Return of the hardware floating-point elementary function

Jérémie Detrey Florent de Dinechin Xavier Pujol
LIP, École Normale Supérieure de Lyon
46 allée d'Italie
69364 Lyon cedex 07, France
{Jeremie.Detrey, Florent.de.Dinechin, Xavier.Pujol}@ens-lyon.fr

Abstract

The study of specific hardware circuits for the evaluation of floating-point elementary functions was once an active research area, until it was realized that these functions were not frequent enough to justify dedicating silicon to them. Research then turned to software functions. This situation may be about to change again with the advent of reconfigurable co-processors based on field-programmable gate arrays. Such co-processors now have a capacity that allows them to accomodate double-precision floating-point computing. Hardware operators for elementary functions targeted to such platforms have the potential to vastly outperform software functions, and will not permanently waste silicon resources. This article studies the optimization, for this target technology, of operators for the exponential and logarithm functions up to double-precision. These operators are freely available from `www.ens-lyon.fr/LIP/Arenaire/`.

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

1 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 [2] 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 [14].

The question wether elementary functions should be implemented in hardware was controversial in the beginning of the PC era [18]. The literature indeed offers many articles describing hardware implementations of FP elementary functions [10, 26, 12, 4, 23, 24, 25]. 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, which leads to much slower performance. 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 pre-computed values [20, 21]. 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 [1]. Research on hardware elementary functions has since then mostly focused on approximation methods for fixed-point evaluation of functions [13, 19, 15, 8].

Lately, a new kind of programmable circuit has also 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, but FPGAs are reconfigurable and therefore offer a flexibility comparable to that of the microprocessor.

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, is not the case of floating-point computing: An FP operation is, as

18th IEEE Symposium on Computer Arithmetic(ARITH'07)
0-7695-2854-6/07 $20.00 © 2007 IEEE

already mentioned, typically ten times slower in FPGA than if computed in the highly optimized FPU of the processor. 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 [22], and accelerated FP computing has been reported in single precision [16], then in double-precision [5, 9]. Mainstream computer vendors such as Silicon Graphics and Cray now build computers with FPGA accelerators—although to be honest, they do not advertise them (yet) as FP accelerators.

With this new technological target, the subject of hardware implementation of floating-point elementary functions becomes a hot topic again. Indeed, previous work has shown that a single instance of an exponential [7] or logarithm [6] operator can provide ten times the performance of the processor, while consuming a small fraction of the resources of current FPGAs. 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 architectures of [6, 7] use a generic table-based approach [8], which doesn't scale well beyond single precision: Its size grows exponentially.

In this article, we demonstrate a more algorithmic approach, which is a synthesis of much older works, including the Cordic/BKM family of algorithms [17], the radix-16 multiplicative normalization of [10], Chen's algorithm [26], an *ad-hoc* algorithm by Wong and Goto [24], and probably many others [17]. All these approaches boil down to the same basic properties of the logarithm and exponential functions, and are synthesized in Section 2. The specificity of the FPGA hardware target are summarized in Section 3, and the optimized algorithms are detailed and evaluated in Section 4 (logarithm) and Section 5 (exponential). Section 6 provides performance results (area and delay) from actual synthesis.

2 Iterative exponential and logarithm

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.

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 [10, 24, 17].

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.

3 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 [11]), 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 square 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. For most current FPGAs, we have $m = 4$, and we will use this value in the sequel. However, older FPGAs used $m = 3$, while the most recent Virtex-5 uses $m = 6$, so there is a trend to increase granularity, and it is important to design algorithm parameterized by m.

  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.

18th IEEE Symposium on Computer Arithmetic(ARITH'07)
0-7695-2854-6/07 $20.00 © 2007 IEEE

- 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.
- In the elementary block, each LUT is followed by a 1-bit register, which may be used or not. For our purpose it means that turning a combinatorial circuit into a pipelined one means using a resource that is present, not using more resources (in practice, however, a pipelined circuit will consume marginally more resources).
- Recent FPGAs include a limited number of small multipliers or mult-accumulators, typically for 18 bits times 18 bits. In this work, we choose not to use them.

4 A hardware logarithm operator

4.1 First range reduction

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 $\log(2-2\epsilon) - \log(2)$, meaning a catastrophic cancellation. To avoid this case, the following error-free transformation is applied to the input:

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

And the logarithm is evaluated as follows:

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

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[1].
  - 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 4.3.

4.2 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.

Let A_0 be the first 5 bits of the mantissa (including `FirstBit`). A_0 is used to index a table which gives an approximation $\widetilde{Y_0^{-1}}$ of the reciprocal of Y_0 on 6 bits. Noting $\widetilde{Y_0}$ the mantissa where the bits lower than those of A_0 are zeroed ($\widetilde{Y_0} = 1.0aaaaa$ or $\widetilde{Y_0} = 0.11aaaaa$, depending on `FirstBit`), the first reciprocal table stores

[1] 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.

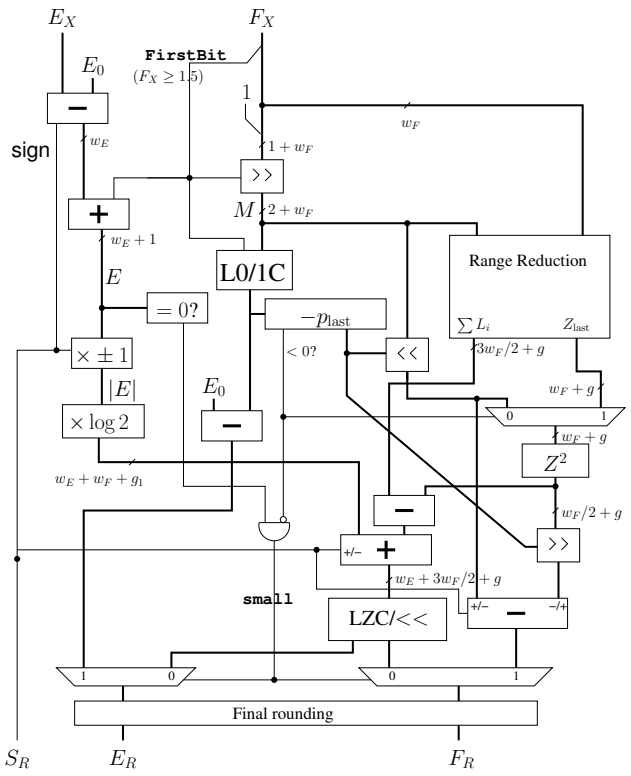

Figure 1. Overview of the logarithm

$$\widetilde{Y_0^{-1}} = 2^{-5} \left\lceil \frac{2^6}{\widetilde{Y_0}} \right\rceil \tag{4}$$

The reader may check that these values ensure $Y_0 \times \widetilde{Y_0^{-1}} \in [1, 1 + 2^{-4}]$. Therefore we define $Y_1 = 1 + Z_1 = Y_0 \times \widetilde{Y_0^{-1}}$ and $0 \leq Z_1 < 2^{-p_1}$, with $p_1 = 4$. The multiplication $Y_0 \times \widetilde{Y_0^{-1}}$ is a rectangular one, since $\widetilde{Y_0^{-1}}$ is a 6-bit only 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):

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

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

The following iterations will similarly build a sequence $Y_i = 1 + Z_i$ with $0 \leq Z_i < 2^{-p_i}$. Note that the sign of $\log(Y_0)$ will always be 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 following Z_i (see Figures 2 and 3).

Let us now define the general iteration, starting from $i = 1$. 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 $2^{-p_i-\alpha_i}$). A_i will be used to address the logarithm table L_i. As suggested in Section 3, we choose $\alpha_i = 4 \quad \forall i > 0$ to minimize resource usage, but another choice could lead to a different area/speed tradeoff. For instance, the architecture by Wong and Goto [24] takes $\alpha_i = 10$. Note that we used $\alpha_0 = 5$, because $\alpha_0 = 4$ would lead to $p_1 = 2$, which seems a worse tradeoff.

```
   Y0 : 1.00110011001100110011001100110                    t=0
   Z1 :       0011001100110011001100000010                 t=2
   Z2 :          010101011100110011000001000001101         t=4
   Z3 :             011010110100101011101110110             t=6
   Z4 :                1001101000001101101100010            t=8
 Z4Sq :                                  0101110010         t=9
LogY4 :                1001101000001100010000000            t=10
   L0 :  .00101011011111101000000011010110101101            t=1
   L1 :       0010100000110010010100111111100101            t=3
   L2 :          0100100000010100010001111100110            t=5
   L3 :             0101100000000011110010000001            t=7
LogY0 :  .00101110101011001001111111111100100001            t=11
```

Figure 2. Single-precision computation of $\log(Y_0)$ for $Y_0 = 1.2$

The following iterations no longer need a reciprocal table: An approximation of the reciprocal of $Y_i = 1 + Z_i$ is defined by

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

The term ϵ_i is a single bit that will ensure $\widetilde{Y_i^{-1}} \times Y_i \geq 1$. We define it as

$$\begin{cases} \epsilon_i = 2^{-p_i-\alpha_i} & \text{if } \alpha_i + 1 \geq p_i \text{ and } MSB(A_i) = 0 \\ \epsilon_i = 2^{-p_i-\alpha_i-1} & \text{otherwise} \end{cases} \tag{7}$$

With definitions (6) and (7), it is possible to show that the following holds:

$$0 \leq Y_{i+1} = 1 + Z_{i+1} = \widetilde{Y_i^{-1}} \times Y_i < 1 + 2^{-p_i-\alpha_i+1} \tag{8}$$

The proof is not difficult, considering (9) below, but is too long to be exposed here. Note that (7) is inexpensive to implement in hardware: The case $\alpha_i + 1 \geq p_i$ happens at most once in practice, and $2^{-p_i-\alpha_i-1}$ is one half-ulp of A_i.

In other words, (8) ensures $p_{i+1} = p_i + \alpha_i - 1$. Or, using α_i bits of table address, we are able to zero out $\alpha_i - 1$ bits of our argument. This is slightly better than [24] where $\alpha_i - 2$ bits are zeroed. Approaches inspired by division algorithms [10] 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.

With $\alpha_i = 4$ on an FPGA, the main cost is not in the L_i table (at most one LUT per table output bit), but in the multiplication. However, a full multiplication is not needed. Noting $Z_i = A_i + B_i$ (B_i consists of the lower bits of Z_i),

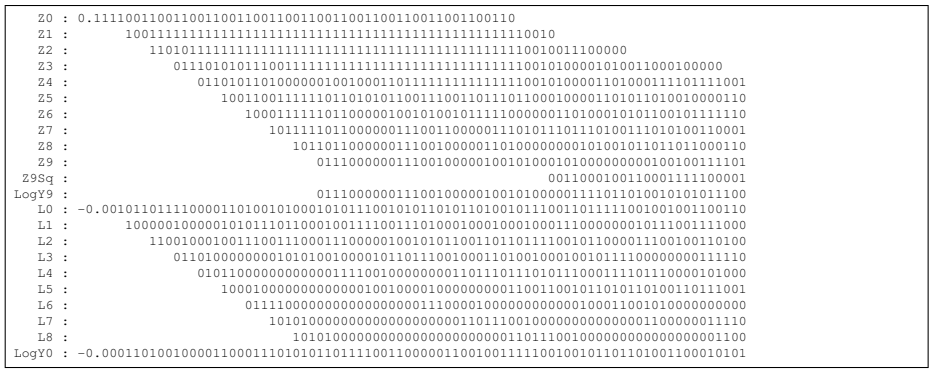

Figure 3. Double-precision computation of $\log(Y_0)$ for $Y_0 = 0.95$.

we have $1 + Z_{i+1} = \widetilde{Y_i^{-1}} \times (1 + Z_i) = (1 - A_i + \epsilon_i) \times (1 + A_i + B_i)$, hence

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

Here the multiplication by ϵ_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).

An important remark is that (8) still holds if the product 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.

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 $p_{\max}$ such that $p_{\max} \geq \lceil \frac{w_F}{2} \rceil$. This sets the target absolute precision of the whole datapath to $p_{\max} + w_F + g \approx \lceil 3w_F/2 \rceil + g$. The computation defined by (9) increases the size of Z_i, which will be truncated as soon as its LSB becomes smaller than this target precision. Figures 2 and 3 give an instance of this datapath in single and double precision respectively. 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, and would save a significant amount of hardware, but a high-throughput architecture is preferable.

Finally, at each iteration, A_i is also used to index a logarithm table L_i (see Figures 2 and 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).

4.3 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 2, 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 $Z_0 < 2^{-p_{\max}}$, 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 $p_{\max}$ ensures that this computation is done on $w_F + g$ bits, hence faithful rounding.

- The real worst case is when $Y_0 = 1 - 2^{-p_{\max}+1}$: In this case we use the range reduction, knowing that it will cancel $p_{\max} - 1$ bits of L_0 one one side, and accumulate rounding errors on the other side. We have $\max$ 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 $\epsilon_i(1 + Z_i)$. Therefore we need $g = \lceil \log_2(3 \times \max) \rceil$ guard bits. For double-precision, this gives $g = 5$.

4.4 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 $L_i \approx \log(1 - (A_i - \epsilon_i))$ 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. As a conclusion, with the FPGA target, the simpler approach is also the better. The same remark will apply to the tables of the exponential operator.

5 A hardware exponential algorithm

5.1 Initial range reduction

The range reduction for the exponential operator is directly inspired from the method presented in [7]. The first step transforms the floating-point input X into a fixed-point number X_{fix} thanks to a barrel shifter. Indeed, if $E_X > w_E - 1 + \log_2(\log 2)$, then the result overflows, while if $E_X < 0$, then X is close to zero and its exponential will be close to $1+X$, so we can loose the bits of X of absolute weight smaller than 2^{-w_F-g}, g being the number of guard bits required for the operator (typically 3 to 5 bits). Thus X_{fix} will be a w_E+w_F+g-bit number, obtained by shifting the mantissa $1.F_X$ by at most w_E-1 bits on the left and w_F+g bits on the right.

This fixed-point number is then reduced in order to obtain an integer E and a fixed-point number Y such that $X \approx E\cdot\log 2 + Y$ and $0 \le Y < 1$. This is achieved by first multiplying the most significant bits of X_{fix} by an approximation to $1/\log 2$ and then truncating the result to obtain E. Then Y is computed as $X_{\text{fix}} - E\cdot\log 2$, requiring a rectangular multiplier.

After computing e^Y thanks to the iterative algorithm detailed in the next section, we have $e^X \approx e^{X_{\text{fix}}} \approx e^Y \cdot 2^E$, and a simple renormalization and rounding step reconstructs the final result. The approximation to $1/\log 2$ is chosen so that this step shifts by at most one bit.

Figure 4 presents the overall architecture of the exponential operator. Details for the computation of e^Y-1 are given in Figure 5 and presented in the next section.

5.2 Iterative range reduction for the fixed-point exponential

Starting with the fixed-point operand $Y \in [0,1)$, we compute e^Y-1 using a refinement of (1). Let $Y_0 = Y$. Each iteration will start with a fixed-point number Y_i and compute Y_{i+1} closer to 0, until Y_k is small enough to evaluate $e^{Y_k}-1$ thanks to a simple table or a Taylor approximation.

At each iteration, Y_i, as for the logarithm operator, is split into subwords A_i and B_i of α_i and β_i bits respectively. A_i addresses two tables (again, the choice of $\alpha_i = m$ for most iterations will optimize FPGA resource usage). The

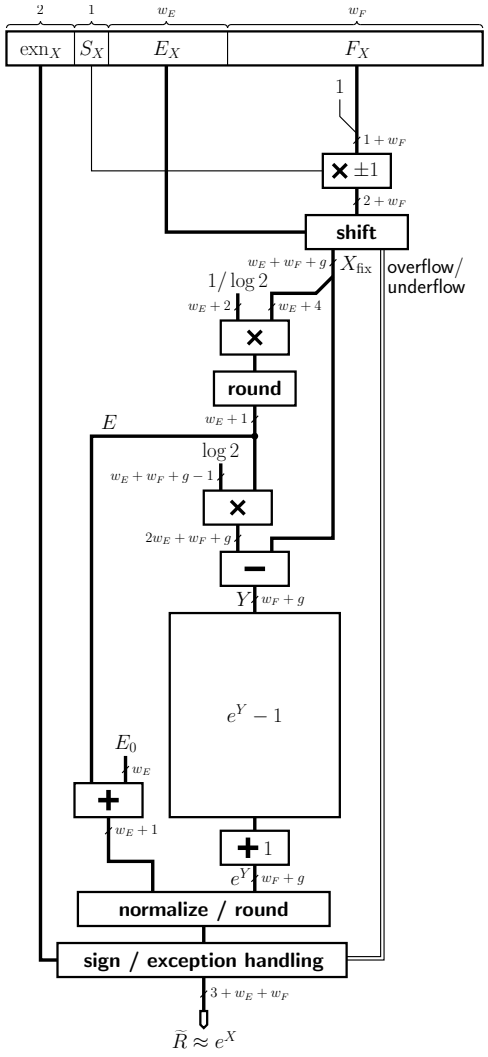

Figure 4. Overview of the exponential

first table holds approximations of $e^{Y_i}-1$ rounded to only α_i bits, which we note $\widetilde{e^{Y_i}}-1$. The second one holds $L_i = \log\left(\widetilde{e^{Y_i}}\right)$ rounded to $\alpha_i+\beta_i$ bits.

Obviously L_i is quite close to Y_i. One may check that computing Y_{i+1} as the difference $Y_i - L_i$ will result in cancelling the $\alpha_i - 1$ most significant bits of Y_i. The number Y_{i+1} fed into the next iteration is therefore a $1+\beta_i$-bit number.

The reconstruction of the exponential uses the following recurrence with decreasing i:

$$\begin{aligned}
&\left(\widetilde{e^{Y_i}}-1\right)\times\left(e^{Y_{i+1}}-1\right)+\left(\widetilde{e^{Y_i}}-1\right)+\left(e^{Y_{i+1}}-1\right)\\
&= \widetilde{e^{Y_i}}\cdot e^{Y_{i+1}} - 1 \quad = \quad \widetilde{e^{Y_i}}\cdot e^{Y_i-L_i} - 1\\
&= \widetilde{e^{Y_i}}\cdot e^{Y_i}\cdot e^{-\log\left(\widetilde{e^{Y_i}}\right)} - 1\\
&= e^{Y_i} - 1.
\end{aligned}$$

Here $e^{Y_{i+1}} - 1$ comes from the previous iterations, and $\widetilde{e^{Y_i}} - 1$ is an α_i-bit number, so the product needs a rectangular multiplier.

This way, the k steps of reconstruction finally give the result $e^{Y_0} - 1 = e^Y - 1$. The detailed architecture of this iterative method is presented Figure 4.

We have performed a detailed error analysis of this algorithm to ensure the faithful rounding of the final result. Due to space restrictions, this analysis is not presented in this article.

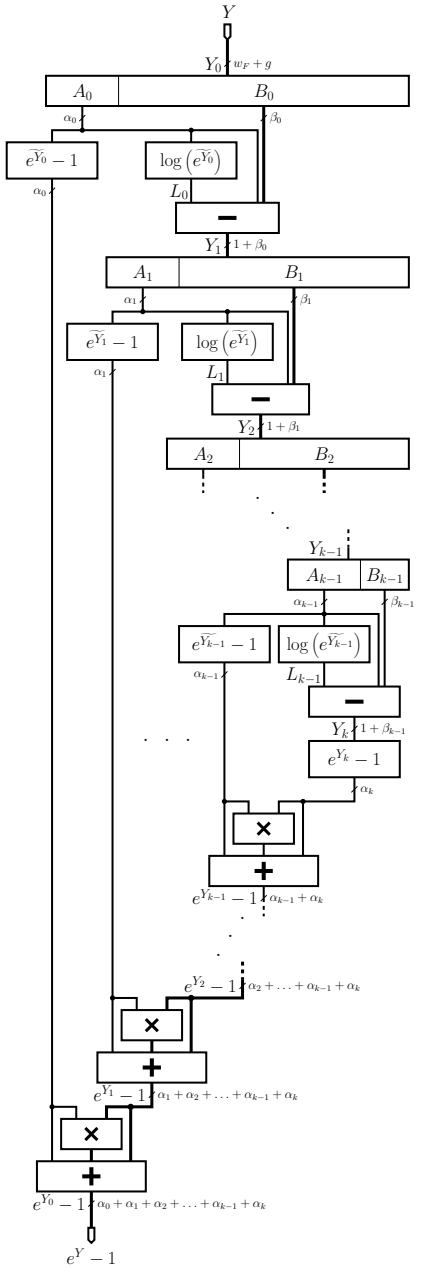

Figure 5. Computation of $e^Y - 1$

6 Area and performance

The presented algorithms are implemented as C++ programs that, given w_E, w_F and possibly the α_i, compute the various parameters of the architecture, and output synthesisable VHDL. Some values of area and delay (obtained using Xilinx ISE/XST 8.2 for a Virtex-II XC2V1000-4 FPGA) are given in Table 1 (where a *slice* is a unit containing two LUTs).

As expected, the operators presented here are smaller but slower than the previously reported ones. More importantly, their size is more or less quadratic with the precision, instead of exponential for the previously reported ones. This allows them to scale up to double-precision. For comparison, the FPGA used as a co-processor in the Cray XD1 system contains more than 23,616 slices, and the current largest available more than 40,000, so the proposed operators consume about one tenth of this capacity. To provide another comparison, our operators consume less than twice the area of an FP multiplier for the same precision reported in [22].

Exponential				
Format	This work		Previous [7]	
(w_E, w_F)	Area	Delay	Area	Delay
(7, 16)	**472**	**118**	480	69
(8, 23)	**728**	**123**	948	85
(9, 38)	**1242**	**175**	–	–
(11, 52)	**2045**	**229**	–	–
Logarithm				
Format	This work		Previous [6]	
(w_E, w_F)	Area	Delay	Area	Delay
(7, 16)	**556**	**70**	627	56
(8, 23)	**881**	**88**	1368	69
(9, 38)	**1893**	**149**	–	–
(11, 52)	**3146**	**182**	–	–

Table 1. Area (in Virtex-II slices) and delay (in ns) of implementation on a Virtex-II 1000

These operators will be easy to pipeline to function at the typical frequency of FPGAs—100MHz for the middle-range FPGAs targeted here, 200MHz for the best current ones. This is the subject of ongoing work. The pipeline depth is expected to be quite long, up to about 30 cycles for double precision, when a double-precision multiplier is typically about 10 cycles. It will also neead about 10% more slices. As mentioned in [7] and [6], one exponential or logarithm per cycle at 100MHz is ten times the throughput of a 3GHz Pentium, and comparable to peak Itanium-II performance [3] using loop-optimized elementary functions. However, the proposed functions consume only a fraction of the FPGA resources.

7 Conclusion and future work

By retargeting an old family of algorithms to the specific fine-grained structure of FPGAs, this work shows that elementary functions up to double precision can be implemented in a small fraction of current FPGAs. The resulting operators have low resource usage and high troughput, but long latency, which is not really a problem for the envisioned applications.

FPGAs, when used as co-processors, are often limited by their input/output bandwidth to the processor. 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 same principles can be used to compute sine and cosine and their inverses, using the complex identity $e^{jx} = \cos x + j \sin x$. The architectural translation of this identity, of course, is not trivial. Besides the main cost with trigonometric functions is actually in the argument reduction involving the transcendental number π. It probably makes more sense to implement functions such as $\sin(\pi x)$ and $\cos(\pi x)$. A detailed study of this issue is also the subject of current work.

References

[1] 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*, pages 93–105, 2006.

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

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

[4] M. Cosnard, A. Guyot, B. Hochet, J. M. Muller, H. Ouaouicha, P. Paul, and E. Zysmann. The FELIN arithmetic coprocessor chip. In *Eighth IEEE Symposium on Computer Arithmetic*, pages 107–112, 1987.

[5] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *ACM/SIGDA Field-Programmable Gate Arrays*, pages 75–85. ACM Press, 2005.

[6] J. Detrey and F. de Dinechin. A parameterizable floating-point logarithm operator for FPGAs. In *39th Asilomar Conference on Signals, Systems & Computers*. IEEE Signal Processing Society, Nov. 2005.

[7] J. Detrey and F. de Dinechin. A parameterized floating-point exponential function for FPGAs. In *IEEE International Conference on Field-Programmable Technology (FPT'05)*. IEEE Computer Society Press, Dec. 2005.

[8] J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In *16th Intl Conference on Application-specific Systems, Architectures and Processors*. IEEE Computer Society Press, July 2005.

[9] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *ACM/SIGDA Field-Programmable Gate Arrays*. ACM Press, 2005.

[10] M. Ercegovac. Radix-16 evaluation of certain elementary functions. *IEEE Transactions on Computers*, C-22(6):561–566, June 1973.

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

[12] P. Farmwald. High-bandwidth evaluation of elementary functions. In *Fifth IEEE Symposium on Computer Arithmetic*, pages 139–142, 1981.

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

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

[15] D. Lee, A. Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Transactions on Computers*, 54(12):1520–1531, Dec. 2005.

[16] 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.

[17] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, 2 edition, 2006.

[18] G. Paul and M. W. Wilson. Should the elementary functions be incorporated into computer instruction sets? *ACM Transactions on Mathematical Software*, 2(2):132–142, June 1976.

[19] J. Stine and M. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21(2):167–177, 1999.

[20] P. T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989.

[21] P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378 – 400, Dec. 1990.

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

[23] W. F. Wong and E. Goto. Fast evaluation of the elementary functions in double precision. In *Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 349–358, 1994.

[24] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, Mar. 1994.

[25] W. F. Wong and E. Goto. Fast evaluation of the elementary functions in single precision. *IEEE Transactions on Computers*, 44(3):453–457, Mar. 1995.

[26] C. Wrathall and T. C. Chen. Convergence guarantee and improvements for a hardware exponential and logarithm evaluation scheme. In *Fourth IEEE Symposium on Computer Arithmetic*, pages 175–182, 1978.