Fixed-Point Implementations of the Reciprocal, Square Root and Reciprocal Square Root Functions

Matei Istoan, Bogdan Pasca

Fixed-Point Implementations of the Reciprocal, Square Root and Reciprocal Square Root Functions

Matei Iştoan
Université de Lyon, INRIA,
INSA-Lyon, CITI, F-69621 Villeurbanne, France

Bogdan Pasca
Altera European Technology Centre,
High Wycombe, UK

***Abstract*—Implementations of the reciprocal, square root and reciprocal square root often share a common structure. This article is a survey and comparison of methods (with only slight variations for the three cases) for computing these functions. The comparisons are made in the context of the same accuracy target (faithful rounding) and of an arbitrary fixed-point format for the inputs and outputs (precisions of up to 32 bits). Some of the methods discussed might require some form of range reduction, depending on the input's range. The objective of the article is to optimize the use of fixed-size FPGA resources (block multipliers and block RAMs). The discussions and conclusions are based on synthesis results for FPGAs. They try to suggest the best method to compute the previously mentioned fixed-point functions on a FPGA, given the input precision. This work compares classical methods (direct tabulation, multipartite tables, piecewise polynomials, Taylor-based polynomials, Newton-Raphson iterations). It also studies methods that are novel in this context: the Halley method and, more generally, the Householder method.**

I. Introduction

FPGAs are becoming increasingly large and complex, with recent devices embedding computational features such as single-precision floating-point operators [1]. If two decades ago researchers were customizing floating-point arithmetic in order for single operators to fit entire devices [2], the thousands of embedded floating-point adders and multipliers available in the Altera Arria10 devices make possible the rapid construction of highly optimized datapaths which run at FPGA nominal frequencies. Floating-point datapaths can automatically cope with high dynamic data ranges, shortening design time often at the expense of performance. Fixed-point versions of addition and multiplication have traditionally been implemented in FPGAs using dedicated circuitry. Embedded ripple carry adder cells and DSP blocks containing small embedded multipliers allow fixed-point datapaths to provide small implementation footprints and shorter latency. The pitfall when choosing fixed-point arithmetic is that the user needs to manage data ranges manually – a task which requires good algorithm understanding – often regarded as a rare skill.

In a fixed-point datapath, well trimmed adders and multipliers will have efficient implementations. On the other hand, fixed-point function computations (such as $\sqrt{x}, 1/\sqrt{x}, 1/x$) are often overlooked in terms of resource requirements. Datapaths embedding a mix of these resources need efficient fixed-point function implementations. This is particularly important since the properties of floating-point arithmetic allows for very efficient implementations of these functions, which may overweigh the advantages of fixed-point adders and multipliers.

This paper focuses on the fixed-point implementation of these functions, for bit-widths ranging from few bits up to 32 bits. The implementations are flexible in terms of input and output formats allowing for fine-grain operator customization. Input range can be restricted to the interval $[1,2)$, or may span the whole input range allowed by the input format. Multiple implementation techniques from the literature are revisited and employed in this generic context, from tabulation, multipartite methods, Taylor series, Newton-Raphson, and higher order methods. A generator which selects the suitable architectures for user-specified constraints was constructed for this paper.

Due to the double-blind review process we are not describing the full details of this generator in this submission.

II. FPGA features

In this section we briefly present FPGA architectural features relevant to this work. We restrict our study to using Altera FPGA architectures, although the generic nature of the core generator of this paper is actually vendor agnostic.

Out of the 3 FPGA families from Altera we focus on the low-cost CycloneV [3] devices. In a nutshell, FPGA devices are composed of millions of small tables (4-6 inputs) which can be programmed to perform any logic function (of 4-6 inputs). These small tables are connected using a flexible, programmable interconnect network which allows interconnecting any two nodes in the FPGA. In Altera devices, these table based resources are encapsulated in Adaptive Logic Modules (ALMs). Each ALM can be configured as 2 independent 4-input lookup tables (LUT), one 6-input LUT, and many combinations of functions which share inputs. Logic resources are usually reported in terms of ALMs in Altera devices.

Contemporary FPGA devices also contains hundreds (often thousands - depending on the device) small flexible multipliers (packaged in DSP blocks) and memory blocks. The DSP blocks in the CycloneV device can be configured in the modes presented in Table I.

Dedicated memory blocks in CycloneV devices have a capacity of 10Kb each (M10K). The relevant configurations for this works are: 2048x5bits, 1024x10bits, 512x20bits, 256x40bits.

TABLE I
SUBSET OF DSP-BLOCK FEATURES FOR CYCLONEV DEVICES

Blocks	Mode	CycloneV
1	9×9	3
	16×16	2
	$18\times 18\rightarrow 32$	2
	18×18	2
	18×19	2
	27×27	1
	2 18×18 MADD	1
	2 18×19 MADD	1
	$18\times 18+36$-bit	1
2	2 27×27 MADD	1

III. BACKGROUND

A. Choosing the Right Method

In order to chose the best method for implementing the functions studied in this paper, the criteria on which to judge the different methods first have to be decided. Having multiple criteria has the disadvantage of complicating the selection process, but it also has the advantage of allowing the user to make the right choice on each separate scenario.

The methods presented in the following are differentiated by three main aspects. This short list of criteria is in no way exclusive, but it highlights some of the common design challenges that users are faced with. It also highlights the constant quest for a better use of *all* the available hardware resources on FPGAs.

Bitwidth: computing just right in terms of precision, means performing meaningful computations and ensuring the target output precision, as well as using a minimal amount of resources. Computing for a specific FPGA target means getting the most out of what the FPGA has to offer. Choosing an implementation method based on the targeted bitwidth can ensure satisfying both points previously made.

Input range: making no assumptions about the inputs can make a design as general as possible. It can also impose additional constraints on the method used (and even make it unsuitable) and incur the use of additional resources. Choosing an implementation while having knowledge about the input ranges allows more specialized solutions.

Approach: table-based, polynomial approximation and iterative methods are the most common classes of methods used for implementing the functions surveyed in this paper. We have provided hybrids between some of these classes. The choice between these approaches is influenced by the previous two criteria, as well as by cost (resource/speed-wise) and the flexibility each method offers.

Before moving on to classifying the different methods into classes, based on the previously mentioned criteria, one important point has to be made. While this article is interested in fixed-point (FxP) implementations, most of the relevant literature deals with floating-point (FP) implementations. This means that previous findings (relevant in FP), may no longer be valid, due to the limited convergence domains of the methods, the need for range reduction, or due to the specificities of the different target platform.

Another significant difference (from most of the existing works) which may bias the findings in this article is allowing the user to specify the input fixed-point format. This is another aspect that is less common in the literature and which justifies to a certain measure the choices made regarding the methods described in the rest of the article.

B. Classes of methods based on input range

Reducing the range of the inputs is used in the methods described throughout the article not just to speed up their convergence, but more critically in order to ensure convergence. Methods initially designed for FP representations require the input to be in the interval $[1,2)$. This is a requirement that no longer holds for this study, due to the fact that the user is allowed to specify the FxP format of the inputs.

The methods can, therefore, be devised into two separate categories. The first are methods with *reduced range* inputs, with an implicit assumption that the inputs are in $[1,2)$. This category of methods requires a scaling of the input, in order to work with all the values of the user specified format. The second category is composed of methods that accept the *full range* of values of the inputs. They do not require scaling and can be used as is.

With only one exception (the case of the multipartite method), the same procedure can be used for scaling the inputs (and re-scaling the outputs at the end), for all the reduced range methods. Details of this procedure are given in what follows.

The range reduction is a classical one and consists of performing the substitution:

$$x = 2^s \cdot x'$$

where x is the input to the algorithm and $x' \in [1,2)$. The value of s is chosen in such a way so that the constraints on x' are satisfied. The methods presented in the next sections will take as input x'.

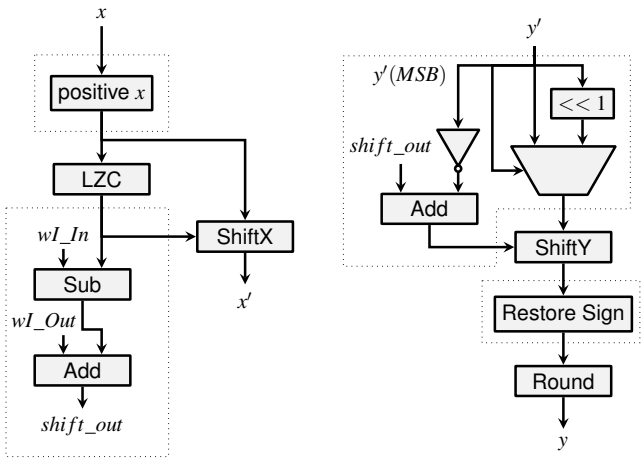
For the post-processing phase of the range reduction, a similar substitution is applied, which results in the following set of equations:

$$\begin{cases} y = 2^{-s} \cdot y' \\ y = 2^{-\frac{s}{2}} \cdot y' \\ y = 2^{\frac{s}{2}} \cdot y' \end{cases}$$

where y is the output of the algorithm and y' represents $\frac{1}{x'}$, $\frac{1}{\sqrt{x'}}$ and $\sqrt{x'}$ respectively, with the semantic of x' previously defined.

Figures 1 (a) and (b) present a schematic of the pre-processing and post-processing phases of the range reduction. The dotted rectangle on the top of Figure 1(a) is needed only for computing the reciprocal, and simplifies the overall architecture. The one in the bottom part of the figure computes the shift amount for the post-processing phase (it is included in this figure as it is more relevant in this context). For the post-processing phase, the dotted rectangle represents logic that is used to align y', and adjust the shift amount accordingly, so that the output is correctly interpreted as a 2's complement number. The block labeled 'Restore Sign' is only needed for

the reciprocal, and performs opposite function of the top dotted block in Figure 1(a). When needed, restoring the sign and rounding are combined into a single block.



(a) Pre-Scaling (b) Post-Scaling

Fig. 1. Scaling range reduction

C. Classes of methods based on approach

While the methods listed in this classification are based on the approach they take on computing the target functions, they have also been selected with the *bitwidth* of the input in mind. So, the possible approaches chosen are the ones which were relevant to the targeted precisions. Even so, and while not being exhaustive, the listing here covers the most used methods.

One of the main components making up contemporary FPGAs are look-up tables (LUT). This makes *table-based methods* an interesting and omnipresent approach. The most straightforward method is direct tabulation. It is suitable for bitwidths up to the size of the number of inputs of the LUT elements (also referred to as logic), or of the block RAM elements. The former is usually 6, while the latter is 10, or 11. The correctly rounded results can be stored directly in the table. Overflow and underflow detection can also be encoded into the stored values, without any additional resources needed.

One drawback of direct tabulation is that the resource requirement increase is exponential in the number of bits to address the table. Several methods try to overcome this shortcoming with as little extra resources as possible. These methods include the ATA method [4], iATA [5], the bipartite method [6], SBTM [7], STAM [8] and the generalized multipartite method [9], [10], [11]. More details about these methods are presented in Section IV.

As the input size increases, it becomes impractical to rely solely on tabulation for implementing the target functions, even when considering the advantages brought by the multipartite methods. Another classical approach are *polynomial approximation-based methods*. The polynomial itself can be obtained by either using a *Taylor series* to approximate the function, or by using a *generic polynomial approximation* (like for example a minimax polynomial). Some relevant examples for the first approach would be [12], [13] for smaller bitwidths (discussed in Section VI-A) or [14] for larger bitwidths (discussed in Section VII-A).

For approximating the target functions on larger bitwidths, an alternative approach to using a polynomial approximation is to use *iterative methods*. The most popular of these methods is the *Newton-Raphson method* (Section VII-B), a zero-finding method which doubles the precision of the initial guess at each iteration. To obtain an even more precise result at each iteration, there are higher order methods, with *Halley's method* being the next in the series after Newton-Raphson. The generalized order n version is known as the *Householder method* [15]. More details on the iterative methods are shown in Section VII-C.

IV. Multipartite Methods

The bipartite method is a well known architecture for approximating functions using only tables and no multipliers. It approximates the function using affine segments. The bipartite method can be thought of as starting with a degree 1 Taylor series in the point A:

$$f(x) = f(A) + f'(A)(x-A)$$

The input interval is split into a power of 2 number of subintervals. A is selected as an easy to compute point in each interval; A can be the top k bits of x, but a better value for the approximation is obtained if the middle of each interval is selected.

The second term is indexed by the remaining bits of x that we denote by $B = x - A$ and the derivative of the function in A, The multiplication is avoided in the bipartite method by performing a worst approximation of the derivative. Instead of storing the slope on each subinterval, one slope is stored for a larger number of subintervals:

$$f'(A) \approx f'(C),$$

where C is indexed by only a few bits of A. By having this rough approximation of the slope, the entire right hand term can be tabulated and indexed by B and C. There are several optimizations regarding symmetry in this table, and the interested reader should consult [8], [10].

V. Approximating on the full range

In this section we will be focusing the discussions on the reciprocal function. In the bipartite architecture, the approximation quality decreases as the derivative increases. The rough approximation of the slope, especially as the input approaches zero, where the reciprocal has the asymptotic behaviour, makes the bipartite method a bad starting point for a full range architecture.

Therefore, we start-off with a degree-1 Taylor polynomial for the full-range approximation architecture. As expected, our experiments using MPFR have confirmed that the accuracy of the 1d Taylor is vastly superior to the bipartite method. Our

proposed architecture will use a fine-tuned Taylor approximation for most of the input range, then use a tabulation technique for the range where the error exceeds the maximum admitted error, knowing that the total error is less than 1ulp.

It is well known that the accuracy of Taylor polynomials is significantly less than that of polynomials returned by the Remez algorithm. Open source tool Sollya [16] provides probably the best polynomial approximations by means of the FPMinimax algorithm. We improved the accuracy of the Taylor polynomials by finding the most accurate point in a 2 dimensional space where the two dimensions were given by ulp movement of $f(A)$ and $f'(A)$. Please note that we selected A to be the exact midpoint of each segment. The error improvement decreases the number of points which require tabulation. Since FPGA memory blocks are monolithic, they have hard thresholds. For instance lowering the number of inputs tabulated by only a few can decrease the total memory impact by half.

Depending on the input format, a large number of inputs will produce outputs which exceed the output range maximum value. For these inputs we choose to saturate the output. Additionally, a number of outputs will be in the representable range but the method error will exceed the admitted value. A tabulation method is used for these inputs. The tabulation architecture may be simple or composed of a base value and an offset (in a similar way to [11]), as depicted in Figure 2.

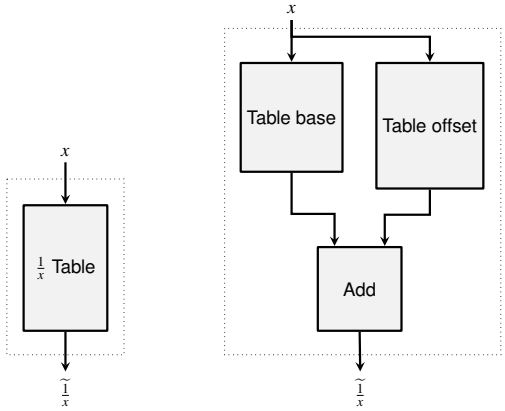


Fig. 2. The two architectures for the tabulation of outputs (method error exceeds the admitted value)

The resources can be lowered by sometimes tabulating the entire range: saturate range + error too large, if the total count of these inputs is less than the memory block threshold. In such case the tabulation comes for free.

The table output is equal to the output width of the architecture. If the number of table elements is large, the number of memory blocks required for implementing this table can increase significantly. Since this table stores the function value for consecutive inputs, the variation between consecutive elements in the table can be quite small. Consequently, in some cases a more efficient architecture can be obtained by sampling the input table and storing the offset values from the sample values in the offset table. Using a simple addition we can exactly reconstruct the content of the initial table, and can decrease memory impact significantly. Determining if the base+offset architecture is more efficient than simple tabulation depends on the input/output format and target FPGA memory block architecture. For one such format a design-exploration stage assesses the memory impact of several decompositions: base+offset. The best architecture overall is selected by first selecting the best candidate among the base+offset architecture and the simple table-only architecture.

The output of the architecture may also underflow for a large number of inputs, also depending on the input/output format. One example is the input/output format with 16-bits of precision (unsigned) and 4 bits of fraction. For this format a separate underflow architecture is used.

The underflow architecture will use tabulation for values which are neither overflow or underflow. It has two implementation options: 1/ tabulation is used for all values which do not underflow (return 0) and 2/ tabulation is used for values which are different to 1ulp or 0. A design-space exploration phase determines the input indices for which values larger will return 1ulp ($idx1ulp$), and for which it would return 0 ($idx0ulp$). Based on these 2 values the possible architectures are explored and the one requiring the fewest memory blocks is selected. If the architectures return the same number of memory blocks, then 1/ is selected since the logic implementation is less. Figure 3 depicts the two architectures of used underflow heavy formats.

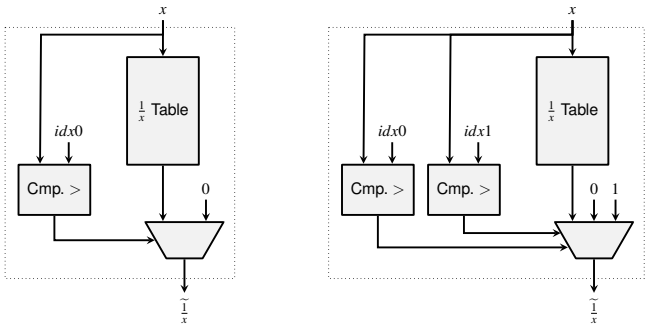


Fig. 3. The two architectures of the underflow architecture

The results section presents the results of all these optimizations across an important number of input/output formats.

VI. Tabulate-and-Multiply Method

A. First Order

One of the shortcomings of the multipartite method is the exponential growth of the memory resources required for tabulation, as the input width increases.

The method introduced in [12] tries to reduce the memory requirements. It is a linear polynomial approximation of x^P (for $P = \pm 2^k$, k integer), and is based on the Taylor series expansion.

The input x is split into two parts, $x = x_1 + x_2$, with $1 \leq x_1 < 2$ and $0 \leq x_2 < 2^{-m}$. The function x^P is then approximated

around the point $x = x_1 + 2^{-m-1}$, with the first two terms of the series:

$$x^P \approx \quad (x_1 + 2^{-m-1})^P + (x_2 - 2^{-m-1}) \cdot P \cdot (x_1 + 2^{-m-1})^{P-1}$$

Taking $(x_1 + 2^{-m-1})^{P-1}$ as a factor we obtain:

$$x^P \approx (x_1 + 2^{-m-1})^{P-1} \cdot (x_1 + 2^{-m-1} + P \cdot (x_2 - 2^{-m-1})) \quad (1)$$

Keeping the notations of [12], we denote

$$C = (x_1 + 2^{-m-1})^{P-1}$$

and

$$x' = x_1 + 2^{-m-1} + P \cdot x_2 - P \cdot 2^{-m-1}$$

The first term, C, can be obtained from a table addressed by x_1. As for x', it can be obtained by using bit manipulations, or relatively simple operations, on x_2 and x_1. Thus, equation (1) can be re-written as:

$$x^P \;\approx\; C \cdot x' \quad (2)$$

and requires a multiplication, aside from the operations already mentioned.

In terms of the accuracy of the approximation, the error is due to the omission of the higher order terms in the Taylor series, and can be roughly approximated as:

$$\varepsilon_{method} \;\approx\; P(P-1)(x_1 + 2^{-m-1})^{P-2} \cdot \frac{1}{2}(x_2 - 2^{-m-1})^2 \quad (3)$$

Which means that $\varepsilon_{method} \approx 2^{-2m-3+\log|P(P-1)|}$. The method error ε_{method} can be slightly improved by replacing C in eq. (2) by $C' = C + P(P-1) \cdot x_1^{P-3} \cdot 2^{-2m-4}$, thus eq. (2) becomes:

$$x^P \;\approx\; C' \cdot x' \quad (4)$$

The second term in the definition of C' try to compensate for the excluded terms in the Taylor series, as shown in [12].

A possible architecture for the method presented in this section is shown in Figure 4.

For the reciprocal, square root and reciprocal square root the 'x Modify' block consists mainly of inverting the bits of x_2 and concatenating them in a given order.

In terms of the implementation, the C' table requires a $2^m \times (w+1)$-bit table (with w the input bitwidth and $m \approx \frac{w}{2}$). The multiplication is of size $(w+1) \times w$, and a truncated multiplier can be used so that the final result is obtained on $w+1$ bits. Obtaining x' requires $\approx w - m$ LUTs, and a w-bit addition is required for the final rounding. This all makes for a quite compact architecture.

B. Second Order

The method presented in Section VI-A is based on the Taylor series, but only utilizes the first two terms of the development. As mentioned in the paragraphs discussing the method's error, the main source for the errors are the terms that are left out of the development. In what follows an extension of the method of Section VI-A is introduced, which makes use of an extra term of the Taylor series, in order to improve the accuracy of the estimation.

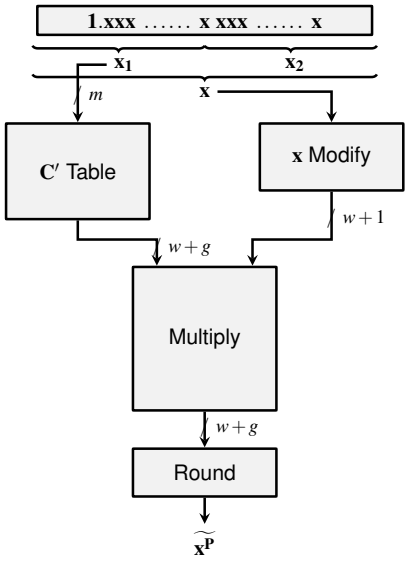


Fig. 4. Tabulate-and-Multiply Method Architecture

The input is split in the same way as before, $x = x_1 + x_2$, with $1 \le x_1 < 2$ and $0 \le x_2 < 2^{-m}$. The function x^P is again approximated around the point $x = x_1 + 2^{-m-1}$:

$$\begin{aligned} x^P \;\approx\; & (x_1 + 2^{-m-1})^P + (x_2 - 2^{-m-1}) \cdot P \cdot (x_1 + 2^{-m-1})^{P-1} \\ & + \frac{1}{2}(x_2 - 2^{-m-1})^2 \cdot P \cdot (P-1) \cdot (x_1 + 2^{-m-1})^{P-2} \end{aligned}$$

Taking again $(x_1 + 2^{-m-1})^{P-1}$ as a factor we obtain:

$$\begin{aligned} x^P \;\approx\; & (x_1 + 2^{-m-1})^{P-2} \cdot [(x_1 + 2^{-m-1})^2 \\ & + P(x_2 - 2^{-m-1})(x_1 + 2^{-m-1}) + (x_2 - 2^{-m-1})^2 \frac{P(P-2)}{2}] \end{aligned}$$

For the sake of brevity and clarity, the rest of the section will deal with the case of the reciprocal (i.e. $P = -1$). The equations for the sqrt ($P = \frac{1}{2}$) and reciprocal sqrt ($P = -\frac{1}{2}$) can be derived in a similar manner, and only incur different alignments for the terms, or differences in the terms stored in the tables. The previous equation thus becomes:

$$\begin{aligned} x^{-1} \;\approx\; & (x_1 + 2^{-m-1})^{-3} \cdot [(x_1 + 2^{-m-1})^2 \\ & + (x_2 - 2^{-m-1})(x_1 + 2^{-m-1}) + (x_2 - 2^{-m-1})^2] \end{aligned} \quad (5)$$

By regrouping the terms, eq. (5) can be written equivalently as:

$$\begin{aligned} x^{-1} \approx \quad & (x_1 + 2^{-m-1})^{-3} \cdot [(x_1 + 2^{-m} - x_2)^2 + \\ & + (x_2 - 2^{-m-1})(x_1 + 2^{-m-1})] \end{aligned} \quad (6)$$

Using the same style of notations as in [12], we denote

$$D = (x_1 + 2^{-m-1})^{-3}$$
$$x' = x_1 + 2^{-m} - x_2$$
$$E = x_1 + 2^{-m-1}$$
$$F = x_2 - 2^{-m-1}$$

And eq. (6) becomes:

$$x^{-1} \approx D \cdot [(x')^2 + E \cdot F] \tag{7}$$

The D term can be read from a table addressed by x_1. The term E can be obtained by simply concatenating a bit of value 1 after the least significant bit (LSB) of x_1, so it does not incur any cost in hardware. The term F can be obtained by flipping the value of the most significant bit (MSB) of x_2, and storing the new sign of F. The term x' can also be obtained by using bit manipulations, in this case it requires inverting the bits of x_2 and concatenating them to x_1. One squaring and two additions are also required, except for the operations already mentioned.

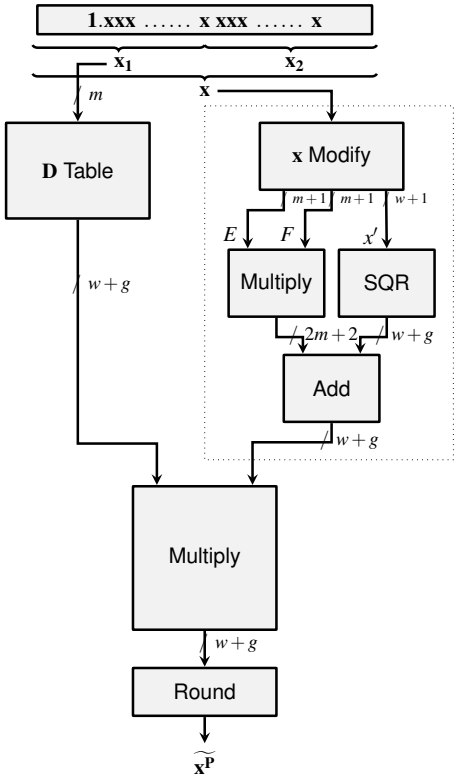


Fig. 5. Second Order Tabulate-and-Multiply Method Architecture

Depending on the trade-offs that the user is willing to make for the operations (or on the available hardware resources), equation (6) can be re-written as:

$$\begin{aligned} x^{-1} \approx\ & (x_1 + 2^{-m-1})^{-3} \cdot [(x_1 + 2^{-m-1})^2 \\ & - (x_2 - 2^{-m-1}) \cdot (x_1 + 2^{-m} - x_2)] \end{aligned} \tag{8}$$

Using the notations:

$$G = (x_1 + 2^{-m-1})^2$$

Equation (8) becomes:

$$x^{-1} \approx D \cdot [G - F \cdot x'] \tag{9}$$

Compared to eq. (7), the method based on eq. (9) only requires two multiplications. On the other hand, it requires storing two coefficients in tables, as opposed to a single one. The terms D and G can be stored in a table addressed by x_1, while F and x' can be obtained by bit manipulations, as mentioned in the previous paragraphs.

The error of the method presented in this section (when computations are performed with infinite accuracy) is, as mentioned in the beginning of the section, due to the ignored terms in the Taylor series development of x^P. Thus, it can be roughly approximated as:

$$\varepsilon_{method} \approx P(P-1)(P-2)(x_1 + 2^{-m-1})^{-4} \cdot \frac{1}{6}(x_2 - 2^{-m-1})^3 \tag{10}$$

Which means that $\varepsilon_{method} \approx 2^{-3m-5+\log|P(P-1)(P-2)/6|}$ (with values of P of -1, $\frac{1}{2}$ and $-\frac{1}{2}$ for the reciprocal, sqrt and reciprocal sqrt respectively). Compared to the method of Section VI-A which offers a result that is correct to approximately $2m$ bits, this method offers a result that is correct to approximately $3m$ bits. A possible architecture for implementing this method, based on equation (7) is shown in Figure 5. The dotted rectangle represents the part of the architecture which is different from equation (9). Figure 6 presents the equivalent part for the dotted rectangle of Figure 5, in order to have a working architecture based on equation (9).

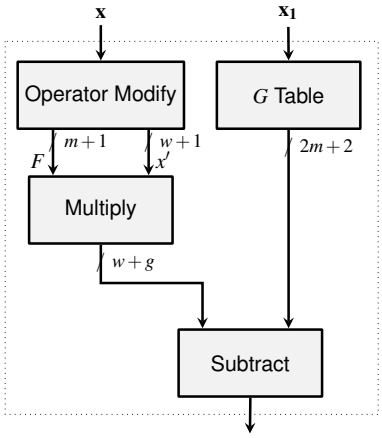


Fig. 6. Alternate Second Order Tabulate-and-Multiply Method Architecture

In terms of the implementation, the squaring of x' and the final multiplication need not be performed on the whole precision of their inputs. Truncating these operations, on the other hand, entails rounding errors, which may prevent from reaching the target output precision (faithfully rounded result). So, a larger intermediary precision is used, adding a number of g extra guard bits to the datapaths.

The total error budget (ε_{total}) is divided as follows: $\varepsilon_{total} = \varepsilon_{final\_round} + \varepsilon_{method} + \varepsilon_{round}$, with a final goal of $\varepsilon_{total} < 2^{-w}$. Half of an ulp (unit in the las place) is allocated to the final rounding ($\varepsilon_{final\_round}$). The rest is divided between the method error (ε_{method}) and the rounding errors (ε_{round}), which are due to the approximations performed during the computations. Determining ε_{method} will give the value that

needs to be used for m, while determining ε_{round} will determine the value of g and the widths of the datapaths.

In order to determine ε_{round}, eq. (7) is re-written to reflect the approximations:

$$x^{-1} \approx \widetilde{D} \cdot [\widetilde{(x')^2} + E \cdot F]$$

and can be further re-written as:

$$x^{-1} \approx (D + \varepsilon_{table}) \cdot [(x')^2 + \varepsilon_{sqr} + E \cdot F]$$

where the tilde terms are the approximations that are obtained from the table and the squarer, respectively, and the ε are the corresponding errors. Developing the equation results in:

$$\begin{aligned} x^{-1} &\approx D \cdot [(x')^2 + E \cdot F] + \varepsilon_{table}[(x')^2 + F \cdot E] \\ &\quad + \varepsilon_{sqr}(D + \varepsilon_{table}) + \varepsilon_{mult} \end{aligned}$$

from which ε_{round} can be identified as:

$$\varepsilon_{round} = \varepsilon_{table}[(x')^2 + F \cdot E] + \varepsilon_{sqr}(D + \varepsilon_{table}) + \varepsilon_{mult}$$

Using the upper bounds for the expressions $(x')^+ E \cdot F$ and D results in:

$$\varepsilon_{round} \leq 4\varepsilon_{table} + \varepsilon_{sqr} + \varepsilon_{sqr}\varepsilon_{table} + \varepsilon_{mult} \tag{11}$$

If the same internal precision is used for all the operations, meaning that $\varepsilon_{table} = \varepsilon_{mult} = \varepsilon_{sqr} = 2^{-w-g}$, and keeping in mind that $\varepsilon_{round} < 2^{-w-2}$, the following expression is found for g:

$$g > 2 + \log(6 + 2^{-w-g})$$

This is, however, a pessimistic value, and testing has shown that lower values of g might also suffice. It is also confirmed by considering individual precisions for the datapaths (given by the ε terms of eq. (11)).

If eq. (9) is used for implementing the method, a similar approach can be used in order to determine ε_{round}. As previously, eq. (9) is then re-written as:

$$x^{-1} \approx \widetilde{D} \cdot [G - \widetilde{F \cdot x'}]$$

and then as:

$$x^{-1} \approx (D + \varepsilon_{table}) \cdot [G - F \cdot x' + \varepsilon_{mult1}] + \varepsilon_{mult2}$$

and finally as:

$$\begin{aligned} x^{-1} &\approx D \cdot [G - F \cdot x'] + G \cdot \varepsilon_{table} - F \cdot x' \cdot \varepsilon_{table} \\ &\quad + D \cdot \varepsilon_{mult1} + \varepsilon_{mult1} \cdot \varepsilon_{table} + \varepsilon_{mult2} \end{aligned}$$

from which ε_{round} can be identified as:

$$\begin{aligned} \varepsilon_{round} &= G \cdot \varepsilon_{table} - F \cdot x' \cdot \varepsilon_{table} + D \cdot \varepsilon_{mult1} \\ &\quad + \varepsilon_{mult1} \cdot \varepsilon_{table} + \varepsilon_{mult2} \end{aligned}$$

Using the same reasoning and the same constraints as before, a relation is found for g, as well:

$$g = 2 + \log(4 - 2^{-m} - 1 + 2^{-w-g})$$

The same remarks can be made for the slightly pessimistic relations found for g here, as the ones made for the previous architecture, in the previous paragraphs.

The methods presented in Section VI can tackle, from a practical point of view, precisions of up to about 24 bits.

VII. Large Precision Methods

A. Taylor-based Method

The methods introduced in Section VI rely on first and second order approximations using the Taylor series (as a side note, the multipartite methods could be seen as well as first order polynomial approximations). As the input width increases, the methods require either an increasing number of terms, as well as a rapidly increasing amount of storage for the coefficients.

The method presented in [14] tries to overcome these obstacles with an approach consisting of three steps. The first step consists of a *range reduction* of the input. The next step is the *evaluation* of the function on the reduced argument using a series expansion of the target function. Last is a *post-processing* phase, which is required due to the first step.

The main premise of the method is a classical one: the evaluation of the function on the reduced argument is less expensive than on the original input. This is due to the higher order terms in the evaluation of the series being shifted out to lower weights than the target precision, thus they do not need to be computed.

The range reduction transforms the input $1 \leq x < 2$ into $-2^{-k} < x' < 2^{-k}$. x' is obtained as $x' = x \cdot \hat{x} - 1$, where $\hat{x}$ is a $k+1$ bit approximation of $\frac{1}{x}$. $\hat{x}$ can be obtained from a table addressed by the k MSBs of x.

The evaluation step consists of approximating the target function (f) at x'. f can be expressed as:

$$f(x') = C_0 + C_1 \cdot x' + C_2 \cdot (x')^2 + C_3 \cdot (x')^3 + \ldots$$

where the coefficients C_i are given by the Taylor series development. The input x' is split into 4 chunks of size k, x'_1 to x'_4; as $|x'| < 2^{-k}$, x'_1 can be left out, as it only contains sign bits. Replacing x' in the previous equation and eliminating terms of weight of less than 2^{-4k} results in:

$$\begin{aligned} f(x') &\approx C_0 + C_1 \cdot x' + C_2 \cdot x'^2_1 \cdot 2^{-4k} + 2C_2 \cdot x'_2 \cdot x'_3 \cdot 2^{-5k} \\ &\quad + C_3 \cdot (x'_2)^3 2^{-6k} \end{aligned}$$

The post processing consists of multiplying the result obtained at the previous step with a term that can be obtained similarly as $\hat{x}$. This term is $\hat{x}$, $\frac{1}{\sqrt{\hat{x}}}$ and $\sqrt{\hat{x}}$ for the reciprocal, sqrt and reciprocal sqrt.

One of the advantages of this method would be the small multipliers and tables required for implementation, which seem to map quite well to the hardware resources available on the current generation FPGAs. Another advantage would be the simple range reduction and post-processing phase.

It would be interesting to see, however, to what degree the method could be improved by replacing the approximation of $f(x')$ by the Taylor series development with a more accurate polynomial approximation.

B. Newton-Raphson Method

Probably the best known iterative method is the Newton-Raphson scheme. This is a root-finding scheme and has quadratic convergence. In order to deduce the method, the

starting point is again the Taylor series, around the point x_n, limited to the first two terms:

$$f(x) \ = f(x_n) + f'(x_n) \cdot (x - x_n)$$

A root of $f(x)$ satisfies $f(x) = 0$, so:

$$f(x_n) + f'(x_n) \cdot (x - x_n) \ \approx \ 0$$

which gives:

$$x \ \approx \ x_n - \frac{f(x_n)}{f'(x_n)}$$

Thus we can obtain the recurrence relation: $x_{n+1} \ = \ x_n - \frac{f(x_n)}{f'(x_n)}$. The error entailed by this iteration can be approximated by a recurrence relation as well:

$$\varepsilon_{n+1} \ \approx \ \varepsilon_n^2 \cdot \frac{f''(x_n)}{2 \cdot f'(x_n)}$$

which shows that the method converges quadratically.

Again, for the sake of brevity, the following discussions will be limited to the case of the reciprocal. In order to compute the reciprocal of a number a, a function f is needed so that $f(\frac{1}{a}) = 0$. A convenient choice for the function f is $f(x) = \frac{1}{x} - a$. Replacing f into the recurrence relation results in the following scheme:

$$x_{n+1} \ = \ x_n \cdot (2 - a \cdot x_n) \tag{12}$$

In terms of the implementation, the iteration given by eq. (12) only requires additions and multiplications, once an initial approximation has been obtained.

We perform an error analysis similar to those of Section VI-B. The total error is again divided as $\varepsilon_{total} = \varepsilon_{final\_round} + \varepsilon_{round} + \varepsilon_{method}$, and $\varepsilon_{total} < 2^{-w}$. Eq. (12) thus becomes:

$$x_{n+1} \ = \ \widetilde{x_n} \cdot (2 - \widetilde{a \cdot \widetilde{x_n}}) \tag{13}$$

This assumes that we have an initial approximation correctly rounded to $m = \lceil \frac{w}{2} \rceil$ bits (which implies an error less than 2^{-m-1}). Writing explicitly the errors in eq. (13), it becomes:

$$x_{n+1} \ = \ (x_n + 2^{-m-1}) \cdot (2 - a \cdot (x_n + 2^{-m-1}) + \varepsilon_{mult1}) + \varepsilon_{mult2}$$

which can be written as:

$$x_{n+1} \ = \ x_n \cdot (2 - a \cdot x_n) + \varepsilon_{round}$$

where ε_{round} represents the rounding errors

$$\begin{aligned} \varepsilon_{round} \ = \ & 2^{-m-1} \cdot (2 - a \cdot x_n) - 2^{-m-1} \cdot a \cdot (x_n + 2^{-m-1}) \\ & + \varepsilon_{mult1} \cdot (x_n + 2^{-m-1}) + \varepsilon_{mult2} \end{aligned} \tag{14}$$

The sum of the first two terms of eq. (14) is of the order of magnitude of 2^{-2m-2}. If ε_{mult1} and ε_{mult2} are of the form 2^{-w-g} (where g gives the number of extra guard bits), eq. (14) shows that at least 3 guard bits are needed.

A good option for the initial approximation is the use of a bipartite/multipartite table. This is also the approach used in this article. For target precisions of up to approx. 32 bits $\lceil \frac{w}{2} \rceil$ falls in the range of values that are suitable for use with the bipartite method. Increasing the number of iterations is also possible, thus increasing the number of multiplications, but decreasing of resources needed to obtain the initial approximation.

C. Generalized Iterative Methods

The Newton-Raphson method discussed in Section VII-B has quadratic convergence. However, there are methods with faster convergence. The second order Newton-Raphson method is also known as the Halley method. In order to deduce the method, we will have the same approach as in the previous section, starting with the Taylor series around the point x_n, but using the first three terms:

$$f(x) \ = f(x_n) + f'(x_n) \cdot (x - x_n) + \frac{1}{2} f''(x_n) \cdot (x - x_n)^2$$

A root of $f(x)$ satisfies $f(x) = 0$, so:

$$f(x_n) + f'(x_n) \cdot (x - x_n) + \frac{1}{2} f''(x_n) \cdot (x - x_n)^2 \ \approx \ 0$$

which can be re-grouped as:

$$x_{n+1} \ = \ x_n - \frac{f(x_n)}{f'(x_n) + \frac{f''(x_n) \cdot (x_{n+1} - x_n)}{2}}$$

Knowing that $x_{n+1} - x_n \ = \ - \frac{f(x_n)}{f'(x_n)}$ results in the iteration formula:

$$x_{n+1} \ = \ x_n - \frac{2 \cdot f(x_n) \cdot f'(x_n)}{2 \cdot (f'(x_n))^2 - f(x_n) \cdot f''(x_n)} \tag{15}$$

Equation (15) is commonly known as the Halley iteration/method. The error entailed by the method can be expressed a recurrence as well, as in the case of the Newton-Raphson iteration:

$$\varepsilon_{n+1} \ \approx \ \varepsilon_n^3 \cdot \frac{f'''(x_n)}{6 \cdot f'(x_n)}$$

which shows the cubic convergence of the method.

The Newton-Raphson and the Halley method are the first and the second in a class of iterative methods known as the Householder methods [15]. The $n+1$-st term in the class has the form:

$$x_{n+1} \ = \ x_n + (n+1) \cdot \frac{\left(\frac{1}{f(x_n)}\right)^{(n)}}{\left(\frac{1}{f(x_n)}\right)^{(n+1)}} \tag{16}$$

where the (n) superscript represents the n-th order derivative. It can be easily verified that choosing $n = 0$ results in the Newton-Raphson iteration, while choosing $n = 1$ results in the Halley iteration. In terms of the error, the Householder method progresses at a pace of order $n+1$ towards the correct solution.

The Halley method (or the Householder method, more generally) replaces the tangent to the function plot by a curve (or a higher order curve, in the case of Householder). This curve has a higher number of derivatives in common with the function plot, at the point of the approximation. This should, in principle, better fit the plot, giving, thus, a better approximation. However, as remarked in [17], in the case of the reciprocal this expansion is not particularly useful. Plugging-in the same function as in the case of the Newton-Raphson iteration in the Halley iteration requires the

computation of the inverse that we are trying to approximate in the first place.

On the other hand, as remarked in [18] and even further back presented in [19] taking a different approach to obtaining the iteration in the first place seems to be more effective.

The starting point for the alternative approach is again the Taylor series around the point x_n:

$$\begin{aligned} f(x) &= f(x_n) + \frac{f'(x_n)}{1!} \cdot (x - x_n) + \frac{f''(x_n)}{2!} \cdot (x - x_n)^2 \\ &+ \frac{f'''(x_n)}{3!} \cdot (x - x_n)^3 + \dots \end{aligned} \tag{17}$$

Again, a root of $f(x)$ satisfies $f(x) = 0$. The value of $x - x_n$ is expressed as a power series of $f(x_n)$:

$$(x - x_n) = a \cdot f(x_n) + b \cdot (f(x_n))^2 + c \cdot (f(x_n))^3 + \dots \tag{18}$$

By replacing eq. (18) in (17) and using the fact that $f(x) = 0$ results in:

$$\begin{aligned} 0 &= f(x_n) \\ &+ \frac{f'(x_n)}{1!} \cdot (a \cdot f(x_n) + b \cdot (f(x_n))^2 + c \cdot (f(x_n))^3 + \dots) \\ &+ \frac{f''(x_n)}{2!} \cdot (a \cdot f(x_n) + b \cdot (f(x_n))^2 + c \cdot (f(x_n))^3 + \dots)^2 \\ &+ \dots \end{aligned} \tag{19}$$

If eq. (19) is seen as an equation with $f(x_n)$ as a variable, the coefficients of the same powers of $f(x_n)$ on the two sides of the equation can be identified. Thus, the a, b, c, ... coefficients can be found:

$$\begin{aligned} a &= -\frac{1}{f'(x_n)} \\ b &= -\frac{f''(x_n)}{2 \cdot (f'(x_n))^3} \\ c &= -\frac{(f''(x_n))^2}{2 \cdot (f'(x_n))^5} \\ &\dots \end{aligned} \tag{20}$$

Replacing the values of eq. (20) in eq. (18) results in an iteration of the form (showing only the first three terms):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{(f(x_n))^2 \cdot f''(x_n)}{2(f'(x_n))^3} - \dots \tag{21}$$

Eq. (21) can be used as an alternative to Halley's method.

In order to obtain the cubic iteration for the reciprocal function, $f(x)$ in eq. (21) is chosen as $f(x) = \frac{1}{x} - a$. This results in:

$$\begin{aligned} x_{n+1} &= x_n \cdot (1 + h_n(1 + h_n)) \\ h_n &= 1 - a \cdot x_n \end{aligned} \tag{22}$$

In order to obtain the iteration for the reciprocal sqrt, $f(x)$ in eq. (21) is chosen as $f(x) = \frac{1}{x^2} - a$. This results in:

$$\begin{aligned} x_{n+1} &= \frac{1}{8} \cdot x_n \cdot (8 + h_n(4 + 3 \cdot h_n)) \\ h_n &= 1 - a \cdot x_n^2 \end{aligned}$$

Both the iteration for the reciprocal and the one for reciprocal sqrt have cubic convergence.

The rest of this section is dedicated to the error analysis for the datapath implementing eq. (22). The evaluation, on the other hand, is done as (where h_n keeps the meaning of eq. (22)):

$$x_{n+1} = x_n + x_n \cdot (h_n + h_n^2) \tag{23}$$

This form of the iteration takes advantage of the fact that $h_n < 2^{-m}$ (the proof and reasoning are similar to the range reduction of [14]). This means that less bits are needed for the squaring of h_n and for the multiplication $x_n \cdot (h_n + h_n^2)$, due to the terms being shifted to the right by $2m$ and m bits respectively.

As for the case of the previous error analyses, the total error is divided in three parts, $\varepsilon_{total} = \varepsilon_{final\_round} + \varepsilon_{round} + \varepsilon_{method} < 2^{-w}$. The final result is rounded, so $\varepsilon_{final\_round} < 2^{-w-1}$. The method error, ε_{method}, was previously discussed. What remains to be determined are the rounding errors, ε_{round}. These are due to the initial approximation and to truncating the multiplications in eq. (23). Making these errors explicit, eq. (23) becomes (where m is the precision of the x_n approximation):

$$\begin{aligned} x_{n+1} &= (x_n + 2^{-m-1}) + (x_n + 2^{-m-1}) \cdot (h_n - a \cdot 2^{-m-1} - \varepsilon_{mult} \\ &+ (h_n - a \cdot 2^{-m-1} - \varepsilon_{mult})^2) \end{aligned}$$

or

$$x_{n+1} = x_n + x_n \cdot (h_n + h_n^2) + \varepsilon_{round}$$

where

$$\begin{aligned} \varepsilon_{round} &= 2^{-m-1} \cdot (1 + h_n + h_n^2) + \\ &+ (x_n + 2^{-m-1}) \cdot (a \cdot 2^{-m-1} + \varepsilon_{mult}) \cdot \\ &\cdot (a \cdot 2^{-m-1} + \varepsilon_{mult} - 1 - 2 \cdot h_n) \end{aligned}$$

which can be re-written as:

$$\begin{aligned} \varepsilon_{round} &= 2^{-m-1} \cdot (1 + h_n + h_n^2) + \\ &+ (x_n + 2^{-m-1}) \cdot [((-a)2^{-m-1} + a^2 2^{-2m-2} - a2^{-m} h_n) + \\ &+ \varepsilon_{mult}(1 - \varepsilon_{mult} - 2h_n + a2^{-m})] \end{aligned} \tag{24}$$

where ε_{mult} denotes the rounding error due to a multiplication or of a squaring.

The terms of equation (24) that do not contain ε_{mult} will result in a term that is of the order of 2^{-3m-3}. Thus, if ε_{mult} is of the form 2^{-w-g}, where g is the number of guard bits, we should ensure that $g > 3$ for a faithfully rounded result. This has also been confirmed through testing.

VIII. Results

In this section we present the resource utilization and performance of cores generated using the previously described methods.

In Table II we present the resources and performance of the extended low order Taylor method that we introduced in Section V for varying input formats. The table is split

into two sides: the left side presents the performance results for the default Taylor series coefficients whereas the right side presents the results for enhanced coefficients, found starting from the previous coefficients. The search methods applied allows exploring the design space and returning the architecture which approaches the functions best in less than 1 second, on recent compute architecture (Core i7).

There are several architectural variations of this solution:

- specialized underflow architecture; this architecture will generally trigger when the formats are of such sort that a significant number of inputs will underflow. There are 2 variations of this architecture: as depicted in Figure 3. An example which triggers the left architecture is the (16,4) format, and which triggers the right is the (16,5) format.
- architecture for which the approximation is sufficiently accurate on the compute range (except overflow). An example of this architecture is given by the (16,15) format.
- architecture for which an extra table is used for inputs for which the 1st degree approximation is not sufficiently accurate. The architecture of the extra table is selected in order to minimize memory block utilization. This table may also capture the inputs which cause the output to overflow, if doing so does not increase the memory block requirements. For the table itself there are two sub-architectures possible, depicted in Figure 2. For instance, format (16,12) will save 1 memory block by using the base+offset architecture, and will have the table indexed by x. Format (16,14) will also save 1 memory block but the table will be indexed by an offset version of x; the range close to 0 will be handled separately as handling it in the table would increase the memory requirements.

Among the presented formats, some show better savings than others in therms of the optimizations effect. For the ones which show little-to-no savings, we observed this was due to the size of memories; we were unable to reduce the number of data elements sufficiently (usually below a power of 2) to trigger a smaller memory block utilization.

The final row in Table II shows the resource requirements and performance for a bipartite implementation which uses an input shifter and leading-zero-counter to normalize the input, and a right shifter to denormalize the result. This type of implementation is regarded as classical when dealing with fixed-point implementations on a generic range. This implementation is generally agnostic to the input/output format, so we only give results for one such format. When comparing this method with the previously optimized method for the whole range we observe that the latency of this implementation is significantly larger. Next, since based on the bipartite implementation, the architecture does not consume any DSP blocks but it consumes more logic (ALMs) than all the presented architectures. In terms of memory blocks, the architecture performs significantly better for inputs ranging from (16,10) to (16,14) but is on par, or consumes more memories for the rest of the formats.

These results show that in the context of a core generator, the user requirements will determine which architecture will be returned. If the objective is latency or ALMs alone then the method for the entire range will be better; if targeting memory blocks, then the tie-break will be played on formats.

In Table III we present the resource requirements together with the performance for cores having inputs in the interval $[1,2)$, and precision varying from 10 bits up to 32 bits. The varying precisions will trigger the different architectures: table-based, bipartite, Taylor-based, and high-order based on Newton-Raphson, Halley and Taylor. As expected, bipartite methods perform well up to 16-bit precisions. The method *Tabulate and Multiply* presented in Section VI-A performs well up to 24 bits. With higher precision one can observe the increase in memory blocks for this architecture. Finally, for 32-bits we compare the requirements and performance of a Newton-Raphson-based implementation bootstrapped by a bipartite approximation, a higher-order Taylor based implementation, and two variations of Halley's method: first bootstrapped by a table, and the second bootstrapped by a bipartite approximation. These implementations show interesting trade-offs: Newton-Raphson+ bipartite shows the highest number of memory blocks (5) but the lowest number of DSPs (2), an average-low number of ALMs, and a short latency; Taylor uses the most ALMs, but fewest memory blocks, highest number of DSPs and highest latency; Halley bootstrapped by a table has the lowest ALM and latency but highest DSP and memory; Halley bootstrapped by bipartite reduces the number of memory blocks to minimum, but slightly increases the number of ALMs.

Table III misses the implementation of the second-order *Tabulate and Multiply* method presented in section VI-B which is expected to lower memory requirements around the 24-bit precisions where the current first order implementation uses 5 memory blocks. The results for this method will be present in the final version of the paper.

Table IV presents the resource utilization and performance of small precision floating-point cores. The utilization numbers are presented in order to contrast these implementations against the fixed-point counterparts. The floating-point implementations are based on a piecewise-polynomial approximation with half-precision using a degree 1 polynomial, whereas remaining precisions presented in the table using a degree 2 polynomial. Implementations use the Horner scheme and the coefficient sizes follow closely [20]. Horner scheme also uses truncated multipliers, however the low degree polynomials and larger monolithic multipliers in the target FPGA device diminish the benefits. Please note that the kernels used in these implementations directly use normalized inputs, so are comparable to Table III.

IX. Conclusion

In this article we have revisited some well known methods for implementation of fixed-point kernels, and depicted their performance on a contemporary FPGA device: an Altera

TABLE II

RESOURCE UTILIZATION AND ESTIMATED PERFORMANCE OF OUR FIRST DEGREE TAYLOR BASED METHOD WORKING ON THE FULL INPUT RANGE, FOR 16-BIT PRECISION INPUTS, VARYING INPUT/OUTPUT FORMATS FROM 15 FRACTIONAL BITS DOWN TO 4 FRACTIONAL BITS. RESULTS ARE GIVEN FOR A CYCLONEV C6 DEVICE. THE FINAL ROW SHOWS THE UTILISATION AND PERFORMANCE OF A GENERIC IMPLEMENTATION OF THE OPERATOR BASED ON THE RANGE-REDUCTION TO THE INTERVAL [1,2). THIS IMPLEMENTATION IS AGNOSTIC TO THE INPUT FORMAT.

Input/Output Format	Resource utilization and Performance Default					Resource utilization and Performance Optimized				
(width, fraction)	ALMs	DSPs	M10Ks	Latency	Frequency	ALMs	DSPs	M10Ks	Latency	Frequency
16,15	61	1	3	7	310MHz	54	1	3	7	310MHz
16,14	101	1	12	9	242MHz	96	1	8	9	235MHz
16,13	107	1	15	9	236MHz	101	1	10	9	240MHz
16,12	103	1	11	9	233MHz	105	1	11	9	232MHz
16,11	88	1	12	7	240MHz	86	1	8	6	235MHz
16,10	95	1	8	7	241MHz	95	1	8	7	236MHz
16, 9	85	1	6	7	228MHz	86	1	6	7	235MHz
16,8	61	1	5	6	241MHz	61	1	5	6	241MHz
16,7	59	1	5	6	237MHz	61	1	5	6	231MHz
16,6	52	1	4	6	303MHz	52	1	4	7	303MHz
16,5	58	1	2	6	308MHz	58	1	2	7	308MHz
16,4	28	0	1	2	315MHz	28	0	1	2	315MHz
16,8 (GRR)	126	0	5	12	240MHz	← Generic Range Reduction				

TABLE III

RESOURCE UTILIZATION AND ESTIMATED PERFORMANCE ARCHITECTURES IMPLEMENTED USING THE METHODS PRESENTED IN THIS ARTICLE. THE TARGET DEVICE IS CYCLONEV C8 SPEEDGRADE. INPUT IS UNSIGNED, AND IN THE INTERVAL [1,2)

Input/Output Format	Implementation	Resource utilization and Performance				
(width, fraction)		ALMs	DSPs	M10Ks	Latency	Frequency
11,10	Tabulation	1	0	3	2	315MHz
16,15	Bipartite	27	0	3	4	315MHz
18,17	Tabulate and Multiply	54	1	1	5	200MHz
20,19	Tabulate and Multiply	48	1	1	6	310MHz
23,22	Tabulate and Multiply	58	1	3	6	310MHz
24,23	Tabulate and Multiply	64	1	5	6	310MHz
32,31	Bipartite+ Newton-Raphson	188	2	5	15	285MHz
32,31	Taylor	282	3	1	23	267MHz
32,31	Halley	155	3	3	15	235MHz
32,31	Bipartite+ Halley	159	3	1	15	228MHz

TABLE IV

FLOATING-POINT IMPLEMENTATIONS OF $1/x$. THE FLOATING-POINT FORMATS DENOTE (EXPONENT WIDTH, FRACTION WIDTH). IMPLEMENTATIONS FLUSH SUBNORMALS TO ZERO. IMPLEMENTATIONS TARGET FAITHFUL ROUNDING

f	**Type**	**I/O**	**Performance**
$1/x$	FP	(5,10)	58ALMs, 1DSP, 0M20K, 3clk, 600MHz
	FP	(8,17)	116ALMs, 2DSP, 0M20K, 5clk, 441MHz
	FP	(8,23)	126ALMs, 2DSPs, 3M20K, 9clk, 450MHz
	FP	(8,26)	152ALMs, 2DSP, 3M20K, 10clk, 424MHz

CycloneV. Unlike previous works which generally focus implementations of normalized inputs ([1,2) or [0.5,1)), we have focused in this work on the the full implementation range. Where possible we have modified the classical architecture to directly handle the full input range; the results of this work were competitive architectures which trade off logic resources for memory blocks. Our results also show that the behaviour and performance of the modified full-range architectures are highly dependent on the input and output format. Therefore, our architectures are more likely to be regarded as implementation tradeoffs. We have shown our results on the reciprocal function, but we have used the same templates for the reciprocal square root and square root functions. We have also revisited higher order methods based on the Newton-Raphson algorithm, Taylor series, and Halley's method, and proposed combined architectures; there, Newton-Raphson and Halley are bootstrapped using a bipartite-based implementation. The results shown in this paper confirm that fixed-point arithmetic is very relevant for contemporary FPGAs containing up to thousands of embedded memory blocks and multipliers, and the low resources of these cores in combination with the low resource of primitive components will keep fixed-point datapaths relevant in resource critical scenarios, and when expertise exists to bound variable ranges.

ACKNOWLEDGEMENTS

This work was supported by the Altera European Technology Centre (that Matei Iştoan was part of at the moment when the works described in this paper started) and the French National Research Agency through the INS program *MetaLibm*.

REFERENCES

[1] M. Langhammer and B. Pasca, "Floating-Point DSP Block Architecture for FPGAs," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 117–125.

[2] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," in *International Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 1995, p. 155.

[3] *Cyclone V Device Handbook*, 2012, http://www.altera.com/literature/hb/cyclone-v/cyclone5_handbook.pdf.

[4] W. Wong and E. Goto, "Fast evaluation of the elementary functions in single precision," *Computers, IEEE Transactions on*, vol. 44, no. 3, pp. 453–457, Mar 1995.

[5] J. Low and C. C. Jong, "A memory-efficient tables-and-additions method for accurate computation of elementary functions," *Computers, IEEE Transactions on*, vol. 62, no. 5, pp. 858–872, May 2013.

[6] D. Das Sarma and D. Matula, "Faithful bipartite rom reciprocal tables," in *Computer Arithmetic, 1995., Proceedings of the 12th Symposium on*, Jul 1995, pp. 17–28.

[7] M. Schulte and J. Stine, "Symmetric bipartite tables for accurate function approximation," in *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, Jul 1997, pp. 175–183.

[8] J. Stine and M. Schulte, "The symmetric table addition method for accurate function approximation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 21, no. 2, pp. 167–177, 1999.

[9] J.-M. Muller, "A few results on table-based methods," *Reliable Computing*, vol. 5, no. 3, pp. 279–288, 1999.

[10] F. de Dinechin and A. Tisserand, "Some improvements on multipartite table methods," in *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, 2001, pp. 128–135.

[11] S.-F. Hsiao, P.-H. Wu, C.-S. Wen, and P. Meher, "Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 62, no. 5, pp. 466–470, May 2015.

[12] N. Takagi, "Generating a power of an operand by a table look-up and a multiplication," in *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, Jul 1997, pp. 126–131.

[13] M. Ito, N. Takagi, and S. Yajima, "Efficient initial approximation for multiplicative division and square root by a multiplication with operand modification," *Computers, IEEE Transactions on*, vol. 46, no. 4, pp. 495–498, Apr 1997.

[14] M. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand, "Reciprocation, square root, inverse square root, and some elementary functions using small multipliers," *Computers, IEEE Transactions on*, vol. 49, no. 7, pp. 628–637, Jul 2000.

[15] J. M. Borwein and P. B. Borwein, *Pi and the AGM: A Study in the Analytic Number Theory and Computational Complexity*. New York, NY, USA: Wiley-Interscience, 1987.

[16] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *Mathematical Software - ICMS 2010*, ser. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327. Heidelberg, Germany: Springer, September 2010, pp. 28–31.

[17] M. Flynn, "On division by functional iteration," *Computers, IEEE Transactions on*, vol. C-19, no. 8, pp. 702–706, Aug 1970.

[18] P. Rabinowitz, "Multiple-precision division," *Commun. ACM*, vol. 4, no. 2, pp. 98–, Feb. 1961.

[19] F. Willers and R. Beyer, *Practical analysis: graphical and numerical methods*, ser. Dover Books on Science. Dover Publications, 1948.

[20] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Rennes, Jul. 2010.