FloPoCo
A generator of non-standard floating-point operators for FPGAs

RAIM, June 3-5, 2008

Bogdan PAȘCA
Cristian KLEIN
(Florent de DINECHIN)
projet Arénaire, ENS-Lyon/INRIA/CNRS/Université de Lyon

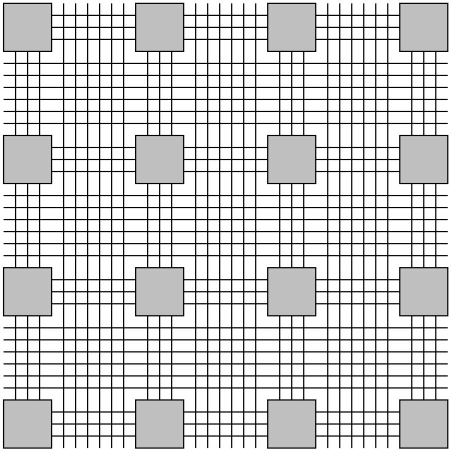
Outline of this presentation

Why:	Floating-point opportunities in an FPGA
What:	Examples of current operators
How:	Not an operator library, but a generator of operators
When:	Conclusion and open questions

`http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

Why?

What are FPGAs?



- reconfigurable logic
- CLB
- routing grid

When are FPGAs really good at floating-point ?

IEEE-754 $+, -, \times$?

- $\oplus$ Massive parallelism on an FPGA
- $\ominus$ Each operator **10x slower** than the processor's
- $\longrightarrow$ Faster than a PC, but no match to GPGPU, Cell, ClearSpeed, ...

Double-precision floating-point logarithm?

- Pentium: 130 cycles @ 2GHz
- FPGA: Specific combinatorial architecture
  - less than 1/10 the area of a large FPGA (and logic only)
  - arbitrarily pipelinable
  - One log per cycle @ 200MHz: **10x faster** than the processor

$\sqrt{x^2 + y^2 + z^2}$?

NOT Economical in a Processor

- Algebraic functions ($1/x$, $\sqrt{x}$, $\dfrac{1}{\sqrt{x^2+y^2+z^2}}$, polynomials, ...)
- Elementary functions (sine, exponential, logarithm...)
- Compound functions ($\log_2(1 \pm 2^x)$, e^{-Kt^2}, ...)
- Floating-point sums, dot products, sums of squares
- Complex arithmetic
- LNS arithmetic
- Decimal arithmetic
- Interval arithmetic
- ...
- Oh yes, basic operations, too, but with a bit of pepper
  - output precision may be different from input precision
  - optimized special cases such as multiplication by a constant

Non-standard arithmetic in processors

How about a processor with all these operations?

- It would be very big
- It would be very expensive
- Processor would be underutilised
- Still, precision would be fixed
- How would they be pipelined?

Current approach in processors

- basic operations are very fast
- elementary functions are microcoded (slower)
- complex functions have to be written in software (even slower)

What?

FloPoCo – Not your neighbor's FPU

FloPoCo is a generator of operators

- written in C++
- generating portable, synthesizable VHDL
- open-source (LGPL)

`http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

Feel free to try it... and even contribute

High-level goals

- Easy to create arithmetic operators for FPGA
  - with tunable input / output precision
  - automatic pipelining
  - fine-tuned for several FPGA targets
- Easy to combine and reuse in more complex designs
  - e.g. reuse shifter and int multiplier in FP multiplier
  - automatic pipeline adjustment
  - automatic precision augmentation
- Automatic test-case generation
  - operator depending testing
  - also allow random / exhaustive testing
  - validated using software arithmetic

What we have now

- Integer Adder / Multiplier
- FP Adder / Multiplier
- Integer / FP constant multiplier
- Long accumulator
- FP Exponential (not pipelined)
- FP Logarithm (not pipelined)
- HOTBM (fixed-point) (not pipelined)
  - $sin(x \cdot \pi/4)$
  - $log(1+x)$
- Automatic test-bench generation for all these operators

Usage

```
flopoco -pipeline=yes -frequency=240 -DSP_blocks=no
FPMultiplier 8 23 8 23 8 23 1 IntAdder 16
```

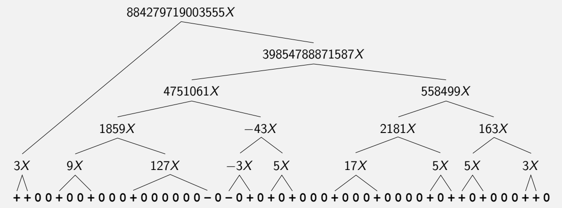
- Command line syntax: a sequence of operator specifications
  - performance specifications (for optimization)
  - operator name and parameters
- **Output**: a **single** VHDL file for **all** the operators (flopoco.vhdl).

Example 1: constant multipliers

Integer

- CSD recoding
- balanced add tree (DAG actually)
- small, fast, no DSP block

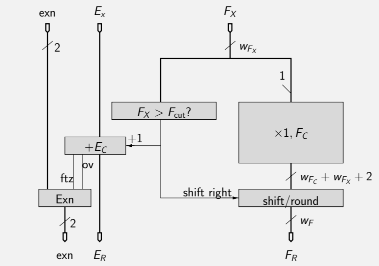
Example: *Pi*/4 on 50 bits



Floating-point

Compared to using a standard FP multiplier:

- smaller mantissa multiplication
- earlier normalization decision



An FP constant multiplier by Pi

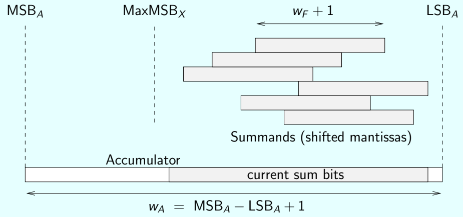
```
> flopoco FPConstMult 11 52 11 52 0 -52 14148475504056881
```

Output:
```
> FPConstMult, wE_in=11, wF_in=52, wE_out=11,
wF_out=52, cst_sgn=0, cst_exp=-52,
cst_sig=14148475504056881
  Number of adders:  16
Output file:  flopoco.vhdl
Final report:
Entity IntConstMult_52_14148475504056881:
  Not pipelined
Entity FPConstMult_14148475504056881bM53_11_52_11_52:
  Not pipelined
```

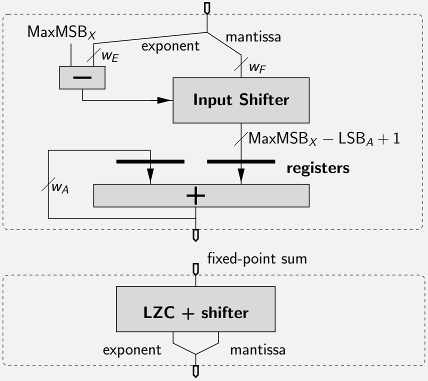
	FPConstMult	**FPMultiplier**	**FPMultiplier+DSP**
Slices	512 (8%)	1608 (26%)	1112 (18%)
4-input LUTs	1000 (8%)	3191 (25%)	478 (3%)
Latency	14ns	21ns	4ns

Example 2: Accumulation of floating-point values

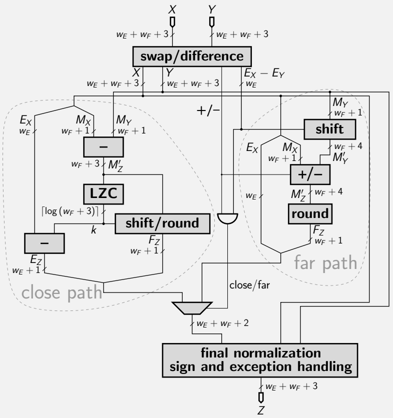
Tailor accumulator to application



Accumulator & post-normalisation unit



FP adder is wasteful



- Provably **avoid** all **rounding errors** in the accumulation
- Only local routing in accumulator
- **Arbitrary frequency** using partial carry-save in the accumulation
  - cut the carry propagation every 30 bits for 400 MHz operation
- Also used in an exact dot-product operator, and more to come.

Example 3: approximation of fixed-point functions

A faithful 15-bit fixed-point approximation of some function on $[0, 1]$

```
flopoco HOTBM "exp(x*x)" 15 15 4
```

```
(...)
Tested 7571 designs.
Best design:  15 15 4 3 rom 3 0 powmult 3 12 adhoc 12 12 2 0 3
6 3 6 powmult 3 12 rom 11 1 1 3 8 3 3 powmult 3 11 rom 11 1 0
3 11 powmult 3 11 rom 12 2 0 3 6 3 6
Score:  493821593
Output file:  flopoco.vhdl
```

- Clever table-based approach, generated VHDL file is 270 KB
- Useful building block for FP elementary functions
- Arbitrary mathematical expression accepted as an input
- FloPoCo uses Sollya for expression parsing and Remez algorithm.

J. Detrey and F. de Dinechin. **Table-based polynomials for fast hardware function evaluation**. In *ASAP 2005*. IEEE.

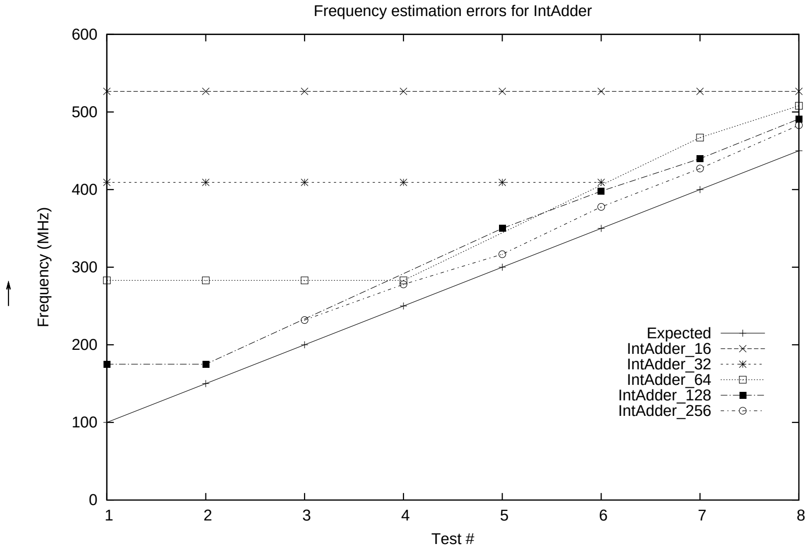
Automatic pipelining

```
flopoco -frequency=220 -DSP_blocks=no FPMultiplier 8 23 8 23 8 23
Entity FPMultiplier_8_23_8_23_8_23:
Pipeline depth = 9
Minimum period:  4.063ns (Maximum Frequency:  246.105MHz)
```

```
flopoco -frequency=330 -DSP_blocks=no FPMultiplier 8 23 8 23 8 23
Entity FPMultiplier_8_23_8_23_8_23:
Pipeline depth = 13
Minimum period:  2.738ns (Maximum Frequency:  365.230MHz)
```

- Very good match with synthesis reports (default target is Virtex4)
- Other operators need more fine tuning
- For now, geared towards high frequencies

Automatic pipelining frequency estimates



Testbench generation

Comes in two flavors:

Basic Testing

```
>flopoco FPLog 8 16 TestBench 10000
```

- Generates 10000 random test cases, plus a few standard ones
- Puts tests directly in VHDL
- Works well for up to 100.000
- Failures are easy to follow

Soak Testing

```
>flopoco FPLog 8 16 BigTestBench 10000000
```

- Puts test vectors into a separate file
- Works for (at least) 10.000.000
- Failures are hard to follow (especially for pipelined designs)

How?

What's wrong with VHDL

- FloPoCo is a **generator** of VHDL floating-point operators for FPGAs.
- It supersedes FPLibrary, a *library* of VHDL floating-point operators for FPGAs.
- Experience with FPLibrary:
  - unpleasant to install (too bulky) – bad design probably
  - a lot of work to pipeline, then pipeline not flexible
  - ugly code (too many parameters) for complex operators, and even simple ones (recursive synthesisable VHDL)
  - no real design space exploration possible

End of 2007, we had several bits of Maple, OCaml, bash and C++ code generating various bits of FPLibrary

courtesy of Jérémie Detrey, Xavier Pujol, Florent de Dinechin

What's nice with generators?

- Many parameters, clean VHDL code
  - For the user: faster compilation and easier debug
- Area/time/accuracy: one size does not fit all
  - optimal design may depend on architectural parameters of target FPGA (e.g. number of LUT inputs)
  - automatic, constraint-oriented pipeline generation
  - ...
- High-level input specification
  - `flopoco HOTBM "exp(x*x)" 15 15 4`,
    function specified as an arbitrary expression
- Complex design-space exploration
  - `Tested 7571 designs.`

A modestly object-oriented approach

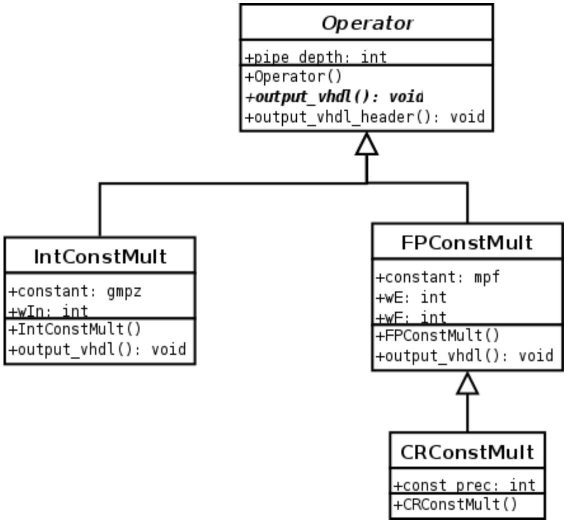
FloPoCo is not a C++-based HDL

- VHDL generation is **printf-based**
- Many helper functions help doing the `printfs`
- But that's mostly all
  - Signals, wires, ... are just syntactic objects
  - No notion of connecting two wires, building an architecture, ...
  - No notion of parallelism, sequentiality, synchronism, ...
  - No checking that no wire is left unconnected...

OO: The `Operator` hierarchy

- The constructor explores the design space and computes all the required internal parameters
- `output_vhdl()` does a lot of `printf`
- Other methods and attributes mostly to **avoid duplicated work**
  - `output_vhdl_header()`
  - `output_licence()`
  - `add_register()`, `output_vhdl_registers()` etc:
    - signals are still just syntactic objects
    - but it saves a lot of `printf`s to declare them this way
    - and a lot of mismatch bugs, and it makes tuning faster etc.
- Methods for automatic **pipeline generation**
  - `set_pipeline_depth()`, `get_pipeline_depth()`
- Methods for automated, parametric **test generation**
  - `getTestIOMap()`, `fillTestCase()`

Inheritance example: constant multipliers



OO: The `Target` hierarchy

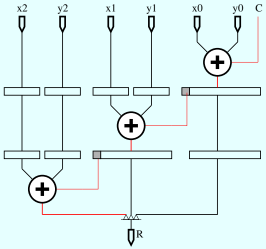
- **Architecture-related** methods and attributes
  - `lut_inputs();`
  - `suggest_submult_size(...);`
  - ...
- **Delay-related** methods and attributes (for pipeline evaluation)
  - `lut_delay()`
  - `adder_delay(int n)`
  - ...
- Methods and attributes related to target **performance and behaviour**, set by the command line
  - `set_pipelined()`
  - `set_frequency()`
  - `use_hard_multipliers()`
  - ...

(currently only Virtex4 operational, Stratix II soon to come)

Automatic pipeline generation

Typical algorithm: rough and greedy

- from the inputs to the outputs,
- **accumulate critical path delay estimates** (using the methods of `Target`)
- **insert registers** to keep this delay below 1/`target_frequency`

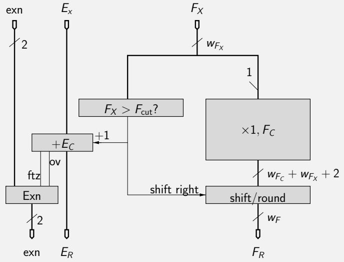


What we get?

- A fair idea of the number of pipeline levels needed for a given operator and a given frequency
- Registers placed reasonably
  - maybe unoptimal results using vanilla vendor tools
  - but good starting point for retiming tools

Pipeline generation for composed operators

Example: FP constant multiplier



FPConstMult instantiates an IntConstMult

- constructor of `FPConstMult` calls constructor of `IntConstMult`
- It calls `IntConstMult::get_pipeline_depth()`
- `FPConstMult` delays signals accordingly (`add_delay_signal()`)
- Both `output_vhdl()` called in order.

Conclusions

Not bad for a 4-month old project

Overall the approach works well, but still early beta

- Few operators really finished
- The devil is in the details
- It's funnier to explore new operators than to polish existing ones
- Some previous code waiting at the door
- Pipelining is still a lot of work

Directions?

Short term

- Add at least one **Altera target**, and **stabilize pipelining infrastructure**
- FPMultiplier pipeline works great for high frequencies, could be improved on the smaller-and-lower end
- Finish FPAdder
- Explore FFDiv and FPSqrt
- pipeline FPLog

Longer term

- Endless list of operators
- Explore using FloPoCo's infrastructure to assemble larger pipelines
- Explore direct interface to some C-to-hardware tool?
- Add generation of formal proofs of numerical quality
- <Insert your request here>

Questions?

Thank you for your attention.

`http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

Bibliography

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.

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.

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.

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.

D. Goldberg. **What every computer scientist should know about floating-point arithmetic.**
ACM Computing Surveys, 23(1):5–47, Mar. 1991.

U. Kulisch. **Circuitry for generating scalar products and sums of floating point numbers with maximum accuracy.**
United States Patent 4622650, 1986.

T. Ogita, S. M. Rump, and S. Oishi. **Accurate sum and dot product.**
SIAM Journal on Scientific Computing, 26(6):1955–1988, 2005.

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.

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.