithmetic beasts escaped to vendor tools

Bit heaps: the mutant biology of arithmetic beasts

Why fantastic arithmetic beasts didn't take over the world (and how to address it)

Backup slides

Don't trust us

FloPoCo can generate an infinite number of operators.
Obviously, I haven't tested them all.

Every operator comes with its specific test bench

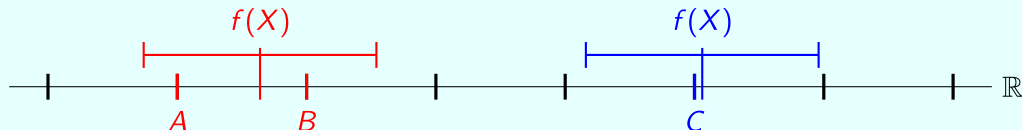
```
./flopoco FixFunctionByPiecewisePoly f="exp(x*x)" lsbIn=-24 lsbOut=-24 d=3
    TestBench
```

- based on operator(X) = quantization(operation(X))
- `emulate()` method is a few lines of code
  - based on trusted stuff such as MPFR and Sollya
  - and we write it first, and it is easy to audit
  - it should really be called `specification()`

It would be too simple, people would complain

Sometimes correct rounding is to expensive to implement, or just impossible to guarantee...

Faithful rounding: the next best thing



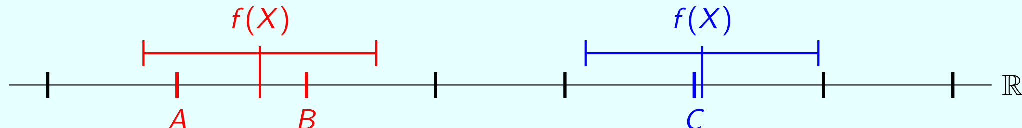
Two equivalent specifications:

- The output Y of the operator may be one of the two numbers surrounding $f(X)$. When $f(X)$ is a machine number, then $Y = f(X)$.
- The difference between the output value Y and $f(x)$ is strictly smaller than u.

It would be too simple, people would complain

Sometimes correct rounding is to expensive to implement, or just impossible to guarantee...

Faithful rounding: the next best thing



Two equivalent specifications:

- The output Y of the operator may be one of the two numbers surrounding $f(X)$. When $f(X)$ is a machine number, then $Y = f(X)$.
- The difference between the output value Y and $f(x)$ is strictly smaller than u.

Slightly less accurate than correct rounding, but still:

if you add one bit to the output, you divide u by 2, hence double the accuracy.

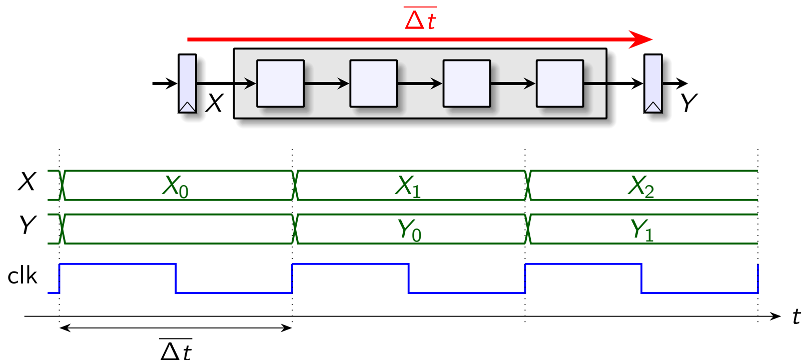
Parenthesis: binary for theoretical physicists (and other signal people)

- $2^{10} \approx 10^3$ (kBytes are actually 1024 bytes).
- Another point of view : $10 \log_{10}(2) \approx 3$
- In other words, 1 bit $\approx$ 3 dB

I don't count signal/noise ratio in dB, I count accuracy in bits.
But it is the same thing.

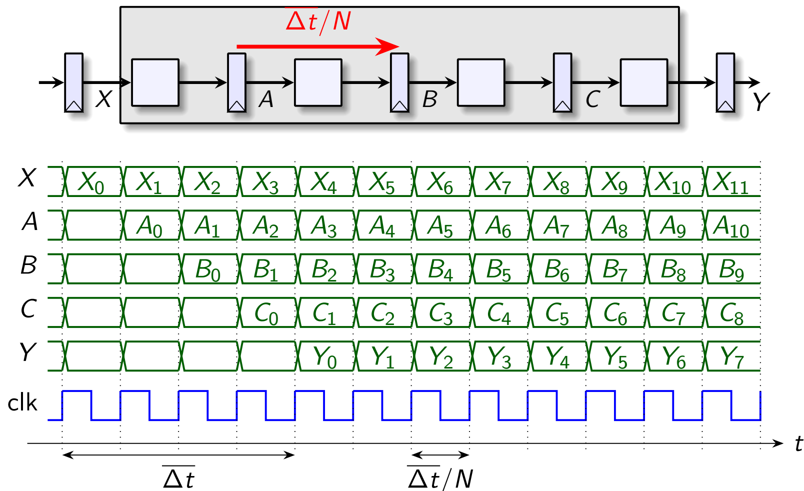
Performance through pipelining

A combinatorial operator, with registers that produce its inputs and consume its outputs

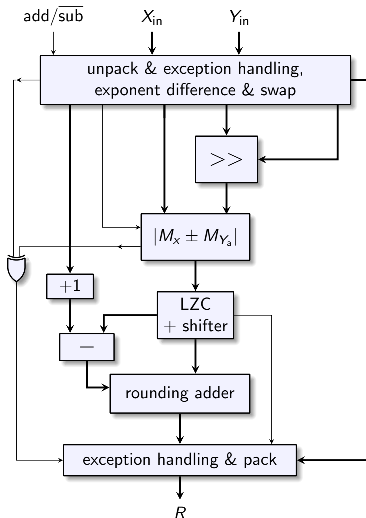


Performance through pipelining

The same operator, pipelined into $N = 4$ stages: frequency can be multiplied by 4.

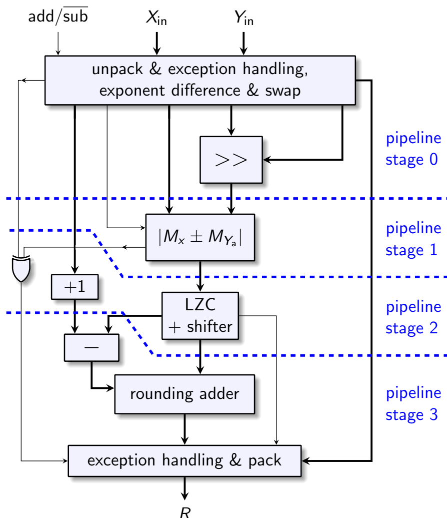


A more realistic example




```
./flopoco fpadd we=8 wf=23
```

A more realistic example




```
./flopoco frequency=200 fpadd we=8 wf=23
```

Adds 3 synchronization barriers:

- FloPoCo reports a pipeline depth of 3,
- meaning that there are 4 pipeline stages

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by other core generators...

Frequency-directed pipelining

The same FPAdder, pipelined for 300MHz:

```
./flopoco frequency=300 FPAdd wE=8 wF=23
```

FloPoCo interface to pipeline construction

"Please pipeline this operator to work at 200MHz"

Not the choice made by other core generators...

... but better because *compositional*

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

Examples of pipeline

```
./flopoco frequency=400 FPAdd wE=8 wF=23
```

```
Final report:
|---Entity FPAdder_8_23_uid2_RightShifter
|      Pipeline depth = 1
|---Entity IntAdder_27_f400_uid7
|      Pipeline depth = 1
|---Entity LZCShifter_28_to_28_counting_32_uid14
|      Pipeline depth = 4
|---Entity IntAdder_34_f400_uid17
|      Pipeline depth = 1
Entity FPAdder_8_23_uid2
   Pipeline depth = 9
```

```
./flopoco frequency=200 FPAdd wE=8 wF=23
```

```
Final report:
(...)
   Pipeline depth = 4
```

Of course the frequency depends on the target FPGA

```
./flopoco target=Zynq7000 frequency=200 FPAdd wE=8 wF=23
```

```
Final report:
(...)
   Pipeline depth = 5
```

```
./flopoco target=VirtexUltrascalePlus frequency=200 FPAdd wE=8 wF=23
```

```
Final report:
(...)
   Pipeline depth = 1
```

Altera and Xilinx targets supported in the stable branch (at various levels of accuracy, in various versions): Spartan3, Zynq7000, Virtex4, Virtex5, Virtex6, Kintex7, VirtexUltrascalePlus, StratixII, StratixIII, StratixIV, StratixV, CycloneII, CycloneIII, CycloneIV, CycloneV.

Frequency-directed pipelining in practice

We do our best but we know it's hopeless

The actual frequency obtained will depend on the whole application (placement, routing pressure etc)...

- best-effort philosophy,
- aiming to be accurate to 10% for an operator synthesized alone
- asking a higher frequency provides a deeper pipeline

Frequency-directed pipelining in practice

We do our best but we know it's hopeless

The actual frequency obtained will depend on the whole application (placement, routing pressure etc)...

- best-effort philosophy,
- aiming to be accurate to 10% for an operator synthesized alone
- asking a higher frequency provides a deeper pipeline

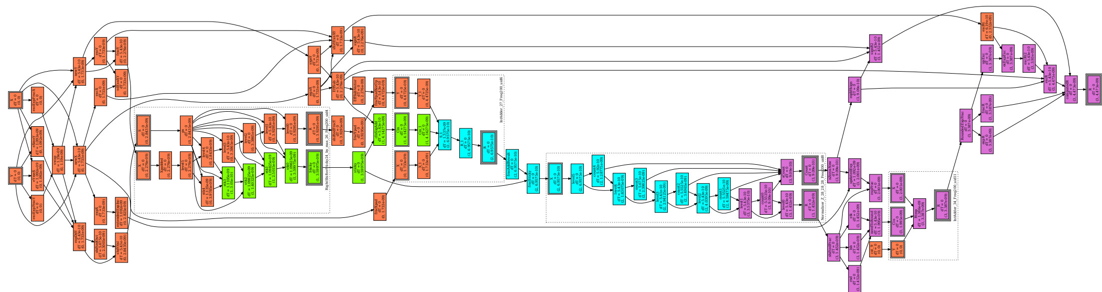
And a big TODO: VLSI targets.

And a few extras

Options `generateFigures` and `dependecyGraph` produce figures...

```
./flopoco frequency=200 dependencygraph=full fpadd we=8 wf=23
```

creates a dot `dot/` directory containing this:

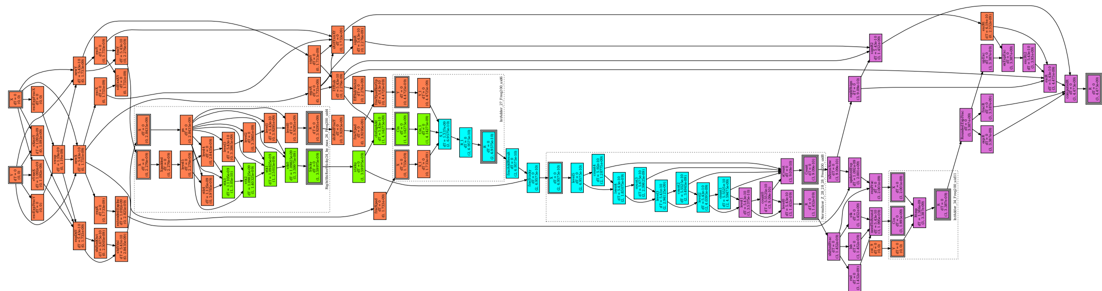


And a few extras

Options generateFigures and dependecyGraph produce figures...

```
./flopoco frequency=200 dependencygraph=full fpadd we=8 wf=23
```

creates a dot dot/ directory containing this:



Helper functions for encoding/decoding FP format, if you want to check the testbench...

- ```
  fp2bin 9 36 3.1415926
  ```
- ```
  bin2fp 9 36 010100000000100100100001111110110100110100010011
  ```