A parameterizable floating-point logarithm operator for FPGAs

Jérémie Detrey and Florent de Dinechin
Laboratoire de l'Informatique du Parallélisme
ENS Lyon – 46, allée d'Italie
F-69364 Lyon cedex 07 – France
Email: {Jeremie.Detrey, Florent.de.Dinechin}@ens-lyon.fr

***Abstract*— As FPGAs are increasingly being used for floating-point computing, a parameterized floating-point logarithm operator is presented. In single precision, this operator uses a small fraction of the FPGA's resources, has a smaller latency than its software equivalent on a high-end processor, and provides about 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 without even resorting to parallelism. The presented operator is freely available from `http://www.ens-lyon.fr/LIP/Arenaire/`.**

I. Introduction

A recent trend in FPGA computing is the increasing use of floating-point. Many libraries of floating-point operators for FPGAs now exist [15], [6], [1], [9], [3], usually 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 [10], [2], [8].

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 both the logarithm and exponential functions, 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 large consensus that they should be implemented in software [14]. Even processors offering machine instructions for such functions (mainly the x86/x87 family) implement them as micro-code. On such systems, it is easy to design faster software implementations: Software can use large tables which wouldn't be economical in hardware [16]. 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 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 10 times slower on an FPGA than its PC equivalent, an elementary function will be more than ten times faster at least for precisions up to single precision.

Writing a *parameterized* elementary function is a completely new challenge: to exploit FPGAs' flexibility, one should not use the same algorithms as used for implementing elementary functions in software [16], [12], [11]. This paper describes an approach to this challenge, which builds upon previous work dedicated to fixed-point elementary function approximations (see [5] and references therein).

The authors are aware of only two previous works on floating-point elementary functions for FPGAs, studying the sine function [13] and studying the exponential function [7]. 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 III will show.

II. A floating-point logarithm

Notations

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

A. Evaluation algorithm

1) Range reduction: We consider here only the case where X is a valid positive floating-point number (*ie.* $\mathrm{exn}_X = 01$ and $S_X = 0$), otherwise the operator simply returns NaN. We therefore have:

$$X = 1.F_X \cdot 2^{E_X - E_0}.$$

1-4244-0132-1/05/$20.00 ©2005 IEEE

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 *(Not a Number)*

TABLE I

VALUE OF X ACCORDING TO ITS EXCEPTION FLAGS exn_X.

If we define $R = \log X$, we obtain:

$$R = \log(1.F_X) + (E_X - E_0) \cdot \log 2.$$

In this case, we only have to compute $\log(1.F_X)$ with $1.F_X \in [1, 2)$. The product $(E_X - E_0) \cdot \log 2$ is then added back to obtain the final result.

In order to avoid catastrophic cancellation when adding the two terms, and consequently maintain low error bounds, we use the following equation to center the output range of the fixed-point $\log$ function around 0:

$$R = \begin{cases} \log(1.F_X) + (E_X - E_0) \cdot \log 2 \\ \qquad\qquad \text{when } 1.F_X \in [1, \sqrt{2}), \\ \log\left(\dfrac{1.F_X}{2}\right) + (1 + E_X - E_0) \cdot \log 2 \\ \qquad\qquad \text{when } 1.F_X \in [\sqrt{2}, 2). \end{cases} \quad (1)$$

We therefore have to compute $\log M$ with the input operand $M \in [\sqrt{2}/2, \sqrt{2})$, which gives a result in the interval $[-\log 2/2, \log 2/2)$.

We also note in the following $E = E_X - E_0$ when $1.F_X \in [1, \sqrt{2})$, or $E = 1 + E_X - E_0$ when $1.F_X \in [\sqrt{2}, 2)$.

2) Fixed-point logarithm: As we are targeting floating-point, we need to compute $\log M$ with enough accuracy in order to guarantee faithful rouding — *ie.* an error of less than one *unit in the last place* (ulp) of the result —, even after a possible normalization of the result. As $\log M$ can be as close as possible to 0, a straightforward approach would require at least a precision of $2w_F$ bits, as the normalization could imply a left shift of up to w_F bits, and w_F bits would still be needed for the final result.

But one can remark that when M is close to 1, $\log M$ is close to $M - 1$. Therefore, a two-step approach consisting of first computing $\log M/(M-1)$ with a precision of $w_F + g_0$ bits and then multiplying this result by $M - 1$ (which is computed exactly) leads to the targeted accuracy at a smaller cost.

The function $f(M) = \log M/(M-1)$ is then computed by a generic polynomial method [5]. The order of the considered polynomial obviously depends on the precision w_F.

3) Reconstruction: As the evaluation of $f(M)$ is quite long, we can in parallel compute the sign of the result: If $E = 0$, then the sign will be the sign of $\log M$, which is in turn positive if $M > 1$ and negative if $M < 1$. And if $E \neq 0$, as $\log M \in [\sqrt{2}/2, \sqrt{2})$, the sign will be the sign of $E \cdot \log 2$, which is the sign of E.

We can then compute in advance the opposite of E and $M - 1$ and select them according to the sign of the result. Therefore, after the summation of the two products $E \cdot \log 2$ and $Y = f(M) \cdot (M - 1)$, we obtain Z the absolute value of the result.

The last steps are of course the renormalization and rounding of this result, along with the handling of all the exceptional cases.

B. Architecture

The architecture of the logarithm operator is given on Figure 1. It is a straightforward implementation of the algorithm presented in Section II-A. Due to its purely sequential dataflow, it can be easily pipelined. The values for the two parameters g_0 and g_1 are discussed in Section II-C.

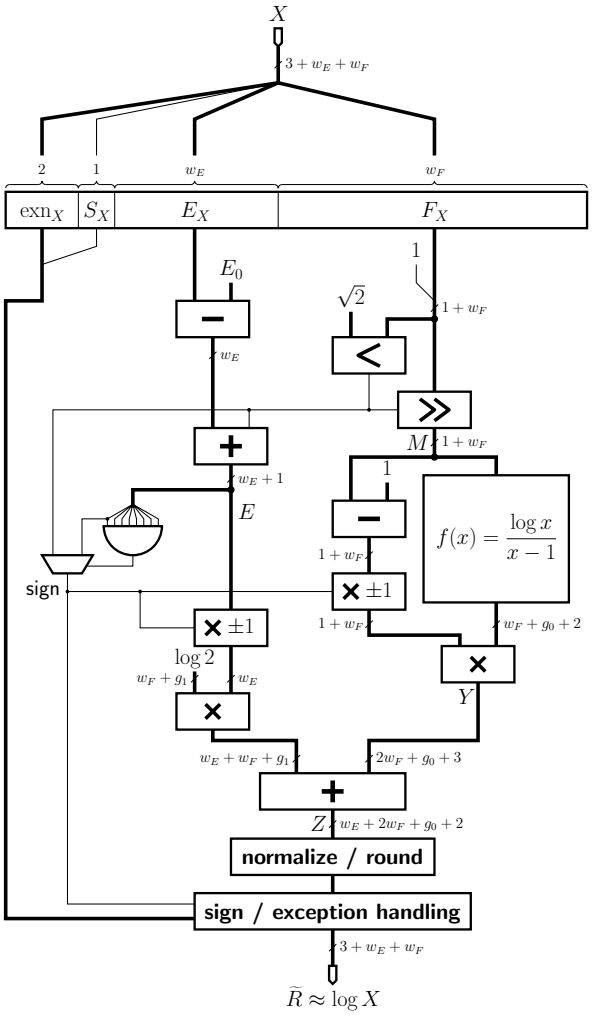


Fig. 1. Architecture of the logarithm operator.

Some comments about this architecture:

- Remark that the boundary between the two cases of Equation (1) does not have to be exactly $\sqrt{2}$, as both alternatives are valid on the whole of $[1, 2)$. This means that the comparison between the mantissa $1.F_X$ and $\sqrt{2}$ may be performed on a few bits only, saving hardware. We do not have any longer that $M \in [\sqrt{2}/2, \sqrt{2})$ and

$\log M \in [-\log 2/2, \log 2/2)$, but we use the smallest approximation to $\sqrt{2}$ that do not increase the bounds of these intervals to the next power of two. Thus the savings in this step do not lead to increased hardware on the subsequent steps.

- The sign of the result is the sign of E when $E \neq 0$. If $E = 0$, we also need to take into account the result of the comparison between $1.F_X$ and $\sqrt{2}$.
- The function $f(x)$ is evaluated using the Higher-Order Table-Based Method (HOTBM) presented in [5]. It involves a piecewise polynomial approximation, with variable accuracy for the coefficients and where all the terms are computed in parallel.
- The normalization of the fixed-point number Z uses a leading-zero counter, and requires shifting Z by up to w_F bits on the left and up to w_E bits on the right.
- Underflow cases are detected by the *sign & exception handling* unit.

As the tables sizes grow exponentially with the precision, this architecture is well suited for precisions up to single precision ($w_F = 23$ bits), and slightly more. Area on Virtex circuits will be given for a range of precision in Section III.

C. Error analysis

In order to guarantee faithful rounding for the final result we need to have a constant bound on the relative error of the fixed-point number $Z = \log X$:

$$\frac{|Z - \widetilde{Z}|}{2^{\lfloor \log_2 |Z| \rfloor}} < 2^{-w_F - 1},$$

so that when rounding the result mantissa to the nearest, we obtain a total error bound of 2^{-w_F}.

We need to consider several cases depending on the value of E:

- When $|E| > 3$, $|Z| > 2$ and the predominant error is caused by the discretization error of the $\log 2$ constant, which is multiplied by E.
- When $E = 0$, on the other hand, the only error is caused by the evaluation of $f(M)$, which is then scaled in the product $f(M) \cdot (M - 1)$. As the multiplicand $M - 1$ is computed exactly, this product does not entail any other error.
- When $|E| = 2$ or 3, both the discretization error from $\log 2$ and the evaluation error from $f(M)$ have to be taken into account. However, in this case, we have $|Z| > 1$. Therefore no cancellation will occur, and the discretization error will not be amplified by the normalization of Z.
- When $|E| = 1$, we have:

$$0.34 < \frac{1}{2}\log 2 \leq |Z| \leq \frac{3}{2}\log 2 < 1.04.$$

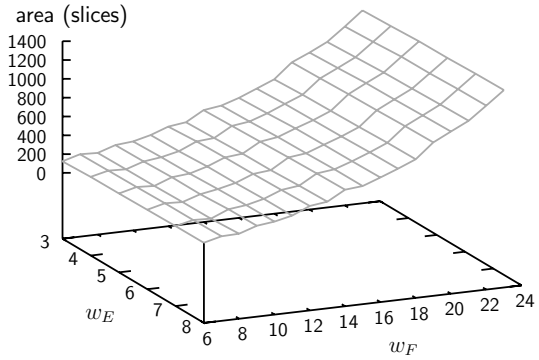
In this case, a cancellation of up to 2 bits can occur, which will multiply the $\log 2$ discretization error by at most 4.

One can then find that using $g_1 = 3$ guard bits for the $\log 2$ constant and bounding the evaluation error $\epsilon_f < 2^{-w_F - 3}$ satisfies all these constraints. The number of guard bits g_0 is given by the evaluation scheme used for $f(M)$, and is typically comprised between 1 and 5 bits.

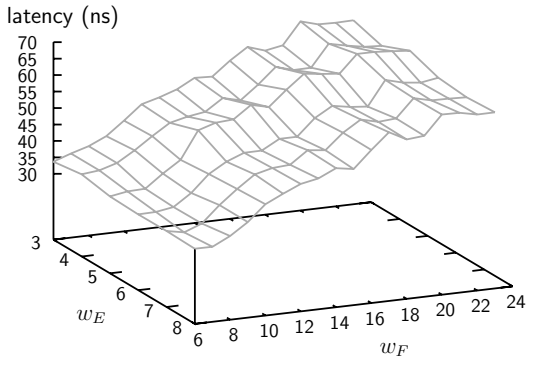
All these choices have been proven valid by exhaustively testing our operators on a Celoxica RC-1000 board (with a VirtexE-2000 FPGA) against a double precision software function, for the whole parameter space defined by $w_E \in [3, 8]$ and $w_F \in [6, 23]$. This exhaustive testing showed that the the result was always faithful, and was correctly rounded to nearest in more than 98% of the cases.

III. Results

We obtained area and delay estimations of the logarithm 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 II, in terms of slices and percentage of FPGA occupation for the area, and in terms of nanoseconds for the latency.



(a) Logarithm operator area



(b) Logarithm operator latency

Fig. 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.

Precision (w_E,w_F)	Multipliers	Logarithm Area (slices	%	mults)	Latency (ns)
$(3, 6)$	LUT-based	123	(2%)	–	34
	18×18	89	(1%)	2	31
$(5, 10)$	LUT-based	263	(5%)	–	42
	18×18	154	(3%)	3	39
$(6, 13)$	LUT-based	411	(8%)	–	48
	18×18	233	(4%)	3	44
$(7, 16)$	LUT-based	619	(12%)	–	57
	18×18	343	(6%)	6	55
$(8, 23)$	LUT-based	1399	(27%)	–	64
	18×18	830	(16%)	9	61

TABLE II
SYNTHESIS RESULTS FOR THE OPERATOR ON XILINX VIRTEX-II.

Therefore we present the results obtained with and without those multipliers in Figure 3.

Most of the results presented here are for the combinatorial version. However, the operator is also available as a pipelined operator, for a small overhead in area, as shown in Figure 4. The pipeline depth, comprised between 5 and 11 cycles, depends on the parameters w_E and w_F. The pipelined operator is designed to run at 100 MHz on the targeted Virtex-II XC2V1000-4 FPGA.

As a comparison, Table III presents the performances for the logarithm operator in single precision, along with the measured performances for a 2.4 GHz Intel Xeon processor, using the single precision operator from the GNU `glibc` (which itself relies on the micro-coded machine instruction `fyl2x`).

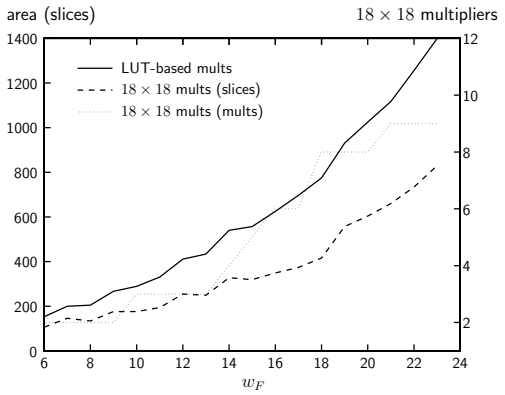
Architecture	Cycles	Latency (ns)	Throughput (10^6 op/s)
2.4 GHz Intel Xeon	196	82	12
100 MHz Virtex-II FPGA	11	64	100

TABLE III
PERFORMANCE COMPARISON BETWEEN INTEL XEON AND VIRTEX-II FOR SINGLE PRECISION ($w_E = 8, w_F = 23$).

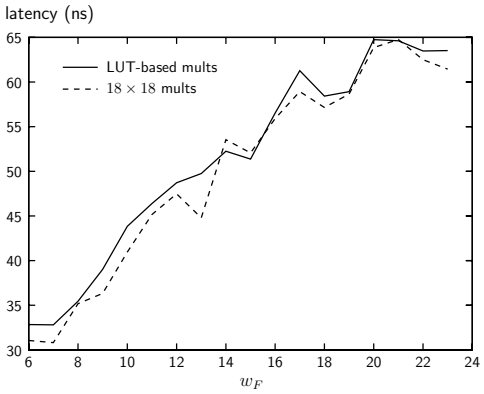
The only other comparable work we could find in the litterature [7] reports 5564 slices for a single precision exponential unit which computes exponentials in 74 cycles fully pipelined at 85 MHz on a Virtex-II 4000. Our approach is much more efficient, because our algorithm is designed from scratch specifically for the FPGA. In contrast, the authors of [7] 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.

IV. CONCLUSION AND FUTURE WORK

Parameterized floating-point implementations for the logarithm function has been presented. For the 32-bit single precision format, its latency matches that of a Xeon processor, and its pipelined version provides several times the Xeon throughput. Besides, it consumes a small fraction of the FPGA's resources.



(a) Logarithm operator area



(b) Logarithm operator latency

Fig. 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.

We should moderate these results by a few remarks. Firstly, our implementation is slightly less accurate than the Xeon one, offering faithful rounding only, where the Xeon uses an internal precision of 80 bits which ensures almost guaranteed correct rounding. Implementations for the logarithm better optimized for single precision could probably be written. Secondly, more recent instruction sets allow for lower latency for the elementary functions. The Itanium 2, for example, can evaluate a single precision logarithm in about 40 cycles (or 20 ns at 2 GHz), and will therefore be just twice slower than our pipelined implementation. However the argument of massive parallelism will still apply.

A future research direction, already evoked, is that the current architecture do 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

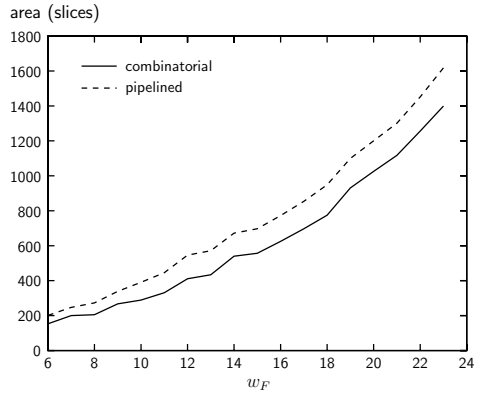


Fig. 4. Area estimations depending on w_F ($w_E = 8$) for the combinatorial and pipelined versions of the operator with LUT-based multipliers.

- and soon in FPGAs [2], [8]. We are also investigating other elementary functions to extend the library, such as the exponential [4].

This work also suggests that a complete library for floating-point on FPGAs is but an intermediate research goal: A longer-term goal is the automatic or assisted generation of arbitrary functions in an optimized way. For instance, if a given application involves a compound floating-point function, such as $\exp x^2$, a specific implementation of this compound function will very probably be more efficient than the combination of several library components. In fixed-point, methods like HOTBM provide this adaptability and flexibility. Floating-point, however, is much more challenging.

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

Acknowledgments

The authors would like to thank Arnaud Tisserand for many interesting discussions and for maintening the servers and software on which the experiments were conducted.

REFERENCES

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

[2] 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.

[3] J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. Technical Report 2004-31, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, Lyon, F-69364, France, June 2004.

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

[5] J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In S. Vassiliadis, N. Dimopoulos, and S. Rajopadhye, editors, *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'05)*, pages 328–333, Samos, Greece, July 2005. IEEE Computer Society.

[6] 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 10th International Symposium on Field-programmable Gate Arrays*, pages 50–55, Monterey, USA, Feb. 2002.

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

[8] 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.

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

[10] G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, Sept. 2002.

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

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

[13] 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.

[14] 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.

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

[16] 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.