Generating high-performance arithmetic operators for FPGAs

Florent De Dinechin, Cristian Klein, Bogdan Pasca

Generating high-performance arithmetic operators for FPGAs

LIP research report 2008-28

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

Abstract

This article addresses the development of complex, heavily parameterized and flexible operators to be used in FPGA-based floating-point accelerators. Languages such as VHDL or Verilog are not ideally suited for this task. The main problem is the automation of problems such as parameter-directed or target-directed architectural optimization, pipeline optimization, and generation of relevant test benches. This article introduces FloPoCo, an open object-oriented software framework designed to address these issues. Written in C++, it inputs operator specifications, a target FPGA and and an objective frequency, and outputs synthesisable VHDL fine-tuned for this FPGA at this frequency. Its design choices are discussed and validated on various operators.

1 Arithmetic operator design

1.1 Floating-point and FPGAs

FPGA-based coprocessors are available from a variety of vendors, and it is natural to try and use them for accelerating floating-point (FP) applications. On floating-point matrix multiplication, their floating-point performance slightly surpasses that of a contemporary processor [6], using tens of operators on the FPGA to compensate their much slower frequency (almost one order of magnitude). However, FPGAs are no match to GPUs here. For other FP operations that are performed in software in a processor (for instance all the elementary functions such as exp, log, trigonometric...) there is much more speed-up potential: One may design a dedicated pipelined architecture on an FPGA that outperforms the corresponding processor code by one order of magnitude while consuming a fraction of the FPGA resources [4]. Implementing the same architecture in a processsor would be wasted silicon, since even the logarithm is a relatively rare function typical processor workloads. For the same reason, GPUs have hardware acceleration for a limited set of functions and in single precision only. In an FPGA, you pay the price of this architecture only if your application needs it. Besides, operators can also be *specialized* in FPGAs. For example, a squarer theoretically requires half the logic of a multiplier; A floating-point multiplication by the constant 2.0 boils down to adding one to the exponent (a 12-bit addition in double-precision), and shouldn't use a full-blown FP multiplier as it does in a processor. Actually it is possible to build an optimized architecture for any multiplication by a constant [2]. Finally, operators can be *fused* on an FPGA, for example the Euclidean norm $\sqrt{x^2+y^2}$ can be implemented more efficiently than by linking

*This work was partly supported by the XtremeData university programme, the ANR EVAFlo project and the Egide Brâncuşi programme 14914RL.

two squarers, one adder and one square root operator.

There are many more opportunities for floating-point on FPGAs [3]. The object of the FloPoCo project[1] is to study and develop such FPGA-specific **Flo**ating-**Po**int **Co**res.

1.2 From libraries to generators

FloPoCo is not a library but a *generator* of operators. Indeed, it is the successor to FPLibrary, a library of operators written in VHDL. Many parts of FPLibrary were actually generated by as many ad-hoc programs, and FloPoco started as an attempt to bring all these programs in a unified framework.

A first reason is that it is not possible, for instance, to write by hand, directly in VHDL or Verilog, an optimized multiplier by a constant for each of an infinite number of constants. However, this task is easy to automate in a program that inputs the constant.

Another reason is the need for 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 are parameterized by the bit-widths of their inputs and outputs. Flexibility also makes it possible to optimize for different hardware targets, with different LUT structure, memory and DSP features, etc. Thus, the more flexible an operator, the more future-proof.

Finally, for complex operators such as elementary function evaluators, the optimal design is the result of a costly design-space exploration, which is best performed by a computer.

VHDL and Verilog are good for describing a library of operators optimized for a given context, but the more flexibility and the more design-space exploration one wants, the more difficult it gets. It is natural to write operator generators instead. A generator inputs user specifications, performs any relevant architectural exploration and construction (sometimes down to pre-placement), and outputs the architecture in a synthesizable format. To our knowledge, this approach was pioneered by Xilinx with their core generator tool[2].

An architecture generator needs a back-end to actually implement the resulting circuit. The most elegant solution is to write an operator 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. The inconvenient is that most of these languages are still relatively confidential and restricted in the FPGAs they support. Basing FloPoCo on a vendor generator would be an option, but would mean restricting it to one FPGA family.

FloPoCo therefore took a less elegant, but more universal route. The generator is written in a mainstream programming language (we chose C++), and it outputs operators in a mainstream HDL (we chose standard synthesisable VHDL). Thus, the FloPoCo generator is portable, 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 2.2 will show how they can nevertheless be optimized to a given FPGA target.

The inconvenient of this approach is that we had to develop a framework, instead of reusing one. Section 2 describes this framework and the way it evolved in a very practical and bottom-up way.

1.3 The arithmetic context

It is important to understand that this framework was developed only with arithmetic operators in view. An arithmetic operator is the implementation of a mathematical function, and this underlying mathematical nature is exploited pervasively in FloPoCo. For instance, an operator may be combinational or pipelined, but will usually involve no feedback loop or state machine (the only current exception is an accumulator). With this restriction, we are able to implement a simple, efficient and automatic approach to pipelining (see section 3) and testbench generation (see section 4). As another example, when generating

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

[2] We would welcome any feedback on early architecture generators

test benches, relevant test patterns may be defined by function analysis, and the expected output is defined as a mathematical function of the input, composed with a well-defined rounding function [1]. These are only a few examples. The design-space exploration for complex operators is based on automated error analysis [4], which is also specific to the arithmetic context.

FloPoCo is not only a generator framework, it is also 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 exp and log. This article is not about these operators, but will be illustrated by actual examples of already implemented operators. FloPoCo is distributed under the LGPL, and interested readers are welcome to try it, use it and improve it.

2 The FloPoCo framework

The FloPoCo generator inputs (currently in the command-line) a list of operator specifications, internally builds a list of `Operator` objects (some of which may be sub-components of the specified operators), then outputs the corresponding VHDL.

2.1 Operators

The core class of FloPoCo 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 of FloPoCo extend this class.

The main method of of `Operator` is `outputVHDL()`, which outputs 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, with many parameters, `GENERATE` constructs in VHDL are best replaced with loops and tests in the C++ code, which makes the VHDL code simpler and easier to debug. In addition, the `Operator` class provides many helper methods which relieve the designer from repetitive or error-prone work, for example entity and component declaration, signal[3] declaration, signal registering, etc .

In short, `Operator` provides black box functionality for known VHDL recipes, but otherwise requires manual output of VHDL code. This approach allowed us to quickly backport existing generators. More importantly, we may tinker with the framework without having to rework existing cores.

Design space exploration, if any, is done in the operator constructor. The input specification (input and output widths, etc.) and the deployment and performance constraints (e.g. VirtexIV, 300MHz) are analysed, and operator attributes are set to be used later by `ouputVHDL()`. For instance, the constructor of an integer constant multiplier internally builds and pipelines a directed acyclic graph (see Figure 1) with several labels on each node [2].

`Operator` also defines other virtual methods for the purpose of pipelining and testbench generation. These will be considered in due course.

2.2 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 very different FPGAs, Xilinx `VirtexIV` 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[4]. To this purpose, a `Target` is given as argument to the constructor of an operator – it also receives an objective frequency, this will be detailed in section 3.

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.

[3] There is a `Signal` class, but it has currently no real signal semantic (it doesn't checks for pending signals, short circuits, etc), it just makes writing `outputVHDL()` easier, and we will not detail it any further.

[4] Of course, it is also possible to have a big `if` that runs completely different code depending on the target.

- **Delay-related methods** provide approximative informations about the delays for signals traveling the FPGA. For example, `adderDelay(int n)` returns the delay of an n-bit addition. These methods will be used for automatic pipelining, see section 3. 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 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: Vendor tools are very good at fitting a design to a given target, and we should rely on them. The complexity of exploiting the details of the target should be left to the back-end tools.

To understand how we approach this issue in FloPoCo, consider the example of integer multiplication support in FPGA. Early FPGAs were logic-only, then came hard multipliers embedded in the FPGA fabric, then these multipliers were promoted to DSP blocks with the addition of accumulators. Current DSP blocks are very complex and differ widely from FPGA to FPGA. Some contain 18x18 multipliers, some 18x24, some 36x36 which can be split in several 18x18 or 9x9 multipliers, with subtle restrictions. All contain internal registers levels, and some an accumulator. What is the best way to abstract this complexity and variety, in a way both concise and generally useful?

The current – probably not definitive – answer is a method called `suggestSubmultSize()` which takes the objective frequency as an argument and returns the input sizes of the largest (possibly rectangular) sub-multiplier that, when written as a `*` in VHDL, runs at the objective frequency on this target. This abstract method will be implemented very differently in subclasses of `Target` that describe actual FPGAs, depending on the availability or not of DSP blocks and on the capabilities of these blocks. With this simple interface, we are able to generate large floating-point multipliers (build by assembling several DSP blocks) whose performance and resource consumption almost match (and sometimes even surpass) the vendor-generated ones, while being more flexible.

For the design-space exploration of future operators, we will need other methods, for example a method that inputs a multiplier size and returns an estimation of its area. And 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.

3 Automatic pipelining

Pipelining a given arithmetic operator for a given context is relatively easy, if tedious. What is important is to evaluate the number of pipeline levels and their approximate location, but the details are best left to the back-end tools, which will actually place the registers after logic optimization as part of the technology mapping step. Recent tools apply, to various extent, retiming techniques (moving registers around to improve the critical path without changing the functionnality of the circuit) [5]. This is also best done after technology mapping.

3.1 Frequency-directed pipelining

Currently, FloPoCo implements frequency-directed pipelining using variations on the following generic algorithm, which is simple (greedy), and linear in the size of the operator. The constructor, working from the output to the input (or the other way round, depending what is expected to give the best results), accumulates estimations of the critical path delay (provided by the `Target` class), and inserts register levels when the current critical path exceeds the reciprocal of the objective frequency. When this process is finished, the constructor sets the `pipelineDepth` attribute of the `Operator` class. When an operator instantiates a sub-component (e.g. `FPMultiplier` instantiates `IntMultiplier`), it may obtain its `pipelineDepth`, which allows it to delay signals accordingly – this works recursively.

FloPoCo does not provide any special framework for the computation of critical paths or the distribution of registers: an ad-hoc program needs to be

written by hand for each operator to implement the previous generic algorithm. This is about as tedious as pipelining VHDL code, but needs to be done only once. In addition, FloPoCo has several facilities for inserting multiple delays on signals which help prevent bugs due to lack of synchronization. The testbench generation framework also takes into account the pipeline depth of the unit under test automatically.

3.2 Discussion

This part of the framework is still being improved. It is not clear yet if we want it to eventually end up as a generic retiming engine using a generic abstraction of an arithmetic circuit. Firstly, it would be a tremendous amount of work to get it right, and we choose to focus our efforts on operator developement. Secondly, it is not obvious that it would be useful. The ad-hoc, per-operator approach is more flexible and more powerful in the design exploration phase, for instance.

Critical path delay estimations are necessarily inaccurate: actual values can only be obtained after placement and routing of the complete application in which the operator takes place, something which is out of the scope of FloPoCo. No guarantee is therefore given that the operator will actually function at the objective frequency. What is easy to ensure, however, is that when the objective frequency is raised, the number of pipeline stages increases and the critical path delay does not increase.

The real question is, do we need to actually place registers, which is the bulk of the work? If all the back-ends implemented retiming [5] efficiently (which is not the case yet), we would just have to evaluate the pipeline depth as a function of the frequency (a comparatively simple task) and place the corresponding number of register levels at the output of the operator, leaving to the tools the task to push them in. Still, retiming is a costly iterative process, and considering its local nature, it should be sped up by a good initial placement. We have little practical experience on this subject and would appreciate feedback.

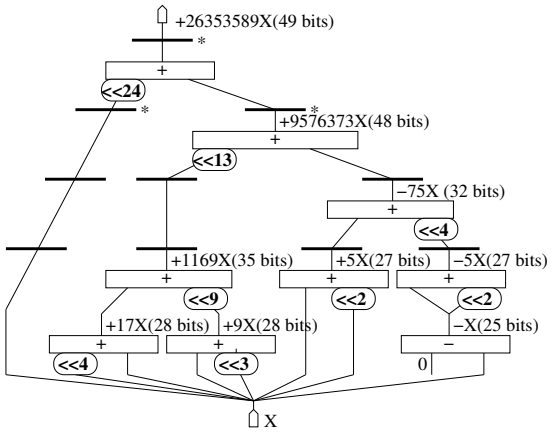

Figure 1: Multiplier by 26353589 pipelined for 200MHz.

`-frequency=`	latency	frequency	area
100	2	146 MHz	176 sl
200	4	264 MHz	199 sl
300	5	373 MHz	218 sl

Table 1: Synthesis results for pipelined operators

3.3 A detailed example

The command

```
flopoco -frequency=200 -target=VirtexIV
      IntConstMult 24 26353589
```

builds a multiplier of a 24-bit integer with the constant 26353589 (the 24 first bits of π), with objective frequency 200MHz, for a VirtexIV. Figure 1 shows the obtained architecture. Note that some adders are much larger than the others. With `-frequency=100`, only the two last levels of registers (marked with a `*`) are generated: The lower levels are grouped in a single pipeline stage. Table 1 provides some synthesis results for Xilinx VirtexIV using ISE 9.1, for three values of the objective frequency.

In this example, the reported frequency is much better than the specified one. This is because the automatic pipeline program for this operator works at the adder level, not at the bit level (splitting

the adders into sub-adders). The current framework would allow for bit-level pipelining for these operators, but it would be much more complex and we currently feel this effort is not justified. We however pipeline adders when they are too large to reach the objective frequency, because we can use for that the existing pipelined `IntAdder` operator.

4 Test case generation

In order to make sure that no bugs leak into the VHDL code generated for each arithmetic operator, FloPoCo was also designed to automate test case generation. Due to the large number of parameters that can be customised, writing test benches by hand for each operator is not an option.

Test benches are pieces of VHDL code (possibly with a few associated data files) which run from a VHDL simulator, give certain inputs (test cases) to the arithmetic operators and test the correctness of the outputs.

Test cases are operator-specific and are generated by doing the equivalent computation in software. We deliberately avoid duplicating the hardware algorithms in software in order to avoid introducing the same bugs. Instead, we relied on well tested libraries such as GMP and MPFR.

Small operators can be exhaustively tested, but it becomes impossible to exhaustively test larger ones, in particular double-precision ones. Instead, our strategy consists in maximising the number of data paths and signal combinations that are tested.

A FloPoCo test bench basically generates two types of test cases. The first is random test cases. The `Operator` class is able to generate uniform random inputs, but it should most of the cases be overriden in an operator-dependent way that focuses the test on the interesting domains of input. For instance, a double-precision exponential returns $+\infty$ for all inputs larger than 1024 and returns 0 for all inputs smaller than -1024. In other terms, the most interesting test domain for this function is when the input exponent is between -10 and 10, a tiny 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. Similarly, 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.

In addition to random test cases, there are also mandatory test cases which test specific situations which a random test would have little chance to hit. Again, these test cases are operator-specific.

5 Conclusion

This article introduced FloPoCo, an open-source software framework for generating high-quality, highly parameterized, pipelined and flexible operators for FPGAs. This framework evolved (and still evolves) in a deliberately bottom-up and practical way focussed on the needs of arithmetic cores. It uses a low-level, highly flexible `printf`-based approach to VHDL generation.

In its current state, FloPoCo 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, for instance to model the internal memory resources.

FloPoCo's automatic pipeline framework can in principle be used to build larger computation pipelines composed of many operators, in a way that automatically delays signals to match the pipelines of the various operators, and does so even when the designer, to optimize resource usage, changes the precision of some operators. We also intend to explore this possibility.

References

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

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

[3] F. de Dinechin, J. Detrey, I. Trestian, O. Creţ, and R. Tudoran. When FPGAs are better at floating-point than microprocessors. Technical Report ensl-00174627, École Normale Supérieure de Lyon, 2007. http://prunel.ccsd.cnrs.fr/ensl-00174627.

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

[5] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5 – 35, 1991.

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