The arithmetic operators you will never see in a microprocessor

Florent de Dinechin
LIP (ENSL-CNRS-Inria-UCBL), École Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France
Email: florent.de.dinechin@ens-lyon.fr

***Abstract*—It has been shown that FPGAs could outperform high-end microprocessors even on floating-point computations, thanks to massive parallelism. Too often, however, such studies re-implement in the FPGA the operators present in a processor. An FPGA can do much better: it can accomodate hardware operators that would make no economical sense in a general-purpose processor, and it can taylor them just right to the needs of the application. This talk tries to survey this idea systematically, discussing its potential, exhibiting some exotic (but useful) operators developed in the FloPoCo project, and listing some of the challenges ahead.**

I. Microprocessor versus FPGA

When designing an arithmetic unit for a processor, a major concern is to cast to hardware only those operators that are the most generally useful. Should a processor's instruction set include the elementary functions [1]? Should a processor include a hardware divider, considering the rarity of division in general code [2]? Such questions have oriented hardware arithmetic research towards one-size-fits-all operators, culminating with the *fused multiply and add*, a single operator that replaces a whole floating-point unit in the most recent instruction sets.

When porting an application to a field-programmable gate array (FPGA), one has, among other things, to design its arithmetic operators. In this case, the previous concern disappears: As soon as one application requires an operator, it makes sense to investigate a specific architecture for it. Squaring or multiplying by $\log(2)$ are examples of operations that will never be included in the instruction set of a general purpose processor, but appear in enough applications to justify developing the corresponding FPGA operators.

We have therefore many more operators to study when targetting FPGAs than when targetting processors. In addition, we have more freedom in the data formats. A processor offers only the choice between single (32-bit) and double (64-bit) precision floating-point, but it is very unlikely that the minimal precision required by a given application is one of these standard formats. As a consequence, most of the time, a processor carries around, and computes on, data with many irrelevant bits. Determining this optimal precision is a challenge *per se* [4], [5], but then we may build operators just right for this precision. Furthermore, we may optimize out the hardware that turns out to be useless in the application context [3].

In this sense, an FPGA implementation has the potential to be much more efficient than its processor counterpart, if we can build it to compute just right.

Computing just right will help conserve power, but it is also a major (and currently vastly underused) lever to improve performance. It is well known that the FPGA implementation of a given operator, say a floating-point double-precision multiplier, is typically one order of magnitude slower than its highly optimized processor counterpart. This is the uncompressible cost of reconfigurability. FPGAs have been shown to catchup thanks to massive parallelism [6], but computing just right is another, complimentary way to catch up. Firstly, it may help reduce both area and latency, hence maximize the operator-level parallelism. Secondly, for more complex operations for which the processor has no hardware support, a just-right architecture may inverse the performance ratio, proving one order of magnitude faster than the best processor software, as has been consistently demonstrated for elementary functions [7].

The goal of the FloPoCo project[1] is to systematically explore this almost virgin land of non-standard arithmetic operators for FPGAs, with an obsession to compute just right. To address the practical challenges of designing such operators, FloPoCo is also an open C++ framework for the easy construction of heavily parameterized operators with flexible pipeline[8]. However, we now focus on some of the non-standard operators already designed in this framework.

II. Some successful non-standard operators

FloPoCo includes a range of specialized multipliers: exact or truncated multipliers or squarers [9], multipliers by a constant using several techniques [10], [7]. The latter are actually *meta-operators*, programs that compute an architecure out of an arbitrary constant, one of the reasons why FloPoCo cannot be a component library, but has to be an architecture generator.

Our most complex meta-operator to date are generators of polynomial approximations to an arbitrary numeric function specified as an expression like $\exp(x * x - 1)$ [11], [12].

Several approaches to floating-point accumulation on FPGAs have been proposed [13], [14], [15]. Instead of these generic approaches, FloPoCo provides application-specific

[1] http://flopoco.gforge.inria.fr/

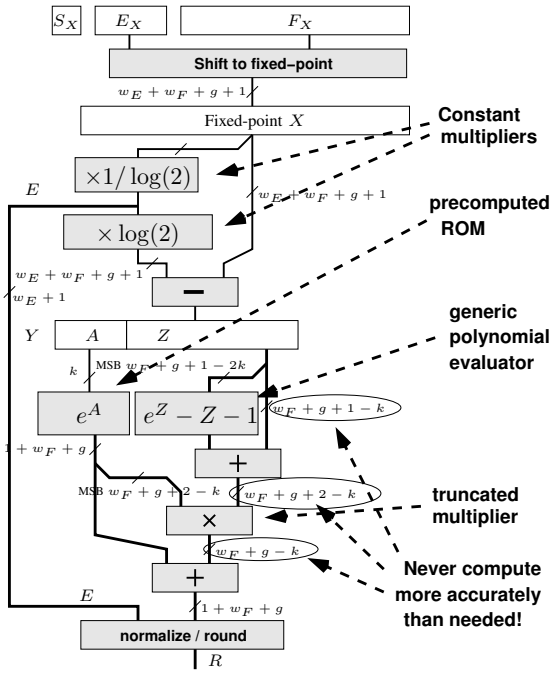

Figure 1. An architecture for floating-point exponential. w_E and w_F are the sizes in bits of the exponent and significand

accumulator, sum-of-products and sum-of-squares [5], [8] that compute just right, provided a designer can express some numerical properties on the data manipulated by her application.

A good illustration of the FloPoCo project is Figure 1 (explained in [7]) that depicts an highly arithmetic-efficient implementation of the floating-point exponential, building on many other non-standard operators.

III. Open challenges and future work

Whether FPGAs will establish themselves as a credible, mainstream computing platform is still unclear. The main challenge is probably to ease their programming. FloPoCo doesn't help here, as it gives more freedom of choice to a programmer who already has to face tasks much more complex than when programming a PC. To make just-right computing mainstream, we will have to design tools that assist us in precision analysis and optimization. Meanwhile, providing efficient coarse operators, such as elementary functions, is a good way to disseminate just right computing while hiding its complexity.

With this goal, the list of operators still to be investigated is endless, even going well beyond those we study in FloPoCo, to fields such as cryptography or random number generation [16].

References

[1] G. Paul and M. W. Wilson, "Should the elementary functions be incorporated into computer instruction sets?" *ACM Transactions on Mathematical Software*, vol. 2, no. 2, pp. 132–142, 1976.

[2] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Transactions on Computers*, vol. 46, no. 2, pp. 154–161, 1997.

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

[4] A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang, "Automating customisation of floating-point designs," in *Field Programmable Logic and Applications*, ser. LNCS, vol. 2438. Springer, 2002, pp. 523–533.

[5] 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.

[6] D. Strenski, "FPGA floating point performance – a pencil and paper evaluation," *HPCWire*, Jan. 2007.

[7] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field Programmable Technologies*, Dec. 2010, pp. 110–117.

[8] ——, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, 2011.

[9] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for floating-point double precision and beyond on FPGAs," in *Highly-Efficient Accelerators and Reconfigurable Technologies*, 2010.

[10] 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.

[11] 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.

[12] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

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

[14] Z. Luo and M. Martonosi, "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques," *IEEE Transactions on Computers*, vol. 49, no. 3, pp. 208–218, 2000.

[15] A. DeHon and N. Kapre, "Optimistic parallelization of floating-point accumulation," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 205–213.

[16] D. B. Thomas and W. Luk, "FPGA-optimised uniform random number generators using LUTs and shift registers," in *Field Programmable Logic and Applications*. IEEE, 2010, pp. 77–82.