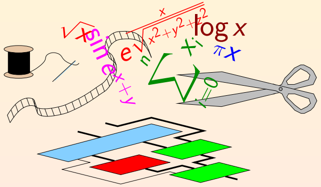
Application-Specific Arithmetic Operators with the FloPoCo HDL core generator

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.



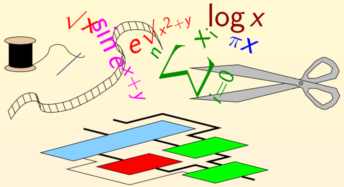
Fantastic arithmetic beasts (and where to find them)

Where to find them? In your applications!

… with a little help of FloPoCo.

www.flopoco.org

From maths
to circuits
with love and care

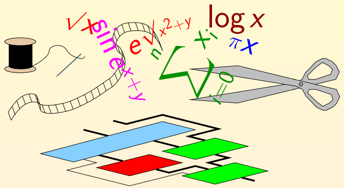


Where to find them? In your applications!

... 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

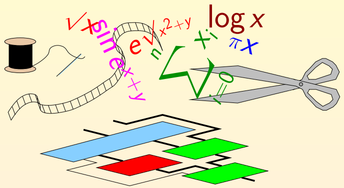
- written in C++, outputting portable, synthesizable VHDL
- open-source, extensible, state of the art

Where to find them? In your applications!

... with a little help of FloPoCo.

www.flopoco.org

From maths
to circuits
with love and care



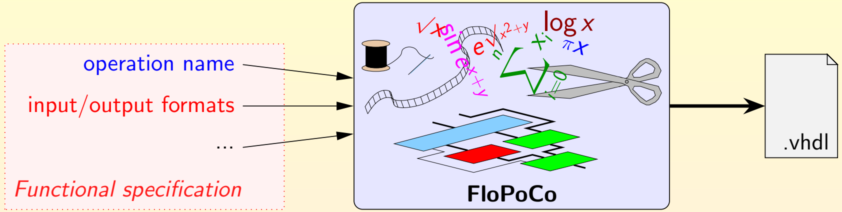
A generator of **application-specific** hardware arithmetic operators

- 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

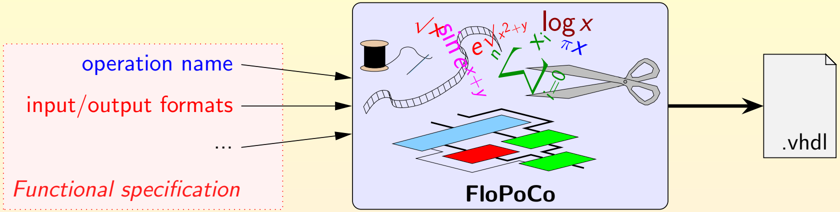
A command-line tool, because we are all geeks here



Start simple: a single precision floating-point adder

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

A command-line tool, because we are all geeks here



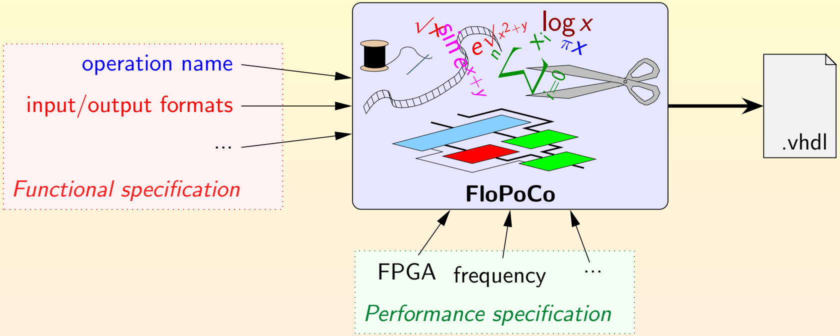
Start simple: a single precision floating-point adder

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

Who needs 24 bits of mantissa (precision 10^{-8}) for HEP instruments?

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

A command-line tool, because we are all geeks here



Start simple: a single precision floating-point adder

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

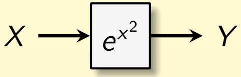
Who needs 24 bits of mantissa (precision 10^{-8}) for HEP instruments? All we want is speed!

```
./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 $[0, 1)$...



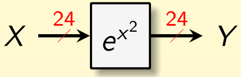
$$\begin{tikzcd} X \arrow[r] & \boxed{e^{x^2}} \arrow[r] & Y \end{tikzcd}$$

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.

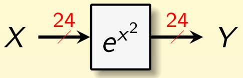


This was not FloPoCo

Suppose you need to evaluate some function,

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

... with inputs and outputs on 24 bits.



There are several ways of doing this in FloPoCo.

Here is one of them.

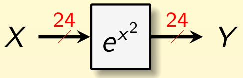
```
./flopoco FixFunctionByPiecewisePoly f="exp(x*x)" lsbIn=-24 lsbOut=-24 d=3
```

This was not FloPoCo

Suppose you need to evaluate some function[1],

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

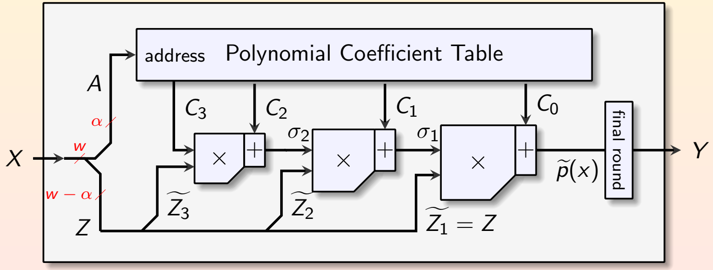
... with inputs and outputs on 24 bits.



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)

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

$\rightarrow$ 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

$\rightarrow$ Clean mathematical specification

An operator, as a *circuit*...

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

- easy to pipeline (no memory, no loop)
- easy to test against its mathematical specification

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

FloPoCo can generate an infinite number of operators

- Obviously, they don't have enough disk space for all of them on Open Logic.

FloPoCo can generate an infinite number of operators

- Obviously, they don't have enough disk space for all of them on Open Logic.
- Obviously, we haven't tested them all.

FloPoCo can generate an infinite number of operators

- Obviously, they don't have enough disk space for all of them on Open Logic.
- Obviously, we haven't tested them all.

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

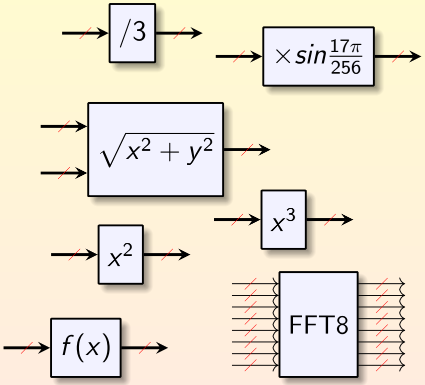
```
./flopoco IntConstDiv wIn=16 d=3 TestBench
```

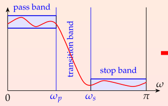
- based on operator(X) = quantization(operation(X))
- implemented as an `emulate()` method
  - a few lines of code only
  - based on trusted arbitrary-precision numerical libraries (MPFR, Sollya)
  - written first, and easy to audit (test-driven development)
  - produces a `test.input` file readable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

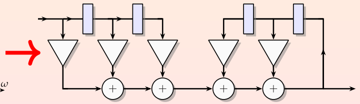
(We do have a CI based on this! Only, no pretense to full coverage)

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
  Other examples: SCM, MCM, CMVM...
  (Chang Sun's talk yesterday)
- 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

- 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

- 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
```

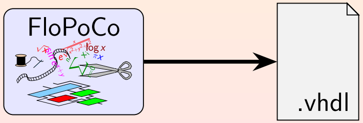
```
./flopoco
```

Open source! Contributions welcome!

(here I should recycle all the slides of Oliver Bründler tuesday about Open Logic)

`http://flopoco.org/`

- Install from source (sorry) but it recently became easy
  - despite the fact that we pull MPFR, Sollya, 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**



All you ever wanted to know about division by 3

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 wE=8 wF=23 d=3
```

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 wE=8 wF=23 d=3
```

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 wE=8 wF=23 d=3
```

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"
```

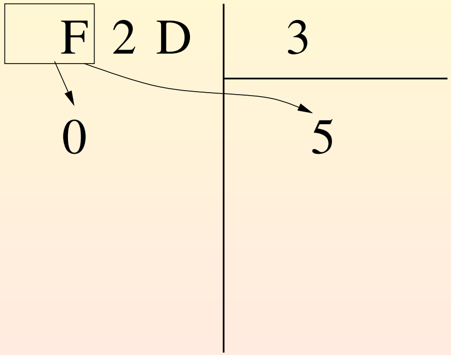
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3

$$\begin{array}{r|l} \mathrm{F}\ 2\ \mathrm{D} & \quad 3 \\ \cline{2-2} & \end{array}$$

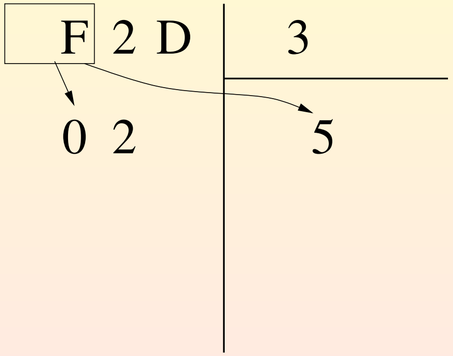
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3



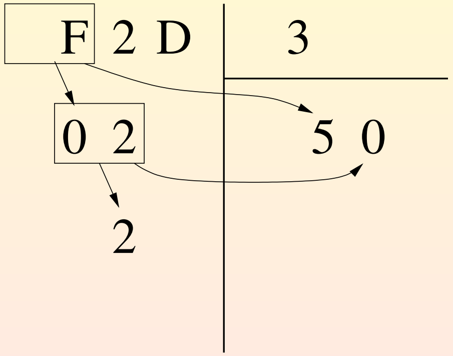
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3



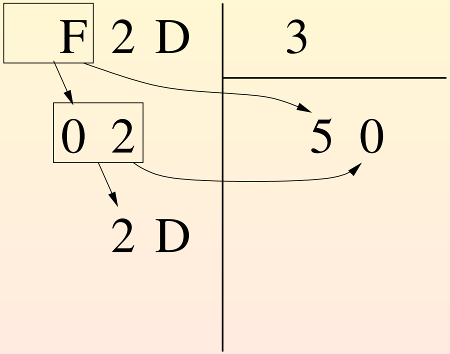
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3



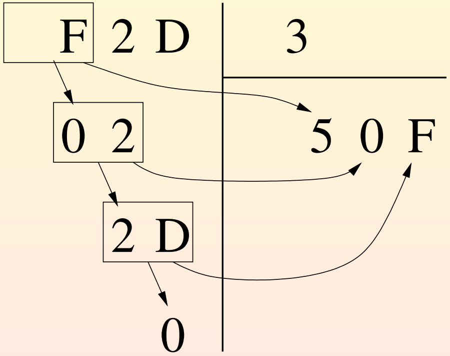
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3

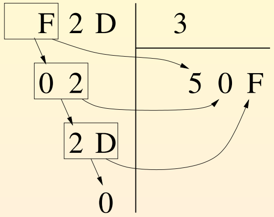


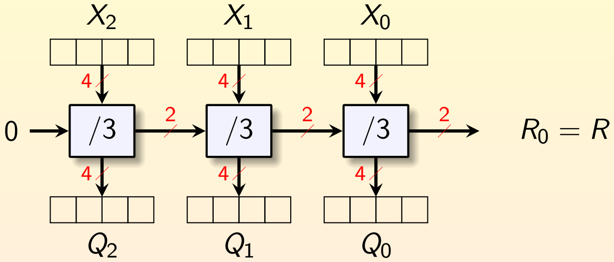
Raise your hand if you think you would be able to divide 3885 by 3?

Dividing an **hexadecimal** number by 3

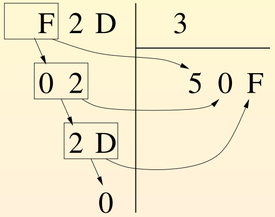


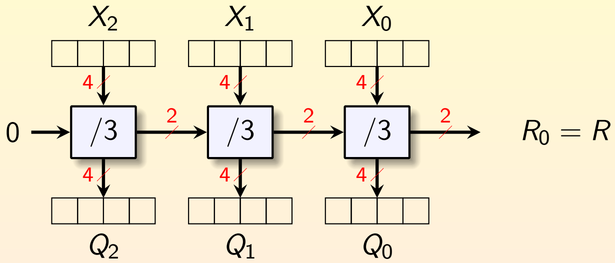
Division by 3 should not be more complex than multiplication by 3





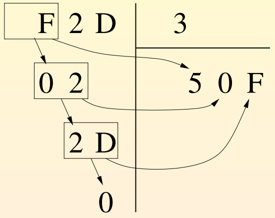
Division by 3 should not be more complex than multiplication by 3

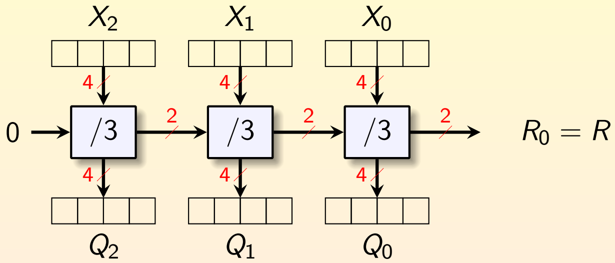




OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

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





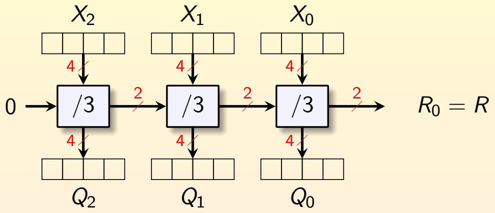
OK, this looks like an architecture, but we still need to build this (smaller) DivBy3 box.

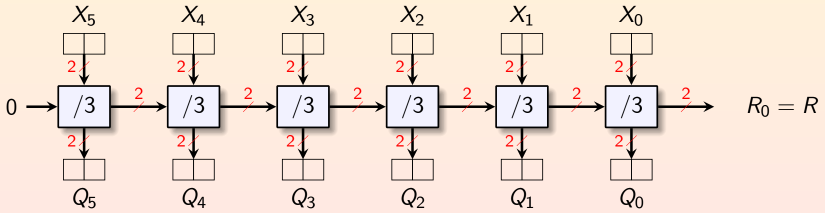
If you don't know how to compute it, then tabulate it

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

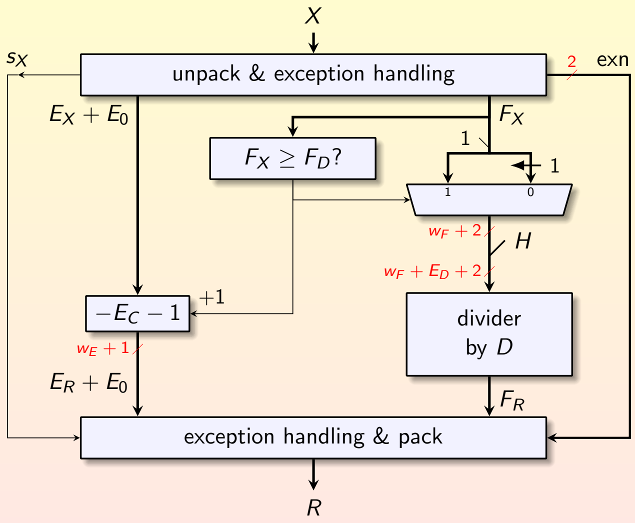
Then FloPoCo does clever things

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:

$$\left\lfloor \frac{2^{s+\epsilon} m}{d} \right\rceil$$

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

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

(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.

Inside FloPoCo

Combinatorial circuits in FloPoCo: just print VHDL

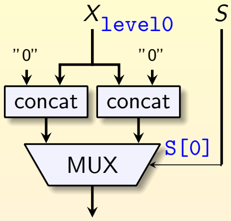
C++ code for a **Barrel Shifter**

```
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;
```

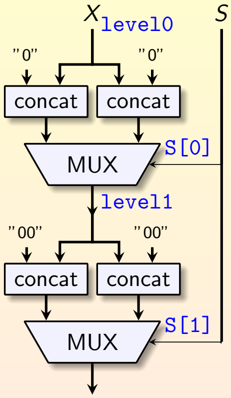
Combinatorial circuits in FloPoCo: just print VHDL

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

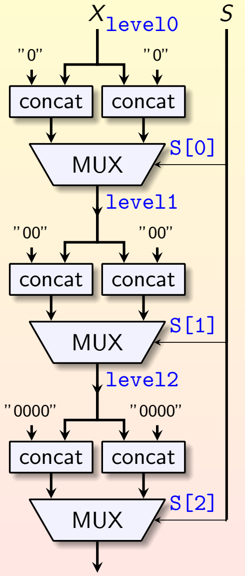
C++ code for a **Barrel Shifter**

```
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: automatic

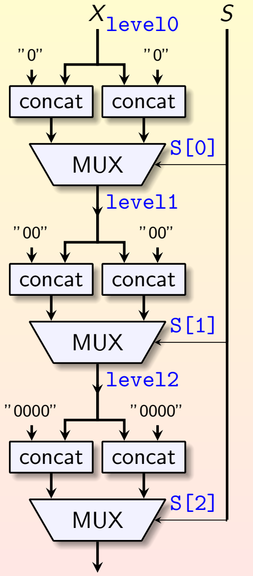
C++ code for a **Barrel Shifter**

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

(...)
```



Pipelined circuits in FloPoCo: automatic

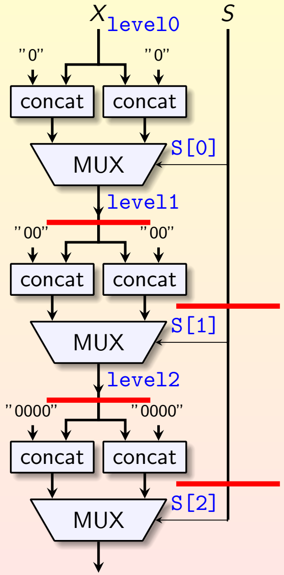
C++ code for a **Barrel Shifter**

```
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<< ";";
}
```

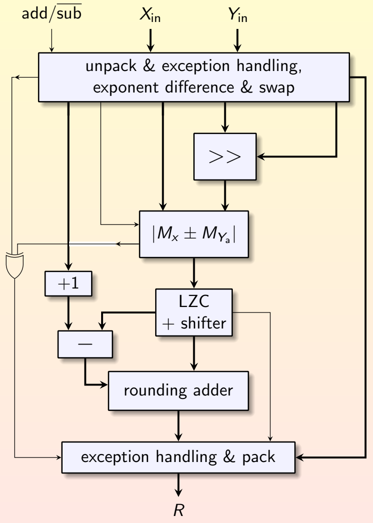
Resulting VHDL code

```
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;

(...)
```

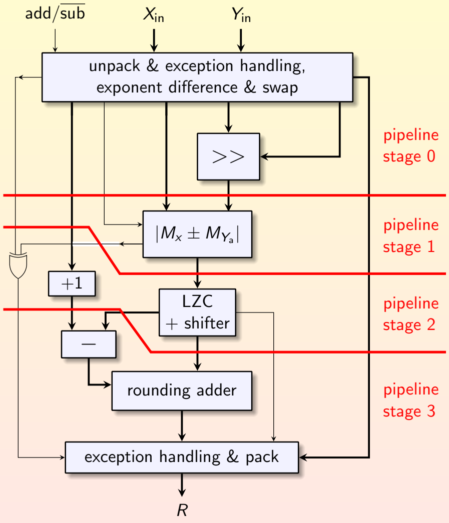


Frequency-directed pipelining is compositional


add/$\overline{\text{sub}}$
X_{in}
Y_{in}
unpack & exception handling, exponent difference & swap
$>>$
$|M_x \pm M_{Y_a}|$
$+1$
LZC + shifter
$-$
rounding adder
exception handling & pack
R

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

Frequency-directed pipelining is compositional



```
./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

An assembly of components working at frequency f
is a component working at frequency f.

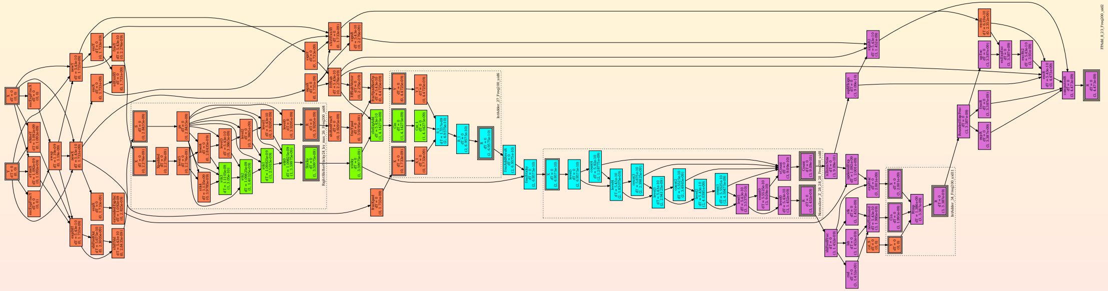
The higher the frequency, the deeper the pipeline

... 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.
- Prediction of *routing delay* is in general hopeless

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
- A logarithmic neuron for machine learning

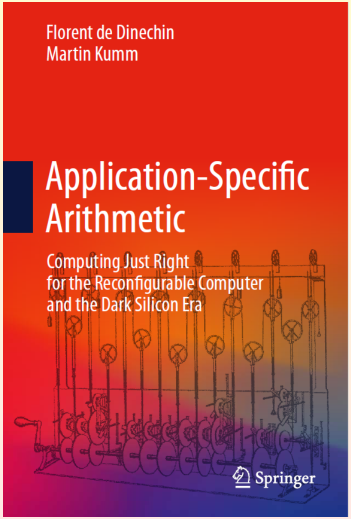
And computing just right

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.

Conclusion

Bibliography (or maybe advertising)


Florent de Dinechin
Martin Kumm
Application-Specific Arithmetic
Computing Just Right for the Reconfigurable Computer and the Dark Silicon Era
Springer

Our book is finally out !
800 pages of fancy hardware arithmetic.

Also (and for free), you can find articles
about most of the subjects described in these slides
on the web page of FloPoCo:

`http://flopoco.org/`

More references there...

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
```

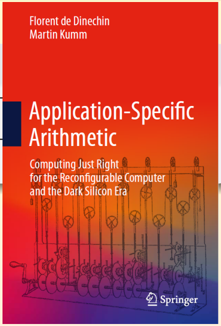
Can you see what is wrong here? $\approx 1/3 \pm 2^{-14}$ Argh! Not Computing Just Right!

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


Florent de Dinechin
Martin Kumm
Application-Specific Arithmetic
Computing Just Right for the Reconfigurable Computer and the Dark Silicon Era
Springer

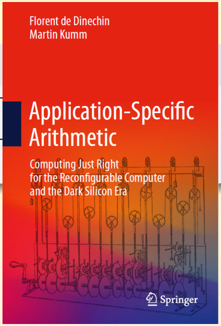
Solution 1: Did I mention that we published this book?

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


Florent de Dinechin
Martin Kumm
Application-Specific Arithmetic
Computing Just Right for the Reconfigurable Computer and the Dark Silicon Era
Springer

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?