A parameterized floating-point exponential function for FPGAs

Jérémie Detrey   Florent de Dinechin
Laboratoire de l'Informatique du Parallélisme
École Normale Supérieure de Lyon
46 allée d'Italie, F-69364 Lyon, France
{ Jeremie.Detrey, Florent.de.Dinechin }@ens-lyon.fr

Abstract

A parameterized floating-point exponential operator is presented. In single-precision, it uses a small fraction of the FPGA's resources and has a smaller latency than its software equivalent on a high-end processor, and ten times the throughput in pipelined version. Previous work had shown that FPGAs could use massive parallelism to balance the poor performance of their basic floating-point operators compared to the equivalent in processors. As this work shows, when evaluating an elementary function, the flexibility of FPGAs provides much better performance than the processor witout even resorting to parallelism.

1. Introduction

A recent trend in FPGA computing is the increasing use of floating-point. Many libraries of floating-point operators for FPGAs now exist [19, 9, 1, 12, 6], typically offering the basic operators $+$, $-$, $\times$, $/$ and $\sqrt{\ }$. Published applications include matrix operations, convolutions and filtering. As FPGA floating-point is typically clocked 10 times slower than the equivalent in contemporary processors, only massive parallelism (helped by the fact that the precision can match closely the application's requirements) allows these applications to be competitive to software equivalent [14, 5, 11].

More complex floating-point computations on FPGAs will require good implementations of elementary functions such as logarithm, exponential, trigonometric, etc. These are the next useful building blocks after the basic operators. This paper describes an exponential function, part of a first attempt to a library of floating-point elementary functions for FPGAs.

Elementary functions are available for virtually all computer systems. There is currently a very large consensus that they should be implemented in software [18]. Even processors offering machine instructions for such functions (mainly the x86/x87 family) implement them as microcode. On such systems, it is easy to design faster software implementations: Software can use large tables which wouldn't be economical in hardware [20]. Therefore, no recent instruction set provides instructions for elementary functions.

Implementing floating-point elementary functions on FPGAs is a very different problem. The flexibility of the FPGA paradigm allows to use specific algorithms which turn out to be much more efficient than a processor-based implementation. We show in this paper that a single-precision function consuming a small fraction of FPGA resources has a latency equivalent to that of the same function in a 3GHz PC, while being fully pipelinable to run at 100MHz. In other words, where the basic floating-point operator ($+$, $-$, $\times$, $/$) is typically 10 times slower on an FPGA than its PC equivalent, an elementary function will be faster for precisions up to single precision.

Writing a *parameterized* elementary func-

tion is a completely new challenge: to exploit this flexibility, one should not use the same algorithms as used for implementing elementary functions in computer systems [20, 16, 15]. This paper describes an approach to this challenge, which builds upon previous work dedicated to fixed-point approximations (see [8] and references therein).

The authors are aware of only two previous works on floating-point elementary functions for FPGAs, studying the sine function [17] and studying the exponential function [10]. Both are very close to a software implementation. As they don't exploit the flexibility of FPGAs, they are much less efficient than our approach, as section 5 will show.

Notations

The input and output of our exponential operator will be $(3 + w_E + w_F)$-bit floating-point numbers encoded in the freely available FPLibrary format [6] as follows:

- F_X: The w_F least significant bits represent the fractional part of the mantissa $M_X = 1.F_X$.

- E_X: The following w_E-bit word is the exponent, biased by $E_0 = 2^{w_E-1} - 1$.

- S_X: The next bit is the sign of X.

- exn_X: The two most significant bits of X are internal flags used to deal more easily with exceptional cases, as shown in Table 1.

exn_X	X
00	0
01	$(-1)^{S_X} \cdot 1.F_X \cdot 2^{E_X - E_0}$
10	$(-1)^{S_X} \cdot \infty$
11	NaN

Table 1. Value of X according to its exception flags exn_X**.**

2. A floating-point exponential

2.1. Special cases

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

$$X_{\max} = (1 + \frac{2^{w_F} - 1}{2^{w_F}}) \cdot 2^{2_E^w - 1 - E_0}.$$

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 $[-88.03, 88.72]$. The remainder of this section only describes the computation on this interval.

2.2. A first algorithm

The straightforward idea to compute the exponential of X is to use the identity

$$e^X = 2^{X/\log(2)} \quad . \tag{1}$$

Therefore, first compute $X/\log(2)$ as a fixed-point value $Y = Y_{\text{int}}.Y_{\text{frac}}$. The integer part of Y is then the exponent of the exponential of X, and to get the fraction of e^X one needs to compute the function 2^x with the fractional part of Y as input.

This approach poses several problems:

- To be sure of the exponent, one has to compute Y with very good accuracy: A quick error analysis shows that one needs to use a value of $\frac{1}{\log(2)}$ on more than $w_E + w_F$ bits, which in practice means a very large multiplier for the computation of $X \cdot \frac{1}{\log(2)}$.

- The accurate computation of $2^{Y_{\text{frac}}}$ will be very expensive as well. Using a table-based method, it needs a table with at least w_F bits of inputs.

The second problem can be solved using a second range reduction, splitting Y_{frac} into two subwords

$$Y_{\text{frac}} = Y_1 + Y_2 \qquad (2)$$

where Y_1 holds the p most significant bits of Y. One may then use the identity

$$2^{Y_1 + Y_2} = 2^{Y_1} \cdot 2^{Y_2} \qquad (3)$$

Again we have a multiplier of size at least $w_F \times (w_F - p)$.

2.3. Improved algorithm

A slightly more complicated algorithm, closer to what is typically used in software [13], solves the previous problems. The main idea is to reduce X to an integer k and a fixed-point number Y such as

$$X \approx k \cdot \log(2) + Y \qquad (4)$$

and then to use the identity

$$e^X \approx 2^k \cdot e^Y \quad . \qquad (5)$$

The reduction to (k, Y) is implemented by first computing $k \approx X \cdot \frac{1}{\log(2)}$, then computing Y as per Eq. (4). The computation of e^Y may use a second range reduction as previously, splitting Y as

$$Y = Y_1 + Y_2 \qquad (6)$$

where Y_1 holds the p most significant bits of Y, then computing

$$e^{Y_1 + Y_2} = e^{Y_1} \cdot e^{Y_2} \qquad (7)$$

where e^{Y_1} will be tabulated.

This looks similar to using (1), however it has two main advantages: First, the computation of k can be approximated, as long as the computation of Y compensates it in such a way that Eq. (4) is accurately implemented. As k is a small integer, this in practice replaces a large multiplication with two much smaller multiplications, one to compute k, the second to compute $k \cdot \log(2)$.

This approximation, however, implies that the final exponent may be $k \pm 1$: the result $2^k \cdot e^Y$ will require a normalization.

Second, computing e^{Y_2} is simpler than computing 2^{Y_2}, because of the Taylor formula

$$e^{Y_2} \approx 1 + Y_2 + T(Y_2) \qquad (8)$$

where $T(Y_2) \approx e^{Y_2} - 1 - Y_2$ will be stored in a second table.

This not only reduces a table-based approximation to a much smaller one (as $Y_2 < 2^{-p-1}$ as will be seen in Section 4, it requires about $w_F - 2p$ input bits instead of $w_F - p$ bits), it also offers the opportunity to save p lines of the final large multiplier by implementing it as

$$\begin{aligned} &e^{Y_1} \cdot (1 + Y_2 + T(Y_2)) \\ &= e^{Y_1} + e^{Y_1} \cdot (Y_2 + T(Y_2)) \end{aligned} \qquad (9)$$

The relevance of this depends on the target architecture. Obviously, it is relevant to FPGAs without embedded multipliers. If such multipliers are available (like the 18×18 of some Virtex architectures), it is relevant if the size of one of the inputs gets smaller than 18. For instance a 18×24 multiplication followed by one addition may be considered more economical than a 24×24 multiplication consuming 4 embedded multipliers (see a detailed study of multiplier implementation on the Virtex family in [2]). Conversely, if the rectangular multiplication consumes 4 embedded multipliers anway, the addition should also be computed by these multipliers. This is the case for single precision.

2.4. A table-driven method

The previous algorithm involves two tables:

- The first, with p input bits and a few more than w_F output bits for e^{Y_1}, can not be compressed.

- The second, with about $w_F - 2p$ input bits and as many output bits, is suited for compression using usual table-based methods derived from the bipartite method. We use the freely available HOTBM method [8].

Compressed or not, the sizes of these tables grow exponentially with their input size. This makes the previous algorithm well suited for precisions up to (and slightly above) single precision ($w_F = 23$ bits). For much smaller precisions, of course, simpler approaches will be more effective. For much larger precisions, like double precision, the algorithm has to be modified. An idea is to repeat the second range reduction several

times, each time replacing p input bits to the tables by one new p-input-bits table and one almost full-size multiplier. Another solution is to compute e^{y_2} using a higher degree polynomial, which also increases the multiplier count. A more detailed study remains to be done.

3. Architecture

The architecture of this operator is given on Figure 1. It is a straightforward implementation of the algorithm. Obviously this architecture is purely sequential and can be pipelined easily.
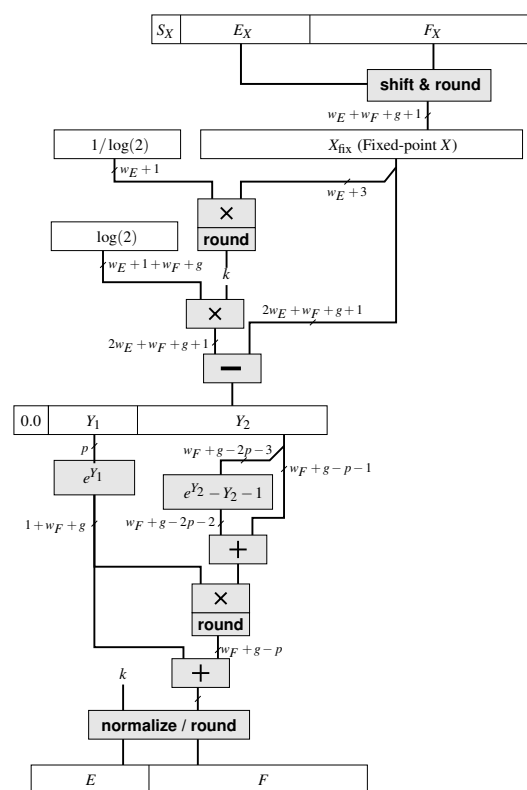

Figure 1. Architecture

The architecture has two parameters, p and g. The first essentially drives a tradeoff between the sizes of the two tables, and its value should be between $p = w_F/4$ and $p = w_F/3$. The second is a number of guard bits used to contain rounding errors, and will be studied in Section 4.

Some more comments about this architecture:

- The *shift & round* operator 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 would mean overflow). If the exponent is negative, it shifts to the right by up to $w_F + g$ positions.

- The range check (which verifies if the exponential of the input is representable in the given format, or if an infinity or a zero should be returned) is performed by the *shift & round* and the first multiplication stages.

- The intermediate value of X_{fix} has $w_E + w_F + g + 1$ bits with a fixed binary point after the $w_E + 1$-th. The computation of $X_{\text{fix}} - k \cdot \log(2)$ will cancel the integer part and the first bit of the fractional part, as shown below in 4.

- The two first multipliers are constant multipliers, for which a range of optimization techniques may apply [3, 4]. This is currently not exploited.

- This figure shows the final multiplication implemented as a multiplier followed by an adder, but as shown in Section 2.3, depending on the target architecture, it may make more sense to have one single multiplier instead.

- Final normalization possibly shifts left the mantissa by one bit (as will be shown in 4.1), then rounds the mantissa, then possibly shifts back right by one bit in the rare case when rounding also changes the exponent. Each shift is associated with an increment/decrement of the exponent.

Several of these blocks can certainly be the subject of further minor optimizations.

4. Error analysis

Error analysis for floating-point algorithms is much more complex than for fixed-point operators. The goal is to obtain a floating-point operator which guarantees faithful rounding, *ie.* an error of less than one *unit in the last place* (ulp) of the result.

There are two aspects to the error analysis. First, the range reduction should implement (4) with the minimum hardware. Second, the whole of the computation should ensure faithful rounding, considering the method and rounding errors involved.

4.1. Range reduction

As k will be the exponent (plus or minus 2) of the final exponential, it fits on a $w_E + 1$ machine word.

If (4) were exact, the value of Y would be in $[-\frac{\log(2)}{2}, \frac{\log(2)}{2}]$, ensuring that e^Y is in $[\frac{\sqrt{2}}{2}, \sqrt{2}]$. Using less accuracy to compute k, we accept that Y will be in an interval larger than $[-\frac{\log(2)}{2}, \frac{\log(2)}{2}]$. It will make little difference to the architecture to increase these bounds to the next power of two, which is $Y \in]-1/2, 1/2[$. One proves easily that this is obtained for all w_E by truncating $1/\log(2)$ to $w_E + 1$ bits, and considering only $w_E + 3$ bits of X_{fix}.

This means that we will have e^Y in $[0.6, 1.7]$, so we will sometimes have to normalize the final result by shifting the mantissa one bit to the right and increasing the exponent by one.

4.2. Faithful rounding

The computation of e^Y involves a range of approximation and rounding errors, and the purpose of this section is to guarantee faithful rounding with a good percentage of correct rounding.

In the following, the errors will be expressed in terms of unit in the last place of Y. It is safe to reason in terms of ulps since all the computations are in fixed point, which makes it easy to align the binary point of each intermediate value. Here the ulp has the value $2^{-w_F + g}$. Then we can make an error expressed this way as small as required by increasing g.

First, note that the argument reduction is not exact. Equation (4) is implemented with an error, due to

- the approximation of $\log(2)$ to $w_E + 1 + w_F + g$ bits (less than one half-ulp),

- X_{fix} which is exact if it was shifted left, but was truncated if shifted right (one ulp),

- the truncation of Y to $w_F + g$ bits (one ulp).

Thus in the worst case we have lost 5 half-ulps.

Now we consider subsequent computations on Y carrying this error.

The table of e^{Y_1} holds an error of at most one half-ulp.

The table of $e^{Y_2} - Y_2 - 1$ is only faithful because it uses the HOTBM compression technique (error up to one ulp, plus another ulp when truncating Y_2 to its most significant part). The previous error on Y is negligible for this table as its result is scaled by 2^{-2p-3}.

Due to the multiplier, the error due to the second table (2 ulps) added to the error on Y_2 (2.5 ulps) may be scaled by the value contained in the first table (less than 1.7). This leads to an error of less than 8 ulps.

The first addition involves no error, we again lose one half-ulp when rounding the result of the multiplication, and the second addition adds the half-ulp error from the first table.

Finally the errors sum up to 9 ulps. Besides we have to take into account that we may need to shift the mantissa left in case of renormalization, so we have to provide one extra bit of accuracy for that. Altogether, we find that $g = 5$ guard bits for the intermediate computations ensure faithful rounding.

We implemented a test procedure that compares the result of this operator on a Celoxica RC-1000 board against a double-precision exponential on the host PC. Exhaustive testing for various precision has confirmed that the result is always faithful. Such a test takes about 12 hours for single precision, most of which is processor time, as next section will show.

A finer error analysis directing slight modifications of the algorithm (replacing some of the truncations by roundings) could probably reduce g.

5. Results

We obtained area and delay estimations of our exponential operator for several precisions. These results were computed using Xilinx ISE and XST 6.3 for a Virtex-II XC2V1000-4 FPGA.

They are shown in Figure 2, and a summary is given in Table 2, in terms of slices and percentage of FPGA occupation for the area, and of nanoseconds for the latency.
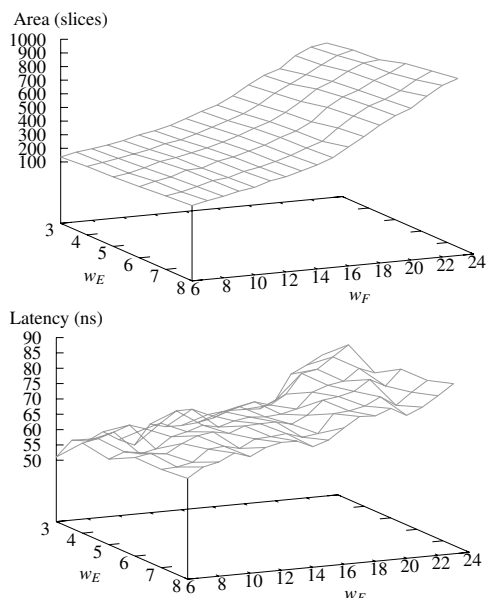

Figure 2. Area and latency estimations depending on w_E and w_F for the combinatorial operator with LUT-based multipliers.

In order to be as portable as possible, we do not require the use of the specific Virtex-II embedded 18×18 multipliers. Therefore we present the results obtained with and without those multipliers in Figure 3.

A single-precision operator has a latency of 85 ns. As a comparison, a 3GHz Pentium IV processor has a latency of 104ns (or 312 cycles for a single precision exponential using the GNU `glibc`, which relies on the machine instruction `f2xm1` which is itself micro-coded).

If most of the results presented here are for the combinatorial version, our operators are also available as pipelined operators, for an overhead of a few slices, as shown in Figure 4. The pipeline depth depends on the parameters w_E and w_F: 10 cycles for single precision, and less for lower precisions. The pipelined operators are designed to run at 100MHz on the targeted Virtex-II

Precision (w_E,w_F)	Using 18×18 multipliers	Area (slices, %)	Latency (ns)
(3, 6)	no	137 (2%)	51
	yes (3)	68 (1%)	47
(5, 10)	no	258 (5%)	63
	yes (4)	135 (2%)	57
(6, 13)	no	357 (6%)	69
	yes (5)	194 (3%)	65
(7, 16)	no	480 (9%)	69
	yes (5)	271 (5%)	68
(8, 23)	no	948 (18%)	85
	yes (9)	522 (10%)	83

Table 2. Synthesis results for the exponential operator on Xilinx Virtex-II.

XC2V1000-4 FPGA.

The only other comparable work we could find in the litterature [10] reports 5564 slices for a single-precision exponential unit which computes exponentials in 74 cycles fully pipelined at 85 MHz on a Virtex-II 4000. Our approach is much more efficient, because our algorithm is designed from scratch specifically for the FPGA. In contrast, the authors of [10] use an algorithm designed for microprocessors. In particular, they internally use fully featured floating-point adders and multipliers everywhere where we only use fixed-point operators.

6. Conclusion and future work

A parameterized floating-point implementation of the exponential function has been presented. For the 32-bit single precision format, its latency matches that of a Pentium IV. Being fully combinatorial, it can then be pipelined so that a single operator will provide several times the PIV throughput. Besides, it consumes a small fraction of the FPGA's resources.

We should moderate these results by two remarks. Firstly, our implementation is slightly less accurate than the Pentium one, offering faithful rounding only, where the Pentium uses an internal precision of 80 bits which ensures almost guaranteed correct rounding. Secondly, more recent instruction sets allow for lower latency for the elementary functions. The Itanium 2, for example, can evaluate a single-precision exponen-
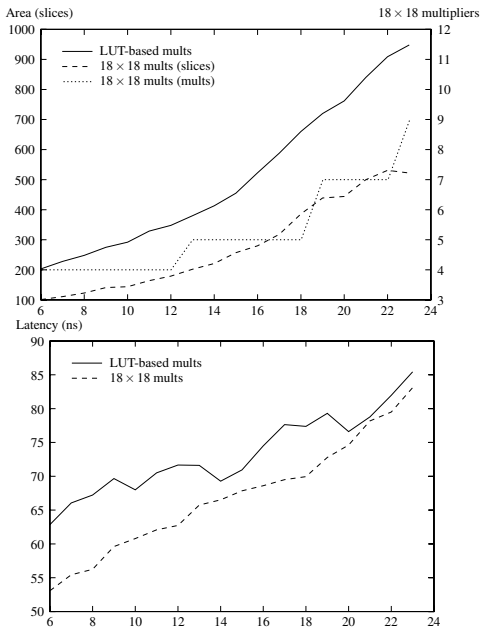
Figure 3. Comparison of area and latency depending on w_F **(**$w_E = 8$**), when using LUT-based multipliers, and when using the embedded** 18×18 **multipliers.**

Figure 4. Area estimations depending on w_F **(**$w_E = 8$**) for the combinatorial and pipelined versions of the operator with LUT-based multipliers.**

tial in about 40 cycles (or 20ns at 2GHz), and will therefore be just twice slower than our pipelined implementation. Thirdly, implementations of the exponential better optimized for single precision could probably be written for these recent processors. However the argument of massive parallelism will still apply.

The presented architecture can probably be fine-tuned. Constant multiplication algorithms should be investigated [3, 4]. The error analysis suggests that some of the truncations should be replaced with roundings to reduce the total error, and therefore the number of guard bits. Another approach would be to use different numbers of guard bits for the argument reduction and the computation of e^Y. Another idea is to remark that the critical paths for positive and negative exponents are quite different, which may allow for optimizations similar to the *close* and *far* paths used in the floating-point adder [6].

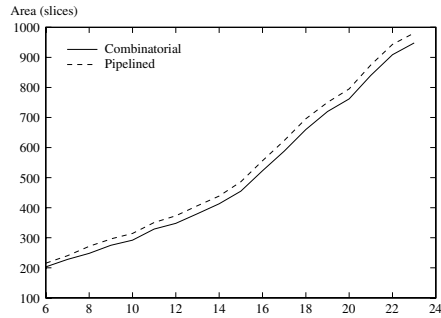Another future research direction, already evoked, is that the current architecture does not scale well beyond single precision: some of the building blocks have a size exponential in the precision. We will therefore explore algorithms which work up to double precision, which is the standard in processors - and soon in FPGAs [5, 11]. We are also investigating other elementary functions, starting with the logarithm [7].

FPLibrary and the operator presented here are available under the GNU Public Licence from `www.ens-lyon.fr/LIP/Arenaire/`.

Acknowledgements

7. References

[1] P. Belanović and M. Leeser. A library of parameterized floating-point modules and their use. In *Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 657–666. Springer, Sept. 2002.

[2] J.-L. Beuchat and A. Tisserand. Small multiplier-based multiplication and division operators for Virtex-II devices. In *Field-Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 513–522. Springer, Sept. 2002.

[3] K. Chapman. Fast integer multipliers fit in FP-

GAs (EDN 1993 design idea winner). *EDN magazine*, May 1994.

[4] F. de Dinechin and V. Lefèvre. Constant multipliers for FPGAs. In *2nd Intl Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE)*, pages 167–173, June 2000.

[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 tool for unbiased comparison between logarithmic and floating-point arithmetic. Technical Report RR2004-31, LIP, École Normale Supérieure de Lyon, Mar. 2004. Extended version of an article published in the Asilomar conference in 2003.

[7] J. Detrey and F. de Dinechin. A parameterizable floating-point logarithm operator for fpga. In *39th Asilomar Conference on Signals, Systems & Computers*. IEEE Signal Processing Society, Nov. 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] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. In *ACM/SIGDA Field-Programmable Gate Arrays*, pages 50–55, Feb. 2002.

[10] C. Doss and R. Riley. FPGA-based implementation of a robust IEEE-754 exponential unit. In *FPGAs for Custom Computing Machines*. IEEE, 2004.

[11] 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*, pages 86–95. ACM Press, 2005.

[12] B. Lee and N. Burgess. Parameterisable floating-point operators on FPGAs. In *36th Asilomar Conference on Signals, Systems, and Computers*, pages 1064–1068, 2002.

[13] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, april 2001.

[14] 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.

[15] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[16] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[17] F. Ortiz, J. Humphrey, J. Durbano, and D. Prather. A study on the design of floating-point functions in FPGAs. In *Field Programmable Logic and Applications*, volume 2778 of *LNCS*, pages 1131–1135. Springer, Sept. 2003.

[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] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machine. In *FPGAs for Custom Computing Machines*, pages 155–162. IEEE, 1995.

[20] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE.