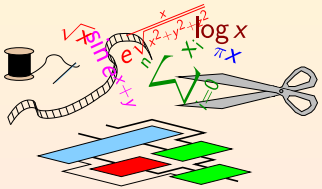
The FloPoCo arithmetic core generator: past, present and future.

Florent **de Dinechin**
florent.de.dinechin@insa-lyon.fr



H.Abdoli, S. Banescu, L. Besème, A. Böttcher, N. Bonfante, N. Brunie, R. Bouarah, V. Capelle, M. Christ, P. Cochard, C. Collange, Q. Corradi, O. Desrentes, J. Detrey, A. Dudermel, P. Echeverría, F. Ferrandi, N. Fiege, L. Forget, M. Grad, M. Hardieck, V. Huguet, T. Hubrecht, 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.

citi lab

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Inria

UNIVERSITÉ DE LYON

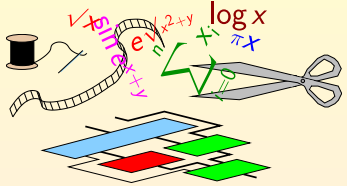
Fantastic arithmetic beasts (and where to find them)

Where to find them? In your applications!

... with a little help of FloPoCo.

www.flopoco.org

From maths
to circuits
with love and care

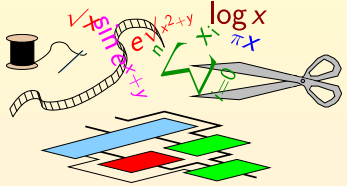


Where to find them? In your applications!

... with a little help of FloPoCo.

www.flopoco.org

From maths
to circuits
with love and care



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

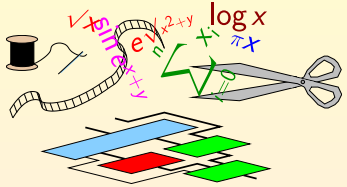
- written in C++, outputting portable, synthesizable VHDL
- open-source, extensible, state of the art

Where to find them? In your applications!

... with a little help of FloPoCo.

www.flopoco.org

From maths
to circuits
with love and care



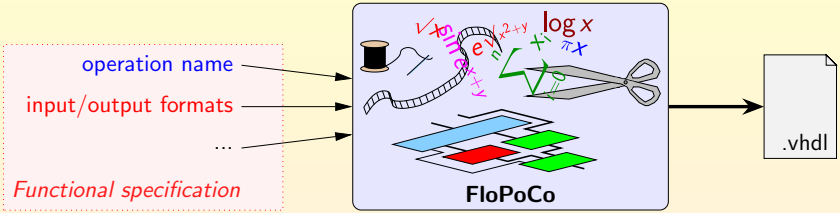
A generator of **application-specific** hardware arithmetic operators

- written in C++, outputting portable, synthesizable VHDL
- open-source, extensible, state of the art

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

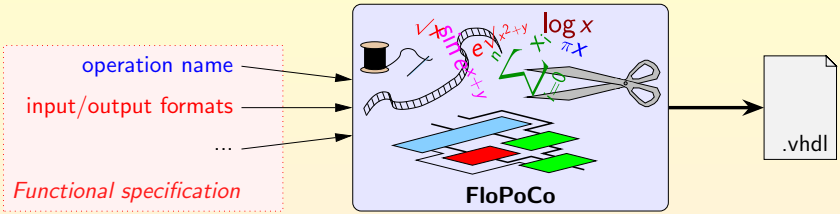
A command-line tool, sorry



Start simple: a single precision floating-point adder

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

A command-line tool, sorry



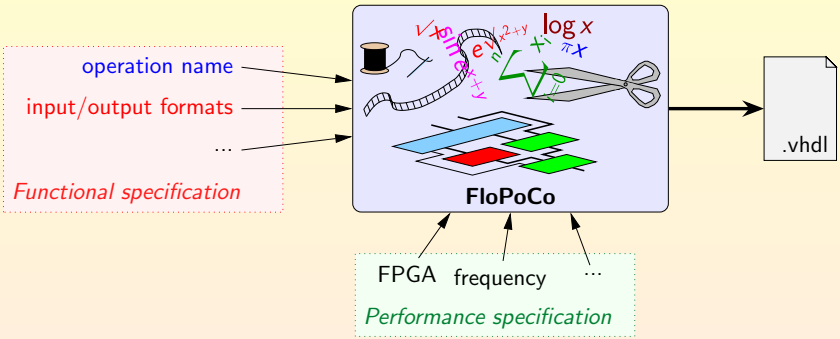
Start simple: a single precision floating-point adder

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

Do you really need 24 bits of mantissa (precision 10^{-8}) ?

```
./flopoco FPAdd wE=6 wF=12
```

A command-line tool, sorry



Start simple: a single precision floating-point adder

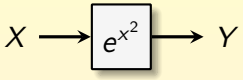
```
./flopoco IEEEFPAdd wE=8 wF=23
```

Do you really need 24 bits of mantissa (precision 10^{-8}) ? All we want is speed!

```
./flopoco FPAdd wE=6 wF=12 target=Zynq7000 frequency=300 dualpath=true
```

This was not FloPoCo.

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

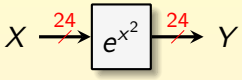


This was not FloPoCo.

Suppose you need to evaluate some function,

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

... with inputs and outputs on 24 bits.

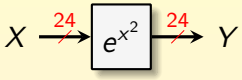


This was not FloPoCo.

Suppose you need to evaluate some function,

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

... with inputs and outputs on 24 bits.



This is FloPoCo:

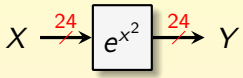
```
./flopoco FixFunctionByPiecewisePoly f="exp(x*x)" lsbIn=-24 lsbOut=-24 d=3
```

This was not FloPoCo.

Suppose you need to evaluate some function

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

... with inputs and outputs on 24 bits.



This is FloPoCo:

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

Cut from the Electrophysiology code from Vincent's talk yesterday

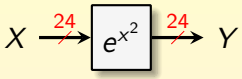
```
4  V; .lookup(-200,200,0.1);
5  Cai; .lookup(3e-4,30.0,3e-4);
6                                                    // ... (220 lines skipped)
7
8  IKACh = GACh*(10.0/(1.0+(9.13652/(pow(ACh, 0.477811)))))*(0.0517+ (0.4516/(1.0 + exp((V+59.53)/17.18))))*(V - E_K);
```

This was not FloPoCo.

Suppose you need to evaluate some function

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

... with inputs and outputs on 24 bits.



This is FloPoCo:

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

Cut from the Electrophysiology code from Vincent's talk yesterday

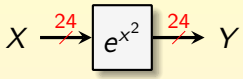
```
4  V; .lookup(-200,200,0.1);
5  Cai; .lookup(3e-4,30.0,3e-4);
6                                                    // ... (220 lines skipped)
7
8  IKACh = GACh*(10.0/(1.0+(9.13652/(pow(ACh, 0.477811))))*(0.0517+ (0.4516/(1.0 + exp((V+59.53)/17.18)))*(V - E_K);
```

This was not FloPoCo.

Suppose you need to evaluate some function

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

... with inputs and outputs on 24 bits.



This is FloPoCo:

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

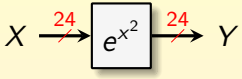
Cut from the Electrophysiology code from Vincent's talk yesterday

```
4  V; .lookup(-200,200,0.1);
5  Cai; .lookup(3e-4,30.0,3e-4);
6                                                    // ... (220 lines skipped)
7
8  IKACh = GACh*(10.0/(1.0+(9.13652/(pow(ACh, 0.477811))))*(0.0517+ (0.4516/(1.0 + exp((V+59.53)/17.18)))*(V - E_K);
```

```
./flopoco FixFunctionByPiecewisePoly
    f="0.0517+(0.4516/(1.0+(exp(((400*x-200+59.53)/17.18)))))"
    lsbIn=-24 lsbOut=-24 d=3
```

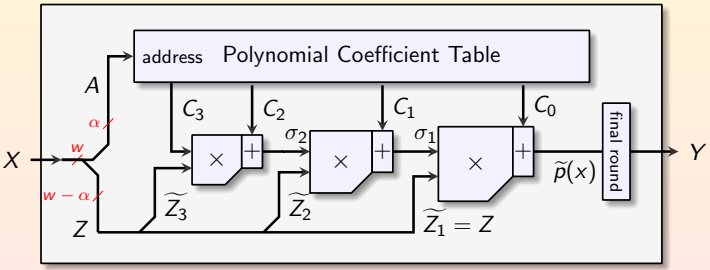
This was not FloPoCo.

Suppose you need to evaluate some function[1],
say $e^{(x^2)}$ on $[0, 1)$...
... with inputs and outputs on 24 bits.



This is FloPoCo:

```
./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)

Application-specific arithmetic

FloPoCo is really about operators that would make absolutely no sense in a processor.

Application-specific arithmetic

FloPoCo is really about operators that would make absolutely no sense in a processor.

This is also FloPoCo

```
./flopoco FPConstDiv d=3 wE=8 wF=23
```

A correctly rounded floating-point divider by 3,

bit-for-bit compatible with a standard divider.

Synthesis results on Kintex7 for single precision

standard correctly rounded divider	
748 LUT + 518 Reg	8 cycles @ 300 MHz

Application-specific arithmetic

FloPoCo is really about operators that would make absolutely no sense in a processor.

This is also FloPoCo

```
./flopoco FPConstDiv d=3 wE=8 wF=23
```

A correctly rounded floating-point divider by 3,
bit-for-bit compatible with a standard divider.

Synthesis results on Kintex7 for single precision

standard correctly rounded divider	
748 LUT + 518 Reg	8 cycles @ 300 MHz

floating-point multiplier by 1/3	
173 LUT, 6.667 ns	(3 cycle @ 400 MHz)

demo effect: pipeline not yet re-implemented for

```
./flopoco FPConstMult we=8 wf=23 constant="1/3"
```

Application-specific arithmetic

FloPoCo is really about operators that would make absolutely no sense in a processor.

This is also FloPoCo

```
./flopoco FPConstDiv d=3 wE=8 wF=23
```

A correctly rounded floating-point divider by 3,

bit-for-bit compatible with a standard divider.

Synthesis results on Kintex7 for single precision

standard correctly rounded divider	
748 LUT + 518 Reg	8 cycles @ 300 MHz

floating-point multiplier by 1/3	
173 LUT, 6.667 ns	(3 cycle @ 400 MHz)

FloPoCo FPConstDiv d=3	
39 LUT + 35 Reg	**1 cycle @ 400 MHz**

demo effect: pipeline not yet re-implemented for

```
./flopoco FPConstMult we=8 wf=23 constant="1/3"
```

Scope of FloPoCo

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

Scope of FloPoCo

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x,y,...) = rounding(operation(x, y,...))
  - FloPoCo uses the same approach for fixed-point operators

→ Clean mathematical specification

Scope of FloPoCo

- An arithmetic operation is a *function* (in the mathematical sense)
  - few well-typed inputs and outputs
  - no memory or side effect (usually)
- An operator is the *implementation* of such a function
  - IEEE-754 FP standard: operator(x,y,...) = rounding(operation(x, y,...))
  - FloPoCo uses the same approach for fixed-point operators

→ Clean mathematical specification

An operator, as a *circuit*...

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

- which means: the easy kind of graph

(approach recently extended to *DSP filters* defined by a transfer function)

FloPoCo can generate an infinite number of operators

- Obviously, there isn't enough disk space for all of them on Open Logic/Pile of Cores/...

FloPoCo can generate an infinite number of operators

- Obviously, there isn't enough disk space for all of them on Open Logic/Pile of Cores/...
- Obviously, we haven't tested them all.

FloPoCo can generate an infinite number of operators

- Obviously, there isn't enough disk space for all of them on Open Logic/Pile of Cores/...
- Obviously, we haven't tested them all.

Don't trust us! Every operator comes with its specific test bench

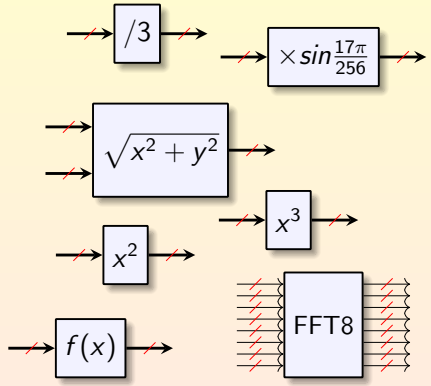
```
./flopoco IntConstDiv wIn=16 d=3 TestBench
```

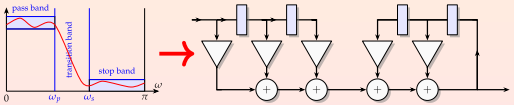
- based on operator(X) = quantization(operation(X))
- implemented as an `emulate()` method
  - a few lines of code only
  - based on trusted arbitrary-precision numerical libraries (MPFR, Sollya)
  - written first, and easy to audit (test-driven development)
  - produces a `test.input` file readable and commented

(We do have a CI based on this! Only, no pretense to full coverage)

Florent is busy until retirement

- Div by 3 was an example of
  operator specialization
- $\sqrt{X^2 + Y^2 + Z^2}$ is an example of
  operator fusion
- x^2 is also an example of context-specific
  resource sharing
  Also **Single Constant Multiplication**, ...
  Constant Matrix Multiplication ...
- We have only scratched the surface of
  function approximation
- Also in scope: **coarser kernels**
  such as Fast Fourier Transforms
  or neuron network layers
- From a frequency response to an IIR





Other examples of pointless operators

Some with readable VHDL, but not all...

- A complete single-precision FPU in a single VHDL file:

```
./flopoco FPAdd wE=8 wF=23 FPMult wE=8 wF=23 FPDiv wE=8 wF=23 FPSqrt wE=8 wF=23
```

- Exact integer constant multiplication (state of the art shift-and-add)

```
./flopoco IntConstMult wIn=16 constant=7654321
```

- Faithful multiplier of a fixed point by a *real* constant (table based)

```
./flopoco FixRealConstMult signedIn=1 msbIn=0 lsbIn=-15 lsbOut=-15
    constant="sin(42*pi/256)"
```

Other examples of pointless operators

Some with readable VHDL, but not all...

- A complete single-precision FPU in a single VHDL file:

```
./flopoco FPAdd wE=8 wF=23 FPMult wE=8 wF=23 FPDiv wE=8 wF=23 FPSqrt wE=8 wF=23
```

- Exact integer constant multiplication (state of the art shift-and-add)

```
./flopoco IntConstMult wIn=16 constant=7654321
```

- Faithful multiplier of a fixed point by a *real* constant (table based)

```
./flopoco FixRealConstMult signedIn=1 msbIn=0 lsbIn=-15 lsbOut=-15
    constant="sin(42*pi/256)"
```

How do you remember the options?

Don't reach for the source code yet, there is help on the command line

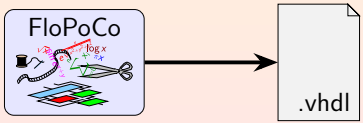
```
./flopoco TaoSort
```

```
./flopoco
```

Open source! Contributions welcome!

`http://flopoco.org/`

- Install from source (sorry) but it recently became easy
  - despite the fact that we pull MPFR, Sollya, WCPG, SCIP, ... **Thanks to them all!**
- These slides use current git master (some day, it will become version 5.0)
- Several older versions available
  - a few operators more, a few operators less
  - some versions so old you need to run them in a docker
  - (also see the "orphaned operators" page)
- License: AGPL, modified for FloPoCo so that **generated VHDL is LGPL**



A short and biased history of the project

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

Very good students tend to write kilolines of (very good?) code...

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

Very good students tend to write kilolines of (very good?) 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 (very good?) 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 (very good?) code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- with contributions to 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 (very good?) code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- with contributions to 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 (very good?) code...

Jérémie Detrey's PhD, 2004-2007

- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- with contributions to 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 (very good?) code...

Jérémie Detrey's PhD, 2004-2007

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

- VHDL, plus 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 (very good?) code...

Jérémie Detrey's PhD, 2004-2007

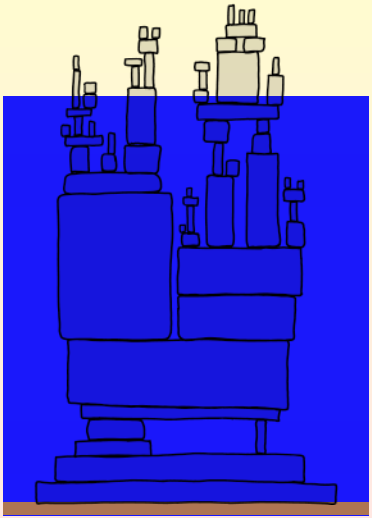
- FPLibrary: open-source VHDL for floating-point $+$, $-$, $\times$, $/$, $\sqrt{\ }$
- then parametric floating-point `sin`, `cos`, `exp`, `log`, ...
- with contributions to generic techniques for hardware function approximation
  - multipartite tables
  - HOTBM (higher-order table method)
- then LNS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

- VHDL, plus bits of Java/Python/C++ to generate some of the VHDL
- Design-space exploration scripts, test-bench generation, etc

Our scientific artifacts after Jérémie's PhD

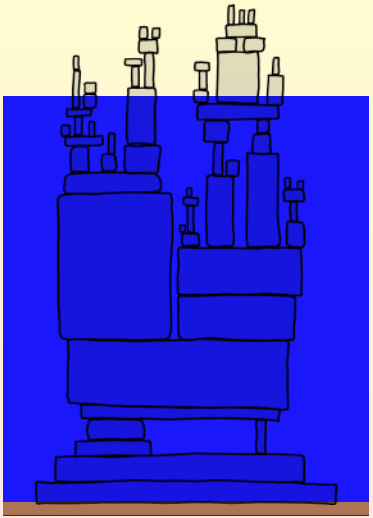


FPLibrary (VHDL available online)

stuff described in Jérémie's PhD

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

Our scientific artifacts after Jérémie's PhD

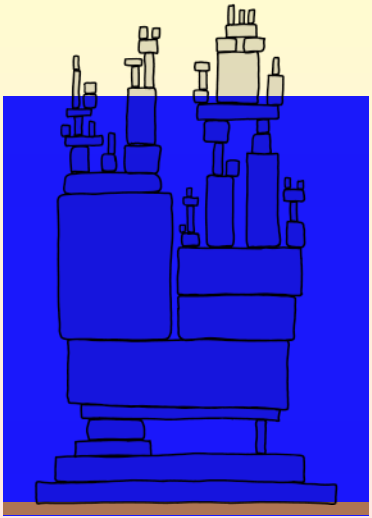


FPLibrary (VHDL available online)

stuff described in Jérémie's PhD
(in French)

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

Our scientific artifacts after Jérémie's PhD



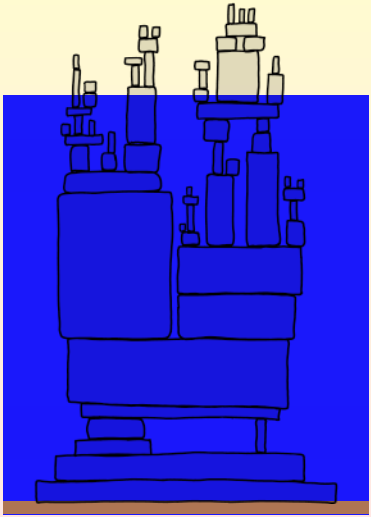
FPLibrary (VHDL available online)

stuff described in Jérémie's PhD
(in French)

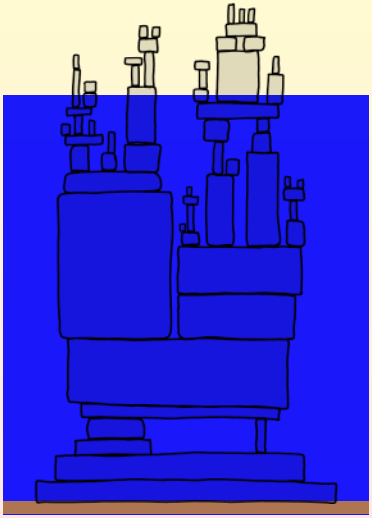
in other words:
work doomed to oblivion when the student leaves
(after his PhD, Jérémie defected to finite-field arithmetic)

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

Hence an Engineering Grand Plan



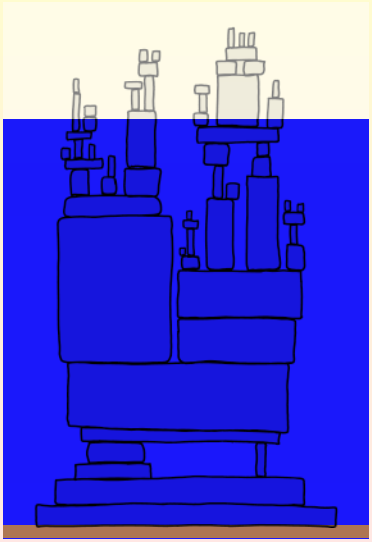
Hence an Engineering Grand Plan



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

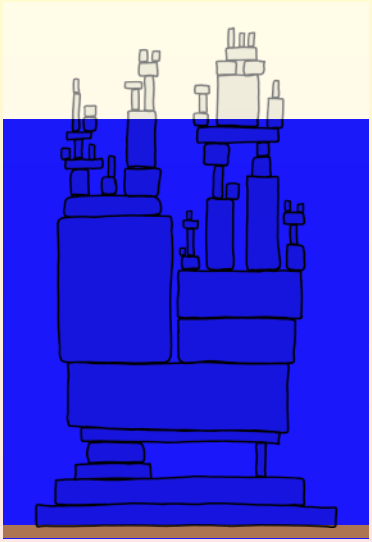


VHDL is generated,
no need to distribute it

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.

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)

- any intern who is half-competent in VHDL can start to develop in FloPoCo

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)
- any intern who is half-competent in VHDL can start to develop in FloPoCo
- and very soon: **automatic pipelining**

  (each submitted paper stupidely stated: *all this will be pipelined in the final version*)

Combinatorial circuits in FloPoCo: just print VHDL

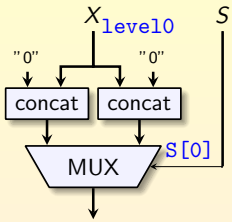
C++ code for a **Barrel Shifter**

```
for (int i=0; i<shiftValueWidth; i++){
   levelWidth = (...) ;
   vhdl<< declare("level" + to_string(i+1),
                  levelWidth)
       << " <= " << "level" << i << " & " << genZeros(1<<i)
       << " when S(" << i-1 << ") = '1' else "
       << genZeros(1<<i) << " & " << level << i<< ";";
}
```

Combinatorial circuits in FloPoCo: just print VHDL

C++ code for a **Barrel Shifter**

```
for (int i=0; i<shiftValueWidth; i++){
   levelWidth = (...) ;
   vhdl<< declare("level" + to_string(i+1),
                  levelWidth)
       << " <= " << "level" << i << " & " << genZeros(1<<i)
       << " when S(" << i-1 << ") = '1' else "
       << genZeros(1<<i) << " & " << level << i<< ";";
}
```



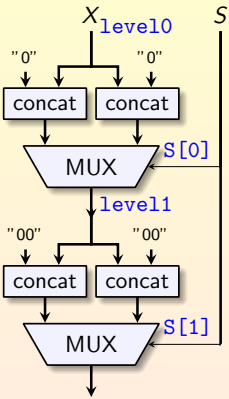
Resulting VHDL code

```
level1 <= level0 & "0" when S(0)='1'
          else "0" & level0;
```

Combinatorial circuits in FloPoCo: just print VHDL

C++ code for a **Barrel Shifter**

```
for (int i=0; i<shiftValueWidth; i++){
   levelWidth = (...) ;
   vhdl<< declare("level" + to_string(i+1),
                  levelWidth)
       << " <= " << "level" << i << " & " << genZeros(1<<i)
       << " when S(" << i-1 << ") = '1' else "
       << genZeros(1<<i) << " & " << level << i<< ";";
}
```



Resulting VHDL code

```
level1 <= level0 & "0" when S(0)='1'
          else "0" & level0;
level2 <= level1 & "00" when S(1)='1'
          else "00" & level1;
```

Combinatorial circuits in FloPoCo: just print VHDL

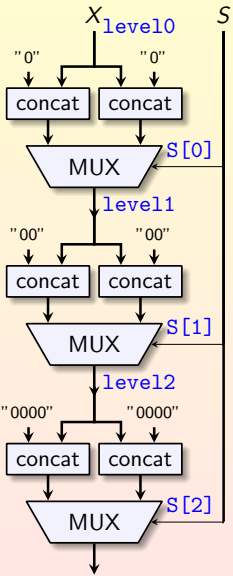
C++ code for a **Barrel Shifter**

```
for (int i=0; i<shiftValueWidth; i++){
   levelWidth = (...) ;
   vhdl<< declare("level" + to_string(i+1),
                  levelWidth)
       << " <= " << "level" << i << " & " << genZeros(1<<i)
       << " when S(" << i-1 << ") = '1' else "
       << genZeros(1<<i) << " & " << level << i<< ";";
}
```

Resulting VHDL code

```
level1 <= level0 & "0" when S(0)='1'
          else "0" & level0;
level2 <= level1 & "00" when S(1)='1'
          else "00" & level1;
level3 <= level2 & "0000" when S(2)='1'
          else "0000" & level2;

(...)
```



Pipelined circuits in FloPoCo: only think about the timing

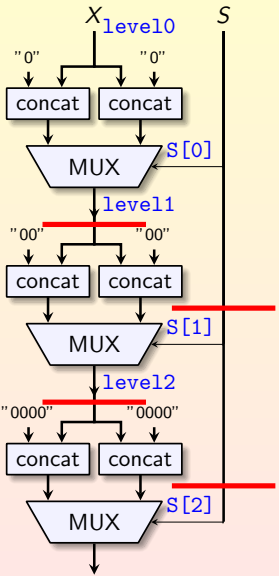
C++ code for a **Barrel Shifter**

```
for (int i=0; i<shiftValueWidth; i++){
 levelWidth = (...) ;
 vhdl<< declare(target->logicdelay(), "level" + to_string(i+1),
                levelWidth)
     << " <= " << "level" << i << " & " << genZeros(1<<i)
     << " when S(" << i-1 << ") = '1' else "
     << genZeros(1<<i) << " & " << level << i<< ";";
}
```

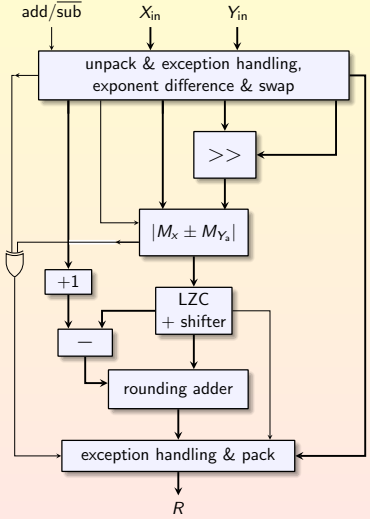
Resulting VHDL code

```
level1 <= level0 & "0" when S(0)='1'
          else "0" & level0;
level2 <= level1_d1 & "00" when S_d1(1)='1'
          else "00" & level1;
level3 <= level2_d1 & "0000" when S_d2(2)='1'
          else "0000" & level2;

(...)
```



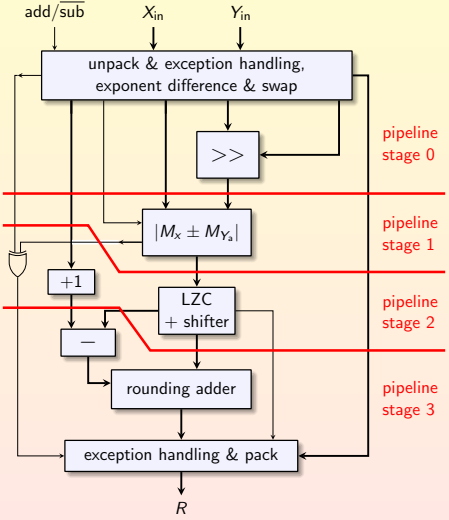
Frequency-directed pipelining is compositional



Assembling components working at frequency f gives a component working at frequency f.

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

Frequency-directed pipelining is compositional



Assembling components working at frequency f gives a component working at frequency f.

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

```
   *** Final report ***
|---Entity RightShifterSticky24_by_max_26_Freq200_uid4
| R: (c0, 3.930000ns) Sticky:  (c1, 1.420000ns)
|---Entity IntAdder_27_Freq200_uid6
| R: (c1, 3.230000ns)
|---Entity Normalizer_Z_28_28_28_Freq200_uid8
| Count:  (c2, 3.470000ns) R: (c2, 4.020000ns)
|---Entity IntAdder_34_Freq200_uid11
| R: (c3, 1.110000ns)
Entity FPAdd_8_23_Freq200_uid2

R: (c3, 2.210000ns)
```

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

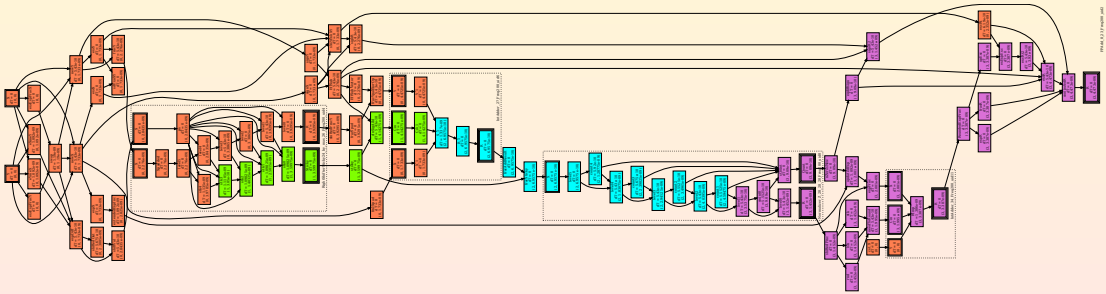
The higher the frequency, the deeper the pipeline

... and the more painful the drawing.

Fortunately, there are options that produce figures...

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

creates a `dot/` directory containing this:



Mostly automatic

- You describe your combinatorial operator in VHDL
  - and you debug it
  - and you optimize it
  - and you debug it again.
- Then you add delay information where needed
  - using methods of the `Target` class, they return a δt (in seconds)
  - example: `Target->adderDelay(int size)` got quite complex on AMD and Intel.
- FloPoCo automatically pipelines and synchronizes
  - and you don't need to debug it! It is correct by construction.
- It is also inaccurate by construction
  - prediction of *routing delay* is in general hopeless...

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 all sorts of **fancy but useful** operations...
- In my humble opinion, this was a visionary paper: submitted to ISFPGA 2008

The engineering foundations to a Scientific Grand Plan

First written in this paper

When FPGAs are better at floating-point than micro

- When? As soon as the processor lacks hardware supp

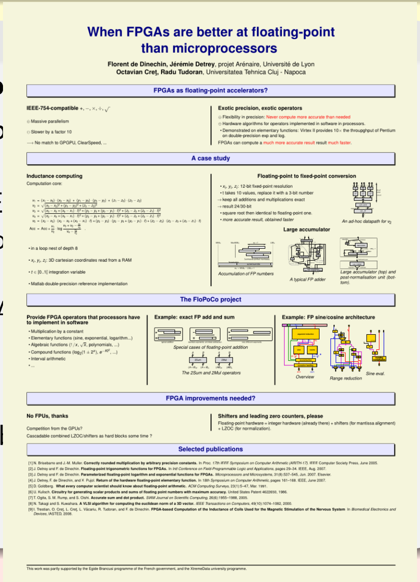
Models	Instruction Distribution					
	Add	Mult.	Div.	Sqrt.	Exp.	Log
bjt	22	30	17	0	2	0
diode	7	5	4	0	1	2
hbt	112	57	51	0	23	18
jfet	13	31	2	0	2	0
mos1	24	36	7	1	0	0
vbic	36	43	18	1	10	4

SPICE

cut fro

Dura /

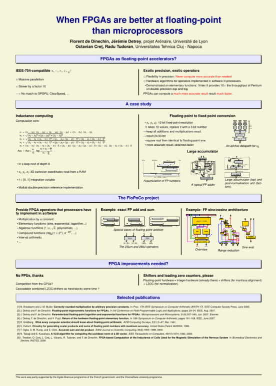
- but also all sorts of **fancy but useful** operations...
- In my humble opinion, this was a visionary paper: su
- Tepid reviews ("prove it", "lack of results")...
- $\Longrightarrow$ poster



Lack of results, prove it

As we all know, a reviewer is always right.

Therefore, we stubbornly wrote



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.

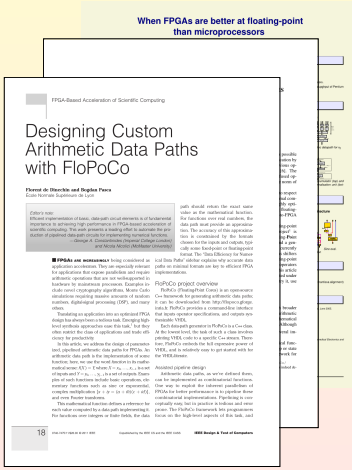
[1] www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/
[2] The only current exception is an accumulator [10] which is indeed designed with a minimal, well isolated feedback loop.

978-1-4244-3892-1/09/$25.00 ©2009 IEEE 59

Lack of results, prove it

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

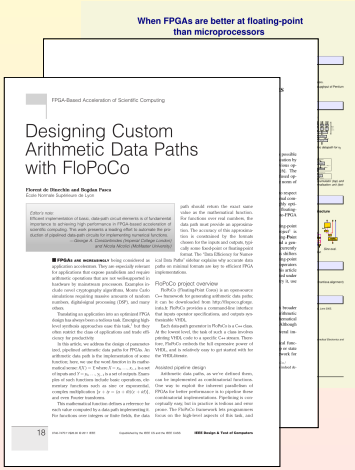
- An FPL 2009 paper:
  Generating high-performance custom floating-point pipelines.
- Its journal version *Designing custom arithmetic data paths with FloPoCo.*
  IEEE Design & Test of Computers, 2011.



Lack of results, prove it

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

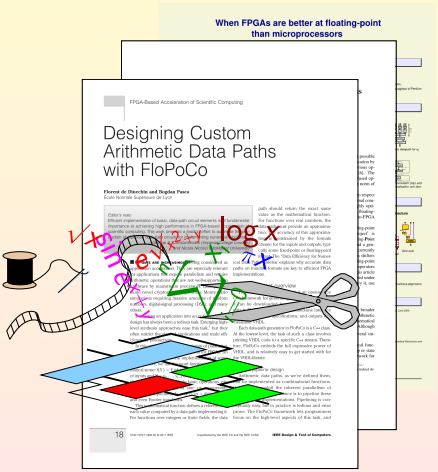
- An FPL 2009 paper:
  Generating high-performance custom floating-point pipelines.
- Its journal version *Designing custom arithmetic data paths with FloPoCo.*
  IEEE Design & Test of Computers, 2011.
  - 400+ GScholar citations,
    TCFPGA Hall of Fame in 2024



Lack of results, prove it

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

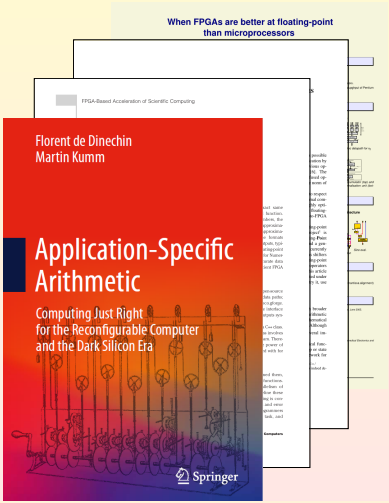
- An FPL 2009 paper:
  Generating high-performance custom floating-point pipelines.
- Its journal version *Designing custom arithmetic data paths with FloPoCo.*
  IEEE Design & Test of Computers, 2011.
  - 400+ GScholar citations,
    TCFPGA Hall of Fame in 2024
  - How many people actually read it?
    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 in 2024
  - 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.



Message to a younger audience

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

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

The curse of success

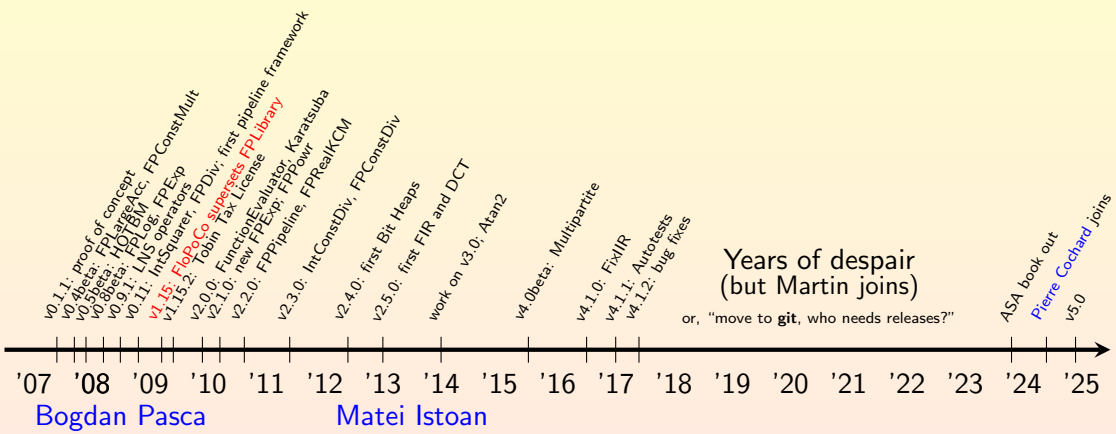
A taxonomy of contributors

Thanks to all contributors (under the dictatorship of **Martin Kumm** and **myself**)

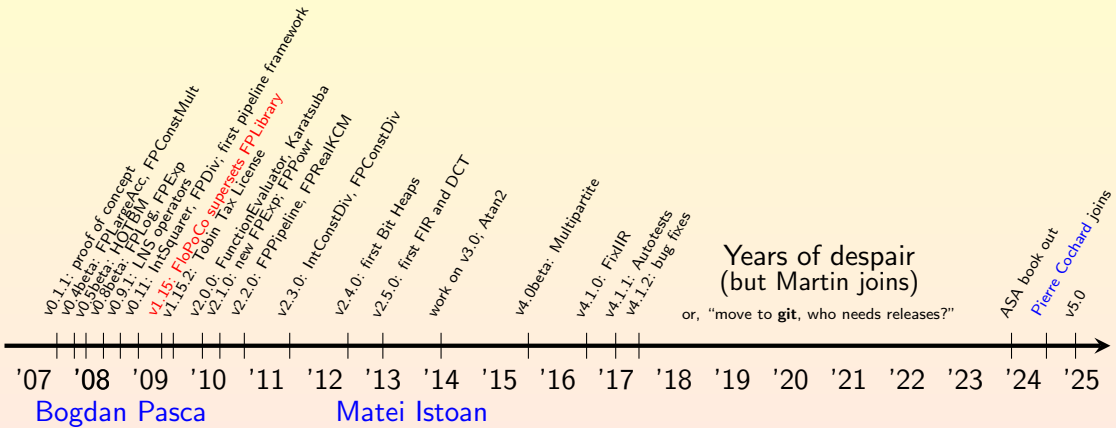
H.Abdoli, S. Banescu, L. Besème, A. Böttcher, N. Bonfante, N. Brunie, R. Bouarah, V. Capelle, M. Christ, P. Cochard, C. Collange, Q. Corradi, O. Desrentes, J. Detrey, A. Dudermel, P. Echeverría, F. Ferrandi, N. Fiege, L. Forget, M. Grad, M. Hardieck, V. Huguet, T. Hubrecht, K. Illyes, M. Istoan, M. Joldes, 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.

- have shown that something is possible (but then I had to throw away all their code) versus their code will be cherished forever
- write code just to get an article published versus write code in the hope that somebody will use it
- C++ gurus versus C++-is-C-with-a-better-printf
- mathematicians with an interest in hardware versus harware engineers with an interest in maths
- people we hired versus strangers from outer space

FloPoCo release timeline

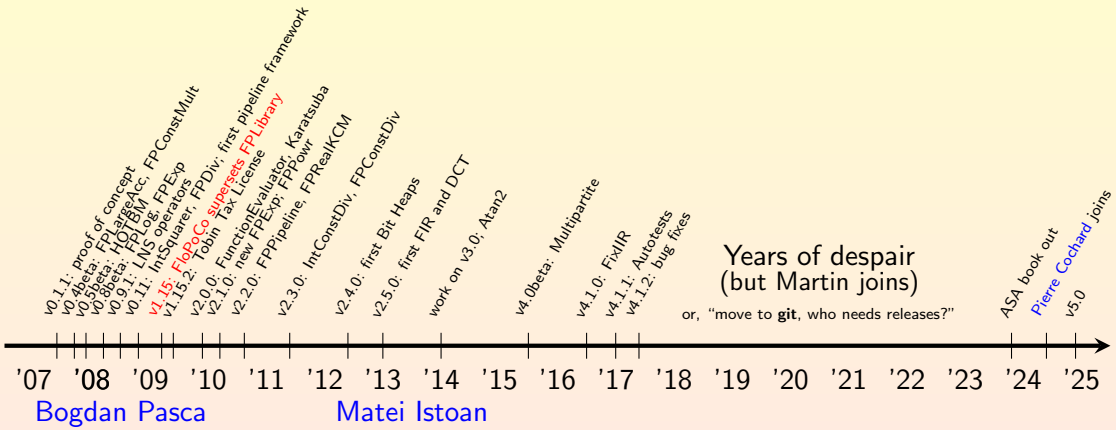


FloPoCo release timeline



More and more distended: need to maintain old stuff!

FloPoCo release timeline



More and more distended: need to maintain old stuff!

And we failed! See the "Orphaned operators" page

Conclusion

Fantastic arithmetic beasts (and where to find them)

A short and biased history of the project

The curse of success

Conclusion

All you ever wanted to know about division by 3

Other stuff we are proud of

- A SotA framework for fixed-point summation (bit heaps)
  (see our ARITH 2025 keynote)
- A SotA tiling approach to multiplier construction
- SotA *Constant Multiplication* (SCM, MCM, CMM, etc.)
- ILP (integer linear programming) for all sorts of arithmetic optimizations
- IIR filters guaranteed without limit-cycle oscillations

And computing just right

Specifying the output format specifies the accuracy:

- no point in computing more accurately: we could not express it;
- no point in computing less accurately: we would output meaningless bits.

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

Example of bug report by a highly valued FloPoCo user

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

Can you see what is wrong here?

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

Example of bug report by a highly valued FloPoCo user

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

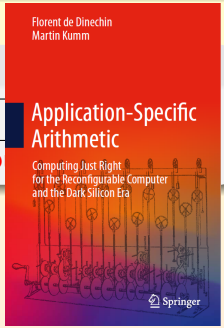
Can you see what is wrong here? $\approx 1/3 \pm 2^{-14}$ Argh! Not Computing Just Right!

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

Example of bug report by a highly valued FloPoCo user

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

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



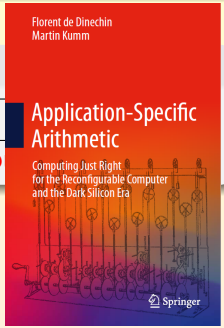
Solution 1: Did I mention that we published this book?

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

Example of bug report by a highly valued FloPoCo user

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

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



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

Solution2

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

(this means a C to hardware compiler, haha)

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

Why move useless bits around?

FloPoCo only solves the easy problem

DONE Good, flexible, versatile application-specific operators

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

Questions? A demo?

All you ever wanted to know about division by 3

Fantastic arithmetic beasts (and where to find them)

A short and biased history of the project

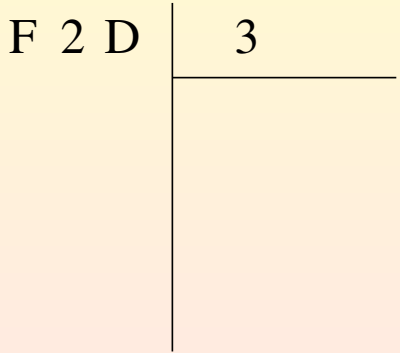
The curse of success

Conclusion

All you ever wanted to know about division by 3

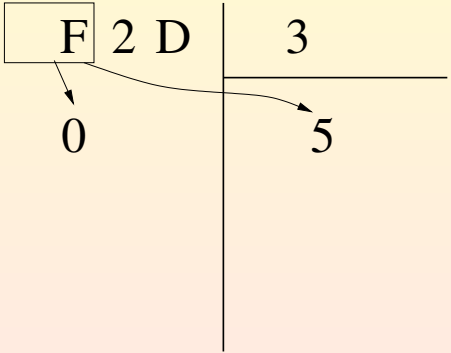
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3



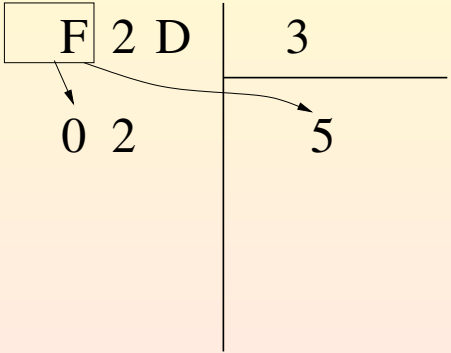
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3



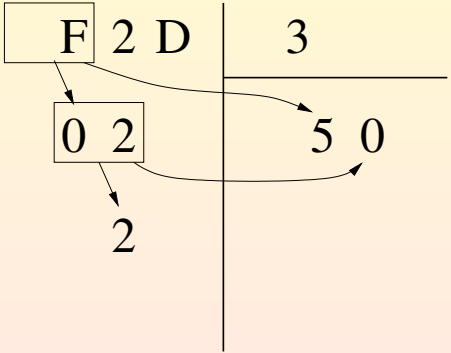
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3



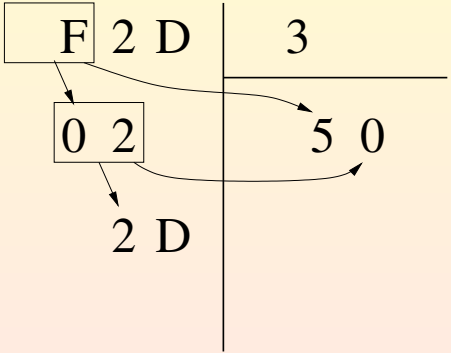
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3



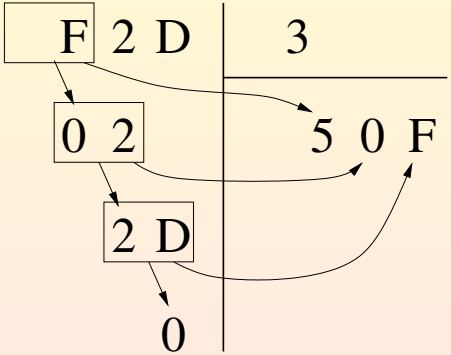
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3

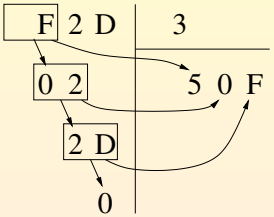


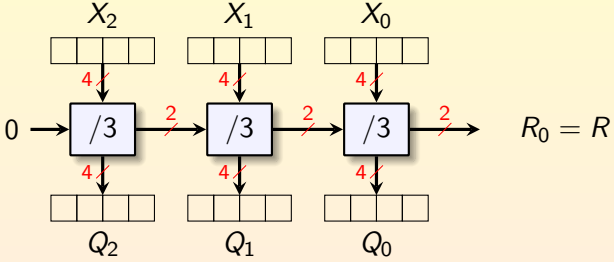
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3

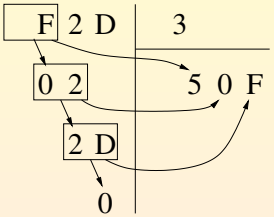


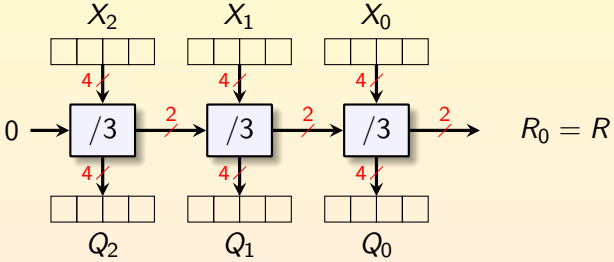
Division by 3 should not be more complex than multiplication by 3





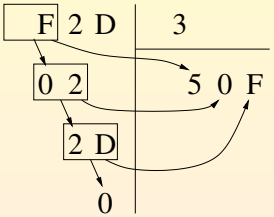
Division by 3 should not be more complex than multiplication by 3

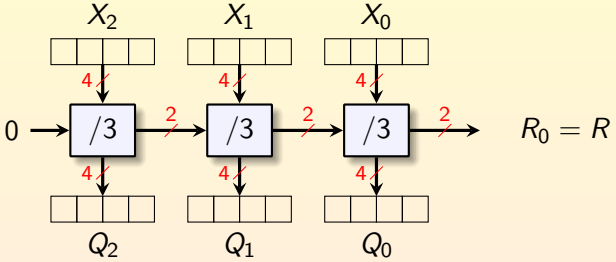




OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

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





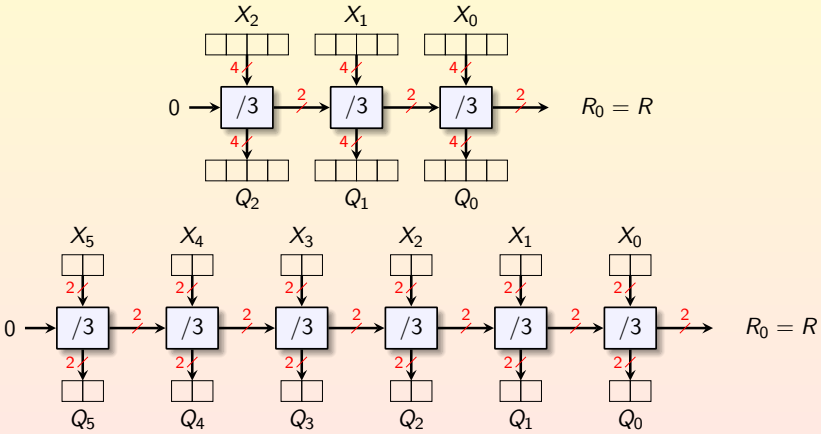
OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

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

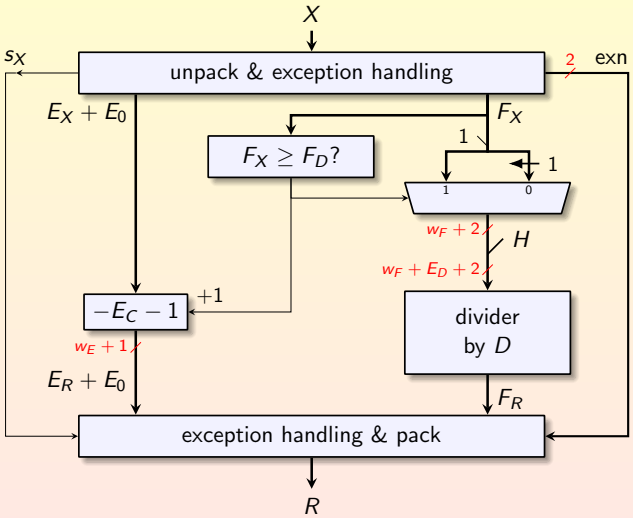
... here a table of 2^6 entries of 6 bits each.
(small enough to be called a truth table and submitted to synthesis tools)

Then FloPoCo does clever things

Two equivalent architectures, targetting Zynq LUT6 (above) and Efinix LUT4 (below)



More good surprises in the floating-point divider by 3



Low latency thanks to pre-normalisation and pre-rounding:

$$\left\lfloor \frac{2^{s+\epsilon} m}{d} \right\rfloor = \left\lfloor \frac{2^{s+\epsilon} m}{d} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{2^{s+\epsilon} m + d/2}{d} \right\rfloor$$

(I agree it is not obvious, but it saves one large adder on the critical path)

What, your taxpayer money is being wasted on stupid division by 3?

(of course the technique works for various values of 3)

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

- Euclidean integer division (quotient and remainder)
  - round-robin addressing with 3 banks of memory
  - pooling layers in quantized neural networks
  - crypto (with much larger constants – other methods needed for large divisor)
- In floating-point
  - serious linear algebra (Jacobi),
  - stencil applications,
  - pooling layers,
  - etc.