FLOATING-POINT TRIGONOMETRIC FUNCTIONS FOR FPGAS

Jérémie Detrey, Florent de Dinechin

LIP, ENS-Lyon
46, allée d'Italie – 69364 Lyon Cedex 07
{Jeremie.Detrey, Florent.de.Dinechin}@ens-lyon.fr

ABSTRACT

Field-programmable circuits now have a capacity that allows them to accelerate floating-point computing, but are still missing core libraries for it. In particular, there is a need for an equivalent to the mathematical library (libm) available with every processor and providing implementations of standard elementary functions such as exponential, logarithm or sine. This is all the more important as FPGAs are able to outperform current processors for such elementary functions, for which no dedicated hardware exists in the processor. FPLibrary, freely available from `www.ens-lyon.fr/LIP/Arenaire/`, is a first attempt to address this need for a mathematical library for FPGAs. This article demonstrates the implementation, in this library, of high-quality operators for floating-point sine and cosine functions up to single-precision. Small size and high performance are obtained using a specific, hardware-oriented algorithm, and careful datapath optimisation and error analysis. Operators fully compatible with the standard software functions are first presented, followed by a study of several more cost-efficient variants.

1. INTRODUCTION

Floating-point and FPGAs FPGAs are increasingly being used as floating-point accelerators. Many libraries of floating-point operators for FPGAs now exist [1, 2, 3, 4, 5], typically offering the basic operators $+$, $-$, $\times$, $/$ and $\sqrt{}$. Published applications include matrix operations and scientific computing [6, 7, 8]. As FPGA floating-point is typically clocked ten times slower than the equivalent in contemporary processors, only massive parallelism allows these applications to be competitive to software equivalent.

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. After exponential [9] and logarithm [10], this paper studies the sine and cosine functions.

Elementary functions Elementary functions are available for virtually all computer systems. There is currently a general consensus that they should be implemented in software, although this question was long controversial [11]. Even processors offering machine instructions for such functions (mainly the x86/x87 family) implement them as micro-code.

Implementing floating-point elementary functions on FPGAs is a new problem. The flexibility of the FPGA paradigm allows one to use specific algorithms which turn out to be much more efficient than processor-based implementations. Previous work [9, 10] has shown that a single precision function consuming a small fraction of FPGA resources has a latency equivalent to that of the same function in a 2.4 GHz PC, while being fully pipelinable to run at 100 MHz. In other words, where the basic floating-point operator ($+$, $-$, $\times$, $/$, $\sqrt{}$) is typically ten times slower on an FPGA than its PC equivalent, an elementary function will be more than ten times faster for precisions up to single precision.

However, to benefit from the flexibility of FPGA, one should not use the algorithms implemented in libms [12, 13, 14]: These algorithms assume the availability of hardware operators for floating-point $+$, $-$, $\times$, $/$, $\sqrt{}$, and will not be efficient if one has to build these out of FPGA resources. Previous similar work, by Ortiz *et al.* [15] for the sine, and by and Doss and Riley [16] for the exponential, use this inefficient approach. In the present paper, the algorithms are hardware-oriented from scratch, and thus lead to architectures which are both much smaller and much faster. In particular, this work builds upon previous work dedicated to the evaluation in hardware of *fixed-point* elementary function (see [17] and references therein).

Contributions The first contribution of this work is the availability of high-quality operators for sine and cosine. Contrary to [15], our operators are properly specified to be software compatible. A careful error analysis guarantees last-bit accuracy, while ensuring that the datapath are never more accurate than needed.

Still, these trigonometric operators are about twice more resource consuming that the logarithm or exponential func-

1-4244-1060-6/07/$25.00 ©2007 IEEE.

tions. The main reason is the floating-point trigonometric argument reduction, which transforms the input argument into the interval $[0, \pi/4]$.

The second, and most novel, contribution of this article is therefore the study of alternative specifications of the functions which reduce the cost of the implementation without sacrificing accuracy. In the end, we offer a choice between functions that will allow for a straightforward translation of Matlab or C code, and several alternatives which consume less resource, but will require some adaptation in the surrounding algorithms. We explain why we believe that these adaptations will, more often than not, simplify the pipeline around the trigonometric functions.

Notations We use throughout this article the notations of the floating-point hardware library FPLibrary [5]. Floating-point numbers are composed of a sign bit S_x, an exponent E_x on w_E, and a mantissa F_x on w_F bits. In single precision, $w_E = 8$ and $w_F = 23$. In addition, two bits exn_x code for exceptional cases such as zeroes, infinities and NaN (*Not a Number*). Figure 1 depicts such a number, whose value is $x = (-1)^{S_x} \times 1.F_x \times 2^{E_x - E_0}$ with $E_0 = 2^{wE-1} - 1$.

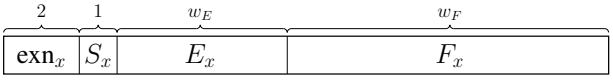


Fig. 1. Format of a floating-point number x.

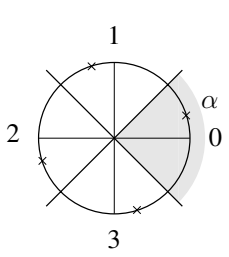
2. THE SINE AND COSINE FUNCTIONS

Argument reduction for radian angles When angles are expressed in radian, the main problem of the evaluation of a trigonometric function is to reduce the – possibly very large – input number x to a reduced argument α in the interval $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$. In other words, one needs to compute an integer k and a real α such that

$$\alpha = x - k\frac{\pi}{2} \in \left[-\frac{\pi}{4}, \frac{\pi}{4}\right].$$

After this argument reduction, the sine and cosine of the original number are deduced from those of y using classical identities depicted on Figure 2. In addition, as $\sin(-\alpha) = -\sin(\alpha)$ and $\cos(-\alpha) = \cos(\alpha)$, only positive α need to be considered.

Here, k is defined as $x \times \frac{2}{\pi}$ rounded to the nearest integer. A variant, used by Markstein for software implementations on IA64 processors [14], uses as reduced argument y the fractional part of $x \times \frac{4}{\pi}$ in order to evaluate $\sin\left(\frac{\pi}{4}y\right)$ and $\cos\left(\frac{\pi}{4}y\right)$. For a hardware implementation this has two advantages. First, compared to computing $\alpha = x - k\frac{\pi}{2}$, it saves one subtraction and one multiplication. Second, we intend to compute the sine and the cosine of the reduced



0	$\sin(x) = \sin(\alpha)$ $\cos(x) = \cos(\alpha)$
1	$\sin(x) = \cos(\alpha)$ $\cos(x) = -\sin(\alpha)$
2	$\sin(x) = -\sin(\alpha)$ $\cos(x) = -\cos(\alpha)$
3	$\sin(x) = -\cos(\alpha)$ $\cos(x) = \sin(\alpha)$

Fig. 2. Argument reduction.

argument using a table-based method [17]. Therefore, the reduced argument will be used as address to tables, and for this an argument in $\left[-\frac{1}{2}, \frac{1}{2}\right]$is better than one in $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$, because the bounds are powers of two. There is a drawback, however: the Taylor series of $\sin\left(\frac{\pi}{4}y\right)$ is less simple than that of $\sin(\alpha)$, and this will mean a more complex evaluation of the sine. All in all, we have performed a detailed study showing that the reduction to y has a lower hardware cost than the reduction to α.

Now the difficult question is to compute $x \times \frac{4}{\pi}$ to sufficient precision. A multiplier by the constant $\frac{4}{\pi}$ shall be used, but for very large x, one needs to store this constant with very large precision, since we are interested in the fractional part of the result. Fortunately, the full integer part k of the product $x \times \frac{4}{\pi}$ need not be computed: We are only interested in its last three bits, which hold all the quadrant and sign information. Bits of k of higher magnitude correspond to integer multiple of the period of the functions. As x is a floating-point number, we shall multiply its mantissa by $\frac{4}{\pi}$, and use its exponent to truncate to the left the constant $\frac{4}{\pi}$, in order to keep only those bits that will be needed to compute the three LSBs of k. This idea is due to Payne and Hanek, and its first mainstream implementation is Ng's for Sun Microsystems' `fdlibm` [18].

At least $2w_F$ bits of $\frac{4}{\pi}$ should be multiplied by the mantissa of x, as the w_F most significant bits of the product, being computed out of a left-truncated $\frac{4}{\pi}$ constant, will be invalid. But this is not enough: the fractional part y of $x \times \frac{4}{\pi}$ can come very close to zero for some values of x. In other words, the first binary digits of y may be a long sequence of zeroes. These zeroes should not be part of the (normal) floating-point number to be returned, and will provoke a left shift at the normalisation phase. Therefore, we need to add yet another g_K guard bits to the constant $\frac{4}{\pi}$. An algorithm by Kahan and Douglas [13] allows one to compute, on a given floating-point domain, the smallest possible value of y and hence the required g_K. Unfortunately, g_K is usually close to w_F.

Finally, a few – namely g – guard bits will also be needed to compensate for the other approximation and rounding errors. To sum up, the trigonometric range reduction requires

- the constant $\frac{4}{\pi}$ stored on roughly $2^{w_E-1} + 3w_F$,
- a shifter to extract roughly $3w_F$ bits from this $\frac{4}{\pi}$,
- a multiplier of size roughly $w_F \times 3w_F$ bits,
- and another shifter to possibly normalise the result.

The hardware cost will therefore be high. However, the delay can be reduced by a dual-path architecture (in the spirit of the one used in FP adders), which will be presented in section 3.

We have considered other argument reduction algorithms from the reference book by Muller [13]. Cody and Waite's technique relies on floating-point operators, and is only useful for small arguments. Variations of the modular range reduction algorithm [19, 20] have also been considered, but these iterative approaches are poorly suited to a pipelined implementation.

A dual sine/cosine architecture In the frequent case when both the sine and the cosine of a value have to be computed (e.g. to compute a rotation), the expensive argument reduction can be shared between both functions. Indeed, as presented above, both sine and cosine have to be computed, since the final result will be one or the other, depending on the quadrant. The proposed operator therefore outputs both the sine and the cosine of the input. If only one of these functions is needed, the constant $\frac{\pi}{4}$ should be replaced with $\frac{\pi}{2}$ in the previous. However, sections 4 and 5 will show that such a single function operator is almost as costly as the dual one. Therefore our reference implementation will be the dual one.

3. REFERENCE IMPLEMENTATION

This implementation is compatible with the spirit of the IEEE-754 standard, and with current best practice in software. It implements an accurate argument reduction to guarantee faithful rounding (or last-bit accuracy) of the result for any input.

The general architecture of this implementation is depicted by Figure 3. Figure 4 shows the argument reduction architecture, and Figure 5 depicts the evaluation of the sine and cosine of the reduced argument. A reconstruction stage implements the identities given by Figure 2 to deduce $\sin x$ and $\cos x$ from $\sin\left(\frac{\pi}{4}y\right)$, $\cos\left(\frac{\pi}{4}y\right)$, the octant, k, and the sign of x. Finally, a small unit handles exceptional cases, such as $\sin(+\infty)$ which should return NaN.

Dual-path argument reduction Let us consider the output of the argument reduction. We wish to compute $\sin\left(\frac{\pi}{4}y\right)$ and $\cos\left(\frac{\pi}{4}y\right)$, both as floating-point numbers. On one side, the cosine will be in $\left(\frac{\sqrt{2}}{2}, 1\right]$, therefore its exponent is known in advance. Hence, computing the mantissa

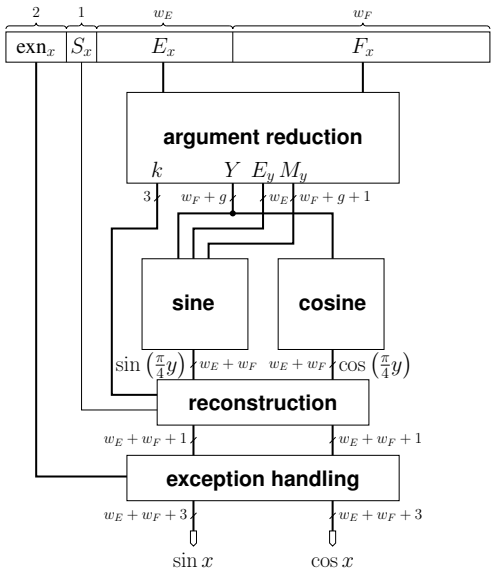


Fig. 3. Overview of the dual sine/cosine operator.

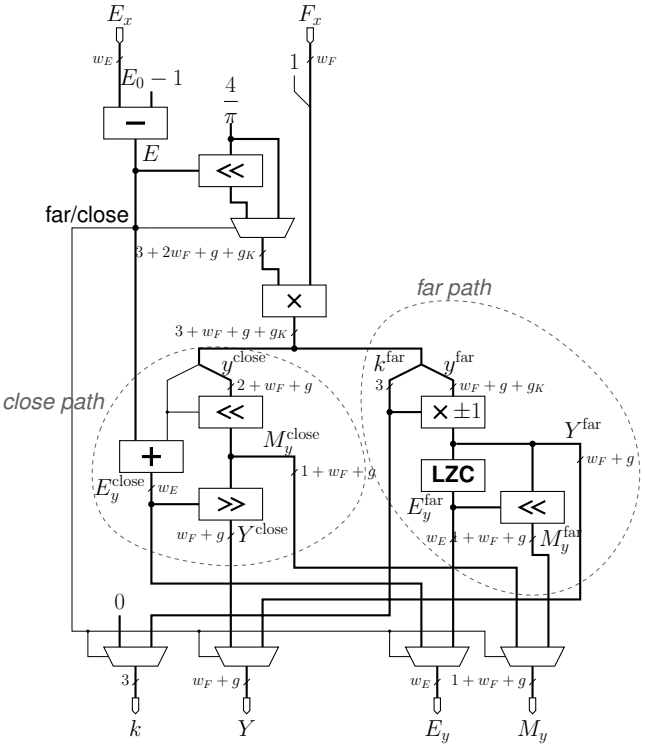


Fig. 4. Detailed view of dual-path argument reduction.

can be done by a *fixed-point* datapath, which only needs a fixed-point value of y. We note this fixed-point value Y. On the other side, the sine may fall close to zero: To compute it, we need a *floating-point* representation of y, whose exponent and mantissa are noted (E_y, M_y).

This floating-point representation is obtained in the gen-

eral case by a leading zero counter (LZC) followed by a barrel shifter. However, there is a special case when the input x is close to zero (in practice when $x < \frac{1}{2}$). In this case the exponent of y may be deduced from that of x, as both numbers are in a constant $\frac{4}{\pi}$ ratio. Therefore M_y and E_y are obtained quickly, but obtaining Y requires a variable shift of M_y, depending on the exponent E_y.

There are thus two exclusive datapaths, illustrated on Figure 4. The *close* path, for values of x close to 0, computes Y out of (E_y, M_y). The *far* path, for values of x far from zero, computes (E_y, M_y) out of Y.

Table-based evaluation of $\cos\left(\frac{\pi}{4}y\right)$ and $\sin\left(\frac{\pi}{4}y\right)$ We have considered the Cordic family of algorithms [13] for the evaluation of the sine and cosine themselves. Such algorithms have long latency and require small hardware when implemented sequentially, however the hardware cost becomes very large as soon as a pipelined version is required. Therefore, in this work, we shall use the HOTBM method [17], which allows for very compact pipelined implementations of elementary functions in fixed-point. This method approximates a function by a minimax polynomial, then builds for the evaluation of this polynomial an optimised parallel architecture out of look-up tables, powering units and small multipliers.

As the cosine may be computed in fixed-point, this method can be used directly. However the first bit, being always 1, is not computed: the function evaluated by HOTBM is

$$f_{\cos}(y) = 1 - \cos\left(\frac{\pi}{4}y\right).$$

As the sine may have a variable exponent, it is best computed as

$$\sin\left(\frac{\pi}{4}y\right) = y \times \frac{\sin\left(\frac{\pi}{4}y\right)}{y}.$$

Now the right-hand term of the product has the following

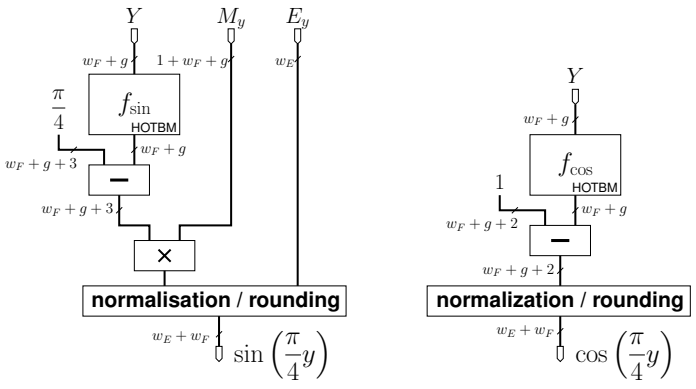


Fig. 5. Evaluation of sine and cosine

Taylor expansion: $\frac{\sin\left(\frac{\pi}{4}y\right)}{y} \approx \frac{\pi}{4} + O(y^2)$. This shows that it may be computed in fixed-point, and also that the exponent of the result will be that of y. The function evaluated by HOTBM for the sine is therefore

$$f_{\sin}(y) = \frac{\pi}{4} - \frac{\sin\left(\frac{\pi}{4}y\right)}{y}.$$

Then we have to multiply a floating-point number (E_y, M_y) by a fixed-point number, which is even simpler than a floating-point multiplication.

Error analysis Our objective is guaranteed faithful rounding (or last-bit accuracy, or a relative error smaller than 2^{-w_F}). This constraint, along with a careful study of the cumulated rounding errors (the HOTBM operators themselves produce faithful results), allows us to determine the number of guard bits required on the datapath. This error analysis is not specially interesting, and is omitted here due to space constraints. The interested reader will follow the bit widths on the previous figures (it turns out that at most $g = 2$ guard bits are required for faithful rounding). The only subtlety is that argument reduction is not exact, therefore the inputs to the HOTBM operators also require to be extended with g guard bits, and their output is faithful with respect to this error-carrying input.

4. DEGRADED IMPLEMENTATIONS

Considering the relatively high area of the reference dual sine/cosine operator, we have explored several alternatives exposing a trade-off between area, delay and precision. These alternatives are presented here, and implementation results will be given in section 5.

A first remark is that if the user is able to bound the range of the input, he can correspondingly reduce w_E, the exponent size, since the operators are fully parameterised and the output doesn't require a large exponent either. This will save some hardware, all the more as the output will be prevented to come as close to zero as with a larger exponent. However, the cost of the operator depends more on w_F, the size of the mantissa, than on w_E.

Functions of πx The standard trigonometric functions are specified with angles in radian to match the pure mathematical functions. However, from an application point of view, this may not be the best choice. For instance, many programs compute something like $\sin(\omega t)$ where the constant ω is defined as a multiple of π, like $\omega = \frac{2\pi}{T}$. In effect, the multiplication by ω performs the conversion of a time into a dimensionless number in radian.

For our purpose, it means that a function computing $\sin(\pi x)$ will be equally satisfying for the programmer, requiring only a change in the constant ω in our example.

Of course, the argument reduction becomes trivial, as the costly multiplication by the irrational number $\frac{4}{\pi}$ is no longer needed. All it takes now is to split x into its integer part and fractional part. This operation is not only very cheap, it is also error-less, which saves a guard bit in the subsequent data-paths, including the inputs to $f_{\sin}$ and $f_{\cos}$. The area and delay are therefore much reduced.

Our opinion is that this operator will prove the most popular, all the more as the upcoming revision of the IEEE-754 standard for floating-point arithmetic should introduce these functions, for reasons quite similar to those presented here.

A floating-point in, fixed-point out operator If an application can be contented with a fixed-point output, we no longer need the g_K guards bits of the $\frac{4}{\pi}$ constant, which saves one third of the multiplier used for the argument reduction.

Besides, this also simplifies the evaluation of $\sin\left(\frac{\pi}{4}y\right)$: First, there is no need anymore to normalise the result, and therefore no need for the floating-point version of y. Second, the HOTBM method can directly evaluate the sine, saving the multiplication by M_y.

Therefore, this (still dual) operator, being less accurate, is much smaller and faster than the reference one.

Single operator Some applications require only one of the functions, sine or cosine. We therefore also propose a version that computes only the sine of x. It requires an argument reduction to a quadrant instead of octant. Thus the reduced argument α will belong to the interval $\left[0, \frac{\pi}{2}\right]$, which is twice larger than for the dual operator.

Otherwise the argument reduction is very similar to the dual operator, with a constant $\frac{2}{\pi}$ instead of $\frac{4}{\pi}$. In particular, the large $w_F \times 3w_F$ multiplier is still necessary.

Besides, the evaluation of the function $f_{\sin}(\alpha)$ on a larger interval will require a larger HOTBM operator.

All in all, the single operator is therfore only slightly smaller than the dual one.

5. RESULTS

All the operators presented in this article have been synthesised, tested, placed and routed for various precisions, using Xilinx ISE/XST 7.1, for a Virtex-II XC2V1000-4. Table 1 gives area results in Virtex-II slices, and latency results after place and route in nanoseconds. In [15], Ortiz *et al.* present a 1431-slice single precision sine operator which runs in 18 105MHz cycles, but which is much less accurate and does not seem to perform any kind of range reduction.

As our designs involve many multiplications (including those hidden in the HOTBM operators), they may benefit from the embedded multipliers present in most current FPGAs. The user may choose, at synthesis time, to use them, or keep them for other parts of the application. Table 1 also gives results using such embedded multipliers, for the reference implementation and the trig-of-πx alternative.

We are currently working on pipelining these operators, which is slightly more difficult than expected as, for instance, the very large multiplier has to be split into many pipeline stages. Current results suggest a pipeline depth of 18 cycles at 100MHz for single precision on Virtex-II -4, providing one sine and one cosine every 10ns. For comparison, the average time for the default libm sine function on a 2.4 GHz Pentium 4 is roughly 200 cycles[1], not pipelined: This is a result every 80ns. Consistently with previous results [9, 10], the FPGA largely outperforms a contemporary processor for single-precision elementary functions.

6. CONCLUSION AND PERSPECTIVES

We have described a family of FPGA operators for the computation of sine and cosine in floating-point. These operators are parameterised by exponent and mantissa size up to single precision, are well specified and are of high numerical quality. Several approaches to the precision/performance tradeoff are proposed. These operators are part of FPLibrary and are freely available from `www.ens-lyon.fr/LIP/Arenaire/`.

In our opinion, the most promising is the version with operators for $\sin(\pi x)$ and $\cos(\pi x)$, which offer reduced area and improved performance without sacrificing any accuracy.

Many small improvements can still be brought to these operators, for instance using constant multiplication compression techniques [21]. However, the real challenge is to tackle double-precision. The current FPLibrary operators for the four operations scale well to double-precision, although optimisations are still possible. Elementary functions based on HOTBM evaluation, however, have an area exponential with respect to precision and are poorly suited beyond 32 bits [17]. We have designed specific algorithms for exponential and logarithm that scale quadratically with precision [22], and one option is to use similar ideas for the sine and cosine, which are mathematically very close to the (complex) exponential. More precisely, the idea would be to use a variant of the Cordic algorithm [13] with a granularity targeted to the FPGA LUT structure.

There are also other directions to explore. A second argument reduction could make the HOTBM approach more competitive. Also, more classical polynomial approximation techniques, along with a careful error analysis for the intermediate precisions, will provide a large implementation space to explore.

[1]This number can be the subject of endless discussions, as in software the time depends a lot on the input value. Here we use the average for 10000 values with a normal law on the exponent between exponent values -20 and 40.

	Dual sine/cosine					Dual sine/cosine of πx					Fixed-point out		Sine alone	
Precision (w_E, w_F)	Area (slices)	*Delay* (ns)	Area (slices + mult.)		*Delay* (ns)	Area (slices)	*Delay* (ns)	Area (slices + mult.)		*Delay* (ns)	Area (slices)	*Delay* (ns)	Area (slices)	*Delay* (ns)
(5,10)	803	*69*	424	7	*61*	363	*58*	244	3	*46*	376	*57*	709	*70*
(6,13)	1159	*86*	537	7	*76*	641	*61*	462	3	*61*	642	*63*	1027	*86*
(7,16)	1652	*91*	816	10	*87*	865	*73*	559	4	*69*	923	*72*	1428	*92*
(7,20)	2549	*99*	1372	17	*96*	1531	*84*	1005	8	*76*	1620	*82*	2050	*101*
(8,23)	3320	*109*	1700	19	*99*	2081	*89*	1365	10	*85*	2203	*88*	2659	*105*

Table 1. Area and latency for various variants of trigonometric operators

However, the complexity of the radian argument reduction is here to stay, and the first thing, before exploring the aforementioned alternatives, is to collect user feedback to see which functions should be developed.

7. REFERENCES

[1] 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*. IEEE, 1995, pp. 155–162.

[2] 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 *Field-Programmable Gate Arrays*. ACM, 2002, pp. 50–55.

[3] P. Belanović and M. Leeser, "A library of parameterized floating-point modules and their use," in *Field Programmable Logic and Applications*, ser. LNCS, vol. 2438. Springer, 2002, pp. 657–666.

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

[5] J. Detrey and F. de Dinechin, "A tool for unbiased comparison between logarithmic and floating-point arithmetic," *Journal of VLSI Signal Processing*, 2007, to appear.

[6] 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.

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

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

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

[10] ——, "A parameterizable floating-point logarithm operator for FPGAs," in *39th Asilomar Conference on Signals, Systems & Computers*. IEEE, 2005.

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

[12] P. T. P. Tang, "Table lookup algorithms for elementary functions and their error analysis," in *10th Symposium on Computer Arithmetic*. IEEE, June 1991.

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

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

[15] 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*, ser. LNCS, vol. 2778. Springer, Sept. 2003, pp. 1131–1135.

[16] 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.

[17] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2005, pp. 328–333.

[18] K. C. Ng, "Argument reduction for huge arguments: good to the last bit," SunPro, Mountain View, CA, USA, Technical Report, July 1992.

[19] M. Daumas, C. Mazenc, X. Merrheim, and J. M. Muller, "Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions," *Journal of Universal Computer Science*, vol. 1, no. 3, pp. 162–175, Mar. 1995.

[20] J. Villalba, T. Lang, and M. A. Gonzalez, "Double-residue modular range reduction for floating-point hardware implementations," *IEEE Transactions on Computers*, vol. 55, no. 3, pp. 254–267, Mar. 2006.

[21] F. de Dinechin and V. Lefèvre, "Constant multipliers for FPGAs," in *Parallel and Distributed Processing Techniques and Applications*, 2000, pp. 167–173.

[22] J. Detrey, F. de Dinechin, and X. Pujol, "Return of the hardware floating-point elementary function," in *18th Symposium on Computer Arithmetic*. IEEE, June 2007.