



**HAL**  
open science

## Multiplicative square root algorithms for FPGAs

Florent de Dinechin, Mioara Maria Joldes, Bogdan Pasca, Guillaume Revy

► **To cite this version:**

Florent de Dinechin, Mioara Maria Joldes, Bogdan Pasca, Guillaume Revy. Multiplicative square root algorithms for FPGAs. 2010. ensl-00475779v1

**HAL Id: ensl-00475779**

**<https://ens-lyon.hal.science/ensl-00475779v1>**

Preprint submitted on 22 Apr 2010 (v1), last revised 1 Nov 2010 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



*Laboratoire de l'Informatique du Parallélisme*

École Normale Supérieure de Lyon

Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

*Multiplicative square root algorithms for  
FPGAs*

Florent de Dinechin,  
Mioara Joldes,  
Bogdan Pasca,  
Guillaume Revy

April 2010

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

Research Report N° 2010-17

**École Normale Supérieure de Lyon**

46 Allée d'Italie, 69364 Lyon Cedex 07, France

Téléphone : +33(0)4.72.72.80.37

Télécopieur : +33(0)4.72.72.80.80

Adresse électronique : lip@ens-lyon.fr



**INRIA**  
RHÔNE-ALPES



# Multiplicative square root algorithms for FPGAs

Florent de Dinechin, Mioara Joldes, Bogdan Pasca, Guillaume Revy

LIP, Arénaire

CNRS/ENSL/INRIA/UCBL/Université de Lyon

46, allée d'Italie, 69364 Lyon Cedex 07, France

Florent.de.Dinechin@ens-lyon.fr, Mioara.Joldes@ens-lyon.fr, Bogdan.Pasca@ens-lyon.fr, Guillaume.Revy@ens-lyon.fr

April 2010

## 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. It allows a much shorter latency and a higher frequency than the classical approach. The cost of IEEE-compliant correct rounding using such approximation algorithms is shown to be very high, and faithful (last-bit accurate) operators are advocated in this case.

**Keywords:** square-root, FPGA

## 1 Introduction

### 1.1 Algorithms for floating-point square root

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

The first family is that of *digit recurrences*, which provide one digit (often one bit) of the result at each iteration. Each iteration consists of additions and digit-by-number multiplications (which have comparable cost) [9]. Such algorithms have been widely used in microprocessors that didn't include hardware multipliers. Most FPGA implementations in vendor tools or in the literature [15, 13, 7] use this approach, which was the obvious choice for early FPGAs which did not yet include embedded multipliers.

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 [19], in more recent instruction sets such as Power/PowerPC and IA64 (Itanium) whose floating-point unit is built around the fused multiply-and-add [16, 3], and in the INVSQRT core from the Altera MegaWizard. Other variations involve piecewise polynomial approximations [10, 18]. On FPGAs, the VFLOAT project [20] uses an argument reduction based on tables and multipliers, followed by a polynomial evaluation of the reduced argument.

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 [2] 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 root [12]. The first contribution of this article is a detailed survey of the available multiplicative algorithms and their suitability to the FPGA target. A second contribution is an implementation of a promising multiplier-based square root based on polynomial evaluation which is, to our knowledge, original in the context of FPGAs.

The conclusion is that it is surprisingly difficult to really benefit from the embedded multipliers, especially for double precision. A problem is correct rounding (mandated by the IEEE-754 standard) which is shown to require a large final multiplication.

The wider goal of this work is to provide the best possible square root implementations in the FloPoCo project<sup>1</sup>.

### 1.2 Multiplicative algorithms fit recent FPGAs

Let us first review the features of recent FPGAs that can be used for computing square roots.

Embedded multipliers available in recent FPGAs are summed up in the following table.

<i>Family</i>	<i>Multipliers</i>
Virtex II to Virtex-4	18x18 signed or 17x17 unsigned
Virtex-5/Virtex-6	18x25 signed or 17x24 unsigned
Stratix II/III/IV	18x18 signed or unsigned

<sup>1</sup><http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/>

It is possible to build larger multipliers by assembling these embedded multipliers [6]. Besides, these multipliers are embedded in more complex DSP blocks that also include specific adders and shifters – we shall leave it to the design tools to make the best use of these resources.

Memories have a capacity of 9Kbit or 144Kbit (Altera) or 18Kbit (Xilinx) and can be configured in shape, for instance from  $2^{16} \times 1$  to  $2^9 \times 36$  for the Virtex-4.

A given FPGA typically contains a comparable number of memory blocks and multipliers. When designing an algorithm for an operator, it therefore makes sense to try and balance the consumption of these two resources. However, the availability of these resources also depends on the wider context of the application, and it is even better to provide a range of trade-offs between them.

### 1.3 Notations and terminology

In all this paper,  $x$ , the input, is a floating-point number on  $w_F$  bits of mantissa and  $w_E$  bits of exponent. IEEE-754 single precision is  $(w_E, w_F) = (8, 23)$  and double-precision is  $(w_E, w_F) = (11, 52)$ .

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 correspond 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. Noting the (normalized) mantissa  $1.F$  with  $F$  a  $w_F$ -bit number, 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.

However, in the specific case of square root, the overhead of obtaining correct rounding is lower than in the general case. Section 2 shows a general technique to convert 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.

## 2 The cost of correct rounding

For square root, correct rounding may be deduced from faithful rounding thanks the following technique, used in [10]. We first compute a value of the square root  $\tilde{r}$  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,  $\tilde{r}$  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  $\tilde{r}$  entails that it is the correctly rounded square root. In the second case, squaring  $\tilde{r}$  and comparing it to  $x$  tells us (thanks to the monotonicity of the square root) if  $\tilde{r} < \sqrt{x}$  or  $\tilde{r} > \sqrt{x}$  (it can be shown that the case  $\tilde{r} = \sqrt{x}$  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.

We use in this work the following algorithm, which is a simple rewriting of the previous idea.

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

With respect to performance and cost, one may observe that the overhead of correct rounding over faithful rounding on  $w_F$  bits 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 [6], 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, it will make sense in most applications to favor faithful rounding on  $w_F + 1$  bits over correct rounding on  $w_F$  bits (for the same relative accuracy bound).

The FloPoCo implementation should eventually offer both alternatives, but in the following, we only consider faithful implementations for approximation algorithms.

## 3 A survey of square root algorithms

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 (coded on  $w_E$  bits with a bias of  $2^{w_E-1} - 1$ , but this is irrelevant to the present article), 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, the square root is

$$\sqrt{X} = 2^{A/2} \times \sqrt{1.F} .$$

- If  $e$  is odd, the square root is

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

In both cases the computation of the exponent of the result is straightforward, and we will not detail it further. The computation of the square root is reduced to computing  $\sqrt{Z}$  for  $Z \in [1, 4[$ . Let us survey the most relevant square root algorithms for this.

In the following, we evaluate the cost of the algorithms for double-precision (1+52 bits of mantissa) for comparison purpose. We also try, for algorithms designed for the VLSI world, to retarget them to make the best use of the available FPGA multipliers which offer a smaller granularity.

Unless otherwise stated, all the synthesis results in this article are obtained for Virtex-4 4vfx100ff1152-12 using ISE 11.3.

### 3.1 Classical digit recurrences

The general digit recurrence, in radix  $\beta$ , is expressed as follows.

- 1:  $R_0 = X - 1$
- 2: **for**  $j \in \{1..n\}$  **do**
- 3:    $s_{j+1} = \text{Sel}(\beta R_j, S_j)$
- 4:    $R_{j+1} = \beta R_j - 2s_{j+1}S_j - s_{j+1}^2\beta^{-j-1}$
- 5: **end for**

Here Sel is a function that selects the next digit (in radix  $\beta$ ) of the square root, and the radix  $\beta$  controls a trade-off between number of iterations and cost of an iteration. More details can be found in textbooks [9].

Most square roots currently distributed for FPGAs use radix 2, including Xilinx LogiCore FloatingPoint (we compare here to 5.0 as available in ISE 11.3) and Altera MegaWizard, but also libraries such as Lee's [13], and FPLibrary, on which FloPoCo is based [7]. The main reason is that in this case, Sel costs nothing.

Pipelined versions perform one or more iterations per cycle. In Xilinx LogiCore, one may chose the latency: less cycles mean a lower frequency, but also lower resource consumption.

The FloPoCo implementation inputs a target frequency and minimizes the latency for it. Based on an approximate model of the delay of an iteration [5], several iterations will be grouped in a single cycle if this is compatible with this 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, the 25 iterations are grouped as 7 + 5 + 5 + 4 + 4. Table 1 shows the improvements this may bring in terms of slices. It also shows that FloPoCo slightly outperforms LogiCore.

### 3.2 Newton-Raphson

These iterations converge to a root of  $f(y) = 1/y^2 - x$  using the recurrence:

$$y_{n+1} = y_n(3 - xy_n^2)/2. \quad (2)$$

The square root can then be computed by multiplying the result by  $x$ , a  $w_F \times w_F$  multiplication: This is inherently inefficient if one wants to compute  $\sqrt{x}$ . However, considering

Table 1: Pipelining of digit-recurrence square root on a Virtex-4 4vfx100ff1152-12 using ISE 11.3. The command line used is `flopoco -frequency=f FPSqrt w_E w_F`

$(w_E, w_F)$	Tool	cycles	Synth. results
(8, 23)	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.
(11, 52)	LogiCore 28 cycles	28	353 MHz, 464 sl.
	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.

the cost of a division (comparable to that of a square root, and higher than that of a multiplication) it is very efficient in situations when one needs to divide by a square root.

Convergence towards  $1/\sqrt{x}$  is ensured as soon as  $y_0 \in (0, \sqrt{3}/\sqrt{x})$ . It is a quadratic convergence: the number of correct bits in the result doubles at each iteration. Therefore, implementations typically first read an initial approximation  $y_0$  of  $1/\sqrt{x}$  accurate to  $k$  bits, from a table indexed by the (roughly)  $k$  leading bits of  $x$ . This saves the  $k$  first iterations.

Let us now evaluate the cost of a double-precision pipelined implementation on a recent FPGA. A  $2^{14} \times 18$  bits ROM may provide an initial approximation accurate to 14 bits. This costs 16 18Kb memories, or 32 9Kb ones.

Then, two iterations will provide 56 correct bits, enough for faithful rounding. If we try to compute just right, the first iteration may truncate all its intermediate computations to 34 bits, including  $x$ : its result will be accurate to 34 bits only. It thus needs

- 2 multipliers for the 34x17-bit product  $xy_0$
- 2 more to multiply the previous result (truncated to 34 bits) by  $y_0$  to obtain  $xy_0^2$
- 2 more for the last multiplication by  $y_0$

The resulting product  $y_1$  may be truncated to 34 bits, and the second iteration, which needs to be accurate to 56 bits, needs

- 6 multipliers for  $xy_1$  (here we keep the 54 bits of  $x$ )
- 6 more for  $xy_1^2$
- and again 6 for the last multiplication by  $y_1$ .

Altogether, we estimate that approximating faithfully  $1/\sqrt{x}$  costs 24 multipliers, and 16 memory blocks. This is slightly better than the 27 multipliers cited in [12], where, probably,  $x$  was not truncated for the first iteration.

Table 2: Comparison of double-precision square root operators. Numbers in *italics* are estimations.

Algorithm	precision	latency	frequency	slices	DSP	BRAM
FloPoCo digit recurrence	0.5 ulp	53 cycles	307 MHz	1740	0	0
Radix-2 <sup>17</sup> digit recurrence	0.5 ulp	<i>30 cycles</i>	<i>300 MHz</i>	?	<i>23</i>	<i>1</i>
VFLOAT [20]	2.39 ulp	17 cycles	<i>&gt;200 MHz</i>	<i>1572</i>	<i>24</i>	<i>116</i>
Polynomial ( $d = 4$ )	1 ulp	<i>25 cycles</i>	<i>300 MHz</i>	?	<i>18</i>	<i>20</i>
Altera ( $1/\sqrt{x}$ ) [12]	1 ulp?	32 cycles	?	900 ALM	27	<i>32 M9K</i>

### 3.3 High-radix digit recurrences

The Cyrix 83D87 co-processor described in [2] is built around a 17x69-bit multiply-and-add unit that is used iteratively to implement larger multiplications, and also division and square root [1]. It is pure coincidence that one dimension is 17 bits, as in FPGA multipliers, but it makes retargetting this algorithm simpler. Variations of this technique have been published, e.g. [11]. The larger dimension is actually the target precision (64 bits for the double-extended precision supported by this coprocessor), plus a few guard bits. We now note it  $w_F + g$ .

The algorithm uses the same iteration as in Section 3.1, only in radix  $\beta = 2^{17}$ . It first computes a 17-bit approximation  $Y$  to the reciprocal square root  $1/\sqrt{X}$ , using a table lookup followed by one Newton-Raphson iteration in [1]. This would consume very few memories and 3 multipliers, but on a recent FPGA, we could choose instead a large ROM consuming 128 36Kb blocks. Then, iteration  $i$  consists of two  $17 \times (17i)$  multiplications, plus additions and logic. One of the multiplications is the computation of the residual  $R_{j+1}$  as above, and the second implements Sel by *selection by rounding*: the multiplication of the residual by  $Y$ , with suitable truncation, provides the next 17 bits of the square root (i.e. the next radix-2<sup>17</sup> digit  $s_{j+1}$ ).

It should be noted that by maintaining an exact remainder, this technique is able to directly provide a correctly rounded result.

Let us now evaluate the cost of this approach on a recent FPGA. We have already discussed the initial approximation to  $1/\sqrt{X}$ . Then we need 4 iteration for  $i = 1$  to  $i = 4$ , each costing two multiplication of  $17 \times 17i$  bits, or  $2i$  embedded multipliers.

The total cost is therefore 20 multipliers, plus possibly 3 for the initial Newton-Raphson if we choose to conserve memory. On a Virtex-5 or 6, the  $24 \times 17$ -bit multipliers may reduce the cost of some of the 4 iterations. On Altera devices, three iterations may be enough, which would reduce multiplier count to only 12. In both cases, this evaluation has to be validated by an actual implementation.

The latency would be comparable to the multiplier count, as each multiplication depends on the previous one. However, for this cost, we would obtain a correctly rounded result.

### 3.4 Piecewise polynomial approximation

In piecewise polynomial approximation, the memory blocks are used to store polynomial coefficients, and there is a trade-off between many polynomials with smaller degree and fewer

Table 3: FloPoCo polynomial square root for Virtex-4 4vfx100ff1152-12 and Virtex5 xc5v1x30-3-ff324. The command line used is `flopoco -target=Virtex4|Virtex5 -frequency=f FPSqrt-Poly  $w_E$   $w_F$  0 degree`

	$(w_E, w_F)$	Degree	cycles	Synth. results	
<i>handcrafted faithful handcrafted, correct rounding</i>	(8, 23)	2	5	339 MHz, 79 slices, 2 BRAM, 2 DSP	
		2	12	237 MHz, 241 slices, 2 BRAM, 5 DSP	
		2	8	318 MHz, 137 slices, 2 BRAM, 3 DSP	
	(9, 36)	3	20	318 MHz, 485 slices, 4 BRAM, 11 DSP	
	<b>FloPoCo, Virtex4, 400 MHz</b>	(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		
4	25		300 MHz, 20 BRAM, 18 DSP		
<i>estimation was:</i>	(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	
<b>FloPoCo, Virtex5, 400 MHz</b>	(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	

polynomials of larger degree [14, 4].

To evaluate the cost for double precision, we first played with the Sollya tool<sup>2</sup> to obtain polynomials. It was found that an architecture balancing memory and multiplier usage would use 2048 polynomials of degree 4, with coefficients on 12, 23, 34, 45, and 56 bits, for a total of 170 bits per polynomial. These coefficients may be stored in 20 18Kb memories. The reduced argument  $y$  is on 43 bits. The polynomials may be evaluated using the Horner scheme, with a truncation of  $y$  to 17, 34, 34, 51 bits in the respective Horner steps [4], so the corresponding multiplier consumption will be  $1 + 4 + 4 + 9 = 18$  embedded multipliers.

This does only provide faithful rounding. Correct rounding would need one more 54-bit square, which currently costs an additional 9 multipliers.

### 3.5 And the winner is...

Table 2 summarizes the cost and performance of the various contenders. The VFLOAT line is copied from [20], which give results for Virtex-II (frequency is extrapolated for Virtex-4). The ALTFP\_INV\_SQRT component is available in the Altera MegaWizard with Quartus 9.1, but its results are inconsistent with [12] (and much worse). This is being investigated.

According to these estimations, the best multiplicative approach seems be the high radix one. However, it will require a lot of work to implement properly, and before we have attempted it, we may not be completely sure that no hidden cost was forgotten. Besides, this work will be specific to the square root.

The next best is polynomial approach, and it has several features which make it a better choice from the point of view of FloPoCo. First, it doesn't seem far behind in terms of performance and resource consumption. Second, we develop a universal polynomial approximator in FloPoCo [4] which will enable the quick development of many elementary functions to arbitrary precision. Working on the square root helps us refine this approximator. More importantly, improvements in this approximator (e.g. to improve performance, or to adapt it to newer FPGA targets) will immediately reflect to the square root operator, so the polynomial approach is more future-proof. We therefore choose to explore this approach in more details in the next section.

<sup>2</sup><http://sollya.gforge.inria.fr/>

## 4 Square root by polynomial approximation

As stated earlier, we address the problem of computing  $\sqrt{Z}$  for  $Z \in [1, 4[$ . We are classically [14] 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 [4] or in the open-source FloPoCo code itself. Let us focus here on specific optimizations related to the square root.

A first idea is to address the coefficient table is to use the most significant bits of  $Z$ . However, as  $Z \in [1, 4[$ , the value `00xxx` is 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  $\tau_{\text{even}}$ .  $[0, 1[$  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 of degree  $d$ :  $p_i(y) = c_{0,i} + c_{1,i}y + \cdots + c_{d,i}y^d$ .
- 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 follow. Let  $r$  be the value computed by the pipeline before the final rounding. It is represented on  $w_F + g$  bits, the max of the size of all the  $c_0$  and the size of the truncated  $y(c_1 + \cdots)$ .

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 stored, instead of each constant coefficient  $c_0$ , the value  $c_0 + 2^{-w_F-1}$  (we use  $\lfloor z \rfloor = \lfloor z + 1/2 \rfloor$ ).

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:

$y = 2^{-k+2} \times 0, 0f_{-k} \cdots f_{-w_F}$  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. The remainder of the evaluation is described in [4].

## 5 Results, comparisons, and some handcrafting

Table 3 summarizes the actual performance obtained from the polynomial square root at the time of writing (the reader is invited to try it out from the FloPoCo SVN repository). All these operators have been tested for faithful rounding, using FloPoCo's testbench generation framework [5].

The polynomials are obtained completely automatically using the polynomial evaluator generator [4], and we still believe that there is some room for improvement. In particular, the heuristics that define the coefficient sizes and the widths of the intermediate data do not yet fully integrate the staircase effects in the costs, due to the discrete sizes of the multipliers and of the embedded memories. For illustration, compare the two first lines of Table 3. The first was obtained one year ago, as we started this work by designing by hand a single-precision square root using a degree-2 polynomial. In this context, it was an obvious design choice to ensure that both multiplications were smaller than  $17 \times 17$  bits. Our current heuristic misses this design choice, and consumes one DSP more, without even saving on the BRAM consumption. For similar reasons, the actual synthesis result differs from our estimated cost, although the overall cost (BRAM+DSP) is similar.

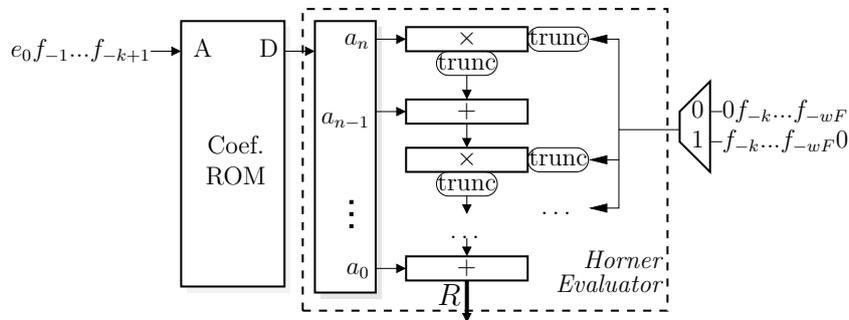


Figure 1: Generic Polynomial Evaluator

We also hand-crafted a correctly rounded version of the single-precision square root, adding the squarer and correction logic described in Section 2. 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 proportionally smaller, but disproportionate nevertheless. Consider also that the correctly rounded multiplicative version even consumes more slices than the iterative one. Indeed, it only has the advantage of latency.

Another optimization that concerns larger polynomials evaluators is the use of truncated multipliers wherever this may save DSP blocks (and still ensure faithful rounding of course). This is currently being explored. As we already mentioned, this optimization will benefit the square root, but also all the other functions that we are going to build around the generic polynomial generator.

## 6 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 best use of available embedded multipliers. It evaluates several possible multiplicative algorithms, and eventually 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 quite large, and therefore suggest sticking to faithful rounding. 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 [12], 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. 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 [8], 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? The answer seems to be yes for software implementations of elementary functions [16, 3], but FPGA have smaller multiplier granularity, and logic.

## Acknowledgements

We would like to thank Claude-Pierre Jeannerod for FLIP and for insightful discussions on this article. Thanks to Marc Daumas for pointing to us the high-radix recurrence algorithm.

## References

- [1] 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.
- [2] 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*, pages 163–170. IEEE, 1993.
- [3] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium<sup>®</sup>-based Systems*. Intel Press, 2002.
- [4] F. de Dinechin, M. Joldes, and B. Pasca. Automatic generation of polynomial-based hardware architectures for function evaluation. LIP Research Report 2010-14, 2010.
- [5] F. de Dinechin, C. Klein, and B. Pasca. Generating high-performance custom floating-point pipelines. In *Field Programmable Logic and Applications*, pages 59–64. IEEE, Aug. 2009.
- [6] F. de Dinechin and B. Pasca. Large multipliers with fewer DSP blocks. In *Field Programmable Logic and Applications*, pages 250–255. IEEE, Aug. 2009.
- [7] J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *Journal of VLSI Signal Processing*, 49(1):161–175, 2007.
- [8] J. Detrey, F. de Dinechin, and X. Pujol. Return of the hardware floating-point elementary function. In *18th Symposium on Computer Arithmetic*, pages 161–168. IEEE, 2007.
- [9] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.
- [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] 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*, 48(8):827–841, 1999.
- [12] M. Langhammer. Foundation for FPGA acceleration. In *Fourth Annual Reconfigurable Systems Summer Institute*, 2008.
- [13] B. Lee and N. Burgess. Parameterisable floating-point operators on FPGAs. In *36th Asilomar Conference on Signals, Systems, and Computers*, pages 1064–1068, 2002.
- [14] D. Lee, A. Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Transactions on Computers*, 54(12):1520–1531, 2005.
- [15] Y. Li and W. Chu. Implementation of single precision floating point square root on FPGAs. In *FPGAs for Custom Computing Machines*, pages 56–65. IEEE, 1997.
- [16] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.

- [17] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, 2nd edition, 2006.
- [18] 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*, 51(12):1377–1388, Dec. 2002.
- [19] D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, 1999.
- [20] X. Wang, S. Braganza, and M. Leiser. Advanced components in the variable precision floating-point library. In *FCCM*, pages 249–258. IEEE Computer Society, 2006.