2010 International Conference on Field Programmable Logic and Applications

Multiplicative square root algorithms for FPGAs

Florent de Dinechin, Mioara Joldes, Bogdan Pasca, Guillaume Revy
Arénaire, LIP (CNRS/INRIA/ENS-Lyon/UCBL), Université de Lyon
ENS-Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France
Email: {Florent.de.Dinechin,Mioara.Joldes,Bogdan.Pasca,Guillaume.Revy}@ens-lyon.fr

***Abstract*—Most current square root implementations for FPGAs use a digit recurrence algorithm which is well suited to their LUT structure. However, recent computing-oriented FPGAs include embedded multipliers and RAM blocks which can also be used to implement quadratic convergence algorithms, very high radix digit recurrences, or polynomial approximation algorithms. The cost of these solutions is evaluated and compared, and a complete implementation of a polynomial approach is presented within the open-source FloPoCo framework. This polynomial approach allows a shorter latency and higher frequency than the digit recurrence approach, and improves over previous multiplicative approaches. However, the cost of IEEE-compliant correct rounding is shown to be very high.**

***Index Terms*—FPGA; square root; floating-point**

I. Introduction

A. Algorithms for floating-point square root

There are two main families of algorithms which can be used to extract square roots.

The first family is that of *digit recurrence* algorithms, which provide one digit (often one bit) of the result at each iteration. Each iteration consists of additions and digit-by-number multiplications [1]. Such algorithms have been widely used in microprocessors that didn't include hardware multipliers. Most FPGA implementations in vendor tools or in the literature [2], [3], [4] use this approach, which was the obvious choice for early FPGAs which did not yet include embedded multipliers. This approach is also implemented in the FloPoCo project[1]. There, an approximate model of the delay of an iteration [5] is used to group several iterations in a single cycle if this is compatible with the user-specified target frequency. As the width of the computation increases as iterations progress, it is possible to pack more iterations in a cycle at the beginning of the computation than at the end. For instance, for a single precision square root pipelined for 100 MHz for Virtex-4, the 25 iterations are grouped as 7 + 5 + 5 + 4 + 4. Table I shows that this leads to state-of-the-art performance.

The second family of algorithms uses multiplications, and was studied as soon as processors included hardware multipliers. It includes quadratic convergence recurrences derived from the Newton-Raphson iteration, used in AMD IA32 processors starting with the K5 [6], in more recent instruction sets such as Power/PowerPC and IA64 whose floating-point unit is built around the fused multiply-and-add [7], [8], and in the INVSQRT core from the Altera MegaWizard [9]. Other variations involve piecewise polynomial approximations [10], [11]. On FPGAs, the VFLOAT project [12] uses an argument reduction based on tables and multipliers, followed by a polynomial evaluation of the reduced argument.

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

TABLE I
Pipelined digit-recurrence square roots on a Virtex-4 (4vfx100ff1152-12) using ISE 11.3.

Precision	Tool input	cycles	Synth. results
SP	FloPoCo 50 MHz	3	49 MHz, 253 sl.
	FloPoCo 100 MHz	6	107 MHz, 268 sl.
	LogiCore 6 cycles	6	86 MHz, 301 sl.
	FloPoCo 200 MHz	12	219 MHz, 327 sl.
	LogiCore 12 cycles	12	140 MHz, 335 sl.
	FloPoCo 400 MHz	25	353 MHz, 425 sl.
	LogiCore 28 cycles	28	353 MHz, 464 sl.
DP	FloPoCo 50 MHz	7	48 MHz, 1014 sl.
	FloPoCo 100 MHz	15	99 MHz, 1169 sl.
	FloPoCo 200 MHz	40	206 MHz, 1617 sl.
	FloPoCo 300 MHz	53	307 MHz, 1770 sl.
	LogiCore 57 cycles	57	265 MHz, 1820 sl.

To sum up, digit recurrence approaches allow one to build minimal hardware, while multiplicative approaches allow one to make the best use of available resources when these include multipliers. As a bridge between both approaches, a very high radix algorithm introduced for the Cyrix processors [13] is a digit-recurrence approach where the digit is 17-bit wide, and digit-by-number multiplication uses the 17x69-bit multiplier designed for floating-point multiplication.

Now that high-end FPGAs embed several thousands of small multipliers, the purpose of this article is to study how this resource may be best used for computing square roots. We evaluate a multiplier-based square root based on polynomial evaluation which is, to our knowledge, original in the context of FPGAs. The wider goal of this work is to provide the best possible square root implementations in FloPoCo.

B. Floating-point issues for square root

We compute the square root of a floating-point number X in a format similar to IEEE-754:

$$X = 2^E \times 1.F$$

where E is an integer, and F is the fraction part of the mantissa, written in binary on w_F bits: $1.F = 1.f_{-1}f_{-2}\cdots f_{-w_F}$ (the indices denote the bit weights).

There are classically two cases to consider.

- If E is even, $\sqrt{X} = 2^{E/2} \times \sqrt{1.F}$.

978-0-7695-4179-2/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.112

TABLE II
COMPARISON OF DOUBLE-PRECISION SQUARE ROOT OPERATORS. NUMBERS IN ITALIC ARE ESTIMATIONS.

Algorithm	precision	latency	frequency	slices	DSP	BRAM
FloPoCo digit recurrence	0.5 ulp	53 cycles	307 MHz	1740	0	0
Radix-2^{17} digit recurrence [14], [15]	0.5 ulp	*30 cycles*	*300 MHz*	?	*23*	*1*
VFLOAT [12]	2.39 ulp	17 cycles	*>200 MHz*	*1572*	*24*	*116*
Polynomial paper and pencil estimation (degree 4)	1 ulp	*25 cycles*	*300 MHz*	?	*18*	*20*
Altera ($1/\sqrt{x}$) [9]	1 ulp?	32 cycles	?	900 ALM	27	*32 M9K*

- If E is odd, $\sqrt{X} = 2^{(E-1)/2} \times \sqrt{2 \times 1.F}$.

In both cases the computation of the exponent of the result is straightforward, and the problem is reduced to computing $\sqrt{Z}$ for $Z \in [1, 4)$.

A detailed survey of multiplicative algorithms for floating-point square root in the context of FPGAs may be found in the extended version of this article [16]. The results, summarized in Table II, led us to implement a polynomial-based algorithm, which we now detail.

II. THE COST OF CORRECT ROUNDING

Given a floating-point format with w_F bits of mantissa, it makes no sense to build an operator which is accurate to less than w_F bits: it would mean wasting storage bits, especially on an FPGA where it is possible to use a smaller w_F instead. However, the literature distinguishes two levels of accuracy.

- **IEEE-754 correct rounding:** the operator returns the FP number nearest to $\sqrt{X}$. This corresponds to a maximum error of 0.5 ulp with respect to the exact mathematical result, where an ulp (*unit in the last place*) represents the binary weight of the last mantissa bit of the correctly rounded result – with our notations, the ulp value is 2^{-w_F}. Correct rounding is the best that the format allows.
- **Faithful rounding:** the operator returns one of the two FP numbers closest to $\sqrt{X}$, but not necessarily the nearest. This corresponds to a maximum error strictly smaller than 1 ulp.

In general, to obtain a faithful evaluation of a function such as $\sqrt{X}$ to w_F bits, one needs to first approximate it to a precision higher than that of the result (we denote this intermediate precision $w_F + g$ where g is a number of guard bits), then round this approximation to the target format. This final rounding performs an incompressible error of almost 0.5 ulp in the worst case, therefore it is difficult to directly obtain a correctly rounded result: one needs a very large g, typically $g \approx w_F$ [17]. It is much less expensive to obtain a faithful result: a small g (typically less than 5 bits) is enough to obtain an approximation on $w_F + g$ bits with a total error smaller than 0.5 ulp, to which we then add the final rounding error of another 0.5 ulp to obtain the 1-ulp bound of faithful rounding.

However, in the specific case of square root, there is a specific technique that converts a faithful square root on w_F+1 bits to a correctly rounded one on w_F bits. This technique is, to our knowledge, due to [10], and its use in the context of a hardware operator is novel.

We first compute a value of the square root $\widetilde{r}$ on $w_F + 1$ bits, faithfully rounded to that format (total error smaller than 2^{-w_F-1}). This is relatively cheap. Now, with respect to the w_F-bit target format, $\widetilde{r}$ is either a floating-point number, or the exact middle between two consecutive floating-point numbers. In the first case, the total error bound of 2^{-w_F-1} on $\widetilde{r}$ entails that it is the correctly rounded square root. In the second case, squaring $\widetilde{r}$ and comparing it to X tells us (thanks to the monotonicity of the square root) if $\widetilde{r} < \sqrt{X}$ or $\widetilde{r} > \sqrt{X}$ (it can be shown that the case $\widetilde{r} = \sqrt{X}$ is impossible). This is enough to conclude which of its two neighbouring floating-point numbers is the correctly rounded square root on w_F bits.

The following algorithm is a simple rewriting of this idea:

$$\circ(\sqrt{X}) = \begin{cases} \widetilde{r} \text{ truncated to } w_F \text{ bits} & \text{if } \widetilde{r}^2 \geq X, \\ \widetilde{r} + 2^{-w_F-1} \text{ truncated to } w_F \text{ bits} & \text{otherwise.} \end{cases} \quad (1)$$

With respect to performance/cost, one may observe that the overhead of correct rounding over faithful rounding is

- a faithful evaluation on $w_F + 1$ bits – this is only marginally more expensive than on w_F bits;
- a square on $w_F + 1$ bits – even with state-of-the-art dedicated squarers [18], this is expensive. Actually, as we are not interested in the high-order bits of the square, some of the hardware should be saved here, but this has not been explored yet.

This overhead (both in area and in latency) may be considered a lot for an accuracy improvement of one half-ulp. Indeed, on an FPGA, if no strict IEEE compliance is required, it may make sense to favor faithful rounding on a larger format ($w_F + 1$ bits) over correct rounding on w_F bits, for the same relative accuracy bound.

III. SQUARE ROOT BY POLYNOMIAL APPROXIMATION

As stated earlier, we address the problem of computing $\sqrt{Z}$ for $Z \in [1, 4)$. We are classically splitting the interval $[1, 4)$ into sub-intervals, and using for each sub-interval an approximation polynomial whose coefficients are read from a table. The state of the art for obtaining such polynomials is the `fpminimax` command of the Sollya tool. The polynomial evaluation hardware is shared by all the polynomials, therefore they must be of same degree d and have coefficients of the same format (here a fixed-point format). We evaluate the polynomial in Horner form, computing just right at each step by truncating all intermediate results to the bare minimum. Space is missing to provide all the details, which can be found

in [19] or in the open-source FloPoCo code itself. Let us focus here on specific optimizations related to the square root.

A first idea to address the coefficient table is to use the most significant bits of Z. However, as $Z \in [1,4)$, the values 00xxx are unused, which would mean that one quarter of the table is never addressed. Besides, the function $\sqrt{Z}$ varies more for small Z, therefore for a given degree d, polynomials on the left of $[1,4)$ are less accurate than those on the right. A solution to both problems is to make two cases according to exponent parity: $[1,2)$ (even case) will be split in as many sub intervals as $[2,4)$, and the sub-intervals on $[1,2)$ will be twice as small as those on $[2,4)$.

Here are the details of the algorithm. Let k be an integer parameter that defines the number of sub-intervals (2^k in total). The coefficient table has 2^k entries.

- If E is even, let $\tau_{\text{even}}(x) = \sqrt{1+x}$ for $x \in [0,1)$: we need a piecewise polynomial approximation for τ_{even}. The interval $[0,1)$ is split into 2^{k-1} sub-intervals $[\frac{i}{2^{k-1}}, \frac{i+1}{2^{k-1}})$ for i from 0 to $2^{k-1}-1$. The index (and table address) i consists of the bits $f_{-1}f_{-2}\cdots f_{-k+1}$ of the mantissa $1.F$. On each of these sub-intervals, $\tau_{\text{even}}(1+\frac{i}{2^{k-1}}+y)$ is approximated by a polynomial $p_i(y)$ of degree d.
- If E is odd, we need to compute $\sqrt{2\times 1.F}$. Let $\tau_{\text{odd}}(x) = \sqrt{2+x}$ for $x \in [0,2)$. The interval $[0,2)$ is also split into 2^{k-1} sub-intervals $[\frac{j}{2^{k-2}}, \frac{j+1}{2^{k-2}})$ for j from 0 to $2^{k-1}-1$. The reader may check that the index j consists of the same bits $f_{-1}f_{-2}\cdots f_{-k+1}$ as in the even case. On each of these sub-intervals, $\tau_{\text{odd}}(1+\frac{j}{2^{k-2}}+y)$ is approximated by a polynomial q_j of same degree d.

The error budget for a faithful evaluation may be summarized as follows. Let r be the value, represented on w_F+g bits, computed by the polynomial pipeline before the final rounding.

For a faithful approximation, we have to ensure a total error smaller than 2^{-w_F}. We must reserve 2^{-w_F-1} for the final rounding: $\epsilon_{\text{final}} < 2^{-w_F-1}$. This final rounding may be obtained at no cost by truncation of r to w_F bits, provided we have added one half-ulp (2^{-w_F-1}) to each coefficient c_0 stored in the table (we use $\lfloor z \rceil = \lfloor z + 1/2 \rfloor$).

The remaining 2^{-w_F-1} error budget is tentatively split evenly between polynomial approximation error: $\epsilon_{\text{approx}} = |\tau(y) - p(y)| < 2^{-w_F-2}$, and the total rounding error in the evaluation: $\epsilon_{\text{trunc}} = |r - p(y)| < 2^{-w_F-2}$.

Therefore, the degree d is chosen to ensure $\epsilon_{\text{approx}} < 2^{-w_F-2}$. As such, d is a function of k and w_F.

This way we obtain 2^k polynomials, whose coefficients are stored in a ROM with 2^k entries addressed by $A = e_0 f_{-1} f_{-2} \cdots f_{-k+1}$. Here e_0 is the exponent parity, and the remaining bits are i or j as above.

The reduced argument Y that will be fed to the polynomials is built as follows.

- In the even case we have $1.f_{-1}\cdots f_{-w_F}$ $= 1 + 0.f_{-1}\cdots f_{-k+1} + 2^{-k+1}0, f_{-k}\cdots f_{-w_F}$.
- In the odd case, we need the square root of $2 \times 1.F$ $= 1f_{-1}.f_{-2}\cdots f_{-w_F}$ $= 1 + f_{-1}.f_{-2}\cdots f_{-k+1} + 2^{-k+2}0, f_{-k}\cdots f_{-w_F}$.

As we want to build a single fixed-point architecture for both cases, we align both cases:

$y = 2^{-k+2} \times 0, 0 f_{-k}\cdots f_{-w_F}$ in the even case, and $y = 2^{-k+2} \times 0, f_{-k}\cdots f_{-w_F}0$ in the odd case.

Figure 1 presents the generic architecture used for the polynomial evaluation. More details can be found in [19].

IV. Results, comparisons, and some handcrafting

Table III summarizes the actual performance obtained from the polynomial square root operator `FPSqrtPoly` from FloPoCo version 2.0.0. All these operators have been tested for faithful rounding, using FloPoCo's testbench generation framework [5].

When the polynomials are obtained completely automatically [19], there is a large design space to explore, and we are still improving the heuristics for that. A difficulty is to integrate the staircase effects in the costs due to the discrete sizes of the multipliers and of the embedded memories. For the important case of single precision, we hardcoded a good choice of parameters (ensuring that the multiplications are all smaller than 17×17 bits) that the current heuristic would miss, hence the *hand-tuned* comment in Table III. This is transparent to the FloPoCo user.

We also hand-crafted a correctly rounded version of the single-precision square root, adding the squarer and correction logic described in Section II. One observes that it more than doubles the DSP count and latency for single precision (we were not able to attain the same frequency but we trust it should be possible). For larger precisions, the overhead will be proportionnally smaller, but disproportionnate nevertheless. Indeed, the correctly rounded multiplicative version even consumes more slices than the iterative one, so it only has the advantage of latency.

Another optimization that concerns larger polynomials evaluators is the use of truncated multipliers [20] wherever this may save DSP blocks (and still ensure faithful rounding of course). This is currently being explored, and any improvement in FloPoCo's generic polynomial evaluator will be inherited by the polynomial-based square root.

V. Conclusion and future work

This article discussed the best way to compute a square root on a recent FPGA, trying in particular to make the

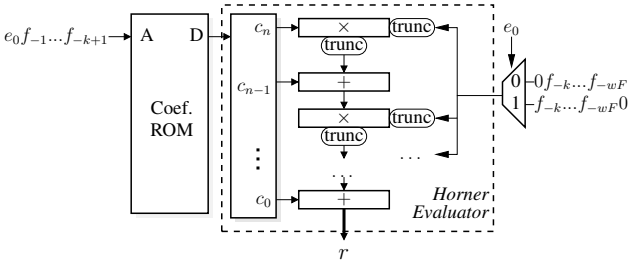

Fig. 1. Generic Polynomial Evaluator

TABLE III
FLOPOCO POLYNOMIAL SQUARE ROOT FOR VIRTEX-4 4VFX100FF1152-12 AND VIRTEX5 XC5VLX30-3-FF324. THE COMMAND LINE USED IS
`flopoco -target=Virtex4|Virtex5 -frequency=`f `FPSqrtPoly` w_E w_F `degree`

	(w_E, w_F)	Degree	cycles	Synth. results
(hand-tuned)	$(8, 23)$	2	5	339 MHz, 79 slices, 2 BRAM, 2 DSP
(hand-tuned, correct rounding)		2	12	237 MHz, 241 slices, 2 BRAM, 5 DSP
FloPoCo, Virtex4, 400 MHz	$(9, 36)$	3	20	318 MHz, 485 slices, 4 BRAM, 11 DSP
	$(10, 42)$	3	20	318 MHz, 525 slices, 7 BRAM, 11 DSP
	$(11, 52)$	3	23	320 MHz, 719 slices, 74 BRAM, 14 DSP
		4	33	318 MHz, 1145 slices, 11 BRAM, 26 DSP
paper and pencil estimation was:		4	*25*	*300 MHz, 20 BRAM, 18 DSP*
FloPoCo, Virtex5, 400 MHz	$(8, 23)$	2	7	419 MHz, 177 LUT, 176 REG, 2 BRAM, 2 DSP
	$(9, 36)$	3	15	376 MHz, 542 LUT, 461 REG, 4 BRAM, 9 DSP
	$(10, 42)$	3	17	364 MHz, 649 LUT, 616 REG, 4 BRAM, 9 DSP
	$(11, 52)$	4	27	334 MHz, 1156 LUT, 1192REG, 6 BRAM, 19 DSP

best use of available embedded multipliers. It compares a state-of-the-art pipelining of the classical digit recurrence, and an original polynomial evaluation algorithm. For large precisions, the latter has the best latency, at the expense of an increase of resource usage. We also observe that the cost of correct rounding with respect to faithful rounding is very large. Fortunately, in the wider context of FloPoCo, a faithful square root is a useful building block for coarser operators, for instance an operator for $\sqrt{x^2+y^2+z^2}$ (based on the sum of square presented in [5]) that would be faithful itself.

Considering the computing power they bring, we found it surprisingly difficult to exploit the embedded multipliers to surpass the classical digit recurrence in terms of latency, performance and resource usage. However, as stated by Langhammer [9], embedded multipliers also bring in other benefits such as predictability in performance and power consumption.

Future works include a careful implementation of a high-radix algorithm, and a similar study around division. Also, the polynomial evaluator that was developed along this work will be used in the near future as a building block for many other elementary functions, up to double precision.

Stepping back, this work asks a wider-ranging question: does it make any sense to invest in function-specific multiplicative algorithms such as the high-radix square root (or the iterative exp and log of [21], or the high-radix versions of Cordic [17], etc)? Or won't a finely tuned polynomial evaluator, computing just right at each step, be just as efficient in all cases?

Acknowledgements: We would like to thank C.-P. Jeannerod for the FLIP library and for insightful discussions on this article. Thanks to M. Daumas for pointing to us the high-radix recurrence algorithm. This work was partly supported by the ANR EVA-Flo project and Stone Ridge Technology.

REFERENCES

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

[2] Y. Li and W. Chu, "Implementation of single precision floating point square root on FPGAs," in *FPGAs for Custom Computing Machines*. IEEE, 1997, pp. 56–65.

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

[4] J. Detrey and F. de Dinechin, "A tool for unbiased comparison between logarithmic and floating-point arithmetic," *Journal of VLSI Signal Processing*, vol. 49, no. 1, pp. 161–175, 2007.

[5] 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.

[6] D. M. Russinoff, "A mechanically checked proof of correctness of the AMD K5 floating point square root microcode," *Formal Methods in System Design*, vol. 14, no. 1, pp. 75–125, 1999.

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

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

[9] M. Langhammer, "Foundation for FPGA acceleration," in *Fourth Annual Reconfigurable Systems Summer Institute*, 2008.

[10] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy, "Faster floating-point square root for integer processors," in *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, 2007.

[11] J. A. Pineiro and J. D. Bruguera, "High-speed double-precision computation of reciprocal, division, square root, and inverse square root," *IEEE Transactions on Computers*, vol. 51, no. 12, pp. 1377–1388, Dec. 2002.

[12] X. Wang, S. Braganza, and M. Leeser, "Advanced components in the variable precision floating-point library." in *FCCM*. IEEE Computer Society, 2006, pp. 249–258.

[13] W. S. Briggs and D. W. Matula, "A 17x69-bit multiply and add unit with redundant binary feedback and single cycle latency," in *11th Symposium on Computer Arithmetic*. IEEE, 1993, pp. 163–170.

[14] W. S. Briggs, T. B. Brightman, and D. W. Matula, "Method and apparatus for performing the square root function using a rectangular aspect ratio multiplier," United States Patent 5,159,566, 1992.

[15] T. Lang and P. Montuschi, "Very high radix square root with prescaling and rounding and a combined division/square root unit," *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 827–841, 1999.

[16] F. de Dinechin, M. Joldes, B. Pasca, and G. Revy, "Multiplicative square root algorithms for FPGAs," LIP 2010-17, Tech. Rep., 2010. [Online]. Available: http://prunel.ccsd.cnrs.fr/ensl-00475779/

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

[18] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 250–255.

[19] 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.

[20] 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.

[21] 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.