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

---

[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.

quickly developing high-performance pipelined operators out of combinatorial ones. This is a major contribution of this work and will be exposed in Section 3.2.

Secondly, when an operator is defined as the implementation of a mathematical function, the error analysis and design-space exploration may be much more accurate than when it is the implementation of a C program [6].

Finally, the mathematical function also defines a reference for testing, as will be detailed in Section 4.

1.3. From libraries to generators

Although early FP operators were proposed as VHDL or Verilog libraries, the current trend is to shift to *generators* of operators (see [11, 12] and references therein). A generator is a program that inputs user specifications, performs any relevant architectural exploration and construction (sometimes down to pre-placement), and outputs the architecture in a synthesizable format. Most FPGA designers are familiar with the Xilinx core generator tool or its Altera MegaWizard counterpart. A generator may simply wrap a library – and for the simplest operators there is no need for more – but it can also be much more powerful.

Firstly, the size of a library including multipliers by all the possible constants would be infinite, but the generation of an architecture for such a multiplier may be automated as a finite program that inputs the constant [5]. The same holds for arbitrary function generators such as HOTBM [9], now in FloPoCo, and other generic operators.

Secondly, generators allow for greater parametrization and flexibility. Whether the best operator is a slow and small one or a faster but larger one depends on the context. FPGAs also allow flexibility in precision: arithmetic cores should be parametrized by the bit-widths of their inputs and outputs. We are also concerned about optimizing for different hardware targets, with different LUT structure, memory and DSP features, etc. The more flexible a generator, the more future-proof.

Finally, generators may perform arbitrarily complex design-space exploration, automated error analysis, and optimization [9, 2].

1.4. Design choices for FloPoCo

A generator could in principle be written in VHDL or Verilog, but these languages are not ideally suited for that. In particular, complex VHDL/Verilog programs turn out to be inefficient and difficult to debug due to the simulation/synthesis paradigm of hardware description languages. Still, an architecture generator needs a back-end to actually implement the resulting circuit. The most elegant solution is to write a generator as an overlay on a software-based HDL such as SystemC, JBits, HandelC or JHDL (among many others). The advantages are a preexisting abstraction

of a circuit, and simple integration with a one-step compilation process. However, most of these languages are still relatively confidential and restricted in the FPGAs they support. Even SystemC synthesizers are far from being commonplace yet. Basing our generator on a vendor generator would be an option, but would mean restricting it to one FPGA family.

FloPoCo took a less elegant, but more universal route. The generator is written in a mainstream programming language (we chose C++, mostly for compatibility with other previous work), and it outputs operators in a mainstream HDL (we chose standard synthesizable VHDL). We thus get a portable generator, and the generated operators can be integrated into most projects, simulated using mainstream simulators, and synthesized for any FPGA using the vendor back-end tools. Section 3.3 will discuss how they can nevertheless be optimized to a given FPGA target.

2. A SIMPLE MOTIVATING EXAMPLE

This framework has been used to write and pipeline very large components, such as the floating-point logarithm described in [2]. We choose to discuss here a simpler, but still representative example: the implementation of a collision detection predicate, testing if the norm of a 3D vector is larger than some threshold. Let x, y, z and $r2$ be four floating-point inputs, and let us design an operator testing whether $x^2 + y^2 + z^2 < r2$.

A first option is to assemble three FP multipliers and two FP adders. The final comparison can be performed by an integer subtraction on the concatenation of the exponent and the mantissa – a well-known property of IEEE-754 FP formats is that positive FP numbers are ordered in the lexicographic order of their binary representation.

Let us now design a specific operator for this computation. There are many optimization opportunities with respect to the previous solution.

1/ The squarers should be simpler than multipliers. They will, in particular, use less DSP blocks [4].

2/ We only add positive numbers. In an off-the-shelf floating-point adder, roughly half the logic is dedicated to effective subtraction, and can be saved here. An optimizing synthesizer would probably perform this optimization[3], but it will not, for instance, reduce the pipeline depth accordingly.

3/ The significands of x^2, y^2 and z^2 are aligned before addition. This may be done in parallel, reducing pipeline depth and register usage with respect to an implementation using two floating-point adders in sequence.

4/ A lot of logic is dedicated to rounding intermediate results, and can be saved by considering the compound operator as an atomic one [6]. We obtain an operator that is not

---

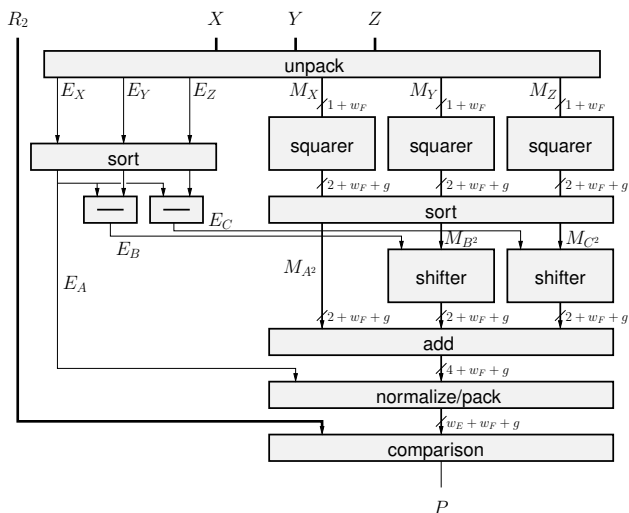[3]A generator of pre-placed/pre-routed cores may miss it.

Fig. 1. Overview of the custom FP collision operator.

only simpler, but also more accurate.

It is typical for a fused operator to be more accurate than the combination of FP ones. Here, the idea is to evaluate $x^2 + y^2 + z^2$ with 1-ulp *(unit in the last place)* accuracy before the comparison, where a combination of standard operators leads to a worst-case error larger than two ulps. More subtly, with an operator built by assembling two FP adders, there are some rare cases when the value of the predicate will change when one exchanges x and z. This is due to the two consecutive roundings. We will be able to design an operator which doesn't have such asymmetries. It is even possible to design an *exact* predicate (performing no rounding before the comparison), although this doesn't really make much sense unless x, y and z are computed exactly themselves.

The chosen architecture is depicted on Figure 1. For more details the interested reader is invited to have a look at the `Collision` operator in the FloPoCo distribution. This architecture computes the three squares, truncates them to $w_F + g$ bits (w_F being the mantissa fraction size, and g a number of guard bits discussed below), then aligns them to the largest one and adds them. Worst-case error analysis shows that there are 5 truncation errors in this computation (the three products, and bits discarded by the two shifters). The number of guard bits g is therefore set to $g = \lceil \log_2(5) \rceil = 3$ so that the accumulated error is always smaller than the unit in the last place of $r2$. Comparison is performed as integer subtraction as previously.

Table 1 shows quantitatively the savings brought by this approach. All the metrics are significantly improved (from 25 to 89 %) by operator specialization and fusion. This does not come from a poor implementation of the classical FP operators, as illustrated by the (8,23) LogiCore FP results. And
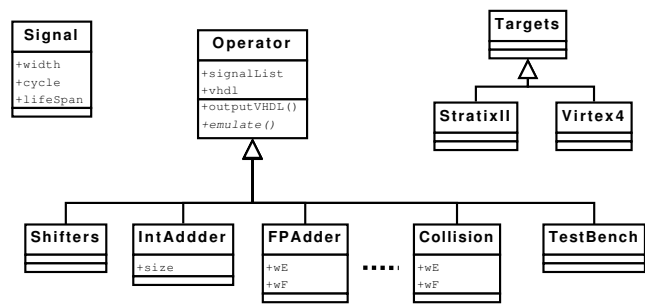

Fig. 2. FloPoCo class hierarchy (very simplified overview)

again, the custom operator is not only smaller and faster, it is also more accurate.

Another meaningful result is design time. Assembling FP operators took two hours, for a pipeline that is fully parametrized by w_E, w_F and target frequency. Assembling the LogiCore version took even less time, but one doesn't obtain a parametrized design. The optimized version took two days to write, including the test-bench. It may seem a lot of design time, but most of the work concerns the sum-of-square operator, which will also be used for future coarser-grain operators such as $\sqrt{x^2 + y^2 + z^2}$ or $\frac{x}{\sqrt{x^2+y^2+z^2}}$. And again, this design is fully parametrized, pipelined, and automatically optimized for a given frequency.

Let us now present in more details the framework used to obtain these results.

3. THE FRAMEWORK

In the following, we assume basic knowledge of object-oriented concepts with the C++ terminology[4]. Figure 2 provides a very simplified overview of the FloPoCo class hierarchy.

3.1. Operators

The core class is `Operator`. From the circuit point of view, an `Operator` corresponds to a VHDL entity, but again, with restrictions and extensions specific to the arithmetic context. All the operators extend this class, including `Collision`, as seen on Figure 1.

The main method of of `Operator` is `outputVHDL()`, which prints out the VHDL code of an operator. To implement this virtual method for an operator, one may simply embed some existing VHDL code in the C++.

However, the standard way of using the framework is to rely on the default implementation of `outputVHDL()` provided by the `Operator` class. This default method takes the VHDL code of the operator architecture from the *vhdl*

---

[4]For an introduction, see `http://en.wikipedia.org/wiki/C++` and references therein.

Precision (w_E, w_F)	Slow version		Fast version	
	area	performance	area	performance
(8,23) LogiCore FP			1282 slices, 20 DSP	43 cycles @ 353 MHz
(8,23) FP	940 slices, 12 DSP	20 cycles @ 210 MHz	1188 slices, 12 DSP	29 cycles @ 289 MHz
(8,23) custom	456 slices, 9 DSP	10 cycles @ 319 MHz	453 slices, 9 DSP	11 cycles @ 368 MHz
(9, 32) FP	1268 slices, 12 DSP	20 cycles @ 171 MHz	1874 slices, 12 DSP	37 cycles @ 302 MHz
(9, 32) custom	629 slices, 9 DSP	10 cycles @ 368 MHz	640 slices, 9 DSP	13 cycles @ 368 MHz
(11, 52) FP	2868 slices, 27 DSP	20 cycles @ 106 MHz	4480 slices, 27 DSP	46 cycles @ 276 MHz
(11, 52) custom	1532 slices, 18 DSP	10 cycles @ 237 MHz	1845 slices, 18 DSP	16 cycles @ 362 MHz

Table 1. Some post-synthesis results for the `Collision` operator (on Virtex-4 xc4vlx15-12-sf363 using ISE 10.1 or 9.1, whichever gave the best results). The FloPoCo command-line used to obtain these results is `flopoco -frequency=` f `Collision` w_E w_F o where f is 200 for the slow version and 400 for the fast version, and o is 0 for the FP version and 1 for the optimized version.

stream, an attribute of `Operator`. Thus, one designs the architecture by having the constructor outputting arbitrary VHDL code to *vhdl*. In addition, signals may be wrapped in methods from `Operator`. A first method is `declare()`, that handles signal declaration. Consider the following code that computes the difference of two exponents of size w_E, in order to determine in `XltY` which is smaller.

```
vhdl << declare("DEXY", wE+1)
    << " <=   ('0' & EX) - ('0' & EY);" << endl;
vhdl << declare("XltY")
    << " <=   DEXY("<< wE<<");" << endl;
```

Here the `declare()` method adds the signal to the *signalList* of the operator (so that `outputVHDL()` will declare it in the VHDL), then simply returns its first argument. Thus, called with $w_E = 8$, the code above simply puts in the *vhdl* stream the string:

```
DEXY <=   ('0' & EX) - ('0' & EY);
XltY <=   DEXY(8);
```

So far we have mostly obfuscated the VHDL with C++ constructs. Let us see how this enables easy pipelining.

3.2. Automatic pipeline management

We now consider the end of the normalise/pack box of Figure 1, which reassembles the result from the exponent datapath and the fraction datapath into a single bit vector. The exponent datapath is in general shorter (computed in less cycles), so there is a need for synchronisation. The cycle difference depends on the pipeline depths of the shifter and add components, which in turn depends on operand sizes, target hardware, and target frequency. Building a working pipeline for a given set of parameters is conceptually simple: simply insert registers to delay the result of the shortest datapath, so as to synchronise it with the result of the longest one. FloPoCo allows to express exactly that in a generic way. Consider the following code excerpt:

```
(...)
// at some cycle
vhdl << declare("finalExp", wE+1)  << " <= " ...
(...)
// at some other cycle
vhdl << declare("finalFraction", wF+g) << " <= " ...
(...)
// enter next cycle
nextCycle();
vhdl << declare("finalSoP", wE+wF+g)  << " <= "
    << use("finalExp") << "(wE-1 downto 0)"
    << " & " << use("finalFraction") << "; ";
```

During the construction of the VHDL in the *vhdl* stream, an `Operator` maintains a *currentCycle* attribute, initially equal to zero. The main function of `nextCycle()` is to increment *currentCycle*.

Every signal declared through `addInput` or `declare` has a *cycle* attribute, which represents the cycle at which this signal is computed. It is 0 for the inputs. For signals declared through `declare()`, it is set to *currentCycle* at the time `declare()` is invoked.

Every signal also possesses an attribute *lifeSpan* which indicates how many cycles it will need to be delayed. This attribute is initialized to 0, then possibly increased by `use()` as we will see below. When the *lifeSpan* of a signal `X` is greater than zero, `outputVHDL()` will create *lifeSpan* new signals, named `X_d1`, `X_d2` and so on, and insert registers between them. In other words, `X_d2` will hold the value of `X` delayed by 2 cycles.

Wrapping a signal in `use()` has the following effect. First, `use("X")` will compare *currentCycle* and the *cycle* declared for `X`, which we note `X`.*cycle*.

– If they are equal, `use("X")` will simply return the string `"X"`.

– If *currentCycle* < `X`.*cycle*, `use("X")` will emit an error message reporting that `X` is being used before the cycle at which it was defined.

– If *currentCycle* > `X`.*cycle*, `use("X")` will delay signal `X` by $n = $ *currentCycle* − `X`.*cycle* cycles. Technically, if

$n = 5$ for instance, `use("X")` just returns `"X_d5"`. It also updates `X.`*lifeSpan* to be at least equal to 5.

`Operator` defines several other functions to manage *currentCycle*, such as `setCycle(int n)`, `setCycleFromSignal(string s)` which sets the *currentCycle* to the *cycle* of the signal whose name is given as an argument (going back if needed), and `syncCycleFromSignal(string s)` which may only advance *currentCycle*. The latter allows to synchronise several signals by setting *currentCycle* to the max of their *cycle*.

For an operator designer, pipelining an operator therefore resumes to 1/ wrapping left-hand side signals in `declare()` and right-hand side signals in `use()`, and 2/ defining the pipeline levels by means of inserting `nextCycle()` or other cycle management calls. Such a pipeline will then automatically adapt to insertions and suppressions of synchronization barriers. Typically, if one is unhappy with the performance, one may decide to break the critical path by inserting a synchronisation barrier in it. This may be as simple as inserting a single `nextCycle()` in the code. It is also possible to wrap some of the `nextCycle()` calls in conditional statements, so that the pipeline adapts to the frequency, the operator generic parameters, etc. For instance, the `IntAdder`, `Shifters`, `FPSqrt`, ... classes of FloPoCo produce operators whose pipeline depth depends on the target frequency. By using these pre-existing subcomponents, our `Collision` operator behaves the same, as Table 1 shows.

It should be noted that this scheme still allows for arbitrary synthesizable VHDL – only the signal names have to be wrapped by `use()`, `declare()`, etc. Also, this scheme gracefully degrades to a combinatorial operator – in this mode `declare()` always declares one single signal and `use()` always returns its argument unmodified.

Space limitation prevents presenting in details how subcomponents are managed in FloPoCo. In short, one has to create an object instance for the sub-component, then there is a `inPortMap()` method that is conceptually similar to `use()`, an `outPortMap()` method conceptually similar to `declare()`, and an `instance()` method that returns the VHDL of the instance, and performs the bookkeeping required for `outputVHDL()` to declare the corresponding component. We invite the interested reader to have a look at `Collision.cpp`.

3.3. Targets

The `Target` class abstracts the features of actual FPGA chips. Classes representing real FPGA chips extend this class (we currently have classes for two FPGAs, Xilinx `Virtex4` and Altera `StratixII`). The idea is to declare abstract methods in `Target`, which are implemented in its subclasses, so that the same generator code fits all the targets.

Of course, it is also possible to have a conditional statement that runs completely different code depending on the target – this will be the case for instance for the `IntMultiplier` class that builds large multipliers, because DSP capabilities are too variable from one target to the other. A `Target` is given as argument to the constructor of any operator.

The methods provided by the `Target` class can be semantically split into two categories:

Architecture-related methods provide information about the architecture of the FPGA and are used in architectural exploration. For instance, `lutInputs()` returns the number of inputs of the FPGA's LUTs.

Delay-related methods provide approximate information about computation time. For example, `adderDelay(int n)` returns the delay of an n-bit addition. These methods are used for frequency-directed pipelining (here for `IntAdder`, but also in the addition-based iterative square root for instance). Some of these methods have an architecture-related dual, for example `suggestAdderSize(double delay)` that returns the size of an adder that will have the required delay.

The difficulty here is to find the right abstraction level for `Target`. On the one hand, we do not hope to provide an exhaustive and detailed description of all the existing – and future – FPGAs. On the other hand, we do not need to: back-end tools are better and better at optimizing a design to a given target, and we should rely on them as much as possible. For instance, it is important to evaluate the number of pipeline levels and their approximate location, but the details of register placement are best left to the back-end tools, which may even apply retiming techniques [13] after logic optimization and technology mapping.

`Target` is still expected to evolve a lot. Other methods will be added when needed, for instance to model the internal memory resources. Also, area estimation methods are needed to assist the design-space exploration of future operators. Of course, the question of target abstraction will remain an open one forever, as new FPGA features keep appearing and new operators present new problems.

4. TEST-BENCH GENERATION

The underlying mathematical nature of an arithmetic operator can be usefully exploited to make test-bench generation easy. For instance, the output from a floating-point operator may be defined as a mathematical function of the inputs (addition or exponential) composed with one of the rounding functions defined in the well established IEEE-754 standard [14]. Defining expected outputs this way is not only simpler than mimicking the architecture, it also minimizes the possibility of making the same mistake in both the operator and its test bench. In FloPoCo, one has to overload the Operator virtual method `emulate()`. Thanks to the bit-accurate

MPFR library [15], this typically takes about ten lines.

Overloading `emulate()` is enough for FloPoCo to be able to create a generic test bench using random inputs. However, function analysis also allows for better, more operator-specific test-case generation. Let us just take two examples.

1/ A double-precision exponential returns $+\infty$ for all inputs larger than 710 and returns 0 for all inputs smaller than -746. In other terms, the most interesting test domain for this function is when the input exponent is between -10 and 10, a fraction of the full double-precision exponent domain (-1024 to 1023). Generating random 64-bit integers and using them as floating-point inputs would mean testing mostly the overflow/underflow logic, which is a tiny part of the operator.

2/ In a floating-point adder, if the difference between the exponents of the two operands is large, the adder will simply return the biggest of the two, and again this is the most probable situation when taking two random operands. Here it is better to generate random cases where the two operands have close exponents.

Such cases are managed by overloading the Operator method `buildRandomTestCases()`. Finally, `buildStandardTestCases()` allows to test corner cases which random testing has little chance to find.

5. CONCLUSION

This article introduced an open-source software framework for generating high-quality, highly parametrized, pipelined and flexible operators for FPGAs. This framework evolved (and still evolves) in a deliberately bottom-up and practical way focused on the needs of arithmetic cores. It is based on embedding VHDL code in C++. Thus it provides the full power of the VHDL language. In addition, it provides facilities for pipeline construction and synchronization, target-based and frequency-based optimisation, and test bench generation.

In its current state, this generator is already an invaluable booster for the core focus of our research, which is to develop new arithmetic operators. In the future, a wider range of design objectives may be added: optimizing for power, for a given pipeline depth, for memory, etc. In addition, we will refine and extend the FPGA model as new needs appear.

We have also shown how the automatic pipeline framework can be used to build larger computation pipelines composed of many operators, which automatically adapt to changes in precision or target frequency. What is now needed is a more user-friendly interface. This could be a graphical interface, but a more ambitious objective will be to integrate FloPoCo in one of the emerging C-to-hardware languages.

6. REFERENCES

[1] D. Strenski, J. Simkins, R. Walke, and R. Wittig, "Reevaluating FPGAs for 64-bit floating-point calculations," *HPC wire*, May 2008.

[2] J. Detrey, F. de Dinechin, and X. Pujol, "Return of the hardware floating-point elementary function," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 161–168.

[3] N. Woods and T. VanCourt, "FPGA acceleration of quasi-Monte Carlo in finance," in *Field Programmable Logic and Applications*, 2008, pp. 335–340.

[4] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Field Programmable Logic and Applications*. IEEE, Aug. 2009.

[5] N. Brisebarre, F. de Dinechin, and J.-M. Muller, "Integer and floating-point constant multipliers for FPGAs," in *Application-specific Systems, Architectures and Processors*. IEEE, 2008, pp. 239–244.

[6] M. Langhammer, "Floating point datapath synthesis for FPGAs," in *Field Programmable Logic and Applications*, 2008, pp. 355–360.

[7] R. Andraka, "Hybrid floating point technique yields 1.2 gigasample per second 32 to 2048 point floating point FFT in a single FPGA," in *High Performance Embedded Computing Workshop*, 2006.

[8] F. de Dinechin, J. Detrey, I. Trestian, O. Creţ, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors," ÉNS Lyon, Tech. Rep. ensl-00174627, 2007, http://prunel.ccsd.cnrs.fr/ensl-00174627.

[9] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2005, pp. 328–333.

[10] F. de Dinechin, B. Pasca, O. Creţ, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *Field-Programmable Technologies*. IEEE, 2008, pp. 33–40.

[11] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek, "Object oriented circuit-generators in Java," in *FPGAs for Custom Computing Machines*. IEEE, 1998.

[12] D. Bakalis, K. Adaos, D. Lymperopoulos, M. Bellos, H. Vergos, G. Alexiou, and D. Nikolos, "A core generator for arithmetic cores and testing structures with a network interface," *Journal of Systems Architecture*, vol. 52, no. 1, pp. 1 – 12, 2006.

[13] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5 – 35, 1991.

[14] ANSI/IEEE, *Standard 754-1985 for Binary Floating-Point Arithmetic (also IEC 60559)*, 1985.

[15] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, pp. 13:1–13:15, June 2007.