A flexible floating-point logarithm for reconfigurable computers
LIP research report RR2010-22

Florent de Dinechin,

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

Abstract

The advent of reconfigurable co-processors based on field-programmable gate arrays has renewed interest in hardware architectures for elementary functions. This article studies operators for the logarithm function in the context of this target technology. An old algorithm is generalized, fine-tuned and implemented as an architecture generator, exposing a wide range of trade-offs between resources (memory, logic and multipliers) and performance (frequency and pipeline depth). A single pipelined operator computes five times more double-precision floating-point logarithms per second than a high-end processor core, while consuming only a few percents of the resources of a high-end FPGA. This generator is available under the LGPL as part of the FloPoCo project.

Keywords *Floating-point elementary functions, hardware operator, FPGA, logarithm.*

I. Introduction

Virtually all the computing systems that support some form of floating-point (FP) also include a floating-point mathematical library (libm) providing elementary functions such as exponential, logarithm, trigonometric and hyperbolic functions, etc. Modern systems usually comply with the IEEE-754 standard for floating-point arithmetic [1] and offer hardware for basic arithmetic operations in single- and double-precision formats (32 bits and 64 bits respectively). Most libms implement a superset of the functions mandated by language standards such as C99 [2].

A. Hardware versus software for the floating-point elementary functions

The question wether elementary functions should be implemented in hardware was controversial in the beginning of the PC era [3]. The literature indeed offers many articles describing hardware implementations of FP elementary functions [4], [5], [6], [7], [8], [9]. In the early 80s, Intel chose to include elementary functions to their first math co-processor, the 8087.

However, for cost reasons, in this co-processor, as well as in its successors by Intel, Cyrix or AMD, these functions did not use the hardware algorithm mentioned above, but were microcoded, thus slow. Indeed, software libms were soon written which were more accurate and faster than the hardware version. For instance, as memory went larger and cheaper, one could speed-up the computation using large tables (several kilobytes) of precomputed values [10], [11]. It would not be economical to cast such tables to silicon in a processor: The average computation will benefit much more from the corresponding silicon if it is dedicated to more cache, or more floating-point units for example. Besides, the hardware functions lacked the flexibility of the software ones, which could be optimized in context by advanced compilers.

These observations contributed to the move from CISC to RISC (Complex to Reduced Instruction Sets Computers) in the 90s. Intel themselves now also develop software libms for their processors that include a hardware libm [12]. Research on hardware elementary functions has since then mostly focused on approximation methods for fixed-point evaluation of functions [13], [14], [15], [16].

B. Floating-point and reconfigurable computing

Lately, a new kind of programmable circuit has been gaining momentum: The FPGA, for Field-Programmable

Gate Array. Designed to emulate arbitrary logic circuits, an FPGA consists of a very large number of configurable elementary blocks, linked by a configurable network of wires. A circuit emulated on an FPGA is typically one order of magnitude slower than the same circuit implemented directly in silicon. For instance, a floating-point adder or multiplier never works at more than 400MHz in this technology. However, FPGAs are reconfigurable and therefore offer much greater flexibility than classical ASICs, including microprocessors. In particular, an operator will consume silicon only if it is useful to the computation under consideration. With this new technological target, the subject of hardware implementation of elementary functions becomes a hot topic again.

FPGAs have been used as co-processors to accelerate specific tasks, typically those for which the hardware available in processors is poorly suited. This, of course, does not seem the case of floating-point computing. Indeed, microprocessors are built with highly optimized floating-point units. However, FPGA capacity has increased steadily with the progress of VLSI integration, and it is now possible to pack many FP operators on one chip: Massive parallelism allows one to recover the performance overhead [17], [18], and accelerated FP computing has been reported in single precision [19], then in double-precision [20], [21]. Mainstream computer vendors such as Silicon Graphics and Cray now build computers with FPGA accelerators. A challenge is to use them as *floating-point* accelerators.

The FloPoCo project[1] helps addressing this challenge by providing high-quality floating-point operators. FloPoCo is an open-source operator generator written in C++. It provides the basic operations of an FPU, but actually focuses on operators not available on processors, for which there is greater acceleration potential [22]. The logarithm is an example of such an operator.

The present article is supported by the `FPLog` operator of FloPoCo, implemented as the `FPLog.cpp` class in the FloPoCo distribution version 1.15.1.

C. Related works, contributions and outline

Previous work has shown that a single instance of an exponential or logarithm operator can provide ten times the performance of the processor, while consuming a small fraction of the resources of current FPGAs [23]. The reason is that such an operator may perform most of the computation in optimized fixed point with specifically crafted datapaths, and is highly pipelined. However, the architecture of [23] uses a generic table-based approach [16] which doesn't scale well beyond single precision: Its size grows exponentially.

In this article, we demonstrate a more algorithmic approach which works well beyond double precision. It is a synthesis of much older works, including the Cordic/BKM family of algorithms [24], the radix-16 multiplicative normalization of [4], Chen's algorithm [5], an *ad-hoc* algorithm by Wong and Goto [8], and probably many others [24]. All these approaches boil down to the same basic properties of the logarithm function, and are synthesized in Section II. The specificity of the FPGA hardware target are summarized in Section III, and the algorithm and its implementation are detailed in Section IV. Section VI provides performance results from actual synthesis, and discusses them. Section VII compares these results with estimations for a finely tuned polynomial approximation method.

This article builds upon an article published in the Arith 17 conference [25]. Focusing only on the logarithm function, it improves [25] in several respects. All the proofs that were omitted in [25] for lack of space are given. This algorithm is generalized to make use of features that have become commonplace in high-performance FPGAs: embedded multipliers and memory blocks. A trade-off is exposed and discussed in this context, supported by experimental results. The choice of the algorithm itself is justified by comparing it with a more classical polynomial approximation approach. Some of the sub-components, such as the constant multiplications, have been optimized. Last but not least, the operators discussed here are pipelined.

II. Iterative reciprocal, logarithm, and exponential

Wether we want to compute the logarithm or the exponential, the idea common to most previous methods may be summarized by the following iteration. Let (x_i) and (l_i) be two given sequences of reals such that $\forall i, x_i = e^{l_i}$. It is possible to define two new sequences (x'_i) and (l'_i) as follows: l'_0 and x'_0 are such that $x'_0 = e^{l'_0}$, and

$$\forall i > 0 \left\{ \begin{array}{rcl} l'_{i+1} & = & l_i + l'_i \\ x'_{i+1} & = & x_i \times x'_i \end{array} \right. \quad (1)$$

This iteration maintains the invariant $x'_i = e^{l'_i}$, since $x'_0 = e^{l'_0}$ and $x_{i+1} = x_i x'_i = e^{l_i} e^{l'_i} = e^{l_i + l'_i} = e^{l'_{i+1}}$.

Therefore, if x is given and one wants to compute $l = \log(x)$, one may define $x'_0 = x$, then read from a table a sequence (l_i, x_i) such that the corresponding sequence (l'_i, x'_i) converges to $(0, 1)$. The iteration on x'_i is computed for increasing i, until for some n we have x'_n sufficiently close to 1 so that one may compute its logarithm using the Taylor series $l'_i \approx x'_n - 1 - (x'_n - 1)^2/2$, or even $l'_i \approx x'_n - 1$. This allows one to compute $\log(x) = l = l'_0$ by the recurrence (1) on l'_i for i decreasing from n to 0.

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

Now if l is given and one wants to compute its exponential, one will start with $(l'_0, x'_0) = (0, 1)$. The tabulated sequence (l_i, x_i) is now chosen such that the corresponding sequence (l'_i, x'_i) converges to $(l, x = e^l)$.

There are also variants where x'_i converges from x to 1, meaning that (1) computes the reciprocal of x as the product of the x_i. Several of the aforementioned papers explicitely propose to use the same hardware to compute the reciprocal [4], [8], [24]. This makes sense in the context of a processor, but in the context of reconfigurable computing, it seems more pertinent to implement an independent, high-quality divider when needed, and only then.

The various methods presented in the literature vary in the way they unroll this iteration, in what they store in tables, and in how they chose the value of x_i to minimize the cost of multiplications. Comparatively, the additions in the l'_i iteration are less expensive.

Let us now study the optimization of such an iteration for an FPGA platform. We need addition, multiplication, and tables of precomputed values.

III. A primer on arithmetic for FPGAs

We assume the reader has basic notions about the hardware complexity of arithmetic blocks such as adders, multipliers, and tables in VLSI technology (otherwise see textbooks like [26]), and we highlight here the main differences when implementing a hardware algorithm on an FPGA.

- An FPGA consists of tens of thousand of elementary blocks, laid out as a rectangular grid. This grid also includes routing channels which may be configured to connect blocks together almost arbitrarily.
- The basic universal logic element in most current FPGAs is the m-input Look-Up Table (LUT), a small 2^m-bit memory whose content may be set at configuration time. Thus, any m-input boolean function can be implemented by filling a LUT with the appropriate value. More complex functions can be built by wiring LUTs together. FPGAs have long used $m = 4$, but some recent circuits use $m = 6$.
  For our purpose, as we will use tables of precomputed values, it means that m-input, n-output tables make the optimal use of the basic structure of the FPGA. A table with $m + 1$ inputs is twice as large as a table with m inputs, and a table with $m - 1$ inputs is not smaller.
- Recent FPGAs also include flexible embedded memory block with a capacity of a few tens of Kbits. For instance, the Virtex-4 memory blocks are configurable from 16K addresses of 1 bit, to 512 addresses of 36 bits. For tables of precomputed values, the choice of using this resources or not may be dictated by the requirements of the rest of the application.
- As addition is an ubiquitous operation, the elementary blocks also contain additional circuitry dedicated to addition. As a consequence, there is no need for fast adders or carry-save representation of intermediate results: The plain carry-propagate adder is smaller, and faster for all but very large additions.
- Recent computing-oriented FPGAs include a large number of small multipliers or multiply-accumulators, typically for 18 bits times 18 bits.

IV. Overview of the logarithm operator

The logarithm is only defined for positive floating-point numbers, and does not overflow nor underflow. Exceptional cases are therefore trivial to handle and will not be mentioned further. A positive input X is written in floating-point format $X = 2^{E_X - E_0} \times 1.F_X$, where E_X is the exponent stored on w_E bits, F_X is the significand stored on w_F bits, and E_0 is the exponent bias (as per the IEEE-754 standard).

Now we obviously have $\log(X) = \log(1.F_X) + (E_X - E_0) \cdot \log 2$. However, if we use this formula, for a small ϵ the logarithm of $1 - \epsilon$ will be computed as $\log(2 - 2\epsilon) - \log(2)$, entailing a catastrophic cancellation. To avoid this case, the following error-free transformation is applied to the input:

$$\begin{cases} Y_0 = 1.F_X,\ E = E_X - E_0 & \text{when } 1.F_X \in [1, 1.5), \\ Y_0 = \frac{1.F_X}{2},\ E = E_X - E_0 + 1 & \text{when } 1.F_X \in [1.5, 2). \end{cases} \tag{2}$$

And the logarithm is evaluated as follows:

$$\log(X) = \log(Y_0) + E \cdot \log 2 \quad \text{with} \quad Y_0 \in [0.75, 1.5). \tag{3}$$

Then $\log(Y_0)$ will be in the interval $(-0.288, 0.406)$. This interval is not very well centered around 0, and other authors use in (2) a case boundary closer to $\sqrt{2}$, as a well-centered interval allows for a better approximation by a polynomial. We prefer that the comparison resumes to testing the first bit of F, called `FirstBit` in the following (see Figure 1).

Now consider equation (3), and let us discuss the normalization of the result: We need to know which will be the exponent of $\log(X)$. There are two mutually exclusive cases.

- Either $E \neq 0$, and there will be no catastrophic cancellation in (3). We may compute $E \log 2$ as a fixed-point value of size $w_F + w_E + g$, where g is a number of guard bit to be determined. This fixed-point sum will be added to a fixed-point value of $\log(Y_0)$ on $w_F + 1 + g$ bits, then a combined leading-zero-counter and barrel-shifter will determine the exponent and mantissa of the result. In this case the shift will be at most of w_E bits.

- Or, $E = 0$. In this case the logarithm of Y_0 may vanish, which means that a shift to the left will be needed to normalize the result[2].
  - If Y_0 is close enough to 1, specifically if $Y_0 = 1 + Z_0$ with $|Z_0| < 2^{-w_F/2}$, the left shift may be predicted thanks to the Taylor series $\log(1+Z) \approx Z - Z^2/2$: Its value is the number of leading zeroes (if `FirstBit=0`) or leading ones (if `FirstBit=1`) of Y_0. We actually perform the shift before computing the Taylor series, to maximize the accuracy of this computation. Two shifts are actually needed, one on Z and one on Z^2, as seen on Figure 1.
  - Or, $E = 0$ but Y_0 is not sufficiently close to 1 and we have to use a range reduction, knowing that it will cancel at most $w_F/2$ significant bits. The simpler is to use the same LZC/barrel shifter than in the first case, which now has to shift by $w_E + w_F/2$.

Figure 1 depicts the corresponding architecture. A detailed error analysis will be given in V-D.

V. Multiplicative range reduction

This section describes the work performed by the box labelled *Range Reduction* on Figure 1. Consider the centered mantissa Y_0. If `FirstBit`$= 0$, Y_0 has the form `1.0xx...xx`, and its logarithm will eventually be positive. If `FirstBit`$= 1$, Y_0 has the form `0.11xx...xx` (where the first `1` is the former implicit 1 of the floating-point format), and its logarithm will be negative.

A. First iteration

Let A_0 be the first α_0 bits of the mantissa (including `FirstBit`), $\alpha_0 > 4$. A_0 is used to index a table which gives an approximation $\widetilde{Y_0^{-1}}$ of the reciprocal of Y_0 on $\alpha_0 + 1$ bits. Noting $\widetilde{Y_0}$ the mantissa where the bits lower than those of A_0 are zeroed ($\widetilde{Y_0} = 1.0a...a$ or $\widetilde{Y_0} = 0.11a...a$, depending on `FirstBit`), the first reciprocal table stores

$$\widetilde{Y_0^{-1}} = 2^{-\alpha_0+1} \left\lceil \frac{2^{\alpha_0-1}}{\widetilde{Y_0}} \right\rceil \tag{4}$$

Theorem V.1. *If $\alpha_0 > 4$, for all $Y_0 \in [0.75, 1.5)$,*

$$Y_0\widetilde{Y_0^{-1}} = 1 + Z_1 \quad \textit{with} \quad 0 \leq Z_1 < 2.5 \cdot 2^{-\alpha_0}$$

[2]This may seem a lot of shifts to the reader. Consider that there are barrel shifters in all the floating-point adders: In a software logarithm, there are many more hidden shifts, and one pays for them even when one doesn't use them.

Proof: The truncation of Y_0 to $\widetilde{Y_0}$ means $\widetilde{Y_0} = Y_0(1-\epsilon)$ with $0 \leq \epsilon < 2^{-\alpha_0}$. Indeed, if `FirstBit` $= 1$, $\widetilde{Y_0} = 0.11a_2...a_{\alpha_0}$. The absolute truncation error is $0 \leq \delta < 2^{-\alpha_0-1}$, and as $Y_0 \geq 1/2$, the corresponding relative error is bounded by $0 \leq \epsilon < 2^{-\alpha_0}$. If `FirstBit` $= 0$, $\widetilde{Y_0} = 1.0a_2...a_{\alpha_0}$, therefore $0 \leq \delta < 2^{-\alpha_0}$, $Y_0 \geq 1$, hence $0 \leq \epsilon < 2^{-\alpha_0}$ as in the other case.

It follows that $\frac{1}{\widetilde{Y_0}} = \frac{1}{Y_0}(1 + \epsilon + \epsilon^2 + ...) = \frac{1}{Y_0}(1 + \epsilon')$ with $0 \leq \epsilon' < 2^{-\alpha_0} + 2^{-\alpha_0-4}$ since $\alpha_0 > 4$.

As $Y_0 \in [0.75, 1.5)$, it follows that $0 < \frac{1}{\widetilde{Y_0}} < 2$ and $0 < \frac{2^{\alpha_0-1}}{\widetilde{Y_0}} < 2^{\alpha_0}$. The ceil operation on this result yields a second error: $\left\lceil \frac{2^{\alpha_0-1}}{\widetilde{Y_0}} \right\rceil = \frac{2^{\alpha_0-1}}{Y_0}(1 + \epsilon')(1 + \epsilon'')$ with $0 < \epsilon'' < 2^{-\alpha_0}$.

Therefore we have $\widetilde{Y_0^{-1}} = \frac{1}{Y_0}(1 + \epsilon' + \epsilon'' + \epsilon'\epsilon'') = \frac{1}{Y_0}(1 + Z_1)$ and $Y_0\widetilde{Y_0^{-1}} = 1 + Z_1$. The bounds on Z_1 are deduced from those on ϵ' and ϵ'': $0 \leq Z_1 < 2.5 \cdot 2^{-\alpha_0}$. ■

This theorem means that the multiplication $Y_0 \times \widetilde{Y_0^{-1}}$ will set to zero the bits of weight 2^{-1} to $2^{-\alpha_0+2}$ of its result.

Actually, in the case $\alpha_0 = 5$, one more bit is set to zero: The max error of the $\lceil\rceil$ operation – which is independent of the other bits of Y_0 – happens to be small enough to ensure $Y_0 \times \widetilde{Y_0^{-1}} \in [1, 1 + 2^{-4}]$. This bit of luck is best proven by enumeration. It doesn't seem to occur for larger values of α_0.

We now define $Y_1 = 1 + Z_1 = Y_0 \times \widetilde{Y_0^{-1}}$ and $0 \leq Z_1 < 2^{-p_1}$, with $p_1 = \alpha_0 - 2$ in the general case, and $p_1 = 4$ in the case $\alpha_0 = 5$. The multiplication $Y_0 \times \widetilde{Y_0^{-1}}$ is a rectangular one, since $\widetilde{Y_0^{-1}}$ is a $\alpha_0 + 1$-bit number. A_0 is also used to index a first logarithm table, that contains an accurate approximation L_0 of $\log(\widetilde{Y_0^{-1}})$ (the exact precision will be given later). This provides the first step of an iteration similar to (1):

$$\begin{aligned} \log(Y_0) &= \log(Y_0 \times \widetilde{Y_0^{-1}}) - \log(\widetilde{Y_0^{-1}}) \\ &= \log(1 + Z_1) - \log(\widetilde{Y_0^{-1}}) \\ &= \log(Y_1) - L_0 \end{aligned} \tag{5}$$

and the problem is reduced to evaluating $\log(Y_1)$.

B. Following iterations

The following iterations will similarly build a sequence $Y_i = 1 + Z_i$ with $0 \leq Z_i < 2^{-p_i}$. However, these iterations will differ in several ways.

- The sign of $\log(Y_0)$ is given by that of L_0, which is itself entirely defined by `FirstBit`. However, $\log(1 + Z_1)$ will be non-negative, as will be all the

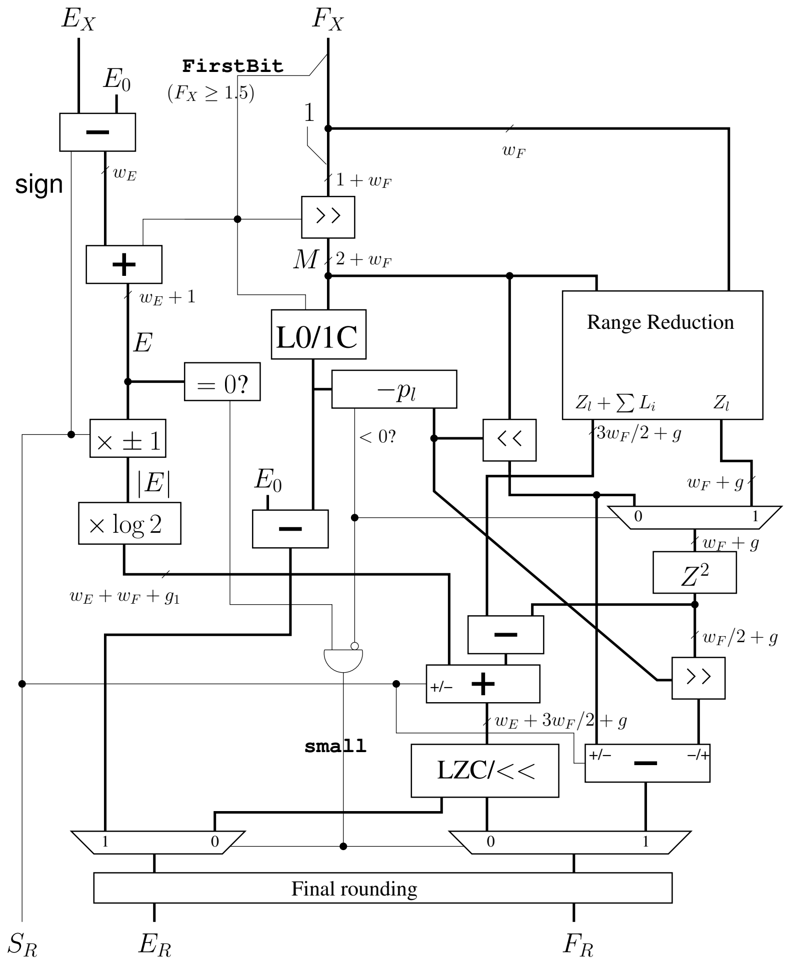

Fig. 1. Overview of the logarithm

following Z_i. This choice, motivated by simplicity, is discussed further in V-F.

- The following iterations no longer need a reciprocal table: A first-order Taylor approximation will be enough.

Let us now describe in detail the general iteration, starting from $i = 1$. We assume we have $Y_i = 1 + Z_i$ with $0 \leq Z_i < 2^{-p_i}$, and we want to build Z_{i+1} with $0 \leq Z_{i+1} < 2^{-p_{i+1}}$ (see Figure 2 for an illustration).

Let A_i be the subword composed of the α_i leading bits of Z_i (bits of absolute weight 2^{-p_i-1} to $2^{-p_i-\alpha_i}$, see Figure 2). An approximation of the reciprocal of $Y_i = 1 + Z_i$ is defined by

$$\widetilde{Y_i^{-1}} = 1 - A_i + E_i. \tag{6}$$

The term E_i is a single bit that will be defined below to ensure that the following holds:

Theorem V.2. *For all $i \geq 1$, we have*

$$0 \leq Y_{i+1} = 1 + Z_{i+1} = \widetilde{Y_i^{-1}} \times Y_i < 1 + 2^{-p_i-\alpha_i+1} \tag{7}$$

or, equivalently,

$$p_{i+1} = p_i + \alpha_i - 1. \tag{8}$$

In other words, using α_i bits in the computation (and, below, as inputs to the tables), we are able to zero out $\alpha_i - 1$ bits of our argument. This is slightly better than Wong and Goto [8] where 8 bits are zeroed using 10 bits. Approaches inspired by division algorithms [4] are able to zero α_i bits (one radix-2^{α_i} digit), but at a higher hardware cost due to the need for signed digit arithmetic.

Let us now try to prove theorem V.2 and define the value of E_i in the process.

Proof: As previously, let us call $\widetilde{Y_i} = 1 + A_i$ the approximation to Y_i obtained by considering only the α_i bits of Y_i of binary weights $-p_i - 1$ to $-p_i - \alpha_i$. This

$$Y_i = \quad 1.\overbrace{0\,0\,0\,0}^{p_i}\,\overbrace{\underbrace{\boxed{0\,0\,1\,1}}_{A_i}}^{\alpha_i}\,\overbrace{\underbrace{\boxed{0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,0\,0\,1\,0}}_{B_i}}^{Z_i}$$

$$Y_{i+1} = \quad 1.\overbrace{0\,0\,0\,0\,0\,0\,0\,0}^{p_{i+1}}\,\overbrace{\boxed{0\,1\,1\,0\,1\,0\,1\,1\,0\,1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,1\,1\,0\,1\,1\,0}}^{Z_{i+1}}$$

Fig. 2. Notations for one step of range reduction

truncation of Y_i corresponds to an absolute error $\widetilde{Y_i} = Y_i - \delta$ with $0 \leq \delta < 2^{-p_i-\alpha_i}$. As $Y_i \geq 1$, this absolute error also corresponds to a relative error $\widetilde{Y_i} = Y_i(1-\epsilon)$ with $0 \leq \epsilon < 2^{-p_i-\alpha_i}$.

It follows that $\dfrac{1}{\widetilde{Y_i}} = \dfrac{1}{Y_i}(1+\epsilon+\epsilon^2+...) = \dfrac{1}{Y_i}(1+\epsilon')$ with $0 \leq \epsilon' < 2^{-p_i-\alpha_i} + 2^{-2p_i-2\alpha_i+1}$.

Besides, the Taylor formula gives $\dfrac{1}{\widetilde{Y_i}} = 1 - A_i + A_i^2 - A_i^3 ... = 1 - A_i + \delta'$ with $0 \leq \delta' < 2^{-2p_i}$. If we use as approximation to $1/Y_i$ the value $1 - A_i = \dfrac{1}{\widetilde{Y_i}} - \delta'$, the product by Y_i could become negative. This is why we add the term $E_i = \max(\delta') = 2^{-2p_i}$. Now we have $1 - A_i + E_i = \dfrac{1}{\widetilde{Y_i}} + \delta''$ with $0 < \delta'' \geq 2^{-2p_i}$.

Finally, $1 - A_i + E_i = \frac{1}{Y_i}(1+\epsilon'+Y_i\delta'') = \frac{1}{Y_i}(1+Z_{i+1})$ with $0 \leq Z_{i+1} < 2^{-p_i-\alpha_i} + 2^{-2p_i-2\alpha_i+1} + 2^{-2p_i}(1+2^{-p_i})$.

To ensure that $0 \leq Z_{i+1} < 2^{-pi-\alpha_i+1}$ it is enough that $p_i > \alpha_i$. As a balanced architecture requires all the α_i to be roughly equal, we will have $p_i \approx i \times \alpha_i$, so this will be true from the third iteration ($i = 2$) onwards.

For the second iteration ($i = 1$), we add a small subtlety. The first iteration has defined $p_1 = \alpha_0 - 2$. To have $p_2 = p_1 + \alpha_1 - 1$, we would need need to take $\alpha_1 = p_1 - 1$ (at most), thus $\alpha_1 = \alpha_0 - 3$. The resulting architecture would not be balanced, in the sense that the first iteration requires 8 times more table storage than the following one, use larger multipliers, etc.

Our current implementation therefore uses for this iteration a value of E_i that is dependent on the value of A_i: $E_i = 2^{-2p_i}$ when the most significand bit of A_i is equal to 1, and $E_i = 2^{-2p_i-1}$ when this bit is equal to 0. This ensures $0 \leq \delta'' < 2^{-2p_i-1}$ in both cases. We may now use $\alpha_1 = p_1 = \alpha_0 - 2$ and still ensure $p_2 = p_1 + \alpha_1 - 1$. The cost is only one additional multiplexer in the computation of Z_{i+1}. ■

To compute Z_{i+1}, a full multiplication is not needed. Noting $Z_i = A_i + B_i$ (B_i consists of the lower bits of Z_i), we have $1 + Z_{i+1} = \widetilde{Y_i^{-1}} \times (1 + Z_i) = (1 - A_i + E_i) \times (1 + A_i + B_i)$, hence

$$Z_{i+1} = B_i - A_i Z_i + E_i(1 + Z_i) \tag{9}$$

Here the multiplication by E_i is just a shift, and the only real multiplication is the product $A_i Z_i$: The full computation of (9) amounts to the equivalent of a rectangular multiplication of $(\alpha_i + 2) \times s_i$ bits. Here s_i is the size of Z_i, which will vary between w_F and $3w_F/2$ (see below).

Finally, at each iteration, A_i is also used to index a logarithm table L_i (see Figure 3). All these logarithms have to be added, which can be done in parallel to the reduction of $1 + Z_i$. The output of the *Range Reduction* box is the sum of $Z_{\max}$ and this sum of tabulated logarithms, so it only remains to subtract the second-order term (Figure 1).

C. Iteration termination and error analysis

An important remark is that theorem V.2 still holds if Z_{i+1} (computed as per (9)) is truncated. Indeed, in the architecture, we will need to truncate it to limit the size of the computation datapath. Let us now address this question.

Let us denote l the index of the last iteration. We will stop the iteration as soon as Z_i is small enough for a second-order Taylor formula to provide sufficient accuracy. This also defines the threshold on leading zeroes/ones at which we choose to use the path computing $Z_0 - Z_0^2/2$ directly.

In $\log(1+Z_i) \approx Z_i - {Z_i}^2/2 + {Z_i}^3/3$, with $Z_i < 2^{-p_i}$, the third-order term is smaller than 2^{-3p_i-1}. We therefore stop the iteration at p_l such that $p_l \geq \lceil \frac{w_F}{2} \rceil$. This sets the target absolute precision of the whole datapath to $p_l + w_F + g \approx \lceil 3w_F/2 \rceil + g$.

The computation defined by (9) increases the size of Z_i. We will therefore truncate Z_i as soon as its LSB becomes smaller than this target precision. Figure 3 give an instance of this datapath in double precision.

Note that the architecture counts as many rectangular multipliers as there are stages, and may therefore be fully pipelined. Reusing one single multiplier would be possible [8], and would save a significant amount of hardware, but a high-throughput architecture is preferable in the FPGA context.

```
   Z0 : 0.1111001100110011001100110011001100110011001100110
   Z1 :     1001111111111111111111111111111111111111111111110010
   Z2 :       110101111111111111111111111111111111111111111100100111 00000
   Z3 :         0111010101110011111111111111111111111111111001010000101001100010 0000
   Z4 :           0110101101000000100100011011111111111111110010100001101000111101111001
   Z5 :             10011001111110110101011001110011011101100010000110101101001000 0110
   Z6 :               100011111101100000100101001011111100000011010001010110010111 1110
   Z7 :                 1011111011000000111001100000111010111011101001110101001100 01
   Z8 :                   101101100000011100100000110100000000101001011011011000110
   Z9 :                     0111000000111001000001001010001010000000001001001111 01
Z9Sq :                                        0011000100110001111100001
LogY9 :                     01110000001110010000010010100000111101101001010 1011100
   L0 : -0.00101101111000011010010100010101110010101101011010010111001101111100100100110 0110
   L1 :     1000001000001010111011000100111100111010001000100010001110000000101110011110 00
   L2 :       110010001001110011100011100000100101011001101101111001011000011100100110100
   L3 :         0110100000000101010010000101101110010001101001000100101111000000000111110
   L4 :           010110000000000000111100100000000110111011101011100011110111000010 1000
   L5 :             1000100000000000000100100001000000000110011001011010110100110111001
   L6 :               0111100000000000000000011100001000000000000010001100101000000000 0
   L7 :                 101010000000000000000000110111001000000000000000011000000111 10
   L8 :                   10101000000000000000000000001101110010000000000000000000 01100
LogY0 : -0.00011010010000110001110101011011110011000001100100111110010010110110100110001010 1
```

Fig. 3. Double-precision computation of $\log(Y_0)$ for $Y_0 = 0.95$. Parameters are $\alpha_0 = 5$ and $\alpha_i = 4$ for $i > 0$

D. Error analysis

We compute $E \log 2$ with $w_E + w_F + g_1$ precision, and the sum $E \log 2 + \log Y_0$ cancels at most one bit, so $g_1 = 2$ ensures faithful accuracy of the sum, assuming faithful accuracy of $\log Y_0$.

In general, the computation of $\log Y_0$ is much too accurate: As illustrated by Figure 3, the most significant bit of the result is that of the first non-zero L_i (L_0 in the example), and we have computed almost $w_F/2$ bits of extra accuracy. The errors due to the rounding of the L_i and the truncation of the intermediate computations are absorbed by this extra accuracy. However, two specific worst-case situation require more attention.

- When $Z_0 < 2^{-p_l}$, we compute $\log Y_0$ directly as $Z_0 - Z_0^2/2$, and this is the sole source of error. The shift that brings the leading one of $|Z_0|$ in position p_l ensures that this computation is done on $w_F + g$ bits, hence faithful rounding.
- The real worst case are when the exponent is zero and the higher bits of the mantissa are $Y_0 = 1 - 2^{-p_l+1}$: In this case we use the range reduction, knowing that it will cancel $p_l - 1$ bits of L_0 one one side, and accumulate rounding errors on the other side. We have l stages, each contributing at most 3 ulps of error: To compute (9), we first truncate Z_i to minimize multiplier size, then we truncate the product, and also truncate $E_i(1 + Z_i)$. Therefore we need $g = \lceil \log_2(3l) \rceil$ guard bits. For instance, for double-precision, we need $g = 4$ or $g = 5$, depending on the choice of α_i discussed below in VI-A.

E. Remarks on the L_i tables

When one looks at the L_i tables, one notices that some of their bits are constantly zeroes: Indeed they hold $L_i \approx -\log(1 - (A_i - E_i))$ which can for larger i be approximated by a Taylor series. We chose to leave the task of optimizing out these zeroes to the logic synthesizer. A natural idea would also be to store only $\log(1 - (A_i - \epsilon_i)) + (A_i - \epsilon_i)$, and construct L_i out of this value by subtracting $(A_i - \epsilon_i)$. However, the delay and LUT usage of this reconstruction would in fact be higher than that of storing the corresponding bits. With the FPGA target, the simpler approach is also the better.

There is another implementation trick. As $L_i \approx -\log(1 - (A_i - E_i))$ with E_i smaller than the unit in the last place of (A_i), all the entries are positive except the one for $A_i = 0$. To avoid having to manage signs in the reconstruction (which has a slight overhead) we add a small offset (equal to E_i) to all the table values except L_0, and we remove from L_0 the sum of all these offsets.

F. Discussion on the choice of unsigned arithmetic

Another option would be to keep all the Z_i as signed, two's compliment numbers. We have explored this option on paper, but it has not been fully implemented. This option is summarized as follows:

- All the Z_i are now signed, and bounded by $|Z_i| < 2^{-p_i}$, which defines p_i;
- Take as A_i the rounded value of Z_i to the bit of weight $2^{-p_i-\alpha_i}$, instead of the truncated value;

- Take as approximation to the inverse $\widetilde{Y_i^{-1}} = 1 - A_i$ (no correcting term E_i)
- The reduction iteration is simplified to $Z_{i+1} = B_i - A_i Z_i$.

We are then able to ensure $p_{i+1} = p_i + \alpha_i$ instead of $p_{i+1} = p_i + \alpha_i - 1$ (the proof is too similar to the previous one to deserve detailing – it also requires special care for the first and second iterations), so it seems we gain one bit per iteration. However we now also need one more bit to address the tables (the sign bit of A_i), so the required table size will be equivalent. The only real gain is to save the addition of the wide term $E_i(1 + Z_i)$, at the expense of a much smaller addition to obtain A_i by rounding, both being in the critical path.

We also now have to manage signed L_i, which means sign-extended additions. This should not impact neither area nor performance.

All things considered, we expect a small reduction in area and no improvement in performance or cycle count. This is currently not validated by an implementation.

VI. Implementation trade-offs

The FloPoCo implementation of the presented algorithms inputs w_E (the exponent size), w_F (the mantissa fraction size), and a third integer parameter introduced below, builds the architecture, and output synthesisable VHDL. It uses several sub-operators: pipelined integer multipliers, an integer squarer [27], a constant multiplier using the KCM algorithm [28], leading zero/one counters and shifters.

The exponent size has little impact on the performance and area of the design, and we will also not discuss it further.

Let us now discuss how to chose the value of the α_i parameters.

A. Setting the parameters

As suggested in Section III, sensible choices of α_i are either m (the LUT input size) if we want a LUT-only implementation (this was the focus of [25]), or, if we want to use embedded RAM and multiplier blocks, the maximum size that will balance their consumption. We want the user in control of this aspect. Any other choices could lead to a different area/speed tradeoff.

The current interface lets the user chose a maximum table input size $\alpha_{\max}$ (an integer between 5 and 16). The default is $\alpha_{\max} = 12$.

The implementation first tries to perform a range reduction using the parameters α_i and p_i set as follows (see V-B):

$$
\begin{aligned}
&\alpha_0 = \alpha_{\max}\\
&p_1 = \alpha_{\max} - 2\\
&\alpha_1 = \alpha_{\max} - 2\\
&p_2 = p_1 + \alpha_1 - 1\\
&\\
&i = 2\\
&\text{while } 2p_i \le w_F\\
&\qquad \alpha_i = \alpha_{\max}\\
&\qquad p_{i+1} = p_i + \alpha_i - 1
\end{aligned}
$$

However, when exiting the while loop, we have usually reduced more than strictly needed. It then makes sense to try to reduce the α_i: removing 1 to some α_i means halving the corresponding L_i table. The sum of the α_i is too large by $p_l - \lfloor w_F/2 \rfloor - 1$ bits. This is the total number of bits that may be removed from the α_i. The heuristic is as follows. First, all the α_i are decremented by the same value, then we decrease in priority the earlier ones, as they have more output bits and this will entail a larger memory saving.

For instance, for double precision,

- starting with $\alpha_{\max} = 12$, we end up with $(\alpha_0, \alpha_1, \alpha_2) = (11, 9, 11)$.
- Starting with $\alpha_{\max} = 10$, we need one more range reduction step and end up with $(\alpha_0, \alpha_1, \alpha_2, \alpha_3) = (9, 7, 8, 8)$.

B. Implementation trade-offs

We may now discuss the main implementation trade-off, taking double-precision as an example. Table I provides the corresponding synthesis results for a Virtex4 (xc4vlx15-12-sf363 using ISE 10.2). The target frequency is set to 200 MHz. The purpose of this table is not to expose all the possible trade-offs, but to convince the reader that the presented implementation is generic enough to be sucessfully targeted to different contexts.

The first line of this table ($\alpha_{\max} = 12$) represents the soft spot for a high-performance architecture with balanced consumption of embedded memories and multipliers. The second line ($\alpha_{\max} = 10$) requires overall less memory: although it needs one more table, each table is much smaller (our tables are expressed as truth tables, and we leave to the synthesis tool, here Xilinx ISE 10, the low-level decomposition into embedded memory blocks). On the other hand it needs more embedded multipliers, because it performs more iterations. The third line uses $\alpha_{\max} = 6$, a value that matches well a LUT-only implementation. We give two results: one where only the tables are implemented as LUTs, and one where the multiplications are also implemented as LUTs[3]. In this case the cycle count

[3] In both cases, this requires editing the generated VHDL to add attributes, or changing default synthesis options in the ISE tool

version	α_i	resources	performance
$\alpha_{\max} = 12$	11, 9, 11	1780 slices, 14 DSP48, 21 RAMB16	29 cycles @ 176 MHz
$\alpha_{\max} = 10$	9, 7, 8, 8	1870 slices, 18 DSP48, 10 RAMB16	35 cycles @ 176 MHz
$\alpha_{\max} = 6$	6, 4, 6, 6, 6, 6	2849 slices, 25 DSP48 4012 slices	29 cycles @ 131 MHz 29 cycles @ 148 MHz

TABLE I. Some implementation trade-offs for double-precision logarithm.

is the same as for $\alpha_{\max} = 12$: Although there are more iterations, the multiplications are smaller, and FloPoCo doesn't pipeline them as deeply as in the first case. As the frequency is lower, this shows that the performance model of the pipeline, internally built by the operator [29], lacks accuracy in this case. Hopefully, it will be refined, so that all frequencies come closer to the target frequency of 200MHz (probably at the expense of a longer latency).

It should be noted that the Virtex DSP blocks are always under-utilized in this architecture. Indeed, we need rectangular multipliers where one dimension is (more or less) α_i, and the other dimension is of the order of w_F, here more than 50. Such multipliers are built by assembling the 17x17-bit multipliers of the DSP48 blocks, but each DSP block is actually used as a α_ix17-bit multiplier. Some Altera FPGA offer the opportunity to partition a 18x18-bit multiplier into two 9x18 ones, and this would ensure near-optimal utilization in the $\alpha_{\max} = 10$ case.

All these results should improve as the FloPoCo framework is refined. In particular, we are currently refining the delay models and the associated generation of sub-components such as multipliers and shifters. The objective of FloPoCo is also to be portable to any FPGA family, which makes this task very complex. These issues are out of scope of this article, although the logarithm generator makes a good case study.

C. Varying the precision

If we consider $\alpha_{\max}$ fixed, the cost of the operator is roughly quadratic with w_F: The number of iterations is proportional to w_F, and each iteration consists of a table look-up and a rectangular multiplication with one dimension constant (roughly $\alpha_{\max}$) and one dimension roughly proportional to w_F. This is illustrated by the synthesis results given in Table II (for a Virtex4 xc4vlx15-12-sf363 using ISE 10.2).

This table also provides results for the previous state of the art: FPLibrary operators[4], which are pipelined versions of those published in [23]. It uses a table-based method which grows exponentially with w_F, and will not be relevant beyond single precision. However, it compares well to the iterative algorithm for single precision, and

[4]http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/

	Slices	DSP48E	RAM blocks
available	37,440	1,056	1,032
used	2,247	14	12
percent	6%	1.3%	1.16%

TABLE III. A double-precision logarithm on the largest Virtex-5 chip

is definitely more attractive for lower precisions. The conclusion is that eventually, the table-based algorithm should be ported to FloPoCo, too.

D. Comparing with processor performance

In this section, we target our operator at the largest computation-oriented Xilinx FPGA currently commercially available, the Virtex-5 XC5VSX240T. Synthesis results for this target are summarized in Table III. The corresponding operator runs at 208MHz and thus computes 200 MFPLog/s. This table also shows that we can theoretically pack 16 such operators on a single FPGA circuit, for a theoretical peak performance of 3.2 GFPLog/s.

By comparison, the best reported double-precision logarithm implementation on a processor are due to Intel on the Itanium-2 (36 cycles/FPLog at 2GHz [30]), exploiting the dual, extended precision fused multiply-and-add of this architecture. On IA32 processors, carefully optimized implementations still require more than 100 cycles at 4GHz [12]. We conclude that the peak single-core performance of a contemporary processor is about 50 MFPLog/s.

If we now exploit parallelism, a four-core processor can offer the throughput of one of our logarithm operators, about 200 MFPLog/s. However, we can also pack 16 logarithm operators on a single FPGA chip. The peak MFPLog/s performance of a high-end FPGA is thus 16 times that of a high-end processor. This is much better than the balanced MFLops comparison one obtains when considering only floating-point additions and multiplications [18].

(w_E, w_F)	resources	performance
(15,63) (double-extended)	2365 slices, 20 DSP48, 17 RAMB16	33 cycles @ 130 MHz
(11,52) (double precision)	1780 slices, 14 DSP48, 21 RAMB16	29 cycles @ 176 MHz
(9, 38)	1194 slices, 11 DSP48, 6 RAMB16	24 cycles @ 208 MHz
(8, 23) (single precision)	601 slices, 5 DSP48, 3 RAMB16	17 cycles @ 250 MHz
(8, 23) FPLibrary	*1073 slices, 0 DSP48, 3 RAMB16*	*11 cycles @ 201 MHz*
(7, 16)	415 slices, 4 DSP48, 2 RAMB16	16 cycles @ 263 MHz
(7,16) FPLibrary	*621 slices, 1 DSP48, 0 RAMB16*	*9 cycles @ 200 MHz*

TABLE II. Maximum frequency operators for several precisions on Virtex-4

k	d	coefficients	multipliers
10	4	54, 44, 34, 24, 14 **10 RAMB16**	44x44, 34x34, 24x24, 14x14 (9 +) 9 + 4 + 4 +1 = **27 DSP48**
12	3	54, 42, 30, 18 **32 RAMB16**	42x42, 30x30, 18x18 (9 +) 9 + 4 +1 = **23 DSP48**

TABLE IV. Estimated cost of double-precision polynomial-based logarithm implementations on Virtex-4

VII. Multiplicative range reduction versus polynomial approximation

As (to our knowledge) all libm implementations use polynomial approximation to compute logarithms, we cannot escape a comparison with this solution. The initial range-reduction is identical, so we get back to the problem of computing $\log Y_0$ with $Y_0 \in [0.75, 1.5]$.

Let us first make some remarks on the evaluation of a polynomial of degree d for an argument z such that $|z| < 2^{-k}$. The Horner scheme allows us to evaluate this polynomial in d additions and d multiplications:

$$p(z) = a_0 + z \times (a_1 + z \times (a_2 + ... + a_d \times z))...)$$

The Horner scheme is very stable if $|z| < 2^{-k}$: any error performed at one step is multiplied by z, in other terms scaled down. An often overlooked consequence of this is that a_1 need not be as accurate as a_0, a_2 need not be as accurate as a_1, etc. As a numerical rule of thumb (valid if the derivatives of the function are reasonably bounded, which is true for the logarithm around 1), if we want p bits of accuracy, we need a_0 accurate at least to p bits, but a_1 may be accurate to $p-k$ bits, a_2 to $p-2k$ bits, etc. Beyond this rule of thumb, the Sollya polynomial approximation tool[5] optimises the actual sizes of the coefficients [31].

What's more, it is possible to truncate also z in the earlier steps of the computation, and still get an accurate result at the end. Numerically, z need not be more accurate than the term it is multiplied to, which is of the order of the corresponding coefficient a_i. Such truncation is never performed in software as it would entail more work, not less, but it can save hardware when targeting an FPGA.

Let us now describe an architecture parameterized by k. The interval $[0.75, 1.5]$ is split into 2^k sub-intervals. On each sub-interval, the logarithm is approximated by a polynomial of degree d, chosen as the smallest degree such that the absolute error of the polynomial approximation is smaller than $2^{-3w_F/2}$ (we still have to manage the vanishing logarithm around 1). We therefore have 2^k polynomials with $d+1$ coefficients each. These coefficient are read from a table indexedby the k leading bits of Y_0, and z is composed of the remaining bits (a $w_E - k$-bit number), considered as an offset with respect to the center of the interval, so that $|z| < 2^k$.

It is easy to obtain the degree corresponding to a given k, using Maple [24] or the `guessdegree` function of the Sollya tool. In turn, the previous rule of thumb allows us to evaluate the coefficient size, hence the memory requirements, and the multiplier sizes, hence the embedded multiplier requirements. This is only an evaluation, and for the purpose of comparison we keep it optimistic with respect to an actual implementation.

For double-precision ($w_F = 53$, $3w_F/2 = 80$), we get for instance the following implementation point: for $k = 13$, we need a polynomial of degree 5. The coefficient sizes are 80, 67, 54, 41, 28, and 15 bits. The total memory needed is $2^{13} \times (80 + 67 + 54 + 41 + 28 + 15)$. Dividing this amount by the size (18Kbits) of an embedded memory block of the Virtex-4 family (RAMB16) we conclude that we need 127 RAMB16. The multiplications are of sizes 40x67, 40x54, 40x41, 40x28 and 40x15. We also divide them by the size of a Virtex-4 DSP48 embedded multiplier (18x18 bits), and we get a DSP48 consumption of 9+9+9+6+3= 36 DSP48 blocks. Comparing these two

[5] http://sollya.gforge.inria.fr/

numbers with Table I, the iterative range reduction seems definitely more attractive.

The problem is that we have suggested to compute with 80-bit absolute accuracy. But this accuracy is only needed when $E = 0$ and Y_0 is very close to 1, because the logarithm vanishes and we need w_F bits of the result. In this region, evaluating the polynomial in floating-point would make much more sense, but be much more expensive.

A trick will save us the price of a full floating-point computation. Let us rewrite the logarithm as

$$\log(1+z) = z \times \frac{log(1+z)}{z}.$$

We may now evaluate $\frac{log(1+z)}{z}$ as a piecewise polynomial in fixed point, to 2^{-w_F} only. Then the multiplication by $z = Y_0 - 1$ is computed exactly – using a square multiplier of $(w_F + g) \times (w_F + g)$ bits – and the product needs to be normalized. The position of the leading bit is almost known already thanks to the L0/1C box of figure 1. The cost of this normalization is thus similar to the cost of the normalizer in the iterative approach.

If we now evaluate the cost of approximating $\frac{log(1+z)}{z}$ as a piecewise polynomial, we get, for double-precision, the implementation points reported in Table IV (which includes, between parentheses, the cost of the final multiplication by z).

Again, the comparison with Table I is favourable to the iterative range reduction. The margin is smaller, but still sufficient to convince us that even a finely optimized polynomial implementation will yield no clear improvement.

Note that many software implementations use a table-based range reduction [11] very similar to our first iteration (typically with $\alpha_0 = 8$) before approximating $\log(1 + Z_1)$ as a polynomial of small degree. This is yet another intermediate option, but there is no reason to believe it will bring in any decisive improvement.

VIII. Conclusion and future work

By retargeting an old algorithm to the specific fine-grained structure of FPGAs, this work shows that elementary functions, up to double precision and beyond, can be implemented in a small fraction of current FPGAs. The resulting operators have low resource usage and high troughput. Their raw performance surpasses the equivalent processor implementations. They have a long latency compared to adders or multipliers, but this latency is still much shorter than that of their software equivalent. They are flexible, exposing a trade-off between memory resources and computing resources.

FPGAs, when used as co-processors, are often limited by their input/output bandwidth to the processor or memory. From an application point of view, the availability of compact elementary functions for the FPGA, bringing elementary functions on-board, will also help conserve this bandwidth.

The roadmap ahead is that of a complete libm, with exponential, [25], sine and cosine [32], [8] and their inverses, $\arctan \frac{x}{y}$ [8], and others.

In the shorter term, the presented implementation will be optimized further, in particular to increase its working frequency. It should also be optimized for lower frequencies, regrouping iterations to reduce the cycle count and the pipeline overhead in this case.

As the most complex operator written in FloPoCo so far, the logarithm will be a precious case study driving improvements to the framework itself [29]. It actually contributed to motivate it.

But this logarithm implementation is also a flagship of the FloPoCo project, supporting the thesis [22] that FPGAs can offer tremendous floating-point performance thanks to non-standard operators.

References

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

[2] ISO/IEC, *International Standard ISO/IEC 9899:1999(E). Programming languages – C*, 1999.

[3] 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.

[4] M. Ercegovac, "Radix-16 evaluation of certain elementary functions," *IEEE Transactions on Computers*, vol. C-22, no. 6, pp. 561–566, 1973.

[5] C. Wrathall and T. C. Chen, "Convergence guarantee and improvements for a hardware exponential and logarithm evaluation scheme," in *4th Symposium on Computer Arithmetic*, 1978, pp. 175–182.

[6] P. Farmwald, "High-bandwidth evaluation of elementary functions," in *5th Symposium on Computer Arithmetic*, 1981, pp. 139–142.

[7] M. Cosnard, A. Guyot, B. Hochet, J. M. Muller, H. Ouaouicha, P. Paul, and E. Zysmann, "The FELIN arithmetic coprocessor chip," in *8th Symposium on Computer Arithmetic*, 1987, pp. 107–112.

[8] W. F. Wong and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 278–294, 1994.

[9] ——, "Fast evaluation of the elementary functions in single precision," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 453–457, 1995.

[10] P. T. P. Tang, "Table-driven implementation of the exponential function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 15, no. 2, pp. 144–157, 1989.

[11] ——, "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 378 – 400, 1990.

[12] C. S. Anderson, S. Story, and N. Astafiev, "Accurate math functions on the Intel IA-32 architecture: A performance-driven design," in *7th Conference on Real Numbers and Computers*, 2006, pp. 93–105.

[13] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," in *12th Symposium on Computer Arithmetic*. Bath, UK: IEEE, 1995, pp. 10–16.

[14] J. Stine and M. Schulte, "The symmetric table addition method for accurate function approximation," *Journal of VLSI Signal Processing*, vol. 21, no. 2, pp. 167–177, 1999.

[15] D. Lee, A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1520–1531, 2005.

[16] 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.

[17] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *ACM/SIGDA Field-Programmable Gate Arrays*. ACM, 2004.

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

[19] G. Lienhart, A. Kugel, and R. Männer, "Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations," in *FPGAs for Custom Computing Machines*. IEEE, 2002.

[20] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Field-Programmable Gate Arrays*. ACM, 2005, pp. 75–85.

[21] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Field-Programmable Gate Arrays*. ACM, 2005, pp. 86–95.

[22] 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.

[23] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing*, vol. 31, no. 8, pp. 537–545, 2007.

[24] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, 2nd ed. Birkhäuser, 2006.

[25] 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.

[26] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

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

[28] K. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, May 1994.

[29] F. de Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *Field Programmable Logic and Applications*. IEEE, Aug. 2009.

[30] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium®-based Systems*. Intel Press, 2002.

[31] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞- approximations," in *18th Symposium on Computer Arithmetic*. IEEE Computer Society Press, 2007, pp. 169–176.

[32] J. Detrey and F. de Dinechin, "Floating-point trigonometric functions for FPGAs," in *Field-Programmable Logic and Applications*. IEEE, 2007, pp. 29–34.