Floating-point exponential functions for DSP-enabled FPGAs

Florent de Dinechin, Bogdan Pasca

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

***Abstract*—This article presents a generator of floating-point exponential operators targeting recent FPGAs with embedded memories and DSP blocks. A single-precision operator consumes just one DSP block, 18Kbits of dual-port memory, and 392 slices on Virtex-4. For larger precisions, a generic approach based on polynomial approximation is used and proves more resource-efficient than the literature. For instance a double-precision operator consumes 5 BlockRAM and 12 DSP48 blocks on Virtex-5, or 10 M9k and 22 18x18 multipliers on Stratix III. This approach is flexible and is demonstrated to scale up to quadruple-precision, while enabling frequencies close to the FPGA's nominal frequency. All the proposed architectures are last-bit accurate for all the floating-point range. They are available in the open-source FloPoCo framework.**

I. Introduction

The exponential function is, after the basic arithmetic operators, one of the next most useful building block for floating-point applications. On FPGAs, it has been used for scientific or financial Monte-Carlo simulations [1], for SPICE simulation [2], in phylogenetic tree reconstruction, in quantum chemistry simulations, and in the implementation of the power function [3] among others.

A. Previous works

Several publications have described exponential implementations. We list them here, and will discuss in more details the choices they made and their performance impact in Section IV.

Earlier works targetted single precision, first by adapting to FPGAs a software algorithm based on floating-point operations [4], then by using a more efficient fixed-point architecture [5]. This architecture was later improved [1], however the table-based method used there doesn't scale up to double-precision, as the size of the tables grows exponentially with the mantissa size.

As FPGAs are increasingly being used for double-precision floating-point, iterative architectures that scale better [6], [7], [8] were adapted for FPGAs [9]. The architecture in [9] was designed with 5-input LUTs in mind, but is poorly suited to DSP-enabled FPGAs, as IV-B will show. It was parameterized in precision, but to our knowledge was never pipelined. Another pipelined, but double-precision only implementation was proposed in [10], [11].

In [12], a CORDIC-based approach using several parallel CORDIC cores was proposed. It has a complex control including input and output FIFOs. Being radix-2 CORDIC, it computes one digit per iteration and thus has a very long latency. Moreover, it is based on a floating-point adder, whereas CORDIC is inherently a fixed-point computation, so there is probably room for improvement there.

From a user point of view, the current state of the art is probably the floating-point exponential function `ALTFP_EXP` provided with Altera Megawizard since 2008 [13]. This implementation exploits the DSP blocks, is parameterized in exponent and mantissa size, and is fully pipelined. Being included in the standard Quartus releases, it is widely available, although only for Altera targets.

Many other publications have addressed the computation of exponential function in ASIC, e.g. [6], [7], [14], [8]. However, it is difficult to evaluate the relevance of such works on FPGAs.

B. Contributions

In the present article, we propose yet another architecture for the floating-point evaluation of the exponential function, and its implementation in the open-source FloPoCo project [1]. Its main specificities are the following.

- The algorithm, based on the usual multiplicative range reduction followed by a polynomial approximation, was chosen with DSP blocks and embedded memories in mind, so it makes efficient use of these resources. For instance, the single-precision version now involves just one 17x17-bit multiplier and 18Kbits of dual-port memory, and runs at 375MHz on a Virtex-4, which is a large improvement in all respects over the state of the art [1].
- As we believe that floating-point on FPGA should exploit the flexibility of the target and therefore not be limited to IEEE single and double precision, the algorithm and implementation proposed here are fully parametrized in exponent and mantissa size. They scale to double-precision and beyond.
- The implementation is pipelined to a user-specified frequency. It is last-bit accurate for all supported mantissa sizes.
- The architectures are generated as synthesizable VHDL portable to any FPGA target. In addition, many target-specific optimizations are performed by the FloPoCo framework [15], [16], [17].

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

978-1-4244-8982-4/10/$26.00 ©2010 IEEE

- A novel variation of the KCM algorithm, multiplying a real constant by an integer, was developed.
- All this work is freely available as the `FPExp` operator of the FloPoCo project, since version 2.1.0. It comes with test vector generation [15]. In general, it should be immediately usable for application designers.

Section II gives an overview of the algorithm used, and Section III discusses some implementation choices. Section IV compares implementation results with the literature, and Section V concludes.

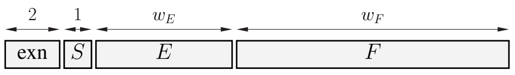
II. Algorithm and architecture

A. A flexible floating-point format for FPGAs

We use in this work the FPLibrary/FloPoCo floating-point format. This format is very similar to the IEEE-754 format and has two integer parameters:

- w_F is the number of bits of the mantissa, and defines the precision of the format;
- w_E is the number of bits of the exponent, and defines the range of the format.

A floating-point number X is represented as the $(3+w_E+w_F)$-bit vector depicted on Figure 1.



exn	Value of X
00	0
01	$(-1)^S \cdot 1.F \cdot 2^{E-E_0}$ where $E_0 = 2^{w_F-1} - 1$
10	$(-1)^S \cdot \infty$
11	NaN *(Not a Number)*

Fig. 1. The FloPoCo floating-point format.

B. Extremal values in the exponential

The exponential function is defined on the set of the reals. However, in this floating-point format, the smallest representable number is

$$X_{\min} = 2^{-E_0}$$

and the largest is

$$X_{\max} = (2 - 2^{-w_F}) \cdot 2^{2^{w_E}-1-E_0}.$$

The exponential should return zero for all input numbers smaller than $\log(X_{\min})$, and should return $+\infty$ for all input numbers larger than $\log(X_{\max})$. In single precision ($w_E = 8$, $w_F = 23$), for instance, the set of input numbers on which a computation will take place is $[-88.03, 88.72]$. In addition, as for small x we have $e^x \approx 1 + x + x^2/2$, the exponential will return 1 for all the input x smaller that 2^{-w_F-2}.

One consequence is that the testing of a floating-point exponential operator should concentrate on numbers between $X_{\min}$ and $X_{\max}$. In FloPoCo's testbench generator for `FPExp`, the exponent of the random inputs is restricted to $[-w_F - 3, w_E - 2]$.

C. Algorithm overview

The algorithm used is similar to what is typically used in software [18].

The main idea is to reduce X to an integer E and a fixed-point number Y such as

$$X \approx E \cdot \log 2 + Y \tag{1}$$

where $Y \in [-1/2, 1/2)$ – we will see below in II-D how to ensure this enclosure.

We may then use the identity

$$e^X \approx 2^E \cdot e^Y \tag{2}$$

so E is almost the exponent of the result, and e^Y almost the mantissa. Indeed, if $Y \in [-1/2, 1/2)$, we have $e^Y \in [0.6, 1.7]$, and a mantissa must be $1.F \in [1, 2)$. Thus the exponent and mantissa of the result may be obtained as

$$\begin{cases} R = 2^E \cdot e^Y & \text{if } e^Y \geq 1) \\ R = 2^{E-1} \cdot (2e^Y) & \text{if } e^Y < 1) \end{cases} \tag{3}$$

This test boils-down in testing the most significant bit of e^Y, and the multiplication by 2 is just a shift.

The architecture of this operator is given on Figure 2. This figure also explicits the alignment of the fixed-point data.

D. Range reduction

To implement equation (1), we have to implement an approximation of

$$E = \left\lfloor \frac{X}{\log 2} \right\rceil \tag{4}$$

where $\lfloor x \rceil$ denotes the rounding of x to the nearest integer. Then,

$$Y = X - E \times \log 2. \tag{5}$$

If computed infinitely accurately, this would ensure $Y \in [-\frac{\log 2}{2}, \frac{\log 2}{2}]$. On one hand, this is not ideal from an architectural point of view, as Y will later be input to a table and $\frac{\log 2}{2}$ is not a power of two (as $\log 2 \approx 0.34$, the next power of 2 is 1/2, so only 69% of the table would be used). On the other hand, implementing (4) and (5) accurately enough would be expensive. A solution to both problems is therefore a relaxed implementation of (4) that will save on the computation of (4) and (5) while ensuring $Y \in [-1/2, 1/2)$. The idea is that the computation of E can be grossly approximate, as long as (5) is accurately implemented. The normalization process (3) will take care of the cases where E was not directly computed as the exact result exponent.

As (4) and (5) are inherently fixed-point computations, the first task is to build a fixed-point representation X_{fix} of the input X. The most significant bit (MSB) of this representation is provided by the condition $X > \log(X_{\max}) \Rightarrow \exp(X) = +\infty$, from which we deduce $X > 2^{w_E+1} \Rightarrow \exp(X) = +\infty$. The MSB of X_{fix} should therefore have weight w_E. The least significant bit is provided by the condition $X < 2^{-w_F-2} \Rightarrow \exp(x) = 1$, which defines a LSB of weight $-w_F - 2$. Actually, we will improve this accuracy to $-w_F - g$ with $g = 3$

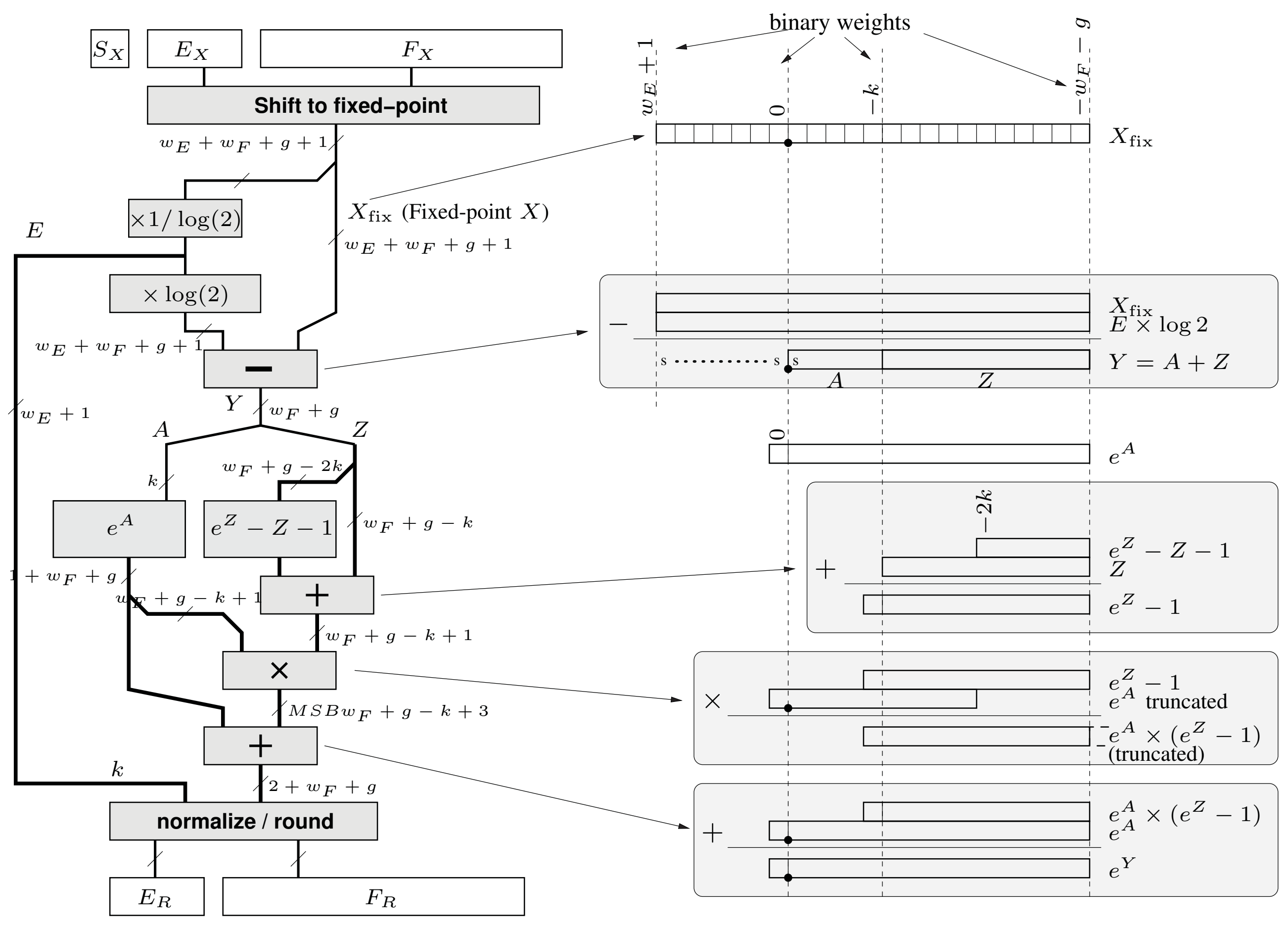


Fig. 2. Architecture and fixed-point data alignment

(see below in III-B) to allow for rounding error accumulation in these g guard bits.

Thus the *shift to fixed point* box on Figure 2 shifts the mantissa by the value of the exponent. More specifically, if the exponent is positive, it shifts to the left by up to w_E positions (more means overflow). If the exponent is negative, it shifts to the right by up to $w_F + g$ positions. This box also generates out-of-range signals (not shown on the figure).

Let us now turn to the relaxed computation of E. E is an integer. Since it is almost the result's exponent (of size w_E), its size in bits will be $w_E + 1$, including one sign bit, the $+1$ preventing overflow in the second case of (3).

An absolute error of ϵ in the computation of $\frac{X_{\text{fix}}}{\log 2}$ before rounding means that E may be misrounded (off by 1 with respect to the ideal $\left\lfloor \frac{X}{\log 2} \right\rceil$) when X is within ϵ of the middle between two multiples of $\log 2$. As a consequence the bound on Y will be $Y \in [-\frac{\log 2}{2} - \epsilon \log 2, \frac{\log 2}{2} + \epsilon \log 2]$. As we want to ensure $Y \in (-1/2, 1/2)$, we deduce the bound $\epsilon < \frac{1-\log 2}{2\log 2} \approx 0.22$.

One way to ensure this bound is to implement (4) as

$$E = \left\lfloor \lfloor X_{\text{fix}} \rfloor_{-3} \times \left\lfloor \frac{1}{\log 2} \right\rceil_{-3} \right\rceil \tag{6}$$

where $\lfloor x \rfloor_{-3}$ and $\lfloor x \rceil_{-3}$ respectively mean a truncation and a rounding such that the LSB of the result is 2^{-3}. Thus the product is computed with an absolute error $\epsilon < (1 + 0.5) \times 2^{-3} = 0.1875$ with respect to $\frac{X_{\text{fix}}}{\log 2}$ (Section III-A below introduces a small improvement on this computation that leads to the same error). This leaves margin for the error of X_{fix} with respect to X, and for the errors on the computation of (5), the sum of which will be smaller than 2^{-w_F} as the sequel will show.

Then, (5) may be implemented as

$$Y = X_{\text{fix}} - E \times \log 2. \tag{7}$$

This fixed-point subtraction cancels the integer part and the first bit of the fractional part.

In this work, we have also considered reducing to $Y \in [0, 1)$ instead of $Y \in [-1/2, 1/2)$. It turns out that guaranteeing this enclosure, especially $Y \geq 0$, is more expensive.

E. Computation of e^Y

Let us now turn to the computation of e^Y. We use a second range reduction, splitting Y as

$$Y = A + Z \tag{8}$$

where A consists of the k most significant bits of Y, and Z consists of the w_F+g-k least significant bits. Then we have

$$e^Y = e^{A+Z} = e^A \cdot e^Z. \quad (9)$$

Here e^A will be tabulated in a table indexed by A, and Z is small enough to enable us to use the Taylor formula

$$e^Z \approx 1 + Z + Z^2/2 + ... \quad (10)$$

This formula has the advantage that the three first coefficients are powers of two, therefore the corresponding multiplications can be mere shifts. Actually we define

$$f(Z) = e^Z - Z - 1 \quad (11)$$

From $0 \leq Z < 2^{-k}$ and $e^Z - Z - 1 \approx Z^2/2 + ...$, we know that the MSB of $f(Z)$ has weight $-2k-1$. As $f(Z)$ will be added to Z, its LSB should have the same weight $-w_F - g$. The useful size of $f(Z)$ is therefore $w_F + g - 2k$. As a consequence, we do not need to compute it out of all the bits of Z. Truncating Z to its $w_F + g - 2k$ MSBs will entail an error of roughly the same weight as the error entailed by the fixed-point format of $f(Z)$.

Out of Z and $f(Z)$, we compute $e^Z - 1 = f(Z) + Z$. This addition may overflow, so the result is on $w_F + g - k + 1$ bits, one more bit than Z.

If $1 + w_F + g < 17$, the final multiplication $e^Y = e^A \cdot e^Z$ may be computed directly as a single DSP block. For larger precisions, the cost of this multiplication is reduced by implementing it as

$$\begin{aligned} & e^A \cdot (1 + Z + f(Z)) \\ &= e^A + e^A \cdot (Z + f(Z)) \end{aligned} \quad (12)$$

Again, the two addends have LSB weight $-w_F - g$. Again, the multiplier inputs need not be more accurate than their output, so we truncate e^A to its LSB $w_F + g - k + 1$ bits.

As we need to truncate the result of this multiplier, we may as well use, for large precisions, truncated multipliers [19], to save DSP and latency.

A final normalization step possibly shifts left the mantissa by one bit, then performs the final rounding. The rounding consists in possibly adding one bit, then truncating. The IEEE-754 format has the nice property that we may use an adder of size $w_E + w_F + 1$ to add the rounding bit to the concatenated exponent and mantissa: carry propagation from mantissa to exponent will handle the possible exponent change due to rounding up.

III. Implementation issues

This computation involves several approximation and rounding errors. The purpose of this section is to guarantee faithful rounding, *ie.* an error of less than one *unit in the last place* (ulp) of the result. Here the ulp has the value 2^{-w_F}, the weight of the last bit of the mantissa $1.F$ of the result.

A. Constant multiplications

As both constant multiplications (by $1/\log 2$ and $\log 2$) multiply a large constant by a small input, it is natural to use the KCM algorithm [20]. For the larger multiplication by the real value $\log 2$, we actually use a variation that is original to our knowledge and that we briefly present now.

Assume we need to multiply a n-bit integer E by a *real* constant K (here $K = \log 2$), and we want an m-bit result with $m \geq n$. The usual techique is to first round the constant to precision m, then use a fixed-point multiplier (that returns an $n+m$-bit result), then again round the result to m bits. We have two rounding to m bits that each introduce one half-ulp of error on the result, so the final result is accurate to 1 ulp. This accuracy can be improved by rounding the constant to more than m bits. On the implementation side, the multiplication by a constant can use the KCM algorithm [20], and the final rounding costs one addition (truncation is also possible, but then the total error is above 1 ulp). The following technique attains the same accuracy, saving hardware in the KCM, and without needing this final adder.

Let α be the LUT input size of the target FPGA. The input E is split into chunks of size α:

$$E = \sum_{i=0}^{p} 2^{i\alpha} E_i$$

therefore

$$KE = \sum_{i=0}^{p} 2^{i\alpha} KE_i \ .$$

We tabulate in LUTs each product $2^{i\alpha}KE_i$ on just the required precision, so that its LSB has value $2^{-\gamma}u$ where u is the ulp of the result, and γ is again a number of guard bits. Each table may hold the correctly rounded value of the product of E_i by the *real* value $\log 2$ to this precision, so entails an error of $2^{-\gamma-1}$ ulp. In the first table, we actually store $KE_0 + u/2$, so that the truncation of the sum will correspond to a rounding of the product. Finally, the value of γ is chosen to ensure 1-ulp accuracy.

This operator is implemented generically as the `FixRealKCM` operator in FloPoCo. Back to the exponential, as $\alpha \in \{4..6\}$ for current FPGAs, and practical values of E are smaller than 15, the value $\gamma = 2$ is usually enough to ensure that this multiplier returns a faithful multiplication by $\log 2$. For the multiplier by $1/\log 2$ we manually set $\gamma = 3$ to mimick (6).

B. Overall error analysis

In the following, all the errors will be expressed in terms of unit in the last place of Y, which has the value 2^{-w_F-g}. Thus errors expressed this way can be made as small as required by increasing g.

First, note that the argument reduction is not exact. As already stated, numerical errors in the computation (6) of E mostly impact the range of Y. Concerning the computation of Y (1), there are two exclusive cases:

- If X is large (its exponent is larger than -2), its mantissa is shifted without loss of information, then the computation of $E \times \log 2$ introduces at most one ulp of error in Y as seen in III-A.
- Or, X is small, its mantissa is shifted right beyond the ulp, so its LSBs are lost, which also entails one error of one ulp in Y. However, in this case $E = 0$, so the computation of $E \times \log 2$ is exact.

In both cases we may thus have an error of at most one ulp on Y. Let us now see how it propagates to e^Y.

e^A is tabulated rounded to the nearest, thus with an error of 1/2 ulp.

$e^Z - Z - 1$ is either tabulated (1/2 ulp) or evaluated through polynomial approximation (1 ulp). As the higher order bits of Z are used, the error on Y (which is the error on Z) is scaled down and becomes negligible.

Then $e^Y - 1$ adds the error on Z and the error on $e^Z - Z - 1$, and thus holds an error of 1.5 or 2 ulps.

The error on the other input to the multiplier (e^A truncated) is of one ulp. The product adds these error as $(a+\epsilon)\times(b+\epsilon') = ab + b\epsilon + a\epsilon' + \epsilon\epsilon'$. Here is another subtlety. This formula shows that the error on $e^Z - Z - 1$ is scaled by the value of e^A. Fortunately, the worst case error will occur for $e^A < 1$, since in this case the result will be shifted left by one bit. In the case $e^A > 1$ the error on $e^Z - Z - 1$ may be scaled up (by up to 1.6) but we will have in this case the extra bit of precision needed for the other case, so it doesn't matter.

Truncating the multiplier result would yields another error of one ulp, however we may instead round it (1/2 ulp only) at very little cost by adding its round bit to the right of e^A, so the addition of e^A will also compute the rounding of the product.

Finally the product holds an error of 3 or 3.5 ulps.

Adding the error on e^A, we deduce that the error on e^Y may be up to 3.5 ulp in the dual table case, and 4 ulp in the polynomial case.

If $e^Y < 1$ the final 1-bit shift will multiply this error by 2, so we need $g = 3$ guard bits.

Previous works need more guard bits for the same final accuracy (5 guard bits in [5], 8 in [12] for instance), hence a wider datapath. This improvement in the present work is partly due to a finer error analysis, partly to a refined implementation, in particular of the multiplication by $\log 2$. It is proportionnally more important for lower precisions.

However, our implementation also allows increasing the parameter g beyond this minimal value of 3. More guard bits will mean a larger percentage of correctly rounded results. This possibility is also useful when building larger faithful operator based on the exponential, for instance the power function [3] (under development in FloPoCo).

C. *The case study of single precision*

Setting $w_F = 23$ and $g = 3$ in the previous architecture, it turns out that $k = 9$ allows for a highly efficient architecture on recent FPGAs.

Firstly, we need altogether $2^9 \times 27$ bits of RAM for e^A and $2^9 \times 9$ bits for $e^Z - Z - 1$. We can group both tables in a single $2^9 \times 36$ table with dual-port access. This perfectly matches one Xilinx BlockRAM, or two Altera M9K.

Secondly, the multiplication is now 18x18 bits, unsigned. This perfectly matches the DSP blocks of Altera chips. On Xilinx chips up to Virtex-4, the multipliers are able of 17x17 unsigned, so the cost is one DSP block plus two 18-bit additions. On Virtex-5 the DSP block is able of 17x24 unsigned, so we only need one addition. One more trick allows us to hide the latency of this addition. We choose to input e^A on 17 bits only instead of 18. To keep the same error bound of one ulp, we now need to round it to 17 bits. This rounding requires an addition (so there is no saving compared to extending the multiplier input to 18 bit), but this addition is now before the multiplier, in parallel to the addition of Z to $e^Z - Z - 1$.

D. *Polynomial approximation for large precisions*

For larger values of w_F, a generic polynomial evaluator [21] is used as a black box. It inputs a function of $[0,1] \to [0,1]$ (here $e^{2^{-k}x} - 2^{-k}x - 1$) with its input and output precisions (given on Figure 2) and a degree, and implements a piecewise polynomial approximation. The input interval is decomposed into smaller intervals, and the number of such intervals is computed so that the generated architecture returns a faithfully rounded result. The architectures are optimized for the target FPGA (currently Xilinx Virtex-4, Virtex-5 and Virtex-6, and Altera Stratix II to IV), making efficient use of the DSP blocks to attain high frequencies.

One advantage of this approach is that it is based on DSPs and memory blocks. Another one is its genericity, as future improvements to the polynomial evaluator will immediately benefit to the exponential. This includes the adaptation of the polynomial evaluator to newer FPGAs, but also performance improvements. For instance, we are improving the polynomial evaluator so that it can make use of truncated multipliers to reduce the DSP count, and this will improve `FPExp`.

More specifically, the function evaluated here is easy to approximate by a low-degree polynomial approximations. It turns out that degree 2 is enough for precision up to double-extended precision.

E. *Parameter selection*

We now have two parameters to set: k, that fixes the input to the e^A table, and the degree d of the polynomial, that fixes the trade-off between area of the coefficient table and DSP count/latency. We have varied these parameters to obtain the best trade-offs, that is a an architecture well balanced between DSP and memory consumption, with memories as full as possible and multipliers used as fully as possible. For instance, for double precision, on all targets the best choice is $k = 9$ and a degree-2 approximation on 512 intervals. The FPExp operator provides a good default choice of these parameters, and an expert mode allows the user to set them manually for a different trade-off.

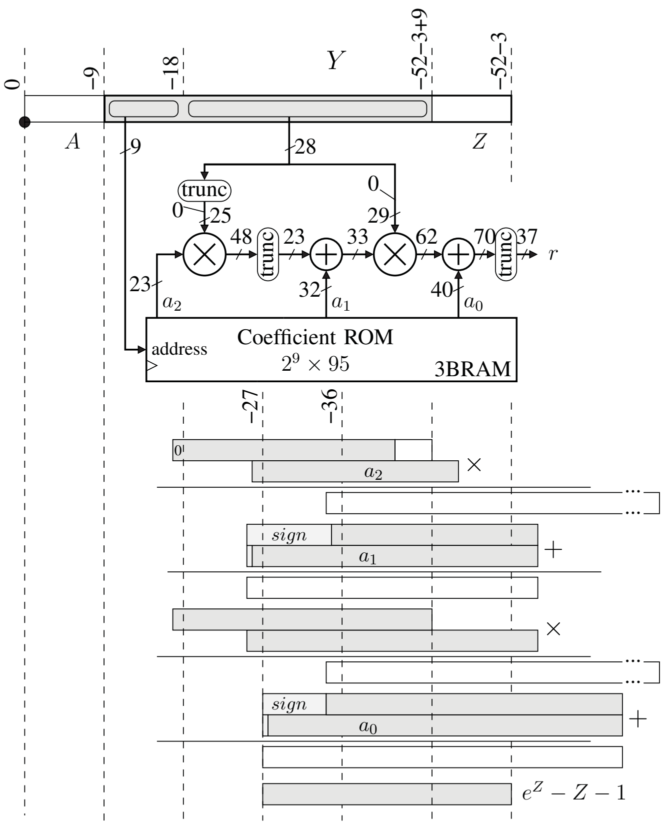


Fig. 3. The architecture evaluating $e^Z - Z - 1$ for Virtex-5/Virtex-6

Figure 3 details one instance of this architecture for Virtex-5.

F. Pipeline tuning

This work was also an in-depth case study for the pipeline generator framework of FloPoCo [15]. This framework enables easy composition and synchronization of components that pipeline themselves to run at a user-provided frequency. While details are out of scope of this article, this explains how we obtain results for different frequencies. For illustration, Figure 4 shows one example of the obtained component hierarchy, with the pipeline information. It also details the sizes of the various multipliers on this example.

```
|---Entity LeftShifter_53_by_max_65:
|       Pipeline depth = 1
|---Entity IntAdder_67:
|       Pipeline depth = 2
|    |---Entity KCMTable_6_11818_unsigned:
|    |       Not pipelined
|    |---Entity KCMTable_6_11818_signed:
|    |       Not pipelined
|    |    |---Entity IntAdder_22:
|    |    |       Pipeline depth = 1
|    |---Entity IntCompressorTree_22_3:
|    |       Pipeline depth = 2
|---Entity IntIntKCM_14_11818_signed:
|       Pipeline depth = 2
|    |---Entity KCMTable_6_51145234580810622639_unsigned:
|    |       Not pipelined
|    |---Entity KCMTable_6_51145234580810622639_signed:
|    |       Not pipelined
|    |    |---Entity IntAdder_72:
|    |    |       Pipeline depth = 2
|    |---Entity IntCompressorTree_72_2:
|    |       Pipeline depth = 2
|---Entity IntIntKCM_12_51145234580810622639_signed:
|       Pipeline depth = 3
|---Entity IntAdder_67:
|       Pipeline depth = 2
|---Entity firstExpTable_11_56:
|       Not pipelined
|    |---Entity TableGenerator_7_89:
|    |       Not pipelined
|    |    |---Entity SignedIntMultiplier_25_23:
|    |    |       Pipeline depth = 2
|    |    |---Entity IntAdder_31:
|    |    |       Pipeline depth = 1
|    |    |    |    |---Entity IntAdder_60:
|    |    |    |    |       Pipeline depth = 1
|    |    |    |---Entity IntCompressorTree_60_2:
|    |    |    |       Pipeline depth = 1
|    |    |---Entity SignedIntMultiplier_27_31:
|    |    |       Pipeline depth = 4
|    |    |---Entity IntAdder_64:
|    |    |       Pipeline depth = 2
|    |---Entity PolynomialEvaluator_d2:
|    |       Pipeline depth = 10
|    |---Entity IntAdder_35:
|    |       Pipeline depth = 1
|---Entity FunctionEvaluator:
|       Pipeline depth = 14
|    |    |---Entity IntAdder_75:
|    |    |       Pipeline depth = 2
|    |---Entity IntCompressorTree_75_2:
|    |       Pipeline depth = 2
|---Entity IntMultiplier_45_45:
|       Pipeline depth = 6
|---Entity IntAdder_57:
|       Pipeline depth = 1
|---Entity IntAdder_65:
|       Pipeline depth = 2
Entity FPExp_11_52:
   Pipeline depth = 38
```

Fig. 4. Component hierarchy for double-precision exponential on Virtex-6

IV. Results

A. Synthesis results

Table I provides synthesis results for several precisions and several FPGA targets, and compares with results from previous papers. Our approach is clearly the most efficient of the literature for all the precisions. It combines very high frequency (close to the nominal DSP block frequency), the lowest DSP and memory consumption, portability to both Xilinx and Altera targets, last-bit accuracy, flexibility in precision, and also flexibility in terms of latency versus frequency.

Note that the synthesis on Stratix III reports 2 DSP blocks for single precision. One is actually unused. The coarse-grain DSP block structure of Altera chips since Stratix III prevent using the 18×18-bit multipliers completely independently.

Of special interest is the last line of this table, which shows that even a quadruple-precision exponential function will consume only one tenth of the resources of a high-end FPGA while still running at a very high frequency.

B. Comparison with other works

In [9], a double-precision combinatorial operator consumes, on VirtexII, 2045 slices for a delay of 229 ns. To our knowledge, it was never pipelined, but we estimate that a high-frequency pipelined would require a doubling of the area and

TABLE I

Synthesis results for various instances of the floating-point exponential operator.

Precision	FPGA	Tool	Performance		Resource Usage				
			f(MHz)	Latency	Logic Usage			DSPs	Memory
					(A)LUTs	Reg.	Slice		
(8,23)	StratixIII	Altera MegaWizard	274	17	527	900		**19** 18-bit elem.	0
		ours	315	10	515	476		**2** 18-bit elem.	**3** M9K
			187	3	670	187			**1** M9K
	VirtexII 1000	[9]	1/123ns	0			728	0	0
	Virtex-4	[9] (see Note 1)	21	0			545	7	0
		ours $f_T = 330$	334	16			399	**1** DSP48	**1** BRAM
		ours $f_T = 200$	261	8			365		
	Virtex-5	*ours* $f_T = 380$	384	17	561	531		**1** DSP48E	**1** BRAM
		ours $f_T = 200$	360	9	545	231			
	Virtex-6 (see Note 2)	*ours* $f_T = 600$	493	23	603	602		**1** DSP48E1	**1** BRAM
		ours $f_T = 50$	179	5	452	189			
(10,40)	Virtex-5	ours (k=5,d=2)	310	30	1377	1141	-	**10** DSP48E	**4** BRAM
	Virtex-6	ours (k=5,d=2)	488	32	1469	1344	-	**10** DSP48E1	**3** BRAM
(11,52)	StratixIII	Altera MegaWizard	213	25	2941	1476	-	**58** 18-bit elem.	0
		ours	327	29	1307	3757	-	**22** 18-bit elem.	**10** M9K
			256	15	1437	1984	-		
	VirtexII 1000	[9]	1/229ns	0			2045	0	0
	Virtex-4	[10]	?	0	1293	105		**71** DSP48	**6** BRAM
		[11]	200	30	13614	19704		0	**29** BRAM
		[12] (CORDIC)	5.25 cycles@100Mhz	>			23455	**36** DSP48	
		ours	319	38	2249	1964	1393	**17** DSP48	**5** BRAM
			178	24	2128	1361	1154		
			97	14	2034	926	1096		
	Virtex-5	*ours*	310	35	1867	1456	-	**12** DSP48E	**5** BRAM
			204	18	1604	1018	-		
			119	12	1601	806	-		
	Virtex-6	*ours*	488	38	1928	1791	-	**12** DSP48E1	**5** BRAM
			221	22	1642	1184	-		
			125	10	1547	629	-		
(15,64)	Virtex-6	ours (k=11, d=2)	486	41	2894	2539	-	**20** DSP48E1	**11** BRAM
(15,112)	Virtex-6	ours (k=14, d=3)	395	69	8071	7725	-	**71** DSP48E1	**123** BRAM

We used QuartusII v9.0 for StratixIII EPSL50F484C2 and ISE 11.5 for Virtex-4 XC4VFX100-12-ff1152, Virtex-5 XC5VFX100T-3-ff1738 and Virtex-6 XC6VHX380T-3-ff1923. f_T is the target frequency provided to FloPoCo.
Note 1: we used the (unpipelined) FPExp operator in FloPoCo until 2.0.0, which is based on the code from [9].
Note 2: the current timing model of Virtex-6 in FloPoCo is very preliminary.

roughly 40 cycles.

In addition, this architecture was based on tables inputting α bits and rectangular multipliers where one dimension was also α (an integer parameter) and the other dimension varied from α to the mantissa size. This was a good design choice for LUT-based FPGAs, but it poorly matches the capabilities of the DSP blocks and embedded memories of modern FPGAs. For a short latency, and to use the DSP blocks optimally, one should choose $\alpha = 17$, but then the tables would be much too large (2^{17} entries). Or, one should chose $\alpha \approx 10$, but then the DSPs would be underutilized.

As Altera Megawizard produces readable source files, we analysed the algorithm used for double precision. The range reduction is the usual one, and the architecture diverges only for the computation of e^Y. Altera's architecture is based on a decomposition of the input as $Y = Y_0 + Y_1 + Y_2 + Y_L$ where Y_0 consists of the 9 leading bits, Y_1 and Y_2 consist of the two following 9-bit chunks, and Y_L consists of the remaining lower bits. The exponential is computed as $e^Y = (e^{y_0} \times e^{y_1}) \times (e^{y_2} e^{y_L})$, where the three first terms are simply read from tables with 2^9 entries, and e^{y_L} is approximated as the Taylor polynomial $e^{Y_L} \approx 1 + Y_L$. This is very similar to the method proposed by Wielgosz et al [10], [11], and both were probably designed independently. However the Altera implementation is generic in precision.

This approach has a potential of lower latency, as the multipliers are organized in tree, and not in sequence as in our proposal. Its drawback is that it doesn't exploit the structure of the numbers. Indeed, the three multiplications are of size roughly 60×60 bits. However, e^{y_1}, e^{y_2}, and e^{y_L} are all of the form $1 + \epsilon$, so at the bit level, we have a lot of predictible multiplications by 0, for which the hardware could be saved. Table I illustrates this waste of resource compared to our approach.

We also remark in Table I that the Altera ALTFP_EXP operators do not use 9Kbit embedded memories, although this design would be a perfect match for them (it should consume $(61 + 51 + 42)/18 = 9$ of them, with a corresponding huge reduction in logic resources).

A final remark is that the two references by Wielgosz et al. [10], [11] seem to use the same architecture, however the first one reports results using DSP blocks, while the second one replaces all the DSPs with logic. This actually makes sense, since in this case the parts of the large multipliers that multiply by zero will indeed be optimized out by the synthesizer.

C. Comparison with microprocessors

This table allows us to compare the theoretical peak performance, in terms of floating-point exponentials, of a large FPGA and a high-end processor. These numbers, of course, should be taken for what they are, as they ignore the critical issue of data movements [11].

The largest Virtex-6 FPGA (XC6VSX475T) could accomodate 168 double-precision exponential cores running above 400 MHz, thus providing a theoretical peak performance over 60 giga double-precision exponentials per second (GDPexp/s).

For a fair comparison, we have to compare to the highest performance software implementation currently available, one which was tuned with comparable effort. To our knowledge, it is the Intel Vector Math Library (VML), which can achieve a peak of 6 cycles/DPExp on Itanium-2 or Core i7. On an 8-core processor running at 3GHz, we obtain a peak performance of 4 GFPExp/s, with a speed-up of 15 in favor of the FPGA. On single precision, the numbers are in excess of 400GSPExp/s for the FPGA while the performance of VML is only improved to 6GSPExp/s. The FPGA speed-up is now above 60.

V. Conclusion and future work

We have presented a state-of-the-art floating-point exponential operator generator. It produces last-bit accurate architectures for a wide range of FPGA targets, for a wide range of precisions up to IEEE-754-2008 quadruple precision, and for a wide range of latency/frequency trade-offs. It is designed to make good use of the DSP blocks and embedded memories of high-end FPGAs, and outperforms previous works in performance and resources consumption.

Hopefully, other elementary function of the same quality will join the exponential, forming a complete open-source mathematical library for FPGAs. To this purpose, the case study of the exponential has already lead to improvements in the pipeline framework and the generic polynomial approximator. These will be improved further. This work also suggests that the FloPoCo framework could be enhanced by attaching an optional fixed-point semantics to the signals, which is being investigated.

Acknowledgements

This work was partly supported by the ANR EVAFlo project and Stone Ridge Technology. Some of the architectures have been hardware-tested thanks to Adacsys technology. Polynomial approximation tools come from the Sollya team: Mioara Joldes, Sylvain Chevillard, Christoph Lauter and Nicolas Brisebarre.

References

[1] P. Echeverría, D. Thomas, M. López-Vallejo, and W. Luk, “An FPGA run-time parameterisable log-normal random number generator,” in *Reconfigurable computing: architectures, tools and applications, LNCS 4943*. Springer-Verlag, 2008, pp. 221–232.

[2] N. Kapre and A. DeHon, “Accelerating spice model-evaluation using FPGAs,” *Field-Programmable Custom Computing Machines*, pp. 37–44, 2009.

[3] P. Echeverría and M. López-Vallejo, “An FPGA implementation of the powering function with single precision floating-point arithmetic,” in *Real Numbers and Computers*, 2008.

[4] C. Doss and R. L. Riley, Jr., “FPGA-based implementation of a robust IEEE-754 exponential unit,” in *Field-Programmable Custom Computing Machines*. IEEE, 2004, pp. 229–238.

[5] J. Detrey and F. de Dinechin, “A parameterized floating-point exponential function for FPGAs,” in *Field-Programmable Technology*. IEEE, 2005.

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

[7] 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.

[8] J. A. Pineiro, M. D. Ercegovac, and J. D. Bruguera, “Algorithm and architecture for logarithm, exponential, and powering computation,” *IEEE Transactions on Computers*, vol. 53, no. 9, pp. 1085–1096, Sep. 2004.

[9] 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.

[10] M. Wielgosz, E. Jamro, and K. Wiatr, “Highly efficient structure of 64-bit exponential function implemented in FPGAs,” in *Reconfigurable computing: architectures, tools and applications, LNCS 4943*. Springer-Verlag, 2008, pp. 274–279.

[11] ——, “Accelerating calculations on the RASC platform: A case study of the exponential function,” in *Reconfigurable computing: architectures, tools and applications, LNCS 5453*. Springer-Verlag, 2009, pp. 306–311.

[12] R. Pottathuparambil and R. Sass, “A parallel/vectorized double-precision exponential core to accelerate computational science applications,” in *Field Programmable Gate Arrays*. ACM, 2009, pp. 285–285.

[13] M. Langhammer, “Foundation for FPGA acceleration,” in *Fourth Annual Reconfigurable Systems Summer Institute*, 2008.

[14] A. Vázquez and E. Antelo, “Implementation of the exponential function in a floating-point unit,” *Journal of VLSI Signal Processing*, vol. 33, no. 1-2, pp. 125–145, Jan. 2003.

[15] F. de Dinechin, C. Klein, and B. Pasca, “Generating high-performance custom floating-point pipelines,” in *Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 59–64.

[16] F. de Dinechin, H. D. Nguyen, and B. Pasca, “Pipelined FPGA adders,” in *Field-Programmable Logic and Applications*, 2010, pp. 422–427.

[17] 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.

[18] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas, “The libm library and floating-point arithmetic for HP-UX on Itanium,” Hewlett-Packard company, Tech. Rep., 2001.

[19] M. Schulte and E. Swartzlander, “Truncated multiplication with correction constant,” in *Workshop on VLSI Signal Processing*, 1993, pp. 388–396.

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

[21] 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.